Egzamin Licencjacki TCS

Notatki

„To jest niesamowite, przez TCS nie jestem w stanie odróżnić sytuacji, w których się stresuję od tych, w których się nie stresuję.”

„Ani przez dzień nie chciałem umrzeć na tych studiach. Nie jest to prawda. Ale gdyby tak było, to byłaby to ich zaleta.”

Maja Giglok
Krzysztof Kosiorowski
Tomasz Kulis
Magdalena Kycler
Jacek Markiewicz
Maciej Mikołajczak
Jakub Wieliczko

redakcja: Maciej Mikołajczak

Kraków, 2026

Spis treści

I Przedmioty matematyczne 4

I.1 Analiza Matematyczna 5
I.1.1 Szeregi liczbowe 5
I.1.2 Ciągłość 6
I.1.3 Pochodna 7
I.1.4 Wzór Taylora 7
I.1.5 Całka Riemanna 8
I.1.6 Pochodne cząstkowe 8
I.1.7 Ekstrema 9
I.1.8 Całka wielu zmiennych 10
I.2 Matematyka Dyskretna 11
I.2.1 Współczynniki dwumianowe 12
I.2.2 Liczby Catalana 12
I.2.3 Twierdzenie Spernera 13
I.2.4 Funkcje tworzące 14
I.2.5 Twierdzenie Ramseya 15
I.2.6 Twierdzenie Brooksa 15
I.2.7 Liczba chromatyczna a liczba klikowa 16
I.2.8 Liczba chromatyczna a liczba kolorująca 18
I.2.9 Przepływy 18
I.3 Metody Algebraiczne Informatyki 20
I.3.1 Grupy 21
I.3.2 Pierścienie i ciała 22
I.3.3 Ciała skończone 24
I.3.4 Liczby zespolone 25
I.3.5 Wyznacznik macierzy 27
I.3.6 Eliminacja Gaussa 28
I.3.7 Przestrzenie wektorowe 29
I.3.8 Odwzorowania liniowe i wieloliniowe 31
I.3.9 Wartości własne 32
I.3.10 Diagonalizacja 33
I.3.11 Przestrzenie euklidesowe i unitarne 35
I.4 Metody Formalne Informatyki 37
I.4.1 Własności liczb naturalnych 38
I.4.2 Liczby rzeczywiste 40
I.4.3 Relacje, iloczyny kartezjańskie 41
I.4.4 Konstrukcja liczb naturalnych 42
I.4.5 Relacje równoważności 45
I.4.6 Twierdzenie Cantora-Bernsteina 46
I.4.7 Lemat Kuratowskiego-Zorna 46
I.4.8 Liczby całkowite i wymierne 48
I.4.9 Liczność zbiorów 48
I.4.10 Twierdzenie Zermelo 49
I.4.11 Liczby porządkowe 50
I.5 Metody Probabilistyczne Informatyki 51
I.5.1 Łańcuchy Markowa 52
I.5.2 Kule i urny 53
I.5.3 Quicksort 55
I.5.4 Igła Buffona 55
I.5.5 Spacer losowy 56
I.5.6 Proces Poissona 57
I.5.7 Centralne Twierdzenie Graniczne 58
I.6 Modele Obliczeń 60
I.6.1 Hierarchia Chomsky'ego 61
I.6.2 Języki bezkontekstowe 62

I.6.3 Języki regularne a automaty skończone . . . 65
I.6.4 Operacje na językach regularnych i bezkontekstowych . . . 68
I.6.5 Twierdzenie Myhilla-Nerode'a . . . 69
I.6.6 Maszyny Turinga . . . 70
I.6.7 Problem stopu . . . 73
I.6.8 Klasy złożoności . . . 75

II Przedmioty programistyczno-algorytmiczne 79
II.1 Analiza Algorytmów . . . 80
II.1.1 Twierdzenie o rekurencji uniwersalnej . . . 80
II.1.2 Problem sekretarki . . . 81
II.1.3 Drzewa rozchylane . . . 82
II.1.4 Złożoność sortowania . . . 84
II.1.5 Randomizowane BST . . . 85
II.1.6 Haszowanie doskonałe . . . 87
II.1.7 Problem DSU . . . 89
II.2 Algorytmy i Struktury Danych 1 . . . 91
II.2.1 Kody Huffmanna . . . 92
II.2.2 Najdłuższy wspólny podciąg . . . 93
II.2.3 Drzewa AVL . . . 93
II.2.4 Najkrótsze ścieżki . . . 95
II.2.5 Minimalne drzewa rozpinające . . . 96
II.2.6 Przepływy . . . 97
II.2.7 Skojarzenia . . . 98
II.3 Algorytmy i Struktury Danych 2 . . . 99
II.3.1 Wyszukiwanie wzorca . . . 100
II.3.2 Tablica sufiksowa . . . 101
II.3.3 Zamiatanie w geometrii . . . 103
II.3.4 Otoczka wypukła . . . 104
II.3.5 Testowanie pierwszości . . . 105
II.3.6 Programowanie liniowe . . . 106
II.3.7 Algorytmy aproksymacyjne . . . 108
II.4 Metody Programowania . . . 109
II.4.1 Algorytm Karatsuby, metoda Strassena . . . 110
II.4.2 Mergesort i quicksort . . . 111
II.4.3 Sortowanie topologiczne, silnie spójne składowe . . . 112
II.4.4 Statystyki pozycyjne . . . 113
II.5 Programowanie Obiektowe . . . 114
II.5.1 Techniki obiektowe . . . 115
II.5.2 Szablony i typy generyczne . . . 118
II.5.3 Pola prywatne i chronione . . . 119
II.5.4 RTTI i refleksja . . . 120
II.5.5 Biblioteki GUI . . . 123
II.5.6 Konstrukcja i destrukcja obiektów . . . 124
II.5.7 Standardowe biblioteki kontenerów . . . 126

III Przedmioty techniczne 128
III.1 Inżynieria Danych . . . 129
III.1.1 Normalizacja . . . 129
III.1.2 Trwałość danych . . . 131
III.1.3 Optymalizacja zapytań . . . 133
III.1.4 Indeksy . . . 134
III.1.5 Klucze w bazach relacyjnych . . . 135
III.1.6 Transakcje . . . 136
III.1.7 Związki encji . . . 139
III.2 Inżynieria Oprogramowania . . . 141
III.2.1 Test Driven Development . . . 142
III.2.2 Dependency injection . . . 143

III.2.3 Wzorce projektowe . . . 144
III.2.4 SOLID . . . 145
III.3 Podstawy Programowania . . . 148
III.3.1 Reprezentacja liczb . . . 149
III.4 Programowanie Niskopoziomowe . . . 152
III.4.1 Urządzenia zewnętrzne . . . 153
III.4.2 Odczyt z dysku . . . 155
III.4.3 Instrukcje wektorowe . . . 156
III.4.4 Zarządzanie i ochrona pamięci . . . 158
III.4.5 Hierarchia pamięci . . . 159
III.4.6 Architektura x86 . . . 160
III.5 Sieci Komputerowe . . . 162
III.5.1 Warstwy sieci . . . 163
III.5.2 Błędy transmisji . . . 164
III.5.3 Protokół IP . . . 165
III.5.4 Protokół TCP . . . 167
III.5.5 Protokół HTTP . . . 168
III.5.6 TLS i SSL . . . 170
III.6 Systemy Operacyjne . . . 172
III.6.1 Komunikacja międzyprocesowa . . . 173
III.6.2 Szeregowanie procesów . . . 175
III.6.3 Deadlock . . . 177
III.6.4 Spooling . . . 178
III.6.5 Segmentacja i stronicowanie . . . 179
III.6.6 Mikrojądro a architektura monolityczna . . . 181
III.6.7 Współdzielenie bibliotek . . . 182
III.6.8 Problem ucztujących filozofów . . . 183

Posłowie **185**

I
Przedmioty matematyczne

I.1 Analiza Matematyczna

I.1.1 Szeregi liczbowe

Szeregi liczbowe. Podstawowe kryteria zbieżności.

Definicja. Ciągiem liczbowym nazywamy dowolną funkcję $a : \mathbb{N} \to \mathbb{R}$. Zazwyczaj przyjmujemy oznaczenie $a_n := a(n)$ i oznaczamy ciąg jako $(a_n)_{n=0}^{\infty}$. Mówimy, że ciąg $(a_n)_{n=0}^{\infty}$ jest zbieżny, jeśli istnieje takie $g \in \mathbb{R}$, że dla każdego $\varepsilon > 0$ istnieje takie $N \in \mathbb{N}$, że dla wszystkich $n \geqslant N$ mamy $|a_n - g| < \varepsilon$. Piszemy wtedy $\lim_{n\to\infty} a_n = g$ lub $a_n \to g$.

Definicja. Niech $(x_n)_{n=0}^{\infty}$ będzie ciągiem liczbowym. Szeregiem liczbowym nazywamy parę ciągów $((x_n)_{n=0}^{\infty}, (s_n)_{n=0}^{\infty})$, gdzie $s_n = x_0 + \ldots + x_n$. Ciąg $(x_n)_{n=0}^{\infty}$ nazywamy ciągiem wyrazów szeregu, a $(s_n)_{n=0}^{\infty}$ ciągiem sum częściowych szeregu. Zazwyczaj stosujemy zapis $\sum_{n=0}^{\infty} x_n$. Mówimy, że szereg jest zbieżny, jeśli jego ciąg sum częściowych jest zbieżny. Jego granicę g nazywamy wtedy sumą szeregu. Piszemy $\sum_{n=0}^{\infty} x_n = g$.

Definicja. Szereg $\sum_{n=0}^{\infty} x_n$ nazywamy bezwzględnie zbieżnym, jeśli szereg $\sum_{n=0}^{\infty} |x_n|$ jest zbieżny.

Twierdzenie (Warunek konieczny zbieżności)**.** Jeśli $\sum_{n=0}^{\infty} x_n$ jest zbieżny, to $x_n \to 0$.

Dowód. Oznaczmy $s_n = \sum_{i=0}^{n} x_i$. Z założenia $s_n \to s$ dla pewnego s. Wtedy dla każdego $\varepsilon > 0$ istnieje $N \in \mathbb{N}$ takie, że dla $n \geqslant N$ mamy $|s_n - s| < \frac{\varepsilon}{2}$. Wtedy $|x_{n+1}| = |s_{n+1} - s_n| \leqslant |s_{n+1} - s| + |s - s_n| < \varepsilon$. □

Warunek konieczny nie jest wystarczający. Na przykład $\sum_{n=1}^{\infty} \frac{1}{n}$ jest rozbieżny mimo, że $\frac{1}{n} \to 0$.

Twierdzenie (Kryterium porównawcze)**.** Niech $(x_n)_{n=0}^{\infty}, (y_n)_{n=0}^{\infty}$ będą takimi ciągami o wyrazach nieujemnych, że $x_n \leqslant y_n$ dla każdego $n \in \mathbb{N}$. Jeśli $\sum_{n=0}^{\infty} y_n$ jest zbieżny, to $\sum_{n=0}^{\infty} x_n$ też. Jeśli $\sum_{n=0}^{\infty} x_n$ jest rozbieżny, to $\sum_{n=0}^{\infty} y_n$ też.

Twierdzenie (Kryterium porównawcze graniczne)**.** Niech $(x_n)_{n=0}^{\infty}, (y_n)_{n=0}^{\infty}$ będą takimi ciągami o wyrazach dodatnich, że $\lim_{n\to\infty} \frac{x_n}{y_n} = g$ dla pewnego $g \in (0, +\infty)$. Wtedy szeregi $\sum_{n=0}^{\infty} x_n$ i $\sum_{n=0}^{\infty} y_n$ mają ten sam charakter zbieżności.

Twierdzenie (Kryterium Cauchy'ego)**.** Niech $(x_n)_{n=0}^{\infty}$ będzie ciągiem. Jeśli $\lim_{n\to\infty} \sqrt[n]{|x_n|} < 1$, to szereg $\sum_{n=0}^{\infty} x_n$ jest zbieżny. Jeśli $\lim_{n\to\infty} \sqrt[n]{|x_n|} > 1$, to ten szereg jest rozbieżny.

Twierdzenie (Kryterium d'Alemberta)**.** Niech $(x_n)_{n=0}^{\infty}$ będzie ciągiem. Jeśli $\lim_{n\to\infty} \frac{x_{n+1}}{x_n} < 1$, to szereg $\sum_{n=0}^{\infty} x_n$ jest zbieżny. Jeśli $\lim_{n\to\infty} \frac{x_{n+1}}{x_n} > 1$, to ten szereg jest rozbieżny.

W obu powyższych kryteriach możemy brać $\limsup$ (granicę górną) zamiast zwykłej granicy.

Twierdzenie (Kryterium Leibniza)**.** Niech $(x_n)_{n=0}^{\infty}$ będzie ciągiem takim, że $x_n \geqslant x_{n+1}$ dla każdego $n \in \mathbb{N}$ i $x_n \to 0$. Szereg $\sum_{n=0}^{\infty} (-1)^n x_n$ jest zbieżny.

Twierdzenie (Kryterium Dirichleta)**.** Niech $(x_n)_{n=0}^{\infty}$ będzie ciągiem takim, że $x_n \geqslant x_{n+1}$ dla każdego $n \in \mathbb{N}$ i $x_n \to 0$. Niech $(y_n)_{n=0}^{\infty}$ będzie takim ciągiem, że dla pewnego $M > 0$ i każdego $k \in \mathbb{N}$ zachodzi $\left|\sum_{n=0}^{k} y_n\right| \leqslant M$. Wtedy szereg $\sum_{n=0}^{\infty} x_n y_n$ jest zbieżny.

I.1.2 Ciągłość

Ciągłość funkcji w punkcie i na zbiorze. Warunki równoważne ciągłości. Własności funkcji ciągłych na zbiorach zwartych.

Definicja. Niech $U \subseteq \mathbb{R}$. Niech $f : U \to \mathbb{R}$ będzie funkcją, $c \in U$. Mówimy, że f jest ciągła w c, jeśli dla każdego $\varepsilon > 0$ istnieje $\delta > 0$ takie, że dla każdego $x \in U$ jeśli $|x - c| < \delta$, to $|f(x) - f(c)| < \varepsilon$. Inaczej mówiąc $\lim_{x \to c} f(x) = f(c)$. Tak postawioną definicję nazywamy definicją ciągłości według Cauchy'ego. Definicja ciągłości według Heinego mówi, że funkcja f jest ciągła w c, jeśli dla każdego $(x_n)_{n=0}^{\infty}$ takiego, że $x_n \to c$ zachodzi $f(x_n) \to f(c)$.

Definicja. Niech $f : U \to \mathbb{R}$ będzie funkcją, $A \subseteq U$. f jest ciągła na A, jeśli jest ciągła w każdym $x \in A$. f jest ciągła, jeśli jest ciągła na U.

Twierdzenie. Niech $f : U \to \mathbb{R}$ i $g : U \to \mathbb{R}$ będą funkcjami ciągłymi w $c \in U$. Wtedy:

1. $|f|$ jest ciągłe w c.
2. $f + g$, $f - g$, $f \cdot g$ jest ciągłe w c.
3. Jeśli $g(c) \neq 0$, to $\frac{f}{g}$ jest ciągłe w c.

Ciągłe są m.in.:

- wielomiany,
- funkcje wymierne (w punktach określenia),
- funkcje wykładnicze,
- funkcje logarytmiczne,
- funkcje trygonometryczne.

Składanie funkcji ciągłych daje funkcję ciągłą.

Twierdzenie (Własność Darboux)**.** Niech $f : U \to \mathbb{R}$ będzie funkcją ciągłą na $[a, b] \subseteq U$. Dla każdego $y \in [f(a), f(b)]$ istnieje takie $c \in [a, b]$, że $f(c) = y$. W szczególności jeśli $f(a) f(b) < 0$, to f ma pierwiastek w (a, b).

Definicja. Niech $U \subseteq \mathbb{R}$. Mówimy, że U jest:

- otwarty, jeśli dla każdego $x \in U$ istnieje takie $\varepsilon > 0$, że $(x - \varepsilon, x + \varepsilon) \subseteq U$.
- domknięty, jeśli $\mathbb{R} \setminus U$ jest otwarty.
- zwarty, jeśli dla każdej rodziny $\{U_i\}_{i \in I}$ zbiorów otwartych takiej, że $U \subseteq \bigcup_{i \in I} U_i$ istnieje podzbiór skończony $I_0 \subseteq I$ taki, że $U \subseteq \bigcup_{i \in I_0} U_i$.

Propozycja. Zbiór K jest domknięty wtedy i tylko wtedy, gdy dla każdego ciągu $(x_n)_{n=0}^{\infty} \subseteq K$ zbieżnego $x_n \to x$ zachodzi $x \in K$.

Twierdzenie (Heine, Borel)**.** Zbiór K jest zwarty wtedy i tylko wtedy, gdy jest domknięty i ograniczony (czyli istnieje takie $M > 0$, że $K \subseteq [-M, M]$).

Propozycja. Jeśli $K \subseteq \mathbb{R}$ jest zwarty a $f : U \to \mathbb{R}$ jest ciągła i $K \subseteq U$, to $f(K)$ jest zwarte.

Twierdzenie (Weierstrass)**.** Jeśli $f : A \to \mathbb{R}$ jest ciągła a A jest zwarte, to f osiąga swoje kresy, czyli istnieją takie $m, M \in A$, że $f(m) \leqslant f(x) \leqslant f(M)$ dla każdego $x \in A$.

I.1.3 Pochodna

Pochodna funkcji jednej zmiennej rzeczywistej (definicja i interpretacja geometryczna). Zastosowanie rachunku różniczkowego do badania przebiegu zmienności funkcji jednej zmiennej.

Definicja. Niech $f : A \to \mathbb{R}$ przy $A \subseteq \mathbb{R}$ będzie funkcją. f jest różniczkowalna w $x \in A$, jeśli $x \in \operatorname{int} A$ (czyli $(x - \varepsilon, x + \varepsilon) \subseteq A$ dla pewnego $\varepsilon > 0$) oraz istnieje granica $\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$. Oznaczamy tę granicę $f'(x)$ i nazywamy pochodną f w x. f jest różniczkowalna, jeśli jest różniczkowalna w każdym punkcie swojej dziedziny.

Możemy interpretować pochodną f w x jako nachylenie prostej stycznej do wykresu f w punkcie $(x, f(x))$. Ta prosta jest zadana równaniem $y = f(x_0) + f'(x_0)(x - x_0)$.

Twierdzenie. Jeżeli funkcja jest różniczkowalna w punkcie x_0, to jest w tym punkcie ciągła.

Twierdzenie (Lagrange)**.** Jeżeli funkcja f jest ciągła na $[a, b]$ i różniczkowalna na (a, b), to istnieje punkt $c \in (a, b)$ taki, że $f'(c) = \frac{f(b)-f(a)}{b-a}$.

Niech $B := (a, b) \subseteq A$ i niech $f : A \to \mathbb{R}$ będzie różniczkowalna we wszystkich punktach B. Jeśli $f'(x) \geqslant 0$ na B, to f jest niemalejąca na B. Jeśli $f'(x) \leqslant 0$ na B, to f jest nierosnąca na B. Jeśli $f'(x) = 0$ na B, to f jest stała na B.

Definicja. Niech $f : A \to \mathbb{R}$ będzie różniczkowalna. Można wtedy określić jej pochodną $f' : A \to \mathbb{R}$. Jeśli f' jest różniczkowalna w $x \in A$, to jej pochodną w x nazywamy drugą pochodną f w x i oznaczamy $f''(x)$. f jest dwukrotnie różniczkowalna, jeśli jej pochodna jest różniczkowalna.

Niech $B := (a, b) \subseteq A$ i niech $f : A \to \mathbb{R}$ będzie dwukrotnie różniczkowalna we wszystkich punktach B. Jeśli $f''(x) \geqslant 0$ na B, to f jest wypukła na B. Jeśli $f''(x) \leqslant 0$ na B, to f jest wklęsła na B. Jeśli $f'(x) = 0$, to x nazywamy punktem krytycznym f. Wtedy jeśli $f''(x) > 0$, to f ma w x minimum, a jeśli $f''(x) < 0$, to f ma w x maksimum.

I.1.4 Wzór Taylora

Wzór Taylora i jego przykładowe zastosowania.

Twierdzenie. Niech $f : A \to \mathbb{R}$ będzie funkcją. Niech $\Omega \subseteq A$ będzie otwarte, $a \in \Omega$. Niech f będzie $(n + 1)$-krotnie różniczkowalna w Ω. Ustalmy $h > 0$ takie, że $[a, a + h] \subseteq \Omega$. Dla pewnego $c \in [a, a + h]$ zachodzi

$$f(a + h) = \sum_{k=0}^{n} f^{(k)}(a) \frac{h^k}{k!} + f^{(n+1)}(c) \frac{h^{n+1}}{(n + 1)!}.$$

Wartość $f^{(n+1)}(c) \frac{h^{n+1}}{(n+1)!}$ nazywamy resztą Lagrange'a. Często stosuje się też wzór Taylora z resztą Peany, w którym resztą jest dowolne $R_n(x)$ takie, że $\lim_{x\to a} \frac{R_n(x)}{|x-a|^n} = 0$.

Wzór Taylora przydaje się do przybliżania funkcji, np. $e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \ldots$ jest zastosowaniem wzoru Taylora dla $a = 0$. Jeśli weźmiemy pierwsze k wyrazów tej sumy, to błąd będzie rzędu $o\left(|x|^k\right)$, co dla małych x jest bardzo dobre.

Jeśli funkcja f jest klasy $\mathcal{C}^\infty$ (ma wszystkie pochodne), to można zdefiniować szereg $\sum_{k=0}^{\infty} f^{(k)}(a) \frac{h^k}{k!}$ zwany szeregiem Taylora. Ten szereg może zbiegać do wartości $f(a + h)$, ale nie musi (może być rozbieżny lub zbiegać do czegoś innego). Jeśli istnieje takie $\varepsilon > 0$, że szereg Taylora jest zbieżny do $f(a + h)$ dla wszystkich $h \in (-\varepsilon, \varepsilon)$, to funkcję f nazywamy analityczną w otoczeniu a.

I.1.5 Całka Riemanna

Całka Riemanna. Podstawowe metody całkowania funkcji jednej zmiennej rzeczywistej. Całkowanie przez części. Całkowanie przez podstawienie.

Definicja. Podziałem $[a,b]$ nazywamy zbiór punktów $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$. Średnią podziału nazywamy wartość $\max_{i=0}^{n-1} x_{i+1} - x_i$.

Definicja. Niech $P = \{x_0 < \ldots < x_n\}$ będzie podziałem $[a,b]$ a $E = \{e_0, \ldots, e_{n-1}\}$ wyborem punktów pośrednich (czyli $e_i \in [x_i, x_{i+1}]$). Niech $f : [a,b] \to \mathbb{R}$ będzie funkcją. Sumą aproksymacyjną Riemanna nazywamy wartość $S(f; P, E) = \sum_{i=0}^{n-1} f(e_i)(x_{i+1} - x_i)$.

Definicja. Niech $[a,b] \subseteq U \subseteq \mathbb{R}$ i niech $f : U \to \mathbb{R}$ będzie funkcją ograniczoną. Niech $(P_n)_{n=0}^{\infty}$ będzie takim ciągiem podziałów, że ich średnie dążą do zera. Niech $(E_n)_{n=0}^{\infty}$ będzie ciągiem wyborów punktów pośrednich w odpowiednich podziałach. Załóżmy, że istnieje takie $L \in \mathbb{R}$, że $S(f; P_n, E_n) \to L$ i wartość L nie zależy od tego, jakie ciągi podziałów i punktów pośrednich zostały wybrane. Wtedy mówimy, że f jest całkowalna w sensie Riemanna na $[a,b]$ a wartość L oznaczamy $\int_a^b f \,\mathrm{d}x$ i nazywamy całką Riemanna funkcji f na $[a,b]$.

Twierdzenie. Niech $f : U \to \mathbb{R}$ będzie ciągła na $[a,b] \subseteq U$. Wtedy f jest całkowalna na $[a,b]$.

Propozycja. Całka jest:

1. Liniowa: $\int_a^b (\alpha f + \beta g) \,\mathrm{d}x = \alpha \int_a^b f(x) \,\mathrm{d}x + \beta \int_a^b g(x) \,\mathrm{d}x$.
2. Addytywna: $\int_a^b f(x) \,\mathrm{d}x = \int_a^c f(x) \,\mathrm{d}x + \int_c^b f(x) \,\mathrm{d}x$.
3. Monotoniczna: jeśli $f(x) \leqslant g(x)$, to $\int_a^b f(x) \,\mathrm{d}x \leqslant \int_a^b g(x) \,\mathrm{d}x$.

Zachodzi $\int_a^b f(x) \,\mathrm{d}x = -\int_b^a f(x) \,\mathrm{d}x$.

Twierdzenie. Niech $f : U \to \mathbb{R}$ będzie różniczkowalna w każdym punkcie $[a,b] \subseteq U$. Niech f' będzie całkowalna na $[a,b]$. Zachodzi $\int_a^b f'(x) \,\mathrm{d}x = f(b) - f(a)$.

Twierdzenie (Całkowanie przez części)**.** Niech $f : U \to \mathbb{R}$ i $g : V \to \mathbb{R}$ będą różniczkowalne w pewnym otoczeniu $[a,b] \subseteq U \cap V$ i takie, że f', g' są ciągłe na $[a,b]$. Wtedy

$$\int_a^b f'(x)\, g(x) \,\mathrm{d}x = f(x)\, g(x)\Big|_a^b - \int_a^b f(x)\, g'(x) \,\mathrm{d}x \,.$$

Twierdzenie (Całkowanie przez podstawienie)**.** Niech $F : U \to \mathbb{R}$ będzie różniczkowalne na $[a,b] \subseteq U$. Niech g będzie ciągłe na $F([a,b])$. Wtedy

$$\int_{F(a)}^{F(b)} g(x) \,\mathrm{d}x = \int_a^b g(F(x))\, F'(x) \,\mathrm{d}x \,.$$

I.1.6 Pochodne cząstkowe

Pochodne cząstkowe. Różniczkowalność funkcji wielu zmiennych. Zależność między istnieniem pochodnych cząstkowych a różniczkowalnością.

Definicja. Niech $U \subseteq \mathbb{R}^n$ i niech $f : U \to \mathbb{R}^m$ będzie funkcją. Ustalmy $i \in [n]$ i punkt $a = (a_1, \ldots, a_n) \in U$. Niech $U_i = \{x_i \in \mathbb{R} : (a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n) \in U\}$. Funkcję $f_i : U_i \ni x \to f(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n) \in \mathbb{R}^m$ nazywamy i-tym odwzorowaniem częściowym f w a. Jeśli f_i jest

różniczkowalne w a, to $f_i'(a)$ nazywamy i-tą pochodną cząstkową f w a i oznaczamy $\frac{\partial f}{\partial x_i}(a)$. To znaczy $\frac{\partial f}{\partial x_i}(a) = \lim_{t\to 0} \frac{f(a+te_i)-f(a)}{t}$ dla wektora jednostkowego $e_i \in \mathbb{R}^n$. Dla dowolnego wektora $u \in \mathbb{R}^n$ możemy zdefiniować pochodna kierunkową w kierunku u jako $\frac{\partial f}{\partial u}(a) = \lim_{t\to 0} \frac{f(a+tu)-f(a)}{t}$.

Definicja. Niech $A \subseteq \mathbb{R}^n$ i niech $f : A \to \mathbb{R}^m$ będzie funkcją. Mówimy, że f jest różniczkowalne w a, gdy $a \in \operatorname{int} A$ (czyli istnieje $\varepsilon > 0$ takie, że $x \in A$ dla każdego $x \in \mathbb{R}^n$ takiego, że $\|x-a\| < \varepsilon$) oraz istnieje odwzorowanie liniowe $L : \mathbb{R}^n \to \mathbb{R}^m$ takie, że dla każdego $\varepsilon > 0$ istnieje takie $r > 0$, że dla każdego $h \in \mathbb{R}^n$ takiego, że $\|h\| < r$ i $a+h \in A$ zachodzi $\|f(a+h) - f(a) - L(h)\| < \varepsilon \|h\|$. Inaczej mówiąc $\lim_{h\to 0} \frac{\|f(a+h)-f(a)-L(h)\|}{\|h\|} = 0$.

Mamy $L(x) = \sum_{i=1}^n L_i x_i$ dla pewnych $L_1, \ldots, L_n \in \mathbb{R}^m$. Jeśli f jest różniczkowalna, to jej pochodne cząstkowe istnieją i $\frac{\partial f}{\partial x_i}(a) = L_i$. W przypadku funkcji o wartościach skalarnych (z $\mathbb{R}$) oznaczamy różniczkę f przez $\nabla f(a) = \left[\frac{\partial f}{\partial x_1}(a), \ldots, \frac{\partial f}{\partial x_n}(a)\right]$ i nazywamy ją gradientem. Utożsamiamy tu funkcję liniową z wektorem poziomym (elementem przestrzeni dualnej).

W ogólnym przypadku jeśli rozpiszemy $f = (f_1, \ldots, f_m)$ przy $f_j : A \to \mathbb{R}$, to mamy $\frac{\partial f}{\partial x_i}(a) = \left(\frac{\partial f_1}{\partial x_i}(a), \ldots, \frac{\partial f_m}{\partial x_i}(a)\right)$. Możemy wtedy rozpisać różniczkę jako

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1}(a) & \ldots & \frac{\partial f_1}{\partial x_n}(a) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(a) & \ldots & \frac{\partial f_m}{\partial x_n}(a) \end{bmatrix}.$$

Tę macierz nazywamy jakobianem f w punkcie a. Oznaczamy ją czasem $D_a f$ lub $\operatorname{Jac}_a f$.

Twierdzenie. Niech $A \subseteq \mathbb{R}^n$. Niech $f : A \to \mathbb{R}^m$ będzie funkcją, $a \in A$. Niech $\Omega \subseteq A$ będzie otwartym otoczeniem a takim, że na Ω funkcja f ma wszystkie pochodne cząstkowe i $\frac{\partial f}{\partial x_i} : \Omega \to \mathbb{R}^m$ jest ciągłe dla każdego i. Wtedy f jest różniczkowalna w a.

Funkcja $\frac{\partial f}{\partial x_i}$ może być różniczkowalna albo przynajmniej mieć pochodne cząstkowe. j-tą pochodną cząstkową $\frac{\partial f}{\partial x_i}$ oznaczamy $\frac{\partial^2 f}{\partial x_j \partial x_i}$. Nazywamy ją pochodną cząstkową f drugiego rzędu. Pojęcie dwukrotnej różniczkowalności jest trochę bardziej skomplikowane niż w przypadku funkcji jednej zmiennej (pochodna funkcji to już nie liczba, tylko macierz, więc odwzorowanie pochodne przypisujące punktowi różniczkę funkcji w tym punkcie ma inny typ niż wyjściowe odwzorowanie). Wystarczy nam wiedzieć, że funkcja jest dwukrotnie różniczkowalna w punkcie, gdy ma ciągłe (w otoczeniu tego punktu) drugie pochodne cząstkowe. Zachodzi

Twierdzenie (Schwarz)**.** Niech $A \subseteq \mathbb{R}^n$. Niech $f : A \to \mathbb{R}^m$ będzie funkcją dwukrotnie różniczkowalną w punkcie $a \in A$. Wtedy $\frac{\partial^2 f}{\partial x_i \partial x_j}(a) = \frac{\partial^2 f}{\partial x_j \partial x_i}(a)$.

I.1.7 Ekstrema

Ekstrema funkcji wielu zmiennych. Efektywne metody wyznaczania ekstremów lokalnych funkcji wielu zmiennych.

Twierdzenie (Warunek konieczny istnienia ekstremum)**.** Niech $A \subseteq \mathbb{R}^n$. Niech $f : A \to \mathbb{R}$ będzie funkcją różniczkowalną w $a \in A$. Jeśli f ma w a ekstremum lokalne, to $\nabla f(a) \equiv 0$ (wszystkie pochodne cząstkowe są zerowe).

Definicja. Niech $A \subseteq \mathbb{R}^n$. Niech $f : A \to \mathbb{R}$ będzie funkcją dwukrotnie różniczkowalną w $a \in A$. Macierz

drugich pochodnych

$$H_f(a) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(a) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(a) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(a) & \dots & \frac{\partial^2 f}{\partial x_n \partial x_n}(a) \end{bmatrix}$$

nazywamy hesjanem f w punkcie a.

Definicja. Niech $M \in \mathbb{R}^{n\times n}$ będzie macierzą kwadratową. Mówimy, że M jest dodatnio (ujemnie) określona, jeśli $x^T Mx > 0$ ($x^T Mx < 0$) dla każdego $x \in \mathbb{R}^n$.

Twierdzenie. Niech $A \subseteq \mathbb{R}^n$. Niech $f : A \to \mathbb{R}$ będzie funkcją dwukrotnie różniczkowalną w $a \in A$. Jeśli $\nabla f(a) \equiv 0$ i hesjan $H_f(a)$ jest dodatnio (ujemnie) określony, to f ma w a minimum (maksimum) lokalne.

Twierdzenie (Kryterium Sylvestera)**.** Niech $M \in \mathbb{R}^{n\times n}$ będzie symetryczną macierzą kwadratową. Niech m_i oznacza i-ty minor wiodący tej macierzy (wyznacznik macierzy powstałej przez odrzucenie ostatnich $n-i$ wierszy i kolumn). M jest dodatnio określona wtedy i tylko wtedy, gdy $m_i > 0$ dla każdego i. M jest ujemnie określona wtedy i tylko wtedy, gdy $(-1)^i m_i > 0$ dla każdego i.

Powyższe twierdzenia dają nam metodę wyznaczania ekstremów lokalnych funkcji różniczkowalnych: patrzymy, gdzie gradient się zeruje a następnie w tych punktach sprawdzamy określoność za pomocą kryterium Sylvestera (hesjan jest symetryczny na mocy twierdzenia Schwarza).

Jednak nie zawsze rozważane przez nas funkcje są (wszędzie) różniczkowalne. Na przykład dla $B = \{(x,y) \in \mathbb{R}^2 : x^2+y^2 \leqslant 1\}$ funkcja $f : B \ni x \to e^{x+y}$ jest różniczkowalna na $\{(x,y) \in \mathbb{R}^2 : x^2+y^2 < 1\}$, ale na $\{(x,y) \in \mathbb{R}^2 : x^2+y^2 = 1\}$ już nie, bo ten zbiór leży poza wnętrzem B. Aby znaleźć ekstrema f musimy więc osobno rozważyć wnętrze B i ten zbiór (brzeg).

Twierdzenie (Mnożniki Lagrange'a)**.** Niech $V \subseteq \mathbb{R}^n$. Niech $h = (h_1, \dots, h_{n-m}) : V \to \mathbb{R}^{n-m}$ posiada wszystkie pochodne cząstkowe i niech będą one ciągłe. Załóżmy, że rząd macierzy $D_a h$ jest maksymalny (wynosi $n-m$) dla każdego $a \in V$. Niech $f : V \to \mathbb{R}$ będzie różniczkowalne. Oznaczmy $F : V \times \mathbb{R}^{n-m} \ni (x,\lambda) \to f(x) - \sum_{j=1}^{n-m} \lambda_j h_j(x) \in \mathbb{R}$. Niech $K = \{x \in V : h(x) = 0\}$. Dla każdego $\widetilde{x}$ będącego ekstremum funkcji $f|_K$ istnieje takie $\widetilde{\lambda} \in \mathbb{R}^{n-m}$, że $\nabla F\left(\widetilde{x}, \widetilde{\lambda}\right) = 0$.

Powyższe twierdzenie daje nam metodę znajdowania ekstremów funkcji zadanych jakimś (ładnym) równaniem: liczymy gradient funkcji F i patrzymy, gdzie się zeruje. Następnie badamy uzyskane punkty innymi metodami. Funkcję h spełniającą warunki z powyższego twierdzenia nazywamy submersją a zbiór K nazywamy m-wymiarową podrozmaitością $\mathbb{R}^n$. W większości praktycznych zastosowań nie przejmujemy się założeniami o h.

I.1.8 Całka wielu zmiennych

> Całkowanie funkcji wielu zmiennych. Twierdzenie Fubiniego. Twierdzenie o zamianie zmiennych.

Definicja. Kostką nazywamy zbiór $I = \prod_{i=1}^n [a_i, b_i] \subseteq \mathbb{R}^n$ dla $a_i \leqslant b_i$ w $\mathbb{R}$. Objętością kostki nazywamy wartość $\operatorname{vol} I = \prod_{i=1}^m (b_i - a_i)$.

Rozważamy podział P kostki I z $\operatorname{vol} I > 0$ dany przez podziały poszczególnych przedziałów: $a_i = t_{i,0} < \ldots < t_{i,k_i} = b_i$ dla $i = 1, \dots, n$. Wraz z podziałem otrzymujemy skojarzone z nim kostki $\prod_{i=1}^n \left[t_{i,j_i-1}, t_{i,j_i}\right]$ dla $j_i \in \{1, \dots, k_i\}$, które porządkujemy leksykograficznie $I_1, \dots, I_\mu$ oraz średnią podziału $\delta(P) = \max_{j=1}^{\mu(P)} \operatorname{diam} I_j$.

Ciąg podziałów zwiemy normalnym, gdy ich średnie dążą do zera. Rozważając układ punktów pośrednich $\Xi = (\xi_1, \dots, \xi_\mu)$ (gdzie $\xi_i \in I_i$) określamy sumę Riemanna dla danej funkcji ograniczonej $f : I \to \mathbb{R}^m$ przez $S(f, P, \Xi) = \sum_{i=1}^{\mu(P)} f(\xi_i) \operatorname{vol} I_i$.

Całką Riemanna f (o ile istnieje) nazywamy element $\int_I f(x)\,\mathrm{d}x \in \mathbb{R}^m$, który jest granicą $S(f, P_\nu, \Xi_\nu)$ niezależną of wyboru normalnego ciągu podziałów P_ν kostki I oraz od wyboru ciągu układów punktów pośrednich Ξ_ν. Funkcję nazywamy całkowalną w sensie Riemanna na I, jeśli istnieje jej całka Riemanna.

Za pomocą powyższej definicji możemy zdefiniować całkę po (w miarę) dowolnym zbiorze $A \subseteq \mathbb{R}^n$. Robimy to przybliżając ten zbiór za pomocą kostek o coraz krótszych bokach i korzystając z addytywności całki. Nie jest to zbyt formalny opis, ale aby zrobić to lepiej musielibyśmy wchodzić w teorię całki Lebesgue'a, a tego nie chcemy robić.

Twierdzenie (Fubini)**.** Niech $A \subseteq \mathbb{R}^n$ będzie zbiorem, na którym funkcja $f : A \to \mathbb{R}^m$ jest całkowalna. Oznaczmy $A' = \{x_1 \in \mathbb{R} : \exists_{y \in A}\ y_1 = x_1\}$ i $A_{x_1} = \left\{\widetilde{x} \in \mathbb{R}^{n-1} : (x_1, \widetilde{x}) \in A\right\}$. Zachodzi

$$\int_A f(x)\,\mathrm{d}x = \int_{A'} \left(\int_{A_{x_1}} f(x_1, \widetilde{x})\,\mathrm{d}\widetilde{x} \right) \mathrm{d}x_1 \,.$$

To twierdzenie pozwala nam liczyć całkę wielowymiarową jako całkę iterowaną: ustalamy wartość na pierwszej współrzędnej, indukcyjnie liczymy całkę o mniejszym wymiarze przy ustalonej wartości funkcji na pierwszej współrzędnej, a następnie to co nam wyjdzie całkujemy po „dziedzinie" pierwszej współrzędnej, co jest całką jednowymiarową.

Całkiem ważne jest, że kolejność współrzędnych nie ma znaczenia: możemy dowolnie wybrać kolejność współrzędnych, po których liczymy kolejne całki.

Ciężko jest stwierdzić całkowalność funkcji wielowymiarowej (zwłaszcza, że nie bardzo znamy definicję całkowalności). W typowej sytuacji zakładamy, że napotkane przez nas funkcje są całkowalne. Z teorii miary Lebesgue'a wynika, że (o ile A jest ładny) teza twierdzenia Fubiniego zachodzi dla funkcji o stałym znaku. Jeśli f przyjmuje zarówno wartości dodatnie, jak i ujemne, to a pewno zachodzi ona, gdy obie funkcje $\max(f, 0)$, $\min(f, 0)$ mają skończone całki. Te całki możemy liczyć jako całki iterowane korzystając z twierdzenia Fubiniego dla tych funkcji.

Twierdzenie (O zamianie zmiennych)**.** Niech $A, B \subseteq \mathbb{R}^n$. Niech $f : \mathbb{R}^n \to \mathbb{R}^m$ będzie funkcją całkowalną na A. Niech $\varphi : B \to A$ będzie dyfeomorfizmem (bijekcją taką, że φ i φ^{-1} mają ciągłe pochodne cząstkowe). Wtedy

$$\int_A f(y)\,\mathrm{d}y = \int_B f(\varphi(x))\,|\det D_x\varphi|\,\mathrm{d}x \,.$$

Ważnym przykładem zamiany zmiennych są współrzędne biegunowe. Przyjmując

$$\varphi : [0,1] \times [0, 2\pi) \ni (r, \theta) \to (r\cos\theta, r\sin\theta) \in \left\{(x,y) \in \mathbb{R}^2 : x^2 + y^2 \leqslant 1\right\}$$

jesteśmy w stanie zamienić trudną całkę po kole (lub podobnych obiektach) na w miarę łatwą całkę po prostokącie. Zauważmy, że tutaj jakobian jest dość przyjemny, bo

$$\det \begin{bmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{bmatrix} = r\left(\cos^2\theta + \sin^2\theta\right) = r.$$

I.2 Matematyka Dyskretna

I.2.1 Współczynniki dwumianowe

Na ile sposobów możemy rozdać n nierozróżnialnych żetonów k rozróżnialnym osobom? Jak zmienia się odpowiedź gdy założymy, że każda osoba powinna otrzymać co najmniej jeden żeton?

Możemy myśleć o tym problemie w następujący sposób: mamy $n + k - 1$ żetonów ułożonych w ciąg. Chcemy wybrać spośród nich $k - 1$. Wtedy możemy dać i-tej osobie wszystkie niewybrane żetony leżące pomiędzy $(i - 1)$-szym a i-tym wybranym żetonem (pierwszej osobie dajemy wszystkie żetony przed pierwszym wybranym żetonem). Każdy wybór żetonów doprowadzi do innego podziału i jednocześnie dla każdego podziału żetonów między osoby jesteśmy w stanie wskazać taki wybór żetonów w tym ciągu, że przedstawiona procedura wyprodukuje ten właśnie podział. Zatem istnieje bijekcja między rozważanymi strukturami i faktycznie wystarczy, że zliczymy ilość możliwych wyborów żetonów w ciągu.

Liczba sposobów, na ile można wybrać ℓ elementów z r-elementowego zbioru to (z definicji) $\binom{r}{\ell}$. Zatem odpowiedzią na pierwsze z zadanych pytań jest $\binom{n+k-1}{k-1}$. W drugim pytaniu możemy rozdać każdej osobie po jednym żetonie a następnie rozdać $n - k$ żetonów bez żadnych ograniczeń. Zatem odpowiedzią jest $\binom{n-1}{k-1}$.

Można jeszcze powiedzieć, że $\binom{r}{\ell} = \begin{cases} \frac{r!}{\ell!(r-\ell)!}, & 0 \leqslant \ell \leqslant r \\ 0, & r < \ell \end{cases}$: mamy $\binom{r}{\ell} \cdot \ell! \cdot (r - \ell)! = r!$, bo prawa strona zlicza permutacje r-elementowego zbioru, a lewa też: najpierw ustalamy pierwsze ℓ elementów, permutujemy je, a potem pozostałe.

I.2.2 Liczby Catalana

Liczby Catalana $(C_n)_{n\in\mathbb{N}}$ zdefiniowane jako liczba ścieżek Dycka długości $2n+1$. Wyprowadzenie wzoru rekurencyjnego. Przykłady rodzin obiektów zliczanych przez liczby Catalana.

Definicja. Nieparzystym słowem (ścieżką) Dycka nazywamy ciąg $x = (x_1, \dots, x_{2n+1}) \in \{-1, 1\}^{2n+1}$ taki, że dla każdego $k \in [2n + 1]$ jest $\sum_{i=1}^{k} x_i > 0$ oraz $\sum_{i=1}^{2n+1} x_i = 1$. Zbiór nieparzystych słów Dycka długości $2n + 1$ oznaczamy S_n. Liczbę $C_n = |S_n|$ nazywamy n-tą liczbą Catalana.

Mamy $C_0 = 1$ (jedyne słowo Dycka długości 1 to (1)). Ogólnie zachodzi wzór rekurencyjny

Twierdzenie.

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

Dowód. Rozważmy pewne słowo Dycka $x = (x_1, \dots, x_{2n+3})$ długości $2n + 3$. Jeśli jedyne sumy częściowe równe 1 to pierwsza i ostatnia, to $(x_2, \dots, x_{2n+2})$ jest ścieżką Dycka długości $2n + 1$. Do tego może to być dowolna ścieżka Dycka, więc istnieje C_n różnych x spełniających tę własność.

Teraz załóżmy, że $\ell \in \{2, \dots, 2n + 2\}$ jest największym takim indeksem, że $\sum_{i=1}^{\ell} x_i = 1$. ℓ musi być nieparzyste, bo dodajemy do siebie 1 i -1, więc nieparzystą wartość możemy osiągnąć tylko sumując nieparzyście wiele składników. Zatem $\ell = 2k + 1$ dla pewnego k. $(x_1, \dots, x_{2k+1})$ jest słowem Dycka długości $2k + 1$ i może być dowolnym takim słowem. Z kolei $(x_{2k+2}, \dots, x_{2n+3})$ jest ciągiem długości $2(n - k) + 2$ takim, że $\sum_{i=2k+2}^{2n+3} x_i = 0$, $\sum_{i=2k+2}^{j} x_i > 0$ dla $j \in \{2k + 2, \dots, 2n + 2\}$ oraz $x_{2n+3} = -1$

(ostatnim elementem słowa Dycka zawsze jest -1). Zatem $(x_{2k+2}, \ldots, x_{2n+2})$ jest słowem Dycka długości $2(n-k)+1$ i może być dowolnym takim słowem. Ostatecznie dostajemy $C_k \cdot C_{n-k} \cdot 1$ różnych x (pierwsze $2k+1$ elementów, kolejne $2(n-k)+1$ i ostatni pewny).

Sumując się po wszystkich możliwych k i dodając przypadek rozważony osobno mamy

$$C_{n+1} = \sum_{k=0}^{n-1} C_k C_{n-k} + C_n \cdot 1 = \sum_{k=0}^{n} C_k C_{n-k}.$$

□

Rekurencja, która zadaje ciąg liczb Catalana jest bardzo naturalna i dlatego ciąg ten zlicza bardzo wiele różnych obiektów. Oto kilka przykładów obiektów, których jest C_n:

1. Parzyste słowa Dycka długości $2n$, czyli ciągi $x = (x_1, \ldots, x_{2n}) \in \{-1, 1\}^{2n}$ takie, że $\sum_{i=1}^{k} x_i \geqslant 0$ dla każdego $k \in [2n]$ oraz $\sum_{i=1}^{2n} x_i = 0$.
2. Ścieżki kratowe Dycka długości $2n$, czyli ścieżki na kracie $n \times n$ idące w jednym kroku w górę lub w prawo i łączące punkt $(0, 0)$ z punktem (n, n) które nie wychodzą nad główną przekątną $y = x$, czyli w każdym punkcie ścieżki (a, b) mamy $a \leq b$.
3. Drzewa narysowane o $n+1$ węzłach (ukorzenione drzewa z zadanym porządkiem na dzieciach każdego węzła).
4. Drzewa binarne o n węzłach (zbiór pusty jest drzewem binarnym, rozróżniamy prawe i lewe dziecko węzła).
5. Pełne drzewa binarne o $2n+1$ węzłach (nie ma węzłów z dokładnie jednym dzieckiem).
6. Triangulacje $(n+2)$-kąta foremnego (podział na trójkąty za pomocą nieprzecinających się odcinków o końcach w wierzchołkach).
7. Możliwe nawiasowania ciągu n operacji binarnych.

W każdym przypadku łatwo wykazać poprawność odpowiedniego wzoru rekurencyjnego. Można też wskazywać bijekcje między różnymi z tych obiektów.

I.2.3 Twierdzenie Spernera

Twierdzenie Spernera o szerokości kraty boolowskiej. Wypowiedź i dowód.

Kratą boolowską rzędu n nazywamy zbiór częściowo uporządkowany $\mathcal{B}_n = (\mathcal{P}([n]), \subseteq)$. Antyłańcuchem nazywamy zbiór elementów zbioru częściowo uporządkowanego (posetu) taki, że żadne dwa jego elementy nie są porównywalne. W przypadku $\mathcal{B}_n$ zbiór $A_1, \ldots, A_k$ jest antyłańcuchem, jeśli dla każdego $i \neq j$ zachodzi $A_i \nsubseteq A_j$. Szerokością posetu nazywamy rozmiar najliczniejszego antyłańcucha w nim. Zachodzi następujące twierdzenie.

Twierdzenie (Sperner 1928)**.** Szerokość $\mathcal{B}_n$ wynosi $\binom{n}{\lfloor \frac{n}{2} \rfloor}$.

Przykładem antyłańcucha o takiej mocy jest zbiór $\lfloor \frac{n}{2} \rfloor$-elementowych podzbiorów $[n]$. Zatem wystarczy pokazać, że żaden antyłańcuch nie może mieć więcej niż $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ elementów. Jednym ze sposobów na udowodnienie tego jest rozważenie dowolnego antyłańcucha $\mathcal{A}$ i stopniowe „spychanie" jego elementów w kierunku środka kraty. Aby sformalizować to podejście potrzebujemy następującego pojęcia.

Definicja. Dla zbioru $\mathcal{T} \subset \binom{[n]}{k}$ jego cieniem dolnym i cieniem górnym nazywamy odpowiednio zbiory

$$\Delta\mathcal{T} = \{A : \exists_{T \in \mathcal{T}, x \in T}\ A = T \setminus \{x\}\}, \qquad \nabla\mathcal{T} = \left\{A : \exists_{T \in \mathcal{T}, x \in [n] \setminus T}\ A = T \cup \{x\}\right\}.$$

Lemat. Dla $\mathcal{T} \subset \binom{[n]}{k}$ zachodzi $|\Delta\mathcal{T}| \geqslant \frac{k}{n-k+1} |\mathcal{T}|$ oraz $|\nabla\mathcal{T}| \geqslant \frac{n-k}{k+1} |\mathcal{T}|$. Wynika z tego, że $|\Delta\mathcal{T}| \geqslant |\mathcal{T}|$ dla $k \geqslant \frac{n+1}{2}$ oraz $|\nabla\mathcal{T}| \geqslant |\mathcal{T}|$ dla $k \leqslant \frac{n-1}{2}$.

Dowód. Zliczamy moc zbioru $W = \{(A,T) : T \in \mathcal{T}, A \in \Delta\mathcal{T}, A \subset T\}$. Jest ona równa $k|\mathcal{T}|$, bo każdy element $\mathcal{T}$ ma dokładnie k swoich elementów cienia. Jednocześnie każdy element cienia może mieć co najwyżej $n-(k-1)$ swoich nadzbiorów w $\mathcal{T}$, więc $|W| \leqslant |\Delta\mathcal{T}|(n-k+1)$, co dowodzi pierwszej nierówności. Podobne zliczenie dla górnego cienia (każdy element $\mathcal{T}$ ma $n-k$ swoich elementów cienia, element cienia ma co najwyżej $k+1$ elementów $\mathcal{T}$) daje drugą nierówność. □

Teraz możemy przejść do właściwego dowodu.

Dowód twierdzenia Spernera. Niech $\mathcal{A}$ będzie antyłańcuchem w $\mathcal{B}_n$ i niech $\mathcal{A}_j = \mathcal{A} \cap \binom{[n]}{j}$. Jeśli $i = \min\{j : \mathcal{A}_j \neq \emptyset\}$, to dla $i \leqslant \frac{n-1}{2}$ zbiór $\mathcal{A}' = (\mathcal{A} \setminus \mathcal{A}_i) \cup \nabla\mathcal{A}_i$ ma nie mniejszą moc od $\mathcal{A}$ oraz dalej jest antyłańcuchem – jeśli coś jest nad cieniem górnym $\mathcal{A}_i$, to jest też nad $\mathcal{A}_i$, więc $\mathcal{A}$ nie byłby antyłańcuchem, gdyby $\mathcal{A}'$ nie był. Podobnie, jeśli weźmiemy $k = \max\{j : \mathcal{A}_j \neq \emptyset\}$ i będzie $k \geqslant \frac{n+1}{2}$. Możemy więc po kolei przesuwać kolejne poziomy bliżej środka kraty. Jeśli $2 \nmid n$, to możemy wybrać dowolny ze środkowych poziomów, bo nierówności z cieniami na to pozwalają. Ostatecznie otrzymujemy antyłańcuch, którego wszystkie elementy są zbiorami o tej samej liczności, równej $\lfloor \frac{n}{2} \rfloor$ lub $\lceil \frac{n}{2} \rceil$. To daje nam tezę. □

I.2.4 Funkcje tworzące

Funkcje tworzące. Wyznaczenie zwartego wzoru na liczby Fibonacciego za pomocą funkcji tworzących.

Szeregiem formalnym nad ciałem $\mathbb{F}$ nazywamy dowolny ciąg $(a_n)_{n=0}^{\infty}$ elementów $\mathbb{F}$. Szereg formalny zwykle przedstawiamy za pomocą napisu $\sum_{n=0}^{\infty} a_n x^n$, gdzie $a_n \in \mathbb{F}$ a x jest zmienną formalną (napisem, który nie jest w żaden sposób zależny algebraicznie od elementów $\mathbb{F}$). Ta postać pozwala nam naturalnie zdefiniować dodawanie i mnożenie takich obiektów: dodajemy „po współrzędnych" $\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (a_n + b_n) x^n$, a mnożymy grupując wyrazy tego samego stopnia $\sum_{n=0}^{\infty} a_n x^n \cdot \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} \sum_{k=0}^{n} a_k b_{n-k} x^n$. Zbiór szeregów formalnych z tymi działaniami, oznaczany $\mathbb{F}[[X]]$, ma strukturę pierścienia.

Funkcją tworzącą ciągu $(a_n)_{n=0}^{\infty}$ nazywamy szereg formalny $\sum_{n=0}^{\infty} a_n x^n$. Często będzie nas interesowało wyznaczenie zwartej postaci funkcji tworzącej (czyli takiej nie korzystającej ze znaku sumy). Dla przykładu $\sum_{n=0}^{\infty} 1x^n = \frac{1}{1-x}$, bo $(1-x)\sum_{n=0}^{\infty} x^n = \sum_{n=0}^{\infty} x^n - \sum_{n=1}^{\infty} x^n = 1$, więc $1-x$ jest odwrotnością $\sum_{n=0}^{\infty} x^n$ w pierścieniu $\mathbb{F}[[X]]$. W analogiczny sposób dostajemy $\frac{1}{1-rx} = \sum_{n=0}^{\infty} r^n x^n$.

Ciąg Fibonacciego $(F_n)_{n=0}^{\infty}$ zadany jest przez $F_0 = 0$, $F_1 = 1$ i $F_n = F_{n-1} + F_{n-2}$ dla $n \geqslant 2$. Niech $F(x)$ będzie jego funkcją tworzącą. Mamy

$$F(x) = \sum_{n=0}^{\infty} F_n x^n = 0 + x + \sum_{n=2}^{\infty} (F_{n-1} + F_{n-2}) x^n = 0 + x + x\sum_{n=2}^{\infty} F_{n-1}x^{n-1} + x^2 \sum_{n=2}^{\infty} F_{n-2}x^{n-2}$$
$$= 0 + x + x\left(\sum_{n=0}^{\infty} F_n x^n - F_0 x^0\right) + x^2 \sum_{n=0}^{\infty} F_n x^n = x + xF(x) + x^2 F(x).$$

To daje nam $F(x)\left(x^2 + x - 1\right) = -x$, czyli $F(x) = \frac{x}{1-x-x^2}$. Możemy rozwiązać równanie kwadratowe w mianowniku i dostać $1 - x - x^2 = \left(x - \frac{-1+\sqrt{5}}{2}\right)\left(x - \frac{-1-\sqrt{5}}{2}\right) = (1-\varphi x)(1-\psi x)$, gdzie $\varphi = \frac{1+\sqrt{5}}{2}$ i $\psi = \frac{1-\sqrt{5}}{2}$. Mamy więc

$$F(x) = \frac{x}{(1-\varphi x)(1-\psi x)} = \frac{A}{1-\varphi x} + \frac{B}{1-\psi x}$$

dla pewnych A, B. Można znaleźć A, B rozwiązując odpowiedni układ równań albo po prostu zgadując, że $A = \frac{1}{\sqrt{5}}$ i $B = -\frac{1}{\sqrt{5}}$ działa. Korzystając z poprzednich faktów o funkcjach tworzących mamy $F(x) = A\sum_{n=0}^{\infty} \varphi^n x^n + B\sum_{n=0}^{\infty} \psi^n x^n$, co daje nam wzór zwany wzorem Bineta:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

I.2.5 Twierdzenie Ramseya

Twierdzenie Ramseya. Wypowiedź i dowód.

Definicja. Liczba Ramseya $R^{(p)}(k;\ell_1,\ell_2,\dots,\ell_k)$, gdzie $k \geqslant 1, \ell_i \geqslant p \geqslant 1$ to najmniejsza taka liczba N, że dla każdego kolorowania $c : \binom{[N]}{p} \to [k]$ istnieje $X \subset [N]$ oraz $\alpha \in [k]$ takie, że $|X| = \ell_\alpha$ oraz $\binom{X}{p} \subseteq c^{-1}(\alpha)$, czyli każdy podzbiór X jest kolorowany tym samym kolorem i X spełnia ograniczenie wielkościowe dla tego koloru.

Twierdzenie (Ramsey 1930)**.** Liczba Ramseya $R^{(p)}(k;\ell_1,\ell_2,\dots,\ell_k)$ jest dobrze zdefiniowana, czyli istnieje liczba spełniająca taką własność.

Dowód. Zastosujemy indukcję po $(p,k,\sum_j \ell_j)$. Dla $p=1$ jest $R^{(1)}(k;\ell_1,\dots,\ell_k) \leqslant \sum_j \ell_j$, a dla $k=1$ jest $R^{(p)}(1;\ell_1) \leqslant \ell_1$. Jeśli $\exists_{j\in[k]}\ \ell_j = p$, to $R^{(p)}(k;\ell_1,\dots,\ell_k) = R^{(p)}(k-1;\ell_1,\dots,\ell_{j-1},\ell_{j+1},\dots,\ell_k)$, bo albo pokolorowaliśmy pewien zbiór p-elementowy kolorem j i warunek dla j jest spełniony, albo nie korzystaliśmy z tego koloru i sytuacja sprowadza się do kolorowania mniejszą liczbą kolorów. Dalej zakładamy $p,k \geqslant 2$ oraz $\forall_{j\in[k]}\ \ell_j \geqslant p+1$.

Niech $L_j = R^{(p)}(k;\ell_1,\dots,\ell_j - 1,\dots,\ell_k)$ (istnieje z indukcji). Niech $N = 1 + R^{(p-1)}(k;L_1,\dots,L_k)$. Pokażemy $R^{(p)}(k;\ell_1,\dots,\ell_k) \leqslant N$. Rozważmy dowolne $c : \binom{[N]}{p} \to [k]$ i wybierzmy dowolne $v \in [N]$. Definiujemy $c' : \binom{[N]\setminus\{v\}}{p-1} \to [k]$ kładąc $c'(X) = c(X \cup \{v\})$. Z definicji N istnieje zbiór Y taki, że $|Y| = L_\alpha$ dla pewnego $\alpha \in [k]$ oraz $\binom{Y}{p-1}$ jest monochromatyczny w c'.

Z definicji L_α w Y istnieje monochromatyczny w c zbiór wielkości ℓ_j (dla $j \neq \alpha$) lub wielkości $\ell_\alpha - 1$. W pierwszy przypadku mamy to, czego szukamy. W drugim przypadku oznaczmy znaleziony zbiór przez T. Wtedy $T \cup \{v\}$ jest rozmiaru ℓ_α i jest monochromatyczny w c, bo jego podzbiory zawierające v i niezawierające v są wszystkie tego samego koloru (odpowiednio z określenia Y i T). Mamy więc to, czego szukamy. □

I.2.6 Twierdzenie Brooksa

Twierdzenie Brooksa o kolorowaniu grafów. Wypowiedź i dowód.

Definicja. Kolorowanie wierzchołkowe grafu G to funkcja $c : V(G) \to S$ dla pewnego zbioru S. Jeśli $\forall_{uv\in E(G)}\ c(u) \neq c(v)$, to kolorowanie nazywamy właściwym. Jeśli $|S| \leqslant k$, to c nazywamy k-kolorowaniem. Liczba chromatyczna grafu G, oznaczana $\chi(G)$, to najmniejsza taka liczba k, że istnieje właściwe k-kolorowanie grafu G.

Propozycja. W grafie G zachodzi $\chi(G) \leqslant \Delta(G) + 1$, gdzie $\Delta(G)$ oznacza maksymalny stopień.

Dowód. Przedstawimy algorytm kolorowania nazywany First-Fit. Rozważmy dowolny porządek liniowy na wierzchołkach $v_1,\dots,v_n$. Kolorujemy je w kolejności tego porządku, definiujemy kolorowanie $c(v_i) = \min\{k \in \mathbb{N}_1 : k \notin c(N_G(v_i) \cap \{v_1,\dots,v_{i-1}\})\}$. Każdy wierzchołek ma co najwyżej $\Delta(G)$ sąsiadów, więc tyle kolorów może być zajętych. □

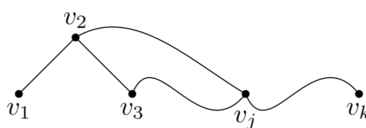
Twierdzenie (Brooks 1941)**.** Dla spójnego grafu G, który nie jest kliką ani nieparzystym cyklem, zachodzi $\chi(G) \leqslant \Delta(G)$.

Dowód. Jeśli $\Delta(G) = 1$, to jedyną możliwością jest $G = K_2$, co jest zabronione przez założenia. Jeśli $\Delta(G) = 2$, to G jest ścieżką lub cyklem (z założeń parzystym) i da się pokolorować G za pomocą 2 kolorów. Dalej zakładamy $\Delta \geqslant 3$. Załóżmy nie wprost, że teza nie zachodzi i rozważmy minimalny na liczbę wierzchołków kontrprzykład G. Oznaczmy $\Delta = \Delta(G)$.

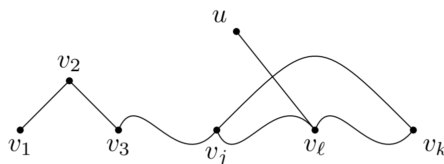
Załóżmy, że w G istnieje wierzchołek v stopnia mniejszego niż Δ. Rozważmy S będące spójną składową $G - v$. Jeśli S jest kliką, to jest kliką rzędu co najwyżej Δ (inaczej któryś z sąsiadów v w G musiałby mieć stopień $> \Delta$), którą można pokolorować za pomocą Δ kolorów. Jeśli S jest nieparzystym cyklem, to można go pokolorować za pomocą $3 \leqslant \Delta$ kolorów. Jeśli S nie jest kliką ani nieparzystym cyklem, to z minimalności G można pokolorować S za pomocą Δ kolorów. Zatem można pokolorować $G - v$ za pomocą Δ kolorów (używając tych samych na każdej spójnej składowej), a następnie pokolorować v za pomocą tych samych kolorów (bo ma mniej niż Δ sąsiadów). To daje sprzeczność z określeniem G. Zatem takie v nie istnieje i dalej zakładamy, że G jest Δ-regularny.

Pokażemy, że z tego, że G jest spójny i nie jest kliką wynika, że istnieją wierzchołki v_1, v_2, v_3 takie, że $v_1v_2, v_2v_3 \in E(G)$ i $v_1v_3 \notin E(g)$. Załóżmy nie wprost, że każdy wierzchołek v sąsiaduje ze wszystkimi sąsiadami swoich sąsiadów. Ze spójności dla dowolnych wierzchołków v, w istnieje ścieżka $vx_1x_2 \dots w$. x_1 jest sąsiadem v, a x_2 jest sąsiadem x_1, więc x_2 jest sąsiadem v. Kontynuując takie rozumowanie dochodzimy do tego, że w jest sąsiadem v. Wobec ich dowolności graf jest kliką, co daje sprzeczność.

Niech $P = v_1v_2v_3 \dots v_k$ będzie najdłuższą ścieżką zaczynającą się od wierzchołków v_1, v_2, v_3 o wcześniej opisanej własności. Wszyscy sąsiedzi v_k na niej leżą, bo inaczej można by ją wydłużyć. Najpierw rozważmy przypadek, w którym ta ścieżka zawiera wszystkie wierzchołki w grafie. Niech $j = \max\{i \in [k] : v_2v_i \in E(G)\}$. Z $\Delta \geqslant 3$ jest $j \geqslant 4$. Kolorujemy wierzchołki algorytmem First-Fit w następującej kolejności: v_1, v_3, które dostają ten sam kolor, potem kolejno $v_4, \dots, v_{j-1}$. Każdy z tych wierzchołków w momencie kolorowania ma niepokolorowanego sąsiada (kolejny wierzchołek na ścieżce), więc zablokowanych jest mniej niż Δ kolorów. Następnie kolorujemy kolejno $v_k, \dots, v_j$, które w momencie kolorowania również mają niepokolorowanego sąsiada (poprzedni na ścieżce lub v_2 dla v_j). Ostatecznie kolorujemy v_2: v_1 i v_3 mają ten sam kolor, więc zablokowanych jest mniej niż Δ kolorów. Zatem zdefiniowane kolorowanie używa co najwyżej Δ kolorów.



Teraz załóżmy, że $V(P) \neq V(G)$. Niech $j = \min\{i \in [k-1] : v_iv_k \in E(G)\}$. $C = v_jv_{j+1} \dots v_k$ jest cyklem. Kolorujemy spójne składowe $G - C$ korzystając z minimalności G (ponownie trzeba rozważyć przypadki klik i cykli, ale one działają tak samo jak przedtem). Teraz ustalamy $\ell = \max\left\{i \in \{j, \dots, k\} : \exists_{u \in V(G) \setminus \{v_j \dots, v_k\}}\ uv_i \in E(G)\right\}$. Z określenia j jest $\ell < k$. Niech u będzie wierzchołkiem jak w definicji ℓ. Wierzchołek $v_{\ell+1}$ nie ma sąsiadów poza C, więc można go pokolorować na ten sam kolor, co u. Następnie kolorujemy kolejne wierzchołki cyklu: $v_{\ell+2}, \dots, v_\ell$. Każdy z nich w momencie kolorowania ma niepokolorowanego sąsiada (a v_ℓ ma dwóch w tym samym kolorze), więc użyjemy co najwyżej Δ kolorów.



□

I.2.7 Liczba chromatyczna a liczba klikowa

Jakie znasz rodziny grafów o ograniczonej liczbie klikowej i nieograniczonej liczbie chromatycznej? Podaj konstrukcje i uzasadnij własności.

Definicja. Liczba klikowa grafu G, oznaczana $\omega(G)$, to maksymalny rozmiar kliki w G. Zachodzi $\omega(G) \leqslant \chi(G)$, bo kliki nie da się pokolorować mniejszą ilością kolorów niż jej wielkość.

Liczba klikowa jest dolnym ograniczeniem na liczbę chromatyczną, ale w ogólności nie ma żadnej zależności w drugą stronę – istnieją przykłady, gdzie liczba klikowa jest równa 2, a liczba chromatyczna jest dowolnie duża. Oto kilka z nich.

Twierdzenie (konstrukcja Zykova)**.** Niech rodzina grafów $\{G_n\}_{n\geqslant 1}$ będzie zadana w następujący sposób: G_1 jest jednowierzchołkowym grafem, G_{i+1} definiujemy rekurencyjnie: bierzemy wszystkie grafy $G_1, \dots, G_i$ i kładziemy obok siebie, dla każdego ciągu wierzchołków $(v_1, \dots, v_i) \in G_1 \times \dots \times G_i$ tworzymy nowy wierzchołek z krawędziami do dokładnie tych wierzchołków. W tak zadanej rodzinie zachodzi $\forall_n\ \omega(G_n) = 2, \chi(G_n) \geqslant n$.

Dowód. Przeprowadzimy dowód indukcyjny. Baza jest oczywista. Gdyby G_{i+1} miał trójkąt, to musiałby on zawierać wierzchołki z różnych mniejszych grafów lub nowododane wierzchołki, ale w obu tych grupach nie ma krawędzi pomiędzy wierzchołkami. Niech ϕ będzie kolorowaniem G_{i+1}. Dla każdego $j \in [i-1]$ kopia grafu G_{j+1} zawiera o jeden kolor więcej niż kopia G_j, a więc z każdej kolejnej z tych kopii można wziąć wierzchołek z niewziętym wcześniej kolorem, uzyskując ciąg $(v_1, \dots, v_i)$ wierzchołków, każdy w innym kolorze. Zatem do pokolorowania nowoutworzonego wierzchołka sąsiadującego z tym zbiorem wierzchołków potrzebny jest nowy kolor, co daje $\chi(G_{i+1}) \geqslant i + 1$. □

Twierdzenie (konstrukcja Mycielskiego)**.** Niech rodzina grafów $\{G_n\}_{n\geqslant 1}$ będzie zadana w następujący sposób: G_1 jest jednowierzchołkowym grafem, G_2 jest jedną krawędzią. G_{i+1} definiujemy rekurencyjnie: bierzemy kopię G_i i kopiujemy każdy jej wierzchołek (bez żadnych krawędzi) a następnie łączymy każdy skopiowany wierzchołek z sąsiadami kopiowanego wierzchołka. Dodajemy nowy wierzchołek w i łączymy go ze skopiowanymi wierzchołkami. W tak zadanej rodzinie zachodzi $\forall_n\ \omega(G_n) = 2, \chi(G_n) \geqslant n$.

Dowód. Przeprowadzimy dowód indukcyjny. Baza jest oczywista. Gdyby G_{i+1} miał trójkąt i należał do niego w, to pozostałymi dwoma wierzchołkami byłyby (niepołączone) skopiowane wierzchołki. Jeśli trójkąt zawiera jakieś skopiowane wierzchołki, to możemy je zastąpić wierzchołkami, które kopiują (nie są z nimi połączone, więc nie ma ich w tym trójkącie). W ten sposób otrzymujemy trójkąt w G_i – sprzeczność. Załóżmy, że mamy kolorowanie G_{i+1} za pomocą i kolorów. Żaden skopiowany wierzchołek nie ma tego koloru, co w. Możemy zdefiniować kolorowanie G_i w następujący sposób: wierzchołki G_i pokolorowane inaczej niż w utrzymują swój kolor, a te pokolorowane tak samo jak w dostają kolor swojej kopii. Jest to poprawne kolorowanie, bo kopie wierzchołków mają identyczne sąsiedztwo, co kopiowane wierzchołki (i kolory tego sąsiedztwa nie zmieniają się w takiej procedurze, bo na wstępie są inne niż kolor w). W ten sposób zdefiniowaliśmy poprawne kolorowanie G_i za pomocą $i - 1$ kolorów – sprzeczność. □

Twierdzenie (konstrukcja Tutte'a)**.** Niech rodzina grafów $\{G_n\}_{n\geqslant 1}$ będzie zadana w następujący sposób: G_1 jest jednowierzchołkowym grafem, G_{i+1} definiujemy rekurencyjnie: bierzemy zbiór wierzchołków I o $|I| = i(k-1) + 1$, gdzie $k = |V(G_i)|$. Dla każdego $X \subset I : |X| = k$ dodajemy nad I graf $G_X \cong G_i$ i pewne skojarzenie doskonałe łączące G_X z X. W tak zadanej rodzinie zachodzi $\forall_n\ \omega(G_n) = 2, \chi(G_n) \geqslant n$.

Dowód. Przeprowadzimy dowód indukcyjny. Baza jest oczywista. Gdyby G_{i+1} miał trójkąt, to musiałyby być w nim 2 wierzchołki z I lub 2 z grafów G_X – one jednak są niepołączone. Załóżmy, że ϕ jest i-kolorowaniem G_{i+1}. Z zasady szufladkowej istnieje zbiór k wierzchołków X o jednym kolorze, jego graf G_X jest pokolorowany bez tego koloru, więc $i-1$ kolorami, co daje sprzeczność z definicją G_X i założeniem indukcyjnym. Zatem $\chi(G_{i+1}) \geqslant i + 1$. □

Twierdzenie (grafy przesunięciowe; Erdős-Hajnal)**.** Niech rodzina grafów $\{G_n\}_{n\geqslant 2}$ będzie zadana w następujący sposób: $V(G_n) = \{(i,j) : 1 \leqslant i < j \leqslant n\}$, $(ij)(k\ell) \in E(G_n) \iff j = k \vee i = \ell$ (przedziały o końcach w liczbach naturalnych, które są takie, że jeden się zaczyna, a drugi kończy). W tak zadanej rodzinie zachodzi $\forall_n\ \omega(G_n) = 2, \chi(G_n) \geqslant \lceil \log n \rceil$.

Dowód. Dla połączonych krawędzią $(i,j), (k,\ell)$ załóżmy bez straty ogólności, że $j = k$. Wtedy jeśli $(d,t)(i,j) \in G_n$, to albo $t = i$, albo $d = j = k$ i w obu przypadkach nie ma krawędzi $(d,t)(k,\ell)$. Zatem G_n nie ma trójkątów. Weźmy kolorowanie $\phi : V(G_n) \to [k]$ i dla każdego $i \in [n]$ zdefiniujmy zbiór $X_i = \{\phi(ij) : j \in \{i+1, \dots, n\}\}$. Mamy $i \neq j \implies X_i \neq X_j$, bo jeśli $\phi(ij) = \alpha$, to X_j nie może zawierać koloru α (poprawność kolorowania), a X_i go zawiera. Zatem $n \leqslant 2^k$, czyli $k \geqslant \lceil \log n \rceil$. □

Twierdzenie (uogólnione grafy przesunięciowe; Erdős-Hajnal)**.** Niech rodzina grafów $\{G_{n,k}\}_{n,k\geqslant 2}$ będzie zadana w następujący sposób: $V(G_n) = \binom{[n]}{k}$, dwie k-krotki $\{x_1 < \ldots < x_k\}$, $\{y_1 < \ldots < y_k\}$ są połączone krawędzią, jeśli $x_2 = y_1, \ldots, x_k = y_{k-1}$ lub $y_2 = x_1, \ldots, y_k = x_{k-1}$ (przesunięcie o jeden element). W tak zadanej rodzinie zachodzi $\forall_n\ \omega(G_{n,k}) = 2, \chi(G_{n,k}) \geqslant \left\lceil \log^k n \right\rceil$.

Dowód. Biorąc dwie k-krotki połączone krawędzią widzimy, że żaden inny sąsiad jednej nie może być sąsiadem drugiej, zatem nie ma trójkątów. Dowód drugiej własności przeprowadzimy indukcyjnie, bazą są standardowe grafy przesunięciowe. Niech $\phi : V(G_{n,k}) \to [r]$ będzie poprawnym kolorowaniem. Dla każdej $(k-1)$-krotki $x_1 < \ldots < x_{k-1} \in \binom{[n]}{k-1}$ definiujemy za jego pomocą zbiór $\phi'(x_1 \ldots x_{k-1}) = \{\phi(x_1, \ldots, x_{k-1}, x_k) : x_k \in \{x_{k-1}+1, \ldots, n\}\}$. Pokażemy, że ϕ' jest poprawnym kolorowaniem $G_{n,k-1}$. Weźmy dowolne $(k-1)$-krotki połączone krawędzią $\{a_1 < \ldots < a_{k-1}\}$ i $\{a_2 < \ldots < a_k\}$. Mamy $\phi(a_1, \ldots, a_k) \in \phi'(a_1, \ldots, a_{k-1})$ i $\phi(a_1, \ldots, a_k) \notin \phi'(a_2, \ldots, a_k)$, bo inaczej dwa sąsiadujące wierzchołki byłyby tak samo pokolorowane. Mamy zatem $2^r \geqslant \chi(G_{n,k-1}) \geqslant \left\lceil \log^{k-1} n \right\rceil$, czyli $r \geqslant \left\lceil \log^k n \right\rceil$. □

I.2.8 Liczba chromatyczna a liczba kolorująca

Liczba chromatyczna a liczba kolorująca grafów.

Definicja. Definiujemy liczbę kolorującą grafu G jako najmniejszą taką liczbę k, że istnieje porządek liniowy $v_1, \ldots, v_n$ na wierzchołkach G taki, że dla każdego $i \in [n]$ jest $|N_G(v_i) \cap \{v_1, \ldots, v_{i-1}\}| < k$. Pisząc znaczkami:

$$\operatorname{col}(G) = \min_{\sigma \in S_n} \max_{i \in [n]} \left| N_G\left(v_{\sigma(i)}\right) \cap \left\{v_{\sigma(1)}, \ldots, v_{\sigma(i-1)}\right\} \right| + 1.$$

Jeśli $\operatorname{col}(G) = k$, to w porządku σ o tym świadczącym wierzchołki mają mniej niż k sąsiadów wcześniejszych od nich w σ, więc zachłanny algorytm kolorujący świadczy o tym, że $\chi(G) \leqslant k = \operatorname{col}(G)$. Do tego zachodzi następujące

Twierdzenie. Zachodzi $\operatorname{col}(G) = \max\{\delta(H) : H \subseteq G\} + 1$.

Dowód. ($\leqslant$) Ustawiamy wierzchołki od końca: niech v_n będzie wierzchołkiem o minimalnym stopniu w G, v_i definiujemy jako wierzchołek o minimalnym stopniu w $G - \{v_{i+1}, \ldots, v_n\}$. Mamy $|N_G(v_i) \cap \{v_1, \ldots, v_{i-1}\}| = \delta(G - \{v_{i+1}, \ldots, v_n\}) \leqslant \max\{\delta(H) : H \subseteq G\}$, co kończy dowód.

($\geqslant$) Wybierzmy $H_0 \subseteq G$ taki, że $\delta(H_0) = \max\{\delta(H) : H \subseteq G\}$. Dla każdego liniowego porządku $v_1, \ldots, v_n$ istnieje pewien wierzchołek v_j, który jest ostatnim w tym porządku wierzchołkiem H_0. Mamy $|N_G(v_j) \cap \{v_1, \ldots, v_{j-1}\}| \geqslant \delta(H_0) = \max\{\delta(H) : H \subseteq G\}$, co kończy dowód. □

Wartość $\max\{\delta(H) : H \subseteq G\}$ nazywamy degeneracją grafu G. Degenerację i liczbę kolorującą G jesteśmy w stanie wyznaczać w czasie wielomianowym (a nawet liniowym) od rozmiaru G, natomiast wyznaczenie $\chi(G)$ jest NP-zupełne (nawet stwierdzenie 3-kolorowalności jest NP-zupełne). Liczbę kolorującą można więc traktować jako przybliżenie liczby chromatycznej. Byłoby miło, gdyby zachodziło $\operatorname{col}(G) \leqslant f(\chi(G))$ dla pewnej funkcji f. Nie jest to jednak prawda: dla pełnego grafu dwudzielnego $K_{n,n}$ liczba chromatyczna to 2, natomiast $K_{n,n}$ ma minimalny stopień n, więc $\operatorname{col}(G) \geqslant n+1$ (a nawet zachodzi równość). Zatem liczbę chromatyczną i kolorującą można odseparować dowolnie bardzo.

I.2.9 Przepływy

Przepływy w sieciach. Twierdzenie o maksymalnym przepływie i minimalnym przekroju.
Wypowiedź i dowód.

Definicja. Siecią przepływową nazywamy piątkę (V, E, s, t, c), gdzie:

- (V, E) jest grafem skierowanym (czyli $E \subseteq V \times V$).
- $s, t \in V$ są wyszczególnionymi wierzchołkami. s nazywamy źródłem – może być tylko początkiem krawędzi. t nazywamy ujściem – może być tylko końcem krawędzi.
- c jest funkcją $c : E \to \mathbb{N}_+$ zwaną funkcją przepustowości.

Definicja. Dla zadanej sieci przepływowej (V, E, s, t, c) przepływem całkowitoliczbowym nazywamy funkcję $f : E \to \mathbb{N}$ spełniającą warunki:

- warunek przepustowości: dla każdej krawędzi $(u, v) \in E$ jest $f(u, v) \leqslant c(u, v)$.
- warunek zachowania przepływu (prawo Kirchoffa): dla każdego wierzchołka $v \in V \setminus \{s, t\}$ jest

$$\sum_{u:(u,v)\in E} f(u, v) = \sum_{w:(v,w)\in E} f(v, w).$$

Wartością przepływu nazywamy $\mathrm{val}(f) = \sum_{v:(s,v)\in E} f(s, v)$. Przepływ jest maksymalny, jeśli jego wartość jest największa możliwa.

Definicja. Przekrojem w sieci przepływowej (V, E, s, t, c) nazywamy każdą parę (S, T), gdzie $S, T \subset V$ stanowią podział V oraz $s \in S, t \in T$. Przepustowość przekroju definiujemy jako

$$c(S, T) = \sum_{\substack{(u,v)\in E:\\ u\in S, v\in T}} c(u, v),$$

a dla zadanego przepływu f przepływem przez przekrój nazywamy wartość

$$f(S, T) = \sum_{\substack{(u,v)\in E:\\ u\in s, v\in T}} f(u, v) - \sum_{\substack{(v,u)\in E:\\ u\in s, v\in T}} f(v, u).$$

Lemat. Niech f będzie dowolną funkcją przepływu w sieci (V, E, s, t, c). Dla każdego przekroju (S, T) zachodzi: $f(S, T) \leqslant c(S, T)$ oraz $\mathrm{val}(f) = f(S, T)$ (w szczególności: przepływ każdego przekroju jest taki sam).

Dowód. Pierwsza własność wynika wprost z definicji przepustowości i przepływu. Drugiej własności dowodzimy indukcyjnie po liczbie wierzchołków S: dla $|S| = 1$ mamy $S = \{s\}$, wtedy teza zachodzi z definicji $\mathrm{val}(f)$. Rozważmy przekrój (S, T) z $|S| \geqslant 2$. Niech $x \in S$, $x \neq s$. $(S', T') = (S \setminus \{x\}, T \cup \{x\})$ jest przekrojem o mniejszym pierwszym zbiorze, indukcyjnie $\mathrm{val}(f) = f(S', T')$. Jest $S' \cup T = V \setminus \{x\}$, więc mamy

$$\begin{aligned} f(S', T') - f(S, T) &= \sum_{\substack{(u_1,x)\in E\\ u_1\in S'}} f(u_1, x) - \sum_{\substack{(x,v_1)\in E\\ v_1\in S'}} f(x, v_1) \\ &+ \sum_{\substack{(u_2,x)\in E\\ u_2\in T}} f(u_2, x) - \sum_{\substack{(x,v_2)\in E\\ v_2\in T}} f(x, v_2) \\ &= \sum_{u:(u,x)\in E} f(u, x) - \sum_{v:(x,v)\in E} f(x, v) = 0, \end{aligned}$$

co kończy dowód. □

Definicja. Przekrój (S', T') w sieci przepływowej nazywamy minimalnym, jeśli zachodzi

$$c(S', T') = \min \{c(S, T) : (S, T) \text{ jest przekrojem}\}.$$

Przepustowość minimalnego przekroju ogranicza z góry wartość funkcji przepływu, bo $c(S, T) \geqslant f(S, T) = \mathrm{val}(f)$.

Definicja. Niech $\mathbb{S}$ będzie siecią przepływową, a f przepływem w niej. Ścieżką powiększającą przepływ f w sieci $\mathbb{S}$ nazywamy ciąg $v_1, v_2, \ldots, v_k$, spełniający dla każdego $i \in [k-1]$ warunek: albo $(v_i, v_{i+1}) \in E$ oraz $f(v_i, v_{i+1}) < c(v_i, v_{i+1})$ (krawędź zgodna ze ścieżką), albo $(v_{i+1}, v_i) \in E$ oraz $f(v_{i+1}, v_i) > 0$ (krawędź przeciwna do ścieżki)

Jeśli istnieje ścieżka powiększająca od s do t, to przepływ f można powiększyć o pewną wartość val $\geqslant 1$, zwiększając o nią przepływ na krawędziach zgodnych ze ścieżką i zmniejszając na przeciwnych do ścieżki.

Twierdzenie (Ford, Fulkerson 1962)**.** Niech $\mathbb{S} = (V, E, s, t, c)$ będzie siecią przepływową a f przepływem w niej. Następujące warunki są równoważne:

- f jest przepływem maksymalnym.
- w sieci nie istnieje ścieżka powiększająca od s do t.
- dla $S = \{v \in V : \text{ istnieje ścieżka powiększająca od } s \text{ do } v\}$ i $T = V \setminus S$ para (S, T) jest przepływem o $f(S, T) = c(S, T)$.

Dowód. ($1 \implies 2$) Istnienie ścieżki powiększającej od s do t gwarantuje, że można powiększyć przepływ.

($2 \implies 3$) Mamy $t \notin S$ oraz $s \in S$ (ścieżka powiększająca może być trywialna), więc (S, T) jest przekrojem. Dla każdej krawędzi $(u, v) \in E$ takiej, że $u \in S, v \in T$ zachodzi $f(u, v) = c(u, v)$, bo inaczej ścieżkę powiększająca od s do u można rozszerzyć o tę krawędź, co przeczy $v \notin S$. Z tych samych powodów dla każdej krawędzi $(v, u) \in E$ takiej, że $u \in S, v \in T$ zachodzi $f(v, u) = 0$. Zatem z definicji przepływu przez przekrój i przepustowości przepływu mamy $f(S, T) = c(S, T)$.

($3 \implies 1$) Mamy $\text{val}(f) = f(S, T) \leqslant c(S, T)$, więc równość zamiast nierówności gwarantuje, że przepływ jest maksymalny. □

I.3 Metody Algebraiczne Informatyki

I.3.1 Grupy

Grupy, homomorfizmy, kongruencje.

- definicja półgrupy, monoidu, grupy z przykładami
- abelowość - przykłady (negatywne i pozytywne)
- generator grupy, grupy cykliczne
- rząd grupy i rząd elementu grupy
- podgrupy i warstwy podgrupy w grupie, indeks podgrupy
- Twierdzenie Lagrange'a dla grup (bez dowodu)
- podgrupy normalne (definicja i nietrywialny przykład dla grupy nieabelowej)
- kongruencje w grupach
- homomorfizm, epimorfizm, monomorfizm, izomorfizm, endomorfizm, automorfizm grup
- jądro, obraz homomorfizmu - podstawowe własności

Definicja. Półgrupą nazywamy parę $(G, \cdot)$, gdzie G jest zbiorem a $\cdot : G \times G \to G$ jest działaniem łącznym, czyli $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ dla $a, b, c \in G$. Monoidem nazywamy półgrupę posiadającą element neutralny e, czyli taki, że $e \cdot a = a \cdot e = a$ dla każdego $a \in G$. Grupą nazywamy monoid, w którym każdy element ma element odwrotny, czyli dla każdego $a \in G$ istnieje $a^{-1} \in G$ takie, że $a \cdot a^{-1} = a^{-1} \cdot a = e$.

- półgrupa: $(\mathbb{Z}_{>0}, +)$,
- monoid: $(\mathbb{Z}_{>0}, \cdot)$, słowa nad zadanym alfabetem z konkatenacją,
- grupa: $(\mathbb{Z}, +), (\mathbb{R} \setminus \{0\}, \cdot)$.

Definicja. Grupa $(G, \cdot)$ jest abelowa (przemienna), gdy $a \cdot b = b \cdot a$ dla wszystkich $a, b \in G$. Grupy abelowe to na przykład $(\mathbb{Z}, +)$ czy $(\mathbb{R} \setminus \{0\}, \cdot)$. Nieabelowe grupy to na przykład S_3 (bo $(12)(23) \neq (23)(12)$) albo grupa macierzy nieosobliwych z mnożeniem $\mathrm{GL}_n(\mathbb{R})$.

Definicja. Podgrupą grupy G nazywamy zbiór $H \subseteq G$, który jest grupą z działaniem indukowanym z G, to znaczy $e \in H$, $a, b \in H \implies ab \in H$, $a \in H \implies a^{-1} \in H$. Równoważnie można powiedzieć, że $H \neq \emptyset$ i $a, b \in H \implies ab^{-1} \in H$. Wprowadzamy oznaczenie $H \leqslant G$.

Definicja. Niech G będzie grupą i $g \in G$. Podgrupą generowaną przez g nazywamy zbiór $\langle g \rangle = \{g^n : n \in \mathbb{Z}\}$, który jest grupą z działaniem indukowanym z G. G jest cykliczna, jeśli istnieje takie g, że $G = \langle g \rangle$. g nazywamy wtedy generatorem grupy. Każda grupa cykliczna jest abelowa. Wszystkie grupy cykliczne o ustalonej liczbie elementów są izomorficzne.

Definicja. Rzędem grupy G nazywamy liczbę jej elementów $|G|$. Rzędem elementu $g \in G$ nazywamy rząd grupy $\langle g \rangle$. Równoważnie jest to najmniejsze takie $n \in \mathbb{N}$, że $g^n = e$ (jeśli takie n nie istnieje, to rząd jest nieskończony). Wtedy mamy $\langle g \rangle = \{e, g, g^2, \ldots, g^{n-1}\}$.

Definicja. Lewostronnymi warstwami podgrupy $H \leqslant G$ nazywamy rodzinę zbiorów $\{aH : a \in G\} = \{\{ah : h \in H\} : a \in G\}$. Analogicznie możemy zdefiniować prawostronne warstwy $\{Ha : a \in G\} = \{\{ha : h \in H\} : a \in G\}$. Zauważmy, że warstwy mogą się pokrywać, to znaczy może zachodzić $aH = bH$ dla $a, b \in G$. Indeks podgrupy H w grupie G, oznaczany $[G : H]$, to liczba lewostronnych (równoważnie: prawostronnych) warstw H w G.

Twierdzenie (Lagrange)**.** Niech G będzie skończoną grupą, $H \leqslant G$. Wtedy $|G| = |H| [G : H]$. W szczególności $|H| \mid |G|$.

Definicja. Niech $H \leqslant G$. H nazywamy podgrupą normalną G (co oznaczamy $H \lhd G$), jeśli dla każdego $a \in G$ mamy $aH = Ha$. Inaczej mówiąc, dla każdego $a \in G$ i $h \in H$ zachodzi $aha^{-1} = h'$ dla pewnego $h' \in H$.

W szczególności każda podgrupa grupy abelowej jest normalna. W grupie nieabelowej S_n (dla $n \geqslant 3$) mamy podgrupę normalną permutacji parzystych A_n. Jest to podgrupa normalna, bo znak jest multiplikatywny, czyli $\operatorname{sgn}\left(aha^{-1}\right) = \operatorname{sgn}(a)^2 \operatorname{sgn}(h) = 1$ dla $h \in A_n$ i $a \in S_n$.

Definicja. Niech G będzie grupą. Kongruencją w G nazywamy relację równoważności $\equiv$ na elementach G zgodną z działaniem, to znaczy jeśli $a \equiv a'$ i $b \equiv b'$, to $ab \equiv a'b'$.

Jeśli $\equiv$ jest kongruencją na elementach G, to możemy zdefiniować grupę ilorazową $G/_{\equiv} = \{[a]_{\equiv} : a \in G\}$ z działaniem $[a]_{\equiv}[b]_{\equiv} = [ab]_{\equiv}$. Jest ono poprawnie zdefiniowane, bo jeśli weźmiemy $a' \in [a]_{\equiv}$ i $b' \in [b]_{\equiv}$, to mamy $[a'b']_{\equiv} = [ab]_{\equiv}$. Jeśli H jest podgrupą normalną G, to $a \equiv_H b \iff ab^{-1} \in H$ jest kongruencją, co daje nam grupę ilorazową podgrupy H zdefiniowaną jako $G/_H := G/_{\equiv_H}$.

Definicja. Niech $(G, \cdot), (G', \star)$ będą grupami. $h : G \to G'$ nazywamy homomorfizmem, jeśli $h(a \cdot b) = h(a) \star h(b)$. Jeśli h jest:

- surjekcją, to nazywamy go epimorfizmem.
- iniekcją, to nazywamy go monomorfizmem.
- bijekcją, to nazywamy go izomorfizmem.
- odwzorowaniem $G \to G$ (czyli $G' = G$), to nazywamy go endomorfizmem.
- endomorfizmem i izomorfizmem, to nazywamy go automorfizmem.

Propozycja. Niech G, G' będą grupami o elementach neutralnych e, e', $h : G \to G'$ homomorfizmem. Zachodzi $h(e) = e'$ i $h\left(a^{-1}\right) = h(a)^{-1}$ dla $a \in G$.

Dowód. Mamy $h(e) = h(e \cdot e) = h(e)\, h(e)$, czyli $e' = h(e)$. Mamy $e' = h(e) = h\left(aa^{-1}\right) = h(a)\, h\left(a^{-1}\right)$. □

Definicja. Obrazem homomorfizmu $h : G \to G'$ nazywamy zbiór $\operatorname{im} h := h(G) \subseteq G'$. Jądrem h nazywamy zbiór $\ker h := h^{-1}(e') = \{g \in G : h(g) = e'\}$, gdzie $e' \in G'$ jest elementem neutralnym.

Obraz homomorfizmu h jest podgrupą G'. Jądro homomorfizmu jest podgrupą normalną G. $\ker h = \{0\}$ wtedy i tylko wtedy, gdy h jest monomorfizmem. Zachodzi tak zwane (pierwsze) twierdzenie o izomorfizmie: istnieje izomorfizm między $G/_{\ker h}$ a $\operatorname{im} h$.

I.3.2 Pierścienie i ciała

Ciała, pierścienie i wielomiany.

- pierścień – definicja i przykłady
- ciało – definicja i przykłady (negatywne i pozytywne, skończone i nieskończone)
- grupa addytywna i multiplikatywna w ciele
- pierścień wielomianów nad ciałem – wielomiany jako ciągi, stopień wielomianu, działania na wielomianach (dodawanie i mnożenie jako pierścień)
- pierwiastki wielomianu, Twierdzenie Bézout'a (bez dowodu)
- Twierdzenie interpolacyjne Lagrange'a (bez dowodu)

Definicja. Pierścieniem nazywamy układ $(R, +, \cdot)$ złożony ze zbioru R i dwóch działań (dodawania i mnożenia), takich, że $(R, +)$ jest grupą abelową (w której element neutralny oznaczamy 0), mnożenie jest łączne i dodawanie jest obustronnie rozdzielne względem mnożenia: $x(y+z) = xy + xz$ i $(x+y)z = xz + yz$. Pierścień nazywamy pierścieniem z jedynką, jeśli istnieje element neutralny mnożenia oznaczany 1. Pierścień jest przemienny, jeśli mnożenie jest przemienne.

Przykłady pierścieni: $\mathbb{Z}$, $\mathbb{Z}/n\mathbb{Z}$ (reszty modulo n), macierze kwadratowe $M_{n\times n}(\mathbb{C})$. Pierścień macierzy nie jest pierścieniem przemiennym.

Definicja. Ciałem nazywamy pierścień przemienny z jedynką $\mathbb{F} = (F, +, \cdot)$, w którym każdy element poza 0 posiada element odwrotny względem mnożenia.

Ciałem jest na przykład $\mathbb{Z}/p\mathbb{Z}$, gdzie p jest pierwsze (istnienie odwrotności wynika z rozszerzonego algorytmu Euklidesa), ale $\mathbb{Z}/n\mathbb{Z}$ dla ogólnego n już nie jest ciałem – jeśli $n = ab$ dla $a, b > 1$, to w takim pierścieniu $a \cdot b = n = 0$. Gdyby a miało odwrotność, to byłoby $b = 0$, czyli $n \mid b$, sprzeczność.

Ciałem jest $\mathbb{Q}$ lub $\mathbb{R}$, ale $\mathbb{Z}$ już nie – nie umiemy odwracać.

Definicja. Grupą addytywną ciała $\mathbb{F}$ nazywamy grupę $(\mathbb{F}, +)$. Grupą multiplikatywną nazywamy grupę $(\mathbb{F}^*, \cdot)$, gdzie $\mathbb{F}^* = \mathbb{F} \setminus \{0\}$.

Definicja. Niech $\mathbb{F}$ będzie ciałem. Wielomianem nad $\mathbb{F}$ nazywamy dowolny ciąg $A = (a_0, a_1, \ldots)$, który od pewnego momentu jest zerowy, czyli istnieje takie $n_0 \in \mathbb{N}$, ze dla $n > n_0$ jest $a_n = 0$. Takie wielomiany tworzą zbiór oznaczany $\mathbb{F}[X]$. Stopień wielomianu definiujemy jako $\deg A = \max\{n \in \mathbb{N} : a_n \neq 0\}$. Jeśli $A = (0, 0, \ldots)$, to przyjmujemy $\deg A = -\infty$.

Wprowadzamy działania na wielomianach. Dodawanie definiujemy współczynnik po współczynniku:

$$(a_0, a_1, \ldots) + (b_0, b_1, \ldots) = (a_0 + b_0, a_1 + b_1, \ldots).$$

Mnożenie definiujemy przez splot:

$$(a_0, a_1, \ldots) \cdot (b_0, b_1, \ldots) = (c_0, c_1, \ldots),$$

gdzie:

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

Wielomiany zazwyczaj zapisujemy jako $A(X) = \sum_{k=0}^{n} a_k X^k$, gdzie X jest zmienną formalną. W takiej formie łatwo jest zapisać mnożenie jako $\sum_{k=0}^{n} a_k X^k \cdot \sum_{\ell=0}^{m} b_\ell X^\ell = \sum_{k=0}^{n+m} \sum_{i=0}^{k} a_i b_{k-i} X^k$. Sprawdzenie, że tak zdefiniowane działania zadają strukturę pierścienia przemiennego z jedynką jest natychmiastowe.

Każdy wielomian $A(X) = \sum_{k=0}^{n} a_k X^k$ wyznacza funkcję wielomianową $A : \mathbb{F} \ni x \mapsto \sum_{k=0}^{n} a_k x^k$. Różne wielomiany mogą produkować tę samą funkcję, na przykład x i x^3 nad $\mathbb{Z}/3\mathbb{Z}$.

Definicja. $a \in \mathbb{F}$ nazywamy pierwiastkiem wielomianu $A(X)$, jeśli istnieje taki wielomian $B(X)$, że $(X - a)B(X) = A(X)$.

Twierdzenie (Bézout)**.** $a \in \mathbb{F}$ jest pierwiastkiem wielomianu $A(X)$ wtedy i tylko wtedy, gdy $A(a) = 0$.

Twierdzenie (Lagrange)**.** Niech $a_0, \ldots, a_n \in \mathbb{F}$ będą różnymi elementami ciała. Ustalmy dowolne elementy $b_0, \ldots, b_n \in \mathbb{F}$. Wtedy istnieje dokładnie jeden wielomian $W(X)$ nad $\mathbb{F}$ stopnia co najwyżej n, dla którego zachodzi $W(a_i) = b_i$. Może on być zadany wzorem

$$W(X) = \sum_{i=0}^{n} b_i \frac{\prod_{j\neq i} X - a_j}{\prod_{j\neq i} a_i - a_j}.$$

I.3.3 Ciała skończone

Ciała skończone, ich przykłady.

- charakterystyka i rząd ciała skończonego (możliwe wartości, związek między nimi, ze znajomością dowodu)
- kiedy grupa multiplikatywna ciała skończonego jest cykliczna? (bez dowodu)
- Małe Twierdzenie Fermata (ze znajomością dowodu), Twierdzenie Eulera (bez dowodu)
- Chińskie twierdzenie o resztach (bez dowodu)
- Tożsamość Bézout'a (bez dowodu) – jaki jest związek z rozszerzonym algorytmem Euklidesa
- Zastosowanie rozszerzonego algorytmu Euklidesa w praktyce (bez dokładnego opisu algorytmu) w rozwiązywaniu liniowego równania modularnego i obliczanie odwrotności modularnych

Definicja. Charakterystyką ciała $\mathbb{F}$ nazywamy najmniejszą taką liczbę $c \in \mathbb{N}_+$, że $c\cdot 1 := \underbrace{1 + \ldots + 1}_{c \text{ razy}} = 0$.

Jeśli taka liczba nie istnieje, to mówimy o charakterystyce równej 0.

Definicja. Rzędem ciała nazywamy liczbę jego elementów.

Propozycja. Jeśli ciało $\mathbb{F}$ ma charakterystykę 0, to ma nieskończony rząd.

Dowód. Elementy $1, 1+1, 1+1+1, \ldots$ są parami różne. Istotnie, gdyby było $n \cdot 1 = m \cdot 1$ dla $n > m$, to $(n-m) \cdot 1 = 0$ daje sprzeczność z charakterystyką 0. □

Propozycja. Jeśli ciało $\mathbb{F}$ ma charakterystykę $n > 0$, to n jest liczbą pierwszą.

Dowód. Załóżmy nie wprost, że $n = ab$ dla $a, b > 1$. Wtedy mamy

$$0 = n \cdot 1 = \underbrace{1 + \ldots + 1}_{n \text{ razy}} = (\underbrace{1 + \ldots + 1}_{a \text{ razy}})(\underbrace{1 + \ldots + 1}_{b \text{ razy}}) = (a \cdot 1)(b \cdot 1).$$

Z tego wynika, $a \cdot 1 = 0$ lub $b \cdot 1 = 0$, co daje sprzeczność z minimalnością n. □

Propozycja. Jeśli ciało $\mathbb{F}$ ma skończony rząd i charakterystykę p, to $|\mathbb{F}| = p^k$ dla pewnego $k \in \mathbb{N}_+$.

Dowód. Możemy patrzeć na $\mathbb{F}$ jak na przestrzeń wektorową nad ciałem $\mathbb{Z}/p\mathbb{Z}$, w której mnożenie przez skalar (element $\mathbb{Z}/p\mathbb{Z}$) zadane jest przez $a \cdot x = \underbrace{x + \ldots + x}_{a \text{ razy}}$. Poprawność tej definicji wynika z tego, że jeśli $a = a' + tp$, to $a' \cdot x = a \cdot x + tp \cdot x = a \cdot x$. Istnieje zatem baza $b_1, \ldots, b_k$ przestrzeni $\mathbb{F}$ nad $\mathbb{Z}/p\mathbb{Z}$. Każdy element $\mathbb{F}$ ma jednoznaczny zapis $a_1 b_1 + \ldots + a_k b_k$ dla $a_1, \ldots, a_k \in \mathbb{Z}_p$, co daje tezę. □

Najprostszym przykładem ciała skończonego jest $\mathbb{Z}/p\mathbb{Z}$ dla pierwszego p, które ma rząd p. Inne ciała skończone powstają z przestrzeni wektorowej nad $\mathbb{Z}/p\mathbb{Z}$ poprzez zadanie na niej mnożenia (zazwyczaj w całkiem nietrywialny sposób). Jeśli h jest nierozkładalnym wielomianem nad $\mathbb{Z}/p\mathbb{Z}$ stopnia n, to zbiór wielomianów nad $\mathbb{Z}/p\mathbb{Z}$ stopnia mniejszego niż n z mnożeniem i dodawaniem modulo h tworzy ciało skończone o p^n elementach.

Twierdzenie. Niech $\mathbb{F}$ będzie ciałem o skończonym rzędzie. Grupa multiplikatywna $(\mathbb{F}^*, \cdot)$ jest cykliczna.

Twierdzenie (Małe Twierdzenie Fermata)**.** Niech p będzie liczbą pierwszą. Wtedy dla $a \in (\mathbb{Z}/p\mathbb{Z})^*$ zachodzi $a^{p-1} = 1$.

Dowód. Zauważmy, że odwzorowanie $(\mathbb{Z}/p\mathbb{Z})^* \ni x \mapsto ax \in (\mathbb{Z}/p\mathbb{Z})^*$ jest bijekcją. Istotnie, jeśli $ax = ax'$ dla $x, x' \in (\mathbb{Z}/p\mathbb{Z})^*$, to mnożąc przez odwrotność a dostajemy $x = x'$. Zatem mamy iniektywność, co wobec równoliczności dziedziny i przeciwdziedziny daje bijektywność. Wobec tego

$$\prod_{i=1}^{p-1} i = \prod_{i=1}^{p-1} ai = a^{p-1} \prod_{i=1}^{p-1} i.$$

Mamy $\prod_{i=1}^{p-1} i \neq 0$, więc mnożąc obustronnie przez odwrotność dostajemy tezę. □

Twierdzenie (Euler)**.** Niech $n \in \mathbb{N}_+$. Niech a będzie względnie pierwsze z n. Wtedy $a^{\varphi(n)} \equiv 1 \pmod n$, gdzie $\varphi(n)$ oznacza funkcję Eulera (liczbę liczb względnie pierwszych z n, mniejszych od n).

Twierdzenie (Chińskie Twierdzenie o resztach)**.** Niech $m, n \in \mathbb{N}_+$ będą względnie pierwszymi liczbami naturalnymi (czyli nie mają wspólnych dzielników, $\gcd(m, n) = 1$). Wtedy odwzorowanie

$$(\mathbb{Z}/mn\mathbb{Z}) \ni x \mapsto (x \bmod m, x \bmod n) \in (\mathbb{Z}/m\mathbb{Z}) \times (\mathbb{Z}/n\mathbb{Z})$$

jest bijekcją. Inaczej mówiąc, układ równań

$$\begin{cases} x \equiv a \pmod m \\ x \equiv b \pmod n \end{cases}$$

ma dokładnie jedno rozwiązanie modulo mn.

Twierdzenie (Tożsamość Bézout'a)**.** Niech $a, b \in \mathbb{Z}$. Istnieją takie liczby całkowite $x, y \in \mathbb{Z}$, że

$$ax + by = \gcd(a, b).$$

Aby wyznaczyć x, y stosujemy rozszerzony algorytm Euklidesa: aby znaleźć $\gcd(a, b)$ rozpisujemy $a = qb+r$ i wywołujemy się rekurencyjnie na b, r. Mamy $\gcd(a, b) = \gcd(b, r)$. Jeśli znajdziemy współczynniki x', y' takie, że $bx' + ry' = \gcd(b, r)$, to znalezienie odpowiednich współczynników dla a, b polega na podstawieniu $x = y'$, $y = x' - q$. Algorytm Euklidesa pozwala nam efektywnie wyznaczać odwrotność a modulo n dla a względnie pierwszego z n. Tożsamość Bézout'a daje nam bowiem $ax + ny = 1$, czyli $x \equiv a^{-1} \pmod n$.

Jeśli chcemy rozwiązać równanie modularne $ax \equiv b \pmod n$, to zaczynamy od wyliczenia $d = \gcd(a, n)$. Jeśli $d \nmid b$, to równanie nie ma rozwiązania (bo gdyby miało, to byłoby $b = ax + tn$ dla pewnego t). Jeśli $d \mid b$, to wystarczy rozwiązać równanie $\frac{a}{d}x \equiv \frac{b}{d} \pmod{\frac{n}{d}}$. Mamy $\gcd\left(\frac{a}{d}, \frac{n}{d}\right) = 1$ i możemy policzyć odwrotność $\frac{a}{d}$.

I.3.4 Liczby zespolone

Ciało liczb zespolonych.

- ciało liczb zespolonych jako rozszerzenie ciała liczb rzeczywistych
- współrzędne kartezjańskie i biegunowe (część rzeczywista i urojona, moduł i argument, postać iloczynu w układzie biegunowym)
- sprzężenie zespolone
- postać trygonometryczna i wykładnicza
- wzór de Moivre'a, pierwiastki n-tego stopnia z jedynki - co to znaczy, że tworzą podgrupę cykliczną
- logarytm naturalny z liczby zespolonej
- Zasadnicze Twierdzenie Algebry (bez dowodu)

Definicja. Ciałem liczb zespolonych nazywamy zbiór $\mathbb{C} = \{a+bi : a, b \in \mathbb{R}\}$, gdzie $i^2 = -1$. Dodawanie definiujemy wzorem $(a + bi) + (c + di) = (a + c) + (b + d)i$, natomiast mnożenie $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$. Liczby rzeczywiste utożsamiamy z liczbami zespolonymi postaci $a + 0i$. Dlatego $\mathbb{R} \subseteq \mathbb{C}$. Mówimy, że $\mathbb{C}$ jest rozszerzeniem ciała liczb rzeczywistych.

Definicja. Zapis $z = x + iy$ nazywamy postacią kartezjańską liczby zespolonej. Definiujemy część rzeczywistą i urojoną z jako $\operatorname{Re}(z) = x$ i $\operatorname{Im}(z) = y$.

Definicja. Płaszczyznę liczb zespolonych utożsamiamy z $\mathbb{R}^2$. Liczbie $x+iy$ odpowiada punkt (x, y). Modułem liczby $z = x+iy$ nazywamy liczbę $|z| = \sqrt{x^2 + y^2}$. Argumentem liczby zespolonej nazywamy taki

kąt φ, że $z = |z|\,(\cos\varphi + i\sin\varphi)$. Argument nie jest wyznaczony jednoznacznie. Jeżeli φ jest argumentem, to $\varphi + 2k\pi$ również. Dlatego definiuje się argument główny $\mathrm{Arg}(z) \in (-\pi, \pi]$.

Zapis $z = r\,(\cos\varphi + i\sin\varphi)$, gdzie $r = |z|$ i $\varphi = \mathrm{Arg}\,(z)$, nazywa się postacią biegunową (trygonometryczną) liczby zespolonej. Mnożenie w postaci biegunowej wygląda następująco:

$$\begin{aligned}&r_1(\cos\varphi_1 + i\sin\varphi_1)\cdot r_2(\cos\varphi_2 + i\sin\varphi_2) =\\ &r_1r_2\,(\cos\varphi_1\cos\varphi_2 - \sin\varphi_1\sin\varphi_2 + i\,(\cos\varphi_1\sin\varphi_2 + \sin\varphi_1\cos\varphi_2)) =\\ &r_1r_2(\cos(\varphi_1+\varphi_2) + i\sin(\varphi_1+\varphi_2)).\end{aligned}$$

Zatem moduły się mnożą, a argumenty dodają.

Definicja. Dla $z = x + iy$ definiujemy sprzężenie $\overline{z} = x - iy$.

Propozycja. Zachodzi

$$\begin{aligned}|zw| &= |z||w|,\\ |z| = 0 &\iff z = 0,\\ z\overline{z} &= |z|^2,\\ \overline{z+w} &= \overline{z} + \overline{w},\\ \overline{zw} &= \overline{z}\,\overline{w},\\ \frac{1}{z} = \frac{\overline{z}}{|z|^2} &\quad \text{dla } z \neq 0.\end{aligned}$$

Korzystając ze wzoru $e^{ix} = \cos x + i\sin x$ (wynika on z definicji tych funkcji jako szeregów potęgowych) dostajemy $z = r(\cos\varphi + i\sin\varphi) = re^{i\varphi}$. Jest to postać wykładnicza liczby zespolonej.

Twierdzenie (de Moivre)**.** Dla $z = r(\cos\varphi + i\sin\varphi)$ oraz $n \in \mathbb{Z}$ zachodzi $z^n = r^n(\cos(n\varphi) + i\sin(n\varphi))$. Równoważnie $(re^{i\varphi})^n = r^ne^{in\varphi}$.

Szukamy rozwiązań (pierwiastków) równania $z^n = 1$. Ponieważ $1 = e^{2k\pi i}$, otrzymujemy $z_k = e^{\frac{2k\pi i}{n}}$ dla $k = 0, \ldots, n-1$. Istnieje więc dokładnie n takich pierwiastków. Tworzą one zbiór:

$$\mu_n = \{1, \zeta, \zeta^2, \ldots, \zeta^{n-1}\},$$

gdzie $\zeta = e^{\frac{2\pi i}{n}}$. ζ nazywamy pierwotnym pierwiastkiem z jedynki stopnia n. Zbiór μ_n z mnożeniem jest grupą cykliczną, której generatorem jest ζ.

Chcemy rozwiązać $e^w = z$. Niech $z = re^{i\varphi}$. Wtedy $w = \ln r + i(\varphi + 2k\pi)$. To prowadzi nas do definicji logarytmu z liczby zespolonej $\log z = \{\ln|z| + i(\mathrm{Arg}\,z + 2k\pi) : k \in \mathbb{Z}\}$. Logarytmem głównym nazywamy wartość

$$\mathrm{Log}(z) = \ln|z| + i\,\mathrm{Arg}(z).$$

Twierdzenie. Każdy niezerowy wielomian $P(X) = a_nX^n + \ldots + a_0$ nad $\mathbb{C}$ stopnia $n \geqslant 1$ posiada przynajmniej jeden pierwiastek zespolony.

W konsekwencji każdy wielomian stopnia n nad $\mathbb{C}$ rozkłada się na iloczyn czynników liniowych $P(X) = a_n(X - \lambda_1)\cdots(X - \lambda_n)$, gdzie pierwiastki liczymy z krotnościami.

I.3.5 Wyznacznik macierzy

Wyznacznik, minory i rozwinięcie Laplace'a.

- definicje wyznacznika: aksjomatyczna i permutacyjna
- definicja minora macierzy
- wzór na rozwinięcie Laplace'a
- macierz dopełnień algebraicznych, macierz dołączona i związek z macierzą odwrotną (bez dowodu)
- rozwiązywanie układów równań liniowych metodą wyznaczników, wzory Cramera (bez dowodu)

Definicja. Niech $M_{n\times n}(\mathbb{K})$ będzie zbiorem macierzy kwadratowych nad ciałem $\mathbb{K}$. Wyznacznikiem nazywamy funkcję $\det : M_{n\times n}(\mathbb{K}) \to \mathbb{K}$ spełniającą następujące własności:

1. det jest n-liniowy, czyli liniowy względem każdej kolumny osobno. Zatem

$$\det([v_1, \ldots, \alpha v_i + \beta w_i, \ldots, v_n]) = \alpha \det([v_1, \ldots, v_i, \ldots, v_n]) + \beta \det([v_1, \ldots, w_i, \ldots, v_n]).$$

2. det jest alternujący, czyli jeśli dwie kolumny macierzy A są identyczne, to $\det(A) = 0$.
3. wyznacznik macierzy jednostkowej wynosi $\det(I_n) = 1$.

Z tych własności wynikają fakty:

- zamiana dwóch kolumn zmienia znak wyznacznika,
- dodanie wielokrotności jednej kolumny do drugiej nie zmienia wyznacznika,
- macierz z zerową kolumną ma wyznacznik równy zero.

Twierdzenie. Jeśli wyznacznik istnieje, to jest wyznaczony jednoznacznie.

Dowód. Niech $A \in M_{n\times n}(\mathbb{K})$. Jeśli kolumny A są liniowo zależne, to $\det(A) = 0$, bo można za pomocą dodawania wielokrotności innych kolumn sprowadzić A do macierzy z zerową kolumną. Zalej zakładamy, że kolumny A są liniowo niezależne.

Wiemy, że $\det(I_n) = \det([e_1, \ldots, e_n]) = 1$. Kolumny A są generowane przez $e_1, \ldots, e_n$, ale przez $e_2, \ldots, e_n$ już nie (wtedy byłyby liniowo zależne). Zatem istnieje taka kolumna $a_i = \sum_{j=1}^{n} \alpha_j e_j$, że $\alpha_1 \neq 0$. Za pomocą mnożenia przez skalar i dodawania kolumn do siebie otrzymamy wyznacznik $\det([a_i, e_2, \ldots, e_n])$. Wektory $a_i, e_2, \ldots, e_n$ dalej generują wszystkie kolumny A (bo można przedstawić e_1 jako ich kombinację liniową). Powtarzając ten argument dla kolejnych wektorów bazowych ostatecznie otrzymamy wyznacznik macierzy składającej się z kolumn macierzy A występujących w jakiejś kolejności. Zamieniając te kolumny miejscami dostaniemy wyznacznik A. □

Niech S_n oznacza zbiór wszystkich permutacji zbioru $\{1, \ldots, n\}$. Wyznacznik macierzy $A = [a_{ij}]$ zadany jest wzorem

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)},$$

gdzie $\operatorname{sgn}(\sigma)$ oznacza znak permutacji. Z tego łatwo wynika, że $\det(A) = \det\left(A^T\right)$ i wszystko co było powiedziane wyżej działa również dla wierszy.

Definicja. Minorem stopnia k macierzy A nazywamy wyznacznik dowolnej podmacierzy $k \times k$ otrzymanej poprzez wybranie k wierszy i k kolumn. Szczególnie ważny jest minor A_{ij}, czyli wyznacznik macierzy otrzymanej przez usunięcie i-tego wiersza i j-tej kolumny. Jest to minor rozmiaru $(n-1) \times (n-1)$. Dopełnieniem algebraicznym elementu a_{ij} nazywamy liczbę $C_{ij} = (-1)^{i+j} A_{ij}$.

Twierdzenie (Rozwinięcie Laplace'a)**.** Wyznacznik można rozwinąć względem dowolnego wiersza:

$$\det(A) = \sum_{j=1}^{n} a_{ij} C_{ij}.$$

Można również rozwinąć względem dowolnej kolumny:

$$\det(A) = \sum_{i=1}^{n} a_{ij} C_{ij}.$$

Te rozwinięcia wynikają z tego, że jeśli ograniczymy się do $\sigma(i) = j$, to element a_{ij} można przenieść na pozycję $(1, 1)$ za pomocą $i + j$ zamian wierszy i kolumn. Wtedy mamy $\sigma(1) = 1$ i wystarczy rozważać permutacje pozostałych elementów. Rozwinięcie Laplace'a pozwala sprowadzać obliczanie wyznaczników stopnia n do wyznaczników stopnia $n - 1$.

Definicja. Macierzą dopełnień algebraicznych macierzy $A = [a_{ij}]$ nazywamy macierz $C = [C_{ij}] = \left[(-1)^{i+j} A_{ij}\right]$. Macierzą dołączoną (adjungowaną) nazywamy macierz $\operatorname{adj}(A) = C^T$.

Zachodzi zależność $A \operatorname{adj}(A) = \operatorname{adj}(A)A = \det(A)I$. W szczególności jeżeli $\det(A) \neq 0$, to: $A^{-1} = \frac{1}{\det(A)} \operatorname{adj}(A)$.

Twierdzenie (Wzory Cramera)**.** Rozważmy równanie macierzowe (układ równań liniowych) $Ax = b$, gdzie $\det(A) \neq 0$. Niech A_i oznacza macierz otrzymaną z A przez zastąpienie i-tej kolumny wektorem b. Jedynym rozwiązaniem układu jest $x_i = \frac{\det(A_i)}{\det(A)}$. Jeśli $\det(A) = 0$, to układ może mieć wiele rozwiązań lub nie mieć żadnych.

I.3.6 Eliminacja Gaussa

Metoda eliminacji Gaussa i jej zastosowania.

- jakie operacje na macierzy nie zmieniają jej wyznacznika lub zmieniają tylko znak?
- opisz metodę eliminacji Gaussa (bez dowodu)
- zastosowania: odwracanie macierzy, rozwiązywanie układów równań liniowych, obliczanie rzędu macierzy i wyznacznika macierzy

Podczas eliminacji Gaussa wykonujemy elementarne operacje na wierszach macierzy W. Istnieją trzy podstawowe typy operacji:

1. zamiana dwóch wierszy W_i i W_j miejscami,
2. pomnożenie wiersza W_i przez niezerową stałą λ,
3. dodanie wielokrotności jednego wiersza do drugiego: $W_i \mapsto W_i + \lambda W_j$.

Pierwsza z tych operacji zmienia znak wyznacznika, druga powoduje jego przemnożenie przez λ (ale nie wpływa na zerowość), a trzecia zupełnie go nie zmienia.

Celem eliminacji Gaussa jest sprowadzenie macierzy $A = [a_{ij}] \in M_{n\times n}(\mathbb{K})$ do prostszej postaci poprzez zerowanie elementów pod diagonalą. Przeprowadzamy ją w następujący sposób:

1. Znajdujemy niezerowy element w pierwszej kolumnie. Powiedzmy, że jest to $a_{i1} \neq 0$. Jeśli takiego elementu nie ma, to macierz ma zerowy wyznacznik. Mimo to możemy kontynuować eliminację: znajdujemy pierwszą kolumnę, w której taki element jest i zamieniamy ją miejscami z pierwszą. Jeśli nie ma takiej kolumny, to kończymy procedurę.
2. Zamieniamy miejscami i-ty wiersz z pierwszym.
3. Odejmujemy wielokrotności nowego pierwszego wiersza od kolejnych wierszy w taki sposób, aby cała pierwsza kolumna się wyzerowała (poza pierwszy elementem).
4. Powtarzamy taką procedurę dla macierzy bez pierwszego wiersza i kolumny.

Taka procedura wymaga $O\left(n^3\right)$ operacji w ciele $\mathbb{K}$. Otrzymana macierz A' jest w postaci górnotrójkątnej, czyli wszystkie elementy macierzy leżące pod diagonalą są zerowe. Wyznacznik takiej macierzy to po

prostu iloczyn elementów na diagonali (wynika natychmiast z definicji wyznacznika przez permutacje). Zatem wyznacznik A jest (z dokładnością do znaku) wyznacznikiem macierzy A'.

Rzędem macierzy A nazywamy maksymalną liczbę liniowo niezależnych wierszy. Zauważmy, że eliminacja Gaussa nie zmienia rzędu macierzy (dodajemy wiersze do siebie). Zatem rząd A to rząd A', a dla macierzy górnotrójkątnej rząd to liczba niezerowych wierszy.

Aby odwrócić macierz A ($\det(A) \neq 0$) wykonujemy na niej eliminację Gaussa i powtarzamy te same operacje na macierzy I. W ten sposób dostajemy macierze A', I'. Następnie wykonujemy procedurę analogiczną do eliminacji Gaussa, ale na kolumnach A' zamiast na wierszach. Ponownie powtarzamy wszystkie operacje dla I'. Otrzymujemy macierze A'', I''. A'' jest macierzą diagonalną (ma niezerowe elementy tylko na diagonali). Wymnażając przez ich odwrotności (nie ma zer na diagonali z $\det(A'') \neq 0$) dostajemy macierz I. Powtórzenie tego samego dla I'' daje nam macierz A^{-1} odwrotną do A. Poprawność tego algorytmu wynika z faktu, że każda z wykonanych operacji może zostać przedstawiona jako mnożenie przez pewną macierz. Mamy $S_1 \dots S_k A = I$, czyli $S_1 \dots S_k I = A^{-1}$.

Rozważmy równanie macierzowe (układ równań liniowych) $Ax = b$. Ma ono jednoznaczne rozwiązanie, jeśli $\det(A) \neq 0$. Możemy wyznaczyć x odwracając A i wyliczając $x = A^{-1}b$. Możemy również przedstawić algorytm korzystający wprost z eliminacji Gaussa. Rozważamy macierz rozszerzoną $(A \mid b)$ (czyli macierz $n \times (n+1)$ powstałą przez dopisanie b do A). Wykonujemy na niej kolejne operacje eliminacji Gaussa macierzy A. Ostatecznie dostaniemy macierz $(A' \mid b')$, gdzie A' jest górnotrójkątna. Układ równań $A'x = b'$ łatwo jest rozwiązać. Jeśli $\det(A') \neq 0$, to ostatni wiersz A' ma niezerowy tylko ostatni współczynnik, co jednoznacznie wyznacza ostatni element x, bo ostatnie równanie naszego układu to $a'_{nn}x_n = b'_n$. Przedostatnie równanie to $a'_{n-1n-1}x_{n-1} + a'_{n-1n}x_n = b'_{n-1}$. Zatem znając x_n można wyznaczyć x_{n-1}. Kontynuując w ten sposób wyznaczamy cały wektor x.

W przypadku, gdy $\det(A) = 0$ sprowadzenie A do postaci górnotrójkątnej również jest pomocne. Możemy stwierdzić, czy rozważany układ równań jest sprzeczny, a jeśli nie jest, to jak wyglądają jego rozwiązania. Trzeba tylko uważać na operacje zamiany kolumn, które mogą się pojawić w eliminacji Gaussa, gdy $\det(A) = 0$. Odpowiadają one zamianie zmiennych (zamienieniu ze sobą elementów wynikowego wektora x). Aby otrzymać rozwiązanie początkowego układu równań należy na koniec odwrócić te zamiany.

I.3.7 Przestrzenie wektorowe

Przestrzenie wektorowe (liniowe).

- definicja z przykładami
- podprzestrzeń przestrzeni wektorowej
- kombinacja liniowa
- zbiór generujący i zbiór liniowo niezależny
- baza przestrzeni wektorowej i wymiar przestrzeni wektorowej
- zbiór $\mathbb{R}_n[X]$ wszystkich wielomianów stopnia nie większego niż n jako przestrzeń wektorowa. Podaj wymiar i kilka przykładowych baz (z dowodem).

Definicja. Niech $\mathbb{K}$ będzie ciałem. Przestrzenią wektorową nad ciałem $\mathbb{K}$ nazywamy zbiór V wyposażony w działanie dodawania $+ : V \times V \to V$ i mnożenie przez skalar $\cdot : \mathbb{K} \times V \to V$. $(V, +)$ jest grupą abelową o elemencie neutralnym 0. Odwrotność elementu $u \in V$ w tej grupie oznaczamy $-u$. Do tego dla wszystkich $u, v \in V$ oraz $\alpha, \beta \in \mathbb{K}$ zachodzi $\alpha(u + v) = \alpha u + \alpha v$, $(\alpha + \beta)u = \alpha u + \beta u$, $(\alpha\beta)u = \alpha(\beta u)$ i $1u = u$.

Kilka przykładów:

- $\mathbb{R}^n$ nad $\mathbb{R}$,
- $\mathbb{C}^n$ nad $\mathbb{C}$,
- zbiór macierzy $M_{m\times n}(\mathbb{R})$ nad $\mathbb{R}$,
- zbiór wielomianów $\mathbb{R}[X]$ nad $\mathbb{R}$,
- zbiór funkcji $f : \mathbb{R} \to \mathbb{R}$.

Zbiór $\mathbb{R}_{>0}$ nie jest przestrzenią wektorową nad $\mathbb{R}$, ponieważ nie jest domknięty względem mnożenia przez -1.

Definicja. Niech V będzie przestrzenią wektorową nad $\mathbb{K}$. Zbiór $W \subseteq V$ nazywamy podprzestrzenią wektorową V, jeśli $(W, +)$ jest podgrupą $(V, +)$ oraz dla każdego $\alpha \in \mathbb{K}$ i $w \in W$ zachodzi $\alpha w \in W$.

Definicja. Kombinacją liniową wektorów $v_1, \ldots, v_n \in V$ nazywamy każdy wektor postaci $\alpha_1 v_1 + \ldots + \alpha_n v_n$ dla $\alpha_1, \ldots, \alpha_n \in \mathbb{K}$.

Definicja. Niech $A \subseteq V$ będzie zbiorem wektorów. Zbiór wszystkich kombinacji liniowych wektorów należących do A oznaczamy $\operatorname{span} A$ lub $\operatorname{Lin} A$. Zauważmy, że jest on podprzestrzenią wektorową V. Tę podprzestrzeń nazywamy podprzestrzenią generowaną (rozpinaną) przez A. Mówimy, że zbiór A generuje tę podprzestrzeń.

Definicja. Zbiór wektorów $A \subseteq V$ nazywamy liniowo niezależnym, jeśli dowolna kombinacja liniowa wektorów z A o niezerowych współczynnikach nie jest równa 0. Inaczej mówiąc, jeśli $\alpha_1 v_1 + \ldots + \alpha_n v_n = 0$ dla pewnych $v_1, \ldots, v_n \in A$ i $\alpha_1, \ldots, \alpha_n \in \mathbb{K}$, to $\alpha_1 = \ldots = \alpha_n = 0$.

Definicja. Bazą przestrzeni wektorowej V nazywamy dowolny zbiór liniowo niezależny $A \subseteq V$, który generuje V, czyli $\operatorname{span} A = V$.

Baza przestrzeni wektorowej V to równoważnie minimalny na zawieranie zbiór generujący V lub maksymalny na zawieranie zbiór liniowo niezależny w V. Istnienie bazy dowolnej przestrzeni wektorowej wynika z indukcji pozaskończonej (lub lematu Kuratowskiego-Zorna): zaczynamy od $A = \emptyset$ i i póki $\operatorname{span} A \neq V$, dodajemy do A wektor z $V \setminus \operatorname{span} A$. Wszystkie bazy przestrzeni wektorowej V są równoliczne (na moc).

Definicja. Wymiarem przestrzeni wektorowej V, oznaczanym $\dim V$, nazywamy moc dowolnej bazy V.

Rozważmy $\mathbb{R}_n[X]$. Jest to zbiór wszystkich wielomianów postaci $a_0 + a_1 X + \ldots + a_n X^n$. Dodawanie i mnożenie przez skalar definiujemy w sposób naturalny. Łatwo się przekonać, że jest to przestrzeń wektorowa (podprzestrzeń przestrzeni wielomianów). Standardową bazą $\mathbb{R}_n[X]$ jest zbiór $\{1, X, \ldots, X^n\}$. Oczywiście generuje on wszystkie wielomiany stopnia co najwyżej n. Jednocześnie jeśli jest $a_0 + \ldots + a_n X^n = 0$, to musi być $a_0 = \ldots = a_n = 0$, bo wielomiany są sobie równe dokładnie wtedy, gdy ich współczynniki są równe. W konsekwencji $\dim(\mathbb{R}_n[X]) = n + 1$.

Przykłady innych baz tej przestrzeni:

- $\left\{1, (X-1), (X-1)^2, \ldots, (X-1)^n\right\}$,
- $\left\{1, (X+1), (X+1)^2, \ldots, (X+1)^n\right\}$,
- $\left\{X^{\underline{k}} : k = 0, \ldots, n\right\}$, gdzie $X^{\underline{k}} = X(X-1)\ldots(X-k+1)$ dla $k > 0$ oraz $X^{\underline{0}} = 1$,
- $\left\{X^{\overline{k}} : k = 0, \ldots, n\right\}$, gdzie $X^{\overline{k}} = X(X+1)\ldots(X+k-1)$ dla $k > 0$ oraz $X^{\overline{0}} = 1$.

Dowód w przypadku każdej z tych baz przebiega tak samo. Zauważmy, że mamy dokładnie po jednym wielomianie każdego stopnia od 0 do n. Zatem jeśli chcemy przedstawić $A(X) = a_0 + \ldots + a_n X^n$ w aktualnie rozważanej bazie, to bierzemy element bazy o stopniu n i przemnażamy go przez taki czynnik, żeby jego współczynnik wiodący wynosił a_n. Odejmując taki wielomian od $A(X)$ dostajemy pewien wielomian $a_0' + \ldots + a_{n-1}' X^{n-1}$, dla którego możemy potworzyć tę operację. W końcu dostaniemy wielomian zerowy, co da nam odpowiednie przedstawienie A w rozważanej bazie. Liniową niezależność pokazujemy następująco: niech $\alpha_0 P_0(X) + \ldots + \alpha_n P_n(X) = 0$, gdzie $P_i(X)$ jest (jedynym) elementem bazy o stopniu i. Mamy $\alpha_n = 0$, bo współczynnik przy X^n w wielomianie zerowym wynosi 0, a po lewej stronie wynosi $\alpha_n p_n^{(n)}$, gdzie $p_n^{(n)} \neq 0$ jest n-tym współczynnikiem $P_n(X)$. Kontynuując takie rozumowanie dostajemy $\alpha_0 = \ldots = \alpha_n = 0$.

I.3.8 Odwzorowania liniowe i wieloliniowe

Odwzorowania liniowe i wieloliniowe, podstawowe własności i przykłady

- homomorfizm przestrzeni wektorowych (liniowych)
- jądro i obraz odwzorowania liniowego
- związek między wymiarem jądra, obrazu i dziedziny odwzorowania liniowego
- rząd odwzorowania liniowego
- odwracalność odwzorowania liniowego
- reprezentacja macierzowa odwzorowania liniowego dla przestrzeni skończenie wymiarowych

Definicja. Niech V, W będą przestrzeniami wektorowymi nad tym samym ciałem $\mathbb{K}$. Odwzorowaniem liniowym (homomorfizmem przestrzeni wektorowych) nazywamy funkcję $h : V \to W$, która jest homomorfizmem między grupami $(V, +)$ i $(W, +)$ a do tego dla $v \in V$ i $\alpha \in \mathbb{K}$ zachodzi $h(\alpha v) = \alpha h(v)$. Inaczej mówiąc, odwzorowanie h jest liniowe, jeśli dla dowolnych $\alpha, \beta \in \mathbb{K}$ oraz $v, w \in V$ zachodzi $h(\alpha v + \beta w) = \alpha h(v) + \beta h(w)$.

Kilka przykładów:

- identyczność: $\mathrm{id}(x) = x$,
- mnożenie przez stałą: $h(x) = \lambda x$,
- projekcja: $h(x, y, z) = (x, y)$,
- pochodna: $\mathrm{D} : \mathbb{R}_n[X] \to \mathbb{R}_{n-1}[X]$ zadana przez $\mathrm{D}(P) = P'$.

Dla dowolnego odwzorowania liniowego h zachodzi $h(0) = 0$ i $h(-v) = -h(v)$.

Definicja. Obrazem homomorfizmu $h : V \to W$ nazywamy zbiór $\mathrm{im}\, h := h(V) \subseteq W$. Jądrem h nazywamy zbiór $\ker h := h^{-1}(0) = \{v \in V : h(v) = 0\}$.

Obraz homomorfizmu h jest podprzestrzenią W. Jądro homomorfizmu jest podprzestrzenią V.

Twierdzenie. Niech $h : V \to W$ będzie homomorfizmem. Wtedy

$$\dim \ker h + \dim \mathrm{im}\, h = \dim V.$$

Dowód. Niech $A = \{v_i\}_{i \in I}$ będzie bazą $\ker h$. Niech $\{w_j\}_{j \in J}$ będzie bazą $\mathrm{im}\, h$ i niech $B = \{u_j\}_{j \in J}$ będą takie, że $h(u_j) = w_j$. Pokażemy, że zbiór $A \cup B$ jest bazą V.

Jeśli mamy $\sum_{k=1}^{n} a_k v_{i_k} + \sum_{\ell=1}^{m} b_\ell u_{j_\ell} = 0$, to stosując obustronnie h mamy $\sum_{\ell=1}^{m} b_\ell w_{j_\ell} = 0$, co daje nam $b_1 = \ldots = b_m = 0$, a więc $a_1 = \ldots = a_n = 0$. Zatem mamy liniową niezależność.

Ustalmy dowolne $v \in V$. Możemy przedstawić $h(v) = \sum_{\ell=1}^{m} b_\ell w_{j_\ell}$. Niech $v' = v - \sum_{\ell=1}^{m} b_\ell u_{j_\ell}$. Mamy $f(v') = f(v) - \sum_{\ell=1}^{m} b_\ell w_{j_\ell} = 0$. Zatem $v' \in \ker h$ i możemy przedstawić $v' = \sum_{k=1}^{n} a_k v_{i_k}$. To daje nam odpowiednie przedstawienie v. □

Definicja. Rzędem odwzorowania liniowego nazywamy wymiar jego obrazu: $\mathrm{rank}(h) = \dim(\mathrm{im}(h))$.

Definicja. Mówimy, że odwzorowanie liniowe $h : V \to W$ jest izomorfizmem, jeśli jest bijekcją. Wtedy istnieje odwzorowanie odwrotne $h^{-1} : W \to V$, które również jest liniowe: jeśli $h^{-1}(w_1) = v_1$ i $h^{-1}(w_2) = v_2$, to $h(\alpha v_1 + \beta v_2) = \alpha w_1 + \beta w_2$, czyli $h^{-1}(\alpha w_1 + \beta w_2) = \alpha v_1 + \beta v_2 = \alpha h^{-1}(w_1) + \beta h^{-1}(w_2)$.

Odwzorowanie liniowe $h : V \to W$ jest izomorfizmem, gdy jest iniekcją i surjekcją, czyli gdy $\ker h = \{0\}$ i $\mathrm{im}\, h = W$. Aby h było izomorfizmem musi zachodzi $\dim V = \dim W$. W przypadku skończenie wymiarowym ($\dim V = \dim W < \infty$), wobec $\dim \ker h + \dim \mathrm{im}\, h = \dim V = \dim W$, wystarczy sprawdzić tylko $\ker h = \{0\}$ lub $\mathrm{im}\, h = W$.

Definicja. Załóżmy, że V ma bazę $v_1, \ldots, v_n$, a W ma bazę $w_1, \ldots, w_m$. Niech $h : V \to W$ będzie homomorfizmem. Każdy element $h(v_j)$ można zapisać w bazie W jako $h(v_j) = a_{1j} w_1 + \ldots + a_{mj} w_m$. Macierz $A = [a_{ij}]_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m}$ nazywamy macierzą odwzorowania liniowego h w zadanych bazach.

Dla dowolnego $x = \sum_{i=1}^{n} x_i v_i \in V$ zachodzi $h(x) = \sum_{i=1}^{n} x_i h(v_i) = \sum_{i=1}^{n} \sum_{j=1}^{m} x_i a_{ij} w_j$. Zatem jeśli utożsamimy x z wektorem $X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$, to $h(x)$ jest zadane wektorem AX. Zatem macierz A jednoznacznie wyznacza odwzorowanie h.

Trochę przeliczeń doprowadza nas do faktu, że jeśli $h : V \to W$ i $f : W \to U$ są homomorfizmami o macierzach A i B, to homomorfizm $f \circ h$ jest zadany macierzą BA. Zatem można utożsamiać odwzorowania liniowe między skończenie wymiarowymi przestrzeniami z macierzami. Składanie odwzorowań sprowadza się w takiej sytuacji do mnożenia macierzy.

Definicja. Odwzorowanie: $F : V_1 \times \ldots \times V_k \to W$ nazywamy wieloliniowym, jeśli po ustaleniu wszystkich argumentów poza jednym pozostaje liniowe względem tego argumentu.

Przykładem odwzorowania wieloliniowego jest wyznacznik. Jeśli $\dim V = n$, to $\det : V^n \to \mathbb{R}$ jest liniowe ze względu na każdą kolumnę osobno, czyli n-liniowe. Dwuliniowy jest na przykład iloczyn skalarny $\langle \cdot, \cdot \rangle : \mathbb{R}^n \times \mathbb{R}^n \ni (x, y) \mapsto \sum_{i=1}^{n} x_i y_i \in \mathbb{R}$.

I.3.9 Wartości własne

Wartości własne i wektory własne macierzy – własności i metody liczenia.

- wartość własna, wektor własny, podprzestrzenie wartości własnej
- wielomian charakterystyczny i jego pierwiastki
- krotność algebraiczna i geometryczna oraz związek miedzy nimi
- wartości własne macierzy (ze znajomością dowodu): ortogonalnych i unitarnych, rzeczywistych symetrycznych i hermitowskich, idempotentnych (macierzy rzutu)
- jeden przykład zastosowania

Definicja. Niech $A \in M_{n\times n}(K)$, gdzie $\mathbb{K} = \mathbb{R}$ lub $\mathbb{K} = \mathbb{C}$. Liczbę $\lambda \in \mathbb{K}$ nazywamy wartością własną macierzy A, jeżeli istnieje niezerowy wektor $v \neq 0$ taki, że $Av = \lambda v$. Wektor v nazywamy wektorem własnym odpowiadającym wartości własnej λ. Analogiczne definicje możemy wprowadzić dla odwzorowania liniowego $h : \mathbb{K}^n \to \mathbb{K}^n$.

Definicja. Podprzestrzenią własną odpowiadającą λ nazywamy zbiór $E_\lambda = \ker(A - \lambda I)$. Jest to podprzestrzeń przestrzeni $\mathbb{K}^n$ (jako jądro odwzorowania liniowego). Wszystkie niezerowe elementy E_λ są wektorami własnymi A (bo $(A - \lambda I)x = 0 \iff Ax = \lambda x$).

Definicja. Wielomianem charakterystycznym macierzy A nazywamy wielomian $\chi_A(t) = \det(A - tI)$.

Twierdzenie. Liczba λ jest wartością własną macierzy A wtedy i tylko wtedy, gdy $\chi_A(\lambda) = 0$.

Dowód. Mamy $Av = \lambda v$ dla pewnego $v \neq 0$ dokładnie wtedy, gdy $(A - \lambda I)v = 0$ ma nietrywialne rozwiązanie. To zachodzi wtedy i tylko wtedy, gdy $\det(A - \lambda I) = 0$. □

Mamy $\deg(\chi_A) = n$ (współczynnik wiodący tego wielomianu to $(-1)^n t^n$). Zatem macierz A ma co najwyżej n różnych wartości własnych. Aby wyznaczyć wartości własne rozwiązujemy równanie $\chi_A(\lambda) = 0$. Następnie dla ustalonego λ znajdujemy wektory własne rozwiązując równanie $(A - \lambda I)x = 0$.

Definicja. Krotnością algebraiczną wartości własnej λ nazywamy jej krotność jako pierwiastka χ_A. Krotnością geometryczną wartości własnej λ nazywamy $\dim(E_\lambda)$.

Krotność geometryczna jest zawsze równa co najwyżej krotności algebraicznej. Może być mniejsza. Jeśli krotność geometryczna każdej wartości własnej macierzy A jest równa jej krotności algebraicznej, to macierz jest diagonalizowalna (istnieje baza złożona z wektorów własnych).

Definicja. Macierz rzeczywista Q jest ortogonalna, gdy $Q^TQ = I$. Macierz zespolona U jest unitarna, gdy $U^*U = I$ ($U^* = \overline{U}^T$ nazywamy sprzężeniem hermitowskim). Dla macierzy rzeczywistych traktowanych jak zespolone unitarność jest równoważna ortogonalności.

Twierdzenie. Jeżeli λ jest wartością własną macierzy ortogonalnej lub unitarnej, to $|\lambda| = 1$.

Dowód. Niech A będzie macierzą unitarną. Wtedy $\|v\| = \|Av\|$. Wobec $Av = \lambda v$ mamy $\|v\| = \|Av\| = \|\lambda v\| = |\lambda| \, \|v\|$. Zatem $|\lambda| = 1$. □

Definicja. Macierz rzeczywista jest symetryczna, gdy $A^T = A$. Macierz zespolona jest hermitowska, gdy $A^* = A$. Dla macierzy rzeczywistych traktowanych jak zespolone hermitowskość jest równoważna symetryczności.

Twierdzenie. Wszystkie wartości własne macierzy symetrycznej lub hermitowskiej są rzeczywiste.

Dowód. Niech $Av = \lambda v$. Wtedy $\langle Av, v\rangle = \lambda\langle v, v\rangle$. Z $A = A^*$ mamy

$$\lambda \langle v, v\rangle = \langle \lambda v, v\rangle = \langle Av, v\rangle = (Av)^* \, v = v^*A^*v = v^*Av = \langle v, Av\rangle = \langle v, \lambda v\rangle = \overline{\lambda} \langle v, v\rangle \, .$$

Stąd $\lambda = \overline{\lambda}$. □

Definicja. Macierz nazywamy idempotentną, gdy $A^2 = A$.

Twierdzenie. Wartości własne macierzy idempotentnej należą do zbioru $\{0, 1\}$.

Dowód. Niech $Av = \lambda v$. Mamy $A^2v = AAv = A\lambda v = \lambda^2 v$. Z drugiej strony $A^2v = Av = \lambda v$. Wobec $v \neq 0$ mamy $\lambda^2 = \lambda$, czyli $\lambda (\lambda - 1) = 0$. □

Jednym z najważniejszych zastosowań wartości własnych jest obliczanie potęg macierzy. Jeśli macierz $A \in M_{n\times n}(\mathbb{K})$ jest diagonalizowalna, to mamy $A = PDP^{-1}$ dla diagonalnej macierzy D i odpowiedniej macierzy przejścia P. Wtedy $A^k = PD^kP^{-1}$. D^k łatwo jest obliczyć, bo wystarczy podnieść elementy na diagonali do odpowiedniej potęgi.

Inne zastosowanie: łańcuchy Markowa. Rozważmy łańcuch Markowa zadany macierzą A. To znaczy, że jeśli π jest rozkładem prawdopodobieństwa stanów (wektorem indeksowanym stanami, w którym wartość to prawdopodobieństwo bycia w danym stanie), to $A\pi$ jest rozkładem prawdopodobieństwa po kolejnym kroku łańcucha. Rozkład π jest stacjonarny, jeśli $A\pi = \pi$, czyli gdy π jest wektorem własnym A dla wartości własnej 1.

I.3.10 Diagonalizacja

Macierze diagonalne i diagonalizacja macierzy.

- co to znaczy, że dwie macierze są podobne?
- co to znaczy, że macierz jest diagonalizowalna?
- kiedy macierz jest diagonalizowalna?
- jak zdiagonalizować macierz i wyznaczyć macierz przejścia przez wyznaczanie wartości i wektorów własnych
- przykład macierzy diagonalizowalnej i niediagonalizowalnej
- co to jest postać Jordana i kiedy istnieje?
- czy istnieje i jak wygląda postać Jordana macierzy (bez dowodu): ortogonalnych i unitarnych, rzeczywistych symetrycznych i hermitowskich, idempotentnych (macierzy rzutu)

Definicja. Macierze $A, B \in M_{n\times n}(\mathbb{K})$ nazywamy podobnymi, jeśli istnieje macierz odwracalna P taka, że $B = P^{-1}AP$. Oznaczamy to $A \sim B$. Jeśli macierze A i B są podobne, to mają ten sam wyznacznik, rząd, wielomian charakterystyczny i wartości własne.

Definicja. Macierz diagonalna to macierz postaci

$$D = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}.$$

Definicja. Macierz A nazywamy diagonalizowalną, jeśli istnieje macierz odwracalna P oraz diagonalna D takie, że $A = PDP^{-1}$. Inaczej mówiąc, $A \sim D$.

Twierdzenie. Niech $A \in M_{n\times n}(\mathbb{K})$. Następujące warunki są równoważne.

1. Macierz A jest diagonalizowalna.
2. Istnieje baza $\mathbb{K}^n$ złożona z wektorów własnych A.
3. Suma wymiarów podprzestrzeni własnych odpowiadających różnym wartościom własnym wynosi n.

Dowód. ($1 \implies 2$) Z założenia mamy $A = PDP^{-1}$ dla macierzy odwracalnej P i diagonalnej D. Niech $v_1, v_2, \ldots, v_n \in \mathbb{K}^n$ będą kolumnami P, a $\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{K}$ elementami na przekątnej D. Równość $AP = PD$ zapisana kolumna po kolumnie daje nam $Av_i = \lambda_i v_i$ dla $i = 1, \ldots, n$. Macierz P jest odwracalna, więc jej kolumny są liniowo niezależne i stanowią bazę $\mathbb{K}^n$.

($2 \implies 3$) Załóżmy, że istnieje baza $B = \{v_1, v_2, \ldots, v_n\}$ przestrzeni $\mathbb{K}^n$ złożona z wektorów własnych macierzy A. Niech $\mu_1, \mu_2, \ldots, \mu_k$ będą wszystkimi różnymi wartościami własnymi macierzy A, a $V_{\mu_i} = \{v \in \mathbb{K}^n : Av = \mu_i v\}$ ich podprzestrzeniami własnymi. Niech $B_i = B \cap V_{\mu_i}$. Mamy $\sum_{i=1}^{k} |B_i| = |B| = n$. Wektory w każdym B_i są liniowo niezależne, więc $\dim V_{\mu_i} \geqslant |B_i|$. Jeśli $v \in V_{\mu_i} \cap V_{\mu_j}$, to $\mu_i v = Av = \mu_j v$. Zatem $v = 0$. Z tego wynika, że

$$n = \sum_{i=1}^{k} |B_i| \leqslant \sum_{i=1}^{k} \dim V_{\mu_i} = \dim\left(V_{\mu_1} + \ldots + V_{\mu_k}\right) \leqslant n,$$

czyli odpowiednia suma wynosi dokładnie n.

($3 \implies 1$) Załóżmy, że suma wymiarów podprzestrzeni własnych odpowiadających różnym wartościom własnym $\mu_1, \mu_2, \ldots, \mu_k$ wynosi n. Niech B_i będzie bazą przestrzeni V_{μ_i}. Mamy $V_{\mu_i} \cap V_{\mu_j} = \{0\}$, więc $B = B_1 \cup B_2 \cup \cdots \cup B_k$ jest bazą $\mathbb{K}^n$. Niech P będzie macierzą, której kolumnami są kolejne wektory z bazy B, a D będzie macierzą diagonalną, która na przekątnej ma wartości własne odpowiadające tym wektorom. Ponieważ kolumny P są liniowo niezależne, macierz P jest odwracalna. Z konstrukcji zachodzi $AP = PD$. □

Powyższe twierdzenie daje nam algorytm diagonalizacji: wyznaczamy wartości własne i wektory własne, znajdujemy bazy przestrzeni własnych, a następnie łączymy je w bazę całej przestrzeni. Konstruujemy P jako macierz przejścia między bazą kanoniczną a tą bazą (czyli robimy dokładnie to, co w dowodzie powyższego twierdzenia).

Dla $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ mamy $\det(A - tI) = (t-3)(t+1)$. Dla $v_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ i $v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ mamy $Av_1 = 3v_1$ i $Av_2 = -v_2$. Kładąc $P = \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ i $D = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}$ mamy $AP = PD$.

Dla $A = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$ mamy $\det(A - tI) = (2-t)^2$. Zatem jedyną wartością własną jest 2. Rozwiązując równanie $A \begin{bmatrix} x \\ y \end{bmatrix} = 2 \begin{bmatrix} x \\ y \end{bmatrix}$ dostajemy $2x + y = 2x$ i $2y = 2y$, co jest spełnione dla dowolnego wektora postaci $\begin{bmatrix} x \\ 0 \end{bmatrix}$. Nie ma dwóch liniowo niezależnych wektorów takiej postaci i A nie jest diagonalizowalna.

Definicja. Blokiem Jordana odpowiadającym λ nazywamy macierz

$$J_k(\lambda) = \begin{pmatrix} \lambda & 1 & 0 & \cdots \\ 0 & \lambda & 1 & \cdots \\ 0 & 0 & \lambda & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix},$$

gdzie na diagonali jest λ, nad diagonalą 1 a poza tym są same 0.

Definicja. Postacią Jordana macierzy A nazywamy macierz blokowo diagonalną

$$J = \begin{bmatrix} J_1 & 0 & 0 & 0 \\ 0 & J_2 & 0 & 0 \\ 0 & 0 & J_3 & 0 \\ 0 & 0 & 0 & \ddots \end{bmatrix}$$

złożoną z bloków Jordana, taką że $A = PJP^{-1}$ dla pewnej macierzy odwracalnej P.

Każda macierz zespolona posiada postać Jordana. Dla macierzy rzeczywistych może być konieczne przejście do $\mathbb{C}$. Diagonalizacja jest szczególnym przypadkiem, gdy wszystkie bloki Jordana mają rozmiar 1.

Każda macierz ortogonalna może zostać zapisana w pewnej bazie ortonormalnej jako

$$\begin{bmatrix} C_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & C_k \end{bmatrix},$$

gdzie $C_i = \begin{bmatrix} \pm 1 \end{bmatrix}$ lub $C_i = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}$ jest macierzą obrotu o kąt φ. Dodatkowo nad $\mathbb{C}$ macierz obrotu jest diagonalizowalna (mimo, że nad $\mathbb{R}$ nie jest, poza trywialnymi przypadkami). Zatem każda macierz unitarna jest diagonalizowalna. Macierz ortogonalna nie musi być diagonalizowalna and $\mathbb{R}$, ale posiada postać Jordana nad $\mathbb{C}$, która jest diagonalna. Warto dodać, że wszystkie wartości własne macierzy unitarnej spełniają $|\lambda| = 1$.

Każda macierz hermitowska (symetryczna) jest diagonalizowalna. Jej wartości własne są rzeczywiste, zatem postać Jordana składa się z liczb rzeczywistych na diagonali. Warto dodać, że dla macierzy hermitowskiej (symetrycznej) A nie tylko istnieje baza składająca się z wektorów własnych, ale nawet ortonormalna baza składająca się z wektorów własnych. Z tego wynika, że istnieje macierz unitarna (ortogonalna) P taka, że P^*AP jest diagonalna.

Każda macierz idempotentna jest diagonalizowalna. Jej wartości własne to 0 i 1, zatem postać Jordana składa się z pewnej ilości zer i jedynek na diagonali.

I.3.11 Przestrzenie euklidesowe i unitarne

Przestrzenie euklidesowe i unitarne – definicje, własności i przykłady.

- iloczyn skalarny w przestrzeniach liniowych (nad $\mathbb{R}$ i nad $\mathbb{C}$),
- definicja przestrzeni euklidesowej (unitarnej)
- norma wektora; norma indukowana przez iloczyn skalarny
- nierówność Cauchy'ego-Schwarza (bez dowodu) i nierówność trójkąta (bez dowodu)
- ortogonalność wektorów, baza ortogonalna (ortonormalna)
- metoda ortogonalizacji Grama-Schmidta
- izometria liniowa i związek z macierzami ortogonalnymi i unitarnymi

Definicja. Niech V będzie przestrzenią wektorową nad $\mathbb{R}$. Iloczynem skalarnym nazywamy odwzorowanie $\langle\cdot,\cdot\rangle : V \times V \to \mathbb{R}$ spełniające:

1. liniowość względem pierwszego argumentu: $\langle \alpha u + \beta v, w\rangle = \alpha\langle u, w\rangle + \beta\langle v, w\rangle$,
2. symetrię: $\langle u, v\rangle = \langle v, u\rangle$,
3. dodatnią określoność: $\langle v, v\rangle \geqslant 0$ oraz $\langle v, v\rangle = 0$ wtedy i tylko wtedy, gdy $v = 0$.

Definicja. Niech V będzie przestrzenią wektorową nad $\mathbb{C}$. Iloczynem skalarnym nazywamy odwzorowanie $\langle\cdot,\cdot\rangle : V \times V \to \mathbb{C}$ spełniające:

1. liniowość względem pierwszego argumentu: $\langle \alpha u + \beta v, w\rangle = \alpha\langle u, w\rangle + \beta\langle v, w\rangle$,
2. sprzężoną symetrię: $\langle u, v\rangle = \overline{\langle v, w\rangle}$,
3. dodatnią określoność: $\langle v, v\rangle \geqslant 0$ oraz $\langle v, v\rangle = 0$ wtedy i tylko wtedy, gdy $v = 0$.

Definicja. Przestrzenią euklidesową nazywamy rzeczywistą przestrzeń wektorową wyposażoną w rzeczywisty iloczyn skalarny. Przestrzenią unitarną nazywamy zespoloną przestrzeń wektorową wyposażoną w zespolony iloczyn skalarny.

W przestrzeni $\mathbb{R}^n$ standardowym iloczynem skalarnym jest $\langle v, w\rangle = \sum_{i=1}^n v_i w_i$. W przestrzeni $\mathbb{C}^n$ standardowym iloczynem skalarnym jest $\langle v, w\rangle = \sum_{i=1}^n v_i \overline{w_i}$.

Definicja. Niech V będzie przestrzenią wektorową. Normą na V nazywamy dowolną funkcję $\|\cdot\| : V \to \mathbb{R}_{\geqslant 0}$ taką, że

1. $\|\alpha v\| = |\alpha|\,\|v\|$,
2. $\|v + w\| \leqslant \|v\| + \|w\|$ (nierówność trójkąta),
3. $\|v\| = 0 \iff v = 0$.

Normą indukowaną przez iloczyn skalarny $\langle\cdot,\cdot\rangle$ nazywamy funkcję $\|v\| = \sqrt{\langle v, v\rangle}$. Wszystkie warunki poza nierównością trójkąta są oczywiste. Nierówność trójkąta wynika z poniższych twierdzeń.

Twierdzenie (Cauchy, Schwarz)**.** Dla dowolnych $u, v \in V$ zachodzi $|\langle u, v\rangle| \leqslant \|u\|\,\|v\|$. Równość zachodzi wtedy i tylko wtedy, gdy wektory są liniowo zależne.

Twierdzenie (Nierówność Minkowskiego)**.** Dla dowolnych u, v zachodzi $\|u + v\| \leqslant \|u\| + \|v\|$. Równość zachodzi wtedy i tylko wtedy, gdy wektory są liniowo zależne.

Definicja. Wektory u, v nazywamy ortogonalnymi, jeśli $\langle u, v\rangle = 0$. Wprowadzamy oznaczenie $u \perp v$.

Definicja. Układ wektorów $v_1, \ldots, v_n$ nazywamy ortogonalnym, jeśli $v_i \perp v_j$ dla $i \neq j$. Układ ortogonalny nazywamy ortonormalnym, gdy dodatkowo $\|v_i\| = 1$ dla każdego i.

Twierdzenie. Każdy układ ortogonalny złożony z niezerowych wektorów jest liniowo niezależny.

Dowód. Niech $v_1, \ldots, v_n$ będzie układem ortogonalnym. Załóżmy, że $\alpha_1 v_1 + \ldots + \alpha_n v_n = 0$. Mnożąc skalarnie przez v_k dostajemy $\alpha_k \langle v_k, v_k\rangle = \langle \alpha_1 v_1 + \ldots + \alpha_n v_n, v_k\rangle = \langle 0, v_k\rangle = 0$. Wobec $\langle v_k, v_k\rangle \neq 0$ mamy $\alpha_k = 0$. Dowolność k daje tezę. □

Widzimy, że dowolny układ ortogonalny o $\dim V$ elementach jest bazą V. Okazuje się, że każdą bazę V można „poprawić” do bazy ortogonalnej, a nawet ortonormalnej. Robimy to za pomocą procedury zwanej ortogonalizacją Grama-Schmidta.

Propozycja. Załóżmy, że $v_1, \ldots, v_n$ są liniowo niezależne. Ustalamy iteracyjnie $u_1 = v_1$, a następnie $u_k = v_k - \sum_{i=1}^{k-1} \frac{\langle v_k, u_i\rangle}{\langle u_i, u_i\rangle} u_i$ dla $k > 1$. Wtedy $u_1, \ldots, u_n$ są ortogonalne. Aby otrzymać układ ortonormalny, normalizujemy $w_i = \frac{u_i}{\|u_i\|}$.

Dowód. Ustalmy indeks $j > 1$. Możemy założyć indukcyjnie, że $u_k \perp u_\ell$ dla $k, \ell < j$. Wystarczy pokazać, że $u_j \perp u_k$ dla $k < j$. Mamy

$$\langle u_j, u_k \rangle = \left\langle v_j - \sum_{i=1}^{j-1} \frac{\langle v_j, u_i \rangle}{\langle u_i, u_i \rangle} u_i, u_k \right\rangle = \langle v_j, u_k \rangle - \frac{\langle v_j, u_k \rangle}{\langle u_k, u_k \rangle} \langle u_k, u_k \rangle = 0.$$

□

Definicja. Odwzorowanie liniowe $f : V \to W$ między przestrzeniami unitarnymi (euklidesowymi) nazywamy izometrią, jeśli jest bijekcją i $\langle f(v), f(w) \rangle = \langle v, w \rangle$ dla wszystkich $v, w \in V$.

Definicja. Macierz $A \in M_{n \times n}(\mathbb{R})$ nazywamy ortogonalną, gdy $A^T A = I$. Macierz $A \in M_{n \times n}(\mathbb{C})$ nazywamy unitarną, gdy $A^* A = I$, gdzie $A^* = \overline{A}^T$.

Propozycja. Następujące warunki są równoważne.

1. A jest macierzą ortogonalną (unitarną).
2. Kolumny A tworzą układ ortonormalny.
3. $\langle Av, Aw \rangle = \langle v, w \rangle$ dla dowolnych wektorów v, w.

Dowód. Rozważamy tylko macierze ortogonalne, dla unitarnych dowód przebiega tak samo.

($1 \iff 2$) Mamy $\left(A^T A\right)_{ij} = \sum_{k=1}^n A^T_{ik} A_{kj} = \sum_{k=1}^n A_{ki} A_{kj} = \langle A_i, A_j \rangle$, gdzie A_i oznacza i-tą kolumnę A. Jest $\left(AA^T\right)_{ii} = 1$ i $\left(AA^T\right)_{ij} = 0$ dla $i \neq j$ wtedy i tylko wtedy, gdy $A^T A = I$.

($1 \implies 3$) $\langle Av, Aw \rangle = (Av)^T Aw = v^T A^T Aw = v^T w = \langle v, w \rangle$.

($3 \implies 1$) Mamy $v^T A^T A w = v^T w$, czyli $v^T \left(A^T A - I\right) w = 0$. Podstawiając wektory bazy kanonicznej $v = e_i$ i $w = e_j$ dostajemy $\left(A^T A - I\right)_{ij} = 0$, więc $\left(A^T A\right)_{ii} = 1$ i $\left(A^T A\right)_{ij} = 0$ dla $i \neq j$. □

Z powyższego twierdzenia wynika, że odwzorowanie liniowe f jest izometrią wtedy i tylko wtedy, gdy odpowiadająca mu macierz jest unitarna.

I.4 Metody Formalne Informatyki

I.4.1 Własności liczb naturalnych

Podaj definicje dodawania, mnożenia, potęgowania i odejmowania w oparciu o twierdzenie o definiowaniu przez indukcję. Udowodnij indukcyjnie własności tych działań takie jak łączność i przemienność.

Twierdzenie (O definiowaniu przez indukcję)**.** Niech A, B będą zbiorami, niech $f : A \to B$ i $g : B \times \mathbb{N} \times A \to B$ będą funkcjami. Istnieje dokładnie jedna funkcja $h : A \times \mathbb{N} \to B$ taka, że

- $h(a, 0) = f(a)$ dla każdego $a \in A$,
- $h(a, n') = g(h(a, n), n, a)$ dla wszystkich $a \in A$ i $n \in \mathbb{N}$,

gdzie n' oznacza następnik n.

Dowód. Niech $e \in B^{A \times m'}$ dla pewnej liczby naturalnej m. Warunki

$$e\left(a,0\right) = f\left(a\right) \text{ dla każdego } a \in A,$$

$$e\left(a,n'\right) = g\left(e\left(a,n\right),n,a\right) \text{ dla każdego } a \in A \text{ i } n \in m$$

oznaczamy $(*)$. Zaczynamy od rozważenia zbioru $H = \left\{e : \exists_{m \in \mathbb{N}}\ e \in B^{A \times m'} \text{ oraz } e \text{ spełnia } (*)\right\}$. Indukcyjnie dowodzimy, że H jest niepusty i zawiera co najmniej jedną funkcję $e : A \times m' \to B$ dla każdej liczby naturalnej m. Robimy to rozważając zbiór

$$P = \left\{m \in \mathbb{N} : \exists_{e \in B^{A \times m'}}\ e \in H\right\}$$

i z funkcji dla n definiujemy funkcję dla n' w jedyny możliwy sposób. Następnie pokazujemy, że $e, e' \in H$ zgadzają się na przecięciu swoich dziedzin. Robimy to nie wprost, biorąc najmniejsze n takie, że się nie zgadzają i dostając sprzeczność z $(*)$. Kończymy biorąc $h = \bigcup H$. Jest to funkcja z poprzedniego faktu. Jeśli istnieje druga funkcja h' spełniająca odpowiednie warunki, to mamy $h\left(a,n\right) \neq h'\left(a,n\right)$ dla pewnego $n \in \mathbb{N}$. Ale h i h' zawężone do $n' \times A$ są elementami H, co daje sprzeczność. $\square$

Definicja. Dodawanie definiujemy za pomocą twierdzenia o definiowaniu przez indukcję dla $A = B = \mathbb{N}$ i $f\left(n\right) = n$, $g\left(m,n,p\right) = m'$. Mamy

$$m + 0 = m,$$
$$m + n' = (m + n)'.$$

Twierdzenie. Dla dowolnych $a, b, c \in \mathbb{N}$ zachodzi $(a + b) + c = a + (b + c)$.

Dowód. Dowód prowadzimy przez indukcję względem c. Dla $c = 0$ jest

$$(a + b) + 0 = a + b,$$
$$a + (b + 0) = a + b.$$

Dalej zakładamy, że $(a + b) + k = a + (b + k)$ dla pewnego $k \in \mathbb{N}$. Pokażemy, że zachodzi to dla k':

$$(a + b) + k' = ((a + b) + k)' = (a + (b + k))' = a + (b + k)' = a + (b + k').$$

Pierwsze przejście to definicja dodawania, drugie to założenie indukcyjne, trzecie i czwarte to ponownie definicja dodawania. Na mocy zasady indukcji matematycznej twierdzenie jest prawdziwe. $\square$

Twierdzenie. Dla dowolnych $a, b \in \mathbb{N}$ zachodzi: $a + b = b + a$.

Dowód. Dowód wymaga udowodnienia dwóch prostych lematów pomocniczych (indukcją po n):

1. $0 + n = n$
2. $m' + n = (m + n)'$

Zakładając prawdziwość lematów prowadzimy indukcję po b. Dla $b = 0$ jest $a + 0 = a$ (z definicji) oraz $0 + a = a$ (z lematu 1). Zatem $a + 0 = 0 + a$. Teraz zakładamy, że $a + k = k + a$. Pokażemy, że zachodzi to dla k':

$$a + k' = (a + k)' = (k + a)' = k' + a.$$

Pierwsze przejście wynika z definicja dodawania, drugie z założenia indukcyjnego, a trzecie z lematu 2. To kończy dowód. □

Definicja. Mnożenie definiujemy za pomocą twierdzenia o definiowaniu przez indukcję dla $A = B = \mathbb{N}$ i $f(n) = 0$, $g(m, n, p) = m + p$. Mamy

$$m \cdot 0 = 0$$
$$m \cdot n' = (m \cdot n) + m$$

Twierdzenie. Mnożenie jest przemienne.

Dowód. Indukcją po n dowodzimy, że $0 \cdot n = 0$ i $m' \cdot n = m \cdot n + n$. Stosując te lematy dowodzimy tezę indukcyjnie. □

Lemat. Dla $m, n, p \in \mathbb{N}$ zachodzi $m \cdot (n + p) = m \cdot n + m \cdot p$.

Dowód. Indukcja po p. □

Twierdzenie. Mnożenie jest łączne, czyli dla $m, n, p \in \mathbb{N}$ zachodzi $(m \cdot n) \cdot p = m \cdot (n \cdot p)$.

Dowód. Indukcja po p. Baza $p = 0$ z definicji. W kroku indukcyjnym mamy

$$(m \cdot n) \cdot p' = (m \cdot n) \cdot p + m \cdot n = m \cdot (n \cdot p) + m \cdot n,$$

$$m \cdot (n \cdot p') = m \cdot (n \cdot p + n) = m \cdot (n \cdot p) + m \cdot n.$$

□

Definicja. Dla ustalonego $m \in \mathbb{N}$ potęgowanie definiujemy następująco:

$$m^0 = 0'$$
$$m^{n'} = m^n \cdot m$$

Definicja. Wprowadzamy funkcję poprzednika $p(n)$:

$$p(0) = 0$$
$$p(n') = n$$

Teraz dla ustalonego $m \in \mathbb{N}$ odejmowanie definiujemy następująco:

$$m - 0 = m$$
$$m - n' = p(m - n)$$

I.4.2 Liczby rzeczywiste

Konstrukcja Cantora liczb rzeczywistych. Porządek na liczbach rzeczywistych. Podaj bez dowodu twierdzenie o rozwinięciu liczby rzeczywistej w szereg.

Definicja. Ciągiem elementów ze zbioru A nazywamy funkcję $a : \mathbb{N} \to A$. Przez a_n oznaczamy $a(n)$.

Definicja. Ciągiem Cauchy'ego liczb wymiernych nazywamy każdy taki ciąg $a : \mathbb{N} \to \mathbb{Q}$, że

$$\forall_{\varepsilon \in \mathbb{Q} \wedge \varepsilon > 0} \exists_{n_0 \in \mathbb{N}} \forall_{k,\ell \geqslant n_0} |a_k - a_\ell| < \varepsilon$$

Definicja. Niech $X = \{a : \mathbb{N} \to \mathbb{Q}$ jest ciągiem Cauchy'ego$\}$. Na zbiorze X określamy relację $\simeq$ tak, że

$$a \simeq b \iff \forall_{\varepsilon \in \mathbb{Q} \wedge \varepsilon > 0} \exists_{n_0 \in \mathbb{N}} \forall_{n \geqslant n_0} |a_n - b_n| < \varepsilon$$

Twierdzenie. $\simeq$ jest relacją równoważności na X.

Dowód. Zwrotność i symetryczność są oczywiste. Wykażemy teraz przechodniość. Załóżmy, że $a \simeq b$ i $b \simeq c$. Ustalmy $\varepsilon > 0$. Niech $n_1, n_2 \in \mathbb{N}$ będą takie, że $\forall_{n \in \mathbb{N} \wedge n \geqslant n_1} |a_n - b_n| < \varepsilon/2$ oraz $\forall_{n \in \mathbb{N} \wedge n \geqslant n_2} |b_n - c_n| < \varepsilon/2$. Weźmy $n_0 = \max(n_1, n_2)$. Wtedy

$$\forall_{n \geqslant n_0} |a_n - c_n| \leqslant |a_n - b_n| + |b_n - c_n| < \varepsilon/2 + \varepsilon/2 = \varepsilon$$

Pierwsza nierówność wynika z nierówności trójkąta. □

Definicja. Zbiorem liczb rzeczywistych nazwiemy zbiór $X/_{\simeq}$. Oznaczymy go poprzez $\mathbb{R}$.

Intuicyjnie, daną liczbą rzeczywistą jest zbiór ciągów Cauchy'ego o wymiernych elementach, które zbiegają do niej w granicy.

Definicja. Dla ciągów a, b definiujemy ciąg $a+b$ jako $(a+b)(n) = a(n)+b(n)$. Podobnie definiujemy ciąg $a \cdot b$ jako $(a \cdot b)(n) = a(n) \cdot b(n)$. To pozwala na zdefiniowanie dodawania i mnożenia liczb rzeczywistych:

$$[a]_{\simeq} + [b]_{\simeq} = [a + b]_{\simeq}$$
$$[a]_{\simeq} \cdot [b]_{\simeq} = [a \cdot b]_{\simeq}$$

Twierdzenie. Te definicje są poprawne, to znaczy nie zależą od wyboru reprezentanta klasy.

Dowód. Jeśli $a \simeq \overline{a}$ i $b \simeq \overline{b}$, to $\left|a_n + b_n - \overline{a}_n - \overline{b}_n\right| \leqslant |a_n - \overline{a}_n| + \left|b_n - \overline{b}_n\right| < \varepsilon$ dla odpowiednio dużych n.

Z definicji ciągu Cauchy'ego istnieje takie $n_0 \in \mathbb{N}$ i $\varepsilon > 0$, że $\forall_{p > n_0} \ |a_p - a_{n_0}| < \varepsilon$. Wtedy dla $M_a = \max\{|a_0|, \ldots, |a_{n_0}|, |a_{n_0+1}| + \varepsilon\}$ mamy $|a_n| \leqslant M$ dla każdego n. Podobnie znajdujemy $M_{\overline{a}}, M_b, M_{\overline{b}}$. Bierzemy $M = \max\left\{M_a, M_b, M_{\overline{a}}, M_{\overline{b}}\right\}$. Jeżeli $|a_n - \overline{a}_n| < \frac{\varepsilon}{2M}$ i $\left|b_n - \overline{b}_n\right| < \frac{\varepsilon}{2M}$, to

$$\left|a_n b_n - \overline{a}_n \overline{b}_n\right| = \left|a_n\left(b_n - \overline{b}_n\right) + \overline{b}_n\left(a_n - \overline{a}_n\right)\right| \leqslant |a_n|\left|b_n - \overline{b}_n\right| + \left|\overline{b}_n\right||a_n - \overline{a}_n| < \varepsilon.$$

□

Definicja. Definiujemy porządek na $\mathbb{R}$ jako

$$[a]_{\simeq} < [b]_{\simeq} \iff \exists_{\varepsilon > 0} \exists_{n_0 \in \mathbb{N}} \forall_{n \geqslant n_0} a_n + \varepsilon < b_n$$

Mówimy wtedy, że ε rozdziela ciągi a, b poczynając od elementu a_{n_0}.

Twierdzenie. $(\mathbb{R}, \leqslant)$ to porządek liniowy.

Dowód. Niech $[a]_{\simeq} \neq [b]_{\simeq}$. Wtedy z definicji

$$\exists_{\varepsilon > 0} \forall_{n \in \mathbb{N}} \exists_{k \geqslant n} |a_k - b_k| \geqslant \varepsilon$$

Weźmy n_a, n_b takie, że dla $k, \ell \geqslant \max(n_a, n_b)$ zachodzi $|a_k - a_\ell| < \varepsilon/3$ oraz $|b_k - b_\ell| < \varepsilon/3$. Zgodnie z formułą powyżej musi istnieć $p \geqslant \max(n_a, n_b)$ takie, że $|a_p - b_p| \geqslant \varepsilon$. Załóżmy, że $a_p + \varepsilon \leqslant b_p$ (w drugą stronę będzie symetrycznie). Weźmy teraz dowolne $r \geqslant p$. Zachodza następujące nierówności:

$$a_p + \varepsilon \leqslant b_p$$
$$a_r - \varepsilon/3 < a_p < a_r + \varepsilon/3$$
$$b_r - \varepsilon/3 < b_p < b_r + \varepsilon/3$$

Prosto z nich wywnioskować, że

$$a_r + \varepsilon/3 < a_p + 2\varepsilon/3 \leqslant b_p - \varepsilon/3 < b_r$$

Czyli $\varepsilon/3$ rozdziela ciągi a, b poczynając od elementu a_p. □

Twierdzenie. Dla każdej liczby rzeczywistej $0 \leqslant x < 1$ istnieje ciąg $a \in 2^{\mathbb{N}}$ taki, że gdy ciąg b jest dany jako $b_n = \sum_{k=0}^{n} \frac{a_k}{2^{k+1}}$, to:

- b jest ciągiem Cauchy'ego,
- $[b]_{\simeq} = x$,
- $\forall_n \exists_{k>n}\, a_k = 0$.

Nazywamy go rozwinięciem x w szereg binarny.

I.4.3 Relacje, iloczyny kartezjańskie

Iloczyn kartezjański i jego własności. Pojęcia relacji, złożenia, relacji odwrotnej, funkcji. Podaj własności tych pojęć.

Definicja. Dla zbiorów x, y parę uporządkowaną definiujemy jako $(x, y) = \{\{x\}, \{x, y\}\}$.

Definicja. Dla zbiórów x, y iloczyn kartezjański definiujemy jako

$$x \times y = \{z \in \mathcal{P}(\mathcal{P}(x \cup y)) : \exists_{a \in x, b \in y} (a, b) = z\}.$$

Intuicyjnie jest to po prostu zbiór wszystkich możliwych sparowań elementów z dwóch zbiorów. To dziwne otypowanie jest dobrane tak, żeby było zgodne z definicją pary.

Propozycja (Podstawowe własności)**.**

- $x \times \varnothing = \varnothing$
- $x \times (y \cup z) = (x \times y) \cup (x \times z)$
- $x \times (y \cap z) = (x \times y) \cap (x \times z)$
- $x \times (y \setminus z) = (x \times y) \setminus (x \times z)$

Propozycja. Iloczyn kartezjański jest monotoniczny po każdej współrzędnej:

$$x \subset y \implies (x \times z) \subset (y \times z)$$
$$x \subset y \implies (z \times x) \subset (z \times y)$$

Definicja. Relacją nazywamy każdy podzbiór iloczynu $x \times y$. Ponadto dla relacji $R \subseteq A \times B, S \subseteq B \times C$ definiujemy

- złożenie: $S \circ R = \{(x, z) \in A \times C : \exists_{y \in B} (x, y) \in R \wedge (y, z) \in S\}$
- odwrotność: $R^{-1} = \{(y, x) \in B \times A : (x, y) \in R\}$

Propozycja (Własności)**.**

- $T \circ (S \circ R) = (T \circ S) \circ R$
- $(S \circ R)^{-1} = R^{-1} \circ S^{-1}$
- $(R \cup S)^{-1} = R^{-1} \cup S^{-1}$
- $(R \cap S)^{-1} = R^{-1} \cap S^{-1}$
- $(R^{-1})^{-1} = R$
- $(R \cup S) \circ T = (R \circ T) \cup (S \circ T)$
- $(R \cap S) \circ T \subseteq (R \circ T) \cap (S \circ T)$

Definicja. Identyczność to relacja $1_X \subseteq X \times X$ taka, że $z \in 1_X \iff \exists_{x \in X}\, z = (x, x)$.

Definicja. Niech $f \subseteq X \times Y$. f nazwiemy funkcją, gdy:

- $\forall_{x \in X} \exists_{y \in Y} (x, y) \in f$,
- $(x, y) \in f \wedge (x, y') \in f \implies y = y'$.

Piszemy wtedy $f : X \to Y$, zaś przez $f(x)$ oznaczamy ten unikalny $y \in Y$ taki, że $(x, y) \in f$.

Definicja. $Y^X = \{f : X \to Y\} = \{f \in \mathcal{P}(X \times Y) : f \text{ jest funkcją }\}$.

Propozycja. Złożenie funkcji jest funkcją.

Propozycja. $f : X \to Y, g : Y \to Z \implies g(f(x)) = (g \circ f)(x)$.

Propozycja (Ekstensjonalność)**.** $f : X \to Y, g : X \to Y \implies (f = g \iff \forall_{x \in X} f(x) = g(x))$.

Definicja. Iniekcja to funkcja $f : X \to Y$ taka, że $f(x) = f(x') \implies x = x'$.

Surjekcja to funkcja $f : X \to Y$ taka, że $\forall_{y \in Y} \exists_{x \in X} y = f(x)$.

Bijekcja to funkcja, która jest iniekcją i surjekcją.

Propozycja. Złożenie iniekcji jest iniekcją. Złożenie surjekcji jest surjekcją. Złożenie bijekcji jest bijekcją.

Twierdzenie. Jeśli $f : X \to Y$ jest bijekcją, to f^{-1} jest bijekcją. Ponadto $f \circ f^{-1} = 1_Y$.

Dowód. Zaczynamy od pokazania, że f^{-1} jest funkcją. f jest surjekcją, więc $\forall_{y \in Y} \exists_{x \in X} y = f(x)$. Wtedy $(x, y) \in f$, czyli $(y, x) \in f^{-1}$. Załóżmy teraz, że dla pewnego $y \in Y$ zachodzi $(y, x) \in f^{-1}$ oraz $(y, x') \in f^{-1}$. Wtedy $(x, y) \in f$ oraz $(x', y) \in f$. Ale f jest iniekcją, więc $x = x'$.

Teraz pokażemy, że f^{-1} to iniekcja. Niech więc $f^{-1}(y) = f^{-1}(y') = x$. Wtedy $(x, y) \in f$ oraz $(x, y') \in f$. Ale f jest funkcją, więc $y = y'$.

Pozostaje wykazać, że f^{-1} to surjekcja. Jako, że f jest funkcją to $\forall_{x \in X} \exists_{y \in Y} (x, y) \in f$. Ale wtedy $f^{-1}(y) = x$.

Aby pokazać drugą część lematu zauważmy, że identyczność jest funkcją. Wystarczy więc sprawdzić, czy $(f \circ f^{-1})(y) = y$ dla wszystkich $y \in Y$. Niech więc dla ustalonego y będzie $f^{-1}(y) = x$. Wtedy $f(x) = y$. Ostatecznie $(f \circ f^{-1})(y) = f(f^{-1}(y)) = f(x) = y$. □

I.4.4 Konstrukcja liczb naturalnych

Konstrukcja liczb naturalnych von Neumanna. Udowodnij twierdzenie o indukcji. Podaj przykładowe własności liczb naturalnych. Udowodnij zasadę minimum i maksimum.

Aksjomat (nieskończoności)**.**

$$\exists_x (\varnothing \in x \wedge (\forall_y y \in x \implies y \cup \{y\} \in x))$$

Każdy zbiór spełniający ten warunek nazywamy zbiorem induktywnym.

Lemat. Jeśli x to niepusty zbiór zbiorów induktywnych, to $\bigcap x$ też jest zbiorem induktywnym.

Dowód. Każdy element x jest zbiorem induktywnym, więc $\forall_{z \in x} \varnothing \in z$, czyli $\varnothing \in \bigcap x$. W celu wykazania drugiej własności weźmy $y \in \bigcap x$. Wtedy $\forall_{z \in x} y \in z$. Wszystkie elementy x to zbiory iduktywne, więc $\forall_{z \in x} y' \in z$. To daje $y' \in \bigcap x$. □

Twierdzenie. Istnieje unikalny najmniejszy ze względu na inkluzję zbiór induktywny.

Dowód. Z aksjomatu nieskończoności istnieje pewien zbiór induktywny x. Niech

$$y = \{z \in \mathcal{P}(x) : z \text{ jest zbiorem induktywnym}\}$$

Zbiór y jest niepusty, bo $x \in y$. Korzystając z poprzedniego lematu wiemy, że $\bigcap y$ jest zbiorem induktywnym. Twierdzimy, że jest on szukanym najmniejszym zbiorem induktywnym. W tym celu weźmy dowolny zbiór induktywny w. Mamy $w \in y$, czyli $\bigcap y \subseteq w$. $\bigcap y$ jest więc podzbiorem każdego zbioru induktywnego.

Jedyność wynika z tego, że gdyby istniały dwa takie zbiory x, y, to wtedy $x \subseteq y$ i $y \subseteq x$, czyli $x = y$. □

Definicja. Najmniejszy ze względu na inkluzję zbiór induktywny nazywamy zbiorem liczb naturalnych i oznaczamy go $\mathbb{N}$.

Przykład. Liczby naturalne (elementy zbioru liczb naturalnych) to kolejne elementy $0 = \varnothing, 1 = \{\varnothing\}, 2 = \{\varnothing, \{\varnothing\}\}, 3 = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}, \ldots$.

Twierdzenie (o indukcji matematycznej)**.** Dla dowolnego zbioru $Z \subseteq \mathbb{N}$ jeśli:

- $\varnothing \in Z$,
- $\forall_x x \in Z \implies x' = x \cup \{x\} \in Z$,

to $Z = \mathbb{N}$.

Dowód. Ustalmy zbiór Z, który spełnia założenia twierdzenia. Wtedy Z jest zbiorem induktywnym, więc z definicji $\mathbb{N}$ mamy $\mathbb{N} \subseteq Z$. Z założeń mamy $Z \subseteq \mathbb{N}$, więc zachodzi równość. □

Twierdzenie. Każdy element liczby naturalnej również jest liczbą naturalną.

Dowód. Dowodzimy przez indukcję. Niech $Z = \{n \in \mathbb{N} : \forall_{y \in n} y \in \mathbb{N}\}$. Sprawdzamy założenia twierdzenia o indukcji:

- Jako, że zbiór $\varnothing$ nie posiada elementów, to każdy jego element jest liczbą naturalną.
- Niech $n \in Z$. Rozważmy $n' = n \cup \{n\}$. Rozważmy $y \in n'$. Jeżeli $y \in n$, to z założenia jest liczbą naturalną. Jeżeli $y \in \{n\}$, to $y = n$, czyli jest liczbą naturalną.

W takim wypadku z twierdzenia o indukcji mamy $Z = \mathbb{N}$, co kończy dowód. □

Podobnie możemy dowieść także innych własności. Przykładowo:

Propozycja. Każda liczba naturalna jest albo zbiorem pustym, albo następnikiem pewnej liczby naturalnej.

Propozycja. Dla dowolnej liczby naturalnej n i dowolnego zbioru y jeśli $y \in n$, to $y \subseteq n$.

Propozycja. Dla liczb naturalnych m, n jeśli $m \subsetneq n$, to $m \in n$.

Propozycja. Dla liczb naturalnych m, n zachodzi $m \subseteq n$ lub $n \subseteq m$.

Propozycja. Dla liczb naturalnych m, n zachodzi jedna z trzech możliwości: $m \in n$, $n \in m$, $m = n$.

Definicja. Wprowadzamy liniowy porządek na liczbach naturalnych jako

$$m \leqslant n \iff m \subseteq n.$$

Twierdzenie (Zasada minimum)**.** Każdy niepusty podzbiór $\mathbb{N}$ zawiera element najmniejszy.

Dowód. Załóżmy nie wprost, że istnieje taki zbiór $X \subseteq \mathbb{N}$, że $X \neq \emptyset$ i X nie ma elementu najmniejszego. Rozważmy zbiór $Z = \{n \in \mathbb{N} : \forall_{k \leqslant n}\ k \in \mathbb{N} \setminus X\}$. Mamy $\emptyset \in Z$, bo inaczej $\emptyset \in X$ jest elementem najmniejszym X. Teraz załóżmy $n \in Z$. Dla $k \leqslant n'$ jeśli $k < n'$, to $k \leqslant n$ i mamy $k \in \mathbb{N} \setminus X$ z $n \in Z$. Jeśli $k = n'$, to musi być $k \in \mathbb{N} \setminus X$, bo inaczej n' jest elementem najmniejszym X. Z twierdzenia o indukcji mamy $Z = \mathbb{N}$. Zatem $X = \emptyset$, bo $X \cap Z = \emptyset$. Sprzeczność. □

Dowód. Dowodzimy poprzez indukcję. Niech

$$Z = \left\{ n \in \mathbb{N} : \forall_x \left(x \subseteq \mathbb{N} \land x \cap n \neq \varnothing \right) \implies \bigcap x \in x \right\}$$

To właśnie $\bigcap x$ będzie naszym elementem najmniejszym: z definicji jest podzbiorem wszystkich elementów x. Sprawdzamy założenia twierdzenia o indukcji:

- $\varnothing \in Z$, bo $x \cap \varnothing = \varnothing$ dla dowolnego x.
- Niech $n \in Z$ i niech $x \subseteq \mathbb{N}$ jest takie, że $x \cap n'$ jest niepuste. Jeśli $x \cap n$ jest niepuste, to $\bigcap x \in x$ z założenia indukcyjnego. Jeśli natomiast $x \cap n = \varnothing$, to mamy $x \cap n' = \{n\}$. W szczególności $n \in x$, czyli $\bigcap x \subseteq n$. Z drugiej strony dla każdego $z \in x$ mamy $n = z$ bądź $n \in z$ (trzecia możliwość jest wykluczona przez $x \cap n = \varnothing$). To oznacza, że $n \subseteq z$. Z tego mamy, że $n \subseteq \bigcup x$. W takim razie dostaliśmy, że $\bigcap x = n \in x$.

Zatem $Z = \mathbb{N}$. Aby dowieść nasze twierdzenie rozważamy niepusty $x \subseteq \mathbb{N}$. Bierzemy $n \in x$. Wtedy $n' \cap x \neq \varnothing$, więc z tego co dowiedliśmy indukcyjnie wynika, że $\bigcap x \in x$, czyli jest jego elementem najmniejszym. □

Twierdzenie (Zasada maksimum)**.** Jeśli x jest niepustym podzbiorem $\mathbb{N}$ ograniczonym z góry, czyli takim, że

$$\exists_{y \in \mathbb{N}} \forall_{z \in x} z \leqslant y,$$

to x ma element największy.

Dowód. Dla $x = \{\emptyset\}$ teza działa. Załóżmy dalej, że $x \neq \{\emptyset\}$. Wtedy każde ograniczenie górne x jest większe od $\emptyset$. Niech $Z = \{n \in \mathbb{N} : n \text{ jest ograniczeniem górnym } X\} \neq \emptyset$. Z Zasady minimum Z ma element najmniejszy y. Jeśli $y \notin X$, to y ma poprzednik (bo $y \neq \emptyset$), który też ogranicza X z góry. Sprzeczność z minimalnością y. □

Dowód. Dowodzimy poprzez indukcję. Niech Z będzie zbiorem ograniczeń górnych, dla których zachodzi nasza teza. Formalnie

$$Z = \left\{ n \in \mathbb{N} : \forall_x \left(x \neq \varnothing \land x \subseteq n \right) \implies \bigcup x \in x \right\}.$$

To właśnie $\bigcup x$ będzie naszym elementem największym: z definicji zawiera wszystkie elementy x. Sprawdzamy założenia twierdzenia o indukcji:

- $\varnothing \in Z$, bo $\varnothing$ nie posiada niepustych podzbiorów.
- Niech $n \in Z$ i niech $x \subseteq n'$ będzie niepuste. Jeśli $n \in x$, to $\bigcup x = n \in x$, bo pozostałe elementy n' są podzbiorami n. Jeśli zaś $n \notin x$, to $x \subseteq n$ i z indukcji mamy $\bigcup x \in x$.

Zatem $Z = \mathbb{N}$. Rozważmy $x \subseteq \mathbb{N}$ ograniczony z góry przez y. Wtedy $\forall_{z \in x}\ z \subseteq y$, więc $x \subseteq y'$. Z warunku z definicji Z mamy $\bigcup x \in x$. □

I.4.5 Relacje równoważności

Relacje równoważności i podziały zbiorów. Pokaż, że relacja równoważności jest środkiem do definiowania pojęć abstrakcyjnych.

Definicja. Relację $R \subseteq X \times X$ nazywamy relacją równoważności (abstrakcji) o polu X, gdy:

- $1_X \subseteq R$ (zwrotność),
- $R^{-1} \subseteq R$ (symetryczność),
- $R \circ R \subseteq R$ (przechodniość).

Definicja. Niech R będzie relacją równoważności o polu X. Klasą abstrakcji elementu $x \in X$ jest zbiór

$$[x]_R = \{y \in X : (x, y) \in R\}.$$

Definicja. Zbiór klas abstrakcji relacji R oznaczamy przez $X/_R$.

Twierdzenie. Niech R będzie relacją równoważności o polu X. Następujące warunki są równoważne:

1. $[x]_R \cap [y]_R \neq \varnothing$.
2. $[x]_R = [y]_R$.
3. $(x, y) \in R$.

Dowód. (1) $\implies$ (2): z powodu pełnej symetrii wystarczy pokazać $[x]_R \subseteq [y]_R$. Weźmy więc $p \in [x]_R$. Z założenia istnieje $z \in [x]_R \cap [y]_R$. Mamy więc $(y, z) \in R$, $(z, x) \in R$ (z symetrii), $(x, p) \in R$. Z przechodniości mamy $(y, p) \in R$.

(2) $\implies$ (3): ze zwrotności $y \in [y]_R$, więc z założenia $y \in [x]_R$, czyli $(x, y) \in R$.

(3) $\implies$ (2): $y \in [y]_R$ ze zwrotności, $y \in [x]_R$ z założenia. Mamy więc niepuste przecięcie. □

Intuicyjnie klasy abstrakcji dzielą zbiór na zbiory elementów „podobnych" do siebie. Poniżej formalizujemy tę intuicję.

Definicja. Niech $X \neq \varnothing$. Rodzinę $r \in \mathcal{P}(\mathcal{P}(X))$ nazywamy rozkładem zbioru X, gdy:

1. $\forall_{C \in r} C \neq \varnothing$,
2. $\bigcup r = X$,
3. $(C \in R \land D \in R \land C \neq D) \implies C \cap D = \varnothing$.

Lemat. Dla relacji równoważności R o polu X zbiór $X/_R$ jest rozkładem X.

Dowód. (1) Każda klasa jest niepusta, bo zawiera element, który ja wyznacza. (2). $\bigcup X/_R \subseteq X$, bo każda klasa jest pewnym podzbiorem X. W drugą stronę dla każdego $x \in X$ zachodzi $x \in [x]_R \in X/_R$. (3) Wynika z poprzedniego twierdzenia. □

Lemat. Dla rozkładu r zbioru X definiujemy relację $R_r \subseteq X \times X$ jako

$$(x, y) \in R \iff \exists_{C \in r}\, x \in C \land y \in C.$$

Wtedy

1. R_r jest równoważnością.
2. $X/_{R_r} = r$.

Dowód. (1) Zwrotność mamy, bo każdy $x \in X$ musi należeć do pewnego zbioru C rozkładu r. Symetryczność jest oczywista. Zostaje przechodniość. Niech $(x, y) \in R_r$ i $(y, z) \in R_r$. Istnieją więc $C, D \in r$ takie, że $x, y \in C$ oraz $y, z \in D$. Ale to oznacza, ze przecięcie C, D jest niepuste, więc $C = D$. To daje $(x, z) \in R_r$.

(2) Najpierw inkluzja w prawo. Niech $C \in {}^{X}/_{R_r}$. Istnieje więc $x \in X$ taki, że $C = [x]_{R_r}$. Niech $D \in r$ będzie taki, ze $x \in D$. Widzimy, ze wtedy $C = D$. Teraz w drugą stronę. Niech $C \in r$. C jest niepusty, więc istnieje $x \in C$. Widzimy, że $C = [x]_{R_r}$. □

I.4.6 Twierdzenie Cantora-Bernsteina

Wypowiedz twierdzenie Cantora-Bernsteina. Wypowiedz i udowodnij twierdzenie Cantora. Czy istnieje zbiór wszystkich zbiorów? Odpowiedź uzasadnij.

Definicja. $A \sim_m B$, gdy istnieje bijekcja $f : A \to B$.

Definicja. $A \leqslant_m B$, gdy istnieje iniekcja $f : A \to B$.

Definicja. $A <_m B$, gdy $A \leqslant_m B$ oraz nie zachodzi $A \sim_m B$.

Twierdzenie (Cantora-Bernsteina)**.** Jeśli $A \leqslant_m B$ oraz $B \leqslant_m A$, to $A \sim_m B$.

Twierdzenie (Cantora)**.** $A <_m \mathcal{P}(A)$ dla dowolnego zbioru A.

Dowód. Łatwo zauważyć, że $A \leqslant_m \mathcal{P}(A)$. Możemy na przykład wziąć iniekcję, która mapuje każdy $a \in A$ na jego singleton $\{a\}$. Załóżmy teraz nie wprost, że istnieje bijekcja $f : A \to \mathcal{P}(A)$. Niech $C = \{z \in A : z \notin f(z)\}$. Z surjektywności f istnieje $z_0 \in A$ takie, że $f(z_0) = C$. Mamy teraz 2 przypadki:

1. $z_0 \in f(z_0)$. Ale wtedy z definicji C mamy, że $z_0 \notin C$ i mamy sprzeczność.
2. $z_0 \notin f(z_0)$. Ale wtedy z definicji C mamy, ze $z_0 \in C$ i też mamy sprzeczność.

□

Wniosek. Nie istnieje zbiór wszystkich zbiorów.

Dowód. Załóżmy nie wprost, że A jest zbiorem wszystkich zbiorów. Wtedy mamy $\mathcal{P}(A) \subseteq A$, co oznacza, że $\mathcal{P}(A) \leqslant_m A$ (identyczność jest iniekcją). Z drugiej strony podobnie jak w poprzednim dowodzie argumentujemy, że $A \leqslant_m \mathcal{P}(A)$. Z twierdzenia Cantora-Bernsteina wiemy więc, że $A \sim_m \mathcal{P}(A)$, co jest sprzeczne z twierdzeniem Cantora. □

I.4.7 Lemat Kuratowskiego-Zorna

Podaj wypowiedź lematu Kuratowskiego-Zorna i przykłady jego zastosowania.

Definicja. Porządek to para (X, R), gdzie R to zwrotna, przechodnia i antysymetryczna relacja na zbiorze X (podzbiór $X \times X$). Jeśli R jest dodatkowo spójna (czyli $(x, y) \in R$ lub $(y, x) \in R$ dla każdej pary (x, y)), to porządek nazywamy liniowym.

Definicja. a jest maksymalnym elementem porządku $(X, \leqslant)$, gdy $\forall_{x \in X}\, a \leqslant x \implies a = x$. Analogicznie definiujemy element minimalny.

Definicja. a jest największym elementem porządku $(X, \leqslant)$, gdy $\forall_{x \in X}\, x \leqslant a$. Analogicznie definiujemy element najmniejszy.

Definicja. Dla porządku $(X, \leqslant)$ oraz $A \subseteq X, b \in X$ mówimy, że b jest majorantą A, gdy $\forall_{a \in A}\, a \leqslant b$. Analogicznie definiujemy minorantę.

Definicja. Łańcuch w porządku (X, R) to $L \subseteq X$ taki, że $(L, R \cap L^2)$ jest porządkiem liniowym.

Twierdzenie (Lemat Kuratowskiego-Zorna)**.** Jeśli w porządku $(X, \leqslant)$ każdy łańcuch ma majorantę (odpowiednio minorantę), to istnieje w nim element maksymalny (odpowiednio minimalny).

Podamy teraz dwa przykłady zastosowania tego lematu.

Twierdzenie. Jeśli A, B są niepustymi zbiorami, to istnieje iniekcja z A w B lub z B w A.

Dowód. Niech $\mathcal{H} = \{f \subseteq A \times B : f \text{ to częściowa iniekcja}\}$. Rozważmy porządek $(\mathcal{H}, \subseteq)$. Sprawdzamy dla niego założenia lematu Kuratowskiego-Zorna. Niech $\mathcal{H}_0 \subseteq \mathcal{H}$ będzie łańcuchem. Zauważmy, że majorantą $\mathcal{H}_0$ jest $\bigcup \mathcal{H}_0$:

- $\forall_{f \in \mathcal{H}_0}\, f \subseteq \bigcup \mathcal{H}_0$ z definicji sumy.
- $\bigcup \mathcal{H}_0 \in \mathcal{H}$, bo gdyby $\bigcup \mathcal{H}_0$ nie było częściową iniekcją, to albo jeden element A byłby mapowany dwukrotnie, albo dwa różne elementy A byłyby mapowane na ten sam element B. W obu przypadkach $\mathcal{H}_0$ nie mogłoby być łańcuchem.

W takim razie z LKZ istnieje w tym porządku element maksymalny $f^{\text{MAX}} \in \mathcal{H}$. Twierdzimy, że f^{MAX} jest funkcją z A w B lub $(f^{\text{MAX}})^{-1}$ jest funkcją z B w A. Wtedy z definicji $\mathcal{H}$ musi już być iniekcją, więc mamy tezę. Załóżmy więc nie wprost, że istnieje $a_0 \in A$ takie, że $\forall_{b \in B}\, (a_0, b) \notin f^{\text{MAX}}$ oraz $b_0 \in B$ takie, że $\forall_{a \in A}\, (a, b_0) \notin f^{\text{MAX}}$. Ale zauważmy, że wtedy $f^{\text{MAX}} \cup \{(a_0, b_0)\} \in \mathcal{H}$, więc f^{MAX} nie mogło być elementem maksymalnym. □

Twierdzenie. Każdy porządek da się rozszerzyć do liniowego.

Dowód. Niech (X, R_0) będzie pewnym porządkiem. Rozważmy zbiór

$$\mathcal{H} = \left\{R \subseteq X^2 : R_0 \subseteq R \text{ i } (X, R) \text{ jest porządkiem}\right\}.$$

Uporządkujmy go za pomocą inkluzji. Sprawdzamy założenia LKZ. Niech $\mathcal{H}_0 \subseteq \mathcal{H}$ będzie łańcuchem. Zauważmy, że jego majorantą jest $\bigcup \mathcal{H}_0$:

- dla $R \in \mathcal{H}_0$ mamy $R \subseteq \bigcup \mathcal{H}_0$.
- $\bigcup \mathcal{H}_0 \in \mathcal{H}$, bo $R_0 \subseteq \bigcup \mathcal{H}_0$ oraz $(X, \bigcup \mathcal{H}_0)$ jest porządkiem.

W takim razie z LKZ mamy element maksymalny $R^{\text{MAX}} \in \mathcal{H}$. Załóżmy nie wprost, że R^{MAX} nie jest porządkiem liniowym. Wtedy istnieją $a, b \in X$ takie, że $(a, b) \notin R^{\text{MAX}}$ oraz $(b, a) \notin R^{\text{MAX}}$. Niech R_1 będzie domknięciem przechodnim $R^{\text{MAX}} \cup \{(a, b)\}$. Oznacza to, że $R_1 = R^{\text{MAX}} \cup S$, gdzie $S = \left\{(x, y) \in X^2 : (x, a) \in R^{\text{MAX}} \wedge (b, y) \in R^{\text{MAX}}\right\}$. Widzimy, że $R^{\text{MAX}} \subsetneq R_1$. Wykazanie, że $R_1 \in \mathcal{H}$ da więc sprzeczność z maksymalnością R^{MAX}. Sprawdźmy, czy rzeczywiście tak jest.

- $R_0 \subseteq R_1$, bo $R_0 \subseteq R^{\text{MAX}} \subseteq R_1$.
- R_1 jest zwrotna, bo R^{MAX} jest zwrotna.
- R_1 jest przechodnia, bo zdefiniowaliśmy ją jako domknięcie przechodnie.
- R_1 jest antysymetryczna: załóżmy nie wprost, że istnieją $x \neq y$ takie, że $(x, y) \in R_1$ i $(y, x) \in R_1$. Mamy teraz 3 opcje. Jeśli $(x, y) \in R^{\text{MAX}}$ oraz $(y, x) \in R^{\text{MAX}}$, to R^{MAX} nie jest antysymetryczna i mamy sprzeczność. Jeśli $(x, y) \in S$, zaś $(y, x) \in R^{\text{MAX}}$, to wtedy $(b, y), (y, x), (x, a) \in R^{\text{MAX}}$ i z przechodniości $(b, a) \in R^{\text{MAX}}$ – sprzeczność. Ostatecznie jeśli $(x, y) \in S$ oraz $(y, x) \in S$, to $(b, y), (y, a) \in R^{\text{MAX}}$ i ponownie z przechodniości mamy $(b, a) \in R^{\text{MAX}}$ – sprzeczność.

□

Wniosek. Każdy zbiór da się liniowo uporządkować, bo możemy zacząć od $(X, 1_X)$.

I.4.8 Liczby całkowite i wymierne

Wykonaj konstrukcję liczb całkowitych. Wprowadź działania na liczbach całkowitych. Podaj konstrukcję liczb wymiernych i działania na nich.

Definicja. Niech $\approx$ będzie relacją na $\mathbb{N} \times \mathbb{N}$ określoną następująco

$$(n, k) \approx (p, q) \iff n + q = p + k.$$

Propozycja. $\approx$ jest relacją równoważności

Definicja. Liczby całkowite definiujemy jako $\mathbb{Z} = {}^{\mathbb{N} \times \mathbb{N}}\!/\!_{\approx}$.

Definicja. Element zero $0 \in \mathbb{Z}$ to $[(0, 0)]_\approx$. Element przeciwny do $x = [(n, k)]_\approx$ to $-x = [(k, n)]_\approx$.

Definicja. Działania na $\mathbb{Z}$ definiujemy następująco:

- dodawanie: $[(n, k)]_\approx + [(p, q)]_\approx = [(n + p, k + q)]_\approx$.
- mnożenie $[(n, k)]_\approx \cdot [(p, q)]_\approx = [(n \cdot p + k \cdot q, n \cdot q + k \cdot p)]_\approx$.
- odejmowanie: $x - y = x + (-y)$.

Definicja. Niech $\mathbb{Z}^* = \mathbb{Z} \setminus \{0\}$. Określamy relację $\sim$ na zbiorze $\mathbb{Z} \times \mathbb{Z}^*$ jako:

$$(a, b) \sim (c, d) \iff a \cdot d = c \cdot b.$$

Propozycja. $\sim$ jest relacją równoważności

Definicja. Liczby wymierne definiujemy jako $\mathbb{Q} = {}^{\mathbb{Z} \times \mathbb{Z}}\!/\!_{\sim}$.

Definicja. Element zero $0 \in \mathbb{Q}$ to $[(0, 1)]_\sim$. Element przeciwny do $x = [(a, b)]_\sim$ to $-x = [(-a, b)]_\sim$.

Definicja. Działania na $\mathbb{Q}$ definiujemy następująco:

- dodawanie: $[(a, b)]_\sim + [(c, d)]_\sim = [(ad + bc, bd)]_\sim$.
- odejmowanie: $[(a, b)]_\sim - [(c, d)]_\sim = [(ad - bc, bd)]_\sim$.
- mnożenie: $[(a, b)]_\sim \cdot [(c, d)]_\sim = [(ac, bd)]_\sim$.
- dzielenie: $[(a, b)]_\sim : [(c, d)]_\sim = [(ad, bc)]_\sim$, gdy $[c, d]_\sim \neq 0$.

I.4.9 Liczność zbiorów

Podaj definicje równoliczności zbiorów oraz zbiorów przeliczalnych i nieprzeliczalnych. Udowodnij równoliczność zbiorów $A^{B^C} \sim_m A^{B \times C}$ oraz $(A \times B)^C \sim_m A^C \times B^C$.

Definicja. Zbiory A, B są równoliczne, gdy istnieje bijekcja $f : A \to B$. Piszemy wtedy $A \sim_m B$.

Definicja. Zbiór A jest przeliczalny, jeśli istnieje $N_0 \subseteq \mathbb{N}$ takie, że $A \sim_m N_0$. Zbiór A jest nieprzeliczalny, jeśli nie jest przeliczalny.

Twierdzenie. $(A^B)^C \sim_m A^{B \times C}$.

Dowód. Zdefiniujmy funkcję $\alpha : (A^B)^C \to A^{B \times C}$ w ten sposób, że dla funkcji $x \in (A^B)^C$ będzie $\alpha(x)(b, c) = x(c)(b)$.

1. α jest iniekcją.

   Niech $x_1 \neq x_2 \in (A^B)^C$. To, że funkcje są różne oznacza tyle, że istnieje $c_0 \in C$ takie, że $x_1(c_0) \neq x_2(c_0)$. To dalej są funkcje, więc dalej mamy, że istnieje $b_0 \in B$ takie, że $x_1(c_0)(b_0) \neq x_2(c_0)(b_0)$. Ale to znaczy, że mamy b_0, c_0 takie, że $\alpha(x_1)(b_0, c_0) \neq \alpha(x_2)(b_0, c_0)$, czyli $\alpha(x_1) \neq \alpha(x_2)$.

2. α jest surjekcją.

   Niech $y \in A^{B\times C}$. Niech $x_0 \in (A^B)^C$ jest taką funkcją, że dla wszystkich $b \in B, c \in C$ zachodzi $x_0(c)(b) = y(b, c)$. Wtedy mamy $\alpha(x_0)(b, c) = x_0(c)(b) = y(b, c)$, więc $\alpha(x_0) = y$.

□

Twierdzenie. $(A \times B)^C \sim_m A^C \times B^C$.

Dowód. Zdefiniujmy funkcję $\beta : (A \times B)^C \to A^C \times B^C$ w ten sposób, że $\beta(x) = (l \circ x, r \circ x)$, gdzie $l(a, b) = a$, zaś $r(a, b) = b$.

1. β jest iniekcją.

   Niech $x_1 \neq x_2 \in (A \times B)^C$. Wtedy istnieje $c_0 \in C$ takie, że $x_1(c_0) \neq x_2(c_0)$. Ale to w szczególności oznacza, że $l(x_1(c_0)) \neq l(x_2(c_0))$ lub $r(x_1(c_0)) \neq r(x_2(c_0))$. Z tego dostajemy, że

$$\beta(x_1) = (l \circ x_1, r \circ x_1) \neq (l \circ x_2, r \circ x_2) = \beta(x_2).$$

2. β jest surjekcją.

   Niech $g \in A^C$ oraz $h \in B^C$. Niech $x_0 \in (A \times B)^C$ to funkcją taką, że dla wszystkich $c \in C$ zachodzi $x_0(c) = (g(c), h(c))$. Wtedy $\beta(x_0) = (l \circ x_0, r \circ x_0) = (g, h)$.

□

I.4.10 Twierdzenie Zermelo

Podaj definicję dobrego porządku. Udowodnij zasadę indukcji pozaskończonej. Wypowiedz twierdzenie Zermelo.

Definicja. Porządek $(X, \leqslant)$ jest dobry, gdy każdy niepusty $Z \subseteq X$ ma w nim element najmniejszy.

Lemat. Dobry porządek zawsze jest liniowy.

Dowód. Bierzemy Z jako zbiór dwuelementowy $\{x, y\}$. □

Definicja (Indukcja pozaskończona)**.** Mówimy, że w liniowym porządku $(X, \leqslant)$ obowiązuje zasada indukcji, gdy dla każdego $\varnothing \neq Z \subseteq X$ jeśli $\forall_{x\in X}\ (\{y \in X : y < x\} \subseteq Z \implies x \in Z)$, to wtedy $Z = X$.

Twierdzenie. W dobrym porządku obowiązuje zasada indukcji.

Dowód. Niech $(X, \leqslant)$ będzie dobrym porządkiem. Załóżmy nie wprost, że istnieje $\varnothing \neq Z \subsetneq X$ taki, że $\{y \in X : y < x\} \subseteq Z \implies x \in Z$ dla każdego $x \in X$. $X \backslash Z$ jest niepusty, więc z definicji dobrego porządku ma element najmniejszy x_0. To oznacza, że dla wszystkich y mniejszych od x_0 zachodzi $y \notin X \setminus Z$. Czyli $\{y \in X : y < x_0\} \subseteq Z$. Z początkowego założenia mamy więc, że $x_0 \in Z$, czyli sprzeczność. □

Twierdzenie (Zermelo)**.** Każdy zbiór da się dobrze uporządkować.

I.4.11 Liczby porządkowe

Podaj definicję liczby porządkowych von Neumanna oraz ich własności. Udowodnij antynomię Burali-Forti.

Aksjomat (regularności, dobrego ufundowania)**.** $\forall_x (x \neq \varnothing \implies \exists_y (y \in x \wedge y \cap x = \varnothing))$

Aksjomat regularności nie pozwala na $x \in x$, ani $x \in y \in x$, ani żadne podobne zagnieżdżone wyrażenie. Wynika to z faktu, że gdyby było $x \in x$, to $\{x\}$ nie spełnia aksjomatu regularności, a gdyby było $x \in y \in x$, to $\{x, y\}$ nie spełnia tego aksjomatu.

Definicja. Zbiór X jest liczbą porządkową, gdy:

- $\forall_{x,y \in X}\, x \in y \vee y \in x \vee x = y$.
- $\forall_{x \in X}\, x \subseteq X$.

Przykład. $0, 1, 2, 3, \ldots, \mathbb{N} =: \omega, \mathbb{N} \cup \{\mathbb{N}\} =: \omega + 1, \mathbb{N} \cup \{\mathbb{N}\} \cup \{\mathbb{N} \cup \{\mathbb{N}\}\} =: \omega + 2, \ldots$

Propozycja. Zbiór pusty jest elementem każdej niepustej liczby porządkowej.

Dowód. Załóżmy, że X jest niepustą liczbą porządkową. Istnieje $x \in X$ takie, że $x \cap X = \emptyset$. Jest $x \subseteq X$, więc $x = \emptyset$. ☐

Lemat. Każdy element liczby porządkowej jest liczbą porządkową.

Dowód. Niech x będzie liczbą porządkową Jeśli $x = \varnothing$, to teza zachodzi. W przeciwnym wypadku niech $y \in x$. Z definicji liczby porządkowej mamy $y \subseteq x$.

- Czyli dla $a, b \in y$ zachodzi $a, b \in x$, więc $a \in b$ lub $b \in a$ lub $a = b$.
- Niech $a \in y$. Jako, że $y \subseteq x$, to $a \in x$. Załóżmy, że $a \not\subseteq y$, czyli istnieje $b \in a$ takie, że $b \notin y$. Mamy $b \in x$ (bo $a \subseteq x$), więc z pierwszej cechy liczb porządkowych $b = y$ bądź $y \in b$. W pierwszym przypadku mamy $y \in a \in y$, a w drugim $b \in a \in y \in b$. Oba te przypadki są sprzeczne z aksjomatem regularności.

☐

Wniosek. Dla liczby porządkowej X i jej elementów $x, y \in X$ jeżeli $x \in y$, to $x \subseteq y$.

Dowód. y jest liczbą porządkową jako element liczby porządkowej, zatem z $x \in y$ wynika $x \subseteq y$. ☐

Wniosek. Dla liczby porządkowej X i jej elementów $x, y \in X$ jeżeli $x \subsetneq y$, to $x \in y$.

Dowód. Z definicji liczby porządkowej mamy $x \in y$ lub $y \in x$. Załóżmy nie wprost, że $y \in x$. x jest liczbą porządkową, więc $y \subseteq x \subsetneq y$. Sprzeczność. ☐

Twierdzenie. Zbiór będący liczbą porządkową jest dobrze uporządkowany inkluzją.

Dowód. Ustalmy liczbę porządkową X i niepusty zbiór $A \subseteq X$. Z aksjomatu regularności istnieje $a \in A$ takie, że $a \cap A = \emptyset$. Ustalmy dowolne $b \in A$ różne od a. Wtedy $a \in b$ lub $b \in a$. To drugie daje $b \in a \cap A$, więc $a \in b$. Zatem $a \subseteq b$. Z tego wynika, że a jest elementem najmniejszym A. ☐

Definicja. Przedziałem początkowym w porządku $(X, \leqslant)$ nazywamy zbiór $Y \subseteq X$ taki, że jeśli $x \in Y$ i $y \leqslant x$ dla pewnego $y \in X$, to $y \in Y$.

Uwaga. W dobrym porządku $(X, \leqslant)$ każdy właściwy przedział początkowy jest postaci $\{x \in X : x < x_0\}$ dla pewnego $x_0 \in X$. x_0 jest minimum zbioru ograniczeń górnych tego przedziału początkowego.

Lemat. Każdy przedział początkowy liczby porządkowej jest liczbą porządkową.

Dowód. Niech X będzie liczbą porządkową a Y jej przedziałem początkowym. Dla wszystkich $a, b \in Y$ mamy $a = b$ lub $a \in b$ lub $b \in a$, bo $a, b \in X$.

Niech $x \in Y$. Wtedy $x \subseteq X$. Ustalmy dowolne $z \in x$. Wtedy $z \in X$ i $z \subseteq x$, więc z definicji przedziału początkowego $z \in Y$. Zatem $x \subseteq Y$. □

Lemat. Każdy właściwy przedział początkowy liczby porządkowej jest jej elementem.

Dowód. Niech A będzie liczbą porządkową i niech C będzie jej właściwym przedziałem początkowym. Wiemy, że $C = \{x \in A : x \subsetneq x_0\}$ dla pewnego $x_0 \in A$. Dla $x \subsetneq x_0$ mamy $x \in x_0$, więc $C \subseteq x_0$. Do tego dla każdego $x \in x_0$ mamy $x \in A$ (bo $x_0 \subseteq A$) oraz $x \subsetneq x_0$. Zatem $x_0 \subseteq C$. To pokazuje, że $C = x_0 \in A$. □

Twierdzenie. Dla dwóch liczb porządkowych jedna jest przedziałem początkowym drugiej.

Dowód. Rozważmy dwie liczby porządkowe A, B. Niech $C = A \cap B$. Pokażemy, że C jest przedziałem początkowym zarówno A, jak i B. Ustalmy $x \in C$. Mamy $x \subseteq A, B$. Dla dowolnego $y \in A$ takiego, że $y \subsetneq x$ mamy $y \in x$, więc $y \in B$, czyli $y \in C$. Zatem C jest przedziałem początkowym A. Analogicznie pokazujemy, że C jest przedziałem początkowym B.

Wiemy, że jeśli $C \neq A$, $C \in A$ i analogicznie dla B. Teraz rozważamy przypadki.

- Jeśli $C = A$ i $C = B$, to mamy $A = B$ i teza zachodzi.
- Jeśli $C = A$ i $C \in B$, to A jest przedziałem początkowym B.
- Jeśli $C = B$ i $C \in A$, to B jest przedziałem początkowym A.
- Jeśli $C \in A$ i $C \in B$, to $C \in A \cap B = C$, co daje sprzeczność z aksjomatem regularności.

□

Twierdzenie (Dychotomia Burkina-Faso)**.** Nie istnieje zbiór wszystkich liczb porządkowych.

Dowód. Załóżmy nie wprost, że X jest takim zbiorem. Pokażemy, że X musi być liczbą porządkową. To będzie oznaczało, że $X \in X$, co jest sprzeczne z aksjomatem o regularności.

- Niech $x, y \in X$ będą różnymi liczbami porządkowymi. Z poprzedniego twierdzenia x jest właściwym przedziałem początkowym y lub y jest właściwym przedziałem początkowym x, czyli $x \in y$ lub $y \in x$.
- Niech $x \in X$. x jest pewną liczbą porządkową. Wiemy, że wszystkie elementy x są liczbami porządkowymi, więc $x \subseteq X$.

□

Twierdzenie. Każdy zbiór dobrze uporządkowany jest izomorficzny z pewną liczbą porządkową (czyli istnieje bijekcja zachowująca porządek).

I.5 Metody Probabilistyczne Informatyki

I.5.1 Łańcuchy Markowa

Łańcuchy Markowa na przykładzie analizy randomizowanego algorytmu dla problemu 2-SAT.

Definicja. Proces stochastyczny to rodzina zmiennych losowych $\{X_t : t \in T\}$ indeksowana zbiorem T (zazwyczaj interpretowanym jako czas). X_t nazywamy stanem procesu w chwili t. Dla co najwyżej przeliczalnego T (zwykle $T = \mathbb{N}$) mówimy, że proces jest z czasem dyskretnym.

Definicja. Proces stochastyczny z czasem dyskretnym (i $T = \mathbb{N}$) jest łańcuchem Markowa, jeśli dla każdego $t \in \mathbb{N}$ i ciągu $(a_0, a_1, \ldots, a_{t-1}, x)$ takiego, że

$$\mathbb{P}\left(\bigcap_{i=0}^{t-1} X_i = a_i \cap X_t = x\right) > 0$$

zachodzi

$$\mathbb{P}\left(X_{t+1} = y \mid \bigcap_{i=0}^{t-1} X_i = a_i \cap X_t = x\right) = \mathbb{P}\left(X_{t+1} = y \mid X_t = x\right)$$

dla każdego y. Jest to automat, który zmienia stany z danym prawdopodobieństwem, ale zależnie tylko od poprzedniego stanu. Zbiór stanów będziemy oznaczać przez S.

Twierdzenie (Randomizowany 2-SAT)**.** Rozważmy formułę postaci $C = \bigwedge_i x_i \vee y_i$, gdzie x, y są jednymi z n zmiennych (lub ich zaprzeczeniami). Zastosujemy następujący algorytm: wybieramy losowe wartościowanie, dopóki formuła jest niespełniona wybieramy losową niespełnioną klauzulę i jeden z jej literałów, a następnie odwracamy jego wartościowanie. Powtarzamy taką operację maksymalnie $m \cdot 2n^2$ razy, gdzie m jest pewnym parametrem. Jeśli formuła jest spełniona zwracamy wartościowanie, inaczej stwierdzamy niespełnialność.

Taki algorytm zwraca poprawną odpowiedź z prawdopodobieństwem co najmniej $1 - \frac{1}{2^m}$.

Dowód. Algorytm może się pomylić tylko, jeśli formuła jest spełnialna, a on stwierdzi niespełnialność. Niech więc będzie spełnialna, a S będzie pewnym spełniającym ją wartościowaniem. Definiujemy X_i jako liczbę zmiennych wartościowanych tak samo jak w S w i-tym kroku algorytmu.

Zachodzi $\mathbb{P}\left(X_{i+1} = 1 \mid X_i = 0\right) = 1$, a dla $j > 0$ jest

$$\mathbb{P}\left(X_{i+1} = j + 1 \mid X_i = j\right) \geqslant \frac{1}{2},$$

$$\mathbb{P}\left(X_{i+1} = j - 1 \mid X_i = j\right) \leqslant \frac{1}{2},$$

bo w poprawianej klauzuli albo obie zmienne nie zgadzają się z S i na pewno jedną zmienimy poprawnie, albo jedna się zgadza i mamy $\frac{1}{2}$ na zmianę poprawnej. Taki proces stochastyczny nie jest łańcuchem Markowa. Zamiast niego analizujemy proces o stanach Y_i takich, że $\mathbb{P}\left(Y_{i+1} = j + 1 \mid Y_i = j\right) = \frac{1}{2}$ oraz $Y_0 = X_0$. To już jest łańcuch Markowa.

Niech h_j oznacza oczekiwaną liczbę kroków potrzebnych do osiągnięcia wartości n zaczynając od j. Mamy $h_n = 0$, $h_0 = h_1 + 1$ oraz $h_j = \frac{h_{j+1}}{2} + \frac{h_{j-1}}{2} + 1$. Można pokazać indukcyjnie, że $h_j = h_{j+1} + 2j + 1$: działa dla $j = 0$, a dalej mamy

$$h_{j+1} = 2h_j - h_{j-1} - 2 = 2h_j - (h_j + 2(j-1) + 1) - 1 = h_j - 2j - 1.$$

Z tego wynika, że $h_0 = h_1 + 1 = h_2 + 1 + 3 = \ldots = \sum_{i=0}^{n-1}(2i+1) = n^2$.

Podzielmy wykonanie algorytmu na segmenty rozmiaru $2n^2$. Niech Z będzie liczbą kroków potrzebną do otrzymania rozwiązania (licząc od początku segmentu). Mamy $\mathbb{E}[Z] \leqslant n^2$. Z nierówności Markowa zachodzi

$$\mathbb{P}\left(Z > 2n^2\right) \leqslant \frac{n^2}{2n^2} = \frac{1}{2}.$$

Zatem prawdopodobieństwo, że algorytm nie zwróci poprawnego wartościowania po m segmentach jest ograniczone przez $\frac{1}{2^m}$. □

I.5.2 Kule i urny

Problem kul i urn ze wzmocnionym feedbackiem (jako ilustracja zastosowania rozkładu wykładniczego).

Definicja. Dla parametru $\lambda > 0$ rozkład o funkcji gęstości $f(x) = \begin{cases} \lambda e^{-\lambda x} & x > 0 \\ 0 & x \leqslant 0 \end{cases}$ nazywamy rozkładem wykładniczym. Prawdopodobieństwo spada wykładniczo wraz ze wzrostem x. Mamy

$$\int_{-\infty}^{\infty} f(x)\,\mathrm{d}x = \int_0^{\infty} \lambda e^{-\lambda x}\,\mathrm{d}x = \lim_{R\to\infty} \int_0^{R} \lambda e^{-\lambda x}\,\mathrm{d}x = \lim_{R\to\infty} -e^{-\lambda x}\Big|_0^R = \lim_{R\to\infty} -e^{-\lambda R} + 1 = 1,$$

więc jest to poprawna gęstość.

Propozycja. Dystrybuanta, wartość oczekiwana i wariancja zmiennej X o rozkładzie wykładniczym z parametrem λ to

$$F(t) = \begin{cases} 1 - e^{-\lambda t} & t > 0 \\ 0 & t \leqslant 0 \end{cases},$$

$$\mathbb{E}[X] = \frac{1}{\lambda},$$

$$\mathrm{Var}(X) = \frac{1}{\lambda^2}.$$

Dowód.

$$F(t) = \int_{-\infty}^{t} f(x)\,\mathrm{d}x = -e^{-\lambda x}\Big|_0^t = 1 - e^{-\lambda t},$$

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x)\,\mathrm{d}x = \int_0^{\infty} \lambda x e^{-\lambda x}\,\mathrm{d}x = -xe^{-\lambda x}\Big|_0^{\infty} - \int_0^{\infty} -e^{-\lambda x}\,\mathrm{d}x = -\frac{1}{\lambda} e^{-\lambda x}\Big|_0^{\infty} = \frac{1}{\lambda},$$

$$\mathbb{E}\left[X^2\right] = \int_{-\infty}^{\infty} x^2 f(x)\,\mathrm{d}x = \ldots = \frac{2}{\lambda^2},$$

a więc $\mathrm{Var}(X) = \frac{1}{\lambda^2}$. □

Lemat. Rozkład wykładniczy ma własność „bez pamięci”, czyli dla zmiennej $X \sim \mathrm{Exp}(\lambda)$ i $s, t > 0$ mamy

$$\mathbb{P}(X > s + t \mid X > t) = \mathbb{P}(X > s).$$

Dowód.

$$\begin{aligned} \mathbb{P}(X > s + t \mid X > t) &= \frac{\mathbb{P}(X > s + t)}{\mathbb{P}(X > t)} = \frac{1 - F_X(s + t)}{1 - F_X(t)} = \frac{e^{-\lambda(s+t)}}{e^{-\lambda t}} \\ &= e^{-\lambda s} = 1 - F_X(s) = \mathbb{P}(X > s). \end{aligned}$$

□

Lemat. Niech X, Y będą niezależnymi zmiennymi o rozkładzie wykładniczym z parametrami λ i μ. Zachodzi $\min(X, Y) \sim \text{Exp}(\lambda + \mu)$. Dodatkowo $\mathbb{P}(X < Y) = \frac{\lambda}{\lambda+\mu}$.

Dowód.

$$\mathbb{P}(\min(X, Y) > t) = \mathbb{P}(X > t \cap Y > t) = \mathbb{P}(X > t)\,\mathbb{P}(Y > t) = e^{-\lambda t} e^{-\mu t} = e^{-(\lambda+\mu)t}.$$

$$\mathbb{P}(X < Y) = \mathbb{P}\left((X, Y) \in \left\{(x, y) \in \mathbb{R}^2 : x < y\right\}\right) = \int_0^\infty \int_0^y f_{X,Y}(x, y)\,\mathrm{d}x\,\mathrm{d}y$$
$$= \int_0^\infty \int_0^y \lambda e^{-\lambda x} \mu e^{-\mu y}\,\mathrm{d}x\,\mathrm{d}y = \int_0^\infty \mu e^{-\mu y}\left(1 - e^{-\lambda y}\right)\mathrm{d}y = \int_0^\infty \mu e^{-\mu y}\,\mathrm{d}y - \int_0^\infty \mu e^{-(\lambda+\mu)y}\,\mathrm{d}y$$
$$= 1 - \left[-\frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)y}\right]\Bigg|_0^\infty = 1 - \frac{\mu}{\lambda+\mu} = \frac{\lambda}{\lambda+\mu}.$$

□

Twierdzenie (Feedback ze wzmocnieniem)**.** Niech X_n, Y_n będą liczbą kul wrzuconych odpowiednio do czerwonej i niebieskiej urny w momencie czasu n. Zaczynamy z $X_0 = x_0, Y_0 = y_0$. Jeśli $X = x$ i $Y = y$, to niech prawdopodobieństwo wybrania kolejnej kuli do czerwonej urny wynosi $\frac{x^p}{x^p+y^p}$ dla ustalonego parametru p.

Dla każdych $x_0, y_0 \in \mathbb{N}_1$ i $p > 1$ z prawdopodobieństwem 1 istnieje $c \in \mathbb{N}$ takie, że jedna z urn otrzyma mniej niż c kul (przy rzucaniu w nieskończoność). Innymi słowy

$$\mathbb{P}\left(\lim_{n\to\infty} X_n = \infty \cap \lim_{n\to\infty} Y_n = \infty\right) = 0.$$

Dowód. Niech T_x będzie zmienną modelującą czas oczekiwania na $(x+1)$-szą kulę w urnie czerwonej mając w niej x kul. W naszym modelu $T_x \sim \text{Exp}(x^p)$, czyli zmienna T_x formalnie nie ma nic wspólnego z kulami i urnami. Podobnie definiujemy U_y. Wcześniej widzieliśmy, że zachodzi

$$\mathbb{P}(T_x < U_y) = \frac{x^p}{x^p + y^p},$$

a więc te zmienne faktycznie dobrze modelują nasz eksperyment – szansa na „skończenie” T_x przed U_y jest taka, jak wrzucenia kuli do urny czerwonej. W momencie skończenia T_x możemy „zrestartować” U_y (czyli skorzystać z własności bez pamięci rozkładu wykładniczego) i zacząć T_{x+1} – odpowiednia nierówność będzie dalej zachodzić. Oczywiście wszystko działa tak samo, gdy skończy się U_y.

Definiujemy

$$F = \sum_{j=x_0}^{\infty} T_j,\ G = \sum_{j=y_0}^{\infty} U_j.$$

Te wartości oznaczają momenty, w których liczba kul wrzuconych do danej urny staje się nieograniczona. Mamy

$$\mathbb{E}[F] = \sum_{j=x_0}^{\infty} \mathbb{E}[T_j] = \sum_{j=x_0}^{\infty} \frac{1}{j^p} < \infty.$$

To samo zachodzi dla G. Zatem $\mathbb{P}(F < \infty) = \mathbb{P}(G < \infty) = 1$ i te liczby są dobrze zdefiniowane z prawdopodobieństwem 1 (ograniczona wartość oczekiwana daje zerowe prawdopodobieństwo tego, że zmienna jest nieograniczona). Mamy $\mathbb{P}(F \neq G) = 1$ (tak zachowują się każde niezależne zmienne ciągłe), czyli $\mathbb{P}(F < G \cup F > G) = 1$. Załóżmy, że $F < G$. Mamy zatem (dla pewnego t)

$$\sum_{j=1}^{t} U_j < F < \sum_{j=1}^{t+1} U_j \implies \sum_{j=1}^{t} U_j < \sum_{i=1}^{m} T_i < \sum_{j=1}^{t+1} U_j$$

dla wszystkich odpowiednio dużych m. To znaczy, że do niebieskiej urny wpadnie t kul i zanim wpadnie kolejna, do czerwonej urny wpadnie nieograniczona liczba kul – czyli w granicy jest $\lim_{n\to\infty} Y_n = t$.

Analogiczny argument działa, gdy $G > F$. Zatem z prawdopodobieństwem 1 zachodzi przedstawiona sytuacja, co kończy dowód. □

I.5.3 Quicksort

Wykorzystaj liniowość wartości oczekiwanej i oblicz oczekiwaną liczbę wykonanych porównań w algorytmie sortowania Quicksort. Możesz przyjąć, że sortujemy tablicę parami różnych elementów.

Definicja. Wartość oczekiwana zmiennej losowej X, oznaczana $\mathbb{E}[X]$, to

$$\mathbb{E}(X) = \sum_i i \cdot \mathbb{P}(X = i),$$

gdzie suma przebiega po obrazie X. Wartość oczekiwana może być nieokreślona, jeśli jest nieskończenie wiele wartości w obrazie i szereg określający wartość oczekiwaną nie jest zbieżny.

Twierdzenie (Liniowość wartości oczekiwanej)**.** Dla dowolnych zmiennych losowych $X_1, \ldots, X_n$ z ograniczonymi wartościami oczekiwanymi zachodzi

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i].$$

Dowód. Dowodzimy dla $n = 2$, reszta z indukcji. Mamy

$$\begin{aligned}
\mathbb{E}[X+Y] &= \sum_k k \cdot \mathbb{P}(X+Y=k) = \sum_{i,j} (i+j)\mathbb{P}((X=i) \cap (Y=j)) \\
&= \sum_i i \sum_j \mathbb{P}((X=i) \cap (Y=j)) + \sum_j j \sum_i \mathbb{P}((Y=j) \cap (X=i)) \\
&= \sum_i i\mathbb{P}(X=i) + \sum_j j\mathbb{P}(Y=j) = \mathbb{E}[X] + \mathbb{E}[Y].
\end{aligned}$$

□

Mamy algorytm sortujący Quicksort (losujemy pivota, dzielimy na mniejsze i większe i odpalamy rekurencyjnie). Jaka jest oczekiwana liczba wykonanych porównań na danych rozmiaru n?

Rozważamy już posortowane wartości $x_1, \ldots, x_n$. Zmienna X – liczba wykonanych porównań. Rozbijamy na $X_{ij} = \begin{cases} 1 & x_i, x_j \text{ były porównane} \\ 0 & \text{wpp} \end{cases}$. Mamy $\mathbb{E}[X] = \sum_{i,j} \mathbb{P}(X_{ij} = 1)$.

Elementy x_i, x_j są porównywane, jeśli jako pivot zostanie wybrany jeden z nich. Nie zostaną porównane, gdy zostanie wybrany element pomiędzy nimi. Zanim to zostanie zdecydowane algorytm może wybrać inne pivoty, leżące na lewo lub na prawo od rozważanych elementów. Decyzja o porównaniu dzieje się, gdy jako pivot zostaje wybrany element z $\{x_i, x_{i+1}, \ldots, x_j\}$. Mamy

$$\mathbb{P}(Q \text{ wybrał za pivot } x_i \text{ lub } x_j \mid Q \text{ wybrał pivota spośród } \{x_i, \ldots, x_j\}) = \frac{\frac{2}{|Z|}}{\frac{j-i+1}{|Z|}} = \frac{2}{j-i+1},$$

gdzie Z jest zbiorem wartości, na których odpalony jest algorytm w momencie decyzji, czy elementy zostaną porównane. Jak widać wielkość Z nie ma znaczenia.

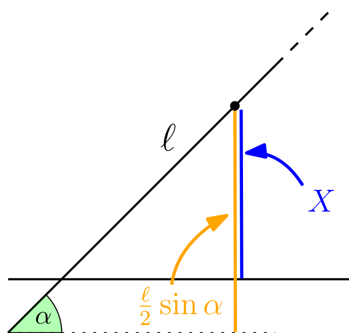
Pozostaje nam policzyć $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^{n} \sum_{i=1}^{n-k+1} \frac{2}{k} = \sum_{k=2}^{n} (n+1-k) \cdot \frac{2}{k} = 2n \ln n + \Theta(n)$. To daje nam szukaną złożoność.

I.5.4 Igła Buffona

Igłę długości ℓ rzucono na podłogę z desek o szerokości d, przy czym $\ell \leqslant d$. Jakie jest prawdopodobieństwo, że igła przetnie krawędź deski?

Zauważmy, że skoro $\ell \leqslant d$, to igła zawsze przetnie co najwyżej jedną krawędź deski. Co więcej, musi to być krawędź, do której środek igły będzie miał najbliżej. Niech więc X będzie zmienną losową, która oznacza odległość środka igły od najbliższej krawędzi deski, zaś α będzie zmienną losową, która odpowiada kątowi ostremu między igłą a krawędziami desek. Widzimy, że $X \sim \text{Uni}[0, d/2]$, zaś $\alpha \sim \text{Uni}[0, \pi/2]$. Wynika to z tego, że $X = \min(Y, d - Y)$ dla $Y \sim \text{Uni}[0, d]$, zaś $\alpha = \min(\beta, \pi - \beta)$ dla $\beta \sim \text{Uni}[0, \pi]$. Patrząc na rysunek zauważamy, że igła przetnie się z krawędzią deski dokładnie wtedy, gdy $\ell/2 \cdot \sin\alpha \geqslant X$. Liczymy więc prawdopodobieństwo takiego zdarzenia:

$$\mathbb{P}(\ell/2 \cdot \sin\alpha \geqslant X) = \int_0^{\pi/2} \int_0^{\ell/2 \cdot \sin\alpha} \frac{2}{\pi} \cdot \frac{2}{d} \,\mathrm{d}x \,\mathrm{d}\alpha = \frac{4}{\pi d} \int_0^{\pi/2} \ell/2 \cdot \sin\alpha \,\mathrm{d}\alpha = \frac{2\ell}{\pi d} \cdot (-\cos\alpha)|_0^{\pi/2} = \frac{2\ell}{\pi d}.$$



Rysunek 1: Igła Buffona.

I.5.5 Spacer losowy

Losowy spacer na płaszczyźnie. Pionek znajduje się w punkcie $(0, 0)$ na płaszczyźnie. W i-tym kroku ($i \geqslant 1$) losowany jest kąt α_i jednostajnie na $[0, 2\pi]$ (i niezależnie od poprzednich losowań), a pionek przesuwa się ze swojej aktualnej pozycji o wektor jednostkowy wyznaczony przez kąt α_i (z osią OX). Jaki jest oczekiwany kwadrat odległości od punktu $(0, 0)$ po n krokach?

Niech X_n będzie zmienną losową oznaczającą kwadrat odległości pionka od punktu $(0, 0)$ po n krokach. Pokażemy teraz indukcyjnie, że $\mathbb{E}[X_n] = n$. Widzimy, że $X_1 = 1$ z prawdopodobieństwem 1, więc oczywiście $\mathbb{E}[X_1] = 1$. Mamy więc bazę indukcji.

Załóżmy teraz, że $n \geqslant 2$. Załóżmy, że po $n - 1$ krokach pionek znajduje się w punkcie (x, y). Zauważmy, że jednostajne wylosowanie kąta α_n względem osi OX możemy równie dobrze traktować jak jednostajne wylosowanie kąta względem prostej przechodzącej przez punkty $(0, 0)$ oraz (x, y). Ponadto zauważmy, że z symetrii wynika, że wystarczy nam wtedy rozważyć losowanie takiego kąta z przedziału $[0, \pi]$. Niech więc $\beta \sim \text{Uni}[0, \pi]$. Wtedy z twierdzenia cosinusów

$$X_n = X_{n-1} + 1 - 2\sqrt{X_{n-1}} \cos\beta.$$

Obkładając wszystko wartością oczekiwaną, korzystając z niezależności zmiennej β od poprzednich kroków pionka oraz z założenia indukcyjnego dostajemy

$$\mathbb{E}[X_n] = \mathbb{E}[X_{n-1}] + 1 - 2 \cdot \mathbb{E}[\sqrt{X_{n-1}}] \cdot \mathbb{E}[\cos\beta] = n - 1 + 1 - 0 = n.$$

Po drodze skorzystaliśmy również z faktu, że $\mathbb{E}[\cos\beta] = 0$, który można łatwo udowodnić:

$$\mathbb{E}[\cos\beta] = \int_0^{\pi} \cos x \,\mathrm{d}x = \sin x|_0^{\pi} = \sin\pi - \sin 0 = 0.$$

I.5.6 Proces Poissona

Proces Poissona. Definicja i potrzebne własności aby wykazać co następuje. Niech $(N(t), t \geqslant 0)$ będzie procesem Poissona o parametrze λ. Wykazać, że jeśli w przedziale czasowym $(0, t]$ zaszło dokładnie jedno zdarzenie, czyli $N(t) = 1$, to czas zajścia tego zdarzenia (X_1) ma rozkład jednostajny na przedziale $(0, t]$ (w szczególności nie zależy od λ).

Definicja. Rozważmy ciąg zdarzeń losowych. Niech $N(t)$ zlicza liczbę zdarzeń w przedziale czasu $[0, t]$. Proces $\{N(t) : t \geqslant 0\}$ nazywamy stochastycznym procesem zliczającym.

Definicja. Proces Poissona z parametrem $\lambda > 0$ to stochastyczny proces zliczający taki, że:

1. $N(0) = 0$
2. proces ma stacjonarne i niezależne przyrosty, czyli
   (a) $\forall_{s,t \geqslant 0}$ $N(t)$ i $N(s+t) - N(s)$ mają ten sam rozkład – proces nie zmienia się w czasie.
   (b) $\forall_{t_1 < t_2 \leqslant t_3 < t_4}$ $N(t_2) - N(t_1)$ i $N(t_4) - N(t_3)$ są niezależne, czyli zdarzenia na niezależnych przedziałach są niezależne.
3. $\lim_{t \to 0} \frac{\mathbb{P}(N(t)=1)}{t} = \lambda$, czyli prędkość pojawiania się zdarzeń wynosi λ.
4. $\lim_{t \to 0} \frac{\mathbb{P}(N(t)\geqslant 2)}{t} = 0$, czyli wystąpienie więcej niż jednego wydarzenia naraz jest nieprawdopodobne.

Twierdzenie. Niech $\{N(t) : t \geqslant 0\}$ będzie procesem Poissona z parametrem λ. Wtedy

$$\forall_{t,s \geqslant 0, n \in \mathbb{N}} \ \mathbb{P}(N(t+s) - N(s) = n) = e^{-\lambda t} \frac{(\lambda t)^n}{n!},$$

czyli taka zmienna ma rozkład Poissona z parametrem λt.

Dowód. Indukcja po n. Niech $P_n(t) := \mathbb{P}(N(t+s) - N(s) = n)$. Dowodzimy bazy $n = 0$. Mamy

$$\begin{aligned} P_0(t+h) &= \mathbb{P}(N(t+h) = 0) = \mathbb{P}(N(t) = 0 \cap N(t+h) - N(t) = 0) \\ &= \mathbb{P}(N(t) = 0) \cdot \mathbb{P}(N(t+h) - N(t) = 0) = \mathbb{P}(N(t) = 0) \cdot \mathbb{P}(N(h) = 0) = P_0(t) \cdot P_0(h), \end{aligned}$$

gdzie najpierw podstawiamy $s = 0$, potem korzystamy z niezależności i stacjonarności. Mamy

$$\begin{aligned} P_0'(t) &= \lim_{h \to 0} \frac{P_0(t+h) - P_0(t)}{h} = \lim_{h \to 0} \frac{P_0(t) \cdot P_0(h) - P_0(t)}{h} = P_0(t) \lim_{h \to 0} \frac{P_0(h) - 1}{h} \\ &= P_0(t) \lim_{h \to 0} \frac{1 - \mathbb{P}(N(h) = 1) - \mathbb{P}(N(h) \geqslant 2) - 1}{h} = P_0(t)(-\lambda - 0) = -\lambda P_0(t), \end{aligned}$$

gdzie w przedostatnim przejściu skorzystaliśmy z obu granic z definicji procesu Poissona. Mamy $\frac{P_0'(t)}{P_0(t)} = -\lambda$, więc całkując obustronnie dostajemy $\ln(P_0(t)) = -\lambda t + C$, a więc

$$P_0(t) = e^{-\lambda t + C},$$

a podstawiając $P_0(0) = 1$ mamy $C = 0$, więc jest tak jak chcemy.

Teraz robimy krok indukcyjny dla $n \geqslant 1$. Mamy

$$P_n(t+h) = \sum_{k=0}^{n} P_{n-k}(t) \cdot P_k(h),$$

bo jeśli $n - k$ zdarzeń będzie miało miejsce w ciągu t jednostek czasu, to pozostałe k musi się wydarzyć w kolejnych h. Mamy

$$\begin{aligned} P_n'(t) &= \lim_{h \to 0} \frac{P_n(t+h) - P_n(t)}{h} = \lim_{h \to 0} \frac{\sum_{k=0}^{n} P_{n-k}(t) \cdot P_k(h) - P_n(t)}{h} \\ &= \lim_{h \to 0} \frac{1}{h} \left(P_n(t)(P_0(h) - 1) + P_{n-1}(t) P_1(h) + \sum_{k=2}^{n} P_{n-k}(t) P_k(h) \right) \\ &= P_n(t) \lim_{h \to 0} \frac{P_0(h) - 1}{h} + P_{n-1}(t) \lim_{h \to 0} \frac{P_1(h)}{h} = P_n(t) \cdot (-\lambda) + P_{n-1}(t) \cdot \lambda, \end{aligned}$$

gdzie ostatnie przejście to zastosowanie granicy z poprzednich przeliczeń i definicji procesu Poissona, a przedostatnie wynika z szacowania

$$\frac{1}{h}\sum_{k=2}^{n} P_{n-k}(t)\, P_k(h) \leqslant \frac{1}{h}\mathbb{P}(N(h) \geqslant 2) \to 0.$$

Rozwiązujemy równanie różniczkowe $P_n'(t)+\lambda P_n(t) = \lambda P_{n-1}(t)$. Przekształcamy domnażając przez $e^{\lambda t}$: $e^{\lambda t}(P_n'(t) + \lambda P_n(t)) = e^{\lambda t}\lambda P_{n-1}(t)$, czyli (wzór na pochodną mnożenia):

$$\frac{\mathrm{d}}{\mathrm{d}t} e^{\lambda t} P_n(t) = e^{\lambda t} \cdot \lambda P_{n-1}(t) = e^{\lambda t} \cdot \lambda e^{-\lambda t}\frac{(\lambda t)^{n-1}}{(n-1)!} = \lambda \frac{(\lambda t)^{n-1}}{(n-1)!}.$$

Teraz całkujemy i mamy $e^{\lambda t} P_n(t) = \lambda^n \cdot \frac{1}{n!} \cdot t^n + C$, licząc w $t = 0$ dostajemy $C = 0$. □

Twierdzenie. Niech X_1 będzie pierwszym międzyczasem procesu Poissona N z parametrem λ. Zmienna $X_1 \mid N(t) = 1$ ma rozkład jednostajny na $[0, t]$.

Dowód.

$$\begin{aligned}\mathbb{P}(X_1 < s \mid N(t) = 1) &= \frac{\mathbb{P}(X_1 < s \cap N(t) = 1)}{\mathbb{P}(N(t) = 1)} = \frac{\mathbb{P}(N(s) = 1) \cdot \mathbb{P}(N(t) - N(s) = 0)}{\mathbb{P}(N(t) = 1)} \\ &= \frac{e^{-\lambda s}\lambda s \cdot e^{-\lambda(t-s)}}{e^{-\lambda t}\lambda t} = \frac{s}{t}.\end{aligned}$$

□

I.5.7 Centralne Twierdzenie Graniczne

Centralne Twierdzenie Graniczne. Warianty mocniejszych wypowiedzi.

Definicja. Mówimy, że ciągła zmienna losowa X ma standardowy rozkład normalny (z parametrami 0 i 1), jeśli jej gęstość to funkcja $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. Taki rozkład oznaczamy $N(0, 1)$.

Lemat. Zachodzi

$$\int_{-\infty}^{\infty} e^{-\frac{x^2}{2}}\, \mathrm{d}x = \sqrt{2\pi}.$$

Dowód. Mamy

$$\int_{-\infty}^{\infty} e^{-\frac{z^2}{2}}\, \mathrm{d}z \cdot \int_{-\infty}^{\infty} e^{-\frac{x^2}{2}}\, \mathrm{d}x = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-\frac{z^2+x^2}{2}}\, \mathrm{d}z\, \mathrm{d}x = \int_0^{2\pi}\int_0^{\infty} e^{-\frac{r^2}{2}} r\, \mathrm{d}r\, \mathrm{d}\theta = \int_0^{2\pi} 1\, \mathrm{d}\theta = 2\pi,$$

gdzie druga równość to przejście na współrzędne biegunowe a później całkujemy przez podstawienie. To daje nam tezę. □

Definicja. Definiujemy funkcję Phi jako

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}\, \mathrm{d}z,$$

czyli dystrybuantę standardowego rozkładu normalnego.

Lemat. Wartość oczekiwana i wariancja rozkładu $N(0, 1)$ to kolejno 0 i 1.

Dowód. Mamy

$$\int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \, \mathrm{d}x = -\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \Big|_{-\infty}^{\infty} = 0.$$

Drugą całkę liczymy przez części:

$$\int_{-\infty}^{\infty} x^2 \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \, \mathrm{d}x = \frac{2}{\sqrt{2\pi}} \int_0^{\infty} x^2 e^{-\frac{x^2}{2}} \, \mathrm{d}x = \frac{2}{\sqrt{2\pi}} \left(x \cdot \left(-e^{-\frac{x^2}{2}} \right) \Big|_0^{\infty} + \int_0^{\infty} e^{-\frac{x^2}{2}} \, \mathrm{d}x \right)$$
$$= \frac{2}{\sqrt{2\pi}} \cdot \frac{\sqrt{2\pi}}{2} = 1.$$

□

Definicja. Zmienna losowa X ma rozkład normalny $N\left(\mu, \sigma^2\right)$ dla $\mu \in \mathbb{R}$ i $\sigma > 0$, jeśli istnieje $Z \sim N(0,1)$ takie, że $X = \mu + \sigma Z$.

Lemat. Jeśli $X \sim N\left(\mu, \sigma^2\right)$, to zachodzi $\mathbb{E}[X] = \mu$ i $\operatorname{Var}(X) = \sigma^2$.

Dowód.

$$\mathbb{E}[X] = \mathbb{E}[\mu + \sigma Z] = \mu + \sigma \mathbb{E}[Z] = \mu.$$
$$\operatorname{Var}(X) = \operatorname{Var}(\mu + \sigma Z) = \sigma^2 \operatorname{Var}(Z) = \sigma^2.$$

□

Lemat. Niech $X \sim N\left(\mu, \sigma^2\right)$. Dystrybuanta i gęstość X to

$$F_X(x) = \Phi\left(\frac{x-\mu}{\sigma}\right),$$

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\left(\frac{x-\mu}{\sigma}\right)^2}{2}}.$$

Dowód.

$$F_X(x) = \mathbb{P}(X \leqslant x) = \mathbb{P}\left(\frac{X-\mu}{\sigma} \leqslant \frac{x-\mu}{\sigma}\right) = \Phi\left(\frac{x-\mu}{\sigma}\right).$$

$$f_X(x) = \Phi'\left(\frac{x-\mu}{\sigma}\right) \frac{1}{\sigma} = \frac{1}{\sigma} \frac{1}{\sqrt{2\pi}} e^{-\frac{\left(\frac{x-\mu}{\sigma}\right)^2}{2}}.$$

□

Lemat. Funkcja tworząca momentów rozkładu normalnego $N\left(\mu, \sigma^2\right)$ to

$$M_X(t) = e^{\frac{t^2\sigma^2}{2} + t\mu}.$$

Dowód.

$$M_X(t) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} e^{tx} \cdot e^{-\frac{\left(\frac{x-\mu}{\sigma}\right)^2}{2}} \, \mathrm{d}x = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{t(\sigma u + \mu)} \cdot e^{-\frac{u^2}{2}} \, \mathrm{d}u$$
$$= \frac{1}{\sqrt{2\pi}} e^{t\mu} \int_{-\infty}^{\infty} e^{-\frac{(u-t\sigma)^2}{2}} \cdot e^{\frac{t^2\sigma^2}{2}} \, \mathrm{d}u = e^{t\mu + \frac{t^2\sigma^2}{2}} \cdot \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{(u-t\sigma)^2}{2}} \, \mathrm{d}u = e^{t\mu + \frac{t^2\sigma^2}{2}},$$

gdzie drugie przejście to podstawienie $u = \frac{x-\mu}{\sigma}$ a ostatnie to zauważenie, że ta całka to dystrybuanta rozkładu normalnego $N(t\sigma, 1)$. □

Definicja. Mówimy, że ciąg dystrybuant $F_1, F_2, \ldots$ zbiega do dystrybuanty F (względem rozkładu), jeżeli

$$\forall_{a \in \mathbb{R}} \; F \text{ jest ciągłe w } a \implies \lim_{n \to \infty} F_n(a) = F(a).$$

Oznaczamy to $F_n \xrightarrow{D} F$ (zbieżność punktowa).

Twierdzenie (Lévy'ego o ciągłości). Niech $Y_1, Y_2, \ldots$ będzie ciągiem zmiennych losowych gdzie Y_i ma dystrybuantę F_i i funkcję tworzącą momentów M_i. Niech Y będzie zmienną losową o dystrybuancie F i funkcji tworzącej momentów M. Jeżeli

$$\forall_t \lim_{n \to \infty} M_n(t) = M(t),$$

to $F_n \xrightarrow{D} F$.

Twierdzenie (Centralne Twierdzenie Graniczne). Niech $X_1, \ldots, X_n$ będą niezależnymi zmiennymi losowymi o takich samych rozkładach z wartością oczekiwaną μ i wariancją σ^2. Dalej połóżmy $\overline{X}_n = \frac{1}{n}\sum_{i=1}^n X_i$. Wówczas dla dowolnych a i b zachodzi

$$\mathbb{P}\left(a \leqslant \frac{\overline{X}_n - \mu}{\frac{\sigma}{\sqrt{n}}} \leqslant b\right) \xrightarrow{D} \Phi(b) - \Phi(a).$$

Dowód. Niech $Z_i = \frac{X_i - \mu}{\sigma}$. Wówczas Z_i mają takie same rozkłady i $\mathbb{E}[Z_i] = 0$, $\mathrm{Var}(Z_i) = 1$. Rozważmy zmienną losową $Y_n = \sum_{i=1}^n \frac{Z_i}{\sqrt{n}} = \frac{\sqrt{n}}{n}\sum_{i=1}^n \frac{X_i - \mu}{\sigma} = \frac{\overline{X}_n - \mu}{\frac{\sigma}{\sqrt{n}}}$. Chcemy pokazać, że

$$\lim_{n \to \infty} \mathbb{E}\left[e^{tY_n}\right] = e^{\frac{t^2}{2}},$$

co da nam tezę z twierdzenia Lévy'ego. Oznaczając $M(t) = \mathbb{E}\left[e^{tZ_i}\right]$ mamy

$$\mathbb{E}\left[e^{tY_n}\right] = \mathbb{E}\left[e^{\frac{t}{\sqrt{n}}\sum_{i=1}^n Z_i}\right] = \left(M\left(\frac{t}{\sqrt{n}}\right)\right)^n.$$

Definiujemy $L(t) = \ln M(t)$. Przeliczamy:

1. $L(0) = \ln M(0) = \ln 1 = 0$, bo $M(0) = \mathbb{E}\left[e^{0 \cdot Z_i}\right] = 1$
2. $L'(0) = \frac{M'(0)}{M(0)} = \mathbb{E}[Z_i] = 0$
3. $L''(0) = \frac{M(0)M''(0) - (M'(0))^2}{(M(0))^2} = \mathbb{E}\left[Z_i^2\right] = 1$

Z ciągłości logarytmu wystarczy pokazać, że

$$nL\left(\frac{t}{\sqrt{n}}\right) \to \frac{t^2}{2}.$$

Korzystając dwukrotnie z reguły de l'Hospitala dostajemy

$$\lim_{n \to \infty} \frac{L\left(\frac{t}{\sqrt{n}}\right)}{n^{-1}} = \lim_{n \to \infty} \frac{-L'\left(\frac{t}{\sqrt{n}}\right) n^{-\frac{3}{2}} t}{-2n^{-2}} = \lim_{n \to \infty} \frac{-L''\left(\frac{t}{\sqrt{n}}\right) n^{-\frac{3}{2}} t^2}{-2n^{-\frac{3}{2}}} = \lim_{n \to \infty} L''\left(\frac{t}{\sqrt{n}}\right)\frac{t^2}{2} = \frac{t^2}{2}.$$

□

Twierdzenie. Niech $X_1, \ldots, X_n$ będzie ciągiem niezależnych zmiennych losowych z $\mathbb{E}[X_i] = \mu_i$ i $\mathrm{Var}(X_i) = \sigma_i^2$. Niech zachodzi

1. $\exists_{M>0} \forall_{i \in [n]} \mathbb{P}(|X_i| < M) = 1$
2. $\lim_{n \to \infty} \sum_{i=1}^n \sigma_i^2 = +\infty$.

Wówczas

$$\mathbb{P}\left(a \leqslant \frac{\sum_{i=1}^n (X_i - \mu_i)}{\sqrt{\sum_{i=1}^n \sigma_i^2}} \leqslant b\right) \xrightarrow{D} \Phi(b) - \Phi(a).$$

Twierdzenie (Berry-Esséen). Istnieje taka stała C, że zachodzi następujące: niech $X_1, \ldots, X_n$ będą niezależnymi zmiennymi losowymi o tym samym rozkładzie ze skończoną wartością oczekiwaną μ i wariancją σ^2. Dalej niech $\rho = \mathbb{E}\left[|X_i - \mu|^3\right] < \infty$ i $\overline{X}_n = \frac{1}{n}\sum_{i=1}^n X_i$. Mamy

$$\left|\mathbb{P}\left(\frac{\overline{X}_n - \mu}{\frac{\sigma}{\sqrt{n}}} \leqslant a\right) - \Phi(a)\right| \leqslant C \cdot \frac{\rho}{\sigma^3\sqrt{n}}.$$

I.6 Modele Obliczeń

I.6.1 Hierarchia Chomsky'ego

Omów hierarchię Chomsky'ego języków, ilustrując klasy i zależności między nimi odpowiednimi przykładami.

Dla ustalonego alfabetu Σ językiem nazywamy *dowolny podzbiór* $L \subseteq \Sigma^*$. Innymi słowy, językiem jest dowolny zbiór skończonych ciągów nad Σ.

Hierarcha Chomsky'ego składa się z 4 warstw.

Języki typu 3: *języki regularne* oznaczane L_3. Jest to najwęższa spośród klas języków. Język L jest regularny wtedy i tylko wtedy, gdy istnieje wyrażenie regularne α nad Σ takie, że $L(\alpha) = L$. Równoważnie, gdy istnieje automat skończony A taki, że $L(A) = L$. Przykład języka regularnego to $L = \{a^n b : n \in \mathbb{N}\}$, gdyż $L = L(a^*b)$.

Języki typu 2: *języki bezkontekstowe* oznaczane L_2. One również mają dwie równoważne charakteryzacje, poprzez *automaty ze stosem* oraz *gramatyki bezkontekstowe*. Zauważmy, że każdy automat skończony jest automatem ze stosem, zatem każdy język regularny jest językiem bezkontekstowym. Przykładem języka bezkontekstowego, który nie jest językiem regularnym jest $L = \{a^n b^n : n \in \mathbb{N}\}$. Generuje go gramatyka $G = (\{S\}, \{a, b\}, P, S\}$, gdzie w skład P wchodzą produkcje $S \to aSb \mid \varepsilon$. Ten język nie jest regularny, co wynika z następującego lematu.

Lemat (O pompowaniu)**.** Jeśli L jest językiem regularnym, to istnieje takie $n > 0$ (zależne od L), że dla każdego $w \in L$ z $|w| \geqslant n$ istnieje postać $w = xyz$ taka, że $|xy| \leqslant n$, $|y| \geqslant 1$ i $\forall_{i\in\mathbb{N}}\ xy^iz \in L$. Inaczej mówiąc: musi istnieć takie niepuste y będące blisko początku słowa, że można je „napompować" – powtórzyć dowolną liczbę razy.

Dowód. Niech $A = (Q, \Sigma, \delta, s, F)$ będzie DFA akceptującym L. Ustalmy $n = |Q|$ i weźmy dowolne słowo $w \in L$ takie, że $m = |w| \geqslant n$. Niech $w = a_0 \dots a_{m-1}$.

Słowo jest akceptowane, a więc istnieje ciąg stanów $q_0, \dots, q_m$ taki, że $q_0 = s$ i $q_m \in F$. Jego długość jest większa niż n, a więc jakiś stan musi się powtarzać. Niech q_i będzie tym stanem, którego pierwsze powtórzenie występuje najwcześniej – niech będzie to q_j. Ustalamy

$$x = a_0 \dots a_{i-1},\ y = a_i \dots a_{j-1},\ z = a_j \dots a_{m-1}.$$

Mamy $|xy| \leqslant n$, bo inaczej q_i nie powtarzałby się najwcześniej. Do tego z $q_i = q_j$ wynika, że słowo y może wystąpić w akceptowanym słowie dowolną liczbę razy – mamy cykl stanów. Zatem $\forall_{i\in\mathbb{N}}\ xy^iz \in L$. Oczywiście y jest niepuste. □

Twierdzenie. Język $L = \{a^n b^n : n \in \mathbb{N}\}$ nie jest regularny.

Dowód. Pokażemy, że L nie spełnia lematu o pompowaniu. Ustalmy dowolne $n > 0$ i rozważmy słowo $w = a^{2n} b^{2n}$. Dla każdego podziału $w = xyz$ takiego, że $|xy| \leqslant n$ zachodzi $xy = a^k$ dla pewnego k. Wobec tego mamy $y = a^\ell$ dla pewnego $\ell > 0$. Gdyby lemat o pompowaniu był spełniony, to $a^{2n-\ell} b^{2n}$ musiałoby należeć do języka, a tak nie jest. Zatem L nie spełnia lematu o pompowaniu i nie jest regularny. □

Języki typu 1: *języki kontekstowe* oznaczane L_1. Na Modelach Obliczeń pojawiły się one jako języki opisywane przez automaty ograniczone liniowo (*Linearly Bounded Automaton*, LBA). Są to niedeterministyczne Maszyny Turinga, które obliczając słowo w dostają na wejściu $\vdash w \dashv$ oraz głowica Maszyny Turinga nigdy nie może wyjść poza ograniczniki. Przykładem języka kontekstowego jest $L = \{a^n b^n c^n : n \in \mathbb{N}\}$. Odpowiednie LBA zaczyna od lewego ogranicznika, idąc w prawo natrafia na a, skreśla je i idzie dalej, napotyka ciąg a i potem b, skreśla je i idzie dalej, po ciągu b napotyka c,

skreśla je i po ciągu c napotyka prawy ogranicznik. Wtedy wraca do lewego ogranicznika. Jeśli cokolwiek stanie się inaczej niż było powiedziane (na przykład napotka inną literę albo nie w takiej kolejności), to maszyna się zapętla. W przeciwnym wypadku zaczyna od nowa, ignorując wykreślone litery. Akceptuje, jeśli przejdzie od lewego do prawego ogranicznika bez napotkania nieskreślonej litery. To, że L nie jest bezkontekstowy pokazujemy za pomocą lematu o pompowaniu w jednym z kolejnych pytań.

Całkiem łatwo zauważyć, że dowolne PDA może być symulowane przez LBA. Załóżmy, że mamy PDA P, które w każdym przejściu umieszcza na stosie co najwyżej M znaków. Wtedy możemy zdefiniować LBA, którego alfabet taśmowy będzie składał się z par (litera alfabetu P, ciąg co najwyżej M symboli stosowych P). Takie LBA może symulować P: przegląda kolejne litery, na przeglądniętych już literach trzyma dodatkowo symbole stosowe dołożone przy ich przeglądnięciu. Jest w stanie znaleźć symbol, który jest na szczycie stosu, a dzięki temu zasymulować ruch P.

Języki kontekstowe są również charakteryzowane przez tak zwane *gramatyki kontekstowe*. Są one uogólnieniem gramatyk bezkontekstowych – produkcje nie są postaci $A \to \gamma$ dla pewnego nieterminala A, ale $\alpha A\beta \to \alpha\gamma\beta$ dla pewnego nieterminala A i form zdaniowych α, β (czyli ciągów teminali i nieterminali). Intuicyjnie mówiąc, mamy kontekst, który stwierdza, czy możemy przejść daną produkcją.

Języki typu 0: *języki rekurencyjnie przeliczalne* oznaczane L_0. Jest to najszersza klasa języków. Język L jest rekurencyjnie przeliczalny, gdy istnieje *Maszyna Turinga* M taka, że $L = L(M)$. Każdy język należący do poprzednich klas jest rekurencyjnie przeliczalny (bo każde LBA jest Maszyną Turinga), jednak istnieją języki rekurencyjnie przeliczalne, które nie są kontekstowe. W szczególności każdy język kontekstowy jest *rekurencyjny* (czyli istnieją dla nich Maszyny Turinga, które nie tylko akceptują słowa należące do języka, ale również zatrzymują się bez akceptacji dla słów nie należących do języka), bo LBA mają ograniczoną taśmę, ilość stanów i ilość symboli taśmowych, więc mają tylko skończenie wiele konfiguracji. Jeśli zasymulujemy więcej niż tyle konfiguracji, to znajdziemy się drugi raz w takiem samej, a więc nasze LBA się zapętliło i wiemy, że możemy się zatrzymać. Wiemy z kolei, że istnieją języki, które nie są rekurencyjnie, na przykład język uniwersalny L_u (język składający się z Maszyn Turinga sparowanych ze słowami, które akceptują).

Te wszystkie rozważania dają nam ciąg

$$L_3 \subsetneq L_2 \subsetneq L_1 \subsetneq L_0,$$

który nazywamy Hierarchią Chomsky'ego.

I.6.2 Języki bezkontekstowe

> Omów lemat o pompowaniu dla języków bezkontekstowych. Podaj przykład języka bezkontekstowego. Podaj przykład języka, który nie jest bezkontekstowy.

Zaczynamy od przedstawienia gramatyk bezkontekstowych i automatów ze stosem. Raczej nie trzeba wprowadzać tych wszystkich definicji odpowiadając na to pytanie, ale na pewno trzeba je znać (albo przynajmniej rozumieć, jakie intuicje na nimi stoją).

Definicja. Gramatyka bezkontekstowa (CFG, context-free grammar) to krotka $G = (N, \Sigma, P, S)$, gdzie:

- N - skończony zbiór zmiennych (nieterminale),
- Σ - skończony alfabet (terminale),
- $P \subseteq N \times (N \cup \Sigma)^*$ - skończony zbiór produkcji,
- $S \in N$ - symbol startowy.

Produkcję (s, w) będziemy zapisywać $s \to w$. Kilka produkcji $s \to w_1, s \to w_2$ możemy zapisać łącznie: $s \to w_1 \mid w_2$.

Definicja. Forma zdaniowa to dowolne słowo z $(N \cup \Sigma)^*$.

Definicja. Dla gramatyki bezkontekstowej $G = (N, \Sigma, P, S)$ definiujemy relację $\to_G$ na formach zdaniowych. Mówimy, że $\alpha \to_G \beta$ (możemy uzyskać β z α w jednym kroku), jeśli

$$\exists_{\alpha_1,\alpha_2,\gamma \in (N\cup\Sigma)^*} \ \exists_{A\in N} \ \ (A,\gamma) \in P \wedge \alpha = \alpha_1 A \alpha_2 \wedge \beta = \alpha_1 \gamma \alpha_2,$$

czyli β powstaje z α przez zamianę nieterminala na formę zdaniową zgodnie z jakąś produkcją.

Przez $\to_G^*$ oznaczamy zwrotne i przechodnie domknięcie $\to_G$, czyli $\alpha \to_G^* \beta$, jeśli możemy uzyskać β z α w pewnej liczbie kroków.

Definicja. Język $L(G)$ generowany przez gramatykę $G = (N, \Sigma, P, S)$ to język

$$L(G) = \{w \in \Sigma^* : S \to_G^* w\}.$$

Ogólniej można zdefiniować

$$L(G, A) = \{w \in \Sigma^* : A \to_G^* w\}.$$

Takie języki nazywamy językami bezkontekstowymi (CFL).

Definicja. Wywodem (derivation) w gramatyce G nazywamy ciąg form zdaniowych $\alpha_0, \ldots, \alpha_n$ takich, że $\forall_{i<n} \ \alpha_i \to_G \alpha_{i+1}$.

Uwaga. $v \in L(G)$ wtedy i tylko wtedy, gdy istnieje wywód $\alpha_0, \ldots, \alpha_n$ w G taki, że $\alpha_0 = S$ i $\alpha_n = v$.

Definicja. Wywód lewostronny $\alpha_0, \ldots, \alpha_n$ to taki wywód, że dla każdego $i < n$ mamy

$$\alpha_i = xAB_i,$$

gdzie $x \in \Sigma^*$, $A \in N$, $B_i \in (N \cup \Sigma)^*$. Przez A rozumiemy nieterminal, który jest zastępowany w danym kroku wywodu. Zatem w wywodzie lewostronnym zawsze zamieniamy najbardziej lewy nieterminal.

Definicja. Drzewo wywodu (parse tree) formy zdaniowej α w gramatyce $G = (N, \Sigma, P, S)$ to ukorzenione drzewo z porządkiem na dzieciach, w którym każdy wierzchołek ma etykietę z $N \cup \Sigma \cup \{\varepsilon\}$. Etykieta korzenia to S. Jeśli wierzchołek ma etykietę A i dzieci $X_1, \ldots, X_n$, to $A \to X_1 \ldots X_n$ jest produkcją w P. Wierzchołki o etykiecie ε są liśćmi i nie mają rodzeństwa. W wyniku konkatenacji etykiet liści otrzymujemy α.

Propozycja. Każde drzewo wywodu można jednoznacznie przypisać do wywodu lewostronnego.

Dowód. Mając zadane drzewo wywodu dla formy zdaniowej α można przejść po nim zgodnie z kolejnością dzieci. W ten sposób z symbolu startowego dostajemy α, a w kolejnych formach zdaniowych zmienia się najbardziej lewy nieterminal. Podobnie wywód lewostronny zadaje drzewo wywodu – budujemy drzewo zgodnie z kolejnymi przejściami w wywodzie. □

Definicja. Automat ze stosem (PDA – pushdown automaton) to krotka

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

gdzie:

- Q – skończony zbiór stanów,
- Σ – skończony alfabet,
- Γ – skończony alfabet stosowy,
- $\delta : (Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \to \mathcal{P}(Q \times \Gamma^*)$ – skończona niedeterministyczna funkcja przejścia,
- $q_0 \in Q$ – stan startowy,
- $Z_0 \in \Gamma$ – stosowy symbol początkowy (sygnalizujący pusty stos),
- $F \subseteq Q$ – zbiór stanów końcowych.

Taki automat działa podobnie jak ε-NFA, ale tym razem ma stos, na który może odkładać różne wartości, które potem są z niego ściągane i uczestniczą w podejmowaniu decyzji.

Definicja. Opis chwilowy (konfiguracja) PDA to trójka (q, w, γ), gdzie q to aktualny stan, w to nieprzetworzona jeszcze część słowa a γ to wszystkie symbole na stosie (czytane od góry).

Definicja. Dla PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ definiujemy $\vdash_P$ jako relację przejścia między konfiguracjami, to znaczy

$$(q, aw, X\beta) \vdash_P (p, w, \alpha\beta) \iff (p, \alpha) \in \delta (q, a, X),$$

gdzie $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$ a pozostałe symbole mają typy wynikające z definicji.

Definiujemy $\vdash_P^*$ jako zwrotne i przechodnie domknięcie $\vdash_P$.

Definicja. Niech $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ będzie PDA. Językiem akceptowanym stanem końcowym tego PDA nazywamy język $L(P) = \{w \in \Sigma^* : (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha) \wedge q \in F\}$.

Definicja. Niech $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ będzie PDA. Językiem akceptowanym pustym stosem tego PDA nazywamy język $N(P) = \{w \in \Sigma^* : (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \varepsilon)\}$.

Twierdzenie. Mamy $L = L(P)$ dla pewnego PDA P wtedy i tylko wtedy, gdy $L = N(Q)$ dla pewnego PDA Q.

Twierdzenie. Język L jest bezkontekstowy wtedy i tylko wtedy, gdy jest akceptowany przez pewien automat ze stosem.

Teraz możemy przejść do odpowiedzi na faktyczne pytanie.

Przykład. Język $L = \{a^n b^n : \mathbb{N}\}$ jest bezkontekstowy. Generuje go gramatyka $G = (\{S\}, \{a, b\}, P, S)$ o produkcjach:

$$S \to aSb|\varepsilon$$

Możemy zapytać, jakie własności muszą spełniać języki, aby były bezkontekstowe. Jednym z nich jest *lemat o pompowaniu dla języków bezkontekstowych.*

Twierdzenie. Niech L będzie językiem bezkontekstowym. Wtedy istnieje taka stała n, że dla każdego słowa $z \in L$ z $|z| \geq n$ istnieje podział $z = uvwxy$ spełniający następujące warunki:

1. $|vwx| \leq n$.
2. $|vx| \geq 1$.
3. $\forall_{k \in \mathbb{N}}\ uv^k wx^k y \in L$.

Zauważmy, że twierdzenie prowadzi nas do następującej obserwacji:

Przykład. Język $L = \{a^n b^n c^n : n \in \mathbb{N}\}$ nie jest bezkontekstowy.

Dowód. Poprowadzimy dowód nie wprost. Niech N będzie stałą z lematu o pompowaniu. Rozważmy słowo $z = a^N b^N c^N$. Wtedy z lematu o pompowaniu istnieje podział $z = uvwxy$ spełniający powyższe warunki. Zauważmy jednak, że $|vwx| \leq N$, więc pewna litera z $\{a, b, c\}$ nie będzie występować w vx. Zatem dopompowanie raz zaburzy równość ilości poszczególnych liter w słowie, więc $uv^2wx^2y \notin L$. To przeczy lematowi o pompowaniu, więc L nie jest bezkontekstowy. □

Pozostaje nam przedstawić *szkic dowodu lematu o pompowaniu.* Przyda nam się do tego *Postać Normalna Chomsky'ego.*

Definicja. Mówimy, że $G = (N, \Sigma, P, S)$ jest w postaci normalnej Chomsky'ego (mówimy, że G jest w CNF), jeśli wszystkie produkcje są postaci $A \to BC$ lub $A \to a$, dla $a \in \Sigma$, $A, B, C \in N$ zaś wszystkie nieterminale są osiągalne i generujące.

Skorzystamy z tego, że dla każdej gramatyki G istnieje gramatyka G' w CNF, że $L(G) \setminus \{\varepsilon\} = L(G')$.

Propozycja. Niech G będzie gramatyką bezkontekstową w postaci normalnej Chomsky'ego. Jeśli w drzewie wywodu słowa w każda ścieżka ma długość co najwyżej n, to zachodzi $|w| \leqslant 2^{n-1}$.

Dowód. Dowód indukcją po długości najdłuższej ścieżki w drzewie. Jeśli ma ona długość 1, to drzewo składa się z korzenia i jego jednego dziecka etykietowanego terminalem, mamy $|w| = 1$. Jeśli jest dłuższa, to korzeń ma dokładnie dwójkę dzieci (bo G jest w CNF). Oba poddrzewa ukorzenione w tych dzieciach mają długość najdłuższej ścieżki co najwyżej $n-1$, a więc z założenia indukcyjnego mamy $|w| \leqslant 2^{n-2} + 2^{n-2} = 2^{n-1}$. □

Dowód Lematu o pompowaniu. Niech $G = (N, \Sigma, P, S)$ będzie taką gramatyką w CNF, że $L(G) = L \setminus \{\varepsilon\}$. Wybierzmy $n = 2^{m+1} + 1$, gdzie $m := |N|$. Weźmy dowolne słowo $z \in L : |w| \geq n$. Wtedy w drzewie wywodu z istnieje ścieżka od korzenia do liścia długości $m+1$. Z *Zasady Szufladkowej Dirichleta* pewien nieterminal powtarza się na niej dwa razy, więc weźmy spośród nich taki nieterminal A, którego poprzednie wystąpienie jest jak najniżej. Wtedy rozpatrując ten wywód, w kontekście dwóch najniższych wystąpień A mamy

$$S \to_G^* uAy \to_G^* uvAxy \to_G^* z := uvwxy.$$

Wskazaliśmy w ten sposób podział $z = uvwxy$, który spełnia *warunek 3.*, gdyż $A \to_G^* v^k A x^k \,|\, w$. Z minimalności wysokości A górne jego wystąpienie jest na wysokości co najwyżej m, zatem $|vwx| \leq 2^m \leq n$. Jest też $|vx| \geq 1$, gdyż w CNF nie ma produkcji wymazujących (dających słowo zerowe). To kończy dowód. □

Ten dowód da się przeprowadzić też bez postaci normalnej Chomsky'ego. Wymaga to rozważenia drzew o większej liczbie dzieci wierzchołków niż 2, ale poza tym dowód przebiega dokładnie tak samo (pod koniec może się okazać, że $|vx| = 0$, ale wtedy będziemy w stanie znaleźć inną ścieżkę, dla której nie będzie to 0).

I.6.3 Języki regularne a automaty skończone

> W jaki sposób języki regularne są charakteryzowane przez automaty skończone? Nakreśl ideę dowodu.

Definicja. Wyrażenia regularne nad alfabetem Σ to obiekty tworzone następująco:

1. Bazowo wyróżniamy trzy rodzaje wyrażeń regularnych: a (takie że $a \in \Sigma$), $\emptyset$, ε.
2. Jeśli E, F to wyrażenia regularne, to $(E + F)$, $(E \cdot F)$, $(E)^*$ to wyrażenia regularne.

Języki regularne otrzymujemy poprzez interpretację wyrażeń regularnych.

$$\forall_{a \in \Sigma}\ L(a) = \{a\}; \quad L(\emptyset) = \emptyset; \quad L(\varepsilon) = \{\varepsilon\}$$
$$L(E + F) = L(E) \cup L(F); \quad L(E \cdot F) = L(E) \cdot L(F); \quad L(E^*) = \bigcup_{n=0}^{\infty} (L(E))^n$$

Definicja. Deterministyczny automat skończony (DFA – deterministic finite automaton) to krotka $A = (Q, \Sigma, \delta, s, F)$, gdzie:

- Q – skończony zbiór stanów,
- Σ – skończony alfabet,
- $\delta : Q \times \Sigma \to Q$ – funkcja przejścia,
- $s \in Q$ – stan początkowy,
- $F \subseteq Q$ – stany końcowe (akceptujące).

Automat zaczyna od zadanego stanu s i dostaje kolejne litery alfabetu, na podstawie których przechodzi do nowych stanów w sposób zadany przez δ. Będą nas interesować sytuacje, w których ostatecznie znajdziemy się w stanie należącym do F.

Definicja. Definiujemy funkcję $\widehat{\delta} : Q \times \Sigma^* \to Q$ w następujący sposób

$$\widehat{\delta}(q, w) = \begin{cases} q, & w = \varepsilon \\ \delta\left(\widehat{\delta}(q, x), a\right), & w = xa, a \in \Sigma, x \in \Sigma^* \end{cases}.$$

Jest to funkcja określająca stan automatu po wielu krokach (zadanych danym słowem).

Definicja. Niech $A = (Q, \Sigma, \delta, s, F)$ będzie DFA. Językiem akceptowanym przez A nazywamy język $L(A) = \left\{w \in \Sigma^* : \widehat{\delta}(s, w) \in F\right\}$.

Definicja. Niedeterministyczny automat skończony (NFA – nondeterministic finite automaton) to krotka $A = (Q, \Sigma, \delta, S, F)$, gdzie:

- Q – skończony zbiór stanów,
- Σ – skończony alfabet,
- $\delta \subseteq Q \times \Sigma \times Q$ – relacja przejścia,
- $S \subseteq Q$ – zbiór stanów początkowych,
- $F \subseteq Q$ – stany końcowe (akceptujące).

Automat niedeterministyczny różni się od deterministycznego tym, że zaczyna w wielu stanach początkowych i wykonuje przejścia zgodne z δ, która tym razem jest relacją – dla zadanego stanu i litery może być wiele poprawnych przejść (lub nie być żadnego).

Definicja. Definiujemy funkcję $\widetilde{\delta} : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ w następujący sposób

$$\widetilde{\delta}(B, a) = \{q \in Q : \exists_{q_b \in B}\ (q_b, a, q) \in \delta\}.$$

Jest to funkcja określająca możliwe stany dla zadanych stanów początkowych i litery.

Definicja. Analogicznie jak przedtem definiujemy funkcję $\widehat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$.

$$\widehat{\delta}(B, w) = \begin{cases} B, & w = \varepsilon \\ \widetilde{\delta}\left(\widehat{\delta}(B, x), a\right), & w = xa, a \in \Sigma, x \in \Sigma^* \end{cases}.$$

Definicja. Niech $A = (Q, \Sigma, \delta, S, F)$ będzie NFA. Językiem akceptowanym przez A nazywamy język $L(A) = \left\{w \in \Sigma^* : \widehat{\delta}(S, w) \cap F \neq \emptyset\right\}$.

Twierdzenie. Niech $L \subseteq \Sigma^*$. Mamy $L = L(A_D)$ dla pewnego DFA A_D wtedy i tylko wtedy, gdy $L = L(A_N)$ dla pewnego NFA A_N.

Dowód. ($\Longrightarrow$) Jeśli DFA $(Q, \Sigma, \delta, s, F)$ akceptuje L, to NFA $(Q, \Sigma, \delta, \{s\}, F)$ również (każdy DFA jest NFA z minimalną różnicą typu krotki).

($\Longleftarrow$) Konstrukcja potęgowa – jeśli $A_N = (Q, \Sigma, \delta, S, F)$ akceptuje L, to akceptuje je też DFA $(\mathcal{P}(Q), \Sigma, \delta_D, S, \mathcal{F})$, gdzie $\delta_D = \widetilde{\delta}$ (czyli $\widehat{\delta}_D = \widehat{\delta}$) oraz $\mathcal{F} = \{B \in \mathcal{P}(Q) : B \cap F \neq \emptyset\}$. □

Definicja. Definiujemy ε-NFA tak samo jak NFA, z tym, że relacja przejścia ma sygnaturę $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ (zakładamy, że $\varepsilon \notin \Sigma$). Oznacza to, że ε-NFA może wykonywać przejścia na słowie pustym, czyli zmienić stan bez „zużycia” litery z wejścia.

Funkcję $\widetilde{\delta}$ definiujemy identycznie jak dla NFA.

Definicja. Definiujemy ε-domknięcie zbioru stanów B w automacie A jako zbiór $B^{A,\varepsilon}$ taki, że

$$B^{A,\varepsilon} = \bigcup_{i=0}^{\infty} B_i^{A,\varepsilon},$$

gdzie

$$B_i^{A,\varepsilon} = \begin{cases} B, & i = 0 \\ \widetilde{\delta}\left(B_{i-1}^{A,\varepsilon}, \varepsilon\right), & i > 0 \end{cases}.$$

Intuicyjnie, są to wszystkie stany, do których da się dojść z B za pomocą przejść ε.

Definicja. Analogicznie jak przedtem dla ε-NFA A definiujemy funkcję $\widehat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$.

$$\widehat{\delta}(B, w) = \begin{cases} B^{A,\varepsilon}, & w = \varepsilon \\ \widetilde{\delta}\left(\widehat{\delta}(B, x), a\right)^{A,\varepsilon}, & w = xa, a \in \Sigma, x \in \Sigma^* \end{cases}.$$

Jest to taka sama funkcja jak dla NFA z tym, że po zużyciu każdej litery dokonujemy ε-domknięcia.

Język akceptowany przez ε-NFA definiujemy identycznie jak dla zwykłego NFA.

Twierdzenie. Niech $L \subseteq \Sigma^*$. Mamy $L = L(A_D)$ dla pewnego DFA A_D wtedy i tylko wtedy, gdy $L = L(A_N)$ dla pewnego ε-NFA A_N.

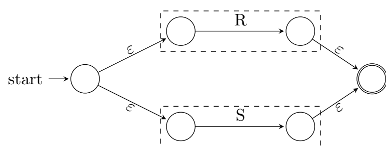
Dowód. ($\Longrightarrow$) Podobnie jak w przypadku NFA, każde DFA jest NFA, które z kolei jest ε-NFA.

($\Longleftarrow$) Jeśli $A_N = (Q, \Sigma, \delta, S, F)$ akceptuje L, to akceptuje je też DFA $\left(\mathcal{P}(Q), \Sigma, \delta_D, S^{A_N,\varepsilon}, \mathcal{F}\right)$, gdzie $\delta_D = \widetilde{\delta}^{A_N,\varepsilon}$ (czyli $\widehat{\delta}_D = \widehat{\delta}$) oraz $\mathcal{F} = \{B \in \mathcal{P}(Q) : B \cap F \neq \emptyset\}$. □

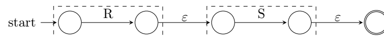
Niedeterministycznych automatów ε-NFA użyjemy do rekurencyjnej konstrukcji automatu akceptującego język regularny. Najpierw zróbmy automaty dla prostych wyrażeń regularych: $a \in \Sigma, \varepsilon, \emptyset$. Możemy to zrobić następująco:

Rysunek 2: Przypadek bazowy – pojedyncza literka i symbol epsilon.

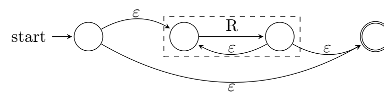
Zostały nam przypadki konstrukcji większych wyrażeń z mniejszych. Załóżmy że dla wyrażeń regularnych R, S mamy automaty które akceptują dokładnie takie języki jak opisują te wyrażenia. Następujące konstrukcje pozwolą nam zaakceptować wyrażenia $R + S$, $R \cdot S$ i $(S)^*$:



Rysunek 3: Wyrażenie $R + S$



Rysunek 4: Wyrażenie $R \cdot S$



Rysunek 5: Wyrażenie $(R)^*$

Zatem pokazaliśmy, że języki regularne są akceptowane przez niedeterministyczne automaty skończone, a więc również przez DFA.

Do tego dla każdego DFA A o stanach $Q = \{q_1, q_2, \ldots, q_n\}$, możemy skonstruować wyrażenie regularne R takie, że $L(R) = L(A)$. Konstrukcja opiera się na stworzeniu podwyrażeń $R_{i,j}^{(k)}$ takich, że

$L(R_{i,j}^{(k)})$ to zbiór takich słów w, że zaczynając w q_i po ich wczytaniu skończymy w stanie q_j, a każdy stan q_t który odwiedzimy pomiędzy spełnia $t \leqslant k$.

W ten sposób możemy pokazać, że klasa języków regularnych to dokładnie klasa języków akceptowanych przez automaty skończone. Robimy to w następujący sposób.

Niech DFA $A = (Q, \Sigma, \delta, s, F)$. Wprowadźmy porządek na stanach $Q = \{q_1, \ldots, q_n\}$. Zauważmy, że $w \in L(A)$ jest równoważne istnieniu ścieżki na stanach $s \rightsquigarrow q_j \in F$, którą automat przechodzi dla tego

słowa. R będzie generowało dokładnie takie słowa. Określamy

$$R_{i,j}^{(0)} = \beta_{i,j} + \sum_{a \in \Sigma : \delta(q_i, a) = q_j} a, \text{ gdzie } \beta_{i,j} = \begin{cases} \emptyset, & i \neq j \\ \varepsilon, & i = j \end{cases}.$$

Teraz dla $0 < k \leqslant n$ możemy zdefiniować $R_{i,j}^{(k)} = R_{i,j}^{(k-1)} + R_{i,k}^{(k-1)} \left(R_{k,k}^{(k-1)}\right)^* R_{k,j}^{(k-1)}$ – mając do dyspozycji nowy stan q_k możemy albo znaleźć ścieżkę bez niego, albo dojść do niego korzystając tylko z mniejszych stanów, potem wrócić do niego dowolną liczbę razy i ostatecznie wyjść z niego i dojść do końca pozostałymi stanami.

Ostatecznie szukane wyrażenie to $R = \sum_{q_j \in F} R_{i,j}^{(n)}$, gdzie $q_i = s$.

I.6.4 Operacje na językach regularnych i bezkontekstowych

Omów zamkniętość klasy języków regularnych oraz klasy języków bezkontekstowych na operacje na językach.

Klasa języków regularnych jest zamknięta na w zasadzie wszystkie popularne operacje na językach. Dowody oparte są o tworzenie nowych automatów lub wyrażeń regularnych

1. **Suma, Katenacja, Gwiazdka Kleene'go:** Dla L_1, L_2 języków regularnych opisywanych przez wyrażenia α, β, wyrażenie $\alpha + \beta$ opisuje $L_1 \cup L_2$. Analogicznie, wyrażenia $\alpha \cdot \beta$ i $(\alpha)^*$ opisują odpowiednio $L_1 \cdot L_2$ i L_1^*.
2. **Dopełnienie:** Jeśli L jest regularny, to istnieje automat DFA $A = (Q, \Sigma, \delta, s, F)$. Biorąc DFA $B = (Q, \Sigma, \delta, s, Q \setminus F)$ dostajemy, że $L(B) = \Sigma^* \setminus L(A)$, więc $\overline{L}$ też jest regularny.
3. **Przecięcie, odejmowanie:** Załóżmy, że L_1, L_2 są regularne. Biorąc DFA A_1, A_2 takie, że $L(A_1) = L_1, L(A_2) = L_2$, możemy skonstruować *automat produktowy*

$$B = (Q_1 \times Q_2, \Sigma, \delta_B, (s_1, s_2), \mathcal{F}_B)$$

taki, że:

$$\delta_B(\{p_1, p_2\}, a) = (\delta_1(p_1, a),\ \delta_2(p_2, a)).$$

Biorąc $\mathcal{F}_B = F_1 \times F_2$ dostajemy $L(B) = L_1 \cap L_2$, zaś biorąc $\mathcal{F}_B = F_1 \times (Q_2 \setminus F_2)$ dostajemy $L(B) = L_1 \setminus L_2$.

Klasa języków bezkontekstowych nie jest zamknięta na pewne operacje. Pokażemy najpierw, na które operacje jest zamknięta, a potem kontrprzykłady na pozostałe.

1. **Suma, Katenacja, Gwiazdka Kleene'go:** Niech L_1, L_2 będą językami bezkontekstowymi nad alfabetem Σ. Wtedy istnieją pewne gramatyki bezkontekstowe G_1, G_2 takie, że dla $i = 1, 2$

$$G_i = (N_i, \Sigma, P_i, S_i); \quad L_i = L(G_i)$$

Skonstruujemy gramatyki bezkontekstowe dla języków $L_1 \cup L_2$, $L_1 \cdot L_2$, L_1^*.

(a) Stwórzmy gramatykę $G_s = (N_1 \cup N_2 \cup \{S\}, \Sigma, P, S)$, gdzie P zawiera zarówno wszystkie produkcje z P_1, P_2, jak i dwie nowe produkcje

$$S \to_{G_s} S_1 \,|\, S_2$$

Nietrudno zauważyć, że $L(G_s) = L_1 \cup L_2$.

(b) W tym przypadku konstrukcja G_c będzie analogiczna, *jednak inna będzie nowa dodawana produkcja*:

$$S \to_{G_c} S_1 S_2.$$

Dostajemy $L(G_c) = L_1 \cdot L_2$.

(c) Tutaj skonstruujemy w podobny sposób $G_* = (N_1 \cup \{S\}, \Sigma, P, S)$, jednak w P zawrzemy wszystkie produkcje z P_1 oraz dwie nowe:

$$S \to S_1 S \,|\, \varepsilon.$$

Zauważmy, że $L(G_*) = L_1^*$

2. **Przecięcie:** klasa języków bezkontekstowych **nie jest zamknięta** na branie przecięć. Zauważmy, że języki $L_1 = \{a^m b^n c^n : m, n \geq 0\}$ oraz $L_2 = \{a^m b^m c^n : m, n \geq 0\}$ są bezkontekstowe (jako katenacje języków bezkontekstowych), jednak $L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\}$ nie jest bezkontekstowy.

3. **Dopełnienie, odejmowanie:** klasa języków bezkontekstowych **nie jest zamknięta** ani na branie dopełnień, ani na odejmowanie. Gdyby była zamknięta na pewną z tych operacji, to przez zamkniętość na sumę można by było uzyskać zamkniętość na przecięcie:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}; \quad L_1 \cap L_2 = (L_1 \cup L_2) \setminus ((L_1 \setminus L_2) \cup (L_2 \setminus L_1)).$$

I.6.5 Twierdzenie Myhilla-Nerode'a

Omów twierdzenie Myhilla–Nerode'a, podając ideę dowodu i związek z minimalizacją automatów skończonych.

Definicja. Niech $A = (Q, \Sigma, \delta, s, F)$ będzie ustalonym DFA. Zdefiniujmy relację A-równoważności na zbiorze stanów Q w następujący sposób:

$$p \equiv_A q \quad \text{wtedy i tylko wtedy, gdy} \quad \forall_{x \in \Sigma^*}\, \widehat{\delta}(p, x) \in F \iff \widehat{\delta}(q, x) \in F.$$

Jeśli stany nie są A-równoważne, to są A-rozróżnialne.

Okazuje się, że powyższa relacja gra kluczową rolę przy minimalizacji automatów skończonych. Dla ustalonego DFA $A = (Q, \Sigma, \delta, s, F)$ szukamy innego DFA, które akceptuje dokładnie $L(A)$ i ma najmniejszą możliwą liczbę stanów. Okazuje się, że konstruując DFA $D = \left(Q/_{\equiv_A}, \Sigma, \delta_D, [s]_{\equiv_A}, F/_{\equiv_A}\right)$, gdzie

$$\delta_D([p]_{\equiv_A}, a) = [\delta(p, a)]_{\equiv_A},$$

a następnie usuwając z niego stany nieosiągalne ze startu dostajemy automat minimalny.

Definicja. Niech L będzie ustalonym językiem nad alfabetem Σ. Zdefiniujmy na zbiorze wszystkich słów Σ^* *relację L-kongruencji* w następujący sposób:

$$x \equiv_L y \quad \text{wtedy i tylko wtedy, gdy} \quad \forall_{u \in \Sigma^*}\, xu \in L \iff yu \in L.$$

Oczywiście ta relacja to relacja równoważności. Okazuje się, że ona jest kluczowa do opisania klasy języków regularnych. Następujące twierdzenie nazywamy **Twierdzeniem Myhilla Nerode'a**.

Twierdzenie. Język L nad alfabetem Σ jest regularny wtedy i tylko wtedy, gdy relacja $\equiv_L$ ma skończenie wiele klas abstrakcji.

Pomysł na dowód. W jedną stronę, jeśli L jest regularny to istnieje pewne DFA, które go akceptuje. Skonstruujemy relację na słowach, której klasy abstrakcji wyznaczane są przez stany tego DFA, która będzie *L-kongruencją*.

W drugą stronę, jeśli mamy skończenie wiele klas abstrakcji pewnej *L-kongruencji*, to możemy skonstruować automat którego stanami będą te klasy abstrakcji. Da się pokazać, że ten automat jest dobrze zdefiniowany i akceptuje dokładnie L.

Dowód. ($\Longrightarrow$) Niech L będzie językiem regularnym. Wtedy istnieje pewny DFA $A = (Q, \Sigma, \delta, s, F)$, który spełnia $L(A) = L$. Wprowadźmy relację $\sim_A$ na zbiorze Σ^*

$$x \sim_A y \quad \iff \quad \widehat{\delta}(s, x) \equiv_A \widehat{\delta}(s, y).$$

Propozycja. Relacja $\sim_A$ jest L-kongruencją.

- Jeśli $x \sim_A y$, to $\widehat{\delta}(s,x) \equiv_A \widehat{\delta}(s,y)$, zatem z definicji A-kongruencji
$$\forall_{w\in\Sigma^*}\ \widehat{\delta}(\widehat{\delta}(s,x),w) \in F \iff \widehat{\delta}(\widehat{\delta}(s,y),w) \in F.$$
Zatem $\forall_{w\in\Sigma^*}\ xw \in L \iff yw \in L$, więc $x \equiv_L y$.
- Jeśli $x \not\sim_A y$, to $\widehat{\delta}(s,x) \not\equiv_A \widehat{\delta}(s,y)$, zatem
$$\exists_{w\in\Sigma^*}\ \neg\big(\widehat{\delta}(\widehat{\delta}(s,x),w) \in F \iff \widehat{\delta}(\widehat{\delta}(s,y),w) \in F\big)$$
Zatem $\exists_{w\in\Sigma^*}\ \neg\big(xw \in L \iff yw \in L\big)$, więc $x \not\equiv_L y$.

To znaczy, że $\sim_A$ ma co najwyżej $|Q|$ klas abstrakcji, a więc L-*kongruencja* ma skończoną liczbę stanów.

($\Longleftarrow$) Przypuśćmy, że relacja $\equiv_L$ ma skończenie wiele klas abstrakcji $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$. Skonstruujemy DFA A, a następnie udowodnimy, że $L(A) = L$.

$$A = (\mathcal{C}, \Sigma, \delta, [\varepsilon]_{\equiv_L}, \mathcal{F}), \qquad \delta(C_i, a) = C_j \iff \exists_{w\in C_i} aw \in C_j, \qquad \mathcal{F} = \{C_i \in \mathcal{C} : C_i \subseteq L\}.$$

Czemu funkcja przejścia jest poprawnie zdefiniowana? Weźmy dowolne dwa słowa x, y, takie, że $x \equiv_L y$ oraz dowolną literkę $a \in \Sigma$. Pokażemy, że $xa \equiv_L ya$.

$$x \equiv_L y \implies \forall_{w\in\Sigma^*}\ \big(xw \in F \iff yw \in F\big) \implies \forall_{w\in\Sigma^*}\ \big(xaw \in F \iff yaw \in F\big) \implies xa \equiv_L ya.$$

Zatem możemy indukcyjnie pokazać, że $\widehat{\delta}([\varepsilon]_{\equiv_L}, w) = [w]_{\equiv_L}$.

Zauważmy też, że dla każdej klasy $C \in \mathcal{C}$ jeśli $C \cap L \neq \emptyset$, to istnieje $w \in C \cap L$, a wtedy wobec $\forall_{u\in C}\ \forall_{x\in\Sigma^*}\ (wx \in L \iff ux \in L)$ zastosowanego do $x = \varepsilon$ mamy $C \subseteq L$. Teraz

$$w \in L \iff [w]_{\equiv_L} \subseteq L \iff \widehat{\delta}\left([\varepsilon]_{\equiv_L}, w\right) \subseteq L \iff \widehat{\delta}\left([\varepsilon]_{\equiv_L}, w\right) \in \mathcal{F} \iff w \in L(A).$$

□

Na koniec warto powiedzieć, że skonstruowany w implikacji automat jest automatem minimalnym dla języka L. Istotnie, zauważmy, że gdyby dla pewnych stanów $C_i, C_j \in \mathcal{C}$ zachodziło $C_i \equiv_A C_j$, to biorąc dowolne słowa u, v dla których zachodzi $C_i = [u]_{\equiv_L}$ i $C_j = [v]_{\equiv_L}$ dostajemy

$$\forall_{x\in\Sigma^*}\ \big(\widehat{\delta}(C_i,x) \in F \iff \widehat{\delta}(C_j,x) \in F\big) \implies \forall_{x\in\Sigma^*}\big(ux \in L \iff vx \in L\big) \implies u \equiv_L v,$$

więc $C_i = C_j$.

I.6.6 Maszyny Turinga

Determinizm i niedeterminizm dla maszyn Turinga: omów oba modele i związek między nimi.

Maszyna Turinga to siódemka $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Jest to model obliczeń, w którego skład wchodzą:

1. **Taśma** – dwustronnie nieskończona taśma złożona z komórek, w każdą z nich wpisany jest jakiś symbol ze zbioru Γ.
2. **Głowica** – element trzymający stan $q \in Q$, w którym obecnie znajduje się Maszyna Turinga. Głowica *wskazuje* na pewną komórkę taśmy.
3. **Funkcja (częściowa) / Relacja przejścia** δ. Opisuje zbiór reguł, względem których działa głowica. O regule możemy myśleć następująco:

   Gdy głowica znajdująca się w stanie p i patrzy na komórkę z symbolem X, to na jaki symbol Y ma go zamienić, do jakiego stanu q ma przejść, oraz czy ma się przesunąć do komórki w lewo (L), czy w prawo (R)?

W zależności od tego, czy maszyna jest deterministyczna, czy niedeterministyczna, typ delty to:

DMT: $\delta : (Q \times \Gamma) \to (Q \times \Gamma \times \{L, R\})$

NMT: $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$

Konfiguracja MT: Stan maszyny przedstawiamy za pomocą ciągu $X_1 X_2 \dots X_k \, q \, X_{k+1} \dots X_n$, gdzie $X_i \in \Gamma$, $q \in Q$. Utożsamiamy go z Maszyną, w której na taśmie:

- Mamy zapisany ciąg $X_1 X_2 \dots X_n$, na lewo od X_1 i na prawo od X_n wszystkie komórki to *blanki* B.
- Głowica *patrzy* na X_{k+1} oraz jest w stanie q.

Konfiguracja początkowa MT dla słowa $w = \{w_i\}_{i=1}^n$ na wejściu to ciąg $q_0 \, w_1 w_2 \dots w_n$.

Ruch (krok) MT wyznaczany jest przez relację $\vdash_M \subseteq \Gamma^* Q \Gamma^* \times \Gamma^* Q \Gamma^*$ na zbiorze konfiguracji w zależności od funkcji/relacji przejścia, na przykład:

$$X_1 X_2 \dots X_k \, p \, X_{k+1} \dots X_n \vdash_M X_1 X_2 \dots q \, X_k \, Y \dots X_n$$

gdy $\big((p, X_{k+1}), (q, Y, L)\big) \in \delta$. Definiujemy $\vdash_M^*$ jako zwrotne i przechodnie domknięcie $\vdash_M$.

Definicja. Maszyna Turinga M akceptuje słowo w, jeśli $q_0 w \vdash_M \alpha f \beta$, dla pewnych $\alpha, \beta \in \Gamma^*, f \in F$. Definiujemy $L(M)$ jako zbiór słów akceptowanych przez M. O języku L mówimy, że jest **rekurencyjnie przeliczalny** (w RE), jeśli istnieje pewna MT M taka, że $L = L(M)$.

No właśnie, ale jaka Maszyna Turinga? *Deterministyczna* czy *Niedeterministyczna*? Oczywiście DMT jest w szczególności niedeterministyczna, ale prawdziwe jest następujące twierdzenie.

Twierdzenie. Dla Niedeterministycznej Maszyny Turinga M $L(M)$ jest **rekurencyjnie przeliczalny**.

Pomysł na dowód jest taki, żeby skonstruować **Deterministyczną Maszynę Turinga** $\beth$, która *symuluje* działanie M. Jej konstrukcja jest jednak całkiem długa, jeśli chce się ją zrobić w miarę formalnie.

Lemat. Deterministyczna Maszyna Turinga i wielotaśmowa Deterministyczna Maszyna Turinga to modele równoważne. Wtedy możemy *bez straty ogólności* korzystać z wielu taśm.

Dowód. Na początek wytłumaczmy, czym jest **wielotaśmowa Maszyna Turinga**. Ustalmy liczbę całkowitą k. k-taśmową Maszyną Turinga $M_k = (Q_k, \Sigma, \Gamma_k, \delta, s, B, F_k)$ nazywamy taką deterministyczną Maszynę Turinga, w której:

- Mamy do dyspozycji k taśm. Dla słowa wejściowego w konfiguracja początkowa trzyma je na jednej z taśm, a pozostałe są wypełnione blankami.
- **Głowica** patrzy na wiele taśm naraz. Początkowo na pierwszej taśmie patrzy na w_0, na pozostałych *gdziekolwiek*.
- Na **opis chwilowy** składa się opis każdej z k taśm, jednak porównując z klasycznym modelem, *na każdej taśmie głowica ma ten sam stan.* **Akceptacja** słowa przebiega analogicznie, czyli musimy dojść do stanu akceptującego.
- **Funkcja przejścia** jest typu

$$\delta : Q_k \times \Gamma_k^k \to Q_k \times (\Gamma_k \times \{L, R\})^k.$$

  Czyli, intuicyjnie, ruch wykonujemy w oparciu o krotkę symboli na które patrzy głowica, wtedy na każdej taśmie wybieramy *na co podmienić dany symbol* oraz *w którą stronę pójść.* Następnie zmieniamy stan głowicy.

Jak zasymulować takie ustrojstwo za pomocą Deterministycznej Maszyny Turinga? Pomysł jest taki, aby na bazie maszyny M_k skonstruować deterministyczną maszynę Turinga z jedną taśmą, w której informację o wielu taśmach trzymamy w następujący sposób:

- $Q = Q_k \times \{L, E\} \times (\Gamma_k \cup \{\square\})^k \cup R$ – w stanie będziemy trzymać nie tylko w jakim obecnie stanie jest M_k, ale również na jakie symbole patrzą odpowiednie głowice. Trzymamy też, czy jesteśmy w stanie *load*, czy *execute*. Dodatkowo R to jakiś zbiór stanów roboczych.

- $\Gamma = (\Gamma_k \times \{0,1\})^k$ – w każdym bloku na taśmie *spłaszczamy* stan k taśm, ponadto przy każdym symbolu trzymamy wartość logiczną (w krotce, pod indeksem $2i$), czy na i-tej taśmie głowica patrzy na daną komórkę.
- **Warunki początkowe:** Na początku, gdy dostajemy na wejściu słowo $w_0, \ldots, w_n$, nasza maszyna przechodząc po taśmie na prawo zamienia komórki na $2k$-krotki w następujący sposób:

$$w_0 \to (w_0, 1, B, 1, \ldots, B, 1); \quad \forall_i \; w_i \to (w_i, 0, B, 0, \ldots, B, 0),$$

  a następnie wraca na najbardziej lewą komórkę (po lewej stronie od wszystkich obliczeń możemy utrzymywać wskaźnik, który oznacza *komórkę za najbardziej lewą odwiedzoną komórką*) i przechodzi do stanu $q_{s,\,L,\,\square,\ldots,\square}$
- **Symulacja kroku *load*:** Będąc w stanie $q_{p,\,L,\,\square,\ldots,\square}$ maszyna przechodzi w prawo, dopóki stan ma nie wszystkie pola wypełnione, w momencie gdy na jakiejś taśmie natrafi na jedynkę, to wpisuje pod odpowiedni kwadrat symbol jaki tej jedynce odpowiada. Gdy zbierze komplet, wraca na początek. Dzięki temu, że mamy komplet, możemy podjąć decyzję o ciągu akcji które podejmiemy w fazie *execute*, na podstawie $\delta_k(p, a_1, a_2, \ldots a_k)$.
- **Symulacja kroku *execute*:** Będzie wyglądać podobnie. Idziemy w prawo, jeżeli trafiamy na wskaźnik na i-tej z symulowanych taśm oraz pod odpowiednim indeksem nie mamy kwadrata, wykonujemy ruch odpowiednio z regułą $\delta_k(p, a_1, a_2, \ldots a_k)$ dla i-tej taśmy, podmieniając odpowiednie symbole w krotce i i-ty symbol w ciągu zamieniając na kwadrat. Po opróżnieniu stanu idziemy do końca w lewo, a następnie przechodzimy do stanu $q_{q,\,L\,\square,\ldots,\square}$ (gdzie q to stan, do którego weszłaby M_k).
- **Akceptacja** opiera się o dojście do stanu $q_{f,\,L,\,\ldots}$, gdzie $f \in F_k$.

□

Lemat. Zadaną Niedeterministyczną Maszynę Turinga $\daleth$ możemy zasymulować na 3-taśmowej Deterministycznej Maszynie Turinga.

Dowód. Ten opis będzie mniej dokładny, bardziej koncepcyjny, ponieważ wcześniej wskazaliśmy różne *techniki programowania na Maszynach Turinga.*

- Utrzymujemy trzy taśmy, na pierwszej dostajemy słowo wejściowe w, na drugiej będziemy trzymali kolejkę konfiguracji (oddzielonych nowo dodanym znakiem #), zaś trzecia będzie służyć jako taśma robocza. Na drugiej taśmie będziemy przed ostatnio odwiedzoną konfiguracją trzymać symbol #.
- W pierwszej fazie przepisujemy na drugą taśmę słowo $\#q_0 w_0 \ldots w_n\#$, następnie ustawiamy wszystkie wskaźniki *na początek*, przechodząc już do głównej pętli algorytmu.
- Algorytm to tak naprawdę *algorytm BFS* w grafie wszystkich możliwych konfiguracji $\daleth$, między którymi krawędź jest wtedy, gdy istnieje w δ przejście między nimi. Algorytm polega na:
  1. Przepisaniu konfiguracji na prawo od # na trzecią taśmę, a następnie zamienieniu kolorów tak, aby następny # był tym czerwonym.
  2. Zasymulowaniu wszystkich możliwych kroków dla konfiguracji na trzeciej taśmie, dopisując uzyskane w ten sposób stany do kolejki na drugiej taśmie (możemy korzystać z dodatkowej pamięci uzyskanej poprzez zwiększenie liczby stanów i dodanie do nich informacji o tym, który stan był sprawdzany ostatnio. *Pamiętajmy, że konstruujemy M w oparciu o $\daleth$*).
  3. Wyczyszczeniu trzeciej taśmy oraz poprawieniu wskaźnika na drugiej taśmie, aby móc powtórzyć rozumowanie.

Warunkiem akceptacji w jest przepisanie z kolejki konfiguracji, w której stan jest akceptujący. Zauważmy, że skoro idziemy *algorytmem BFS*, to przechodzimy w kolejności rosnących odległości, więc jeśli $\daleth$ akceptuje w, to istnieje taki wybór przejść, że ta akceptacja następuje po n krokach, a dzięki temu liczbę konfiguracji które musi zasymulować M do nastąpienia akceptacji możemy ograniczyć przez $(|\delta|+1)^{n+1}$. □

I.6.7 Problem stopu

Omów złożoność obliczeniową problemu stopu oraz jego dopełnienia (z dowodami).

Zarówno sformułowanie, jak i dalsze rozważania problemu stopu ściśle związane są z faktem, że potrafimy jednoznacznie kodować Maszyny Turinga. W związku z tym możemy rozpatrywać problemy ich dotyczące w kategoriach analizy ich kodów. Jednym z takich problemów jest PROBLEM STOPU. Jednak zanim przeprowadzimy jego analizę, skupmy się na samym kodowaniu.

Definicja. Kodowaniem Maszyny Turinga $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ jest słowo nad alfabetem $\{0,1\}^*$. Przypuśćmy, że stany M to $Q = \{q_1, q_2, \ldots q_k\}$, symbole taśmowe to $\Gamma = \{X_1, \ldots, X_n\}$, zaś q_2 jest jedynym stanem akceptującym. Niech $\big(X_1, X_2, X_3\big) = \big(0, 1, B\big)$. Przejścia w lewo i prawo będziemy oznaczać jako D_1 oraz D_2. Kodowanie M wygląda następująco.

- Na początku występuje blok kodujący liczbę stanów oraz symboli taśmowych
$$0^k 1110^n 111.$$
- Następnie występuje blok kodujący relację przejścia δ. Występuje on w postaci kodu krotek $\big(q_i, X_j, q_k, X_l, D_t\big)$ wypisanego jeden po drugim i oddzielonego 11. Krotki te kodujemy jako
$$0^i 10^j 10^k 10^l 10^t.$$
Zauważmy, że $i, j, k, l, t > 0$, zatem aby oddzielić krotki od siebie wystarczy popatrzeć na miejsca w których występuje 11.

Zauważmy, że istnieje wiele kodowań jednej Maszyny Turinga, a nie każde słowo nad $\{0,1\}^*$ jest poprawnym kodowaniem. W takiej sytuacji będziemy kodowaną maszynę utożsamiać z maszyną akceptującą język pusty.

Definicja. Na słowach $\{0,1\}^*$ możemy wprowadzić porządek liniowy
$$u \preccurlyeq v \quad \iff \quad \big(|u| < |v| \vee (|u| = |v| \wedge u \leq_{lex} v)\big).$$
Wtedy każdej maszynie M możemy nadać numer, odpowiadający pewnemu jej kodowaniu występującemu w ciągu $w_1, w_2, \ldots$. Myślmy o tym w ten sposób: kolejne słowa kodują ciąg maszyn $M_1, M_2, \ldots$.

Wykorzystując powyższe kodowanie, stworzymy język który nie jest RE.

Definicja. Język $L_d = \{w_i : w_i \notin L(M_i)\}$ nazywamy językiem przekątniowym.

Twierdzenie. Język L_d nie jest w RE.

Dowód. Załóżmy ku sprzeczności, że L_d jest w RE. Wtedy istnieje pewna Maszyna Turinga M taka, że $L_d = L(M)$. Ma ona swój numer $M = M_i$, więc odpowiada jej kodowanie w_i. Rozpatrzmy przypadki:

- $w_i \in L_d \implies w_i \in L(M_i) \implies w_i \notin L_d$.
- $w_i \notin L_d \implies w_i \notin L(M_i) \implies w_i \in L_d$.

W obu przypadkach mamy sprzeczność. □

Kolejną cegiełką, z której skorzystamy, będzie *Maszyna Uniwersalna*. Służy ona do symulowania akceptacji słowa w przez Maszynę M. Na wejściu dostaje ona parę słów: słowo w, którego przynależność do $L(M)$ sprawdzamy oraz kodowanie maszyny Turinga M, którą będziemy symulować.

Maszynę Uniwersalną M_u zrealizować możemy za pomocą 4-taśmowej Maszyny Turinga, analogicznie do tej skonstruowanej przy symulacji maszyny niedeterministycznej. Główną różnicą jest fakt, że informacji o możliwych ruchach nie trzymamy w kodzie M_u, tylko odczytujemy z kodowania trzymanego na jednej z taśm. Realizacja dekodera jest jednak trudna, więc w dalszej części rozwiązania będziemy odwoływać się do istnienia i do ewentualnych modyfikacji *Maszyny Uniwersalnej.*

Poznaliśmy już **problemy rekurencyjnie przeliczalne** (RE). Skupmy się teraz na bardziej interesującej z punktu widzenia programistów podklasie problemów.

Definicja. Deterministyczna MT M **ma własność stopu**, jeżeli dla każdego słowa wejściowego w wykonuje ona ciąg skończenie wielu kroków, a następnie rozstrzyga czy akceptuje czy odrzuca w. Mówimy o języku L, że jest **rozstrzygalny** (w R), gdy istnieje DMT M mająca własność stopu taka, że $L = L(M)$.

W oczywisty sposób R $\subseteq$ RE. Za chwilę pokażemy przykład języka rozróżniającego klasy R i RE. Do tego przyda nam się następujący lemat.

Lemat. Niech L będzie językiem. Jeśli $L, \overline{L} \in$ RE, to $L, \overline{L} \in$ R.

Dowód. Skoro $L, \overline{L} \in$ RE, to istnieją Maszyny Turinga $M, \overline{M}$ takie, że $L = L(M)$ oraz $\overline{L} = L(\overline{M})$. Zauważmy teraz, że dowolne słowo $w \in \Sigma^*$ zostanie zaakceptowane przez pewną z maszyn $M, \overline{M}$ po skończenie wielu krokach. To prowadzi nas do konstrukcji maszyny M_R takiej, że $L = L(M_R)$.

- Maszyna będzie wykorzystywała Maszynę Uniwersalną do symulacji maszyn $M, \overline{M}$.
- Dostając w na wejściu, najpierw symulujemy pierwszy krok M na w, następnie pierwszy krok $\overline{M}$ na w, następnie drugi krok M na w i tak dalej.
- W końcu któraś z maszyn zaakceptuje w. Jeśli to będzie M to akceptujemy, jeśli $\overline{M}$, to odrzucamy.

Analogiczną konstrukcję możemy przeprowadzić dla języka $\overline{L}$. □

Definicja. Problem stopu to język $L_{\text{HALT}} = \{\langle w, M\rangle : M \text{ zatrzymuję się na } w\}$, gdzie w to słowo wejściowe, zaś M zakodowana Maszyna Turinga.

Twierdzenie. Problem stopu jest w RE \ R. Dopełnienie problemu stopu nie jest w RE.

Dowód. Aby udowodnić rekurencyjną przeliczalność, możemy skonstruować Maszynę Turinga H, która dla danych wejściowych $\langle w, M\rangle$ symuluje w na M z użyciem Maszyny Uniwersalnej:

1. Uruchamiamy M na wejściu w.
2. Jeżeli M się zatnie, oznacza to, że $\langle w, M\rangle \in L_{\text{HALT}}$. Wtedy H wchodzi w stan akceptujący, tym samym **akceptując**.
3. W przeciwnym wypadku M nigdy nie zatrzyma się na w. Wtedy H będzie symulować M w nieskończoność, zatem też **nie zaakceptuje** danych wejściowych.

Zatem maszyna H akceptuje parę $\langle w, M\rangle$ wtedy i tylko wtedy, gdy M zatrzymuje się na w, co dowodzi, że $L(H) = L_{\text{HALT}}$. Zatem pokazaliśmy, że $L_{\text{HALT}} \in$ RE.

Aby pokazać drugą część twierdzenia załóżmy ku sprzeczności, że $L_{\text{HALT}} \in$ R. Oznacza to, że istnieje deterministyczna Maszyna Turinga S, która ma własność stopu i poprawnie rozstrzyga problem stopu. Wykorzystując maszynę S, skonstruujemy nową Maszynę Turinga M_d, która rozstrzygałaby język przekątniowy L_d.

Dla dowolnego wejścia w_i (będącego i-tym słowem z uporządkowania i kodowaniem maszyny M_i) maszyna M_d działa następująco.

1. Uruchamia maszynę S na wejściu $\langle w_i, M_i\rangle$. Ponieważ S rozstrzyga problem stopu, to zawsze po skończonej liczbie kroków udzieli jednoznacznej odpowiedzi.
2. Jeśli S **odrzuci** wejście (co oznacza, że M_i w ogóle nie zatrzymuje się na wejściu w_i), to maszyna M_i w oczywisty sposób nie może zaakceptować w_i. Z definicji oznacza to, że $w_i \in L_d$. W takiej sytuacji nasza maszyna M_d **akceptuje** wejście.
3. Jeśli S **zaakceptuje** wejście (co oznacza, że M_i na pewno zatrzyma się na w_i), maszyna M_d używa Maszyny Uniwersalnej do zasymulowania działania M_i na słowie w_i. Ponieważ z kroku poprzedniego ta instancja ma własność stopu, symulacja bezpiecznie zakończy się w skończonym czasie.
4. Po zakończeniu symulacji sprawdzamy wynik: jeśli M_i zaakceptowała w_i (zatem $w_i \notin L_d$), maszyna M_d **odrzuca** wejście. Z kolei jeśli M_i odrzuciła w_i (zatem $w_i \in L_d$), maszyna M_d **akceptuje** wejście.

Wykazaliśmy, że maszyna M_d dla każdego możliwego słowa w_i w skończonej liczbie kroków prawidłowo określa jego przynależność do L_d. Oznaczałoby to, że język przekątniowy jest rozstrzygalny, a więc $L_d \in$ R. Jednak $L_d \notin$ RE, sprzeczność.

Zauważmy teraz, że na podstawie lematu powyżej, skoro $L_{\text{HALT}} \in \text{RE} \setminus \text{R}$, to $\overline{L_{\text{HALT}}} \notin \text{RE}$. To kończy dowód. □

I.6.8 Klasy złożoności

Omów klasy złożoności: PTIME, NPTIME oraz coNPTIME. Podaj przykład problemu, który jest w PTIME oraz przykłady jezyków zupełnych dla NPTIME i coNPTIME. Nakreśl dowód twierdzenia Cooka.

Definicja. Obliczeniem Maszyny Turinga M nazywamy ciąg kroków $q_0 w \vdash_M \cdots \vdash_M \Gamma^*\{q_{acc}, q_{rej}\}\Gamma^*$. Mówimy, że niedeterministyczna Maszyna Turinga M działa w czasie $T : \mathbb{N} \to \mathbb{N}$, jeśli dla słowa wejściowego w każde obliczenie składa się z co najwyżej $T(|w|)$ kroków.

Definicja. Mówimy, że deterministyczna (lub niedeterministyczna) Maszyna Turinga M działa w czasie wielomianowym, jeśli istnieje wielomian $P \in \mathbb{R}[X]$, taki, że M działa w czasie P.

Te definicje pozwolą nam wprowadzić dwie bardzo szczególne klasy złożoności problemów rozstrzygalnych, mianowicie PTIME oraz NPTIME.

Definicja. Język $L \subseteq \Sigma^*$ *należy do PTIME*, jeśli istnieje **deterministyczna** Maszyna Turinga M działająca w czasie wielomianowym taka, że $L = L(M)$.

Definicja. Język $L \subseteq \Sigma^*$ *należy do NPTIME*, jeśli istnieje **niedeterministyczna** Maszyna Turinga M działająca w czasie wielomianowym taka, że $L = L(M)$.

Nie jest jasne, czy klasy PTIME oraz NPTIME są równe. Prekursorem, który postawił pytanie o zależność między nimi jest Stephen Cook, który prowadził wówczas badania nad klasyfikacją systemów automatycznego dowodzenia twierdzeń względem ich złożoności.[1] Przeniósł on swoje rozumowanie na problemy rozstrzygalne, w szczególności tworząc ich podział na klasy (m.in. PTIME i NPTIME) ze względu na relację, którą teraz nazywamy *redukcją Cook'a*. Szczególnym przypadkiem tej redukcji jest *redukcja Karpa*, którą omówimy.

Definicja. Mówimy, że problem $L_1 \in \Sigma_1^*$ redukuje się wielomianowo (w czasie wielomianowym) do problemu $L_2 \in \Sigma_2^*$, co oznaczamy $L_1 \leq_P L_2$, jeśli istnieje deterministyczna MT M działająca w czasie wielomianowym taka, każde dane wejściowe $w \in \Sigma_1^*$ przekształca na $\phi(w) \in \Sigma_2^*$ w taki sposób, że

$$w \in L_1 \iff \phi(w) \in L_2.$$

Zauważmy, że nasza maszyna prowadzi obliczenie, jednak nie liczy się akceptacja lub nie, tylko to co zostanie na taśmie, czyli $\phi(w)$. Intuicyjnie, *redukując wielomianowo, przekształcamy instancję L_1 na instancję problemu L_2, więc L_1 jest co najwyżej tak trudny jak L_2.*

Relacja $\leq_P$ dzieli zbiór problemów rozstrzygalnych na mniejsze klasy. Aby o tym porozmawiać, musimy wprowadzić jeszcze jedną definicję.

Definicja. O problemie L mówimy, że jest C-trudny, jeśli każdy problem $L' \in C$ spełnia $L' \leq_P L$. O problemie L mówimy, że jest C-zupełny, jeśli L jest C-trudny i dodatkowo $L \in C$.

Oczywiście każdy problem o którym wiemy, że jest PTIME, jest *P-zupełny*, bo algorytm świadczący o redukcji $L_1 \leq_P L_2$ może dla zadanego wejścia w w czasie wielomianowym rozstrzygnąć przynależność w do L_1 a następnie w zależności od wyniku wziąć $\phi(w)$ jako pozytywną lub negatywną instancję L_2 (gdy zhardcode'ujemy dwie, to obydwie będą miały stałą długość; patologicznym przypadkiem jest $L_2 \in \{\emptyset, \Sigma_2^*\}$). Okazuje się, że istnieją też problemy *NP-zupełne*. Jako pierwszy udowodnił to właśnie Stephen Cook.

[1] Problem istnienia modelu dla zdania w logice pierwszego rzędu / wyższych rzędów jest nierozstrzygalny, jednakże przy próbach dowodzenia prawdziwego twierdzenia można na tych systemach wprowadzić pewną metrykę, mierzącą w jak szybko dany system znajdzie ten model.

Definicja. Problem SAT to problem stwierdzenia, czy zadana formuła zdaniowa ϕ jest spełnialna. Dla uproszczenia, możemy myśleć o tym problemie w taki sposób, że każda formuła zadana na wejściu jest w postaci CNF (jednak w dalszych rozważaniach problem pozostanie równoważny, nawet przy użyciu dowolnych operatorów boolowskich).

Twierdzenie. Problem SAT jest w NP.

Dowód. Załóżmy, że zadana formuła zdaniowa ϕ ma n zmiennych. Wtedy w szczególności $n = O(|\phi|)$. Możemy skonstruować niedeterministyczną Maszynę Turinga, która:

- *w fazie niedeterministycznej* wygeneruje wartościowanie n zmiennych.
- *w fazie deterministycznej* podstawi te wartości za zmienne, a następnie wyewaluuje (np. idąc algorytmem postorder DFS po drzewie operatorów) wartość ϕ.
- akceptuje, jeśli ewaluacja ϕ jest prawdziwa.

Obydwie fazy zajmują $P(|\phi|)$ kroków dla pewnego wielomianu P, zatem maszyna ta świadczy o przynależności SAT $\in$ NPTIME. □

Twierdzenie (Cook, Levin)**.** Problem SAT jest NP-zupełny.

Inne przykłady problemów NP-zupełnych:

- 3-SAT: SAT, ale na wejściu dostajemy formułę w postaci CNF w której każda klauzula ma dokładnie 3 literały.
- Vertex-Cover: na wejściu dostajemy graf G oraz liczbę k zapisaną binarnie. Naszym zadaniem jest rozstrzygnąć, czy istnieje pokrycie wierzchołkowe grafu o rozmiarze co najwyżej k.
- Hamiltonian-Cycle: na wejściu dostajemy graf G. Naszym zadaniem jest rozstrzygnąć, czy G ma cykl Hamiltona.

Twierdzenie. Problem 3-CNF-Tautology polegający na tym, że dla zadanej formuły ϕ w postaci 3-CNF (koniunkcja klauzul składających się z 3 alternatyw) chcemy odpowiedzieć, czy jest tautologią, jest w PTIME.

Dowód. Przedstawiony problem jest równoważny sprawdzeniu, czy $\neg\phi$ (będąca w postaci 3-DNF, czyli alternatywa klauzul składających się z 3 koniunkcji) jest niespełnialna. $\neg\phi$ obliczamy w czasie wielomianowym. $\neg\phi$ jest niespełnialna wtedy i tylko wtedy, gdy każda klauzula zawiera zarówno literał x, jak i $\neg x$. Na to już potrafimy napisać algorytm wielomianowy. □

Przed przedstawieniem dowodu *Twierdzenia Cook'a* skupimy się na jeszcze jednej klasie złożoności, coNPTIME.

Definicja. Język $L \subseteq \Sigma^*$ *należy do coNPTIME*, jeśli język $\overline{L}$ należy do NPTIME. Innymi słowy, język L należy do coNP, jeśli istnieje wielomianowa niedeterministyczna Maszyna Turinga M, która ma akceptujące obliczenie dokładnie dla słów $x \notin L$.

Uwaga. Jeśli analogicznie zdefiniujemy problemy coP, to P = coP. Nie wiadomo, czy NP = coNP.

Lemat. Jeśli L jest NP-trudny, to $\overline{L}$ jest coNP-trudny.

Dowód. Weźmy dowolny problem $L' \in$ NP, wtedy $L' \leq_P L$, więc istnieje taka wielomianowa redukcja ϕ, że $\forall_{w \in \Sigma^*}\, w \in L' \iff \phi(w) \in L$. To znaczy, że $\forall_{w \in \Sigma^*}\, w \notin L' \iff \phi(w) \notin L$, czyli mamy $\forall_{w \in \Sigma^*}\, w \in \overline{L'} \iff \phi(w) \in \overline{L}$. Zatem ϕ jest też wielomianową redukcją między $\overline{L'}$ oraz $\overline{L}$. Dowód kończy fakt, że dopełnienie każdego problemu z coNP jest w NP, zatem każdy problem coNP redukuje się wielomianowo do $\overline{L}$. □

Wniosek. Dopełnienia problemów NP-zupełnych są coNP-zupełne.

Przykłady problemów coNP-zupełnych:

- UNSAT: dana jest na wejściu formuła ϕ. Naszym zadaniem jest rozstrzygnięcie, czy formuła ta jest niespełnialna.
- TAUTOLOGY: dana jest na wejściu formuła ϕ. Naszym zadaniem jest rozstrzygnięcie, czy formuła ta jest tautologią.

Dowód Twierdzenia Cooka. Pokazaliśmy już, że SAT $\in$ NP. Pokażemy teraz, że SAT jest NP-trudny. W tym celu weźmy dowolny problem $L \in$ NP, pokażemy redukcję $L \leq_P$ SAT.

Skoro $L \in$ NP, to istnieje niedeterministyczna MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ działająca w czasie $T \in \mathbb{R}[X]$ taka, że $L = L(M)$. Zatem dla każdego słowa wejściowego $w \in L$ istnieje obliczenie akceptujące mające co najwyżej $T(|w|)$ kroków.

Pomysł jest taki: utożsamimy obliczenie akceptujące M z wypełnieniem grida nad $\Sigma \cup Q$ o wymiarach $I \times J$, gdzie $I = \{-T(|w|), \dots, 0, \dots, T(|w|)\}$ oraz $J = \{0, 1, \dots, T(|w|)\}$. W tym gridzie, wiersz po wierszu, będziemy trzymać opisy chwilowe kolejnych kroków obliczenia M. Następnie zrobimy formułę $\phi(w)$, której spełnialność jest równoważna istnieniu takiego obliczenia akceptującego M na w, które się mieści w tabelce. Dodatkowo, zagwarantujemy $|\phi(w)| = O(P(|w|))$, abyśmy mogli tę redukcję zrealizować w czasie wielomianowym. Wtedy $w \in L \iff \phi(w) \in$ SAT.

W formule $\phi(w)$ będziemy mieli zmienne $x_{i,j,a}$ dla $i \in I, j \in J, a \in \Gamma \cup Q$. Prawdziwość takiej zmiennej oznacza, że w tabelce na polu o koordynatach (i, j) (czyli w i-tej kolumnie i w j-tym wierszu) stoi symbol a. Stworzymy formuły $\phi_1, \phi_2, \phi_3, \phi_4$ takie, że $\phi(w) = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$. Formuły te będą oznaczać kolejno:

- ϕ_1 – na każdym polu jest dokładnie jeden symbol.
- ϕ_2 – w pierwszym wierszu jest konfiguracja startowa.
- ϕ_3 – w ostatnim wierszu jest konfiguracja, w której jesteśmy w stanie akceptującym.
- ϕ_4 – przejścia między konfiguracjami są poprawne.

Aby wszystko spiąć, dodajemy do δ takie przejścia dla stanów akceptujących, że nie ważne na jaki symbol patrzymy, zamieniamy go na blanka i idziemy w prawo. Nie wpływa to na akceptację ani na wielomianowość, ale gwarantuje, że jeśli zaakceptujemy szybciej niż po $T(|w|)$ krokach, to w ostatnim wierszu tabelki będzie konfiguracja akceptująca.

Zaczynamy od ϕ_1. Określamy

$$A_{i,j} = \bigvee_{a \in \Gamma \cup Q} x_{i,j,a}, \qquad B_{i,j} = \bigwedge_{\substack{(a,\, b) \in (\Gamma \cup Q)^2 \\ a \neq b}} \neg x_{i,j,a} \vee \neg x_{i,j,b}.$$

Wtedy

$$\phi_1 = \bigwedge_{(i,j) \in I \times J} A_{i,j} \wedge B_{i,j}$$

Teraz wymusimy konfigurację początkową i końcową

$$\phi_2 = \bigwedge_{i \in I \setminus ([|w|] \cup \{0\})} x_{i,0,B} \wedge x_{0,0,q_0} \wedge \bigwedge_{i \in [|w|]} x_{i,0,w_i}, \qquad \phi_3 = \bigvee_{f \in F} \bigvee_{i \in I} x_{i,T(|w|),f}.$$

Na koniec potrzebujemy zagwarantować poprawność przejść. Chcemy, aby między każdymi dwoma wierszami występowało przejście w lewo lub w prawo, więc dodamy dwie reguły:

$$\phi_4 = \bigwedge_{(i,\, j) \in I \times J} \bigwedge_{\substack{a \in \Gamma \\ q \in Q}} (P_l(i, j, a, q) \vee P_r(i, j, a, q)) \wedge K(i, j)$$

Intuicja jest następująca:

- $K(i, j)$ – jeśli $(i,\, j)$ wraz z sąsiednimi komórkami w wierszu nie jest stanem, to przepisz symbol występujący w $(i,\, j)$ do następnego wiersza: $(i,\, j+1)$.
- $P_l(i, j, a, q)$ – jeśli $(i,\, j)$ jest stanem q patrzącym na a, to wybierz którąś z reguł $(p, X, -1)$ z $\delta(a, q)$, a następnie rozpisz według niej komórki $(i-1,\, j+1), (i,\, j+1), (i+1,\, j+1)$.

Przydadzą nam się dwie pomocnicze formuły:

- $NQ(i, j)$ – w bloku $(i,\ j)$ nie jest wpisany stan.

$$NQ(i, j) = \bigwedge_{q \in Q} \neg x_{i,j,q}.$$

- $E(i, j, i', j')$ – bloki $(i,\ j)$ i $(i',\ j')$ mają te same symbole.

$$E(i, j, i', j') = \bigwedge_{a \in Q \cup \Gamma} (x_{i,j,a} \iff x_{i',j',a}).$$

Z ich pomocą możemy zdefiniować

$$K(i, j) = (NQ(i-1,\ j) \land NQ(i, j) \land NQ(i+1, j)) \implies E(i, j, i, j+1))$$

oraz

$$P_l(i, j, a, q) = \bigvee_{(q',a',-1) \in \delta(a,\ q)} \big(x_{i,j,q} \land x_{i+1,j,a}\big) \implies \big(x_{i-1,j+1,q'} \land E(i-1, j, i, j+1) \land x_{i+1,j+1,a'}\big).$$

Ruch prawostronny możemy zakodować analogicznie. □

II
Przedmioty programistyczno-algorytmiczne

II.1 Analiza Algorytmów

II.1.1 Twierdzenie o rekurencji uniwersalnej

Twierdzenie o rekurencji uniwersalnej, szkic dowodu.

Twierdzenie (O rekurencji uniwersalnej)**.** Niech $a > 0$, $b > 1$ będą pewnymi stałymi, a $f(n)$ funkcją nieujemną, określona dla każdego odpowiednio dużego $n \in \mathbb{R} \geqslant n_0 > 1$. Rozważmy rekurencję

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n), & n \geqslant n_0 \\ \Theta(1), & n < n_0 \end{cases} .$$

Niech $d = \log_b a$. Wówczas:

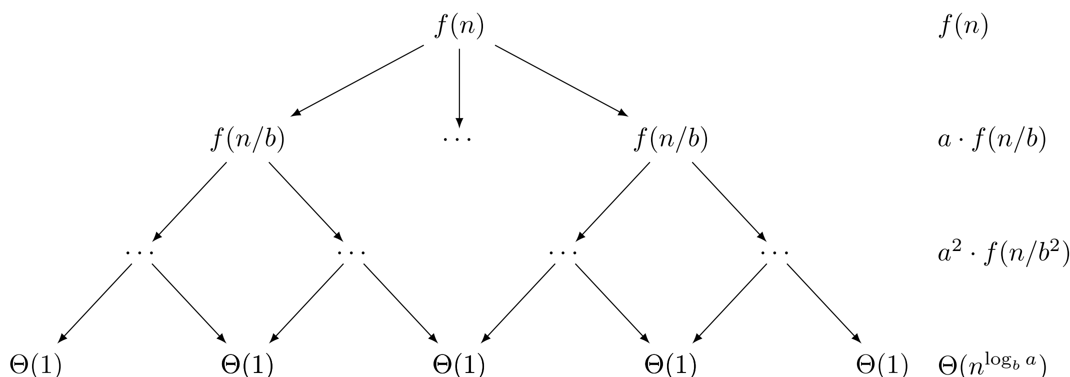
1. Jeśli $f(n) = \mathcal{O}(n^{d'})$ dla $d' < d$, to $\boxed{T(n) = \Theta(n^{\log_b a})}$.
2. Jeśli $f(n) = \Theta(n^d \lg^k n)$ dla pewnego $k \geqslant 0$, to $\boxed{T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)}$.
3. Jeśli $f(n) = \Omega(n^{d'})$ dla $d' > d$ oraz f spełnia warunek regularności:

$$af(n/b) \leqslant cf(n) \quad \text{dla pewnej stałej } c < 1,$$

to $\boxed{T(n) = \Theta(f(n))}$.

Twierdzenie działa również, jeśli $aT\left(\frac{n}{b}\right)$ zastąpimy przez $a'T\left(\left\lfloor\frac{n}{b}\right\rfloor\right) + a''T\left(\left\lceil\frac{n}{b}\right\rceil\right)$ dla pewnych $a', a'' \geqslant 0,\ a' + a'' = a$

Dowód. Rozważmy drzewo rekurencji dla powyższego równania. Zakładamy dla uproszczenia, że n jest potęgą b, a warunek brzegowy zachodzi dla $n = 1$.



Całkowity koszt $T(n)$ to suma kosztów na wszystkich poziomach drzewa. Głębokość drzewa wynosi $\log_b n$. Na poziomie i (gdzie $i = 0, 1, \ldots, \log_b n - 1$) mamy a^i węzłów, z których każdy wykonuje pracę $f(n/b^i)$. Ostatni poziom zawiera $a^{\log_b n} = n^{\log_b a}$ liści kosztujących $\Theta(1)$. Zatem:

$$T(n) = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + \Theta(n^{\log_b a}).$$

1. W tym przypadku koszt na poziomie i to

$$a^i f(n/b^i) = a^i \mathcal{O}\left(n^{\log_b a - \varepsilon} \cdot b^{-i \log_b a + i\varepsilon}\right) = \mathcal{O}\left(n^{\log_b a - \varepsilon} \cdot b^{i\varepsilon}\right)$$

dla pewnego $\varepsilon > 0$. Mamy $\log_b n$ poziomów, więc złożoność będzie dominowana przez ilość wywołań rekurencyjnych, a dokładniej przez ilość liści, która powstanie. W każdym z nich wykonamy pracę $\mathcal{O}(1)$. Każdy poziom rekurencji zwiększa ilość wywołań a razy, zmniejszając n b-krotnie. Poziomów rekursji będzie zatem $\log_b n$, a samych liści $a^{\log_b n}$. Mamy

$$a^{\log_b n} = a^{\log_a n/\log_a b} = \left(a^{\log_a n}\right)^{1/\log_a b} = n^{\log_b a}.$$

2. Mamy, że $f(n) = \Theta(n^d \lg^k n)$. Policzmy sumaryczną złożoność na i-tym poziomie rekursji:

$$a^i f\left(\frac{n}{b^i}\right) = \mathcal{O}\left(a^i \left(\frac{n}{b^i}\right)^d \lg^k \left(\frac{n}{b^i}\right)\right) = \mathcal{O}\left(n^d \lg^k \left(\frac{n}{b^i}\right)\right).$$

Dla poziomów bliskich korzenia $\lg(n/b^i) \approx \lg n$. Zatem na każdym z $\log_b n$ poziomów wykonywana jest praca rzędu $\Theta(n^d \lg^k n)$. Poziomów jest $\lg n$. Zatem sumaryczna złożoność wyniesie $\lg n \cdot f(n) =$ $\boxed{\Theta(n^d \lg^{k+1} n)}$

3. W tym przypadku wywołania będą na tyle „rzadkie”, że złożoność będzie zdominowana przez $f(n)$. Na poziomie i koszt wynosi $a^i f(n/b^i) \leqslant ca^{i-1} f(n/b^{i-1}) \leqslant \ldots \leqslant c^i f(n)$. Sumując koszty poziomów wewnętrznych, otrzymujemy

$$\sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \leqslant \sum_{i=0}^{\log_b n-1} c^i f(n) = f(n) \sum_{i=0}^{\log_b n-1} c^i.$$

Ponieważ $c < 1$, nieskończony szereg geometryczny jest zbieżny do stałej $\frac{1}{1-c}$. Zatem

$$T(n) \leqslant f(n) \cdot \frac{1}{1-c} + \Theta(n^d) = \mathcal{O}(f(n)).$$

Ale wykonujemy co najmniej jedno wyliczenie $f(n)$, zatem złożoność to sumarycznie $\boxed{\Theta(f(n))}$.

□

II.1.2 Problem sekretarki

Problem sekretarki i jego analiza metodą funkcji tworzących (szkic – opis kolejnych kroków dowodu).

Definicja (Problem sekretarki)**.** Mamy n kolejnych kandydatów na sekretarkę, parami różnych jeśli chodzi o kompetencje. Przeglądamy ich po kolei. Jeśli w danym momencie nie mamy sekretarki albo obecny kandydat jest lepszy od obecnej to zwalniamy poprzedniego i zatrudniamy nowego. Ile wykonamy operacji „zatrudnij”?

Analiza probabilistyczna

Możemy spróbować odpowiedzieć na pytanie, ile *średnio* wykonamy takich operacji. Bez straty ogólności można zmapować kompetencje tak, by wszystkie były liczbami z zakresu od 1 do n. Oznaczmy je przez $a_1, \ldots, a_n$. Niech X będzie liczbą zatrudnionych osób, poszukujemy $\mathbb{E}[X]$. Niech X_i oznacza indykator, czy i-ty kandydat został zatrudniony. Oczywiście mamy $X = \sum X_i$, a zatem też $\mathbb{E}[X] = \sum \mathbb{E}[X_i] = \sum \mathbb{P}(X_i)$. Zauważmy, że sytuacja, w której zatrudniamy nową sekretarkę oznacza, że $a_i = \max(a_1, \ldots, a_i)$. Zakładając, że wszystkie permutacje występują z rozkładem jednostajnym szansa tego wynosi $1/i$ (bo największy element musi zostać umieszczony jako ostatni). Otrzymujemy zatem

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{P}(X_i) = \sum_{i=1}^{n} 1/i = H_n = \ln n + \mathcal{O}(1).$$

Analiza funkcjami tworzącymi

Niech X_n oznacza liczbę ponownych zatrudnień jeśli mamy n kandydatów. Niech $p_{n,k} = \mathbb{P}(X_n = k)$. Jak wcześniej wspomnieliśmy n-ty kandydat ma szansę $1/n$ zostania wybranym, z czego dostajemy

$$p_{n,k} = \frac{1}{n} p_{n-1,k-1} + \frac{n-1}{n} p_{n-1,k} \qquad \text{dla } n \geqslant 2, k \geqslant 1,$$

$$p_{n,0} = \frac{n-1}{n} p_{n-1,0} \qquad \text{dla } n \geqslant 2,$$

$$p_{1,k} = \begin{cases} 1, & k = 0 \\ 0, & k > 0 \end{cases}.$$

Niech $G_n(x)$ będzie funkcją tworzącą dla $p_{n,k}$, czyli $G_n(x) = \sum_{k=0}^{\infty} p_{n,k} x^k$. Zauważmy, że jest to wielomian, bo $p_{n,k} = 0$ dla $k \geqslant n$. Dla $n \geqslant 2$ mamy

$$\begin{aligned} G_n(x) = \sum_{k=0}^{\infty} p_{n,k} x^k &= \frac{1}{n} \sum_{k=1}^{\infty} p_{n-1,k-1} x^k + \frac{n-1}{n} \sum_{k=0}^{\infty} p_{n-1,k} x^k \\ &= \frac{x}{n} \sum_{k=0}^{\infty} p_{n-1,k} x^k + \frac{n-1}{n} \sum_{k=0}^{\infty} p_{n-1,k} x^k \\ &= \frac{x}{n} G_{n-1}(x) + \frac{n-1}{n} G_{n-1}(x) \\ &= \frac{n+x-1}{n} G_{n-1}(x). \end{aligned}$$

Chcemy policzyć $\mathbb{E}[X_n]$. Zauważmy, że $G_n'(x) = \sum_{k=1}^{\infty} k p_{nk} x^{k-1}$, więc $G_n'(1) = \mathbb{E}[X]$. Policzmy $G_n'(x)$ korzystając z pochodnej iloczynu

$$G_n'(x) = \left(\frac{n+x-1}{n} G_{n-1}(x) \right)' = \frac{1}{n} G_{n-1}(x) + \frac{n+x-1}{n} G_{n-1}'(x).$$

Podstawmy $x = 1$. Wtedy $G_n(1) = \sum_{k=0}^{\infty} p_{n,k} = 1$, bo suma prawdopodobieństw wynosi 1. Otrzymujemy

$$G_n'(1) = \frac{1}{n} + G_{n-1}'(1).$$

Z tego oraz z faktu, że $G_1'(1) = 0$ (bo $G_1(x) = 1$) prostą indukcją pokazujemy, że

$$G_n'(1) = \sum_{i=2}^{n} 1/i = \boxed{H_n - 1}.$$

II.1.3 Drzewa rozchylane

Drzewa rozchylane (splay), analiza amortyzowana (szkic).

Drzewem rozchylanym nazywamy drzewo BST z operacją rozchylenia (czyli właśnie „splay"). Operacja splay(x, S) znajduje w drzewie S ostatni węzeł v na ścieżce wyszukującej x (czyli x lub liść, do którego x zostałby dowiązany przy wstawianiu), a następnie wykonuje rotacje AVL tego węzła w górę tak, że ostatecznie powstaje drzewo S' o korzeniu v.

Jeśli wykonujemy splay(x, S), y jest rodzicem x, a z rodzicem y, to mamy kilka opcji:

- x jest lewym (prawym) dzieckiem y i y jest lewym (prawym) dzieckiem z. W takiej sytuacji wykonujemy operację „zig-zig", czyli dwie rotacje w prawo (lewo) – pierwsza względem z, a druga względem y (najpierw podnosimy y, a potem x – dla złożoności będzie ważne, że robimy to w tej kolejności).

- x jest prawym (lewym) dzieckiem y i y jest lewym (prawym) dzieckiem z. W takiej sytuacji wykonujemy operację „zig-zag", czyli rotujemy z w górę, najpierw w lewo (prawo) względem y a potem w prawo (lewo) względem z. Inaczej mówiąc, wykonujemy podwójną rotację względem z.
- x jest lewym (prawym) dzieckiem y i y jest korzeniem. W takiej sytuacji wykonujemy operację „zig", czyli jedną rotację w prawo (lewo) względem y.

Zauważmy, że jeśli y jest korzeniem splay(x, S), to w zbiorze S nie ma żadnego klucza pomiędzy x a y (w posortowanym ciągu kluczy). Za pomocą operacji splay można łatwo zaimplementować operacje słownikowe.

- Wyszukiwanie x: robimy splay(x, S) i patrzymy do korzenia.
- Wstawianie x: robimy splay(x, S), dostając drzewo o korzeniu y. Tworzymy nowy korzeń x. Jeśli $y < x$, to x przejmuje prawe poddrzewo y i dowiązuje y jako lewe dziecko. Jeśli $y > x$, to x przejmuje lewe poddrzewo y i dowiązuje y jako prawe dziecko.
- Usuwanie x: robimy splay(x, S), dostając drzewo o korzeniu x. Usuwamy ten korzeń i mamy teraz dwa drzewa S_1 i S_2. Wykonujemy splay(x, S_1), czyli wyciągamy jako korzeń największy element z S_1. Ma on puste prawe poddrzewo, więc możemy tam dowiązać S_2.

Okazuje się, że amortyzowana złożoność pojedynczej operacji na drzewie splay jest logarytmiczna. Niech $w_S(x)$ oznacza rozmiar poddrzewa wierzchołka x w S. Niech $r_S(x) = \lfloor \lg w_S(x) \rfloor$. Tę wartość nazywamy rangą wierzchołka. Definiujemy funkcję potencjału $\Phi(S) = \sum_{x \in S} r_S(x)$. Przez $\widehat{c}(\cdot)$ będziemy oznaczać amortyzowany koszt operacji, a przez $c(\cdot)$ koszt rzeczywisty.

Lemat. Niech $x \in S$, niech t będzie korzeniem S, $S' = \text{splay}(S, x)$. Wtedy $\widehat{c}(\text{splay}(S, x)) \leqslant 3(r_S(t) - r_S(x)) + 1$.

Dowód. Jeśli $x = t$, to teza zachodzi. Dalej zakładamy, że następuje ciąg $k > 0$ rotacji, które tworzą kolejne drzewa $S_0, \ldots, S_k$, gdzie $S_0 = S$ i $S_k = S'$. Niech x, y_i, z_i będą argumentami i-tej rotacji, czyli y_i jest rodzicem x w S_i, a z_i rodzicem y_i. Oznaczmy koszt amortyzowany i-tej operacji przez $\widehat{c}_i = 1 + \Phi(S_i) - \Phi(S_{i-1})$. Zachodzi $\widehat{c}(\text{splay}()) = c(\text{splay}()) + \Phi(S') - \Phi(S) = \sum_{i=1}^{k} \widehat{c}_i$. Wystarczy więc pokazać, że $\widehat{c}_i \leqslant 3\left(r_{S_i}(x) - r_{S_{i-1}}(x)\right)$ dla $i = 1, \ldots, k-1$ i $\widehat{c}_k \leqslant 1 + 3\left(r_{S_k}(x) - r_{S_{k-1}}(x)\right)$.

Będziemy rozważać rotacje w prawo, rotacje w lewo są symetryczne. Dla ustalonego i będziemy pisać $y = y_i$, $z = z_i$ oraz $r_v = r_{S_{i-1}}(v)$, $r'_v = r_{S_i}(v)$ dla $v \in \{x, y, z\}$ Rozważamy przypadki.

1. Operacja „zig-zig". Mamy $r'_x = r_z$, więc $\widehat{c}_i = 1 + \Phi(S_i) - \Phi(S_{i-1}) = 1 + r'_x + r'_y + r'_z - r_x - r_y - r_z = 1 + r'_y + r'_z - r_x - r_y$.
   - Zakładamy $r'_x > r_x$. Wtedy z $r'_y, r'_z \leqslant r'_x$ oraz $r_y \geqslant r_x$ mamy $\widehat{c}_i \leqslant 1 + 2r'_x - 2r_x = 1 + 2(r'_x - r_x) \leqslant 3(r'_x - r_x)$.
   - Zakładamy $r'_x = r_x$. Wtedy $r'_y \leqslant r'_x = r_x \leqslant r_y$ oraz $r'_z \leqslant r'_x = r_x$. Załóżmy nie wprost, że $r'_z = r_x = a$. Poddrzewa x w S_{i-1} i z w S_i mają co najmniej 2ª węzłów i są rozłączne. Zatem poddrzewo x w S_i ma co najmniej $2 \cdot 2^a + 1$ elementów (bo jeszcze y). Zatem $r'_x \geqslant a + 1 > r_x$ – sprzeczność. Mamy więc $r'_z - r_x \leqslant -1$. Wobec tego $\widehat{c}_i = 1 + \left(r'_y - r_y\right) + (r'_z - r_x) \leqslant 1 + 0 - 1 = 0 = 3(r'_x - r_x)$.
2. Operacja „zig-zag". Podobnie jak wcześniej $r'_x = r_z$ i $\widehat{c}_i = 1 + r'_y + r'_z - r_x - r_y$.
   - Zakładamy $r'_x > r_x$. Tu dzieje się dokładnie to samo, co w analogicznym przypadku operacji „zig-zig".
   - Zakładamy $r'_x = r_x$. Podobnie jak przedtem mamy $r'_y \leqslant r_y$ i $r'_z \leqslant r_x$. Do tego mamy $r_y = r_x$ (bo $r_y \leqslant r'_x$ i $r_x \leqslant r_y$). Chcemy pokazać, że $\left(r'_y - r_y\right) + (r'_z - r_x) \leqslant -1$. Załóżmy nie wprost, że $r'_y - r_y = 0$ i $r'_z - r_x = 0$. Wtedy $r'_y = r_y = r_x = r'_z = a$. Poddrzewo x w S_i zawiera w sobie poddrzewa y i z, więc $r'_x \geqslant a + 1 > r_x$ – sprzeczność. Zatem mamy $\widehat{c}_i \leqslant 1 + \left(r'_y - r_y\right) + (r'_z - r_x) \leqslant 0 = 3(r'_x - r_x)$.
3. Operacja „zig". Może się wydarzyć tylko przy ostatniej rotacji, gdy z_k nie istnieje. Wtedy $r'_x = r_y$ i $\widehat{c}_k = 1 + r'_x + r'_y - r_x - r_y = 1 + r'_x - r_x$. Mamy $r'_x - r_x \geqslant 0$, więc można to przeszacować przez $1 + 3(r'_x - r_x)$.

□

Twierdzenie. Koszt ciągu m operacji na początkowo pustym drzewie splay S wynosi $\mathcal{O}(m \lg n)$, gdzie n jest maksymalnym rozmiarem S podczas wykonywanych operacji.

Dowód. Wiemy, że operacja splay kosztuje co najwyżej $3 \lg n + 1$. Wyszukiwanie i wstawianie to jeden splay i kilka operacji w czasie stałym. Przy wstawianiu dodatkowo rośnie potencjał, ale o co najwyżej $\lg n$ (ranga nowego korzenia). Przy usuwaniu wykonujemy dwa razy splay i kilka operacji w czasie stałym. Ostatecznie mamy $\mathcal{O}(\lg n)$ na każdą operację i początkowy potencjał 0. □

Jeśli początkowo S ma w sobie jakieś elementy, to mamy złożoność $\mathcal{O}((m+n) \lg n)$, bo musimy jeszcze uwzględnić początkowy potencjał, który mógł być zużyty.

Złożoność ciągu operacji na drzewach splay jest taka sama, jak dla drzew AVL lub kopcodrzew. Ich zaletą jest fakt, że nie musimy trzymać żadnych dodatkowych informacji w drzewie (np. balans, priorytety). Do tego dobrze nadają się do struktur implementujących jakąś formę pamięci cache, bo elementy na których operowaliśmy niedawno są wysoko w drzewie. Minusem jest fakt, że złożoność pojedynczej operacji może być liniowa.

II.1.4 Złożoność sortowania

Dolne ograniczenia na złożoność sortowania w przypadku pesymistycznym i średnim.

Znamy algorytmy sortujące w czasie $\mathcal{O}(n \log n)$. Czy da się szybciej? Będziemy pracować w modelu drzew decyzyjnych: jedyne możliwe operacje to porównywanie elementów, które wykonuje za nas czarna skrzynka w czasie $\mathcal{O}(1)$. Większość sensownych algorytmów sortujących wpisuje się w ten model.

Definicja. Drzewo decyzyjne sortujące to algorytm zapisany jako drzewo binarne, które spełnia:

- każdy węzeł wewnętrzny zawiera porównanie $a[i] \leqslant a[j]$ dla pewnych elementów sortowanego ciągu.
- obliczenie rozpoczyna się w korzeniu.
- jeśli wynik porównania jest „tak” to sterowanie przechodzi do lewego poddrzewa, a jeśli nie, to do prawego.
- na końcu każdej ścieżki-obliczenia jest liść zawierający wynik – permutacja elementów dająca kolejność rosnącą.

Pesymistyczna złożoność sortowania

Propozycja. Pesymistyczna złożoność działania algorytmu to wysokość jego drzewa decyzyjnego.

Propozycja. Wysokość drzewa binarnego o m liściach wynosi co najmniej $\log m$.

Dowód. Pełne drzewo binarne wysokości h ma co najwyżej 2^h liści. □

Propozycja. Każde drzewo sortujące n elementów ma co najmniej $n!$ liści.

Dowód. Bo tyle jest możliwych wyników algorytmu – permutacji elementów. □

Wniosek. Każdy algorytm sortujący metodą porównań działa pesymistycznie w czasie co najmniej $n \log n - 1.45n$.

Dowód. Składając poprzednie propozycje mamy, że pesymistyczna złożoność wynosi co najmniej $\log(n!)$. Z aproksymacji Stirlinga wiemy, że

$$n! \approx \sqrt{2\pi n}(n/e)^n.$$

To oznacza, że

$$\log(n!) \approx \log\left(\sqrt{2\pi n}(n/e)^n\right) = n(\log n - \log e) + O(\log n) \approx n\log n - 1.45n.$$

□

Średnia złożoność sortowania

Propozycja. Zakładając, że wszystkie permutacje elementów na wejściu są jednakowe prawdopodobne, średnia złożoność sortowania jest średnią arytmetyczną głębokości wszystkich liści.

Dalej przez T_m oznaczamy pełne drzewo binarne o m liściach, zaś dla drzewa T przez $D(T)$ oznaczamy sumę głębokości wszystkich liści T.

Lemat. $D(T_m) \geqslant m\log m$.

Dowód. Dowodzimy indukcją po m. Dla $m = 1$ działa. Niech więc $m > 1$. Jako, że drzewo jest pełne, to oba poddrzewa korzenia są niepuste. Lewe poddrzewo ma więc pewne i liści, zaś prawe $m - i$ liści, gdzie $0 < i < m$.

$$D(T_m) = i + D(T_i) + (m-i) + D(T_{m-i}) \geqslant m + i\log i + (m-i)\log(m-i).$$

Funkcja $f(x) = x\log x + (m-x)\log(m-x)$ ma minimum w $x = m/2$. Zatem

$$D(T_m) \geqslant m + 2\cdot m/2\log(m/2) = m + m\log m - m = m\log m.$$

□

Wniosek. Każdy algorytm sortujący metodą porównań działa średnio w czasie co najmniej $n\log n - 1.45n$.

Dowód. Wiemy już, że drzewo sortujące n elementów ma $n!$ liści. Z lematu wynika więc, że suma głębokości liści wynosi co najmniej $n!\log(n!)$. Średnia głębokość to w takim razie

$$n!\log(n!)/n! = \log(n!) \approx n\log n - 1.45n.$$

□

Okazuje się, że istnieją algorytmy, które prawie osiągają to ograniczenie. Korzystają one z tak zwanych drzew turniejowych i wykonują co najwyżej o $n - 1$ porównań więcej, niż pokazane minimum.

II.1.5 Randomizowane BST

Randomizacja drzew poszukiwań binarnych: model permutacyjny, kopcodrzewa, złożoność operacji (idea dowodu w oparciu o złożoność quicksortu).

W modelu permutacyjnym rozważamy losową permutację $a_1, \ldots, a_n$ zbioru $\{1, \ldots, n\}$, której elementy wstawiamy kolejno do drzewa BST. Interesuje nas średnia złożoność operacji wyszukiwania w tak uzyskanym drzewie T. Złożoność rozważamy w dwóch przypadkach: gdy wyszukiwanie się nie powiedzie, i gdy się powiedzie.

W przypadku nieudanego wyszukiwania interesuje nas oczekiwana suma długości ścieżek do liści zewnętrznych (czyli pustych węzłów). Oznaczmy ją przez $G(n)$ dla drzewa o n elementach. Takie drzewo ma dokładnie $n+1$ liści zewnętrznych. Jeśli j jest korzeniem T, to lewe poddrzewo ma $j-1$ elementów, a prawe $n-j$. Zatem zachodzi $G(0) = 0$ i

$$G(n) = n + 1 + \sum_{j=1}^{n} \frac{1}{n}\left(G(j-1) + G(n-j)\right),$$

bo korzeń jest wybieramy jednostajnie. Jest to identyczne równanie jak dla quicksortu – losujemy podział na połowy i wywołujemy się rekurencyjnie. Zatem jest $G(n) = 2n \ln n + \mathcal{O}(n) \approx 1.4n \lg n + \mathcal{O}(n)$. Średnia długość ścieżki do liścia zewnętrznego wynosi $\Theta(\log n)$.

W przypadku wyszukiwania z sukcesem interesuje nas suma długości ścieżek do wszystkich węzłów drzewa (poza liśćmi zewnętrznymi). Indukcją po głębokości drzewa łatwo pokazać, że jest ona równa sumie długości ścieżek do liści zewnętrznych pomniejszonej o $2n$. Zatem ta suma w oczekiwaniu wynosi $G(n) - 2n$ i oczekiwana długość ścieżki wyszukiwania to $\Theta(\log n)$.

Niech T będzie drzewem BST otrzymanym w wyniku dodania do niego kolejnych elementów losowej permutacji $\{1, \ldots, n\}$. Interesuje nas oczekiwana głębokość tego drzewa, czyli $\mathbb{E}[h(T)]$. Niech X_n będzie zmienną losową oznaczającą wysokość T. Niech $Y_n = 2^{X_n}$. Niech R_n będzie zmienną losową oznaczającą korzeń T. Niech $Z_{n,i}$ będzie indykatorem tego, że i jest korzeniem.

Jeśli $R_n = i$, to lewe i prawe poddrzewa są losowe i maja odpowiednio $i-1$ i $n-i$ elementów. Zatem $Y_n = 2\max(Y_{i-1}, Y_{n-i})$ przy $Y_1 = 1$ i $Y_0 = 0$. Przy ustalonym T tylko jedna zmienna $Z_{n,i}$ przyjmuje wartość 1, więc $Y_n = \sum_{i=1}^{n} Z_{n,i} \cdot 2\max(Y_{i-1}, Y_{n-i})$. $Z_{n,i}$ jest niezależne od Y_{i-1} i Y_{n-i}, bo $Z_{n,i}$ zależy tylko od położenia i w permutacji, a dwie pozostałe zmienne zależą od permutacji elementów innych niż i. Mamy

$$\mathbb{E}[Y_n] = \sum_{i=1}^{n} \mathbb{E}[Z_{n,i} \cdot 2\max(Y_{i-1}, Y_{n-i})] = \frac{2}{n}\sum_{i=1}^{n} \mathbb{E}[\max(Y_{i-1}, Y_{n-i})] \leqslant \frac{2}{n}\sum_{i=1}^{n} \mathbb{E}[Y_{i-1} + Y_{n-i}]$$
$$= \frac{4}{n}\sum_{i=0}^{n-1} \mathbb{E}[Y_i].$$

Propozycja. Zachodzi $\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$.

Dowód. $\sum_{i=0}^{n-1} \binom{i+3}{3} = \sum_{i=4}^{n+3} \binom{i-1}{3}$, a to zlicza wybory 4 liczb spośród $1, \ldots, n+3$ sumując się po największej wybranej liczbie. □

Pokażemy indukcyjnie, że $\mathbb{E}[Y_n] \leqslant \frac{1}{4}\binom{n+3}{3}$. Dla $n = 0, 1$ teza działa. Dla $n > 1$ mamy

$$\mathbb{E}[Y_n] \leqslant \frac{4}{n}\sum_{i=0}^{n-1} \mathbb{E}[Y_i] \leqslant \frac{4}{n}\sum_{i=0}^{n-1} \frac{1}{4}\binom{i+3}{3} = \frac{1}{4} \cdot \frac{4}{n}\binom{n+3}{4} = \frac{1}{4}\binom{n+3}{3}.$$

Korzystając z nierówności Jensena i wypukłości 2^x mamy $2^{\mathbb{E}[X_n]} \leqslant \mathbb{E}\left[2^{X_n}\right] = \mathbb{E}[Y_n] \leqslant \frac{1}{4}\binom{n+3}{3}$. To daje nam $\mathbb{E}[X_n] \leqslant 3\lg n + \mathcal{O}(1)$.

Zatem wysokość drzewa BST powstałego w wyniku wstawienia do niego losowej permutacji elementów jest logarytmiczna. Okazuje się, że jeśli to takich operacjach wstawiania wykonamy operacje usunięcia losowych elementów, to drzewo dalej ma głębokość logarytmiczną, ale jeśli potem dokonamy kolejnych losowych wstawień, to oczekiwana głębokość rośnie. Eksperymentalnie wykazano, że średnia długość ścieżki to wtedy $\Theta(\sqrt{n})$.

Aby zachować dobre własności wynikające z losowego rozmieszczenia kluczy w drzewie BST stosujemy tak zwane kopcodrzewa (treap). Kopcodrzewem nazywamy drzewo binarne, które w węzłach trzyma dwa parametry – klucz i priorytet. Kopcodrzewo jest drzewem BST ze względu na klucz i max-kopcem ze względu na priorytet (czyli w każdym poddrzewie maksimum jest w korzeniu). Dla zadanego zbioru kluczy z priorytetami istnieje dokładnie jedno kopcodrzewo zawierające te klucze – element o największym priorytecie jest korzeniem, na lewo są te z mniejszym kluczem, na prawo z większym i kontynuujemy ten argument w poddrzewach. Z tego wynika, że jeśli zaimplementujemy słownik za pomocą kopcodrzewa, to postać kopcodrzewa po ciągu operacji nie zależy od kolejności tych operacji.

Słownik implementujemy za pomocą kopcodrzewa następująco.

- Wyszukiwanie: jak w BST.
- Wstawianie: jak w BST, wstawiamy nowy element x jako liść, jeśli warunek kopca jest zaburzony, to dokonujemy rotacji AVL x w górę aż będzie dobrze.

- Usuwanie: ustawiamy priorytet usuwanego elementu na 0 i rotujemy w dół (w prawo lub lewo, zależnie od tego, który następnik ma większy priorytet) aż do liścia, a następnie odcinamy.

Wszystkie te operacje są liniowe od wysokości kopcodrzewa.

Łatwo zauważyć, że jeśli posortujemy elementy kopcodrzewa w kolejności malejących priorytetów, to drzewo BST powstałe przez wstawienie ich w takiej kolejności jest dokładnie tym kopcodrzewem. Jeśli będziemy losować priorytety jednostajnie, każda permutacja elementów będzie tak samo prawdopodobna. Zatem kopcodrzewo będzie miało strukturę losowego drzewa BST, a o takim wiemy, że ma oczekiwaną wysokość logarytmiczną. Zatem wszystkie operacje na kopcodrzewie mają oczekiwany czas logarytmiczny od liczby elementów w nim.

II.1.6 Haszowanie doskonałe

Uniwersalne rodziny funkcji haszujących i ich zastosowanie w haszowaniu doskonałym.

Rozważamy problem wstawienia n kluczy należących do uniwersum U do m-elementowej tablicy T. Żeby to miało sens zakładamy $n \leqslant m$.

Definicja. Rodzina $\mathcal{H}$ funkcji haszujących jest uniwersalna, gdy

$$\forall_{x,y\in U}\, x \neq y \implies |\{h \in \mathcal{H} : h(x) = h(y)\}| \leqslant \frac{|\mathcal{H}|}{m}.$$

Intuicyjnie chodzi o to, żeby wybranie losowej funkcji haszującej z $\mathcal{H}$ dawało nie większe prawdopodobieństwo kolizji na elementach x, y niż niezależne losowanie dla nich adresów w T, czyli $1/m$.

Lemat. Jeśli funkcję haszującą h wybrano z uniwersalnej rodziny $\mathcal{H}$ i użyto do wstawienia n kluczy do m-elementowej tablicy T, gdzie $n \leqslant m$, to oczekiwana liczba kolizji z danym $x \in U$ wynosi mniej niż 1.

Dowód. Losujemy $h \in \mathcal{H}$. Dla $y, z \in U$ niech $I_{y,z}$ będzie zmienną indykatorową zdarzenia $h(y) = h(z)$. Z uniwersalności $\mathcal{H}$ mamy

$$\mathbb{E}[I_{y,z}] = \mathbb{P}(I_{y,z} = 1) \leqslant \frac{1}{m}.$$

Niech C_x będzie liczbą elementów kolidujących z x. $C_x = \sum_{y\neq x} I_{x,y}$. W takim razie

$$\mathbb{E}[C_x] = \mathbb{E}\left[\sum_{y\neq x} I_{x,y}\right] \leqslant \frac{n-1}{m} < 1.$$

□

Przykład. Załóżmy, że $U \subseteq \mathbb{Z}_p$ dla pewnej liczby pierwszej p. Niech

$$\mathcal{H}_{p,m} = \left\{h_{a,b} : \mathbb{Z}_p \ni x \mapsto ((ax + b) \bmod p) \bmod m \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\right\}.$$

Twierdzimy, że $\mathcal{H}_{p,m}$ jest uniwersalną rodziną funkcji haszujących.

Dowód. Ustalamy $x, y \in \mathbb{Z}_p$ takie, że $x \neq y$. Losujemy $h_{a,b} \in \mathcal{H}_{p,m}$. Niech $r = (ax + b) \bmod p$, $s = (ay + b) \bmod p$. Oczywiście $r \neq s$. Pokażemy, że dla ustalonych x, y każda para $(r, s) \in \mathbb{Z}_p \times \mathbb{Z}_p$ jest jednakowo prawdopodobna. W tym celu zauważmy, że $f(a, b) = ((ax + b) \bmod p, (ay + b) \bmod p)$ jest bijekcją pomiędzy $S_1 = \mathbb{Z}_p^* \times \mathbb{Z}_p$ i $S_2 = \{(r, s) \in \mathbb{Z}_p \times \mathbb{Z}_p : r \neq s\}$. Jest tak, bo $|S_1| = |S_2| = p(p - 1)$ oraz z (r, s) można jednoznacznie odtworzyć (a, b). W takim razie

$$\mathbb{P}(h_{a,b}(x) = h_{a,b}(y)) = \mathbb{P}(r = s \bmod m).$$

gdzie r, s są losowane jednostajnie z S_2. Szacujemy to drugie prawdopodobieństwo. Dla ustalonego r, niech $S_3 = \{s \in \mathbb{Z}_p : s \neq r, s = r \bmod m\}$. Widzimy, że $|S_3| \leqslant \frac{p-1}{m}$. Z tego wynika, że

$$\mathbb{P}(r = s \bmod m) = \frac{|S_3|}{p-1} \leqslant \frac{1}{m}.$$

□

Haszowanie doskonałe

Ustalamy zbiór kluczy $S \subseteq U$, $|S| = n$. Szukamy $h : U \to \{0, 1, \dots, m-1\}$ takiego, że $m = \mathcal{O}(n)$ oraz h jest obliczalna w czasie $\mathcal{O}(1)$. Możemy założyć, że $U = \mathbb{Z}_p$ dla pewnego pierwszego p.

Ideą rozwiązania będzie dwupoziomowe haszowanie uniwersalne:

1. $m := n$, rozrzucamy S w $A[0..n-1]$ z możliwymi kolizjami za pomocą $h : U \to \{0, \dots, n-1\}$.
2. $S_j := \{x \in S : h(x) = j\}$, $n_j := |S_j|$, elementy S_j haszujemy do tablicy $A_j[0..m_j-1]$, gdzie $m_j = n_j^2$. Wynikową tablicą będzie $\bigcup A_j$.

Przechodząc do szczegółów: na poziomie 1 losujemy $h_{a,b} \in \mathcal{H}_{p,m}$. Ustalamy

$$h(x) := h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

Propozycja. $\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$.

Dowód.

$$\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] = \mathbb{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] = \mathbb{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\mathbb{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] = n + 2\mathbb{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right].$$

Ta wartość oczekiwana, która nam została, to łączna liczba wszystkich kolizji. Z lematu o haszowaniu uniwersalnym

$$\mathbb{E}[\text{liczba kolizji}] \leqslant \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}.$$

Ostatecznie dostajemy

$$\mathbb{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \leqslant n + 2\frac{n-1}{2} < 2n.$$

□

Oznaczamy $M = \sum_{j=0}^{m-1} n_j^2$. Z nierówności Markowa mamy

$$\mathbb{P}(M > 4n) \leqslant \frac{\mathbb{E}[M]}{4n} < 1/2.$$

W takim razie po co najwyżej kilku losowaniach dostaniemy $h_{a,b}$ takie, że $M \leqslant 4n$.

Na poziomie drugim dla każdego S_j losujemy $h^j \in \mathcal{H}_{p,m_j}$. Niech

$$C_j = \left\{\{x, y\} \subseteq S : x \neq y \wedge h^j(x) = h^j(y)\right\}.$$

Propozycja. $\mathbb{P}(C_j \neq \varnothing) < 1/2$.

Dowód. Z lematu o haszowaniu uniwersalnym

$$\mathbb{P}(h^j(x) = h^j(y)) \leqslant \frac{1}{m_j} = \frac{1}{n_j^2}.$$

W takim razie

$$\mathbb{E}[|C_j|] \leqslant \binom{n_j}{2} \frac{1}{n_j^2} < 1/2.$$

Z nierówności Markowa

$$\mathbb{P}(C_j \neq \varnothing) = \mathbb{P}(|C_j| \geqslant 1) \leqslant \frac{\mathbb{E}[|C|]}{1} < 1/2.$$

□

Po kilku losowaniach otrzymamy więc funkcję, która nie spowoduje żadnych kolizji.

II.1.7 Problem DSU

Problem sumowania zbiorów rozłącznych, rozwiązanie drzewowe z kompresją ścieżek, szkic analizy i wykorzystanie w niej logarytmu iterowanego.

Mamy uniwersum U o n elementach. Rozważamy pewną rodzinę zbiorów rozłącznych $F \subseteq 2^U$ taką, że $\bigcup F = U$. Zbiory należące do F posiadają swoje identyfikatory. Chcemy efektywnie wykonywać dowolne ciągi operacji:

- MakeSet(x) – utwórz zbiór $\{x\}$, zwykle wykonywane na początku dla wszystkich elementów U.
- Union(S_i, S_j) – połącz zbiory o identyfikatorach S_i, S_j, to znaczy usuń je z F i dodaj ich sumę.
- Find(x) – podaj identyfikator zbioru, do którego należy element x.

Brutalne rozwiązanie polegałoby na trzymaniu tablicy ID, która każdemu elementowi U przyporządkowuje identyfikator zbioru, do którego obecnie należy. Operacje Find działają wtedy w czasie $\mathcal{O}(1)$, ale koszt pojedynczej operacji Union to $\Theta(n)$.

Dalej zakładamy, że w ciągu jest $m = \Omega(n)$ operacji Find oraz co najwyżej $n-1$ operacji Union. Koszt inicjalizacji pomijamy – będzie wynosił $\mathcal{O}(n)$.

Rozwiązanie drzewowe

Każdy zbiór reprezentujemy za pomocą drzewa. Reprezentantem zbioru jest element w jego korzeniu. Każdy element trzyma link do swojego rodzica w drzewie. Operacja Union polega teraz na podpięciu korzenia jednego drzewa do drugiego, zaś operacja Find(x) na przebyciu ścieżki w górę drzewa od wierzchołka x do korzenia.

W pesymistycznym przypadku operacje Union tworzą najpierw n-wierzchołkową ścieżkę, a następnie Find pyta m razy o liść. To daje pesymistyczną złożoność $\Theta(nm)$. Będziemy chcieli to jakoś poprawić.

Łączenie według rang

W każdym wierzchołku trzymamy dodatkowo rank(x) – początkowo ustawione na 0. Operacja Union będzie zawsze podpinała korzeń o mniejszej wartości rank do tego o większej. W przypadku, gdy są równe łączymy dowolnie, ale nowemu korzeniowi zwiększamy rank o 1.

Indukcją po kolejnych operacjach Union możemy pokazać następujące 2 fakty:

Propozycja. Rank korzenia jest zawsze równy głębokości jego drzewa.

Propozycja. Drzewo (a nawet poddrzewo), którego korzeń ma rangę r posiada co najmniej 2^r wierzchołków.

Wniosek. Wszystkie drzewa będą miały zawsze wysokość co najwyżej $\mathcal{O}(\log n)$.

To pozwala zbić pesymistyczną złożoność operacji Find do $\mathcal{O}(\log n)$. Całkowita złożoność wykonania całego ciągu operacji wynosi więc co najwyżej $\mathcal{O}(m \log n + n)$.

Kompresja ścieżek

W celu dalszej optymalizacji zauważmy, że przy każdej operacji Find możemy przestawić wszystkim odwiedzonym na ścieżce do korzenia wierzchołkom ich wskaźnik do rodzica od razu na korzeń. Pozwoli to w przyszłości skrócić kolejne przejścia w górę drzewa. Zauważmy, że to sprawia, że pierwsza z propozycji dotyczących łączenia według rang już nie zachodzi, ale druga nadal tak.

W celu analizy złożoności po tej poprawce potrzebujemy zdefiniować logarytm iterowany.

Definicja. $\log^* n := \min \left\{ k : \log^{(k)} n \leqslant 1 \right\}$, gdzie $\log^{(k)} n$ oznacza k-krotne złożenie funkcji log.

W praktyce $\log^* n \leqslant 5$, bo $\log^* n \geqslant 6$ wymaga $n > 2^{65536}$. Będziemy też potrzebowali funkcji odwrotnej:

$$F(0) = 1,$$
$$F(n) = 2^{F(n-1)}.$$

Twierdzenie (Hopcroft)**.** Pesymistyczny czas działania drzewowej implementacji DSU z łączeniem według rang i kompresją ścieżek na ciągu składającym się z $m \geqslant n$ operacji Find oraz co najwyżej $n-1$ operacji Union wynosi $\mathcal{O}((m+n)\log^* n)$.

Dowód.

1. Jeśli węzeł x jest właściwym potomkiem węzła y, to $\mathrm{rank}(x) < \mathrm{rank}(y)$. Czyli na dowolnej ścieżce w górę drzewa rangi są silnie rosnące. Wynika to wprost z tego jak działa operacja Union oraz kompresja.
2. W dowolnym momencie liczba węzłów o randze r nie przekracza $n/2^r$. Jest tak ponieważ kompresja nie zmienia rang poszczególnych węzłów, a węzeł o randze r ma swoje poddrzewo rozmiaru co najmniej 2^r.
3. Dzielimy węzły na grupy: $G_k = \{x : \log^*(\mathrm{rank}(x)) = k\}$. Wtedy grupa G_k składa się z węzłów, których rangi należą do zbioru $\{F(k-1)+1, \ldots, F(k)\}$.
4. Jeśli $G_k \neq \varnothing$, to $k \leqslant \log^* n - 1$. Jest tak, bo z punktu 2. żadna ranga nie przekracza $\log n$, natomiast
$$\log^*(\log n) = \log^* n - 1.$$
5. $|G_k| \leqslant n/F(k)$, gdyż sumując po wszystkich możliwych rangach w grupie dostajemy
$$\begin{aligned} |G_k| &\leqslant n/2^{F(k-1)+1} + \ldots + n/2^{F(k)} \leqslant n/2^{F(k-1)+1}(1 + 1/2 + 1/4 + \ldots) \\ &= 2 \cdot n/2^{F(k-1)+1} = n/2^{F(k-1)} = n/F(k). \end{aligned}$$
6. Analizujemy całkowity koszt wszystkich operacji Find metodą zespoloną. Podczas przechodzenia ścieżką w górę drzewa księgujemy koszt przeglądu węzła na 2 sposoby:
   - Jeśli x to korzeń lub $\mathrm{up}(x)$ to korzeń lub x należy do innej grupy niż $\mathrm{up}(x)$, to koszt przypisujemy danej operacji Find. W ten sposób przypiszemy jej koszt co najwyżej $\log^* n + 1$. Łącznie przypiszemy więc koszt $\mathcal{O}(m \log^* n)$.
   - W przeciwnym wypadku księgujemy koszt w węźle.
7. W całym ciągu operacji Find łączny koszt przypisany węzłowi x nie przekracza $F(k) - F(k-1)$, gdzie $k = \log^*(\mathrm{rank}(x))$. Będzie tak, bo przy każdym przejściu przez x ranga jego rodzica się zwiększa. To oznacza, ze po co najwyżej tylu odwiedzeniach jego rodzic będzie już musiał być z innej grupy.
8. Dla ustalonego k łączny koszt przypisany wierzchołkom z grupy G_k wynosi
$$|G_k| \cdot (F(k) - F(k-1)) \leqslant |G_k| \cdot F(k) \leqslant n.$$
   Łącznie przypiszemy więc koszt $\mathcal{O}(n \log^* n)$.

□

Okazuje się, że amortyzowany koszt operacji Find w takiej realizacji nie jest stały, ale istnieje jeszcze lepsze ograniczenie górne niż logarytm iterowany. Definiujemy funkcję Ackermanna

$$A(i,j) = \begin{cases} 2^j, & i = 1 \\ A(i-1, 2), & i \geqslant 2, j = 1 \\ A(i-1, A(i, j-1)), & i, j \geqslant 2 \end{cases}.$$

Przez $\alpha(m, n)$ oznaczamy funkcję odwrotną do funkcji Ackermanna. Dla praktycznych wartości m, n jest $\alpha(m, n) \leqslant 4$. Okazuje się, kosz m operacji Find i Union można oszacować przez $\mathcal{O}(m\alpha(m, n))$.

II.2 Algorytmy i Struktury Danych 1

II.2.1 Kody Huffmanna

Kody Huffmana, złożoność algorytmu, własności problemu uzasadniające poprawność metody zachłannej.

Mamy zadany alfabet C wraz z częstotliwością występowania każdego znaku $p : C \to [0, 1]$. Chcemy tak zakodować znaki z C za pomocą 0 i 1, żeby wartość oczekiwana długości zakodowanego ciągu była jak najmniejsza. Aby dekodowanie znaków było łatwe chcemy, aby otrzymany kod był prefiksowy – kod żadnego znaku nie jest prefiksem kodu innego znaku.

Taki kod można reprezentować pełnym drzewem binarnym, w którym liście oznaczają znaki, a ścieżka od korzenia do liścia definiuje kod odpowiedniego znaku (przejście w prawo oznacza 1, w lewo 0). Mając takie drzewo jesteśmy w stanie łatwo odkodować ciąg znaków: zaczynamy w korzeniu, idziemy kolejnymi krawędziami zgodnie z kodowaniem, w momencie dotarcia do liścia poznajemy znak, który właśnie zdekodowaliśmy, więc możemy cofnąć się do korzenia i kontynuować.

Chcemy znaleźć takie drzewo kodowania T, które minimalizuje oczekiwaną długość zakodowanego ciągu znaków. Inaczej mówiąc, chcemy, aby oczekiwana długość kodu $\sum_{a \in C} p(a) |T(a)|$ była minimalna.

Robimy to w następujący sposób:

1. Umieszczamy wszystkie znaki (węzły drzewa będące liśćmi) w kolejce priorytetowej Q (gdzie priorytetem jest prawdopodobieństwo).
2. Dopóki w Q są co najmniej dwa węzły, ściągamy z Q dwa węzły x i y o najmniejszym priorytecie, tworzymy nowy węzeł z o dzieciach x, y i priorytecie będącym sumą priorytetów x i y. Umieszczamy z na Q.

W ten sposób powstaje drzewo T (ukorzenione w ostatnim węźle, który zostanie na Q). Cały algorytm ma złożoność $\mathcal{O}(|C| \log |C|)$ (pojedyncza operacja na kolejce priorytetowej zaimplementowanej za pomocą kopca ma złożoność logarytmiczną).

Niech T' będzie pewnym optymalnym drzewem dla C. Niech $x, y \in C$ będą znakami o najmniejszej częstotliwości. Teraz:

1. W T' istnieją dwa najgłębiej położone liście a, b będące swoim rodzeństwem (idziemy najdłuższą ścieżką z korzenia do węzła, którego dzieci są liśćmi – te liście to a i b). Zamieniając miejscami a, b z x, y w tym drzewie dostajemy drzewo z nie mniejszą oczekiwaną długością kodu. Zatem to drzewo też jest optymalne i możemy założyć, że w T' liście x i y są rodzeństwem.
2. Niech T'' będzie drzewem otrzymanym z T' przez zastąpienie x, y i ich rodzica przez liść z o prawdopodobieństwie będącym sumą prawdopodobieństw x i y. T'' jest optymalnym drzewem dla $C' = C \setminus \{x, y\} \cup \{z\}$, bo gdyby istniało lepsze drzewo $\widetilde{T''}$, to dawałoby ono lepsze od T' drzewo dla C.

Pierwszą z tych własności nazywamy własnością zachłannego wyboru, a drugą własnością optymalnej podstruktury. Za ich pomocą możemy udowodnić, że T jest optymalne. Robimy to indukcją po $|C|$. Baza $|C| = 2$ jest oczywista. Załóżmy $|C| > 2$. Z (1) wiemy, że istnieje optymalne drzewo T', które łączy x z y (tak samo jak T). Z T i T' konstruujemy drzewa $\widetilde{T}$ i $\widetilde{T'}$ dla alfabetu $C \setminus \{x, y\} \cup \{z\}$ jak w (2). Z założenia indukcyjnego $\widetilde{T}$ jest optymalne, a z (2) $\widetilde{T'}$ jest optymalne. Zatem oczekiwane długości kodu dla nich są takie same, czyli oczekiwane długości kodu dla T i T' też są takie same.

Generalnie każdy dowód poprawności algorytmu zachłannego przebiega w taki sposób: podejmujemy jakąś lokalną decyzję i pokazujemy, że istnieje rozwiązanie optymalne, które podejmuje tę samą decyzję (zachłanny wybór), a następnie pokazujemy, że optymalne rozwiązanie otrzymanego w ten sposób podproblemu daje optymalne rozwiązanie początkowego problemu (optymalna podstruktura).

II.2.2 Najdłuższy wspólny podciąg

Znajdowanie najdłuższego wspólnego podciągu metodą programowania dynamicznego i optymalizacja pamięci algorytmem Hirschberga.

Niech $X = (x_1, \ldots, x_n)$ i $Y = (y_1, \ldots, y_m)$ będą dwoma ciągami (nad dowolnym alfabetem). Chcemy znaleźć ich najdłuższy wspólny podciąg (Longest Common Subsequence, LCS) – najdłuższy taki ciąg $Z = (z_0, \ldots, z_k)$, że $z_i = x_{f(i)}$ i $z_i = y_{g(i)}$ dla pewnych ściśle rosnących funkcji f, g.

Zastosujemy algorytm dynamiczny. Niech $X_i = (x_1, \ldots, x_i)$ i $Y_i = (y_1, \ldots, y_i)$. Wyznaczymy $c(i, j)$ będące długością LCSu X_i i Y_j. Zachodzi

$$c(i,j) = \begin{cases} 0, & i = 0 \text{ lub } j = 0 \\ c(i-1, j-1) + 1, & i, j > 0, x_i = y_j \\ \max(c(i, j-1), c(i-1, j)), & i, j > 0, x_i \neq y_j \end{cases}.$$

Niech $Z = (z_1, \ldots, z_k)$ będzie LCSem X_i i Y_i. Jeśli $x_i = y_j$ i $z_k \neq x_i$, to Z można wydłużyć o element $x_i = y_j$, co daje lepsze rozwiązanie – sprzeczność. Zatem $z_k = x_i = y_j$. Z bez ostatniego elementu jest wspólnym podciągiem X_{i-1} i Y_{j-1}, więc jest najdłuższym takim podciągiem. Jeśli $x_i \neq y_j$, to $z_k \neq x_i$ i Z jest LCSem X_{i-1} i Y_j lub $z_k \neq y_j$ i Z jest LCSem X_i i Y_{j-1}. Zatem przedstawiony wzór jest poprawny.

Możemy więc wyznaczyć wartość c dla wszystkich nm możliwych argumentów. Szukana długość LCSu X_n i Y_m to $c(n, m)$. Aby odtworzyć taki podciąg możemy podczas wyznaczania $c(i, j)$ zapamiętać, którą z trzech opcji $c(i-1, j-1)+1, c(i-1, j), c(i, j-1)$ wybraliśmy. Teraz możemy odzyskać rozwiązanie: zaczynamy w stanie (n, m) i cofamy się do stanu $(n-1, m-1)$, $(n-1, m)$ lub $(n, m-1)$ zależnie od tego, którą opcję wybrał algorytm dynamiczny. Jeśli wybrał $(n-1, m-1)$, to dopisujemy odpowiednią literę do rozwiązania. Kontynuujemy w ten sposób aż dojdziemy do $i = 0$ lub $j = 0$.

Przedstawiony algorytm ma złożoność czasową $\Theta(nm)$ i pamięciową $\Theta(nm)$. Jeśli nie potrzebujemy odtwarzać rozwiązania, to możemy zmodyfikować ten algorytm do złożoności pamięciowej $\Theta(\min(n, m))$: zauważmy, że wyliczając $c(i, j)$ potrzebujemy pamiętać tylko $c(i-1, j)$, $c(i, j-1)$ i $c(i-1, j-1)$. Zatem możemy po kolei rozważać wszystkie $k = 0, \ldots, n+m$ i dla ustalonego k wyznaczać rozwiązania dla i, j takich, że $i + j = k$ za pomocą rozwiązań dla i, j takich, że $i + j = k - 1$ lub $i + j = k - 2$. To pozwala nam pamiętać tylko liniowo wiele wartości w dany momencie.

Aby odtworzyć rozwiązanie w czasie $\Theta(nm)$ i pamięci $\Theta(\min(n+m))$ stosujemy algorytm Hirschberga. Zauważmy, że dla każdego LCSa X i Y istnieje takie k, że ten LCS jest konkatenacją pewnego LCSa $\widetilde{X} = \left(x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}\right)$ i $Y_k = (y_1, \ldots, y_k)$ oraz pewnego LCSa $\widetilde{X'} = \left(x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n\right)$ i $Y'_k = (y_{k+1}, \ldots, y_m)$. Możemy zastosować opisany przedtem algorytm i znaleźć długość α_j LSCa $\widetilde{X}$ i $Y_j = (y_1, \ldots, y_j)$ dla każdego $j = 1, \ldots, m$ (stosujemy poprzedni algorytm tylko raz, ale zapamiętujemy wszystkie wartości $c\left(\lfloor \frac{n}{2} \rfloor, j\right)$, co dokłada nam tylko liniową ilość pamięci). Podobnie znajdujemy długość β_j LSCa $\widetilde{X'}$ i $Y'_j = (y_{j+1}, \ldots, y_m)$ dla każdego $j = 1, \ldots, m$. k jest tym indeksem, który maksymalizuje $\alpha_k + \beta_k$. Zatem jesteśmy w stanie znaleźć go w czasie liniowym. Następnie wywołujemy się rekurencyjnie na ciągach $\widetilde{X}$ i Y_k oraz $\widetilde{X'}$ i Y'_k. Bazą algorytmu są krótkie ciągi, dla których możemy normalnie odtworzyć rozwiązanie.

Złożoność czasowa zadana jest równaniem rekurencyjnym $T(n, m) = T\left(\frac{n}{2}, k\right) + T\left(\frac{n}{2}, m-k\right) + nm$. Łatwo pokazać indukcyjnie, że $T(n, m) \leqslant 2nm$. W każdym wywołaniu rekurencyjnym możemy wykonać obliczenia, zwolnić pamięć, a dopiero potem wywołać się rekurencyjnie. Zatem całościowo nie zużyjemy więcej pamięci, niż w jednym wywołaniu rekurencyjnym, co daje odpowiednią złożoność.

II.2.3 Drzewa AVL

Technika rotacji i jej użycie w algorytmach drzew zrównoważonych AVL.

Definicja. Niech U będzie zbiorem elementów, na którym zadany jest pewien liniowy porządek. Drzewem BST (Binary Search Tree) nazywamy drzewo binarne, w którym węzły utożsamiamy z elementami

U i dla dowolnych węzłów x, y jeśli x jest w lewym poddrzewie y, to $x < y$ oraz jeśli x jest w prawym poddrzewie y, to $x > y$.

Definicja. Drzewem AVL (Adelson-Velsky, Landis) nazywamy drzewo BST, w którym dla każdego węzła v wysokość poddrzew o korzeniach left(v) i right(v) różni się o co najwyżej 1. Różnicę $b(v) = h(\text{right}(v)) - h(\text{left}(v))$ nazywamy współczynnikiem zrównoważenia (balansem).

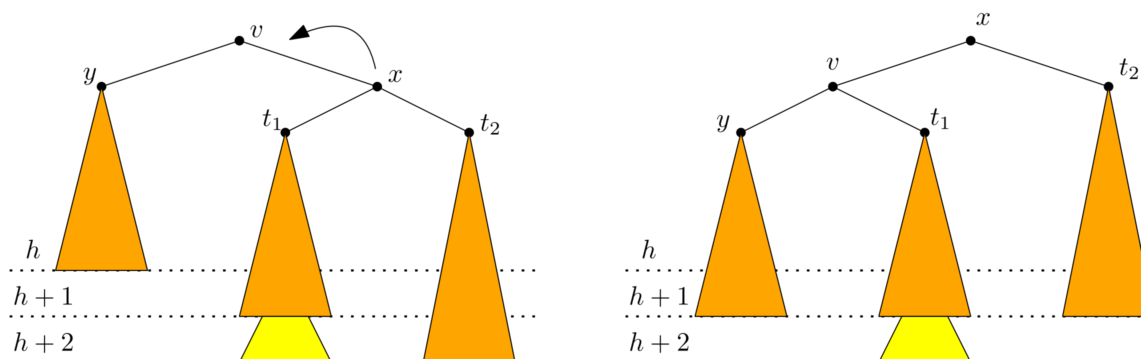
Twierdzenie. Wysokość drzewa AVL o n węzłach nie przekracza $\Theta(\log n)$.

Dowód. Jeśli drzewo binarne ma n węzłów, to ma $n + 1$ liści zewnętrznych (pustych wskaźników w węzłach). Niech $N(h)$ oznacza najmniejszą możliwą liczbę liści zewnętrznych w drzewie AVL o wysokości h. Mamy $N(0) = 2$, $N(1) = 3$ oraz $N(h) = N(h-1) + N(h-2)$, bo minimalne drzewo AVL o wysokości h powstaje przez dołączenie do korzenia dwóch minimalnych drzew o wysokościach $h - 1$ i $h - 2$. Zatem $N(h) = F(h+3)$, gdzie $F(i)$ jest i-tą liczbą Fibonacciego. Ze wzoru Bineta $F(k) \geqslant \left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$ (można też tego dowodzić indukcyjnie). W drzewie AVL o n węzłach i wysokości h mamy $n+1 \geqslant N(h) \geqslant \left(\frac{1+\sqrt{5}}{2}\right)^{h+1}$, czyli $h \leqslant \mathcal{O}(\log n)$. □

Drzewa BFS istnieją po to, żeby implementować za ich pomocą słowniki, czyli struktury danych, w których możemy trzymać elementy U i wykonywać operacje wyszukania, wstawienia i usunięcia. Wyszukiwanie polega na przechodzeniu od korzenia w dół drzewa w kierunku wynikającym z warunku z definicji drzewa BST. Wstawianie wygląda tak samo – wyszukujemy element i jeśli nie istnieje, to umieszczamy nowy węzeł z tym elementem w miejscu, gdzie powinien być. Usuwanie jest trochę trudniejsze – znajdujemy element, jeśli ma co najwyżej jedno dziecko, to usuwamy go i zastępujemy go w strukturze drzewa tym dzieckiem. Jeśli ma dwójkę dzieci, to patrzymy do jego prawego poddrzewa i znajdujemy tam najbardziej lewy węzeł (czyli idziemy raz w prawo a potem tylko w lewo). Ten najbardziej lewy węzeł możemy łatwo usunąć (bo nie ma lewego dziecka) i wstawić go na miejsce usuwanego węzła – warunek BST dalej jest zachowany.

Wszystkie te operacje działają w czasie liniowym od wysokości drzewa, czyli np. dla drzew AVL są szybkie. Mogą one jednak zmieniać wysokość drzewa i naruszać zbalansowanie węzłów. Aby wynikowe drzewo pozostało drzewem AVL, musimy naprawić ewentualne niezbalansowania powstałe w wyniku takich operacji. Zauważmy, że takie niezbalansowania pojawiają się tylko na ścieżce od korzenia do dodawanego (usuwanego) węzła i dla niezbalansowanego węzła v zachodzi $|b(v)| = 2$. Niech v będzie najniższym niezbalansowanym węzłem (niezbalansowania poprawiamy iteracyjnie od dołu). Jeśli x jest dzieckiem v o wyższym poddrzewie a y o niższym, to niezbalansowanie mogło powstać w wyniku dodania węzła do poddrzewa x tak, że jego wysokość wzrosła lub w wyniku usunięcia węzła z poddrzewa y tak, że jego wysokość spadła. Przywracamy balans za pomocą tak zwanych rotacji.

Jeśli x jest prawym dzieckiem v i $b(x) \geqslant 0$, to stosujemy rotację w lewo względem v: v staje się lewym dzieckiem x, a lewe dziecko x zostaje prawym dzieckiem v (patrz Rysunek 6). Jeśli x jest lewym dzieckiem v i $b(x) \leqslant 0$, to stosujemy analogiczną rotację w prawo względem v: v staje się prawym dzieckiem x, a prawe dziecko x zostaje lewym dzieckiem v.



Rysunek 6: Przykład lewej rotacji.

Jeśli x jest prawym dzieckiem v i $b(x) < 0$, to stosujemy prawą rotację względem x a następnie lewą rotację względem v. Jeśli x jest lewym dzieckiem v i $b(x) > 0$, to stosujemy lewą rotację względem x a następnie prawą rotację względem v. Rysunki do tego można zobaczyć na Wikipedii.

II.2.4 Najkrótsze ścieżki

Algorytm Warshalla-Floyda oraz algorytm Johnsona do znajdowania najkrótszych ścieżek dla wszystkich par wierzchołków grafu.

Niech G będzie skierowanym grafem z wagami na krawędziach. Niech n będzie jego liczbą wierzchołków, a m liczbą krawędzi. Naszym celem jest znalezienie najkrótszej (w sensie sumy wag) ścieżki między dwoma wierzchołkami w G (lub między wszystkimi parami wierzchołków). Całkiem zdrowym założeniem jest w takiej sytuacji, że nasz graf nie ma ujemnych cykli – jeśli taki cykl istnieje, to można nim przechodzić dowolnie wiele razy, osiągając dowolnie krótkie ścieżki.

Algorytm **Warshalla-Floyda** wyznacza odległości między każdą parą wierzchołków grafu przy założeniu, że w grafie nie ma ujemnego cyklu. Robi to w czasie $\mathcal{O}\left(n^3\right)$ i stosuje podejście dynamiczne. Niech naszym zbiorem wierzchołków będzie $\{1, \ldots, n\}$. Przez $d^{(k)}(i, j)$ oznaczamy długość najkrótszej ścieżki z i do j, w której jako wierzchołki pośredni występują tylko wierzchołki ze zbioru $\{1, \ldots, k\}$. Mamy $d^{(0)}(i, j) = w(i, j)$ (waga krawędzi lub $+\infty$ w przypadku jej braku). Zachodzi

$$d^{(k)}(i, j) = \min\left(d^{(k-1)}(i, j), d^{(k-1)}(i, k) + d^{(k-1)}(k, j)\right),$$

bo jeśli rozważana najkrótsza ścieżka nie zawiera k, to jej długość to $d^{(k-1)}(i, j)$, a jeśli zawiera k, to nie dwa razy (przejście z k do k musi mieć nieujemną wagę, bo inaczej mamy ujemny cykl), zatem można ją podzielić na część idącą do k i z k. Wyznaczając odpowiednie wartości kolejno dla $k = 0, \ldots, n$ dostajemy wszystkie szukane odległości. Dodatkowo możemy łatwo konstruować odpowiednie ścieżki – gdy wyliczamy $d^{(k)}(i, j)$ możemy stwierdzić, jaka jest ostatnia krawędź na ścieżce świadczącej o tej wartości (zakładając, że mamy odpowiednie krawędzi wyznaczone dla $k - 1$). To ostatecznie pozwala nam odtworzyć wynik.

Algorytm **Bellmana-Forda** wyznacza odległość z ustalonego wierzchołka v do wszystkich innych wierzchołków grafu przy założeniu, że w grafie nie ma ujemnego cyklu (osiągalnego z v). Robi to w czasie $\mathcal{O}(nm)$ i działa w następujący sposób:

```
∀_{u≠v} d(u) ← +∞
d(v) ← 0
for i = 1, ..., n do
    for (r, s) ∈ E(G) do
        if d(r) + w(r, s) < d(s) then
            d(s) ← d(r) + w(r, s)
        end if
    end for
end for
```

Po pierwszym wykonaniu zewnętrznej pętli $d(u)$ będzie długością najkrótszej ścieżki długości 1 od v do u. Po drugim będzie to długość najkrótszej ścieżki długości 2 i tak dalej. Nie ma ujemnych cykli, więc każda najkrótsza ścieżka ma długość co najwyżej n i ostatecznie algorytm zwróci poprawny wynik. Zauważmy, że za pomocą tego algorytmu można wykrywać, czy w grafie istnieje ujemny cykl – możemy wykonać algorytm Bellmana-Forda a następnie wykonać jeszcze jedno przejście zewnętrznej pętli. Jeśli poprawni ono jakiś wynik, to w grafie istnieje osiągalny z v ujemny cykl (a jak nie poprawi, to taki cykl nie istnieje). Zatem aby znaleźć ujemny cykl w grafie możemy dodać do G sztuczne źródło s, które ma wychodzącą krawędź wagi 0 do każdego wierzchołka i sprawdzić, czy istnieje ujemny cykl osiągalny z s.

Algorytm **Dijkstry** wyznacza odległość z ustalonego wierzchołka v do wszystkich innych wierzchołków grafu przy założeniu, że wagi są nieujemne. Zaczyna on od zainicjalizowania Q będącego kolejką priorytetową, na której priorytetem wierzchołka u jest wartość $d(u)$ (obliczona odległość z v). Następnie działa w następujący sposób:

```
for all u ∈ V(G) do
    Q.insert(u)
end for
∀_{u≠v} d(u) ← +∞
d(v) ← 0
while ! Q.empty() do
    u ← Q.pop_min()
    for all (u,w) ∈ E(G) do
        if d(u) + w(u,w) < d(w) then
            d(w) ← d(u) + w(u,w)
        end if
    end for
end while
```

Poprawność algorytmu wynika z faktu (dowodzonego indukcyjnie), że w momencie ściągania u z Q wartość $d(u)$ jest poprawna (jeśli coś ściągamy, to nic na kolejce nie ma mniejszej odległości, więc nic nie poprawi ściąganej odległości, bo wagi są nieujemne). Zakładamy, że trzymamy krawędzie grafu w listach sąsiedztwa i możemy po prostu przeiterować się po krawędziach w wewnętrznej pętli. Jeśli Q jest zwykłą listą, to znajdowanie minimum działa w czasie liniowym i złożoność to $\Theta\left(n^2 + m\right)$. Możemy implementować Q jako kopiec: znajdowanie minimum i ściąganie go z kopca w czasie $\Theta(\log n)$, w momencie zmiany wartości $d(w)$ zmiana pozycji wierzchołka w w kopcu w czasie $\Theta(\log n)$ (w tym celu pamiętamy pozycję każdego wierzchołka w kopcu), co razem daje $\Theta((n+m)\log n)$. Można jeszcze zastosować kopiec Fibonacciego, który pozwoli nam zaimplementować operację zmniejszenia priorytetu w amortyzowanym czasie stałym, co da ostateczną złożoność $\Theta(n \log n + m)$.

Algorytm **Johnsona** wyznacza odległość między każdą parą wierzchołków grafu przy założeniu, że w grafie nie ma ujemnego cyklu (osiągalnego z v). Robi to w czasie $\mathcal{O}(nm \log n)$, czyli dla grafów rzadkich jest szybszy od Warshalla-Floyda. Działa on w następujący sposób:

1. Wykonujemy algorytm Bellmana-Forda na znajdowanie ujemnego cyklu, czyli tworzymy źródło s i puszczamy z niego ten algorytm, wyznaczając odległość $S(u)$ każdego wierzchołka u z s. Dla każdej krawędzi (u,w) definiujemy nową wagę tej krawędzi jako $w'(u,w) = w(u,w) + S(u) - S(w)$. Mamy $w'(u,w) \geqslant 0$ z nierówności trójkąta, bo $S(w) \leqslant S(u) + w(u,w)$. Jednocześnie suma wag na dowolnej ścieżce od v_0 do v_k zmieniła się o tę samą wartość $S(v_0) - S(v_k)$, czyli dokładnie te same ścieżki są najkrótsze.
2. Wykonujemy algorytm Dijkstry na nowych wagach n razy z każdym wierzchołkiem jako początkowym.
3. Na podstawie wyników dla zmienionych, nieujemnych wag odzyskujemy wyniki dla oryginalnych wag.

Można jeszcze dodać, że każdy z przedstawionych algorytmów pozwala łatwo odtworzyć rozwiązanie, bo przy każdej zmianie pamiętanej odległości jesteśmy w stanie zapamiętywać, jaka jest ostatnia krawędź ścieżki, która świadczy o tej zmianie.

II.2.5 Minimalne drzewa rozpinające

> Minimalne drzewa rozpinające w grafie, algorytmy Jarnika-Prima oraz Kruskala, uzasadnienie poprawności, analiza złożoności.

Niech $G = (V,E)$ będzie nieskierowanym ($|V| = n$ i $|E| = m$), spójnym grafem z nieujemnymi wagami na krawędziach. Chcemy znaleźć jego minimalne drzewo rozpinające (Minimal Spanning Tree, MST), czyli taki podzbiór krawędzi $E' \subseteq E$, że $T = (V,E')$ jest drzewem i $\sum_{e \in E'} w(e)$ jest najmniejsze możliwe.

Algorytm **Jarnika-Prima** jest zachłannym algorytmem znajdującym MST. Konstruuje on MST iteracyjnie wierzchołek po wierzchołku i działa w następujący sposób:

1. Zaczynamy od drzewa składającego się z jednego (dowolnego) wierzchołka v.

2. Tworzymy kolejkę priorytetową, na której są wierzchołki $V \setminus \{v\}$. Priorytetem jest minimalna ważona odległość wierzchołka od aktualnego MST (zaczynamy od 0 dla v i $+\infty$ dla pozostałych).
3. Dopóki w drzewie nie ma wszystkich wierzchołków, ściągamy wierzchołek z kolejki priorytetowej i dodajemy go do MST wraz z krawędzią realizującą najmniejszą odległość od MST (ten wierzchołek na pewno jest połączony krawędzią z MST, bo inaczej wierzchołki na ścieżce z niego do MST miałyby mniejszą odległość). Następnie aktualizujemy priorytety wierzchołków uwzględniając krawędzie wychodzące z właśnie dodanego wierzchołka.

Taki algorytm działa w czasie $\Theta\left((n+m)\log n\right)$ jeśli kolejka priorytetowa jest kopcem i $\Theta\left(n\log n + m\right)$ jeśli kolejka jest kopcem Fibonacciego.

Poprawności algorytmu dowodzimy indukcyjnie. Niech G_i będzie skonstruowanym przez nas drzewem po i-tym kroku (składającym się z i wierzchołków). Pokażemy, że istnieje optymalne drzewo Y_i dla całego grafu zawierające w sobie G_i. Na początku G_1 jest jednym wierzchołkiem, który jest zawarty w każdym drzewie rozpinającym. Niech G_{i-1} będzie zawarte w Y_{i-1}. Konstruując G_i znajdujemy krawędź e łączącą wierzchołek v należący do G_{i-1} z u nienależącym do G_{i-1}. Jeśli Y_{i-1} nie zawiera e, to zawiera pewną ścieżkę łączącą v z u. Ta ścieżka musi zawierać pewną krawędź f łączącą wierzchołek G_{i-1} z wierzchołkiem poza G_{i-1}. Mamy $w\left(f\right) \geqslant w\left(e\right)$ z wyboru e. Zatem ścieżkę łączącą v z u w Y_{i-1} można zastąpić krawędzią e nie zwiększając sumy wag w drzewie. Z tego wynika, że istnieje MST Y_i zawierające e, co kończy dowód.

Algorytm **Kruskala** również znajduje MST. Robi to iteracyjnie, krawędź po krawędzi i działa w następujący sposób:

1. Zaczynamy od inicjalizacji struktury DSU (Disjoint Set Union) na wierzchołkach G: każdy wierzchołek jest w swoim osobnym zbiorze.
2. Sortujemy krawędzie po ich wagach.
3. Przechodzimy krawędzie w kolejności rosnących wag. Jeśli aktualna krawędź łączy wierzchołki z różnych zbiorów, to łączymy je i dodajemy krawędź do MST.

Poprawności algorytmu dowodzimy identyczną techniką jak w algorytmie Jarnika-Prima. Strukturę DSU implementujemy mniej więcej tak: na każdym rozłącznym zbiorze utrzymujemy strukturę drzewa. Pamiętamy korzeń drzewa i rozmiar drzewa, a dla każdego węzła drzewa pamiętamy jego rodzica (dla korzenia jest to on sam). Aby sprawdzić, czy elementy są w tym samym zbiorze idziemy odnośnikami do rodziców aż do korzeni i sprawdzamy, czy wyszły nam te same korzenie. Aby połączyć dwa zbiory sprawdzamy, który jest mniejszy i ustawiamy korzeń większego zbioru jako rodzica korzenia mniejszego zbioru. W ten sposób nasze drzewa będą miały głębokość co najwyżej logarytmiczną od liczby elementów w strukturze DSU. Zatem operacje na DSU będą działały w czasie $\mathcal{O}\left(\log n\right)$. Cały algorytm Kruskala ma więc złożoność $\mathcal{O}\left(m\log m + m\log n\right) = \mathcal{O}\left(m\log n\right)$. Możemy zaimplementować DSU lepiej i zapewnić złożoność operacji $\mathcal{O}\left(\log^* n\right)$ lub nawet $\mathcal{O}\left(\alpha\left(n\right)\right)$ (gdzie α jest odwrotną funkcją Ackermanna), ale nie zmienia to ostatecznej złożoności algorytmu.

II.2.6 Przepływy

> Algorytm Edmondsa-Karpa znajdowania minimalnego przepływu z wykorzystaniem metody BFS i szkic analizy jego złożoności.

Pojęcie przepływu jest zdefiniowane przy odpowiednim pytaniu z Matematyki Dyskretnej. Tam również udowodniliśmy twierdzenie Forda-Fulkersona: przepływ w sieci przepływowej jest maksymalny wtedy i tylko wtedy, gdy nie istnieje dla niego ścieżka powiększająca. To twierdzenie daje nam metodę wyznaczania maksymalnego przepływu znaną jako metoda Forda-Fulkersona: znajdujemy ścieżkę powiększającą od źródła s do ujścia t, wyznaczamy, ile przepływu może przez nią przepłynąć, a następnie aktualizujemy przepływ zgodnie z tą wartością.

Znajdowanie ścieżki możemy zaimplementować za pomocą dowolnego przeszukiwania grafu: wystarczy rozważyć „graf rezydualny” w którym istnieją te krawędzie, po których da się przesunąć dodatkowy prze-

pływ w aktualnym przepływie (czyli nienasycone krawędzie sieci przepływowej i krawędzie odwrotne do tych, które mają niezerową wartość w przepływie). Ten graf jest ważony, waga krawędzi oznacza, ile przepływu można przez nią puścić (czyli puścić więcej przepływu lub cofnąć przepływ w sieci przepływowej).

Zakładając, że przepływ ma wartości całkowitoliczbowe wykonamy co najwyżej $|f|$ wyszukiwań ścieżek powiększających, gdzie $|f|$ jest wartością maksymalnego przepływu (bo każda taka ścieżka zwiększa wartość przepływu o co najmniej 1). Daje nam to złożoność $\mathcal{O}(|f|(n+m)) = \mathcal{O}(|f|m)$, co może być bardzo duże (w szczególności nie wielomianowe).

Okazuje się, że jeśli będziemy wyszukiwać ścieżki powiększające metodą BFS (czyli zawsze znajdować najkrótsze ścieżki powiększające), to dostaniemy wielomianowy algorytm znany jako algorytm Edmondsa-Karpa.

Krawędź krytyczna na ścieżce powiększającej to krawędź, która świadczy o dodatkowym przepływie, jaki można puścić przez sieć przepływową (czyli jej waga jest równa minimum wag na tej ścieżce powiększającej). Krawędź krytyczna dla danej ścieżki jest wysycana podczas zwiększania przepływu. Każda ścieżka powiększająca posiada co najmniej jedną krawędź krytyczną.

Pokażemy, że podczas wykonywania algorytmu Edmondsa-Karpa każda z $2m$ krawędzi w grafie rezydualnym może być krawędzią krytyczną co najwyżej $\frac{n}{2}$ razy. Zatem wykonamy co najwyżej $\mathcal{O}(nm)$ wyszukiwań ścieżek powiększających, co da nam całkowitą złożoność $\mathcal{O}(nm^2)$.

Propozycja. Odległości wierzchołków od źródła s w grafie rezydualnym nie maleją w trakcie wykonywania algorytmu. W szczególności długości ścieżek powiększających nie maleją w trakcie wykonywania algorytmu.

Dowód. Przez $d(v)$ i $d'(v)$ oznaczamy odległość v od s w grafie rezydualnym przed i po pewnym zwiększeniu przepływu, odpowiednio. Te grafy rezydualne oznaczamy G_f i G'_f, odpowiednio. Załóżmy nie wprost, że dla pewnego v zachodzi $d'(v) < d(v)$. Dodatkowo załóżmy, że wartość $d'(v)$ jest minimalna, czyli dla wierzchołków będących bliżej s taka nierówność nie zachodzi. Rozważmy najkrótszą ścieżkę od s do v w G'_f. Niech u będzie poprzednikiem v na niej. Wtedy (u, v) należy do G'_f i zachodzi $d'(v) = d'(u) + 1$. Wobec minimalności $d'(v)$ mamy $d'(u) \geqslant d(u)$.

Jeśli (u, v) należy do G_f, to z nierówności trójkąta mamy $d(v) \leqslant d(u)+1$. Zatem $d(v) \leqslant d'(u)+1 = d'(v)$, co jest sprzeczne z $d(v) > d'(v)$.

Jeśli (u, v) nie należy do G_f, ale należy do G'_f, to nasz algorytm musiał znaleźć w G_f ścieżkę powiększającą zawierającą (v, u). Z minimalności tej ścieżki mamy $d(u) = d(v) + 1$, co daje nam $d(v) = d(u) - 1 \leqslant d'(u) - 1 = d'(v) - 2$. To również jest sprzeczność. □

Ustalmy krawędź (u, v). Jeśli jest ona krytyczna na ścieżce powiększającej, to w grafie rezydualnym zachodzi $d(v) = d(u)+1$. Zwiększając przepływ wysycamy tę krawędź. Aby ponownie mogła się ona stać krytyczną musimy najpierw zwiększyć przepływ za pomocą pewnej ścieżki powiększającej zawierającej (v, u). W momencie takiego zwiększenia musi być $d'(u) = d'(v) + 1 \geqslant d(v) + 1 = d(u) + 2$. Zatem odległość od s do u wzrosła co najmniej o 2. Długość dowolnej ścieżki od s do u wynosi co najwyżej n, więc (u, v) może być krawędzią krytyczną co najwyżej $\frac{n}{2}$ razy. To kończy dowód.

II.2.7 Skojarzenia

> Algorytm Hopcrofta-Karpa znajdowania najliczniejszego skojarzenia w grafie dwudzielnym i jego złożoność (bez analizy).

Niech $G = (A \sqcup B, E)$ będzie grafem dwudzielnym. Chcemy znaleźć w nim skojarzenie maksymalne – najliczniejszy możliwy zbiór krawędzi taki, że w żadnych dwóch nie powtarzają się wierzchołki. Możemy łatwo sprowadzić ten problem do przepływu. Orientujemy krawędzie od A do B i nadajemy im wagę 1. Dodajemy do G źródło s i ujście t. Dodajemy krawędzie od s do każdego wierzchołka A i od każdego wierzchołka B do t. Te krawędzie również mają wagę 1. Łatwo zauważyć, że maksymalny przepływ w takiej sieci składa się z wyboru wierzchołków z A, wyboru wierzchołków z B i ich sparowania za pomocą

krawędzi z E, czyli jest dokładnie skojarzeniem. Algorytm Forda-Fulkersona działa w czasie $\mathcal{O}(|f|\,m)$. W tym zastosowaniu jest $|f| \leqslant n$, więc mamy złożoność $\mathcal{O}(nm)$.

Istnieje lepszy algorytm, zwany algorytmem Hopcrofta-Karpa. Działa on w czasie $\mathcal{O}(m\sqrt{n})$.

Niech M będzie pewnym skojarzeniem w G. Wolnymi wierzchołkami nazywamy wierzchołki nienależące do M. Ścieżką powiększającą nazywamy ścieżkę, która zaczyna się w wolnym wierzchołku, idzie na zmianę krawędziami spoza M i z M i kończy się w wolnym wierzchołku. Łatwo zauważyć, że taka ścieżka odpowiada dokładnie pojęciu ścieżki powiększającej w zdefiniowanej przedtem sieci przepływowej. Zwiększenie przepływu przez tę ścieżkę to w języku skojarzeń zastąpienie M przez różnicę symetryczną M i tej ścieżki. Z twierdzenia Forda-Fulkersona skojarzenie nie jest maksymalne dokładnie wtedy, gdy istnieją dla niego jakieś ścieżki powiększające.

Ideą algorytmu Hopcrofta-Karpa jest znajdowanie wielu rozłącznych ścieżek powiększających jednocześnie i rozszerzanie pewnego skojarzenia M. Robimy to w następujący sposób:

1. Niech $U \subseteq A$ będzie zbiorem wszystkich wolnych wierzchołków w A. Za pomocą algorytmu BFS tworzymy warstwowanie naszego grafu. Zaczynamy BFSa z wierzchołków U (zerowa warstwa), przechodzimy do ich sąsiadów (pierwsza warstwa) i tak dalej. Robimy to w taki sposób, że na zmianę chodzimy krawędziami nieskojarzonymi i skojarzonymi, czyli z wierzchołków z A wychodzimy krawędziami nieskojarzonymi, a z wierzchołków z B wychodzimy krawędziami skojarzonymi.

2. Kończymy BFSa na pierwszej warstwie k takiej, że jest w niej co najmniej jeden wolny wierzchołek z B. Oznaczmy zbiór takich wierzchołków przez V. Są one końcami pewnych ścieżek powiększających.

3. Za pomocą algorytmu DFS znajdujemy maksymalny (na zawieranie) zbiór rozłącznych ścieżek powiększających o długości k. Zaczynamy DFSa w (kolejnych) wierzchołkach V. Idziemy tylko krawędziami, które prowadzą do poprzedniej warstwy i na zmianę nieskojarzonymi i skojarzonymi krawędziami. Zauważmy, że jeśli odwiedzamy pewien wierzchołek, to możemy go na stałe oznaczyć jako odwiedzonego – nie ma sensu, aby DFS zaczęty w kolejnym wierzchołku go odwiedzał, bo jeśli aktualny DFS się uda, to znajdzie ścieżkę powiększającą zawierającą ten wierzchołek, a jeśli się nie uda, to kolejny też by się nie udał. Zatem wszystkie DFSy można wykonać w czasie $\mathcal{O}(m)$.

4. Za pomocą znalezionych ścieżek powiększamy M i zaczynamy algorytm od nowa. Kończymy, gdy BFS nie wykryje żadnych ścieżek powiększających.

Z faktu, że dodajemy do M maksymalny na zawieranie zbiór ścieżek powiększających długości k wynika, że po i iteracjach takiego algorytmu wszystkie ścieżki powiększające w grafie mają długość co najmniej $i + 1$. Załóżmy, że wykonaliśmy $\sqrt{n}$ iteracji. Niech M^* będzie pewnym skojarzeniem maksymalnym. W różnicy symetrycznej M i M^* wszystkie wierzchołki mają stopnie co najwyżej 2, więc ta różnica symetryczna jest sumą cykli i ścieżek. Cykle idą na zmianę krawędziami z M i M^*, a ścieżki są ścieżkami powiększającymi w M. Wiemy, że mają długość co najmniej $\sqrt{n}$, a więc jest ich co najwyżej $\sqrt{n}$. Zatem dodanie jeszcze $\sqrt{n}$ ścieżek powiększających do M da nam skojarzenie maksymalne, czyli wystarczy wykonać jeszcze kolejne $\sqrt{n}$ iteracji. Ostatecznie wykonamy około $2\sqrt{n}$ iteracji, co daje odpowiednią złożoność.

II.3 Algorytmy i Struktury Danych 2

II.3.1 Wyszukiwanie wzorca

Wyszukiwanie wzorca metodą prefikso-sufiksów (KMP), uogólnienie na wiele wzorców (Aho-Corasic).

Niech $w = w_1 \dots w_n$ będzie słowem nad pewnym alfabetem. Prefikso-sufiksem w nazywamy taki prefiks $w_1 \dots w_i$, który jest również sufiksem w, czyli jest równy $w_{n-i+1} \dots w_n$. Algorytm KMP (Knuth, Morris, Pratt) ma na celu wyznaczenie tablicy prefikso-sufiksów KMP słowa w, gdzie

$$\mathrm{KMP}\,(j) = \max\left\{1 \leqslant i < j : w_1 \dots w_i \text{ jest sufiksem } w_1 \dots w_j\right\}.$$

Robi to w następujący sposób:

```
KMP(0) ← −1
i ← −1
for j = 1, ..., n do
    while i ⩾ 0 ∧ w[j] ≠ w[i + 1] do
        i ← KMP(i)
    end while
    i ← i + 1
    KMP(j) = i
end for
```

Poprawność wynika z tego, że jeśli $i = \mathrm{KMP}(j-1)$ i $w[j] = w[i+1]$, to najdłuższy prefikso-sufiks $w_1 \dots w_{j-1}$ można przedłużyć o jeden znak (i nie da się o więcej, bo wtedy mielibyśmy za małą wartość $\mathrm{KMP}(j-1)$). Jeśli ten ostatni znak się nie zgadza, to zauważmy, że $\mathrm{KMP}\,(\mathrm{KMP}(j-1))$ również jest prefikso-sufiksem $w_1 \dots w_{j-1}$. Co więcej, jest najdłuższym takim prefikso-sufiksem krótszym od $w_1 \dots w_{\mathrm{KMP}(j-1)}$. Zatem jeśli poprzedniego prefikso-sufiksu nie dało się przedłużyć, to próbujemy przedłużyć ten (bo jeśli szukany prefikso-sufiks istnieje, to jest przedłużeniem pewnego prefikso-sufiksu $w_1 \dots w_{j-1}$). Jeśli nie znajdziemy pasującego prefikso-sufiksu, to ostatecznie ustalamy $\mathrm{KMP}(j) = 0$.

Zauważmy, że każde wykonanie wewnętrznej pętli zmniejsza i o co najmniej 1 i nie wykonujemy zmniejszeń poniżej -1. i zaczyna z wartością -1 i możemy zwiększyć i co najwyżej n razy (za każdym razem o 1), więc zmniejszyć możemy też co najwyżej n razy. To dowodzi złożoności $\mathcal{O}\,(n)$.

Teraz zastosowanie: wyszukiwanie wzorca. Mamy tekst $t = t_1 \dots t_n$ i wzorzec $p = p_1 \dots p_m$ z $m \leqslant n$. Chcemy znaleźć pierwsze wystąpienie p w t, czyli najmniejsze takie i, że $t_i \dots t_{i+m-1} = p_1 \dots p_m$. Trywialnie możemy przykładać wzorzec do każdej pozycji w tekście, co daje złożoność $\mathcal{O}\,(mn)$. Możemy zastosować algorytm KMP: rozważamy słowo $p\#t$, gdzie # jest literą spoza alfabetu, i obliczamy tablicę KMP tego słowa. Zauważmy, że $\mathrm{KMP}(j) \leqslant m$, bo # występuje w tym słowie tylko raz. Jeśli $\mathrm{KMP}(j) = m$, to znaleźliśmy słowo p jako sufiks $p\#t_1 \dots t_{j-m-1}$, czyli wystąpienie p w t. Przechodząc całą tablicę możemy znaleźć wszystkie wystąpienia p w t.

Uogólnimy ten algorytm na wiele wzorców za pomocą algorytmu Aho-Corasick. Mamy tekst t, $|t| = n$ i zbiór wzorców $P = \{p_1, \dots, p_k\}$ taki, że $\sum_{i=1}^{k} |p_i| = m$. Litery z tych tekstów pochodzą z alfabetu Σ. Szukamy w t wszystkich wystąpień wszystkich wzorców z P. Zrobimy to konstruując odpowiedni automat DFA i puszczając go na słowie t.

Zaczynamy od skonstruowania drzewa Trie dla zbioru P: tworzymy ukorzenione drzewo, w którym krawędzie są etykietowane literami z Σ. Zaczynamy od samego korzenia. Bierzemy kolejne wzorce z P i przechodzimy w dół drzewa od korzenia krawędziami etykietowanymi kolejnymi literami rozważanego słowa. W razie potrzeby dodajemy do drzewa nowe wierzchołki. W momencie, gdy skończymy przechodzić danym słowem, oznaczmy ostatni wierzchołek jako ten, w którym kończy się to słowo. Z wierzchołkiem

v takiego drzewa możemy utożsamić słowo s_v, które powstaje z liter na krawędziach ścieżki od korzenia do tego wierzchołka. W szczególności korzeniowi odpowiada słowo puste.

Zauważmy, że drzewo Trie jest właściwie DFA (z korzeniem jako stanem startowym), w którym brakuje niektórych przejść. Naszym celem będzie uzupełnienie ich w taki sposób, aby przepuszczenie słowa w przez taki automat kończyło się w takim stanie, że słowo odpowiadające temu stanowi (jako wierzchołkowi drzewa Trie) jest najdłuższym częściowym wystąpieniem pewnego słowa z P w słowie w. Inaczej mówiąc, chcemy, aby słowo odpowiadajace temu stanowi było najdłuższym sufiksem w, który jednocześnie jest prefiksem pewnego słowa z P. Oczywiście istniejące przejścia zachowują taką własność.

Będziemy dokładać przejścia do naszego DFA w kolejności BFSa puszczonego w korzeniu. Do korzenia dodajemy przejścia prowadzące do niego samego. Dla pozostałych wierzchołków definiujemy tak zwane połączenie sufiksowe. Prowadzi ono z v do takiego wierzchołka u, że s_u jest właściwym sufiksem s_v i jest najdłuższym takim sufiksem spośród wszystkich słów odpowiadających wierzchołkom drzewa Trie.

Aby znaleźć połączenie sufiksowe v rozważamy jego rodzica r w drzewie Trie. Niech c będzie literą prowadzącą z r do v. Wtedy połączenie sufiksowe v polega na przejściu połączeniem sufiksowym r i następnie literą c.

Mając połączenie sufiksowe stanu v możemy zdefiniować jego brakujące przejścia. Jeśli jesteśmy w stanie v i chcemy przejść literą c (dla której przejście nie jest zdefiniowane), to przechodzimy połączeniem sufiksowym i przechodzimy literą c (to przejście jest zdefiniowane, bo sufiks jest wcześniej w kolejności BFS).

Zauważmy, że dla P składającego się z jednego słowa drzewo Trie jest ścieżką, a algorytm Aho-Corasick zachowuje się dokładnie jak KMP. Po przejściu pierwszymi i literami słowa w znajdziemy się w stanie, którego głębokość odpowiada wartości KMP(i). Dowód poprawności algorytmu Aho-Corasick jest identyczny jak algorytmu KMP.

Mając tak zdefiniowane DFA możemy przepuścić przez nie kolejne litery słowa t i jeśli w pewnym momencie weszliśmy do stanu oznaczonego jako koniec pewnego słowa z P, to znaleźliśmy dopasowanie wzorca. Skonstruowane DFA ma $\mathcal{O}(m)$ stanów, każdy ma $|\Sigma|$ wychodzących krawędzi. Zatem złożoność opisanych operacji to $\mathcal{O}(m|\Sigma| + n|\Sigma|)$.

II.3.2 Tablica sufiksowa

Tablice sufiksowe, tworzenie za pomocą algorytmu KMR, definicje funkcji LCP i LCP2, zastosowanie LCP2 do wyszukiwania łańcucha w tekście.

Algorytm Karpa-Millera-Rosenberga (KMR) i związane z nim tablice sufiksowe są innym niż KMP i Aho-Corasick podejściem do wyszukiwania wzorca w tekście. Niech $x = x_1 \dots x_n$ będzie słowem, $m > 0$ liczbą całkowitą. Definiujemy tablicę $\text{KMR}_m[1 \dots n-m+1]$ w następujący sposób:

$\text{KMR}_m(i) = k$ wtedy i tylko wtedy, gdy podsłowo $x_i \dots x_{i+m-1}$ jest k-te w kolejności leksykograficznej w zbiorze wszystkich różnych podsłów x długości m.

Możemy myśleć o tej wartości jak o haszu tego podsłowa, który pozwala nam łatwo porównywać podsłowa długości m.

Zauważmy, że $\text{KMR}_m(i) = \text{KMR}_m(j)$ wtedy i tylko wtedy, gdy słowa $x_i \dots x_{i+m-1}$ oraz $x_j \dots x_{j+m-1}$ są równe. Z tego wynika, że $\text{KMR}_{2m}(i) = \text{KMR}_{2m}(j)$ wtedy i tylko wtedy, gdy $\text{KMR}_m(i) = \text{KMR}_m(j)$ i $\text{KMR}_m(i+m) = \text{KMR}_m(j+m)$. Ta obserwacja pozwala skonstruować następujący algorytm wyznaczania tablicy KMR w przypadku, gdy $m = 2^q$:

1. Obliczamy KMR_1 przypisując literom ich pozycje w alfabecie.
2. Zakładamy, że mamy już KMR_r. Chcemy wyznaczyć KMR_{2r}.
3. Tworzymy ciąg trójek $(\text{KMR}_r(i), \text{KMR}_r(i+r), i)$ dla $i = 1, \dots, n-2r+1$ i sortujemy go leksykograficznie.

4. Dzielimy posortowany ciąg na grupy o tej samej wartości pierwszych dwóch współrzędnych. Jeśli trójka (g, h, i) jest w k-tej grupie (patrząc leksykograficznie), to $\mathrm{KMR}_{2r}(i) = k$.

Iteracja ostatnich dwóch kroków pozwala nam wyznaczyć KMR_m dla $m = 2^q$. Jeśli m nie jest potęgą dwójki, to robimy to samo dla $m' = 2^{\lfloor \log_2 m \rfloor}$, a następnie wykonujemy jeszcze jedną iterację, w której drugi element w trójce to $\mathrm{KMR}_{m'}(i + m - m')$.

Ten algorytm działa w pamięci $\mathcal{O}(n)$. Wykonujemy w nim $\log m$ razy sortowanie, więc naiwna złożoność to $\mathcal{O}(n \log n \log m)$. Zauważmy jednak, że sortowane elementy przyjmują co najwyżej n^3 wartości (poza być może pierwszą iteracją, jeśli alfabet jest nieograniczony). Zatem można zastosować radixsorta i posortować liniowo, co da nam złożoność $\mathcal{O}(n \log m)$.

Za pomocą tablicy KMR możemy łatwo znaleźć wystąpienia wzorca p w tekście t. Jeśli $|t| = n$ i $|p| = m$, to obliczamy KMR_m dla słowa $x = t\#p$, gdzie $\#$ jest poza alfabetem. Wtedy $k = \mathrm{KMR}_m(n + 2)$ to numer podsłowa p. Możemy przejść całą tablicę i znaleźć inne podsłowa o numerze k.

Niech $x = x_1 \dots x_n$ będzie słowem. Oznaczmy i-ty sufiks x przez $x^{(i)} = x_i \dots x_n$. Definiujemy tablicę sufiksową $\mathrm{SUF}[1 \dots n]$ w następujący sposób:

$$\mathrm{SUF}(j) = i \text{ wtedy i tylko wtedy, gdy } x^{(i)} \text{ jest } j\text{-tym leksykograficznie sufiksem } x.$$

SUF jest pewną permutacją $\{1, \dots, n\}$. Jeśli chcemy wyszukać wzorzec y długości m w tekście x długości n, dla którego mamy wyznaczoną tablicę sufiksową, to można zastosować wyszukiwanie binarne. Wiemy, że jeśli y jest podsłowem x na pozycji i, to jest prefiksem $x^{(i)}$. Zatem porównujemy y z sufiksem o indeksie $\mathrm{SUF}(k)$, gdzie $k = \left\lfloor \frac{n}{2} \right\rfloor$. Jeśli nie znajdziemy dopasowania, to wiemy, czy należy kontynuować poszukiwanie wśród sufiksów mniejszych, czy większych. To daje nam złożoność $\mathcal{O}(m \log n)$. Analogicznie możemy wyznaczyć liczbę wystąpień wzorca w tekście (znajdujemy największy i najmniejszy pasujący sufiks i bierzemy różnicę).

Tablica sufiksowa jest w pewnym sensie odwrotnością tablicy KMR – zamiast przypisywać słowu indeks przypisujemy indeksowi słowo. Jeśli chcemy wyznaczyć tablicę sufiksową dla słowa x długości n, to rozważamy słowo $y = x\#^{n-1}$, gdzie $\#$ jest literą spoza alfabetu, które uznajemy za mniejszą od wszystkich liter. Wyznaczamy KMR_n słowa y. Mamy $\mathrm{SUF}(i) = j \iff \mathrm{KMR}_n(j) = i$. To daje nam algorytm wyznaczania tablicy sufiksowej w czasie $\mathcal{O}(n \log n)$. Istnieją też liniowe algorytmy wyznaczające tablicę sufiksową, korzystające na przykład z konstrukcji tak zwanego drzewa sufiksowego.

Czasami wyznacza się też tak zwane tablice LCP i LCP2 (longest common prefix), które pomagają w efektywnym wykorzystywaniu tablicy sufiksowej. Określamy $\mathrm{LCP}(i)$ jako długość najdłuższego wspólnego prefiksu sufiksów o indeksach $\mathrm{SUF}(i)$ i $\mathrm{SUF}(i + 1)$. $\mathrm{LCP2}(i, j)$ oznacza długość najdłuższego wspólnego prefiksu sufiksów o indeksach $\mathrm{SUF}(i), \dots, \mathrm{SUF}(j)$.

Pokażemy, jak wykorzystać LCP2 do usprawnienie przedstawionego przedtem algorytmu wyszukiwania wzorca y w słowie x. Załóżmy, że wykonujemy właśnie wyszukiwanie binarne na podtablicy tablicy sufiksowej pomiędzy indeksami L i R. Niech M będzie środkowym indeksem. Oznaczmy przez ℓ i r długość najdłuższego wspólnego prefiksu y i odpowiednio $\mathrm{SUF}(L)$ oraz $\mathrm{SUF}(R)$. Na początku wyznaczamy te wartości wprost. Oznaczmy $k = \mathrm{LCP2}(L, M)$. Rozważymy przypadki, gdy $\ell > r$, sytuacja, gdy $\ell < r$ lub $\ell = r$ jest podobna.

- Jeśli $k > \ell$, to y pasuje do sufiksu $\mathrm{SUF}(M)$ aż do ℓ-tej pozycji, znak $(\ell + 1)$-szy w $\mathrm{SUF}(M)$ jest taki sam jak w $\mathrm{SUF}(L)$, czyli mniejszy niż w y. Zatem idziemy w prawo, czyli $L := M$, wartość ℓ nie zmienia się.
- Jeśli $k < \ell$, to $(k + 1)$-szy znak w y jest taki sam jak w $\mathrm{SUF}(L)$, czyli mniejszy niż w $\mathrm{SUF}(M)$. Zatem idziemy w lewo, czyli $R := M$, $r := k$.
- Jeśli $k = \ell$, to w słowach y i $\mathrm{SUF}(M)$ pierwsze ℓ znaków pokrywa się. Porównujemy kolejne znaki, aż znajdziemy indeks s, na którym się różnią. Idziemy odpowiednio w prawo lub lewo.

Zauważmy, że jeśli dokonamy porównania litery y z literą w odpowiednim sufiksie x z sukcesem (będą one takie same), to już nigdy nie będziemy wykonywać porównań dla tej litery. Jednocześnie mamy co najwyżej jedno nieudane porównanie w każdej iteracji wyszukiwania binarnego. To daje nam złożoność $\mathcal{O}(\log n + m)$.

Na koniec możemy wspomnieć o sposobach wyznaczania LCP2. Zaczynamy od wyznaczenia LCP. Robimy to po kolei dla $x^{(1)}, \dots, x^{(n)}$. Zauważamy, że jeśli policzyliśmy LCP dla $x^{(k)}$ wprost, to odpowiednia

wartość dla $x^{(k+1)}$ będzie mniejsza o co najwyżej 1, zatem można zacząć brutalne dopasowywania od późniejszej pozycji. To daje nam złożoność liniową. Następnie zauważmy, że nie potrzebujemy znać wszystkich wartości LCP2, tylko te, o które będziemy pytać w wyszukiwaniu binarnym. Takich indeksów jest liniowo wiele i tworzą one drzewo binarne, w którego liściach wartością LCP2 jest po prostu LCP. Można obliczać wartości w pozostałych węzłach tego drzewa biorąc minima z ich dzieci.

II.3.3 Zamiatanie w geometrii

Technika zamiatania w geometrii obliczeniowej, zastosowanie do znajdowania wszystkich przecięć w zbiorze odcinków na płaszczyźnie.

Ogólna idea techniki zamiatania: rozważamy jakieś obiekty na płaszczyźnie, sortujemy je po jednej ze współrzędnych i patrzymy na nie po kolei. Przedstawimy ją dokładniej na przykładzie algorytmu znajdującego przecięcia w zbiorze odcinków.

Na początku opiszemy algorytm, który stwierdza, czy istnieją dwa przecinające się odcinki w zbiorze odcinków S.

Rozważamy zbiór wszystkich końców odcinków. Będziemy je nazywać zdarzeniami. Sortujemy je po współrzędnej x. Będziemy je zamiatać od lewej do prawej, to znaczy utrzymywać strukturę „miotły", która będzie szła od lewej do prawej i reagowała w odpowiedni sposób na napotykane zdarzenia. Miotła będzie drzewem BST (setem), które trzyma w sobie odcinki posortowane po współrzędnej y aktualnego punktu ich przecięcia z miotłą. W momencie zdarzenia „początek odcinka" dodajemy ten odcinek na miotłę, a w momencie zdarzenia „koniec odcinka" ściągamy go z miotły. Zauważmy, że jeśli dwa odcinki się przecinają, to w pewnym momencie przed tym przecięciem są sąsiednie na miotle. Zatem w momencie dodawania odcinka możemy sprawdzać, czy przecina się z odcinkami, z którymi sąsiaduje, a w momencie ściągania odcinka sprawdzamy, czy nowe sąsiednie odcinki się przecinają.

Mając odcinek AB i punkt C możemy sprawdzić, czy C leży na lewo, czy na prawo od AB (czy jest z nim współliniowy). Robimy to licząc iloczyn wektorowy AB i AC, czyli wyznacznik $\begin{vmatrix} (B-A).x & (C-A).x \\ (B-A).y & (C-A).y \end{vmatrix}$. Ma on znak taki jak sinus kąta między tymi wektorami, więc jak jest dodatni, to C leży na lewo od AB. Sprawdzanie przecięcia odcinków realizujemy zauważając, że jeśli odcinki się przecinają, to albo koniec jednego leży na drugim, albo końce jednego leżą po różnych stronach drugiego i na odwrót.

Zauważmy, że nie musimy rekalkulować porządku elementów na miotle – jeśli dwa odcinki na miotle zmieniają kolejność, to się przecinają.

Pojedyncza operacja na miotle ma złożoność logarytmiczną, zatem ten algorytm ma złożoność $\mathcal{O}(n \log n)$. Zaimplementowanie tego algorytmu poprawnie wymaga rozważenia wielu przypadków brzegowych (np. odcinki pionowe, odcinki nachodzące na siebie), ale nie zmienia to nic w idei algorytmu.

Aby zliczać wszystkie przecięcia odcinków wystarczy lekka modyfikacja tego algorytmu. Trzymamy zdarzenia w kolejce priorytetowej, w momencie wykrycia przecięcia (które na pewno będzie na prawo od miotły) dodajemy punkt przecięcia jako nowe zdarzenie. Obsługa takiego zdarzenia polega na zamianie kolejności przecinających się odcinków na miotle i sprawdzeniu, czy nowe pary odcinków sąsiednich generują nowe przecięcia. Mamy złożoność $\mathcal{O}((n+p)\log n)$, gdzie p to liczba przecięć, czyli pesymistycznie $\mathcal{O}(n^2 \log n)$ – gorzej niż algorytm trywialny, ale dla małej liczby przecięć jest dużo lepiej. Na kolejce trzymamy $\Theta(n+p)$ zdarzeń, czyli znowu pesymistycznie mamy złożoność pamięciową $\Theta(n^2)$. Możemy to poprawić – gdy odcinki przestają sąsiadować, to usuwamy z kolejki ich punkt przecięcia. W momencie gdy zaczynają sąsiadować dodajemy go z powrotem. W ten sposób w kolejce jest co najwyżej $n-1$ punktów przecięcia czekających na obsługę.

II.3.4 Otoczka wypukła

Algorytmy Jarvisa i Grahama znajdowania wypukłej otoczki, problem optymalności obu algorytmów.

Mając ustalony punkt p_0 możemy zdefiniować współrzędną kątową innego punktu p_1 jako kąt nachylenia wektora $p_1 - p_0$ do osi OX. Oznaczamy ten kąt $\alpha(p_0, p_1)$. Mając dwa punkty p_1, p_2 możemy stwierdzić, czy $\alpha(p_0, p_1) < \alpha(p_0, p_2)$ sprawdzając, czy p_2 leży na prawo od odcinka p_0p_1, czy na lewo. Mając taką metodę porównywania współrzędnych kątowych możemy posortować zbiór punktów kątowo za pomocą za pomocą dowolnego algorytmu sortującego przez porównania.

Otoczką wypukłą zbioru punktów P nazywamy wielokąt wypukły o wierzchołkach w punktach P, w którym zawierają się wszystkie punkty P. Naszym celem będzie znalezienie listy L punktów stanowiących wierzchołki otoczki, w kolejności przeciwnej do ruchu wskazówek zegara. Zazwyczaj zakładamy, że na otoczce umieszczamy tylko niezbędne punkty (czyli nie ma trzech współliniowych punktów), ale nie we wszystkich zastosowaniach jest to pożądane.

Pierwszym algorytmem jaki przedstawimy będzie algorytm Jarvisa. Działa on w następujący sposób:

1. Znajdujemy punkt p_0 o najmniejszej współrzędnej y. Jeśli jest takich wiele, to wybieramy ten o największej współrzędnej x. Inaczej mówiąc, wybieramy najbardziej prawy spośród najbardziej dolnych punktów. Wstawiamy p_0 do L.
2. Spośród wszystkich punktów P wybieramy taki punkt p_1, że $\alpha(p_0, p_1)$ jest najmniejsze możliwe (w przypadku równości wybieramy ten najdalszy od p_0). Dodajemy go to L.
3. W kolejnych krokach wybieramy p_{i+1} takie, że kąt $\alpha(p_i, p_{i+1})$ jest najmniejsze możliwe, ale większe niż $\alpha(p_{i-1}, p_i)$ (ten warunek ma znaczenie w lewej części otoczki, za jej punktem najwyższym – będą istniały punkty wewnątrz otoczki o mniejszym kącie, niż punkt, który powinniśmy wybrać). Analogicznie jak przedtem rozwiązujemy remisy.
4. Kończymy, gdy wrócimy do p_0.

Łatwo zauważyć, że taki algorytm jest poprawny. Mając n punktów wybór odpowiedniego minimum wymaga $\Theta(n)$ operacji. Robimy to k razy, gdzie k jest liczbą punktów na otoczce. Zatem złożoność algorytmu to $\Theta(nk)$. Pesymistycznie jest to $\Theta(n^2)$, ale jeśli mamy gwarancję, że otoczka jest mała, to algorytm jest liniowym.

Innym pomysłem jest algorytm Grahama. Działa on tak:

1. Znajdujemy punkt p_0 najniższy w P. W przypadku remisów wybieramy najbardziej lewy taki punkt. Dodajemy p_0 do L.
2. Sortujemy pozostałe punkty kątowo względem p_0. Przy remisie punkty bardziej odległe od p_0 są większe. Oznaczmy posortowane punkty przez $p_1, \ldots, p_{n-1}$. Dodajemy p_1 do L.
3. Przeglądamy pozostałe punkty w kolejności tego sortowania. Załóżmy, że rozważamy właśnie p_i. Niech q_1 będzie ostatnim punktem w L, a q_0 przedostatnim. Jeśli q_1 nie leży na prawo od odcinka q_0p_i, to usuwamy go z L. Powtarzamy taką operację, aż nie usuniemy punktu. Wtedy dodajemy p_i do L.

W ten sposób po posortowaniu wyznaczyliśmy otoczkę w czasie liniowym. Zatem cały algorytm ma złożoność $\Theta(n \log n)$. Jest więc lepszy w pesymistycznym przypadku od algorytmu Jarvisa.

Okazuje się, że nie istnieje algorytm o lepszej złożoności (zakładając, że naszym modelem obliczeń są drzewa decyzyjne). Możemy bowiem zredukować sortowanie do problemu otoczki wypukłej. Mając n liczb rzeczywistych $x_1, \ldots, x_n$ traktujemy je jak punkty na osi OX. Rzutujemy je na parabolę, czyli rozważamy punkty $(x_1, x^2), \ldots, (x_n, x_n^2)$. Parabola jest wypukła, więc wszystkie te punkty będą na otoczce, a dodatkowo będą posortowane. To pozwala nam znaleźć posortowaną kolejność oryginalnych liczb.

II.3.5 Testowanie pierwszości

Liczby pierwsze, algorytm Millera-Rabina, złożoność, prawdopodobieństwo poprawności w przypadku liczby dowolnej (bez dowodu).

Test Fermata. Na wejściu dostajemy liczbę całkowitą n. Sprawdzamy jej pierwszość:

1. Losujemy $a \in [1, n-1]$.
2. Jeśli $\gcd(a, n) > 1$, to n jest złożona.
3. Jeśli $a^{n-1} \equiv 1 \pmod n$ stwierdzamy, że n jest pierwsza, w przeciwnym razie jest złożona.

Jeśli liczba n jest pierwsza, to algorytm da nam prawdziwą odpowiedź. Jeśli jest złożona, to może się zdarzyć, że $a^{n-1} \equiv 1 \mod n$. Wtedy mówimy, że n jest pseudopierwsza przy podstawie a.

Może się zdarzyć, że złożone n jest pseudopierwsze przy każdej podstawie (względnie pierwszej z n). Na przykład dla $n = 561 = 3 \cdot 11 \cdot 17$ z $2, 10, 16 \mid 560$ wynika, że $a^{560} \equiv 1 \pmod{561}$ (małe twierdzenie Fermata na czynnikach pierwszych).

Złożone liczby, które są pseudopierwsze przy każdej względnie pierwszej z nimi podstawie nazywamy liczbami Carmichaela. Wiemy, że takich liczb jest nieskończenie wiele. Dla nich test Fermata nie działa.

Twierdzenie (Test Millera-Rabina)**.** Na wejściu dostajemy liczbę całkowitą n. Sprawdzamy jej pierwszość:

1. Najpierw sprawdzamy parzystość.
2. Przedstawiamy $n - 1 = k \cdot 2^s$, gdzie k jest nieparzyste.
3. Losujemy $a \in [1, \ldots, n-1]$ i upewniamy się, że $\gcd(a, n) = 1$.
4. Wyznaczamy $(r_1, \ldots, r_s) = \left(a^k, a^{2k}, \ldots, a^{2^{s-1}k}, a^{n-1}\right) \mod n$.
5. Jeśli $r_s \neq 1$ lub $r_i \neq \pm 1$ i $r_{i+1} = 1$ dla pewnego $i < s$, to n jest złożona.
6. W przeciwnym wypadku stwierdzamy, że n jest pierwsza.

Ten test zawsze działa dla liczb pierwszych, dla złożonych myli się z prawdopodobieństwem co najwyżej $\frac{1}{2}$.

Dowód. Poprawność algorytmu dla liczb pierwszych wynika z małego twierdzenia Fermata i tego, że $x^2 \equiv 1 \pmod p$ implikuje $x \equiv \pm 1 \pmod p$. Załóżmy teraz, że n jest złożone.

Załóżmy, że $n = p^\alpha$ jest potęgą liczby pierwszej. Załóżmy, że n spełnia test Fermata przy podstawie $p+1$ (oczywiście $p + 1 \perp n$), to znaczy $(p+1)^{p^\alpha - 1} \equiv 1 \pmod{p^\alpha}$. Redukując modulo p^2 dostajemy

$$1 \equiv (p+1)^{p^\alpha - 1} = \sum_{i=0}^{p^\alpha - 1} \binom{p^\alpha - 1}{i} p^i \equiv \binom{p^\alpha - 1}{0} + \binom{p^\alpha - 1}{1} p = p^{\alpha+1} - p + 1 \equiv 1 - p \pmod{p^2},$$

ale $1 \not\equiv 1 - p \pmod{p^2}$ – sprzeczność. Zatem n nie spełnia testu Fermata (a więc również testu Rabina-Millera) dla podstawy $p+1$. Zauważmy, że $\left\{a \in \mathbb{Z}_n^* : a^{n-1} \equiv 1 \pmod n\right\}$ jest podgrupą $\mathbb{Z}_n^*$. Wykazaliśmy, że nie jest ona całą grupą, a więc zawiera co najwyżej połowę elementów $\mathbb{Z}_n^*$. Zatem dla co najmniej połowy $a \in \mathbb{Z}_n^*$ już test Fermata wykaże złożoność.

Załóżmy teraz, że $n = uv$ dla $\gcd(u, v) = 1$. Dla s takiego, że $\frac{n-1}{2^s}$ jest nieparzyste mamy $(-1)^{\frac{n-1}{2^s}} = -1$. Zatem istnieje największe takie $m = \frac{n-1}{2^t}$, że dla pewnego $b \in \mathbb{Z}_n^*$ zachodzi $b^m \not\equiv 1 \pmod n$. Jeśli $m = n - 1$, to już test Fermata wykaże złożoność n. Możemy więc założyć, że $m < n - 1$ i $a^{2m} \equiv 1 \pmod n$ dla każdego $a \in \mathbb{Z}_n^*$.

Pokażemy, że istnieje b' takie, że $b'^m \not\equiv \pm 1 \pmod n$. Możemy założyć, że $b^m \equiv -1 \pmod n$. Z chińskiego twierdzenia o resztach istnieje takie x, że $x \equiv 1 \pmod u$ i $x \equiv b \pmod v$. Wtedy $x^m \equiv 1 \pmod u$ i $x^m \equiv -1 \pmod v$. Zatem x^m nie może przystawać do 1 ani -1 modulo n. Możemy więc przyjąć $b' = x$. Zbiór $\{a \in \mathbb{Z}_n^* : a^m \equiv \pm 1 \pmod n\}$ jest podgrupą $\mathbb{Z}_n^*$. Dowiedliśmy właśnie, że nie jest ona całą grupą,

zatem podobnie jak w poprzednim przypadku dla co najmniej połowy $a \in \mathbb{Z}_n^*$ test Rabina-Millera wykaże złożoność. □

Można pokazać, że test Millera-Rabina myli się z prawdopodobieństwem nie większym niż $\frac{1}{4}$, a w praktyce jeszcze mniejszym. Jego wielokrotne powtórzenie daje nam algorytm mylący się z bardzo małym prawdopodobieństwem.

W jednej iteracji algorytmu wykonujemy $O(\log n)$ mnożeń liczb długości $\log n$. Zatem całkowita złożoność to $O\left(\log^3 n\right)$ (lub lepsza przy lepszym mnożeniu).

Jeśli $n < 2^{64}$, to wystarczy przyjąć za a wszystkie liczby pierwsze do 37.

Algorytm Millera-Rabina jest najczęściej stosowanym w praktyce algorytmem sprawdzającym pierwszość. Co ciekawe, istnieje wielomianowy, deterministyczny algorytm sprawdzający pierwszość. Jest to tak zwany algorytm AKS (Agrawal, Kayal, Saxena).

II.3.6 Programowanie liniowe

Programowanie liniowe – postać standardowa i dopełnieniowa, interpretacja geometryczna, klasyfikacja ze względu na liczbę rozwiązań, algorytm sympleks (szkic) i podstawowe fakty dotyczące jego złożoności.

Programem liniowym nazywamy trójkę $L = (A, b, c)$, gdzie $A = [a_{ij}] \in \mathbb{R}^{m\times n}$, $c = [c_j] \in \mathbb{R}^n$, $b = [b_i] \in \mathbb{R}^m$. Naszym celem jest zmaksymalizowanie funkcji celu $\sum_{j=1}^n c_j x_j$ zachowując warunki $\sum_{j=1}^n a_{ij}x_j \leqslant b_i$ dla $i = 1,\dots,m$ i $x_j \geqslant 0$ dla $j = 1,\dots,n$. W bardziej zwartym zapisie maksymalizujemy $c^T x$ przy $Ax \leqslant b$ i $x \geqslant 0$. Taką postać programu liniowego nazywamy postacią standardową. Możemy rozważać też inne postaci, które jednak da się łatwo zamienić na postać standardową. Na przykład:

- jeśli funkcja celu jest minimalizacją, to możemy zamienić min $c^T x$ na max $(-c)^T x$.
- jeśli mamy ograniczenie większościowe, czyli $a_i^T x \geqslant b_i$, to możemy zamienić je na $(-a_i)^T x \leqslant b_i$.
- jeśli mamy ograniczenia równościowe, czyli $a_i^T x = b_i$, to możemy zamienić je na $a_i^T x \leqslant b_i$ i $(-a_i)^T x \leqslant b_i$.

Istnieje tak zwana postać kanoniczna, w której nie mamy warunków nieujemności. Możemy ją zamienić na postać standardową zamieniając każdą zmienną x_i na dwie zmienne x_i^+ i x_i^-, zastępując x_i przez $x_i^+ - x_i^-$ i dodając $x_i^+, x_i^- \geqslant 0$.

W postaci dopełnieniowej mamy tylko ograniczenia równościowe. Aby zamienić postać standardową na dopełnieniową dla każdej nierówności $a_i^T x \leqslant b_i$ wprowadzamy nową zmienną dopełnieniową s_i i zastępujemy tę nierówność przez $a_i^T x + s_i = b_i$ oraz $s_i \geqslant 0$.

Niech $L = (A, b, c)$ będzie programem liniowym w postaci standardowej. $x \in \mathbb{R}^n$ takie, że $x \geqslant 0$ i $Ax \leqslant b$ nazywamy rozwiązaniem dopuszczalnym. Jeśli L posiada rozwiązanie dopuszczalne, to mówimy, że L jest dopuszczalny. W przeciwnym wypadku jest sprzeczny. Jeśli L ma rozwiązanie dopuszczalne lecz nie istnieje maksimum funkcji celu, to L jest nieograniczony.

Zbiór punktów x spełniających $a^T x = b$ dla $a, x \in \mathbb{R}^n$ i $b \in \mathbb{R}$ jest hiperpłaszczyzną $\mathbb{R}^n$ (czyli podprzestrzenią wymiaru $n - 1$). Z kolei zbiór punktów spełniających $a^T x \leqslant b$ jest półprzestrzenią. Zbiór rozwiązań dopuszczalnych programu liniowego jest przecięciem takich półprzestrzeni i hiperpłaszczyzn, czyli wielościanem wypukłym (być może nieograniczonym).

W dwóch wymiarach (czyli dla dwóch zmiennych) ograniczenia równościowe to proste. Ograniczenia nierównościowe to proste i wszystko nad (lub pod) taką prostą. Przecinając ze sobą obszary nad (lub pod) prostymi dostajemy wielokąt wypukły. Możemy rozważyć prostą normalną (prostopadłą) do wektora funkcji celu i przesuwać taką prostą w kierunku tego wektora. Ostatni punkt rozważanego wielokąta, z jakim ta przesuwana prosta będzie się przecinać, jest punktem maksymalizującym tę funkcję. Tę obserwację można uogólnić na więcej wymiarów.

Niech L będzie programem liniowym a P wielościanem jest rozwiązań dopuszczalnych.

- Wierzchołkiem P nazywamy dowolny jego punkt, który jest jedynym rozwiązaniem optymalnym dla pewnej funkcji celu.
- Punktem ekstremalnym P nazywamy dowolny jego punkt, który nie jest wypukłą kombinacją dwóch różnych punktów $y, z \in P$ (czyli nie zachodzi $x = ay + (1 - a)\,z$).
- Bazowe rozwiązanie dopuszczalne L to takie rozwiązanie dopuszczalne x, dla którego istnieje n (gdzie n jest liczbą zmiennych) liniowo niezależnych ograniczeń (to znaczy: takich, że odpowiadające im wiersze A są liniowo niezależne), które dla x są spełnione z równością. Taki punkt x jest wtedy przecięciem hiperpłaszczyzn, które te ograniczenia zadają i dla ustalonych ograniczeń jest on jedyny.

Okazuje się, że te trzy pojęcia są tożsame: x jest wierzchołkiem wtedy i tylko wtedy, gdy jest punktem ekstremalny i wtedy i tylko wtedy, gdy jest bazowym rozwiązaniem dopuszczalnym. Jeśli L jest ograniczonym programem liniowym, to ma rozwiązanie optymalne będące bazowym rozwiązaniem dopuszczalnym. Zatem istnieje algorytm „brut force" rozwiązujący program liniowy. Wystarczy wybrać n ograniczeń spośród wszystkich m i rozwiązać układ równań liniowych zadany tymi ograniczeniami za pomocą eliminacji Gaussa. Robimy to dla wszystkich wyborów n ograniczeń i wybieramy ten punkt, który maksymalizuje funkcję celu. Daje nam to złożoność $\mathcal{O}\left(\binom{m}{n}n^3\right)$.

Istnieje trochę lepszy algorytm znany jako metoda sympleks Dantziga. Zaczyna ona z pewnego bazowego rozwiązania dopuszczalnego (wierzchołka). Okazuje się, że jeśli to rozwiązanie nie jest optymalne, to istnieje sąsiedni wierzchołek, dla którego funkcja celu ma większą wartość (sąsiedni, czyli zadany przez zbiór równań różniący się o jedno równanie). Algorytm sympleks wybiera taki wierzchołek i idzie do niego, a następnie powtarza, aż nie będzie w stanie poprawić wyniku.

W praktyce wykonuje się obliczenia na postaci dopełnieniowej. Mamy n zmiennych początkowych (zwanych dalej zmiennymi bazowymi) i m zmiennych dopełnieniowych. Zmienna dopełnieniowa mówi, o ile można jeszcze zwiększyć wartość funkcji ograniczającej, zanim ograniczenie zostanie osiągnięte. Zaczynamy w pewnym wierzchołku, czyli mamy pewne wartości zmiennych początkowych, dla których n zmiennych dopełnieniowych jest równych 0, a pozostałe mają jakąś wartość. Funkcja celu jest postaci $c_1x_1 + \ldots + c_nx_n$, gdzie $x_1, \ldots, x_n$ to zmienne początkowe. Wybieramy pewne $c_i > 0$ (jeśli takiego nie ma, to jesteśmy w maksimum, bo zmienne muszą być dodatnie) i zwiększamy zmienną x_i o największą możliwą wartość, na którą pozwalają zmienne dopełnieniowe. Jest to wartość, dla której pewna zmienna dopełnieniowa x_j zaczyna wynosić 0 (jeśli taka zmienna nie istnieje, to program jest nieograniczony). Chcemy zamienić x_j w zmienną bazową, a x_i w zmienną dopełnieniową. Robimy to korzystając z ograniczenia, które właśnie stało się równością i zapisując x_i za pomocą x_j i pozostałych zmiennych bazowych. Podstawiamy takie wyrażenie pod x_i w funkcji celu. Następnie powtarzamy taką procedurę dla kolejnej zmiennej.

Taki algorytm może się zapętlić, ale jeśli zawsze będziemy wybierać zmienne o najmniejszym możliwym indeksie (tak zwana reguła Blanda), to na pewno się zatrzyma. Mamy wtedy złożoność $\mathcal{O}\left(\binom{m}{n}nm\right)$ – w jednym kroku przekształcamy każde z m ograniczeń długości n, a kroków wykonujemy co najwyżej $\binom{m}{n}$. Istnieją programy liniowe, dla których ten algorytm faktycznie wykonuje wykładniczo wiele operacji. Być może istnieje taki sposób wybierania zmiennych, dla którego sympleks działa wielomianowo, ale nie wiemy, jaki.

Pozostało zastanowić się, jak wyznaczyć początkowy wierzchołek. Jeśli $b_i \geqslant 0$ dla każdego i, to oczywiście rozwiązanie zerowe działa. Jeśli tak nie jest, to wprowadzamy nową zmienną x_0 i dopisujemy ją ze współczynnikiem -1 do każdej nierówności. W tak powstałym programie liniowym chcemy minimalizować funkcję x_0 (wartość tej zmiennej). Oczywiście znajdziemy wierzchołek początkowy, bo dla odpowiednio dużego x_0 rozwiązanie zerowe działa. Wykonując algorytm simpleks znajdziemy punkt $x_1, \ldots, x_n$ i wartość x_0, dla której spełnia on wszystkie równania. Jeśli $x_0 = 0$, to mamy wierzchołek startowy dla właściwego algorytmu. Jeśli nie, to początkowy program był sprzeczny.

II.3.7 Algorytmy aproksymacyjne

Algorytmy aproksymacyjne dla problemów obliczeniowo trudnych na przykładzie problemów POKRYCIE-WIERZCHOŁKOWE i KOMIWOJAŻER (współczynnik 2).

Dla instancji zadanego problemu optymalizacyjnego często istnieje wiele rozwiązań dopuszczalnych, a wśród nich mamy rozwiązania optymalne, czyli o minimalnym/maksymalnym koszcie.

Definicja. Algorytm aproksymacyjny to taki, który znajduje dowolne rozwiązanie dopuszczalne.

W najczęstszej sytuacji rozważany problem jest NP-trudny, a nam zależy na wielomianowym algorytmie, który znajduje rozwiązanie bliskie optymalnemu. Potrzebujemy więc jakiejś sensownej miary tej „bliskości".

Definicja. Dla instancji I definiujemy przez $A(I)$ koszt rozwiązania znaleziony przez algorytm A, zaś przez $OPT(I)$ koszt rozwiązania optymalnego. Wtedy funkcja $r(n)$ jest współczynnikiem efektywności algorytmu A, jeżeli

$$\max(A(I)/OPT(I), OPT(I)/A(I)) \leqslant r(n)$$

dla każdej instancji I o rozmiarze n.

Zauważmy, że dla algorytmu znajdującego rozwiązanie optymalne mamy $r(n) = 1$, zaś dla algorytmów aproksymacyjnych będzie $r(n) \geqslant 1$, przy czym im mniejsze, tym lepszy jest nasz algorytm. Najlepiej jest oczywiście, gdy $r(n)$ jest stałą niezależną od n.

POKRYCIE WIERZCHOŁKOWE

Mamy dany graf $G = (V, E)$. Szukamy najmniejszego zbioru wierzchołków $S \subseteq V$ takiego, że dla każdej krawędzi e co najmniej jeden z jej końców należy do S. Wersja decyzyjna tego problemu jest, jak wiadomo, NP-zupełna.

Rozważamy następujący algorytm aproksymacyjny:

```
S ← ∅
for e = uv ∈ E do
    if u ∉ S and v ∉ S then
        S ← S ∪ {u, v}
    end if
end for
return S
```

Niewątpliwie działa on w czasie wielomianowym.

Twierdzenie. Dla tego algorytmu zachodzi $r(n) \leqslant 2$.

Dowód. Rozważmy M – zbiór krawędzi, które spowodowały dodanie czegoś do S. Zauważmy, że ich końce muszą być parami różne, czyli M jest skojarzeniem. To oznacza, że każde pokrycie wierzchołkowe musi zawierać co najmniej jeden koniec każdej z tych krawędzi. W szczególności

$$A(G)/2 = |M| \geqslant OPT(G).$$

□

Zauważmy, że jeżeli G będzie zbiorem parami rozłącznych krawędzi, to nasz algorytm rzeczywiście wybierze dokładnie 2 razy więcej wierzchołków niż rozwiązanie optymalne. Jest to więc najlepsze możliwe szacowanie współczynnika efektywności tego algorytmu.

Ciekawostka: jest to najlepszy znany algorytm aproksymacyjny dla tego problemu.

PROBLEM KOMIWOJAŻERA

Mamy dany graf ważony pełny $G = (V, E, d)$. Szukamy w nim cyklu przechodzącego przez wszystkie wierzchołki o minimalnej sumie wag. Jest to klasyczny problem, który w wariancie decyzyjnym jest NP-zupełny. W tym przypadku skupiamy się na tak zwanym wariancie z nierównością trójkąta, tzn.

$d(u,v) \leqslant d(u,w) + d(w,v)$ dla dowolnych wierzchołków $u, v, w \in V$. Po dodaniu takiego bardzo naturalnego założenia problem nadal jest NP-zupełny.

Rozważamy następujący algorytm aproksymacyjny:

$s \leftarrow$ dowolny wierzchołek
$T \leftarrow$ minimalne drzewo rozpinające ukorzenione w s
$L \leftarrow$ lista wierzchołków w kolejności DFS drzewa T (każda krawędź dwukrotnie)
$C \leftarrow$ lista L po usunięciu powtórzeń wierzchołków (korzystamy z tego, że graf jest pełny)
return C

Widzimy, że działa on w czasie wielomianowym.

Twierdzenie. Dla tego algorytmu zachodzi $r(n) \leqslant 2$.

Dowód. Oznaczamy przez $c(A)$ sumę wag dla $A \subseteq E$. Widzimy, że $c(T) \leqslant OPT(G)$, bo usuwając jedną krawędź z optymalnego rozwiązania otrzymujemy poprawne drzewo rozpinające. Z definicji L mamy $c(L) = 2 \cdot c(T)$. Z nierówności trójkąta dostajemy $c(C) \leqslant c(L)$ (jeżeli na liście L są kolejno wierzchołki x, y, z, to usunięcie z niej wierzchołka y wymienia krawędzie xy, yz na krawędź xz). Składając wszystko razem

$$c(C) \leqslant c(L) = 2 \cdot c(T) \leqslant 2 \cdot OPT(G)$$

□

Ciekawostka: dla tego wariantu problemu komiwojażera znamy jeszcze lepsze algorytmy aproksymacyjne. Można za to pokazać, że dla ogólnej wersji problemu nie istnieje żaden algorytm aproksymacyjny o stałym współczynniku efektywności, o ile P $\neq$ NP.

II.4 Metody Programowania

II.4.1 Algorytm Karatsuby, metoda Strassena

Algorytm Karatsuby mnożenia liczb binarnych oraz idea szybkiego mnożenia macierzy metodą Strassena (bez dokładnej znajomości wzorów) jako przykłady zastosowania techniki „dziel i zwyciężaj" konstrukcji algorytmów, z wykorzystaniem twierdzenia o rekurencji uniwersalnej do oszacowania złożoności.

Zazwyczaj gdy operujemy na liczbach zakładamy, że operacje takie jak dodawanie i mnożenie jesteśmy w stanie wykonać w czasie stałym. Przestaje to być prawdą, gdy zaczynamy zajmować się bardzo dużymi liczbami, których nie jesteśmy w stanie zmieścić w jednym słowie maszynowym (ani nawet w kilku). Wtedy musimy zastanowić się, w jaki sposób dokładnie wykonujemy operacje na liczbach i jaką mają złożoność.

Liczby całkowite reprezentujemy jako ciągi binarne (być może ze znakiem). Dodawanie jest proste, bo naiwne dodawanie liczb n-cyfrowych cyfra po cyfrze ma złożoność $\mathcal{O}(n)$, co jest optymalne. Z mnożeniem jest już gorzej, bo klasyczny algorytm mnożenia działa w czasie $\mathcal{O}\left(n^2\right)$. Mamy lepsze algorytmy: Karatsuba $\mathcal{O}\left(n^{\log_2 3}\right)$, Toom-Cook $\mathcal{O}\left(n^{\log_3 5}\right)$, Schönhage-Strassen $\mathcal{O}(n \log n \log\log n)$, Fürer $\mathcal{O}\left(n \log n 2^{\mathcal{O}(\log^* n)}\right)$ z 2007 i Harver, van der Hoeven $\mathcal{O}(n \log n)$ z 2019.

Tutaj zajmiemy się algorytmem Karatsuby. Mamy do pomnożenia liczby A, B długości n (zakładamy $n = 2^k$ dopełniając zerami). Dzielimy je na pół $A = A_1 \cdot K + A_0$, $B = B_1 \cdot K + B_0$, gdzie $K = 2^{\frac{n}{2}}$.

Zachodzi $AB = A_1B_1K^2 + (A_0B_1 + A_1B_0)K + A_0B_0$. Do tego możemy przekształcić $A_0B_1 + A_1B_0 = (A_0 + A_1) \cdot (B_0 + B_1) - A_0B_0 - A_1B_1$. Zatem wystarczy wykonać 3 rekurencyjne mnożenia zamiast 4 (i kilka dodawań). Daje nam to rekursję $T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n)$. Mamy $n^{\log_2 3} > n$ i na mocy twierdzenia o rekurencji uniwersalnej jest $T(n) = \mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}\left(n^{1.59}\right)$.

Podobną ideę możemy zastosować do mnożenia macierzy, co da nam tak zwany algorytm Strassena. Mamy do pomnożenia macierze kwadratowe A, B rozmiaru $n \times n$, gdzie $n = 2^n$ (dopełniamy macierze zerami). Chcemy wyznaczyć $C = AB$. Możemy rozbić te macierze na macierze $\frac{n}{2} \times \frac{n}{2}$:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \qquad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} \qquad .$$

Zachodzi

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}, \qquad C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2},$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}, \qquad C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}.$$

Rekurencyjne wykonanie tych 8 mnożeń daje nam rekursję $T(n) = 8T\left(\frac{n}{2}\right) + \mathcal{O}\left(n^2\right)$ o rozwiązaniu $T(n) = \mathcal{O}\left(n^3\right)$. Na szczęście można sprytnie poprzekształcać te równania. Okazuje się, że dla

$$P_1 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$P_2 = (A_{1,1} + A_{1,2})B_{2,2}$$
$$P_3 = (A_{2,1} + A_{2,2})B_{1,1}$$
$$P_4 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$P_5 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$P_6 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$
$$P_7 = (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2})$$

mamy

$$C_{1,1} = P_5 + P_4 - P_2 + P_6, \qquad C_{1,2} = P_1 + P_2,$$
$$C_{2,1} = P_3 + P_4, \qquad C_{2,2} = P_5 + P_1 - P_3 - P_7.$$

To pozwala nam wykonać tylko 7 mnożeń rekurencyjnych. Mamy równanie $T(n) = 7T\left(\frac{n}{2}\right) + \mathcal{O}\left(n^2\right)$. Wobec $n^{\log_2 7} > n^2$ twierdzenie o rekurencji uniwersalnej daje nam $T(n) = \mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}\left(n^{2.81}\right)$.

Oba te algorytmy są przykładami zastosowania techniki „dziel i zwyciężaj", która polega na podzieleniu problemu na mniejsze podproblemy, rozwiązaniu ich rekurencyjnie a następnie scaleniu rozwiązań w jedno. Przedstawione algorytmy mają zdecydowanie większą stałą niż algorytmy naiwne, co wynika z faktu, że wykonujemy dużo więcej dodawań, a do tego mamy narzut związany z wywołaniami rekurencyjnymi. Dlatego dla małych danych wciąż lepiej stosować algorytmy naiwne.

II.4.2 Mergesort i quicksort

> Sortowanie przez scalanie i sortowanie szybkie (quicksort): idea działania, złożoność, zalety, wady.

Sortowanie przez scalanie (mergesort) działa na zasadzie „dziel i zwyciężaj". Procedura sortowania przebiega następująco:

1. Dzielimy sortowany ciąg na pół i wywołujemy się rekurencyjnie na połówkach, bazą rekursji są jednoelementowe tablice.
2. Po posortowaniu połówek scalamy rozwiązanie: alokujemy drugą tablicę i tworzymy wskaźniki na pierwszy element w pierwszej i drugiej połówce; umieszczamy na pierwszym indeksie w nowej tablicy mniejszy z tych elementów i zwiększamy odpowiedni wskaźnik, kontynuując w ten sposób dostajemy posortowaną tablicę.
3. Podstawiamy wartości z posortowanej tablicy w wejściową tablicę.

Taki algorytm ma równanie rekurencyjnie $T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$, co daje nam złożoność $\mathcal{O}(n \log n)$.

Mergesort jesteśmy w stanie zaimplementować tak, aby był stabilny (elementy równe sobie pozostają w niezmienionej kolejności względem siebie) – wystarczy podczas scalania najpierw prać elementy z lewej połowy. Ma złożoność pamięciową $\Theta(n)$, co potencjalnie jest problemem. W praktyce nie trzeba stosować kosztownej rekursji, bo drzewo wywołań rekurencyjnych zawsze wygląda tak samo i możemy wykonywać odpowiednie operacje iteracyjnie od dołu tego drzewa – najpierw scalamy podtablice jednoelementowe, potem dwuelementowe, potem czteroelementowe i tak dalej.

Sortowanie szybkie (quicksort) to randomizowany algorytm sortowania. Działa w następujący sposób:

1. Losujemy z sortowanej tablicy element, który nazywamy pivotem.
2. Idziemy wskaźnikiem i od lewej do prawej strony tablicy, a wskaźnikiem j od prawej do lewej. Jeśli elementy wskazywany przez i jest większy od pivota a wskazywany przez j mniejszy, to zamieniamy je ze sobą. Kończymy, gdy wskaźniki się zejdą.
3. W tej chwili elementy na lewo od pivota są nie większe od tych na prawo od pivota. Zatem możemy wywołać się rekurencyjnie na obu połówkach (pivot nie jest w żadnej).

Istnieją różne sposoby wybierania pivota. Niektóre wersje algorytmu wybierają zawsze pierwszy element tablicy albo ostatni element tablicy. W takim przypadku złożoność jest kwadratowa, bo na przykład dla posortowanych tablic jedno z podzadań jest puste. Jednak jeśli wejściowa tablica jest jednostajnie losowa (albo na wstępie ją przelosowaliśmy), to oczekiwana złożoność to $\mathcal{O}(n \log n)$ – jest to równoważne z wybieraniem pivota losowo, a ten przypadek analizowaliśmy w odpowiednim pytaniu z Metod Probabilistycznych Informatyki.

Żaden z wariantów algorytmu quicksort nie jest stabilny – elementy równe mogą być dowolnie poprzestawiane. W przedstawionym wariancie mamy pesymistyczną złożoność pamięciową $\Theta(n)$, bo taka może być głębokość drzewa rekursji. Aby temu zapobiec możemy wywołać się rekurencyjnie na krótszej połowie tablicy, a następnie zacząć wykonywać algorytm od początku na dłuższej połowie. Wtedy drzewo rekursji ma głębokość $\Theta(\log n)$.

Algorytm quicksort ma wiele wersji, które różnią się sposobem wyboru pivota i algorytmem dzielenia tablicy na pół. Przedstawiliśmy oryginalny pomysł Hoare'a, który dodatkowo zakładał, że pierwszy element jest pivotem. Sedgewick zaproponował, aby jako pivot brać medianę z pierwszego, środkowego i ostatniego elementu, co w oczekiwaniu działa trochę lepiej. Trzecim pomysłem jest metoda Lomuto, która bierze za pivota ostatni element a następnie idzie od lewej po elementach i umieszcza te nie większe od pivota na początku tablicy za pomocą odpowiednich zamian. Taka wersja jest prostsza implementacyjnie, ale daje złożoność kwadratową na przykład dla ciągu składającego się tylko z jednej wartości. Aby temu zapobiec możemy dzielić tablicę na trzy części: elementy mniejsze od pivota, większe i równe.

W zastosowaniach praktycznych quicksort jest lekko szybszy niż mergesort (w oczekiwaniu wykonuje około $1.4n \log n$ operacji), a do tego ma mniejszą złożoność pamięciową. Nie jest jednak stabilny.

II.4.3 Sortowanie topologiczne, silnie spójne składowe

Sortowanie topologiczne i znajdowanie silnie spójnych składowych grafu skierowanego metodą DFS.

Przeglądanie grafu w głąb (Depth First Search, DFS) działa w następujący sposób: wybieramy jakiś wierzchołek startowy, odwiedzamy go (czyli odnotowujemy ten fakt w odpowiedniej tablicy), a następnie wywołujemy się rekurencyjnie na jego sąsiadach (wierzchołkach, do których da się z niego dojść), pomijając wierzchołki już odwiedzone. Po wyjściu z rekursji powtarzamy tę procedurę z kolejnym nieodwiedzonym wierzchołkiem jako wierzchołkiem startowym. Przeglądniemy każdy wierzchołek i krawędź dokładnie raz, więc złożoność to $\mathcal{O}(n+m)$, gdzie n jest liczbą wierzchołków a m jest liczbą krawędzi. Czasem wygodnie jest nam rozróżniać trzy stany odwiedzenia wierzchołka. Wierzchołki białe są nieodwiedzone, wierzchołki szare to takie, w których wywołaliśmy się rekurencyjnie i ta rekursja dalej trwa, a czarne to takie, które odwiedziliśmy i których rekursja już się zakończyła. Podczas przeglądania grafu w taki sposób możemy wypisywać wierzchołki w kolejności wchodzenia do ich rekursji lub w kolejności wychodzenia z ich rekursji. Nazywamy te porządki na wierzchołkach odpowiednio porządkiem preorder i postorder.

Niech G będzie grafem skierowanym. Porządkiem topologicznym na G nazywamy taki porządek na wierzchołkach, że jeśli (u, v) jest krawędzią, to u jest przed v w tym porządku. Jeśli graf posiada cykl, to oczywiście nie istnieje dla niego porządek topologiczny. Jeśli graf jest acykliczny, to znajdujemy porządek topologiczny za pomocą algorytmu DFS: przeglądamy graf i zapisujemy porządek postorder w tablicy f ($f(u)$ to indeks u w tym porządku). Okazuje się, że odwrócenie f daje nam porządek topologiczny. Aby to zobaczyć rozważmy krawędź (u, v). W pewnym momencie DFSa badamy tę krawędź – u jest aktualnym wierzchołkiem i przeglądamy jego sąsiedztwo. Rozważamy przypadki.

- Jeśli v jest biały, to wejdziemy do niego i zanim wyjdziemy z rekursji u będziemy musieli wyjść z rekursji v, co daje nam $f(v) < f(u)$.
- Jeśli v jest szary, to mamy cykl w G – idziemy ścieżką rekursji z v do u, a następnie krawędzią (u, v). To daje sprzeczność.
- Jeśli v jest czarny, to mamy $f(v) < f(u)$, bo z rekursji v już wyszliśmy.

Przedstawiony algorytm pozwala sprawdzać, czy G jest acykliczny. Wystarczy wyznaczyć opisany porządek i sprawdzić, czy jest porządkiem topologicznym. Podobny algorytm oparty o DFS pozwala sprawdzać acykliczność grafów skierowanych.

Silnie spójną składową grafu skierowanego G nazywamy każdy maksymalny na inkluzję podgraf indukowany $G[X]$ taki, że dla każdej pary wierzchołków $u, v \in X$ istnieje ścieżka z u do v i z v do u. Graf nazywamy silnie spójnym, jeśli posiada tylko jedną silnie spójną składową. W wielu zastosowaniach wierzchołki znajdujące się w jednej silnie spójnej składowej mogą zostać uznane za tożsame. Wtedy wygodnie jest rozważać graf silnie spójnych składowych, który jest acykliczny (bo wierzchołki leżące na jednym cyklu są osiągalne od siebie).

Za pomocą algorytmu DFS znajdziemy silnie spójne składowe i każdemu wierzchołkowi przypiszemy etykietę odpowiadającą jego silnie spójnej składowej. Niech G będzie naszym grafem a G^T jego grafem

transponowanym (ten sam graf z odwróconymi krawędziami). Za pomocą DFSa obliczamy porządek postorder wierzchołków, czyli funkcję f jak w wyżej. Następnie wykonujemy DFS na G^T, wybierając wierzchołki startowe w kolejności malejących wartości $f(u)$. Wszystkie wierzchołki osiągnięte z jednego wierzchołka startowego tworzą silnie spójną składową.

Aby udowodnić ten fakt działamy indukcyjnie po wierzchołkach startowych. Zakładamy, że $C_1, \ldots, C_{k-1}$ są silnie spójnymi składowymi takimi, że C_i to dokładnie wierzchołki osiągnięte z wierzchołka startowego v_i (baza $k = 1$ jest trywialna). Rozważmy kolejny wierzchołek startowy v_k. Niech T_k będzie grafem indukowanym przez wierzchołki osiągnięte z v_k. Niech C_k będzie silnie spójną składową zawierającą v_k. Mamy $C_k \subseteq T_k$, bo silnie spójne składowe G^T są takie same jak w G. Składowe $C_1, \ldots, C_{k-1}$ mogą być osiągalne z v_k, ale są już czarne, więc T_k ich nie obejmie. Niech C_{k+1} będzie silnie spójną składową różną od $C_1, \ldots, C_k$. Zauważmy, że wartości f dla wierzchołków z C_{k+1} są mniejsze niż dla v_k. Gdyby w G istniała krawędź z C_{k+1} do C_k, to z dowolnego wierzchołka C_{k+1} można dojść do v_k, więc pierwszy DFS w pewnym wierzchołku C_{k+1} dotarłby do v_k i v_k byłby wcześniej w postorderze niż ten wierzchołek C_{k+1} – sprzeczność. Zatem w G nie ma krawędzi z C_{k+1} do C_k, a więc w G^T nie ma krawędzi z C_k do C_{k+1} i T_k nie obejmie żadnego wierzchołka z C_{k+1}. Wobec tego mamy $T_k \subseteq C_k$, co kończy dowód.

II.4.4 Statystyki pozycyjne

Algorytm „mediana median" znajdowania k-tego elementu zbioru i twierdzenie (bez dowodu), z którego wynika złożoność algorytmu.

Mamy zbiór S porównywalnych elementów o mocy n. Chcemy znaleźć k-ty najmniejszy jego element. Najprościej jest posortować S i wybrać k-ty element, co daje złożoność $\mathcal{O}(n \log n)$. Jeśli k jest bliskie 1 lub n, to można zastosować kopiec: dla k bliskiego 1 tworzymy min-kopiec na elementach S w czasie liniowym, a następnie wyciągami minimum k razy, co daje złożoność $\mathcal{O}(n + k \log n)$. Podobnie dla k bliskiego n i max-kopca.

Możemy rozwiązać ten problem za pomocą metody podobnej do quicksortu:

1. Wybieramy dowolny element a zbioru S.
2. Dzielimy S na trzy zbiory: $S_1 = \{x \in S : x < a\}$, $S_2 = \{x \in S : x = a\}$, $S_3 = \{x \in S : x > a\}$.
3. Jeśli $|S_1| \geqslant k$, to wywołujemy się rekurencyjnie na S_1.
4. Jeśli $|S_1| + |S_2| \geqslant k$, to zwracamy a.
5. W przeciwnym wypadku wywołujemy się rekurencyjnie na S_3 szukając elementu o pozycji $k - |S_1| - |S_2|$.

Pesymistycznie mamy złożoność $\Theta(n^2)$ jak w quicksorcie, ale w oczekiwaniu (dla losowej permutacji) już $\mathcal{O}(n)$.

Możemy wymusić deterministyczną złożoność liniową poprzez mądry wybór a. Robimy to w następujący sposób:

1. Dla małego S sortujemy i zwracamy wynik.
2. Dzielimy S na 5-cio elementowe podzbiory i sortujemy każdy (za pomocą stałej liczby operacji).
3. Niech Q będzie zbiorem median tych 5-cio elementowych podzbiorów. Rekurencyjnie znajdujemy medianę Q. Wybieramy ją jako a.

Jeśli $x \in S$ należy do któregoś z 5-cio elementowych podzbiorów o medianie co najmniej takiej jak a i jest w nich na pozycji 3, 4 lub 5, to $x \geqslant a$. Takie elementy stanowią co najmniej $\frac{1}{4}$ elementów S (połowa elementów w połowie podzbiorów). Zatem $|S_1| \leqslant \frac{3}{4}|S|$. Podobnie $|S_3| \leqslant \frac{3}{4}|S|$. Mamy równanie rekurencyjne $T(n) \leqslant T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) + cn$ dla pewnego c. Indukcyjnie pokazujemy, że $T(n) \leqslant 20cn = \mathcal{O}(n)$.

Ten fakt wynika też z tak zwanego twierdzenia Akry-Bazzi'ego.

Twierdzenie. Niech $a_1, \ldots, a_k \in \mathbb{R}_+$ i $b_1, \ldots, b_k \in \mathbb{R}$ będą takie, że $b_i > 1$ dla każdego i. Rozważmy rekurencję postaci $T(n) = f(n) + \sum_{i=1}^{k} a_i T\left(\frac{n}{b_i}\right)$, gdzie $f(n)$ jest określona i dodatnia dla odpowiednio dużych liczb. Niech p będzie takie, że $\sum_{i=1}^{k} \frac{a_i}{b_i^p} = 1$ (istnieje z ciągłości tego wyrażenia jako funkcji p). Wtedy zachodzi

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}} \,\mathrm{d}x\right)\right).$$

W naszym zastosowaniu mamy $\frac{1}{5} + \frac{1}{4/3} < 1$, więc $p < 1$. Mamy $\int_1^n \frac{cx}{x^{1+p}} \,\mathrm{d}x = c\frac{x^{1-p}}{1-p}\Big|_1^n = \Theta\left(n^{1-p}\right)$, więc $T(n) = \mathcal{O}(n)$.

II.5 Programowanie Obiektowe

II.5.1 Techniki obiektowe

Kompozycja, dziedziczenie i polimorfizm jako obiektowe techniki programowania. Omów ich zastosowania i podaj przykłady w językach C++ i Java.

Programowanie obiektowe (Object Oriented Programming, OOP) opiera się na modelowaniu programu za pomocą obiektów, które łączą w sobie dane oraz metody umożliwiające wykonywanie operacji na tych danych. Do najważniejszych technik obiektowych należą kompozycja, dziedziczenie i polimorfizm.

Dziedziczenie wprowadza relację pokrewieństwa na klasach. Klasa pochodna przejmuje pola i metody klasy bazowej, a następnie może je rozszerzać lub modyfikować. O obiektach klasy pochodnej myśli się jak o obiektach klasy bazowej, ale bardziej wyspecjalizowanych. Umożliwia to ponowne wykorzystywanie kodu, ułatwia modelowanie zależności między obiektami oraz pozwala tworzyć hierarchie klas, co tworzy strukturę, z której będzie korzystał polimorfizm. Warto zaznaczyć, że C++ wspiera wielodziedziczenie (jedna klasa dziedziczy po kilku klasach bazowych), natomiast Java pozwala na dziedziczenie tylko po jednej klasie (wielodziedziczenie realizuje się poprzez interfejsy).

Przykład klasy w C++: mamy w niej metodę abstrakcyjną (więc jest to klasa abstrakcyjna), która musi być zaimplementowana w każdej nieabstrakcyjnej klasie pochodnej. Umożliwia to modelowanie abstrakcyjnych pojęć, dla których wiele metod będzie działać tak samo, jednocześnie odkładając implementację bardziej konkretnych funkcjonalności do bardziej konkretnych obiektów.

```
#include <iostream>
#include <cmath>

class Shape {
public:
    virtual double area() const = 0; // metoda abstrakcyjna

    void printArea() const {
        std::cout << "Pole = " << area() << std::endl;
    }

    virtual ~Shape() = default;
};

class Circle : public Shape {
    double r;
public:
    Circle(double r) : r(r) {}

    double area() const override {
        return M_PI * r * r;
    }
};

class Rectangle : public Shape {
    double a, b;
public:
    Rectangle(double a, double b) : a(a), b(b) {}

    double area() const override {
        return a * b;
    }
};
```

W Javie poza klasami i klasami abstrakcyjnymi mamy jeszcze interfejsy, które nie mogą mieć pól innych niż statyczne. Interfejsy pozwalają nam zdefiniować pewien typ funkcjonalności, którą klasy mogą implementować.

```
interface Printable {
    void print();
}

class Printer implements Printable {
    @Override
    public void print() {
        System.out.println("Printing document");
    }
}

class PDFDocument implements Printable {
    @Override
    public void print() {
        System.out.println("Displaying PDF");
    }
}
```

Polimorfizm oznacza możliwość traktowania obiektów różnych klas w jednolity sposób poprzez wspólny interfejs lub klasę bazową. Jest on realizowany przez metody wirtualne (C++) lub przesłanianie metod (Java). Korzystając z hierarchii klas, pozwala pisać kod niezależny od konkretnego typu obiektu. W poniższych przykładach program może operować na obiektach typu `Animal`, nie wiedząc, czy są to psy, koty czy inne zwierzęta.

```
class Animal {
public:
    virtual void makeSound() {
        std::cout << "Animal sound\n";
    }

    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Woof!\n";
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Meow!\n";
    }
};
```

```
class Animal {
    public void makeSound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
```

```
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

Taki polimorfizm nazywamy polimorfizmem dynamicznym. Polega on na przesłanianiu metod i stosowania różnych implementacji w klasach pochodnych dla metod o tej samej sygnaturze. Polimorfizm statyczny polega na tworzeniu metod o tej samej nazwie, ale rozróżnianych przez argumenty jakie przyjmują. Taki mechanizm nazywamy przeciążaniem metod – ta sama nazwa metody, ale różne parametry.

Kompozycja polega na umieszczaniu obiektów różnych klas jako pól (składowych) innej klasy. Pozwala ona na budowanie złożonych obiektów z prostszych komponentów, zwiększa modularność kodu i pozwala łatwo wymieniać implementacje poszczególnych elementów (zwłaszcza, jeśli typy tych składowych są interfejsami, które mogą być implementowane przez różne klasy). Kompozycja stanowi alternatywny do dziedziczenia sposób budowania złożonych systemów. W praktyce często stosuje się zasadę *Prefer composition over inheritance*, ponieważ kompozycja nie wiąże klas tak ciasno jak dziedziczenie i pozwala na zmianę zachowania w czasie działania programu.

```
class Engine {
public:
    void start() {
        std::cout << "Engine started\n";
    }
};

class Car {
private:
    Engine engine;       // kompozycja
public:
    void startCar() {
        engine.start();
    }
};
```

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine = new Engine();

    public void startCar() {
        engine.start();
    }
}
```

Warto dodać, że w kompozycji obiekty składowe istnieją jako część większej całości, bez której nie mają one sensu. Są niszczone gdy tylko główny obiekt przestanie istnieć. Jeśli łączymy ze sobą istniejące niezależnie od siebie obiekty, to mamy do czynienia z agregacją.

II.5.2 Szablony i typy generyczne

Szablony w C++ i typy generyczne w Javie. Omów ich zastosowania, wyjaśnij różnice i podaj przykłady implementacji, które z nich korzystają.

Zarówno szablony (templates) w C++, jak i typy generyczne (generics) w Javie służą do pisania kodu niezależnego od konkretnego typu danych. Pozwalają tworzyć klasy i funkcje, które mogą działać na różnych typach bez powielania kodu. Ich najczęstsze zastosowania to tworzenie uniwersalnych struktur danych, takich jak listy, wektory, stosy, mapy (STL w C++: `vector<T>`, `map<K,V>`, Java Collections Framework: `ArrayList<E>`, `HashMap<K,V>`) oraz tworzenie algorytmów niezależnych od typu (np. sortowanie, wyszukiwanie). Dodatkowo oba mechanizmy zapewniają bezpieczeństwo typów na etapie kompilacji (nie trzeba rzutować obiektów i ryzykować błędów rzutowania w czasie uruchomienia programu).

W C++ szablony działają na zasadzie zaawansowanego systemu makr przetwarzanych przez kompilator. Dla każdej unikalnej kombinacji typów, z jakimi wywołany zostanie szablon, kompilator generuje w pamięci osobną klasę. Proces ten nazywa się konkretyzacją (instantiating). Dzięki temu nie ma narzutu na wydajność, bo kompilator optymalizuje kod pod konkretny typ. Do tego szablony mogą przyjmować wartości jako parametry (np. rozmiar tablicy: `template<typename T, int Size>`). Wadą jest wydłużony czas kompilacji oraz większy rozmiar kodu binarnego programu.

```cpp
template<typename T>
class Box {
    T value;

public:
    Box(T v) : value(v) {}

    T getValue() const {
        return value;
    }
};

Box<int> b1(10);
Box<std::string> b2("Hello");
```

W Javie typy generyczne pozwalają określić typ parametryczny klasy lub metody. Kompilator Javy sprawdza poprawność typów w czasie kompilacji, a następnie wymazuje je (type erasure). W wyjściowym kodzie parametry typów są zastępowane klasą Object (lub inną klasą w przypadku podania typów ograniczających). Dzięki temu czas kompilacji jest krótki, a w pamięci istnieje tylko jedna klasa. Wadą jest narzut wydajnościowy spowodowany ukrytym rzutowaniem. Do tego typami generycznymi nie mogą być typy prymitywne (nie można stworzyć `List<int>`, trzeba użyć klasy opakowującej `List<Integer>`). Nie można też utworzyć instancji typu generycznego (`new T()`).

```java
class Box<T> {
    private T value;

    public Box(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

Box<Integer> b1 = new Box<>(10);
Box<String> b2 = new Box<>("Hello");

class Utils {
    public static <T> T maximum(T a, T b) {        // metoda generyczna
        return a;
    }
```

```
}

public static <T extends Comparable<T>>       // ograniczenie na typ
T maximum(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
}
```

Generalnie szablony w C++ są dużo potężniejsze od typów generycznych w Javie. Umożliwiają specjalizację szablonów (stworzenie specjalnej implementacji dla wybranego, konkretnego typu) i dają możliwość wykonywania części obliczeń podczas kompilacji. Typy generyczne służą głównie zapewnieniu bezpieczeństwa typów i do wygodniejszego korzystania z biblioteki Collections.

II.5.3 Pola prywatne i chronione

Mechanizmy kontroli dostępu do pól i metod obiektów w językach C++ i Java. Omów role poszczególnych poziomów dostępu w projektowaniu i wyjaśnij różnice (pomiędzy poziomami i pomiędzy językami).

Mechanizmy kontroli dostępu określają, które części programu mogą korzystać z pól i metod danej klasy. Są one podstawowym narzędziem realizującym hermetyzację i enkapsulację, czyli ukrywania szczegółów implementacji obiektu i udostępniania jedynie dobrze zdefiniowanego interfejsu. Kontrola dostępu pozwala chronić stan obiektu przed niekontrolowaną modyfikacją, ukrywać szczegóły implementacji, oddzielać interfejs klasy od jej implementacji oraz zwiększać bezpieczeństwo i łatwość utrzymania kodu.

Projektując daną klasę chcemy odebrać innym częściom programu dostęp do pól i metod, które stanowią część szczegółów implementacyjnych, a dać dostęp do tych, które stanowią interfejs naszej klasy. Dzięki temu nikt z zewnątrz nie będzie mógł np. zmienić stanu obiektu naszej klasy w niespójny sposób. Nikt też nie będzie w swoim kodzie korzystał w metod, których działanie może zostać w przyszłości zmienione, bo są tylko częścią implementacji klasy.

W C++ mamy trzy specyfikatory dostępu: `public`, `protected` i `private`. Element publiczny jest dostępny z dowolnego miejsca programu, prywatny tylko z wnętrza klasy, a chroniony dodatkowo z wnętrza klas pochodnych (przy czym mamy dostęp tylko do pól klasy bazowej wewnątrz własnego typu, nie możemy korzystać z pól chronionych dowolnego obiektu klasy bazowej). Dla klas zdefiniowanych za pomocą słowa kluczowego `class` domyślny specyfikator dostępu to `private`, a dla `struct` jest to `public`.

Do tego przy dziedziczeniu również możemy określić specyfikator dostępu. Wtedy wszystkie pola klasy bazowej będą miały co najmniej tak restrykcyjny dostęp, jak ten specyfikator sugeruje. Domyślne dziedziczenie dla `class` jest prywatne, a dla `struct` publiczne. Istnieje mechanizm funkcji i klas zaprzyjaźnionych: możemy mieć dostęp do wszystkich pól danej klasy w pewnej funkcji lub klasie, jeśli zostanie ona określona wewnątrz klasy jako zaprzyjaźniona. Przyjaźń nie jest dziedziczona – dzieci klasy zaprzyjaźnionej nie dziedziczą prawa dostępu.

```
class Animal {
    int priv;
protected:
    int age;
public:
    int pub;

    friend void print(Animal&);
};

void print(Animal& a) {
    std::cout << a.priv;
}

class Dog : protected Animal {
```

```
public:
    void setAge(int a) {
        priv = a; // error
        age = a;
    }
};

Dog dg;
dg.age = 2; // error
dg.setAge(2);
dg.pub = 1; // error
```

W Javie również mamy specyfikatory `private`, `protected` i `public`, przy czym dodatkowo istnieje domyślny poziom dostępu (bez specyfikatora), nazywany package-private. Do takich pól ma dostęp tylko kod znajdujący się w tym samym pakiecie. Specyfikator `protected` jest rozszerzeniem dostępu package-private: umożliwia on dostęp z wnętrza tego samego pakietu i dodatkowo z klas pochodnych. Poza tym działa to tak samo jak w C++ (poza tym, że klasa-dziecko zawsze ma dostęp do pól chronionych rodzica). Nie ma specyfikatora dostępu przy dziedziczeniu ani mechanizmu klas i funkcji zaprzyjaźnionych (ich rolę częściowo spełnia dostęp pakietowy). Specyfikatory dostępu są sprawdzane nie tylko podczas kompilacji jak w C++, ale też w trakcie działania programu.

```
class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }
}
```

Jeśli mamy klasę zagnieżdżoną wewnątrz klasy, to w Javie klasa zewnętrzna i wewnętrzna mają pełen dostęp do swoich pól. Natomiast w C++ klasa zagnieżdżona ma dostęp do prywatnych elementów klasy zewnętrznej, ale w drugą stronę już nie – aby klasa zewnętrzna miała dostęp do prywatnych pól klasy wewnętrznej trzeba skorzystać z mechanizmu przyjaźni.

II.5.4 RTTI i refleksja

Omów mechanizmy RTTI w C++ i refleksji w Javie. Zaprezentuj przykłady ich zastosowań.

W programowaniu obiektowym często korzystamy z polimorfizmu – mamy zmienne o pewnym typie statycznym, które tak naprawdę są typu będącego klasą pochodną odpowiedniej klasy bazowej. Czasami program musi sprawdzić rzeczywisty typ obiektu w czasie wykonania i odpowiednio zareagować. W C++ służy do tego mechanizm RTTI (Run-Time Type Information). Działa on dla klas polimorficznych, czyli zawierających przynajmniej jedną metodę wirtualną. Informacja o typie jest pobierana z tabeli metod wirtualnych (vtable).

RTTI w C++ opiera się na stosowaniu operatorów `typeid` i `dynamic_cast<T>`. Operator `typeid` zwraca obiekt typu `std::type_info`, który zawiera zależne od implementacji informacje a typie, takie jak jego nazwa czy dane potrzebne do porównania dwóch typów ze sobą. Na obiektach typu `std::type_info` można stosować operator równości sprawdzający identyczność typów. Operator `dynamic_cast<T>` pozwala bezpiecznie rzutować wskaźniki lub referencje zgodnie z hierarchią klas. Jeśli rzutowanie wskaźnika się nie powiedzie, zwraca `nullptr`. Jeśli rzutowanie referencji się nie powiedzie, rzuca wyjątek `std::bad_cast`. Umożliwia to bezpieczne rzutowanie obiektów w hierarchii polimorficznej, gdy nie mamy pewności, z jakim obiektem podrzędnym pracujemy. Stosowanie `dynamic_cast<T>` nie zawsze wymaga

użycia RTTI, niektóre konwersje (np. z klasy pochodnej na bazową) mogą się odbyć bez pozyskiwania informacji o typie podczas wykonania programu.

```
#include <iostream>
#include <typeinfo>

class Parent {
public:
    virtual ~Parent() = default; // klasa polimorficzna (ma funkcję wirtualną)
};

class Child : public Parent {
public:
    void performChildAction() {
        std::cout << "Child-specific method\n";
    }
};

int main() {
    Parent* basePointer = new Child();

    std::cout << "Dynamic object type: " << typeid(*basePointer).name() << std
        ::endl; // nazwa należna od implementacji

    Child* childPointer = dynamic_cast<Child*>(basePointer); // bezpieczna
        konwersja w dół używająca RTTI
    if (childPointer != nullptr) {
        childPointer->performChildAction();
    } else {
        std::cout << "Downcasting failed: object is not of type Child.\n";
    }

    delete basePointer;
    return 0;
}
```

RTTI dostarcza stosunkowo niewiele informacji. Daje możliwość sprawdzania zgodności typów, ale nie pozwala np. odczytywać listy metod czy pól klasy. Refleksja w Javie to dużo potężniejszy mechanizm, który pozwala programowi nie tylko na sprawdzanie typu obiektu, ale na pełną inspekcję i modyfikację struktury klas, interfejsów, pól i metod w czasie uruchomienia, nawet jeśli były one prywatne. W Javie każda klasa ma swój obiekt-reprezentację typu `Class<?>`. Za pomocą pakietu `java.lang.reflect` programista może przeglądać wszystkie pola, metody i konstruktory klasy (w tym prywatne), tworzyć instancje klas dynamicznie (na podstawie nazwy w formacie `String`), wywoływać metody oraz zmieniać wartości pól, omijając standardowe mechanizmy kontroli dostępu. Możemy nawet tworzyć nowe klasy ładując dynamicznie bytecode (np. z jakiegoś pliku) i przekazując go do JVM'a, aby stworzył z niego klasę.

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

class Secret {
    private String secret = "value";

    private void execute() {
        System.out.println("Hidden method");
    }
}

public class Main {
    public static void main(String[] args) {
        try {
```

```
            Class<?> clazz = Class.forName("Secret"); // dynamiczne wczytanie
                klasy
            System.out.println(clazz.getName());

            Method[] methods = clazz.getMethods(); // wszystkie publiczne
                metody
            for (Method m : methods) System.out.println(m.getName());

            Field[] fields = clazz.getDeclaredFields(); // wszystkie pola
                zadeklarowane w definicji klasy, w tym niepubliczne
            for (Field f : fields) System.out.println(f.getName());

            Object ins = clazz.getDeclaredConstructor().newInstance(); //
                tworzenie obiektu

            // dostęp do prywatnego pola
            Field codeField = clazz.getDeclaredField("secret");
            codeField.setAccessible(true);
            System.out.println("Original value: " + codeField.get(ins));
            codeField.set(ins, "modified");
            System.out.println("Modified value: " + codeField.get(ins));

            // dostęp do prywatnej metody
            Method missionMethod = clazz.getDeclaredMethod("execute");
            missionMethod.setAccessible(true);
            missionMethod.invoke(ins);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Refleksja jest bardzo potężnym narzędziem, ale wymaga sporego narzutu wydajnościowego (sprawdzanie zabezpieczeń w czasie wykonania programu, brak możliwości zastosowania kompilacji JIT), którego nie ma przy RTTI (mamy tylko dodatkowy wskaźnik w vtable dla klas polimorficznych). Dynamiczne manipulowanie strukturą naszego programu nie jest dobrą praktyką programistyczną i należy się tego wystrzegać. Mimo to refleksja ma zastosowanie w frameworkach takich jak Spring, które implementują automatyczne wstrzykiwanie zależności i tworzenie obiektów na podstawie adnotacji. Stosują ją również narzędzia do testowania takie jak JUnit, który wykonuje dynamiczne wyszukiwanie i uruchamianie metod oznaczonych adnotacją `@Test`. Poniżej przykład kodu, który korzysta z adnotacji dostępnych w Springu, aby stworzyć prosty serwer webowy. Programista nie musi ręcznie przypisywać funkcji `hello` do endpointa `/hello`, pisze jedynie adnotację, a Spring korzystając z refleksji odnajduje ją.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {
    public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
    }
    @GetMapping("/hello")
    public String hello(@RequestParam(value = "name", defaultValue = "World")
        String name) {
      return String.format("Hello %s!", name);
    }
}
```

II.5.5 Biblioteki GUI

Na przykładzie obiektowego języka programowania przedstaw strukturę klas służących do obsługi GUI. Opisz technikę sterowania kierowanego zdarzeniami.

W obiektowych systemach GUI struktura klas opiera się na wzorcu projektowym Kompozyt (Composite). Pozwala on traktować pojedyncze komponenty (np. przycisk) oraz ich grupy (np. panel zawierający przyciski) w sposób jednolity. Wszystkie elementy graficzne dziedziczą po wspólnej klasie bazowej, tworząc drzewiastą hierarchię komponentów. Przedstawimy taką strukturę na podstawie biblioteki Swing, która bazuje na starszym AWT (Abstract Window Toolkit). Mamy klasę `Component`, która reprezentuje obiekt posiadający reprezentację graficzną, który można wyświetlić na ekranie i który może wchodzić w interakcję z użytkownikiem. Po niej dziedziczy klasa `Container`, która reprezentuje komponent zawierający w sobie inne komponenty. Są to bardzo ogólne klasy AWT, po których dziedziczą konkretne klasy biblioteki Swing.

`JFrame` jest klasą reprezentującą okno aplikacji. Do takiego okna można dodawać komponenty, które zwykle są typu `JComponent` (dziedziczy po `Container` z AWT i jest bazową klasą wszystkich komponentów w Swingu poza takimi jak `JFrame`). Istnieją komponenty agregujące, takie jak `JPanel`, które umożliwiają połączenie komponentów w jedną grupę i późniejsze wyświetlenie ich na ekranie w jednym miejscu (za pomocą menedżerów układu takich jak `BorderLayout`). Mamy komponenty takie jak `JScrollPane`, które opakowują inny komponent i dodają do niego jakąś funkcjonalność (w tym przypadku scrollbar). Na samym dole drzewa komponentów leżą komponenty atomowe, które są konkretnymi obiektami na ekranie, np. `JButton`, `JLabel`, `JCheckBox`.

Takie wykorzystanie dziedziczenia umożliwia ponowne wykorzystanie bardzo dużej ilości kodu (np. implementującej wyświetlanie komponentów na ekranie). Do tego dzięki polimorfizmowi zazwyczaj nie musimy przejmować się konkretnym typem danego komponentu. To umożliwia nam łatwe stworzenie opisanej struktury drzewiastej.

W tradycyjnych programach konsolowych to kod decyduje o kolejności wykonywania operacji (przepływ jest sekwencyjny). W aplikacjach GUI przepływ programu jest determinowany przez akcje użytkownika (kliknięcie myszą, wciśnięcie klawisza) lub zdarzenia systemowe. Wymaga to od nas innego podejścia do sterowania przepływem programu. Stosujemy technikę sterowania kierowanego zdarzeniami, opartą na wzorcu projektowym Observer – obiekty deklarują chęć bycia powiadomionym o danym zdarzeniu i po dostaniu o nim informacji wykonują pewną akcję.

Mechanizm sterowania kierowanego zdarzeniami opiera się na istnieniu nieskończonej pętli zdarzeń, w Swingu nazywanej Event Dispatch Thread (EDT). Odbiera ona informacje przychodzące od systemu operacyjnego (dotyczące na przykład interakcji z użytkownikiem), opakowuje je w obiekty reprezentujące konkretne zdarzenia (w Swingu `ActionEvent`) i wrzuca na kolejkę zdarzeń. Następnie pobiera zdarzenia z kolejki i powiadamia obiekty, które zadeklarowały chęć otrzymywania powiadomień o zdarzeniach tego typu. Dzieje się to za pomocą obiektów klasy `ActionListener`, które mogą zostać dodane jako nasłuchujące odpowiednich typów zdarzeń, takich jak kliknięcie w ustalony przycisk. Taka architektura pozwala na asynchroniczne reagowanie na zdarzenia przychodzące od użytkownika. Jest też łatwo rozszerzyć ją o kolejne zdarzenia, bez ingerencji w już istniejący kod.

```
1  import javax.swing.*;
2  import java.awt.event.ActionEvent;
3  import java.awt.event.ActionListener;
4  import java.awt.BorderLayout;
5
6  // kontener najwyższego poziomu - główne okno aplikacji
7  public class App extends JFrame {
8      public App() {
9          setTitle("GUI Example");
10         setSize(400, 200);
11         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         setLocationRelativeTo(null);
13
14         // tworzenie komponentów
15         JPanel mainPanel = new JPanel();
```

```
        JButton clickMeButton = new JButton("Click Me!");

        // budowanie hierarchii drzewiastej
        mainPanel.add(clickMeButton);
        add(mainPanel, BorderLayout.CENTER);

        // rejestrowanie chęci powiadomienia o zdarzeniu
        clickMeButton.addActionListener(new ActionListener() {
            // metoda wywołana asynchronicznie w momencie zdarzenia
            @Override
            public void actionPerformed(ActionEvent event) {
                JOptionPane.showMessageDialog(App.this, "Button clicked.");
            }
        });
    }

    public static void main(String[] args) {
        // bezpieczne uruchomienie aplikacji GUI w dedykowanym wątku EDT
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                App app = new App();
                app.setVisible(true);
            }
        });
    }
}
```

II.5.6 Konstrukcja i destrukcja obiektów

Porównaj mechanizmy konstrukcji i niszczenia obiektów w językach C++ i Java. Wskaż mocne i słabe punkty każdego z podejść.

Zarówno w C++, jak i w Javie konstrukcja obiektów odbywa się za pomocą konstruktorów. Funkcjonują one jednak w trochę inny sposób. W C++ mamy do dyspozycji konstruktory domyślne, parametryczne (mające jakieś argumenty), kopiujące i przenoszące. W Javie nie ma konstruktorów kopiujących i przenoszących.

W C++ obiekty standardowo umieszczane są na stosie. Pamięć na nie jest przydzielana automatycznie i bardzo szybko (wystarczy przesunąć wskaźnik stosu). W momencie zakończenia wykonywania danego bloku kodu następuje destrukcja obiektów w nim zadeklarowanych – wykonuje się destruktor. Jest to funkcja, której zadaniem jest zwolnienie zasobów zajętych przez destruowany obiekt.

```
class Point {
    int x, y;

public:
    Point(int a, int b) : x(a), y(b) {}
};

int main() {
    {
        Point p(1,2);
    } // następuje destrukcja
}
```

W C++ możemy alokować pamięć na konstruowane obiektu w sposób dynamiczny, na stercie. Wtedy tworzony obiekt jest tam umieszczany i nie przestaje istnieć do momentu ręcznego zakończenia jego życia

i zwolnienia pamięci. W szczególności obiekty mogą alokować pamięć dynamicznie w swoich konstruktorach. W takim przypadku należy pamiętać o zwolnieniu tej pamięci w destruktorze.

```
class Point {
    int *c;

public:
    Point(int a, int b) {
        c = new int[2];
        c[0] = a;
        c[1] = b;
    }

    ~Point() {
        delete[] c;
    }
};

int main() {
    Point *p;
    {
        p = new Point(1,2);
    } // obiekt przeżywa
    delete p;
}
```

Kluczową techniką w C++ jest RAII (Resource Acquisition Is Initialization). Polega ona na tym, że zasoby (pamięć, otwarte pliki, połączenia z bazą danych) są blokowane w konstruktorze obiektu, a zwalniane automatycznie w jego destruktorze. Dzięki temu programista korzystający z obiektów nie musi pamiętać o zwalnianiu zasobów (o ile sam nie alokuje czegoś dynamicznie).

```
class File {
    FILE* f;
public:
    File(const char* name) {
        f = fopen(name, "r");
    }

    ~File() {
        fclose(f);
    }
};
```

W nowszych wersjach C++ powstały tak zwane inteligentne wskaźniki, które umożliwiają korzystanie z RAII w przypadku pamięci alokowanej na stercie. `std::unique_prt` jest obiektem reprezentującym jedyny wskaźnik na dane na stercie, które zostaną zwolnione w momencie jego zniszczenia. Taki obiekt może zostać przeniesiony, ale nie skopiowany. Z kolei `std::shared_prt` implementuje zliczanie istniejących referencji do danych. Obiekt na stercie żyje tak długo, jak długo istnieje choć jeden `shared_ptr` na niego wskazujący. Poniżej zastosowanie zakładające, że `Point` ma zaimplementowany konstruktor przenoszący.

```
{
    unique_ptr<Point> p;
    {
        unique_ptr<Point> tmp = make_unique<Point>(1,2);
        p = std::move(tmp);
    } // dane nie znikają
} // dopiero teraz
```

W Javie wszystkie obiekty konstruowane są na stercie. Mimo to programista nie musi przejmować się zwalnianiem pamięci po nich. W momencie utraty wszystkich referencji do danego obiektu staje się on kandydatem do usunięcia przez tak zwany Garbage Collector – automatyczny proces działający w

tle maszyny wirtualnej (JVM), który co jakiś czas skanuje stertę i usuwa nieużywane obiekty. Moment usunięcia nie jest określony przez programistę. Można jedynie zasugerować maszynie wirtualnej, aby wykonała proces usuwania niepotrzebnych obiektów, ale nie jesteśmy w stanie tego wymusić. Historycznie istniała metoda `protected void finalize()` wykonywana przez Garbage Collector przed usunięciem pamięci obiektu, ale ze względu na niedeterministyczny charakter procesu niszczenia obiektów nigdy nie można było mieć pewności, że się wykona. Dlatego została zdeprecjonowana.

```
1  class Point {
2      private int x, y;
3
4      public Point(int x, int y) {
5          this.x = x;
6          this.y = y;
7      }
8  }
9
10 Point p = new Point(1,2);
11 p = null;
12 System.gc(); // to nie musi nic zrobić
```

Z powodu braku destruktorów nie funkcjonuje mechanizm RAII i zdobyte zasoby muszą zostać zwolnione ręcznie lub za pomocą bloku try-with-resources.

```
1 FileInputStream file = new FileInputStream("a.txt");
2 file.read();
3 file.close();
4
5
6 try (FileInputStream file = new FileInputStream("a.txt")) {
7     file.read();
8 }
```

Podsumowując, C++ daje nam większą swobodę w sposobie zarządzania pamięcią, udostępnia zarówno szybkie tworzenie obiektów na stosie, jak i wolniejszą ale bardziej trwałą stertę. Mechanizm destruktorów powoduje, że moment zniszczenia obiektu jest pewny, co umożliwia funkcjonowanie bardzo wygodnego RAII. Ten system jest bardzo wydajny, bo nie potrzebuje Garbage Collectora, a do tego daje programiście dużą kontrolę nad pamięcią. Jego wadą jest spore ryzyko błędów – łatwo się pomylić w zarządzaniu pamięcią, co prowadzi do wycieków pamięci lub podwójnego jej zwalniania. Takie błędy są bardzo trudne do wykrycia i naprawienia. Z kolei Java eliminuje potrzebę ręcznego zarządzania pamięcią – wszystkim automatycznie zajmuje się Garbage Collector. Minusem jest brak deterministycznego niszczenia, co uniemożliwia RAII oraz większy narzut związany z działaniem Garbage Collectora w maszynie wirtualnej.

II.5.7 Standardowe biblioteki kontenerów

Przedstaw standardowe biblioteki kontenerów dostępnych w językach C++ i Java. Opisz zastosowania poszczególnych klas.

Kontenery są strukturami danych służącymi do przechowywania i organizowania kolekcji obiektów. Zarówno C++, jak i Java dostarczają bogate biblioteki standardowe zawierające gotowe implementacje najczęściej używanych struktur danych. W C++ podstawową biblioteką jest STL (Standard Template Library). W Javie odpowiednikiem jest Java Collections Framework (JCF).

Kontenery sekwencyjne (tablice i listy) służą do przechowywania elementów w określonej liniowej kolejności. Najczęściej stosowana jest dynamiczna tablica, czyli `std::vector` w C++ i `ArrayList` w Javie. Jest to najbardziej uniwersalny kontener, idealny, gdy potrzebujemy bardzo szybkiego dostępu do elementów po indeksie (czas stały) oraz gdy elementy są dopisywane głównie na końcu. Warto wiedzieć, że `std::vector` gwarantuje ciągłość pamięci w RAMie, dzięki czemu cache zachowuje się dobrze. `ArrayList` nie daje takich gwarancji – przechowuje tablicę referencji do obiektów rozrzuconych w pamięci. Innym

kontenerem sekwencyjnym jest lista dwukierunkowa, czyli `std::list` w C++ i `LinkedList` w Javie. Stosowana, gdy zachodzi potrzeba częstego wstawiania lub usuwania elementów ze środka, początku lub końca kolekcji (czas stały przy znanym iteratorze). Jej wadą jest brak dostępu do konkretnego elementu po indeksie, odszukanie elementu wymaga przejścia całej listy.

Kontenery asocjacyjne i zbiory służą do przechowywania unikalnych elementów (zbiory) lub par klucz-wartość (mapy), gdzie klucz pozwala na szybkie odnalezienie wartości. Mogą to być struktury oparte na drzewach BST lub tablicach haszujących. Te pierwsze to `std::set` i `std::map` w C++ oraz `TreeSet` i `TreeMap` w Javie. Mają zastosowanie, gdy chcemy mieć dostęp do elementów posortowanych (i na przykład wykonywać na nich wyszukiwanie binarne) lub chcemy znaleźć wszystkie elementy mniejsze od ustalonego. Są one zaimplementowane za pomocą drzew czerwono-czarnych, zatem czas wyszukiwania, wstawiania i usuwania jest logarytmiczny. Struktury oparte na tablicach haszujących to odpowiednio `std::unordered_set` i `std::unordered_map` w C++ oraz `HashSet` i `HashMap` w Javie. Mają zastosowanie jako szybkie słowniki lub gdy chcemy sprawdzić unikalność jakiegoś elementu w rozważanym zbiorze. Najbardziej wydajne, gdy nie interesuje nas kolejność elementów. Są to struktury randomizowane, których oczekiwany czas operacji jest stały.

Dodatkowo istnieją tak zwane adaptery kontenerów, które ograniczają interfejsy zwykłych kontenerów, aby wymusić konkretny sposób dostępu. Są to stosy, kolejki i kolejki dwustronne, w C++ odpowiednio `std::stack`, `std::queue` i `std::deque`. W Javie kolejka i kolejka dwustronna są interfejsami (`Queue` i `Deque`) implementowanymi na przykład przez `ArrayDeque`. Istnieje klasa `Stack`, ale według dokumentacji lepiej jest używać `ArrayDeque` jako stosu (jest to szybsze). Istnieją też kolejki priorytetowe, odpowiednio `std::priority_queue` i `PriorityQueue` implementowane za pomocą kopca.

Mimo zbieżności struktur, obie biblioteki opierają się na innych fundamentach. W C++ kontenery są szablonami. Istnieją osobne od kontenerów biblioteki iteratorów i algorytmów. Iteratory dzielą się na kilka typów: input iterators oraz output iterators (do czytania i pisania), forward iterators (przechodzenie po elementach w jednym kierunku, jak w `std::forward_list`), bidirectional iterators (przechodzenie w obu kierunkach, jak w `std::set`), random access iterators (dostęp indeksowy, jak w `std::vector`). Istnieją zaimplementowane w STLu algorytmy, które przyjmują odpowiednie iteratory i wykonują operacje na przedziale przez nie zdefiniowanym, jak na przykład `std::sort`, `std::reverse`, `std::lower_bound`, które przyjmują odpowiednio random access iterators, bidirectional iterators i forward iterators.

W Javie architektura oparta jest na typach generycznych i hierarchii interfejsów (np. `List`, `Set`, `Map`) implementowanych przez konkretne struktury. Ogólnie wszystkie kontenery implementują interfejs `Collection`. Iteratory działają podobnie, jako intefejsy. Istnieje interfejs `Iterator` definiujący podstawową funkcjonalność przechodzenia do przodu po elementach kolekcji. Rozszerza go interfejs `ListIterator`, który dodatkowo umożliwia przechodzenie w tył. Intefejs `Iterator` ma metodę `remove`, a `ListIterator` dodatkowo `set` i `add`, które umożliwiają modyfikowanie kolekcji, po której iterator się iteruje. Zależnie od implementacji mogą one jednak nie istnieć, w takim przypadku rzucają `UnsupportedOperationException`. Konkretne implementacje kontenerów udostępniają swoje implementacje iteratorów.

W C++ występuje mechanizm unieważniania iteratorów (iterator invalidation): jeśli kontener zostanie zmodyfikowany (inaczej niż za pośrednictwem iteratora), to dalsze korzystanie z wcześniej utworzonego iteratora jest Undefined Behaviour. W Javie z kolei działa mechanizm Fail-Fast: jeśli kolekcja zostanie zmodyfikowana w trakcie iteracji za pomocą metod samej kolekcji, iterator rzuci `ConcurrentModificationException` przy próbie skorzystania z niego. W C++ kontenery przechowują kopie obiektów, w Javie kontenery przechowują wyłącznie referencje do obiektów alokowanych na stercie

W C++ istnieją tablice o stałym rozmiarze `std::array<T,size>` oraz wspomniane wcześniej jednokierunkowe listy `std::forward_list`, których nie ma w Javie. W Javie z istnieją `LinkedHashMap` i `LinkedHashSet`, które łączą tablice haszującą z listą dwukierunkową i zapewniają stały czas operacji, zachowując jednocześnie kolejność elementów.

W Javie istnieją do tego współbieżne kontenery, które umożliwiają wielowątkowy dostęp do danych i ich modyfikację. Są to na przykład `ConcurrentHashMap`, `ConcurentSkipListMap` (mapa zaimplementowana za pomocą struktury Skip List) czy `ArrayBlockingQueue` (kolejna z możliwością blokowania się na pustej lub pełnej kolejce). W C++ nie ma odpowiedników takich struktur.

III
Przedmioty techniczne

III.1 Inżynieria Danych

Podstawowe definicje

SZBD – System Zarządzania Bazą Danych.

Superklucz – zbiór atrybutów relacji, które jednoznacznie wyznaczają pozostałe atrybuty relacji. Minimalne na zawieranie superklucze nazywamy kluczami. Kluczem głównym relacji nazywamy pewien jej wybrany, ustalony klucz.

Atrybuty podstawowe – atrybuty relacji, które należą do pewnego jej klucza. Pozostałe atrybuty to atrybuty wtórne.

Atrybuty złożone – atrybuty relacji składające się z wielu wartości, na przykład zbiory, tabele. Atrybuty, które nie są złożone nazywamy atomowymi.

Relacje zagnieżdzone – to takie, w których mogą występować atrybuty złożone.

III.1.1 Normalizacja

Na czym polega normalizacja bazy danych i w jakim celu się ją stosuje?

Schemat bazy danych to struktura całej bazy danych (lub jej części), definiująca relacje (tabele), ich atrybuty (kolumny), typy danych, klucze główne i obce oraz ograniczenia integralnościowe.

Redundancja to wielokrotne, niepotrzebne przechowywanie tej samej informacji w bazie danych. Anomalia modyfikacji to błędy, niespójności lub ograniczenia pojawiające się podczas operacji na bazie danych, które są wynikiem tego, że występuje redundancja.

Normalizacja to technika projektowania bazy danych mająca na celu stworzenie schematów relacji, które nie posiadają niepożądanych cech. Celem normalizacji jest ograniczenie redundancji.

Jak przebiega?

- Na podstawie zależności funkcyjnych między atrybutami identyfikujemy te spośród schematów relacyjnych, które obarczone są ryzykiem redundancji i anomalii modyfikacji.
- Po tym jak wykryjemy wszystkie występujące zależności funkcyjne następuje **dekompozycja** schematów na mniejsze (*minimalizują niebezpieczeństwo rozspójnienia danych)*.
- Przeprowadzamy testy, które wykryją, czy nie naruszyliśmy zasad postaci normalnych.

Pierwsza postać normalna (1NF) – wartości atrybutów są atomowe, nie ma powtórzeń. W miejsce relacji zagnieżdżonych (schematów z atrybutami wielowartościowymi) tworzymy mniejsze relacje.

Dekompozycja: Schemat $R(X, \{Y\})$ z atrybutem wielowartościowym (lub relacją zagnieżdżoną) $\{Y\}$ rozkładamy na relację $R_1(X)$ oraz nową relację $R_2(C, Y)$, gdzie C jest kluczem głównym R. Na przykład z

studenci(nr_indeksu, nazwisko, {egzaminy(kurs, ocena)})

robimy

studenci(nr_indeksu, nazwisko), egzaminy(nr_indeksu, kurs, ocena).

Definicja. Zależność funkcyjna ($X \to Y$) zachodzi w schemacie relacji, jeżeli dla każdej dopuszczalnej instancji tej relacji dwie krotki mające równe wartości na atrybutach ze zbioru X muszą mieć również równe wartości na atrybutach ze zbioru Y. Zależność funkcyjna $X \to Y$ nazywamy pełną, gdy Y nie jest funkcyjnie zależny od żadnego właściwego podzbioru X. W przeciwnym wypadku zależność jest częściowa. Zależność funkcyjna $X \to Y$ jest nietrywialna, gdy Y nie jest podzbiorem X.

Druga postać normalna (2NF) – relacja jest w 1NF oraz żaden atrybut wtórny nie jest częściowo funkcyjnie zależny od jakiegokolwiek klucza relacji.

Dekompozycja: Jeżeli pełna zależność funkcyjna $X \to Y$ narusza warunki 2NF, to relację $R(X, Y, Z)$ rozkładamy na dwie relacje: $R_1(X, Y)$ oraz $R_2(X, Z)$. Na przykład w

egzaminy(nr_indeksu, nazwisko, przedmiot, ocena)

kluczem jest numer indeksu i przedmiot, ale nazwisko wynika z numeru indeksu. Rozbijamy więc na

studenci(nr_indeksu, nazwisko), egzaminy(nr_indeksu, przedmiot, ocena).

Trzecia postać normalna (3NF) – relacja jest w 1NF oraz dla dowolnej nietrywialnej zależności funkcyjnej $X \to A$ zachodzi jeden z warunków: X jest superkluczem lub A jest atrybutem podstawowym.

Dekompozycja: Jeżeli nietrywialna zależność $X \to A$ narusza warunki 3NF (X nie jest superkluczem), to relację R o zbiorze wszystkich atrybutów U rozkładamy na dwie mniejsze relacje: $R_1(X, A)$ (gdzie X staje się kluczem) oraz $R_2(U \setminus \{A\})$. Na przykład w

pracownicy(imię, nazwisko, instytut, adres)

instytut nie jest superkluczem, a wyznacza adres. Rozbijamy więc na

pracownicy(imię, nazwisko, instytut), instytuty(instytut, adres).

Postać normalna klucza elementarnego (EKNF) – relacja jest w 1NF oraz dla dowolnej nietrywialnej zależności funkcyjnej $X \to A$ zachodzi jeden z warunków: X jest superkluczem lub A jest atrybutem należącym do klucza głównego (zwanego też podstawowym, elementarnym).

Dekompozycja: Jeżeli nietrywialna zależność $X \to A$ narusza warunki EKNF (X nie jest superkluczem), to relację R o zbiorze wszystkich atrybutów U rozkładamy na dwie mniejsze relacje: $R_1(X, A)$ (gdzie X staje się kluczem) oraz $R_2(U \setminus \{A\})$. Na przykład dla

pracownicy(nazwisko, wydział, dziekan)

kluczem może być nazwisko i dziekan lub nazwisko i wydział. Dziekan i wydział nawzajem od siebie zależą. Przy każdym wyborze klucza głównego jedna z tych zależności narusza EKNF. Rozbijamy na

pracownicy(nazwisko, wydział), wydziały(wydział, dziekan).

Postać normalna Boyce'a-Codda (BCNF) – relacja jest w 1NF oraz dla każdej nietrywialnej zależności funkcyjnej $X \to A$ zbiór X jest superkluczem tej relacji.

Dekompozycja: Jeżeli nietrywialna zależność $X \to A$ narusza warunki BCNF (X nie jest superkluczem), relację R o zbiorze wszystkich atrybutów U rozkładamy na dwie mniejsze relacje: $R_1(X, A)$ (gdzie X staje się kluczem) oraz $R_2(U \setminus \{A\})$. Rozważmy relację

pracownicy(nazwisko, uniwersytet, telefon).

Z nazwiska i uniwersytetu wynika telefon, a z telefonu uniwersytet. Jeśli przyjmiemy nazwisko i uniwersytet za klucz główny, to relacja jest w EKNF. Nie jest jednak w BCNF. Musimy rozbić

uniwersytety(telefon, uniwersytet), pracownicy(nazwisko, telefon).

Definicja. Niech X, Y, Z będą zbiorami atrybutów, które razem zawierają wszystkie atrybuty relacji R. Niech Z będzie rozłączne z $X \cup Y$. Zależność wielowartościowa ($X \twoheadrightarrow Y$) zachodzi w schemacie relacji R, gdy dla dowolnych krotek t_1, t_2 takich, że $t_1[X] = t_2[X]$ istnieje krotka t_3 taka, że $t_1[X, Y] = t_3[X, Y]$ oraz $t_2[Z] = t_3[Z]$. Zależność wielowartościowa jest nietrywialna, gdy Y nie jest podzbiorem X i Z jest niepusty.

Intuicyjnie zależność wielowartościowa znaczy, że przy ustalonym X wartości Y i Z występują we wszystkich możliwych kombinacjach. Dlatego należy rozdzielić je od siebie – gdy będą w różnych tabelach dalej będziemy wiedzieć, że należy je połączyć „każdy z każdym".

Przykład. W relacji studenci(id, kurs, hobby) mamy iloczyn kartezjański wszystkich kursów i hobby danego studenta. Wtedy id $\twoheadrightarrow$ kurs i id $\twoheadrightarrow$ hobby, bo jeśli mamy dwa wpisy dla jednego studenta, to musi istnieć wpis z takim kursem i hobby, jak odpowiednio dla pierwszego i drugiego wpisu.

Propozycja. Każda zależność funkcyjna jest zależnością wielowartościową.

Dowód. Niech $X \to Y$ będzie zależnością funkcyjną. Niech Z będzie zbiorem atrybutów nie występujących w $X \cup Y$. Jeśli mamy krotki t_1, t_2 takie, że $t_1[X] = t_2[X]$, to $t_1[Y] = t_2[Y]$. Zatem t_2 jest krotką, której istnienia żąda definicja zależności wielowartościowej. □

Czwarta postać normalna (4NF) – relacja jest w 1NF oraz dla każdej nietrywialnej zależności wielowartościowej $X \twoheadrightarrow Y$ zbiór X jest superkluczem tej relacji.

Dekompozycja: Jeżeli nietrywialna zależność wielowartościowa $X \twoheadrightarrow Y$ narusza 4NF, to oznaczamy $Z = U \setminus (X \cup Y))$ i dekomponujemy relację na dwie niezależne tabele: $R_1(X, Y)$ oraz $R_2(X, Z)$. Wspomnianą wyżej relację

studenci(id, kurs, hobby)

rozbijamy na

studenci(id, kurs), studenci_hobby(id, hobby).

Hierarchia postaci normalnych: 1NF $\subsetneq$ 2NF $\subsetneq$ 3NF $\subsetneq$ EKNF $\subsetneq$ BCNF $\subsetneq$ 4NF.

Normalizacja baz danych prowadzi do:

- zmniejszenia ilości danych bez utraty wiedzy (pozbycie się redundancji),
- uniknięcia problemów anomalii dodawania, aktualizacji i usuwania (nie trzymamy tej samej wiadomości w różnych miejscach),
- konieczności łączenia tabel (czyli spowolnienia wykonywania pewnych zapytań),
- umożliwienia tworzenia większej liczby indeksów, (czyli przyspieszenia wykonywania niektórych zapytań),
- bardziej efektywnego przetwarzania i składowania tabel na dysku,
- szybszego zarządzania transakcjami.

III.1.2 Trwałość danych

> Omów mechanizmy zapobiegające utracie danych w przypadku wystąpienia awarii.

Baza danych przechowuje swoje dane głównie w pamięci nieulotnej (np. na dysku). Taka pamięć jest jednak wolniejsza od pamięci RAM. Dlatego wszystkie operacje są wykonywane w oparciu o pamięć podręczną. Dane są modyfikowane w buforze w pamięci RAM, a potem dopiero przerzucane na dysk.

SZBD musi pilnować, aby zmiany dokonane przez zatwierdzoną transakcję były trwałe i nie zniknęły w razie awarii, natomiast zmiany transakcji wycofanej lub przerwanej przez awarię zostały cofnięte. Sprawa jest o tyle skomplikowana, że część zmian mogła istnieć jedynie w buforze i nie została zapisana, a część jest już na dysku.

Typy awarii

- błędy podczas wykonywania transakcji: błędy logiczne *(np. naruszenie ograniczeń integralnościowych)* oraz błędy systemowe *(np. zakleszczenia)*,
- awarie systemu: awaria sprzętu *(np. wyłączenie zasilania)* oraz błąd w oprogramowaniu,
- uszkodzenia dysku.

System bazodanowy powinien odtworzyć bazę po zaistnieniu awarii pierwszych dwóch typów. Z trzecim rodzajem awarii można poradzić sobie przez replikację bazy i rozproszenie danych.

W przypadku awarii przeprowadzane są dwa rodzaje operacji.

UNDO – wycofuje z dysku „brudne" zmiany, które system zdążył tam wgrać, ale ich transakcja nigdy nie została ukończona/zatwierdzona – zapewnia atomowość.

REDO – odtwarza zmiany wykonane przez zatwierdzone transakcje, które nie zdążyły zapisać się na dysku twardym przed awarią – zapewnia trwałości.

Strategie SZBD

Czy SZBD pozwala niezatwierdzonej transakcji nadpisać ostatnio zmienioną wartość w pamięci trwałej?

- STEAL: tak
- NO-STEAL: nie

Czy SZBD wymaga, aby wszystkie modyfikacje wykonane przez transakcję były odzwierciedlone w pamięci trwałej zanim transakcja może zostać uznana za zatwierdzoną?

- FORCE: tak
- NO-FORCE: nie

Stosując strategię NO-STEAL + FORCE możemy uniknąć wykonywania sekwencji UNDO (na dysku nie ma zmian do wycofania) i REDO (zatwierdzone transakcje musiały zapisać zmiany). Jest to jednak nieefektywne i może prowadzić do samozakleszczenia transakcji, gdy zabraknie miejsca w buforze. Generalnie chcemy, aby SZBD mógł zapisywać dane na dysk, gdy brakuje mu miejsca w RAMie (czyli STEAL) oraz nie musiał zapisywać danych zakończonej transakcji, aby ją zatwierdzić (czyli NO-FORCE), bo wymaga to częstego pisania na dysk, a to jest wolne. Do tego współczesne bazy danych blokują pojedyncze wiersze (rekordy), a nie całe strony pamięci. Jeśli jedna strona zawiera rekord zmodyfikowany przez Transakcję A (zatwierdzoną) i rekord zmodyfikowany przez Transakcję B (niezatwierdzoną), to przy strategii NO-STEAL + FORCE nie da się tej strony ani poprawnie zapisać (bo B jeszcze trwa), ani przetrzymać w RAM-ie (bo A chce zrobić Commit). Dlatego podstawową kombinacją strategii zarządzania buforem danych jest kombinacja STEAL + NO-FORCE. Musimy przez to implementować algorytmy radzące sobie z awariami, ale bardzo dużo zyskujemy przy normalnej pracy systemu.

Protokół WAL (Write-Ahead Log) wykorzystuje strategię STEAL + NO-FORCE, zapewniając atomowość i trwałość bazy. Poza plikami z danymi przechowywać będziemy plik z logami (dziennik), w którym zapisywane będą wszystkie zmiany dokonywane przez transakcje. Taki plik znajduje się poza pamięcią ulotną i zawiera wystarczające informacje, aby wykonać odpowiednie operacji UNDO i REDO. Rozpoczęcie i zatwierdzenie każdej transakcji jest odnotowywane w odpowiednim wpisie. Zanim transakcja zostanie uznana za zatwierdzoną, wszystkie informacje o niej muszą się znaleźć w logu zapisanym na dysku.

Checkpoints

- Służą do tego, aby logi nie rosły w nieskończoność i aby skrócić czas odzyskiwania bazy.
- Gdy system tworzy checkpoint, zrzuca na dysk wszystkie dotychczasowe wpisy z loga oraz rzeczywiste zmodyfikowane strony danych.
- W przypadku awarii system może całkowicie zignorować transakcje, które zostały zatwierdzone przed ostatnim checkpointem. System skupia się tylko na REDO i UNDO transakcji po zapisie checkpointu.

Algorytm ARIES *(Algorithm for Recovery and Isolation Exploiting Semantics)*

Popularna metoda odtwarzania bazy korzystająca z protokołu WAL, gdzie każdy wpis w logu posiada swój numer LSN (Log Sequence Number). Do tego trzymamy tabelę transakcji (gdzie są identyfikatory niezatwierdzonych transakcji i numery LSN ich ostatnich zmian) oraz tabelę brudnych stron (identyfikatory brudnych stron i numery LSN pierwszych wpisów, które je brudzą). Algorytm działa w trzech głównych fazach:

- **Analiza:** System czyta dziennik, by odtworzyć stan z momentu tuż przed awarią. Przeszukuje logi pod kątem niezatwierdzonych transakcji oraz „brudnych stron”, na których wystąpiły modyfikacje.
- **REDO:** Algorytm odtwarza historię – aplikuje ponownie wszystkie zmiany począwszy od najstarszej nieutrwalonej modyfikacji aż do momentu awarii.

- **UNDO:** System czyta dziennik od końca i wycofuje wszelkie operacje wykonane przez transakcje, które w chwili awarii nie były jeszcze zatwierdzone (zapisuje przy tym nowe logi, aby nie zapętlić się w razie wystąpienia kolejnej awarii).

III.1.3 Optymalizacja zapytań

Przedstaw podstawowe techniki stosowane w procesie optymalizacji zapytań w bazach danych.

Proces optymalizacji polega na przekształceniu zapytania SQL (wskazującego *co* pobrać) w optymalny plan wykonania (określający *jak* pobrać dane najniższym kosztem). Zapytanie zapisane przez użytkownika może zostać wykonane na wiele równoważnych sposobów. Zadaniem optymalizatora zapytań jest wybór najkorzystniejszego planu wykonania.

Zapytanie SQL ⟶ Algebra relacyjna ⟶ Dostosowane wyrażenie ⟶ Plan fizyczny

Zaczynamy od zapytania SQL, które przekształcamy do wyrażenia w algebrze relacyjnej. Następnie korzystamy z praw algebry relacyjnej i przekształcamy zapytanie bez zmiany jego wyniku:

- *Pchanie selekcji w dół (Predicate Pushdown):* Filtrowanie wierszy (`WHERE`) na jak najwcześniejszym etapie, przed kosztownymi złączeniami tabel.
- *Pchanie rzutowania w dół (Projection Pushdown):* Wczesne odrzucanie kolumn, których nie ma w ostatecznym wyniku (`SELECT`), by nie obciążać pamięci.
- *Spłaszczanie podzapytań (Subquery Unnesting):* Przekształcanie mało wydajnych podzapytań w standardowe operacje złączenia (`JOIN`).
- *Zmiana kolejności operacji:* Wykorzystanie przemienności, łączności i rozdzielności działań, aby wybrać najtańszą kolejność (np. kolejność wykonywania JOINów).
- *Eliminacja redundantnych warunków:* Usuwanie tautologii (np. `WHERE 1=1`) lub sprzeczności (np. `WHERE ID = 5 AND ID = 6`).

Następnie dokonujemy optymalizacji fizycznej – decydujemy, jak dane zostaną pobrane z dysku/pamięci i jakie konkretne algorytmy zostaną użyte.

- Możemy wykorzystywać indeksy, aby szybko odnajdywać rekordy o danej wartości danego atrybutu. Dzięki temu zamiast liniowego przejścia po tabeli mamy logarytmiczną operację na strukturze indeksu (na przykład B-drzewie).
- Mamy różne strategie wykonywania złączeń (Join Strategies), na przykład Nested Loop Join (podwójna pętla przeglądająca tabele), Index Nested Loop Join (pętla, w której rekordy drugiej tabeli są znajdowane przez indeks), Sort-Merge Join (posortowanie obu tabel i przejście przez nie), Hash Join (budowa tablicy haszującej dla mniejszej tabeli i przeszukiwanie przez drugą relację).
- Do grupowania i agregacji (klauzula `GROUP BY`) wykorzystujemy indeksy lub agregację haszującą (za pomocą tablic haszujących).
- Do sortowania możemy wykorzystać indeksy, możemy wyeliminować z zapytania redundantne sortowania. Sortowanie dużych zbiorów danych możemy realizować za pomocą zewnętrznego sortowania przez scalanie – sortujemy małe zbiory danych (mieszczące się w RAMie), a następnie scalamy je. Czasem stosujemy sortowanie nawet, gdy użytkownik nie prosi o to wprost, np. przy usuwaniu duplikatów, złączaniu albo agregacji.

Nowoczesne systemy stosują optymalizację kosztową (Cost Based Optimization, CBO). Optymalizator generuje wiele potencjalnych planów wykonania i wybiera ten o najniższym oszacowanym koszcie (I/O, procesor). Szacuje ten koszt na podstawie statystyk tabel – analizuje zebrane metryki (np. rozkład wartości, wielkość tabel, liczba różnych wartości), by oszacować ilość przetwarzanych wierszy. Może na przykład zdecydować, czy bardziej opłaca się zastosować indeks, czy przeszukać całą tabelę (np. gdy warunek wyszukiwania jest spełniony przez większość rekordów i ostatecznie i tak musimy przejść przez większość tabeli).

Poza tym stosowana jest optymalizacja heurystyczna wykorzystująca proste reguły, które zazwyczaj działają:

- wykonuj selekcję jak najwcześniej,
- wykonuj projekcję jak najwcześniej,
- unikaj dużych wyników pośrednich,
- najpierw wykonuj najbardziej selektywne złączenia,
- wykorzystuj dostępne indeksy.

III.1.4 Indeksy

> Czym są indeksy w bazach danych? Kiedy warto je tworzyć? Jakie struktury danych są najczęściej wykorzystywane jako indeksy?

Indeks to pomocnicza struktura danych przechowująca **klucze wyszukiwania** (atrybuty, po których sortujemy; nie muszą być kluczami relacji) oraz **adres fizyczny** krotki.

- Indeksy muszą zapewniać efektywne wyszukiwanie, dodawanie i usuwanie elementów.
- SZBD powinien potrafić określić, który z istniejących indeksów powinien być wykorzystany przy wykonywaniu danego zapytania.
- Domyślnie indeksy są tworzone automatycznie dla klucza głównego oraz atrybutów unikalnych.
- Więcej indeksów oznacza przyspieszenie wyszukiwania, ale zwiększenie zużycia pamięci i wydłużenie czasu aktualizacji danych. Należy tworzyć je z rozwagą.
- Mogą być założone nie tylko na wartościach atrybutów, ale również na wartościach funkcji obliczanych na odpowiedniej kolumnie.

Zatem **kiedy warto tworzyć indeksy?** Gdy korzyści z szybszego wyszukiwania przewyższają koszty związane z ich wolniejszą modyfikacją oraz zużyciem miejsca na dysku. A dokładniej

- dla kolumn na których często używamy `WHERE`.
- dla kolumn i kluczy obcych używanych w `JOIN`.
- dla kolumn które często sortujemy lub grupujemy (`ORDER BY/GROUP BY`).
- dla kolumn o dużej liczbie unikalnych wartości w bardzo dużych tabelach.

A **jakie struktury danych są najczęściej wykorzystywane jako indeksy?**

$B+$**-drzewo** – to zbalansowana, ukorzeniona i M-arna struktura drzewiasta, która przechowuje dane posortowane według pewnego klucza. Umożliwia wyszukiwanie, wstawianie i usuwanie elementów oraz dostęp sekwencyjny w czasie $\mathcal{O}(\log N)$.

Budowa i właściwości:

- **Węzły wewnętrzne:** przechowują tablicę z parami `(wskaźnik na podrzewo, klucz)`. Każdy węzeł o k kluczach posiada $k+1$ potomków. Mają rolę „nawigacyjną".
- **Liście:** przechowują właściwe dane w postaci par `(wartość (często adres na dysku), klucz)`.
- Każdy wierzchołek (poza korzeniem) musi być wypełniony co najmniej w połowie (od $M/2-1$ do $M-1$ kluczy). Przepełnione liście/węzły są dzielone, a te ze zbyt małą liczbą elementów – łączone.

Najszybciej możemy zbudować $B+$-drzewo sortując po kluczu, a potem budując indeks od dołu – metoda *bulk loading*.

Wyszukiwanie w $B+$-drzewie implementujemy jak w drzewie BST, przy czym w węźle wewnętrznym znajdujemy dziecko, do którego chcemy zejść, za pomocą wyszukiwania binarnego w tablicy trzymanej w węźle. Wstawianie i usuwanie implementujemy podobnie.

Zalety:

- mała liczba odczytów z dysku,
- dobry kompromis między dostępem swobodnym i sekwencyjnym (efektywne dla obu),
- dobrze sprawdzają się nie tylko dla warunków równościowych, ale również dla zapytań zakresowych,
- świetnie działają na każdym poziomie hierarchii pamięci.

Indeks zgrupowany – kolejność danych w liściach odpowiada kolejności danych zapisanych na stronach pliku. Opłaca się je stosować, jeśli pracujemy na danych w konkretnym porządku lub gdy pracujemy na zakresach lub danych zgrupowanych. Stosuje się je na atrybutach, których wartości rzadko się powtarzają i zmieniają. Dobrze jest, jeśli są dodawane w porządku rosnącym.

Indeks niezgrupowany – kolejność danych na stronach nie ma znaczenia.

Indeks gęsty – wskazuje dokładny adres każdego rekordu dla danego klucza. Zapewnia najszybsze wyszukiwanie, ale zajmuje więcej pamięci i jest bardziej kosztowny w utrzymaniu podczas dodawania lub usuwania danych.

Indeks rzadki – wskazuje jedynie na pierwszy blok danych zawierający szukany rekord. Stanowi kompromis – pozwala zaoszczędzić pamięć, wciąż skutecznie przyspieszając wyszukiwanie.

Indeks częściowy – jest tworzony tylko dla określonego podzbioru wierszy w tabeli. Zajmuje znacznie mniej miejsca, szybciej się aktualizuje i idealnie optymalizuje zapytania celujące wyłącznie w ten konkretny fragment danych.

Indeks pokrywający – zawiera nie tylko atrybuty, na który tworzony jest indeks, ale również pewne inne kolumny o których się spodziewamy, że będą potrzebne w zapytaniach. Kosztem zwiększenia rozmiaru indeksów unikamy potrzeby czytania pamięci z innego miejsca dysku.

Indeks haszowy – wykorzystuje tablice haszujące, doskonale sprawdza się przy zapytaniach równościowych (randomizowany stały czas dostępu), jednak nie znajduje zastosowania przy wyszukiwaniach zakresowych (funkcja haszujaca niszczy naturalny porządek na danych). Mniejszy rozmiar niż $B+$-drzewa, wielkość tablic haszujących nie zależy od rozmiaru samych haszowanych wartości, a ich rozmiary rosną skokowo od liczby trzymanych elementów, a nie liniowo jak w przypadku $B+$-drzew.

Indeks bitmapowy – dla każdego wiersza tabeli tworzymy ciąg bitów (bitmapę) długości takiej, jaka jest liczba unikalnych wartości kolumny (kolumn), na której zakładamy indeks. W tym ciągu mamy 1 na tej wartości, jaką ma ta kolumna w tym wierszu. Ponieważ wraz ze wzrostem liczby możliwych wartości rośnie rozmiar indeksu bitmapowego, stosuje się go najczęściej dla atrybutów o niewielkiej dziedzinie. SZBD może błyskawicznie łączyć wiele indeksów bitmapowych za pomocą szybkich operacji bitowych, ale modyfikacje danych są bardzo kosztowne.

III.1.5 Klucze w bazach relacyjnych

Omów pojęcie kluczy (podstawowych, wtórnych, obcych) w teorii relacyjnych baz danych.

Superklucz relacji $R(A_1, \ldots, A_n)$ to zbiór atrybutów jednoznacznie identyfikujący krotki tej relacji, czyli taki zbiór atrybutów $\{B_1, \ldots, B_k\} \subseteq \{A_1, \ldots, A_n\}$, że dla dowolnych krotek t_1 i t_2 w dowolnej instancji relacji R zachodzi

$$t_1[B_1, \ldots, B_k] = t_2[B_1, \ldots, B_k] \implies t_1 = t_2.$$

Przykład. Superkluczem jest zbiór wszystkich atrybutów relacji. Superklucz może posiadać nadmiarowe atrybuty.

Klucz relacji definiujemy jako minimalny w sensie inkluzji superklucz, czyli taki nie mający nadmiarowych atrybutów. Klucze służą do jednoznacznej identyfikacji rekordów, zapewnienia integralności danych, definiowania powiązań między tabelami, eliminacji redundancji i anomalii. Kluczem prostym nazywamy klucz składający się z jednego atrybutu. Klucz złożony to taki składający się z wielu atrybutów.

Klucze podstawowe i wtórne.

- **Kluczem podstawowym (głównym)** relacji nazwiemy jeden ustalony, wybrany (arbitralnie) klucz.
- Pozostałe klucze to **klucze wtórne**.
- Ogólnie dowolny klucz nazywamy czasem **kluczem kandydującym**.

Klucz obcy to zbiór atrybutów w danej relacji, który stanowi bezpośrednie odwołanie do klucza głównego innej relacji. Wymaga on zgodności typów danych, a każda niepusta wartość klucza obcego musi wskazywać na dokładnie jedną, istniejącą krotkę w tabeli docelowej (o tej samej wartości na kluczu głównym) – tak zwana integralność referencyjna. Służy do reprezentowania związków między tabelami.

Przykład:

działy(id_dzialu, nazwa_dzialu)

id_dzialu	nazwa_dzialu
10	Informatyka
20	Kadry

pracownicy(id_prac, pesel, nazwisko, id_dzialu)

id_prac	pesel	nazwisko	id_dzialu
1	90010112345	Kowalski	10
2	85020298765	Nowak	20
3	92030311223	Wiśniewski	NULL

- **Superklucz:** - np. {id_prac}, {pesel}, {id_prac, nazwisko}, {pesel, id_dzialu}.
- **Klucze:** {id_prac} {pesel}.
- **Klucz podstawowy (główny):** {id_prac}
- **Klucze wtórne:** {pesel}
- **Klucz obcy:** Łączący tabelę *pracownicy* z tabelą *działy*. Zbiór {id_dzialu} w tabeli *pracownicy* jest kluczem obcym wskazującym na klucz główny tabeli *działy*. Jeden z pracowników ma NULL – oznacza to, że nie przypisano go jeszcze do żadnego działu (jest to zgodne z definicją klucza obcego).

Przy definicji klucza obcego w języku SQL można określić zachowanie systemu w momencie usunięcia rekordu nadrzędnego (tego, na który klucz obcy wskazuje). Mamy opcje:

- `ON DELETE CASCADE` – usunięcie rekordu nadrzędnego powoduje usunięcie rekordów zależnych.
- `ON DELETE SET NULL` – po usunięciu rekordu nadrzędnego wartość klucza obcego zostaje ustawiona na NULL.
- `ON DELETE RESTRICT` – usunięcie rekordu nadrzędnego jest zabronione, jeśli istnieją rekordy zależne.

Istnieją analogiczne reguły dla operacji `UPDATE`.

III.1.6 Transakcje

Omów pojęcie transakcji i przedstaw jej własności.

Operacja **odczytu** pobiera z relacji wartości krotki. Oznaczamy odczyt krotki A przez $r(A)$. Operacja **zapisu** zmienia wartość krotki. Wyróżniamy jej trzy rodzaje: INSERT, UPDATE i DELETE. Oznaczamy taką operację dla krotki A przez $w(A)$.

TRANSAKCJA to ciąg jednej lub wielu operacji odczytu i zapisu, traktowany jako niepodzielna całość *(wykonują się wszystkie operacje, albo nie wykonuje się żadna)*. Oznaczenia:

- `b` (begin) – początek transakcji.
- `c` (commit) – zatwierdzenie transakcji.
- `a` (abort / rollback) – wycofanie (przerwanie) transakcji.
- Relacja poprzedzania $x \to y$ oznacza, że operacja x jest wykonywana przed operacją y.

Przykładowo, transakcja przelewu z konta K_2 na konto K_9 postępuje tak:

$$\mathrm{b}_1 \to \mathrm{r}_1(K_2) \to \mathrm{w}_1(K_2) \to \mathrm{r}_1(K_9) \to \mathrm{w}_1(K_9) \to \mathrm{c}_1.$$

Zadaniem SZBD będzie bezpieczne realizowanie dziejących się równolegle transakcji na podstawie tego, jakiego rodzaju operacje wykonują (odczyty i zapisy do odpowiednich krotek – SZBD nie patrzy w tym wypadku na to, jakie konkretnie wartości pojawiają się w tych krotkach).

Współbieżne wykonywanie transakcji wymaga zachowania czterech własności, znanych jako **ACID**.

Atomowość (Atomicity) – transakcja jest niepodzielna *(albo w pełni zatwierdzone są wszystkie jej operacje, albo wycofane są wszystkie)*.

- Niezatwierdzone zmiany nie są widoczne dla innych transakcji w systemie.
- Przerwanie transakcji może nastąpić z winy użytkownika/aplikacji, albo z interwencji SZBD (np. wykrycie zakleszczenia lub ryzyka rozspójnienia bazy). W takiej sytuacji wszystkie zmiany dokonane przez transakcję są wycofywane i nikt z zewnątrz nigdy ich nie widzi.
- Jak zapewnić atomowość SZBD najczęściej loguje zmiany zamiast wprowadzać je bezpośrednio do bazy i wprowadza je dopiero, gdy transakcja ma zostać zatwierdzona). Czasem (ale rzadko) stosuje się *shadow paging* – tworzenie kopii modyfikowanych stron, które transakcja modyfikuje i udostępnienie ich do odczytu dopiero po zatwierdzeniu transakcji.

Spójność (Consistency) – po zatwierdzeniu transakcji muszą być spełnione wszystkie warunki poprawności nałożone na bazę *(baza prawidłowo modeluje daną część rzeczywistości)*. W trakcie trwania transakcji (pomiędzy `b` a `c`) pojedyncze operacje mogą *chwilowo* naruszać te ograniczenia. Ważny jest tylko poprawny stan przed `b` i po `c`.

Izolacja (Isolation) – dwie transakcje wykonywane w tym samym czasie nie wpływają na siebie. Z punktu widzenia pojedynczej transakcji wygląda to tak, jakby była ona w danej chwili jedyną operacją w systemie (szeregowalność). Mamy dwa rodzaje strategii zapewniających izolację.

- **Protokoły pesymistyczne** nie pozwalają na zaistnienie problemu (np. przez odgórne blokowanie dostępu do krotek).
- **Protokoły optymistyczne** zakładają, że konflikty zdarzają się rzadko. Pozwalają transakcjom działać, a przy zatwierdzaniu sprawdzają, czy wystąpił konflikt. Jeśli tak – transakcja jest wycofywana.

Trwałość (Durability) – po udanym zatwierdzeniu transakcji jej efekty na stałe pozostają w bazie danych i nie zostaną utracone, nawet w przypadku twardej awarii (np. utraty zasilania, błędu systemu operacyjnego). Model awarii zakłada, że ginie tylko zawartość pamięci RAM, a dane na dysku zostają. Aby zapewnić trwałość możemy fizycznie zapisywać modyfikacje na dysku przed zakończeniem transakcji lub przechowywać dziennik logów (protokół WAL), który pozwala na odtworzenie bazy po restarcie (więcej o tym przy pytaniu o zapobieganiu utracie danych).

Realizacja transakcji

Niech $\mathcal{T} = \{T_1, \ldots, T_n\}$ będzie zbiorem transakcji. **Realizacją** (lub planem wykonania) transakcji z tego zbioru nazywamy częściowy porządek $(\mathcal{O}_\mathcal{T}, \leqslant)$, w którym spełnione są trzy warunki:

1. $\mathcal{O}_\mathcal{T}$ jest zbiorem wszystkich operacji z tych transakcji (wraz z ich startami `b` i zakończeniami `c`/`a`).
2. Porządek $\leqslant$ jest zgodny z wewnętrznym porządkiem każdej transakcji ($\to_i$) *(oznacza to, że jeśli w kodzie transakcji odczyt był przed zapisem, system musi tego przestrzegać)*.
3. Jeśli dwie operacje o i o' żądają dostępu do tej samej krotki i co najmniej jedna z nich to operacja zapisu (w), to system musi ustalić ich dokładną kolejność (zachodzi $o \leqslant o'$ albo $o' \leqslant o$).

Rodzaje realizacji:

- **Sekwencyjna** to realizacja, w której transakcje w ogóle się nie przeplatają. Dla każdych dwóch transakcji, wszystkie operacje jednej poprzedzają wszystkie operacje drugiej (wykonują się jedna po drugiej).
- **Współbieżna** to realizacja, która nie jest sekwencyjna. Operacje z różnych transakcji przeplatają się w czasie, co zwiększa wydajność systemu, ale wymaga stosowania mechanizmów izolacji.
- **Równoważne** – dwie realizacje nazywamy równoważnymi, jeżeli dają dokładnie ten sam stan końcowy bazy danych oraz zbiór wartości odczytanych przez każdą pojedynczą transakcję jest w obu realizacjach identyczny.
- **Szeregowalna (Poprawna)** – realizacja współbieżna, która jest równoważna pewnej realizacji sekwencyjnej *(czyli system przeplata instrukcje, ale ostateczny efekt jest taki, jakby transakcje wykonały się po kolei).*

Problem szeregowalności
`INPUT:` Realizacja R dla zbioru transakcji $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$
`OUTPUT:` Czy R jest szeregowalna?
Zgodnie z twierdzeniem Papadimitriou'a z 1979 problem szeregowalności jest NP-zupełny.

Celem SZBD jest znalezienie takiej realizacji ciągu transakcji, która będzie szeregowalna. Może wybrać dowolną taką realizację spośród wielu, niekoniecznie sobie równoważnych. Jak widzimy jest to problem trudny, ale będziemy się starać.

Operacje konfliktowe to takie, które spełniają poniższe warunki:

- należą do różnych transakcji,
- dotyczą tego samego obiektu bazodanowego,
- co najmniej jedna jest operacją zapisu.

Anomalie wynikające z konfliktów.

- **Niepowtarzalne odczyty** (konflikt read-write, transakcja odczytuje różne wartości)

  Schemat: $r_1(A) \to r_2(A) \to w_2(A) \to c_2 \to r_1(A) \to c_1$.
- **Brudne odczyty** (konflikt write-read, transakcja odczytuje zmiany innej, która nie została zatwierdzona)

  Schemat: $r_1(A) \to w_1(A) \to r_2(A) \to w_2(A) \to c_2 \to a_1$.
- **Nadpisywanie niezatwierdzonych zmian** (konflikt write-write, transakcja nadpisuje zmiany innej, która nie została zatwierdzona)

  Schemat: $w_1(A) \to w_2(A) \to w_2(B) \to w_1(B) \to c_2 \to c_1$.

Dwie realizacje (R_1 i R_2) są uznawane za **konfliktowo równoważne**, jeżeli każda para operacji konfliktowych występuje w obu realizacjach dokładnie w tej samej kolejności. Realizacja jest **konfliktowo szeregowalna**, jeśli jest konfliktowo równoważna pewnej realizacji sekwencyjnej. W praktyce oznacza to, że można ją przekształcić w bezpieczną realizację sekwencyjną wyłącznie poprzez zamianę miejscami operacji, które ze sobą nie konfliktują.

Okazuje się, że realizacja jest konfliktowo szeregowalna wtedy i tylko wtedy, gdy w jej grafie konfliktów (graf skierowany na transakcjach, gdzie krawędź to operacje konfliktowa; krawędź idzie w kierunku późniejszej operacja) nie ma cykli. Taka realizacja jest dobra, ale na przykład w przypadku wystąpienia awarii niewystarczająca – może być tak, że jedna transakcja dokona zmian, potem druga je odczyta i zakończy się, a pierwsza przez swoim zakończeniem zostanie wycofana (lub nastąpi awaria). Gdyby transakcja nie została wycofana, to wszystko byłoby dobrze (druga transakcja wyglądałaby, jakby wydarzyła się w całości po pierwszej). Jednak wycofanie transakcji spowodowało błąd, którego pojęcie konfliktowej szeregowalności nie wykryło.

Blokada to zmienna skojarzona z danym obiektem w bazie (np. krotką), określająca, czy można na nim wykonać konkretną operację. Jest to podstawowy mechanizm zapobiegania konfliktom.

Rodzaje blokad:

- **0** – dane nie są zablokowane (wolny dostęp).
- **S (shared lock; blokada dzielona)** – umożliwia współdzielony dostęp do zasobu. Wiele transakcji może jednocześnie założyć blokadę S i dokonywać odczytu tej samej krotki, ale żadna nie może jej zmienić.
- **X (exclusive lock; blokada wyłączna)** – daje absolutną wyłączność na dany zasób. Tylko jedna transakcja może ją założyć i jako jedyna ma prawo do modyfikowania zawartości krotki.

Aby zapewnić konfliktową szeregowalność za pomocą blokad, stosuje się tak zwany protokół 2PL (two-phase locking). Dzieli się on na dwie fazy:

1. Faza rozszerzania (Growing phase): Transakcja może uzyskiwać nowe blokady, ale nie może żadnej zwolnić.
2. Faza kurczenia (Shrinking phase): Transakcja może zwalniać blokady, ale nie może już uzyskać żadnej nowej.

W praktyce stosuje się Rygorystyczny 2PL (Strict 2PL), gdzie wszystkie blokady wyłączne (X) są zwalniane dopiero po zakończeniu transakcji (commit/abort), co zapobiega powstawaniu kaskadowych wycofań (sytuacji, gdy wycofanie jednej transakcji wymusza wycofanie drugiej).

Stosując przedstawione metody (i jeszcze trochę innych) SZBD jest w stanie uniknąć błędów związanych ze współbieżnością (musi przy tym czasem radzić sobie z możliwymi deadlockami). Standard SQL definiuje następujące poziomy izolacji transakcji, które SZBD musi być w stanie zapewnić.

1. poziom 0 (`read uncommitted`): dozwolone są odczyty niezatwierdzonych danych.
2. poziom 1 (`read committed`): tylko zatwierdzone zmiany mogą zostać odczytane, jednak niepowtarzalne odczyty są dopuszczalne,
3. poziom 2 (`repeatable read`): zmienione dane mogą być odczytane tylko po zatwierdzeniu, pomiędzy dwoma odczytami tej samej danej przez dwie transakcje żadna inna transakcja nie może tej danej zmieniać, mogą się jednak pojawić fantomy (sytuacje, gdy jedna transakcja dodaje nowe krotki, przez co inna może dostawać niepowtarzalne odczyty),
4. poziom 3 (`serializable`): realizacja transakcji jest szeregowalna.

III.1.7 Związki encji

Omów sposób modelowania baz danych za pomocą związków encji.

Główną część procesu projektowania bazy stanowi podejmowanie decyzji, jak reprezentować różne typy „bytów”. Zbiory obiektów (materialnych lub abstrakcyjnych) przechowywanych w bazie i opisanych tymi samym własnościami (atrybutami) będziemy nazywać zbiorami **encji (entity)**.

Na przykład w bazie uniwersytetu możemy mieć zbiór „kurs” zawierający **encje** opisane atrybutami kurs_id, nazwa, kierunek oraz ects. Instancją takiej encji może być krotka

(ID, inżynieria danych, informatyka analityczna, 6).

Atrybuty proste to takie, których wartości nie możemy podzielić na części.

Atrybuty złożone to takie, które mogą mieć wiele składowych.

Atrybuty encji można podzielić ze względu na to, ile wartości przechowują. Mogą być jednowartościowe lub wielowartościowe (np. być zbiorem kilku wartości).

Atrybuty pochodne to takie, których wartości obliczane są za pomocą pewnej funkcji zastosowanej na wartościach innych atrybutów. Ich wartości nie wprowadza się ani nie przechowuje, wyliczamy je, gdy są potrzebne *(np. `wiek()`, gdy encja opisana jest atrybutem `data_urodzenia`).*

Związek (relationship) to wzajemna zależność pomiędzy dwiema lub więcej encjami. Klasyfikujemy ze względu na:

- stopień (narny, binarny, ternarny, n-arny) określający, ile encji uczestniczy w związku.
- klasę przynależności; związek R jest **obowiązkowy**, gdy dla zadanego zbioru encji E (który bierze udział w R) każda encja z tego zbioru jest w co najmniej jednym związku z R. W przeciwnym wypadku R jest **opcjonalny**.
- typ asocjacyjny; dla związków binarnych mamy następujące typy asocjacyjne:
  - jeden do jeden $(1 : 1)$ – każda instancja pierwszej encji jest w związku z co najwyżej jedną instancją drugiej encji i odwrotnie,
  - jeden do wielu $(1 : M)$ – instancje pierwszej encji mogą wchodzić w związki z wieloma instancjami drugiej encji, natomiast każda instancja drugiej encji jest w związku z co najwyżej jedną instancją pierwszej encji,
  - wiele do wielu $(M : N)$ – każda instancja pierwszej encji jest w związku z dowolną liczbą instancji drugiej encji i odwrotnie.

Związek również może mieć atrybuty, które nazywamy deskryptywnymi (lub opisowymi). Na przykład encja „student” może być w związku z encją „kurs”, a ten związek może mieć atrybut „ocena”.

Instancja związku reprezentuje związek między instancjami encji.

Dla zbioru encji wprowadzamy pojęcia superklucza, klucza i klucza głównego analogicznie jak w modelu relacyjnym. Dla zbioru związków również możemy definiować klucze. Jeśli R jest związkiem między zbiorami encji E_1 i E_2, to dla związku typu $1 : M$ kluczem głównym R jest klucz główny E_2, a dla związku typu $M : N$ jest to suma kluczy E_1 i E_2.

Encję nazwiemy **słabą**, gdy jej wystąpienia istnieją tylko w kontekście wystąpień innych encji. Pozostałe encje nazywamy **silnymi**. Dla zbioru E słabych encji klucz główny jest zdefiniowany nie tylko w oparciu o atrybuty encji z E, ale zależy również od zbiorów encji, z którymi encje z E wchodzą w związki. W szczególności słabe encje nie posiadają własnego identyfikatora. Na przykład encja „zajęcia” jest słabą encją i zależy od encji „kurs”. Instancje encji „zajęcia” identyfikujemy na podstawie klucza głównego dla zbioru encji „kurs” oraz atrybutów „semestr” oraz „rok”.

Związki wyłączne to takie, że jedna instancja danej encji może wejść tylko w jeden z nich.

Zbiór encji może zawierać podzbiór encji, które w pewien sposób odróżniają się od pozostałych. W szczególności mogą być opisane atrybutami, które nie mają zastosowania do pozostałych encji. Mówimy wtedy o **encjach specjalizacji** (lub podencjach). Zbiory encji specjalizacji mogą być rozłączne, ale nie muszą. Na przykład encja „osoba” może mieć podencje „pracownik” i „student”. Doktoranci mogą mieć zarówno pracowników, jak i studentów.

Mając te wszystkie pojęcia możemy modelować różne byty, które występują w naszej bazie danych. Określamy encje, jakie występują w naszej bazie, a następnie związki między nimi. Wtedy może okazać się, że niektóre encje mają nadmiarowe atrybuty. Na przykład encja „pracownik” może mieć atrybut „nazwa_instytutu” i być w związku z encją „instytut”, która też ma taki atrybut. Zatem „pracownik” ma nadmiarowy atrybut – można go usunąć, a informacja o przynależności pracownika do instytutu będzie przechowywana w odpowiedniej instancji związku. Dzięki temu unikamy też założenia, że pracownikowi przypisane jest tylko jedno miejsce pracy (jeśli takie ograniczenie rzeczywiście jest prawdziwe, to możemy je zamodelować na poziomie związku).

Związki n-arne możemy zamodelować za pomocą wielu związków binarnych. Mając związek R w którym uczestniczą zbiory encji A, B, C możemy stworzyć nowy zbiór encji E. Encje z E będą modelować instancję związku z R. Następnie tworzymy relacje binarne R_A, R_B, R_C. Instancje R_A które łączą encję z A z tymi encjami z E, które modeluje instancje związku R, do których należy ta encja z A. Analogicznie dla R_B i R_C.

Model związków encji jest abstrakcyjny, aby go zaimplementować musimy go przenieść do pewnego modelu implementacyjnego baz danych. Musimy znaleźć sposób reprezentowania encji (zarówno silnych, jak i słabych), związków (wraz z ich arnościami, typami i klasami przynależności), atrybutów oraz kluczy encji i związków oraz hierarchii encji w języku tabel, atrybutów relacji oraz kluczy podstawowych i obcych.

Przekształcanie silnych encji na relacje jest proste – zbiór encji staje się tabelą, atrybuty encji to atrybuty tabeli, klucz główny encji to klucz główny tabeli. Pojedyncza encja jest krotką w tabeli.

W przypadku atrybutów złożonych rozbijamy je na ich składowe atrybuty proste i to je umieszczamy w schemacie tabeli.

Jeśli encja ma atrybut wielowartościowy, to tworzymy dla niego osobną tabelę, która trzyma wartość atrybutu i klucz główny tej encji (który staje się teraz kluczem obcym). Encja ma swoją osobną tabelę, bez tego atrybutu. Wyjątkiem jest sytuacja, gdy encja ma tylko dwa atrybuty: ten wielowartościowy a i drugi e, będący kluczem. Wtedy tworzymy jedną tabelę, w której kluczem głównym jest $\{a, e\}$.

Jeśli mamy słabą encję, to odpowiadająca jej tabela będzie zawierać atrybuty tej encji oraz klucz główny encji, od której ta słaba encja zależy (staje się on tu kluczem obcym).

Sposób przedstawiania związku w modelu relacyjnym jest zależny od typu związku.

- Związki binarne typu $1 : 1$ reprezentujemy dodając klucz obcy dla tabeli, dla której związek jest obowiązkowy, lub dla tabeli o mniejszym rozmiarze.
- Związki binarne typu $1 : M$ reprezentujemy przez dodanie klucza obcego do tabeli po stronie „wiele".
- Związki unarne typu $1 : 1$ reprezentujemy przez dodanie klucza obcego do danej tabeli.
- Związki typu $M : N$ reprezentujemy przez wprowadzenie dodatkowej tabeli.

Encje specjalizacji możemy modelować na różne sposoby.

- Możemy mieć jedną tabelę dla encji i jej podencji. Wtedy w schemacie znajdą się wszystkie możliwe atrybuty – zarówno wspólne, jak i te specyficzne dla podencji. Jeśli encja nie posiada danego atrybutu, to ustawiamy wartość NULL.
- Możemy stworzyć osobną tabelę dla każdej podencji. W schematach tabel dla podencji znajdą się wszystkie atrybuty dziedziczone z encji wyższego rzędu oraz atrybuty charakteryzujące daną podencję. Nie działa to za dobrze, gdy podencje nie muszą być rozłączne (wtedy musimy dodatkowo na przykład stosować poprzedni punkt). Jeśli każda encja nadrzędna ma cechy jednej z podencji, to nie musimy tworzyć dodatkowej tabeli dla encji nadrzędnej.
- Możemy stworzyć oddzielne tabele dla wspólnych atrybutów i oddzielne dla atrybutów specyficznych dla podencji. Dodatkowo tabele atrybutów specyficznych mają klucze obce wskazujące na atrybuty encji nadrzędnej.

Do modelowania związków wyłącznych możemy zamodelować oba związki w jednej tabeli. Jeśli mamy na przykład encję „usługa", która może być w związku „zlecenie" z encją „osoba" lub encją „firma", to do tabeli usługi dodajemy klucze obce oznaczające osobę i firmę. Jeden z nich zawsze musi być NULLem. Inną opcją jest dodanie atrybutu „typ_klienta" oraz „id_klienta", który wskazuje na klucz główny w odpowiedniej tabeli (jest to albo osoba, albo firma).

III.2 Inżynieria Oprogramowania

III.2.1 Test Driven Development

Na czym polega wytwarzanie sterowane testami (test-driven development)? Wymień podstawowe praktyki z nim związane.

Metodyka tworzenia oprogramowania, w której proces implementacji jest bezpośrednio sterowany przez uprzednio przygotowane testy automatyczne.

Flow pracy w TDD polega na cyklu **Red-Green-Refactor**:

1. **Red:** Piszemy pojedynczy test jednostkowy dla funkcjonalności, która jeszcze nie istnieje. Uruchomienie testu musi zakończyć się niepowodzeniem (kod nie kompiluje się lub test zgłasza błąd), co potwierdza, że nowy test faktycznie sprawdza brakujący element.
2. **Green:** Następnie piszemy minimalną ilość kodu niezbędną do tego, aby nowo utworzony test (oraz wszystkie dotychczasowe) zakończył się sukcesem. Na tym etapie jakość kodu jest drugorzędna, liczy się spełnienie wymagań testu.
3. **Refactor:** Modyfikujemy kod w celu poprawy jego struktury, czytelności i architektury, bez zmiany jego zewnętrznego zachowania. Poprawność refaktoryzacji jest stale kontrolowana przez zielony status zestawu testów.

Podstawowe praktyki związane z TDD

Skuteczne stosowanie TDD opiera się na zestawie fundamentalnych praktyk inżynierskich:

- **Pisanie testu przed kodem produkcyjnym:** nie implementujemy żadnej funkcjonalności bez uprzedniego posiadania failującego testu.
- **Krótkie cykle iteracyjne:** przejście przez pełny cykl Red-Green-Refactor powinno zajmować od kilku do kilkunastu minut, co ma zapobiegać tworzeniu zbyt skomplikowanych struktur kodu na raz.
- **Zasada jednej odpowiedzialności testu:** każdy test powinien sprawdzać tylko jedną, precyzyjnie zdefiniowaną funkcjonalność.
- **Tworzenie minimalnego kodu produkcyjnego:** implementacja dokładnie takiej logiki, jaka jest wymagana do zaliczenia testu.
- **Ciągłe uruchamianie regresji:** wszystkie istniejące testy są uruchamiane po każdej zmianie w kodzie, co chroni przed wprowadzaniem błędów regresyjnych.
- **Testy jako dokumentacja:** zestaw testów jest traktowany jako precyzyjna, zawsze aktualna specyfikacja techniczna systemu, która opisuje, jak oprogramowanie powinno się zachowywać.

Stosowanie TDD powoduje, że jakość kodu jest wysoka a jego architektura przemyślana – kod jest modularny (bo jego fragmenty powstałe podczas jednego cyklu spełniają kolejne zadanie) i napisany tak, by łatwo było go testować. Istnienie pakietu testów powoduje, że błędy w kodzie są wyłapywane szybko i zazwyczaj dokładnie wiadomo, jaka zmiana je spowodowała. Dają one programiście pewność, że jego nowe zmiany nie zepsuły istniejącej logiki. Mimo to TDD ma też swoje wady. Wymaga zmiany sposobu myślenia o pisaniu kodu i umiejętności pisania testów, przez co ma całkiem spory próg wejścia. Do tego w początkowej fazie pisania projektu obowiązek tworzenia testów znacząco spowalnia tworzenie kodu, a nie oferuje tak wielu korzyści (bo jeszcze nie istnieje codebase, którego poprawność chcemy zachować).

III.2.2 Dependency injection

Na czym polega wstrzykiwanie zależności (dependency injection)? Dlaczego ułatwia testowanie i integrację składników systemu?

Dependency injection (DI) to wzorzec projektowy/technika programistyczna stanowiąca kluczową implementację zasady odwrócenia sterowania (*Inversion of Control*), która polega na przekazaniu kontroli nad implementacją zależności z jakich korzysta pewna klasa poza tę klasę (tego pojęcia używa się też do określenia technik, które zmieniają typowy sekwencyjny sposób wykonywania kodu; tutaj chodzi o coś innego).

DI polega na odebraniu klasie odpowiedzialności za samodzielne tworzenie i konfigurowanie obiektów, od których jest zależna, i delegowaniu tego zadania zewnętrznemu mechanizmowi (np. kontenerowi DI). Klasa definiuje jedynie swoje wymagania (najczęściej poprzez abstrakcyjne interfejsy), a gotowe instancje są jej dostarczane z zewnątrz. Zazwyczaj dzieje się to poprzez przekazanie odpowiednich obiektów w konstruktorze (constructor injection) lub poprzez udostępnienie przez obiekt tak zwanych setterów, czyli metod, które umożliwiają ustawienie wartości pewnego pola tego obiektu. Ta druga technika ma szczególne zastosowanie, gdy wstrzykiwany obiekt będzie się zmieniał w trakcie wykonywania programu. Istnieje też opcja tak zwanego wstrzykiwania przez pola (field injection) – zależności są wstrzykiwane bezpośrednio bo prywatnych pól klasy przy użyciu refleksji. Jest ona często stosowana, gdy korzystamy z frameworków, które implementują mechanizm takiego wstrzykiwania. Jest to bardzo wygodny sposób na wykonanie DI, bo zazwyczaj nie wymaga zbyt wiele dodatkowego kodu (zwykle jest to jedna adnotacja przy odpowiednim polu), ale może być nieczytelny i bardzo mocno wiąże pisany kod z konkretnym frameworkiem.

Dlaczego DI ułatwia testowanie jednostkowe? DI jest fundamentalnym warunkiem efektywnego stosowania testów jednostkowych, ponieważ umożliwia całkowitą izolację testowanego komponentu:

- **Eliminacja twardych powiązań:** Bez DI klasa biznesowa sama tworzy np. obiekt bazy danych lub klienta API. Testowanie jej zmusza do komunikacji z realną infrastrukturą sieciową, co czyni testy wolnymi i podatnymi na błędy zewnętrzne.
- **Stosowanie atrap (Mocks/Stubs):** Dzięki DI, na potrzeby środowiska testowego, możemy wstrzyknąć klasie mock, który jedynie symuluje prawdziwą zależność i natychmiast zwraca z góry zaprogramowane dane. Testy wykonują się wówczas w milisekundy, są w pełni powtarzalne i niezależne od czynników zewnętrznych.

Dlaczego DI ułatwia integrację składników systemu? W kontekście architektury całego systemu DI działa jak uniwersalny interfejs łączący niezależne moduły oprogramowania:

- **Komunikacja poprzez abstrakcję:** Komponenty systemu nie znają nawzajem swoich konkretnych implementacji, komunikują się wyłącznie za pomocą abstrakcyjnych interfejsów. Pozwala to zespołom programistycznym na w pełni równoległą pracę nad różnymi modułami po uprzednim ustaleniu wspólnego kontraktu.
- **Zasada wymienności (Plug-and-Play):** Jeśli zachodzi potrzeba wymiany technologii (np. migracja z bazy danych MySQL na PostgreSQL), nie ma potrzeby modyfikacji kodu logiki biznesowej aplikacji. Tworzy się nową klasę implementującą istniejący interfejs, a zmianę konfiguracji wprowadza się wyłącznie w konfiguracji kontenera DI.
- **Separacja warstw i przejrzystość kodu:** Klasy odpowiedzialne za logikę aplikacji nie muszą same tworzyć potrzebnych im obiektów ani zajmować się ich konfiguracją. Wszystko to zostaje przeniesiona na zewnątrz. Dzięki temu kod nie jest zaśmiecony kwestiami technicznymi, a każda klasa skupia się wyłącznie na swoim głównym zadaniu, co znacznie zwiększa czytelność i ułatwia rozwój programu.

III.2.3 Wzorce projektowe

Co to jest wzorzec projektowy? Przedstaw (np. za pomocą diagramów UML) dwa przykłady wzorców służących do eliminacji/odwracania zależności.

Wzorzec projektowy (*design pattern*) to uniwersalne, sprawdzone rozwiązanie problemu powszechnie występującego przy projektowaniu oprogramowania. Jest to abstrakcyjny szablon opisujący strukturę klas, obiektów oraz ich wzajemne relacje i interakcje, ułatwiający tworzenie elastycznego i łatwego w utrzymaniu kodu. Dzięki stosowaniu takich szablonów proces tworzenia oprogramowania jest szybszy, a pisany kod zrozumiały (inni programiści będą rozpoznawać zastosowane wzorce projektowe). Do tego istnienie ustalonych pojęć na pewne techniki stosowane przy pisaniu kodu ułatwia komunikację z zespole.

Wzorce projektowe zostały spopularyzowane przez książkę „Design Patterns: Elements of Reusable Object-Oriented Software". Pojawia się w niej podział na trzy klasy wzorców projektowych.

- Kreacyjne (konstrukcyjne) – związane z procesem tworzenia obiektów (np. Singleton, Fabryka).
- Strukturalne – związane ze składaniem większych struktur z klas i obiektów (np. Adapter, Dekorator).
- Behawioralne (czynnościowe) – związane z komunikacją między obiektami i podziałem odpowiedzialności (np. Strategia, Obserwator).

W projektowaniu obiektowym dąży się do luźnego powiązania (loose coupling) fragmentów kodu ze sobą. Zgodnie z zasadą DIP (Dependency Inversion Principle) z zestawu SOLID moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Oba powinny zależeć od abstrakcji, a abstrakcje nie powinny zależeć od szczegółów implementacyjnych. Oto kilka wzorców projektowych, których głównym celem jest realizacja zasady odwrócenia zależności.

Factory Method. Często tworzymy w naszym kodzie obiekty, z których funkcjonalności potem korzystamy. Jeśli robimy to wprost (za pomocą operatora `new`), to tworzymy zależność naszego kodu od pewnych konkretnych klas. Jest to problematyczne, jeśli mamy kilka różnych implementacji danej funkcjonalności i zależnie od sytuacji chcemy korzystać z różnych (czyli chcemy tworzyć obiekty różnych klas). Do tego przy pojawieniu się nowych implementacji będziemy musieli zmieniać istniejący kod. Ten problem rozwiązuje wzorzec projektowy Fabryka. Zamiast bezpośrednio tworzyć instancje obiektów w logice biznesowej, proces ten jest delegowany do specjalnej metody wytwórczej (zwanej fabryką). Obiekty zwracane przez tę metodę nazywamy produktami. W klasie bazowej (lub interfejsie) modelującej naszą wysokopoziomową funkcjonalność fabryka nie zwraca obiektu konkretnej klasy, a jedynie wysokopoziomowy interfejs definiujący funkcjonalność, którą produkt ma wykonywać. W klasach pochodnych nadpisujemy tę metodę i decydujemy o tym, jakiej klasy obiekt zostanie ostatecznie utworzony i zwrócony. Dzięki temu nasz wysokopoziomowy kod jest zupełnie oddzielony od procesu konstrukcji obiektów.

Przykład: mamy klasę `Logistics`, która ma metodę `createTransport()` zwracającą obiekt typu `Transport`. W klasie pochodnej `RoadLogistics` ta funkcja zwraca obiekt typu `Truck`, a w klasie `SeaLogistics` typu `Ship`. Następnie w innych funkcjach zdefiniowanych w `Logistics` korzystamy z `createTransport` i funkcjonalności zdefiniowanych w interfejsie `Transport`. Klasy pochodne ustalają konkretną implementację tego interfejsu.

Istnieje uogólnienie tego wzorca zwane Abstract Factory. W tym wzorcu fabryka nie jest metodą, a osobną klasą, która ma różne metody tworzące obiekty. Konkretne implementacje tej klasy zwracają w nim obiekty, które są ze sobą w jakiś sposób powiązane.

Strategy. Często w pisanych przez nas metodach chcemy dać użytkownikowi metody możliwość wyboru, w jaki konkretnie sposób metoda się zachowa. Na przykład mając funkcję sortującą chcemy mieć możliwość wyboru klucza sortowania. Ogólniej, chcemy, aby pisana przez nas klasa udostępniała funkcjonalność implementowaną przez pewien algorytm (na przykład funkcja znajdowania trasy w aplikacji mapy). Czasem chcemy zmieniać to, jak konkretnie ten algorytm działa. Aby móc to zrobić w sposób łatwy do rozszerzenia stosujemy wzorzec Strategia. Polega on na ekstrakcji poszczególnych algorytmów wykonujących dane zadanie na różne sposoby i umieszczenie ich w odrębnych klasach, zwanych strategiami. Pierwotna klasa, zwana kontekstem, musi zawierać pole służące przechowywaniu odniesienia do którejś ze strategii. Kontekst deleguje pracę powiązanemu obiektowi typu strategia, zamiast wykonywać

ją samodzielnie. Kontekst nie jest odpowiedzialny za wybór stosownego algorytmu dla danego zadania. To klient przekazuje żądaną strategię kontekstowi, który współpracuje ze wszystkimi strategiami za pośrednictwem tego samego, ogólnego interfejsu, który eksponuje pojedynczą metodę uruchamiającą algorytm ukryty w danej strategii. Tym sposobem kontekst staje się niezależny od konkretnych strategii, więc można dodawać kolejne algorytmy, lub modyfikować istniejące, bez zmieniania kodu kontekstu lub kodu innych strategii.

III.2.4 SOLID

Wymień 5 zasad SOLID. Omów krótko wybrane dwie, prezentując przykłady naruszenia tych zasad.

1. **S (Single Responsibility Principle)** – zasada pojedynczej odpowiedzialności.
2. **O (Open/Closed Principle)** – zasada otwarte/zamknięte.
3. **L (Liskov Substitution Principle)** – zasada podstawienia Liskov.
4. **I (Interface Segregation Principle)** – zasada segregacji interfejsów.
5. **D (Dependency Inversion Principle)** – zasada odwrócenia zależności.

1. **S - Single Responsibility Principle (SRP)**

   Klasa powinna realizować wyłącznie jedno, spójne zadanie. Jej kod modyfikujemy tylko wtedy, gdy zmieniają się wymagania dotyczące dokładnie tej jednej, konkretnej funkcjonalności.

```java
// ŹLE: Klasa realizuje skrajnie różne zadania. Trzeba będzie ją
    zmodyfikować przy zmianie struktury bazy, formatu pliku, serwera.

public class Faktura {
    public double obliczSume() { /* logika biznesowa */ }
    public void zapiszDoBazyDanych() { /* logika zapisu (SQL) */ }
    public void wygenerujPdf() { /* logika formatowania */ }
    public void wyslijEmail() { /* logika sieciowa */ }
}
```

```java
// DOBRZE: Wydzielamy dedykowane klasy, np. Faktura (logika biznesowa),
    FakturaRepository, PdfGenerator, EmailSender.
```

2. **O - Open/Closed Principle (OCP)**

   Klasy/moduły/funkcje powinny być otwarte na rozbudowę, ale zamknięte na modyfikacje. Powinniśmy móc dodawać do systemu nowe funkcjonalności poprzez dopisywanie nowego kodu (np. nowych klas implementujących wspólny interfejs), a nie poprzez ingerencję w już istniejący i przetestowany kod.

```java
// ŹLE: Wymaga modyfikacji klasy przy nowym typie klienta.

public class KalkulatorZnizek {
    public double obliczKwote(double kwota, String typKlienta) {
        if (typKlienta.equals("Zwykly")) {
            return kwota;
        } else if (typKlienta.equals("Premium")) {
            return kwota * 0.9;
        } else if (typKlienta.equals("VIP")) {
            return kwota * 0.8;
        }
        return kwota;
```

```
    }
}
```

```
// DOBRZE: Używamy polimorfizmu.

// Wspólny interfejs dla wszystkich reguł zniżkowych
public interface RegulaZnizki {
    double obliczKwotePoZnizce(double kwota);
}

// Osobne klasy dla każdego typu zniżki (dodanie nowej zniżki = nowa klasa)
public class ZnizkaZwykla implements RegulaZnizki {
    public double obliczKwotePoZnizce(double kwota) {
        return kwota;
    }
}

public class ZnizkaPremium implements RegulaZnizki {
    public double obliczKwotePoZnizce(double kwota) {
        return kwota * 0.9;
    }
}

public class ZnizkaVIP implements RegulaZnizki {
    public double obliczKwotePoZnizce(double kwota) {
        return kwota * 0.8;
    }
}

// Kalkulator
public class KalkulatorZnizek {
    public double obliczKwote(double kwota, RegulaZnizki regula) {
        return regula.obliczKwotePoZnizce(kwota);
    }
}
```

3. **L - Liskov Substitution Principle (LSP)**

   Obiekt klasy bazowej musi być w stanie poprawnie zastąpić klasę bazową (to znaczy sensownie, bez błędów, implementować wszystkie jej metody). Jeśli klasa B dziedziczy po klasie A, to każda metoda przyjmująca obiekt klasy A powinna działać poprawnie również po przekazaniu obiektu klasy B (bez niespodziewanych wyjątków czy błędów logicznych).

```
// ŹLE: Pingwin dziedziczy po Ptaku, ale nie potrafi latać.
// Wywołanie metody lec() na obiekcie Pingwin rzuci wyjątek, co psuje dział
    anie programu oczekującego standardowego Ptaka.

public class Ptak {
    public void lec() {
        System.out.println("Latam!");
    }
}

public class Pingwin extends Ptak {
    @Override
    public void lec() {
        throw new UnsupportedOperationException("Pingwiny nie latają!");
    }
}
```

```
// DOBRZE: Rozdzielamy hierarchię. Nie każdy ptak lata, więc wyodrębniamy
    osobną klasę lub interfejs dla ptaków latających.

```

```
public class Ptak {
    public void jedz() {
        System.out.println("Jem.");
    }
}

public class PtakLatajacy extends Ptak {
    public void lec() {
        System.out.println("Latam!");
    }
}

public class Pingwin extends Ptak {
    // Pingwin implementuje tylko to, co naprawdę potrafi jako Ptak.
}
```

4. **I - Interface Segregation Principle (ISP)**

   Należy unikać projektowania rozbudowanych, wielofunkcyjnych interfejsów na rzecz wielu wąsko wyspecjalizowanych. Klasy nie mogą być zobligowane do implementowania metod, których nie wykorzystują. Lepiej stworzyć kilka mniejszych, bardziej precyzyjnych interfejsów niż jeden wielki, który zmusza klasy do pisania pustych metod lub wyrzucania wyjątków.

```
// ŹLE: Interfejs Pracownik jest zbyt ogólny.
// Klasa Programista jest zmuszona do zaimplementowania metody
    zarzadzajZespolem(), której nie potrzebuje.

public interface Pracownik {
    void koduj();
    void zarzadzajZespolem();
}

public class Programista implements Pracownik {
    public void koduj() { /* pisanie kodu */ }
    public void zarzadzajZespolem() {
        // Pusta metoda lub wyjątek - Programista nie zarządza zespołem!
    }
}
```

```
// DOBRZE: Dzielimy duży interfejs na mniejsze, dedykowane role.

public interface ProgramistaTask {
    void koduj();
}

public interface MenadzerTask {
    void zarzadzajZespolem();
}

// Teraz każda klasa implementuje tylko to, czego faktycznie używa
public class Developer implements ProgramistaTask {
    public void koduj() { /* pisanie kodu */ }
}

public class TeamLeader implements ProgramistaTask, MenadzerTask {
    public void koduj() { /* pisanie kodu */ }
    public void zarzadzajZespolem() { /* zarządzanie ludźmi */ }
}
```

5. **D - Dependency Inversion (DI)**

   Architektura oprogramowania powinna opierać się na abstrakcjach. Moduły wysokiego poziomu (logika biznesowa) nie mogą zależeć od modułów niskiego poziomu (infrastruktura, narzędzia).

Obie warstwy powinny zależeć od abstrakcji (interfejsów). Ponadto abstrakcje nie powinny zależeć od szczegółów, ale szczegóły od abstrakcji.

```java
// ŹLE: Klasa wysokiego poziomu (SystemPowiadomien) bezpośrednio zależy od
    konkretnej klasy niskiego poziomu (SmsService).
// Zmiana sposobu wysyłki na Email wymaga modyfikacji kodu klasy
    SystemPowiadomien.

public class SmsService {
    public void wyslijSms(String wiadomosc) { /* logika wysyłki SMS */ }
}

public class SystemPowiadomien {
    private SmsService smsService = new SmsService(); // Sztywne powiązanie
        (Tight Coupling)

    public void powiadomOUproszczeniu(String tekst) {
        smsService.wyslijSms(tekst);
    }
}
```

```java
// DOBRZE: Wprowadzamy abstrakcję (interfejs). SystemPowiadomien nie wie,
    jak technicznie realizowana jest wysyłka.
// Konkretna usługa jest wstrzykiwana (np. przez konstruktor).

// Abstrakcja
public interface UsługaWysylki {
    void wyslij(String wiadomosc);
}

// Klasy niskiego poziomu implementują interfejs
public class SmsService implements UsługaWysylki {
    public void wyslij(String wiadomosc) { /* wysyłka SMS */ }
}

public class EmailService implements UsługaWysylki {
    public void wyslij(String wiadomosc) { /* wysyłka Email */ }
}

// Klasa wysokiego poziomu zależy wyłącznie od interfejsu
public class SystemPowiadomien {
    private final UsługaWysylki uslugaWysylki;

    // Konstruktor pozwala na wstrzyknięcie dowolnej implementacji (
        Dependency Injection)
    public SystemPowiadomien(UsługaWysylki uslugaWysylki) {
        this.uslugaWysylki = uslugaWysylki;
    }

    public void powiadomOUproszczeniu(String tekst) {
        uslugaWysylki.wyslij(tekst);
    }
}
```

III.3 Podstawy Programowania

III.3.1 Reprezentacja liczb

Omów reprezentację liczb całkowitych i rzeczywistych w komputerze (system binarny, szesnastkowy, zapis stało- i zmiennoprzecinkowy).

System binarny i szesnastkowy

Liczby reprezentowane są w systemie binarnym tj. za pomocą bitów, dwóch symboli standardowo oznaczanych 0 (fizycznie odpowiadające niskiemu poziomowi napięcia lub brakowi prądu) oraz 1 (analogicznie odpowiednio wysokie napięcie/przepływ prądu).

W związku z ograniczeniami skończonego świata, używane w komputerach operacje działają jedynie na skończonym podzbiorze liczb, który wynika z długości reprezentacji pojedynczej wartości jako ciągu 0 oraz 1; długość ta jest stała dla każdej wartości na której operujemy (czyli jest mocy 2^N gdzie N to długość reprezentacji).

Powyższe rozwiązanie prowadzi do znacznego przyspieszenia wszystkich operacji względem zapisu o zmiennej długości, jednakże jako efekt uboczny pozwala ono na klasę błędów underflow/overflow (przepełnienia), które ujawniają się w momencie wykonania operacji, której rozwiązanie znalazłoby się poza reprezentowalnym podzbiorem wartości. Podczas gdy najbezpieczniejszym rozwiązaniem byłoby rzucenie wyjątku, ze względów wydajności procesor jedynie ustawia odpowiednią flagę i kontynuuje ewaluację programu (wymagana jest więc ręczna weryfikacja, co może prowadzić do błędów i dziur bezpieczeństwa w przypadku zapomnienia o niej).

Co więcej, podczas gdy przy znaczącej większości operacji zakres wartości rzędu powszechnych 32 lub 64 bitów jest wystarczający, w niektórych specyficznych zastosowaniach, np. w kryptografii, obecność dużych liczb jest wymagana dla implementacji niektórych algorytmów. W związku z tym istnieją biblioteki pozwalające na pracę z dużymi liczbami (najbardziej powszechna jest GMP – GNU Multiple Precision); niektóre języki programowania (np. Python, Erlang) pozwalają na dowolnie długie wartości całkowite w ramach ich domyślnego typu liczby całkowitej (kosztem ich wydajności), podczas gdy inne (np. JavaScript, Haskell) oferują osobne typy spełniające taką rolę w ramach ich biblioteki standardowej.

Jak ustaliliśmy wyżej, liczby reprezentujemy jako ciąg dwóch wartości logicznych, w praktyce odpowiadających fizycznym wartością prądu w procesorze.

Zapis binarny polega na reprezentacji liczby jako ciągu znaków 0 oraz 1. Dokładna interpretacja tego ciągu znaków zależy od kontekstu w jakim pracujemy (liczby naturalne, całkowite czy rzeczywiste oraz *końcowości* porządku bajtów[2]).

System szesnastkowy polega na reprezentacji ciągu binarnego korzystając ze zbioru 16 wartości tj. od 0 do 9 i od a do f. Wartości te konwertujemy poprzez przypisanie wartości od 0 do 15 symbolom alfabetu szesnastkowego i przyrównanie ich do standardowego zapisu liczb całkowitych na 4 bitach (o którym mowa w sekcji niżej).

Jest on użyteczny, ponieważ pozwala na kompresję zapisu czterokrotnie, a co za tym idzie jest bardziej przejrzysty dla ludzi (którzy lepiej radzą sobie z bogatym zbiorem symboli; patrz alfabet; niż z dłuższym ciągiem dwóch symboli). Wykorzystywane są często również inne systemy (32-symbolowy i 64-symbolowy), jednak system szesnastkowy ma fundamentalną zaletę: ponieważ we współczesnych komputerach podstawową jednostką reprezentacji danych jest bajt, składający się z 8 bitów, to ciąg dwóch symboli szesnastkowych jest w bijekcji z wartością jednego bajta – jedyne inne systemy posiadające taką własność to system binarny (powszechny lecz długi), tetralny tj. czwórkowy (krótszy, lecz nie używany

[2] W angielskim nazywa się to *endianness*, co wynika z książki „Przygody Guliwiera”, gdzie Lilipuci kłócą się czy rozbić jajko od grubej strony, i.e. *big end*, czy cieńszej *little end*, a co za tym idzie dzielą się na *big end-ians* i *little end-ians*. Więcej o tym można poczytać w notatkach z Programowania Niskopoziomowego. :)

gdyż nadal dłuższy od szesnastkowego), omawiany heksadecymalny/szesnastkowy oraz 256-symbolowy (który w pewnym sensie jest używany do reprezentacji liter, jednak to jest dłuższa historia którą tutaj pominiemy).

W związku z powyższym, system szesnastkowy jest często używany do zapisu danych binarnych, gdzie potrzebna jest umiejętność porównania ich „na oko” przez człowieka np. w wartościach skrótów kryptograficznych.

Liczby naturalne i całkowite

Standardową metodą reprezentacji liczb naturalnych w komputerze jest interpretacja ciągu n bitów jako liczba w systemie pozycyjnym o bazie 2.

Mając ciąg k bitów, interpretujemy każdy z nich jako posiadający pewną wartość – wartością liczby jest wtedy suma wartości jej bitów, tj. cyfr. W systemie binarnym i-ty od *prawej* bit reprezentuje wartość 0 jeżeli bit jest symbolem 0, albo 2^{i-i} jeżeli jest on symbolem 1. Jak zostało wspomniane jest to system pozycyjny, a więc zachowuje się on analogicznie do znanego systemu dziesiętnego, tyle że za podstawę systemu przyjmujemy 2 a nie 10.

Przykładowo w 1010, bity mają odpowiednio wartości $2^{4-1} = 8$, 0 (jakby był symbolem 1 to byłoby $2^{3-1} = 4$), $2^{2-1} = 2$ oraz 0, co po sumowaniu daje wartość całego ciągu wynoszącą 10.

Powyższy system pozwala na reprezentację wszystkich wartości z zakresu od 0 do 2^{k-1} przez ciąg bitów długości k. Problem pojawia się jednak przy próbie reprezentacji podzbioru liczb całkowitych, a co z tego wynika liczb ujemnych.

Najbardziej intuicyjny sposób reprezentacji liczb jest poprzez potraktowanie jednego z bitów (najczęściej pierwszego / najbardziej znaczącego) jako bitu znaku – oznacza to, że 1010 oznaczałoby w tym systemie wartość -2 – pierwszy bit od lewej oznacza w tym wypadku, że liczba jest ujemna. Niestety ten sposób reprezentacji ma jedną fundamentalną wadę – sprawia on, że istnieją osobne reprezentacje dla wartości 0 oraz -0, co jest niezgodne z powszechnie obowiązującymi prawami matematyki.

W związku z powyższym, w komputerach korzysta się z alternatywnej reprezentacji, tzw. metody komplementu dwójki (*two's complement*).

Idea jest następująca – liczba jest nadal ujemna wtedy, i tylko wtedy, gdy najbardziej znaczący bit jej reprezentacji wynosi 1. Jeżeli liczba nie jest ujemna, jej wartość jest analogiczna do wartości w systemie dla liczb naturalnych – np. 0101 oznacza 5 w obu systemach. Jeżeli liczba jest ujemna, jej wartość odpowiada negacji wartości otrzymanej poprzez wpierw odwrócenie wszystkich bitów a następnie dodanie 1 do otrzymanego ciągu. Jako przykład, ponieważ ciąg 1010 ma ustawiony najbardziej znaczący bit, reprezentuje on liczbę ujemną – w związku z tym korzystając z algorytmu, wartość jaką dla niego przyjmujemy jest wartość ciągu 0101 plus jeden, co daje minus 0110 w systemie binarnym lub -6 w systemie dziesiętnym.

Metoda komplementu dwójki sprawia, że możemy reprezentować wszystkie możliwe wartości od -2^{k-1} do $2^{k-1} - 1$ dla ciągu długości k bitów. Usuwa ona problem z podwójną reprezentacją zera (chociaż operacja negacji ma przez to dwa punkty stałe – 0 oraz najmniejszą wartość którą możemy reprezentować).

Reprezentacja ta ma kluczową dodatkową zaletę, która czyni ją najbardziej powszechnym sposobem reprezentacji – sprawia ona, że dodawanie liczb całkowitych działa poprawnie podczas korzystania z metody dodawania pozycyjnego dla liczb naturalnych. Dodawanie pozycyjne w systemie binarnym polega na dodawaniu pisemnym od najmniej do najbardziej znaczących bitów. Okazuje się, że jeżeli dodamy w ten sposób dwie liczby zapisane w systemie komplementu dwójki, wynikiem będzie inna reprezentacja liczby w tym systemie odpowiadająca poprawnej wartości.

Własność ta sprawia, że procesory nie potrzebują implementować różnych operacji dodawania na potrzeby liczb naturalnych i całkowitych, co oszczędza miejsca na kości krzemu i ułatwia ich optymalizację.

Podczas gdy wszystkie metody działają dla dowolnej długości ciągu bitów $k \geqslant 1$, standardowo operuje się na wartościach:

- bajtach, tj. 8-bitowych wartościach np. na potrzeby znaków, kolorów w standardowym formacie RGB, oraz danych o nieznanej długości (reprezentujemy je jako ciąg bajtów, jest to podstawowa długość danych);

- 16-bitowych, częściej używanych w dawniejszych procesorach, jednak nadal spotykanych np. w sieci (zakresy portów), mikroprocesorach, oraz przy potrzebie optymalizacji pamięci;
- 32-bitowych, najbardziej powszechnego rozmiaru na wszystkich używanych architekturach procesorów (x86-64, ARM, RISC-V, etc.)
- 64-bitowych, dostępnej we wszystkich nowoczesnych procesorach na potrzeby przechowywania większego zakresu danych (np. czasu).

Większe rozmiary będące potęgami dwójki również są używane np. do identyfikatorów, jednak zazwyczaj nie mają operacji w procesorze implementujących dla nich podstawowe operacje arytmetyczne (można je jednak wprost zaimplementować kilkoma operacjami dla 64-bitowych wartości).

Liczby rzeczywiste

Fun fact: **liczby rzeczywiste nie istnieją** (nie są rzeczywiste, nie mogą Cię skrzywdzić). Ponieważ nasz świat jest (z lokalnego punktu widzenia) skończony, nie jesteśmy w stanie reprezentować wszystkich możliwych wartości zbioru liczb rzeczywistych. Ponieważ jest to jednak użyteczne, reprezentujemy jedynie pewien podzbiór (analogicznie jak dla liczb naturalnych).

Najprostszym sposobem reprezentacji jest reprezentacja o stałoprzecinkowa (*fixed-point*) – reprezentujemy liczbę jako ciąg k bitów, gdzie jako jej wartość traktujemy wartość liczby całkowitej odpowiadającej danemu ciągowi bitów podzieloną przez 2^ℓ. W praktyce oznacza to, że mając reprezentację binarną liczby stawiamy przecinek, tj. kropkę ~~dziesiętną~~ binarną.

Reprezentacja ta ma podstawową zaletę – operuje się na takich liczbach analogicznie do liczb naturalnych. Niestety szybko można zauważyć jej wady – nie jesteśmy w stanie za jej pomocą reprezentować bardzo małych ani dużych liczb, co jest niekorzystne z punktu widzenia przeróżnych możliwych zastosowań.

W związku z tym, najczęstszą reprezentacją liczb rzeczywistych jest reprezentacja zmiennoprzecinkowa, a dokładniej reprezentacja zgodna ze standardem IEEE 754. W owej reprezentacji ciąg k bitów dzielimy na:

- pojedynczy bit znaku
- ℓ bitów *wykładnika*, które specyfikują rząd wielkości
- pozostałe $k - \ell - 1$ bitów *mantysy*.

Jako wartość liczby przyjmujemy $\pm M \cdot 2^E$, gdzie:

- liczba jest nieujemna wtedy i tylko wtedy gdy bit znaku wynosi 0 (analogicznie jak przy reprezentacji liczb całkowitych bez komplementu dwójki; oznacza to, że możemy mieć dwa zera, co dla prostoty akceptujemy);
- liczba M to *mantysa*, czyli liczba zakodowana w końcowej części reprezentacji z przedziału od 1 domkniętego do 2 otwartego (jak w notacji wykładniczej). Zauważmy, że ponieważ początek tej liczby zawsze będzie wynosił 1 z tego założenia, możemy go pominąć i zaoszczędzić jeden bit pamięci – oznacza to, że przyjmujemy $M = 1.m$, gdzie m to wartość otrzymana z reprezentacji binarnej wymienionych wcześniej bitów mantysy (jako liczby naturalnej – zawsze dodatnia).
- liczba E to *wykładnik* zakodowany w środkowej części, wyznacza rząd wielkości wartości. Może być on dodatni lub ujemny, lecz w porównaniu do dotychczasowych reprezentacji liczb całkowitych stosuje się jeszcze inny sposób – zachodzi bowiem $E = e - 2^{\ell-1} + 1$, gdzie e to wartość zakodowana w bitach wykładnika jako liczba naturalna, czyli „przesuwamy” faktyczną wartość o tzw. *bias*.

Owa reprezentacja pozwala nam na zarówno dokładną reprezentację bardzo małych liczb, jak i mniej dokładną reprezentację bardzo dużych liczb – zauważmy, że im większy jest wykładnik, tj. rząd wielkości wartości, tym większa jest różnica pomiędzy dwoma kolejnymi elementami.

W przypadku 32-bitowych liczb zmiennoprzecinkowych (typ `float`), standard przewiduje 8 bitów na rzecz wykładnika; dla 64-bitowych reprezentacji (typ `double`) 11 bitów.

Niestety życie nie jest piękne, więc standard ma jeszcze trochę magicznych sztuczek:

- warto zauważyć, że nie ma obecnie sposobu reprezentacji dla 0 (ponieważ mantysa jest od 1 do 2); w związku z tym, jeżeli wykładnik przyjmuje najmniejszą możliwą wartość (tj. jego bity są

równe 00...00), wtedy jako wartość mantysy przyjmuje się $M = 0.m$ a nie $M = 1.m$; zaletą tego jest możliwość reprezentacji zera oraz znacząco mniejszych liczb (tzw. *subnormals*), wadą jest fakt że z wykluczeniem zera operacje na tych małych liczbach potrafią być znacząco wolniejsze (w związku z tym niektóre procesory mają flagę które zaokrąglają je do zera, co w efekcie przyspiesza patologiczne operacje).

- w ramach standardu, jeżeli wykładnik zawiera największą wartość tj. ma reprezentację 11...11, podczas gdy mantysa jest wyzerowana, to wartość tej reprezentacji traktowana jest jako nieskończoność (oraz minus nieskończoność w zależności od znaku); dodanie tych wartości jest bardzo użyteczne – stanowią one m.in. intuicyjne granice porównań i elementy neutralne operacji min/max (w połączeniu z poprzednim punktem oznacza to, że zakres możliwych wykładników dla „normalnych liczb" jest trochę mniejszy niż wynika z naturalnej interpretacji – dla float'ów jest to od -126 do 127; dla double'i od -1022 do 1023)
- warto zauważyć, że wyróżnienie nieskończoności sprawia, że wartości z wykładnikiem postaci 11...11 i niezerową mantysą są teraz ciężkie do użycia (intuicyjnie powinny być one większe od nieskończoności); według standardu jednak, te wszystkie wartości to tzw. NaN'y czyli *Not a Number*. Owe wartości mają następujące kluczowe własności:
  - powstają na skutek niezdefiniowanych operacji (np. $0/0$, ∞/∞, $\sqrt{-1}$) oraz propagacji $\text{NaN} + x = \text{NaN} \cdot x = \text{NaN} - x = \cdots = \text{NaN}$;
  - wszystkie z operacji $\text{NaN} < x$, $\text{NaN} = X$ i $\text{NaN} > x$ są zawsze fałszywe dla dowolnej wartości x; w szczególności fakt ten oznacza, że $\text{NaN} \neq \text{NaN}$ oraz, że liczby rzeczywiste nie tworzą liniowego porządku (co jest o tyle problematyczne, że w przypadku korzystania ze struktur operujących na porządku, jak zbiory czy mapy za pomocą struktur drzewiastych, trzeba weryfikować brak operacji na kluczach będących NaN'ami w celu uniknięcia błędów w programie).

III.4 Programowanie Niskopoziomowe

III.4.1 Urządzenia zewnętrzne

Opisz mechanizmy komunikacji CPU z urządzeniami zewnętrznymi.

Na początku warto wspomnieć o tym, jak możemy podzielić urządzenia – podstawowy podział możemy zrobić względem prędkości:

- wolne – obsługują dane (tj. przyjmują albo dostarczają) znacznie wolniej od procesora, np. drukarka (przyjmuje), klawiatura (dostarcza).
- średnie – obsługują dane wolniej od procesora, jednak przy podobnym rzędzie wielkości tej szybkości, np. karty sieciowe, wolniejsze dyski, kontrolery USB, karty dźwiękowe.
- szybkie – obsługują dane szybciej od procesora[3], np. karty graficzne, dyski NVMe.

To jest podstawowy podział, który decyduje o tym, z jakiego sposobu komunikacji korzystamy (o czym niżej).

Możemy jeszcze podzielić urządzenia dodatkowo na podstawie tego, jak informują o wykonaniu pewnego zadania (gotowości).

- Niektóre urządzenia mogą być *polled*, tj. zaciągane. Procesor musi bezpośrednio prosić je o raport gotowości (np. co jakiś czas / przed chęcią zlecenia zadania). Zaletą jest prostota i brak potrzeby synchronizacji (czekamy aż będzie gotowe i jest), jednakże wadą jest marnowanie czasu na inne operacje czekając na jeden rezultat (możemy sprawdzać rzadziej, ale wtedy tracimy responsywność).
- Urządzenia *asynchroniczne* informują procesor bezpośrednio o swojej gotowości poprzez przerwania. Procesor dostaje w ten sposób informację, którą może rozpatrzyć albo zakolejkować na później. Oczywistą zaletą jest owa asynchroniczność, gdyż nie musimy czekać na urządzenie i możemy robić inne rzeczy, wadą jest komplikacja ciągu egzekucji (jak dostaniemy przerwanie, to jak nazwa wskazuje przerywamy egzekucję i przechodzimy do rozpatrywania tego faktu – na szczęście możemy je chwilowo wyłączać, wtedy czekają na swoją kolej).
- Urządzenia mogą też wspierać *sampling* – jest to system dosyć podobny do pollingu, jednak różniący się tym, że urządzenie jest zawsze gotowe (ma jakieś dane do przekazania). W pollingu pytamy o gotowość i czekamy aż dostaniemy informację o niej zanim przeczytamy dane, w samplingu po prostu czytamy dane i z nich korzystamy bez blokowania się czekając na urządzenie. Jest to w dużej mierze najlepsze rozwiązanie, ale też niepraktycznie dla znaczącej ilości operacji (oraz jest ono read-only).

Co ważne, obsługa gotowości jest osią prostopadłą lecz zależną do szybkości – w teorii możemy spotkać urządzenia z każdą kombinacją, jednakże niektóre pary naturalnie spotykane są częściej (np. urządzenia wolne najwięcej otrzymują z bycia asynchronicznym, a często ciężko jest dla nich zrobić sensowny sampling).

W celu uwzględnienia zapotrzebowania na wspieranie urządzeń o różnych możliwych specyfikacjach, architektura x86-64 wspiera kilka równorzędnych metod komunikacji z urządzeniami.

- Komunikacja przez porty (port-mapped IO) opiera się na zbiorze portów, odrębnym od przestrzeni adresowej (niekoniecznie przechodzącym przez RAM). W architekturze x86-64 porty adresowane są shortami (jest ich 64k) i komunikuje się przez nie za pomocą instrukcji `in` oraz `out`. Zachowują się podobnie jak pipe'y w systemach operacyjnych; do jednego portu możemy mieć przypisane jedno urządzenie do odczytu i drugie do zapisu – mogą to być różne urządzenia (np. dwa chipy

[3] Technicznie rdzeń CPU ma taktowanie rzędu kilku GHz i jest niemal zawsze szybszy w pojedynczych operacjach, ale urządzenia te generują ogromną przepustowość danych (throughput), która zatkałaby CPU, gdyby musiało samo przenosić każdy bajt.

drukarki), wynika to z faktu, że portów jest dosyć mało w porównaniu do pamięci. Porty wspierają też naturalny polling (wysyłamy ping, dostajemy pong). Z racji, że operacja na danych wymaga kilku operacji procesora (standardowo czytamy/piszemy po bajt na instrukcję), metoda nadaje się tylko do wolnych urządzeń.

- Drugim fundamentalnym sposobem jest wykorzystanie memory-mapped IO. Rejestry sterujące i pamięć urządzenia są tu odwzorowane w tej samej przestrzeni adresowej co RAM. Część adresów fizycznych nie trafia do pamięci, tylko jest przechwytywana przez kontroler pamięci / chipset i kierowana do urządzenia, komunikujemy się więc zwykłymi instrukcjami dostępu do pamięci (`mov` i przyjaciele), przez co możemy względnie szybko przekazywać (w obie strony) dużą ilość danych. Przy MMIO trzeba jednak uważać na kilka rzeczy:
  - zachowanie cache'u dla operacji które przekazujemy przez MMIO jest nietrywialne; czasem możemy po prostu wyłączyć cache, jednak gdy chcemy mieć cache musimy uważać na semantykę aby poprawnie przekazywać informacje.
  - żeby odczyty i zapisy były poprawne kompilator nie może optymalizować ani przestawiać tych dostępów do pamięci (w C możemy użyć użyć `volatile` aby tego uniknąć), a sam procesor może wymagać barier pamięci, bo porządek zapisów do urządzenia bywa istotny.

  Co ważne, MMIO domyślnie oznacza jedynie **mapowanie przestrzeni adresowej**. W zmapowanym zakresie przestrzeni komunikujemy się wprost z urządzeniem, lecz nie dotykamy faktycznej pamięci RAM (tym zajmuje się opisana niżej technika DMA).

Warto zaznaczyć, że np. urządzenia ARM i ogólnie architektury RISC-V nie wspierają port-mapped IO (cały przepływ danych idzie przez pamięć), a przy coraz szybszych urządzeniach odchodzi się od korzystania z portów na rzecz MMIO również po części na x86-64. Jedną z zalet MMIO jest też fakt, że można komunikować się z urządzeniami jako programy nieuprzywilejowane, pod warunkiem że system przydzieli nam odpowiednią stronę pamięci wirtualnej (standardowo porty są dostępne w trybie uprzywilejowanym).

Powyższe dwie metody mówią jak rozmawiamy z urządzeniem, jednak obydwie są synchroniczne i obciążają CPU (przez co nie nadają się do rozmów z szybkimi urządzeniami). Drugą oś, czyli powiadamianie o gotowości oraz odciążenie procesora z samego przepychania danych, realizują w procesorze przerwania (interrupt'y) i DMA (Direct Memory Access).

- Przerwania to jedna z najstarszych metod komunikacji asynchronicznej. Urządzenie, które zakończy swoją pracę (i wróci do trybu gotowości) informuje o tym procesor wysyłając mu żądanie przerwania, tj. Interrupt Request (IRQ). Jak nazwa wskazuje, procesor po otrzymaniu takiego żądania przerywa pracę i przechodzi do jego obsługi, zapisując obecny stan przed skokiem do adresu w pamięci zarejestrowanego (tj. wpisanego w odpowiedniej tablicy bezpośrednio w pamięci procesora) dla obsługi danego przerwania. Kiedyś każde urządzenie wysyłało prośbę o przerwanie do kontrolera przerwań (przez odpowiednie piny), obecnie najczęściej robi się to przez MMIO (kontroler zapisuje pod odpowiednim adresem w pamięci, procesor to wychwytuje i interpretuje). Przerwania nie zawsze będą obsługiwane natychmiast – możemy zapauzować je odpowiednimi instrukcjami oraz zamaskować tylko niektóre z nich (bardzo podobnie jak sygnały w POSIX'ie, które były modelowane właśnie na przerwaniach).
- Direct Memory Access w mniejszym stopniu zajmuje się asynchronicznością (nadal często korzysta z przerwań), jednak rozwiązuje problem obsługi szybkich urządzeń – przy standardowym MMIO to nadal procesor musi wykonywać instrukcje aby zapisywać dane otrzymane od urządzenia do standardowej pamięci RAM. Direct Memory Access pozwala innym urządzeniom na zapisywanie bezpośrednio do pamięci RAM; pamięć ta może być następnie wykorzystana przez procesor w dogodnym czasie („na spokojnie") albo przekazana dalej innym urządzeniom bez konieczności przechodzenia przez procesor – na przykład możemy zlecić karcie graficznej obliczenie, a po otrzymaniu odpowiedzi od razu wysłać je po sieci naszemu koledze. Kiedyś DMA szło przez osobny chip na płycie głównej (urządzenia gadały z nim a on zapisywał do pamięci), obecnie urządzenia typu GPU-/karta sieciowa mają chipy u siebie i bezpośrednio proszą pamięć o odczyt/zapis w dane miejsce (nadal jest jednak chip memory-management który kontroluje, że dostały one „błogosławieństwo" od CPU na zapis w danym miejscu). Podobnie jak z klasycznym MMIO caching jest nietrywialnym zadaniem.

Podsumowując, na podstawie prędkości urządzenia możemy korzystać z następujących metod:

- wolne $\implies$ port-mapped IO (albo proste MMIO), zwykle polling lub przerwania; CPU obsługuje bajt po bajcie, ale skoro urządzenie i tak jest wolne, nie szkoda na to cykli.
- średnie $\implies$ MMIO + przerwania (żeby nie tracić czasu na polling), często DMA do większych transferów.
- szybkie $\implies$ MMIO + DMA + przerwanie na zakończenie obliczenia (gotowość); CPU nie nadążyłby kopiować, więc oddaje transfer urządzeniu.

Analogicznie na podstawie mechanizmu sprawdzania gotowości korzystamy z następujących narzędzi:

- polling $\implies$ sprawdzanie pingami do portów / czytanie statusu z pamięci.
- asynchronicznie $\implies$ przerwania.
- sampling $\implies$ czytanie z pamięci MMIO/DMA.

III.4.2 Odczyt z dysku

Przedstaw szczegółowo sekwencję wydarzeń następujących przy odczycie przez program obszaru pamięci związanego przez `mmap` z plikiem dyskowym.

`mmap` jest wywołaniem systemowym umożliwiającym powiązanie fragmentu pliku z obszarem pamięci wirtualnej procesu. Za jego pomocą realizuje się MMIO (Memory-Mapped Input Output) oraz Memory-Mapped Files (czyli mechanizm, który tu opiszemy). Po wykonaniu `mmap` program nie musi czytać z pliku dyskowego za pomocą syscalla `read`, lecz korzysta z danych tak, jakby znajdowały się w zwykłej pamięci.

Zanim program odczyta dane, wywołuje funkcję `mmap`. System operacyjny (w trybie jądra) tworzy wtedy w strukturach jądra opis nowego obszaru pamięci wirtualnej procesu (w Linux'ie jest to struktura `vm_area_struct`) i łączy ten obszar z deskryptorem pliku dyskowego. W tym momencie system nie ładuje żadnych danych z dysku do RAM-u ani nie tworzy fizycznych wpisów w tablicy stron procesu. Rezerwuje jedynie zakres adresów wirtualnych.

Przy próbie odczytu danych przez program procesor (a dokładniej jednostka MMU) przechwytuje adres wirtualny i próbuje go przetłumaczyć na adres fizyczny. MMU widzi w Tablicy Stron, że strona nie istnieje w pamięci RAM – Bit obecności jest wyzerowany. Zgłaszany jest wyjątek Page Fault. Powoduje on przekazanie kontroli do jądra. Jądro sprawdza strukturę `vm_area_struct` procesu, aby ustalić, czy ten dostęp do pamięci był legalny. Gdyby nie był, to proces zostałby zabity. Ponieważ adres wirtualny znajduje się w obszarze legalnie zmapowanym przez `mmap`, to system wie, że to nie jest błąd programu (Segmentation Fault), ale prawidłowe żądanie odczytania danych z pliku.

Jądro sprawdza, czy żądany fragment pliku nie znajduje się już przypadkiem w pamięci RAM w globalnej pamięci podręcznej systemu (tzw. Page Cache lub Disk Cache), ponieważ inny program mógł go wcześniej otworzyć. Jeśli danych nie ma w RAM-ie, system musi zainicjować operację I/O. Alokuje więc nową ramkę pamięci RAM. Następnie tworzone jest żądanie do sterownika pamięci masowej o skopiowanie odpowiedniego bloku danych z pliku na dysku do nowo przydzielonej ramki w RAM. Odczyt ten jest wolny, więc w międzyczasie jądro zawiesza proces i pozwala procesorowi działać dalej. Po zakończeniu kopiowania danych, kontroler dysku zgłasza sprzętowe przerwanie do procesora. Jądro odbiera sygnał, że dane są już w RAM-ie.

Następnie (po wykonanym kopiowaniu lub zajrzeniu do Disk Cache) jądro aktualizuje wpis w Tablicy Stron procesu i wiąże adres wirtualny z adresem fizycznym odpowiedniej ramki. Bit obecności jest zapalany, a dodatkowo ustalane są odpowiednie prawa dostępu (podane przez proces podczas wywołania `mmap`). Potem następuje aktualizacja danych w cache'u TLB (procesor robi to automatycznie).

Dopiero teraz system operacyjny przełącza procesor z powrotem w tryb użytkownika i pozwala programowi kontynuować swoje działanie. Instrukcja odczytu pamięci wykonuje się po raz drugi. Tym razem MMU odnajduje prawidłowy wpis w Tablicy Stron i pobiera dane bezpośrednio z RAM-u. Przy odczytach dalszych danych nie występuje już Page Fault, bo cała strona znajduje się już w pamięci RAM.

System operacyjny często przewiduje dalsze odczyty. Jeżeli program czyta dane sekwencyjnie, to jądro może z wyprzedzeniem wczytać kolejne strony. Jest to tak zwany mechanizm read-ahead. Znacząco poprawia on wydajność odczytów. Przy braku pamięci RAM system przerzuca jakąś inną ramkę na dysk i robi miejsce dla nowej.

Jeden z argumentów `mmap` przyjmuje jedno z makr `MAP_PRIVATE` lub `MAP_SHARED`. Pierwsze z nich oznacza, że zmiany danych powinny być prywatne dla procesu. Przy pierwszym zapisie następuje mechanizm Copy-on-Write i tworzona jest prywatna dla procesu kopia strony. W przypadku `MAP_SHARED` zmodyfikowane strony są współdzielone i mogą zostać zapisane z powrotem do pliku (w przypadku `MAP_PRIVATE` nie jest to możliwe).

III.4.3 Instrukcje wektorowe

> Omów metody programowania z wykorzystaniem instrukcji wektorowych (SSE).

Podstawą programowania wektorowego jest paradygmat SIMD (Single Instruction, Multiple Data). W przeciwieństwie do tradycyjnego przetwarzania skalarnego (gdzie jedna operacja przetwarza jedną parę liczb), instrukcje wektorowe pozwalają na wykonanie tej samej operacji matematycznej na całym wektorze danych w jednym cyklu procesora.

SSE (Streaming SIMD Extensions) jest rozszerzeniem do architektury x86 wprowadzającym instrukcje SIMD. Zostało wprowadzone przez Intel w 1999 roku. SSE początkowo dodawało 8 nowych rejestrów 128-bitowych `xmm0` do `xmm7`. W architekturze 64-bitowej dodano 8 kolejnych: `xmm8` do `xmm15`. Te rejestry są rozszerzane do 256-bitowych rejestrów `ymm0` do `ymm15`. Istnieją też 64-bitowe rejestry `mmx0` do `mmx7` wprowadzone we wcześniejszym rozszerzeniu MMX. Te rejestry służyły do operacji na liczbach całkowitych, a obliczenia w nich wykonywane były współdzielone z FPU (Floating Point Unit), przez co procesor nie mógł wykonywać operacji na nich i na liczbach zmiennoprzecinkowych jednocześnie. Oryginalne SSE miało tylko instrukcje do operowania na 32-bitowych floatach (liczbach zmiennoprzecinkowych pojedynczej precyzji). W SSE2 dodano wsparcie dla liczb podwójnej precyzji oraz liczb całkowitych (w wersji 64-, 32-, 16- lub 8-bitowej).

Instrukcje SSE dzielą się na instrukcje skalarne (operacja jest wykonywana tylko na jednym elemencie rejestru) oraz pakowane (operacja wykonywana równolegle na wszystkich elementach rejestru). Instrukcje działające na floatach mają zwykle końcówki `ss` (scalar single-precision) lub `ps` (packed single-precision). Instrukcje działające na double'ach to odpowiednio `sd` i `pd`. Instrukcje działające na liczbach całkowitych mają końcówkę odpowiadającą długości liczby (`d` to double word, czyli 32 bity, `q` quad word, czyli 64 bity, `dq` to 128 bitów).

- `movd` przenosi 32 lub 64 bity między ogólnym rejestrem/pamięcią a rejestrem `xmm`, w przypadku 64 bitów przenoszonych do `xmm` następuje zero-extension (wyższe bity są zerowane). Istnieje instrukcja `movq` przenosząca 64 bity, to tylko różnica w składni, tak naprawdę to ta sama instrukcja. Obie te instrukcje operują na liczbach całkowitych.
- `movdqa` przenosi 128 bitów między rejestrami `xmm` lub pamięcią. Dane muszą być wyrównane do 16 bitów, w przeciwnym wypadku zgłaszany jest błąd. Tego wymogu nie ma instrukcja `movdqu`, ale jest wolniejsza. Obie operują na liczbach całkowitych.
- `movdq2q` przenosi dolne 64 bity z rejestru `xmm` do rejestru `mmx`. Na odwrót działa `movq2dq` (występuje zero-extension). Oczywiście są to operacje całkowitoliczbowe (bo `mmx` wspiera tylko takie).
- `movss` przenosi jednego floata między rejestrami `xmm` lub pamięcią (jeśli źródłem jest rejestr `xmm`, to następuje zero-extension, jeśli pamięć, to nie). `movaps` robi to samo dla 4 floatów, wymaga wyrównania, `movups` nie wymaga, ale jest wolniejsze.
- `movlps` przenosi 2 floaty między dolnymi 64 bitami rejestru `xmm` a pamięcią. `movhps` robi to samo dla górnych bitów.
- Te same instrukcje dla double'i mają końcówkę `sd` (`pd`) zamiast `ss` (`ps`).

- Operacje na liczbach zmiennoprzecinkowych mają wersje skalarne i pakowane. Dla dodawania jest to `addss`, `addps`, `addsd`, `addpd`. Podobnie działa odejmowanie `subss`, mnożenie `mulss`, dzielenie `divss`, pierwiastkowanie `sqrtss`, maksimum `maxss` i minimum `minss`. Dla floatów dodatkowo istnieją `rsqrtss` i `rsqrtps`, które szybko przybliżają wartość $1/\sqrt{x}$, co jest bardzo przydatne przy liczeniu rzeczy związanych z grafiką.
- Instrukcje `haddps`, `hsubps` pozwalają dodawać/odejmować sąsiednie wartości z tego samego rejestru. Analogicznie działają `haddpd` i `hsubpd`.
- `cmpss` (i przyjaciele) porównuje liczby zmiennoprzecinkowe, ustawia odpowiednie bity pierwszego rejestru na 1 jeśli zachodzi równość, na 0 w przeciwnym wypadku. `comiss` porównuje pojedyncze floaty (nie ma pakowanego odpowiednika) i informuje o tym w tradycyjny sposób (poprzez ustawienie flag w rFLAGS). Przydaje się to do skoków warunkowych, które korzystają z tego mechanizmu. `ucomiss` dodatkowo może stwierdzić, że wartości nie mają porządku (unordered), co dzieje się, gdy jedna jest NaN-em. Podobnie działa `comisd` i `ucomisd`.
- `shufps` przyjmuje dwa źródła. Możemy ustawić pierwsze 32 bity docelowego rejestru (który jest tym samy, co pierwsze źródło) na dowolną z 4 liczb zapisanych w pierwszym źródle. Podobnie drugie 32 bity. Druga połowa docelowego rejestru robi to samo dla drugiego źródła. To, co zostanie gdzie wsadzone zależy od wartości podanej jako kolejny argument. Analogicznie działa `shufpd`.
- `unpckhps` patrzy na górne 2 liczby z dwóch źródeł i wsadza je na zmianę w rejestr docelowy. Analogicznie `unpcklps`.
- Możemy dokonywać konwersji typów instrukcjami typu `cvtsi2sd` (Integer to Scalar Double), `cvtss2si` (Scalar Single-precision to Integer).
- Mamy operacje bitowe `andps`, `orps`, `xorps`, `andnps` (AND NOT).
- Operacje całkowitoliczbowe możemy wykonywać z przepełnieniem (ignorowanie przepełnienia), np. `paddb` (16 liczb 8-bitowych), `paddw` (liczby 16-bitowe), podobnie `psub?` albo `pmul?`. Możemy też wykonywać je z wysyceniem (mamy największą i najmniejszą liczbę, których nie da się przekroczyć), np. `paddsb` (liczby 8-bitowe ze znakiem), `padduw` (16-bitowe bez znaku) i inne podobne. Mamy też operacje logiczne `por, pand, pandn, pxor`.
- Mieszanie liczb całkowitych działa za pomocą instrukcji typu `pshufd`, która w każdej z czterech liczb rejestru docelowego umieszcza dowolną z czterech liczb ze źródła.

Warto dodać, że rozróżnienie instrukcji przenoszących dane i bitowych na zmiennoprzecinkowe i całkowitoliczbowe nie jest jedynie różnicą semantyczną – operacje zmiennoprzecinkowe są wykonywane przez FPU, a całkowitoliczbowe przez ALU. Współczesne procesory mają fizycznie rozdzielone potoki wykonawcze: potok dla liczb całkowitych (Integer Domain) oraz potok dla liczb zmiennopozycyjnych (Floating-Point Domain). Jeśli załadujemy do pamięci liczby zmiennoprzecinkowe, a potem wykonamy na nich np. operację binarną całkowitoliczbową, to procesor musi fizycznie przesłać te dane z potopu FP do potoku Integer. Takie przełączenie domeny powoduje mikroarchitektoniczne opóźnienie zwane Bypass Delay (może kosztować od 1 do kilku dodatkowych cykli zegara).

Jeśli chcemy korzystać z operacji wektorowych nie pisząc wprost w Assemblerze, to możemy zdać się na automatyczną wektoryzację stosowaną przez kompilatory. Wymaga to odpowiednich flag (na przykład `-O3` lub bardziej konkretne `-ftree-vectorize`). Kod źródłowy jest wtedy czysty i działa na wszystkich platformach, ale nie mamy pewności, że kompilator zrobi dokładnie to, co chcemy. Możemy też korzystać z funkcji wbudowanych dających nam bezpośredni dostęp do instrukcji wektorowych. Wtedy nasz kod zadziała tylko na platformach, które to wspierają.

```
1  #include <immintrin.h> // nagłówek do instrucji SIMD
2
3  void add_vectors(float* a, float* b, float* result) {
4      // ładowanie 4 floatów do rejestrów XMM (wymagane wyrównanie pamięci)
5      // inicjalizacja za pomocą alignas(16) float a[4]
6      __m128 va = _mm_load_ps(a);
7      __m128 vb = _mm_load_ps(b);
8
9      __m128 vres = _mm_add_ps(va, vb);
10     _mm_store_ps(result, vres);
```

```
}
```

III.4.4 Zarządzanie i ochrona pamięci

Opisz sprzętowe mechanizmy zarządzania i ochrony pamięci.

Aby sprawnie zarządzać pamięcią, system operacyjny nie udostępnia działającym procesom adresów w pamięci RAM wprost. Zamiast tego działa mechanizm pamięci wirtualnej. Funkcjonuje on dzięki tak zwanemu **stronicowaniu**. Pamięć (zarówno wirtualna, jak i rzeczywista) jest podzielona na bloki (zwane stronami w przypadku pamięci wirtualnej i ramkami dla pamięci rzeczywistej; zazwyczaj o rozmiarze 4kB). W procesorze istnieje jednostka hardware'owa MMU (Memory Management Unit), która dokonuje mapowania bloków wirtualnych na rzeczywiste. Dzięki temu procesowi wydaje się, że ma dostępną dla siebie całą przestrzeń adresową, a tymczasem procesor mapuje tę przestrzeń na różne miejsca w fizycznym RAM-ie. Dzięki temu programy nie muszą być trzymane w RAM-ie w ciągłym bloku, co daje systemowi operacyjnemu większą swobodę w zarządzaniu pamięcią. Do tego przesuwanie danych procesu przez system operacyjny jest proste, bo nie wymaga to żadnej zmiany po stronie procesu – wystarczy zmienić mapowanie w MMU. Jeśli w RAM-ie brakuje pamięci dla wszystkich działających procesów, to system operacyjny może przenieść część stron tych procesów na dysk. Dzięki temu możliwe jest sprawniejsze radzenie sobie z sytuacjami, gdy pamięć operacyjna jest bardzo silnie wykorzystywana.

Do swojego działania MMU wykorzystuje tak zwaną Tablicę Stron (Page Table), która znajduje się w RAM-ie. Jak jednak wiemy odczyty z RAM-u są bardzo wolne, więc wymóg spojrzenia do niej za każdym razem, gdy tłumaczymy jakiś adres wirtualny na rzeczywisty znacząco spowalniałby system. Dlatego wewnątrz CPU umieszcza się specjalną, bardzo szybką pamięć podręczną zwaną TLB (Translation Lookaside Buffer). TLB przechowuje historię ostatnich translacji adresów (tak zwana asocjacyjna pamięć podręczna). Dzięki temu translacje ostatnio używanych stron odbywają się szybko, wewnątrz CPU (analogiczny mechanizm jak przy pamięci cache).

Współczesne architektury (np. x86-64) wykorzystują wielopoziomowe tablice stron (multi-level page tables). Gdyby tablica stron była płaska, dla 64-bitowej przestrzeni adresowej zajmowałaby ona gigantyczne rozmiary. Dzięki strukturze drzewiastej (w x86-64 stosuje się obecnie 4 lub 5 poziomów tablic), system operacyjny musi alokować pamięć na tablice stron tylko dla tych obszarów pamięci wirtualnej, które proces faktycznie wykorzystuje.

Sprzęt nie tylko tłumaczy adresy, ale też rygorystycznie kontroluje uprawnienia do każdego obszaru pamięci podczas każdego cyklu procesora. Każdy wpis w tablicy stron (oprócz adresu fizycznego) zawiera specjalne flagi (bity) sprawdzane przez sprzęt.

- **Bit obecności (Valid/Invalid bit)**. Określa, czy dana strona znajduje się w pamięci RAM. Jeśli jej tam nie ma, to procesor generuje wyjątek Page Fault. Wtedy kontrolę przejmuje system operacyjny, który decyduje, co zrobić dalej. Jeśli proces zaalokował pamięć, ale system fizycznie jej jeszcze nie przydzielił, to robi to teraz (mechanizm Lazy Allocation). Jeśli pamięć została zaalokowana i potem przeniesiona na dysk, to system przywraca ją do RAM-u. W przypadku nieuprawnionego odwołania do pamięci system zabija program i zgłasza błąd Segmentation Fault (w Linux'ie).
- **Bity praw dostępu (R / W / X)**. Określają, czy stronę można czytać (R), pisać po niej (W) i wykonywać kod na niej umieszczony (X). Aby zapobiec wykonaniu złośliwego kodu znajdującego się w buforze danych bit X jest ustawiony tylko dla stron zawierających kod programu.
- **Bit Użytkownik/Przełożony (User/Supervisor bit)**. Określa, czy strona należy do przestrzeni użytkownika, czy jądra. Zwykłe procesy nie mają dostępu do stron jądra.
- **Bit modyfikacji (Dirty bit)**. Ustawiany przez sprzęt, gdy strona zostanie zmodyfikowana (zapisana). Dzięki temu system operacyjny, przesuwając stronę z RAM-u na dysk, wie, czy musi ją zapisać (bo uległa zmianie), czy może ją po prostu porzucić (bo jej kopia na dysku jest aktualna).
- **Bit dostępu (Accessed bit)**. Ustawiany przez sprzęt przy każdym odczycie lub zapisie do strony. Pomaga systemowi operacyjnemu w implementacji algorytmów zastępowania stron (np. LRU –

Least Recently Used), wskazując, które strony są rzadko używane i można je przenieść na dysk.

Aby oddzielić od siebie kod wykonywany przez użytkownika i jądro system stosuje tak zwane pierścienie bezpieczeństwa (protection/privilege rings). W architekturze x86 istnieją 4 pierścienie (od 0 do 3), ale współczesne systemy wykorzystują tylko dwa z nich – Ring 0 (Tryb Jądra) i Ring 3 (Tryb Użytkownika). Procesor trzyma w odpowiednim rejestrze, w jakim trybie aktualnie się znajduje. W trybie jądra działa kod systemu operacyjnego, ma on nieograniczone uprawnienia i może wykonywać wszystkie instrukcje procesora (w tym na przykład zmieniać dane w MMU). Z kolei w trybie użytkownika działają wszystkie procesy użytkownika. Mają bardzo ograniczone uprawnienia – nie mogą bezpośrednio komunikować się ze sprzętem ani dotykać pamięci jądra. Aby wykonać potrzebną im operację (na przykład alokację pamięci) muszą poprosić o to jądro przez odpowiedni syscall. W przypadku próby wykonania nieuprawnionej operacji procesor zgłasza przerwanie sprzętowe General Protection Fault, które przekazuje kontrolę do systemu operacyjnego. Ten zazwyczaj natychmiastowo zabija winny proces.

III.4.5 Hierarchia pamięci

Czym jest hierarchia pamięci i jakie są konsekwencje jej istnienia dla wydajności programów?

Współczesne jednostki centralne (CPU) wykonują instrukcje znacznie szybciej, niż pamięć RAM jest w stanie dostarczać im dane. Powoduje to występowanie tak zwanego efektu *memory wall* – jeśli CPU potrzebuje do działania jakichś danych, to musi na nie bardzo długo czekać. Aby zmniejszyć narzut z tym związany istnieje tak zwana hierarchia pamięci. Pamięć wyżej w hierarchii ma krótszy czas dostępu (procesor czeka przez mniej cykli), ale co za tym idzie jest droga w produkcji (bo musi być fizycznie blisko procesora, przez co musimy ją upakować na mniejszym obszarze). Pamięć niżej w hierarchii jest wolniejsza, ale za to może być bardzo pojemna.

1. **Rejestry procesora**. Bezpośrednia pamięć operacyjne procesora, przechowuje aktualnie przetwarzane dane. Czas dostępu to 1 cykl procesora, ale za to jest jej bardzo mało, nawet kilkadziesiąt bajtów.
2. **Pamięć podręczna (cache)**. Jest to szybka pamięć, która ma za zadanie ukrywać dużą różnicę szybkości między procesorem a RAM-em. Dzieli się na trzy warstwy, oznaczane L1, L2 i L3. Pamięć L1 jest najszybsza (dostęp w kilka cykli), ale za to najmniejsza (kilkadziesiąt kilobajtów). Jest dedykowana dla każdego rdzenia osobno. Pamięć L2 jest większa (setki kilobajtów lub małe megabajty), ale nieco wolniejsza (kilkanaście cykli). Zwykle również jest dedykowana dla konkretnego rdzenia. Pamięć L3 jest jeszcze większa (kilkanaście do kilkuset megabajtów) i często współdzielona przez rdzenie procesora. Czas dostępu to kilkadziesiąt cykli.
3. **Pamięć operacyjna (RAM)**. Główna pamięć operacyjna, przechowuje programy i dane potrzebne działającym procesom. Znacznie wolniejsza od cache (dostęp rzędu kilkuset cykli), ale o pojemnościach liczonych w gigabajtach.
4. **Pamięć masowa**. Pamięć nieulotna, zazwyczaj w postaci dysków SSD lub HDD. Bardzo wolna w porównaniu do RAM, służąca do trwałego przechowywania danych. Jeśli brakuje nam pamięci w RAM-ie, to część danych z niego może zostać przerzucona tymczasowo na dysk (funkcjonuje tu mechanizm pamięci wirtualnej). Narzut czasowy takiej operacji jest jednak ogromny.

Hierarchia pamięci funkcjonuje dobrze dzięki tak zwanej zasadzie lokalności odwołań (locality of reference). Programy komputerowe zazwyczaj nie uzyskują dostępu do pamięci w sposób całkowicie losowy. Wyróżniamy dwa rodzaje lokalności.

- **Lokalność czasowa (Temporal Locality)**. Jeśli program raz odwołał się do jakiejś komórki pamięci, to istnieje duże prawdopodobieństwo, że w najbliższej przyszłości odwoła się do niej ponownie (np. zmienne w pętlach). Dlatego jeśli program używa jakichś danych, to są one umieszczane w pamięci cache.
- **Lokalność przestrzenna (Spatial Locality)**. Jeśli program odwołał się do danej komórki pamięci, to istnieje duże prawdopodobieństwo, że wkrótce odwoła się do komórek sąsiednich (np. sekwencyjne przetwarzanie tablicy). Dlatego jeśli program używa jakichś danych, to do pamięci

cache przesyłany jest ciągły blok pamięci, do którego używane dane należą, aby przy przyszłym odwołaniu do nich nie trzeba było ponownie odwoływać się do pamięci RAM (co jest wolne).

Ponieważ pamięć RAM jest dużo większa od pamięci cache, adresy w pamięci RAM muszą być w jakiś sposób mapowane na adresy w pamięci cache tak, aby było wiadomo gdzie w cache'u szukać odpowiednich danych. Jednocześnie chcemy, aby adresy leżące blisko siebie mogły być trzymane w cache'u jednocześnie (czyli nie były mapowane na te same adresy). Dlatego miejsce w cache'u, do którego trafia dany adres pamięci RAM odpowiada jego kilku najmniej znaczącym bitom.

W używanych praktycznie architekturach adres w cache'u nie jest jednak pojedynczą jednostką pamięci, a małą tablicą – może trzymać ustaloną, małą liczbę wartości. Tę liczbę nazywamy associativity cache'u (pasmowością). Cache jest też opisywany przez inne parametry, takie jak rozmiar pamięci, rozmiar linii (jednostka, po której pamięć jest skwantowana) oraz czas dostępu (latency – jak szybko odpowiada, liczone w cyklach procesora). Duża pasmowość zwiększa latency oraz powoduje, że jednostka jest droższa (bo musi więcej robić). Dlatego pamięć podręczna blisko procesora ma małą pasmowość, a pozostała trochę większe.

Programista powinien mieć te mechanizmy na uwadze podczas tworzenia programów. Jego celem powinno być optymalizowanie liczby tak zwanych trafień do cache'u (Cache Hit), czyli sytuacji, gdy procesor znajduje dostępne dane w cache'u i nie musi odwoływać się do pamięci RAM. W tym celu w jednym bloku kodu (zwłaszcza wykonywanym dużo razy, na przykład w pętli) staramy się nie odwoływać do wielu bardzo różnych adresów, a do kilku sąsiadujących. Dlatego na przykład tablice (i struktury typu `std::vector`) są szybkie, jeśli przeglądamy ich elementy po kolei i wygrywają pod tym względem na przykład z listami wiązanymi (`std::list`), w których sąsiednie elementy mogą leżeć w bardzo różnych miejscach pamięci.

Rozważmy dwa sposoby iteracji po tablicy dwuwymiarowej w C++.

```
1 for (int i=0; i<n; i++) for (int j=0; j<n; j++) sum += A[i][j];
2
3 for (int i=0; i<n; i++) for (int j=0; j<n; j++) sum += A[j][i];
```

W pierwszym z nich wewnętrzna pętla odwołuje się do adresów leżących blisko siebie (notacja `A[i][j]` oznacza `*(A + i*n + j)`). Dlatego pierwszy kod wykona się znacznie szybciej niż drugi. Efekt ten jest spotęgowany, gdy n jest potęgą dwójki. Wtedy bowiem kolejne elementy, do których odwołujemy się w drugiej iteracji mają te same ostatnie bity adresów, a co za tym idzie nawzajem wyrzucają się z cache'u. W rezultacie cały czas musimy korzystać z pamięci RAM, co powoduje znaczące spowolnienie.

III.4.6 Architektura x86

Omów architekturę nowych procesorów rodziny x86 na podstawie stosowanych do nich określeń: deeply pipelined, speculative, out-of-order, superscalar, complex instruction set computer.

Warto przypomnieć sobie jak działa wykonywanie instrukcji w procesorze:

1. fetch – ładujemy instrukcję z pamięci/cache'u
2. decode – rozpoznajemy instrukcje i argumenty
3. prepare – przygotowujemy argumenty (np. ładujemy z pamięci / z rejestrów) do obliczenia
4. execute – wykonujemy obliczenie (procesor przekazuje je do odpowiedniej jednostki)
5. commit – zapisujemy wynik do rejestru/pamięci (jakoś w sposób widoczny dla innych instrukcji)

Celowo omówimy cechy w trochę innej kolejności.

- Deeply-pipelined – pipelining polega na wykonywaniu wcześniej wymienionych instrukcji równolegle. W najprostszy sposób moglibyśmy stwierdzić, że wykonujemy wszystkie kroki dla jednej instrukcji przez rozpoczęciem kolejnej – zauważmy jednak, że każdy z kroków wykonywany jest przez inną część procesora; w związku z tym w momencie gdy załadowaliśmy jedną operację i

przekazaliśmy do zdekodowania, możemy zacząć ładować drugą. Analogicznie, po zdekodowaniu operacji i przekazaniu argumentów do odpowiedniej jednostki obliczenowej (np. ALU – Arithmetic Logic Unit odpowiedzialnej za podstawowe operacje arytmetyczne), możemy zacząć dekodować kolejną przez zakończeniem obliczenia.

W ten sposób jesteśmy w stanie znacząco przyspieszyć procesor, jako że każda jego część wykonuje pewne obliczenia w danej chwili (klasyczne zastosowanie zrównoleglenia niezależnych obliczeń).

Współczesne procesory x86 mają bardzo długie pipeline'y (często od 14 do ponad 20 etapów, a nie tylko wymienione wyżej 5). Im dłuższy pipeline, tym prostsze są operacje na każdym etapie, co pozwala na osiągnięcie wyższych częstotliwości taktowania.

- Out-of-order – zauważmy, że jeżeli dwie instrukcje operują na rozłącznym zbiorze rejestrów (bądź jedna z nich na pamięci), to z punktu widzenia kolejnych instrukcji nie interesuje nas szczególnie w jakiej kolejności zostaną wykonane; założenie o rozłączności wynika z tego, że operacje muszą być niezależne, gdyż nie możemy skorzystać z wyniku obliczenia przed jego wykonaniem.

  Ponieważ nowoczesne procesory dekodują instrukcję bardzo szybko (szybciej niż wykonanie), jesteśmy w stanie zamienić kolejność ich wykonania w ramach pipeliningu.

  Jako przykład, jeżeli mamy instrukcje wykonywane na tym samym podzespole obliczeń A, B, C, D zajmujące odpowiednio $20, 5, 10, 25$ cykli, gdzie C zależy od A, D zależy od B i mamy dostępne 2 moduły obliczeniowe, to wykonanie instrukcji w kolejności A, B, D, C (gdzie zaczynamy D bezpośrednio po A nie czekając na C) zajmuje 30 cykli a nie 45.

  Mechanizm ten jest dodatkowo wspierany przez *register renaming* – zauważmy, że dwie operacje `add ecx, eax` i `mov eax, edx` operują na jednym rejestrze, więc tradycyjnie nie moglibyśmy ich wykonać w innym porządku. W praktyce jednak nowoczesne procesory posiadają więcej rejestrów general-purpose (np. 64) niż jest „nazwanych" w kodzie maszynowym/asemblerze (16). Oznacza to, że procesor może wykonać trick, gdzie prawdziwy ciąg operacji to `add ecx, old_eax; eax oznacza od teraz new_eax; mov new_eax, edx`, co pozwala uniezależnić wystąpienie rejestru `eax` w instrukcjach i tym samym wykonać obliczenia w dowolnej kolejności, w tym równolegle (patrz punkt niżej).

- Superscalar – procesor może wykonywać kilka instrukcji naraz; jako że w nowoczesnych procesorach istnieje kilka równoległych podmodułów wykonujących ten sam typ obliczeń (np. ALU – Arithmetic Logic Unit który wykonuje podstawowe operacje typu dodawanie, porównanie, etc.), nowoczesne procesory widząc ciąg kilku instrukcji `add` (ale nie tylko, bo różnorodnych też) rozpoczynają i delegują pracę kolejnej instrukcji przed zakończeniem poprzedniej, co efektywnie zrównolegla główną część pracy (właściwe obliczenie) i naturalnie przyspiesza wykonywanie (np. gdy pięć niezależnych instrukcji wykonuje się w czasie jednej).

  Łączy się to w dużej mierze z poprzednią ideą `out-of-order execution` – skoro wiemy, że porządek na operacjach jest ważny tylko z punktu widzenia grafu relacji między rejestrami/pamięcią, możemy wykonywać wiele operacji naraz i korzystać z ich rezultatów jak tylko jedna z nich się skończy (pierwsze procesory będące jedynie superskalarne lecz nie out-of-order wykonywały kilka instrukcji ale czekały jak wszystkie się skończą i zapiszą swój wynik w odpowiednim porządku).

- Speculative – napotykamy na skok warunkowy w ramach egzekucji programu, co możemy zrobić? Najbardziej oczywistym rozwiązaniem jest poczekanie aż instrukcja od której zależy skok dojdzie do fazy `commit` i na jej podstawie kontynuacja egzekucji po skoku – zauważmy jednak, że takie podejście oznacza, że każdy skok zatrzymuje pipeline oraz ogranicza zastosowanie innych wcześniej opisanych metod optymalizacji (out-of-order, superscalar). Zamiast tego nowoczesne procesory przewidują (spekulują) najbardziej prawdopodobny rezultat skoku i na jego podstawie rozpoczynają wykonanie kolejnych instrukcji; jeżeli procesor dobrze zgadł, może zcommitować wyniki zakolejkowanych operacji, jeżeli źle, procesor nie commituje wyników operacji (odrzuca je) i zaczyna wykonywać te z poprawnej ścieżki egzekucji.

  Mechanizm przewidywania skoków nazywamy *branch prediction* (nie jest to to samo – mechanizm speculative execution wie co ma spekulować przez branch prediction, ale w teorii może spekulować nie tylko skoki). Ponieważ w ramach programowania często wykorzystuje się np. pętle, gdzie instrukcje warunkowe w kolejnych iteracjach zachowują się podobnie, taki sposób przewidywania

okazuje się bardzo przydatny. W najprostszym modelu procesor przewiduje jak zadziała skok poprzez zapisanie jak skoczył w ramach ostatniej instrukcji ostatnie kilka razy – może on np. trzymać 2-bitową wartość bez znaku która rośnie jeżeli skoczymy i maleje jeżeli nie (jeżeli wychodzimy poza zakres wartości to jej nie zmieniamy zamiast overflow'ować) – przewidywanie procesora jest następnie wybierane na podstawie górnego bitu tej wartości.

Warto mieć mechanizm branch prediction na względzie przy pisaniu głęboko zagnieżdżonych wyrażeń warunkowych (if'ów), ponieważ jeżeli wszystkie ścieżki kodu są względnie często odwiedzane, to jest to bardzo niekorzystne z punktu widzenia tej optymalizacji.

- CISC (complex instruction set computer) – w kontraście do RISC (restricted instruction set computer) jest jednym z podejść do przyspieszenia często używanych operacji w ramach procesora.

  Każdy współczesny procesor musi wspierać pewnien zbiór podstawowych operacji, np. operacje na liczbach całkowitych (dodawanie/odejmowanie/mnożenie), komunikację z urządzeniami zewnętrznymi, zarządzanie pamięcią (`mov`) oraz instrukcje kontrolne (skoki i skoki warunkowe). Podstawowa różnica między procesorami jest taka, jak podchodzą do bardziej złożonych operacji – czyli takich, które można zaimplementować za pomocą kilku wyżej wymienionych podstawowych operacji.

  Procesory CISC (np. x86-64) posiadają bardzo bogaty zestaw instrukcji, które pozwalają na szybkie wykonywanie niektórych skomplikowanych instrukcji korzystając z rozwiązania hardware'owego. Procesory RISC (np. ARM oraz RISC-V) podchodzą do tego inaczej – podczas gdy też czasem mają rozszerzenia, ich zbiór instrukcji jest znacząco mniejszy; w zamian za to mogą one jednak pozwolić sobie (w związku z zaoszczędzonym czasem R&D oraz miejscem na krzemie) na lepszą optymalizację istniejących instrukcji oraz wcześniej wymienionych metod (pipelining/superscalar).

  Podstawową różnicą w podejściu obecnych modeli jest podejście do dostępu pamięci – w x86 instrukcje typu `add/sub/inc` mają wiele typów adresacji, co pozwala im dodawać wartości zarówno w rejestrach jak i bezpośrednio w pamięci (nawet z możliwym przesunięciem). W porównaniu, w modelach RISC standardowo operuje się jedynie na rejestrach, a wszystkie operacje na pamięci przechodzą przez `load/store` (odpowiedniki `mov` z x86 – to trochę kompiluje sprawę np. przy operacjach atomicznych, ale są na to rozwiązania).

  Inne przykłady instrukcji w CISC niewystępujących w RISC to `loop` (które trywialnie tłumaczy się na odjęcie wartości i skok warunkowy) czy prefiksy `rep`, które wpływają na niektóre specyficzne instrukcje do pracy nad stringami.

  Ciekawostka: jeżeli zignorujemy komunikację z urządzeniami (możemy ją realizować przez shared memory), to w praktyce dla działającego komputera wystarczy jedna instrukcja, czyli tak zwane OISC (one instruction set computer); najbardziej znanym przykładem tego jest `subleq A B C`, która odpowiada następującemu ciągowi wyrażeń (m to nasza tablica reprezentująca pamięć): $m[B] = m[B] - m[A]; \text{if } m[B] \leq 0 \text{ then jump to } C$.

III.5 Sieci Komputerowe

III.5.1 Warstwy sieci

Warstwy. Jakie są konsekwencje warstwowej konstrukcji technologii sieciowych? Opisz jak jest fizycznie realizowana (np. w sieci 1000BASE-T) ramka Ethernet przesyłająca pakiet IPv4 zawierający fragment strumienia TCP podczas pobierania pliku z serwera HTTP.

Model ISO-OSI definiuje następujący podział sieci na warstwy.

1. warstwa fizyczna
2. warstwa łącza danych (Ethernet, WiFi, bezpośrednie połączenia)
3. warstwa sieci (IP, duże sieci)
4. warstwa transportowa (TCP, UDP)
5. warstwa sesji
6. warstwa prezentacji
7. warstwa aplikacji

W modelu TCP/IP stosujemy trochę mniej dokładny podział na warstwy.

1. warstwa dostępu do sieci
2. warstwa internetu
3. warstwa transportowa
4. warstwa aplikacji

Dalej odwołując się do warstw będziemy mieli na myśli warstwy w modelu ISO-OSI.

Modelowanie sieci za pomocą warstw opiera się na zasadzie abstrakcji i enkapsulacji. Każda warstwa świadczy usługi warstwie wyższej, nie interesując się tym, jak ta warstwa działa, ani w jaki sposób dane zostaną przesłane przez warstwy niższe. Dzięki temu stosowane technologie są modularne i wymienne. Można zmienić medium transmisyjne z kabla miedzianego (Ethernet) na fale radiowe (Wi-Fi) w warstwie 1 i 2, a przeglądarka internetowa (warstwa aplikacji) oraz system operacyjny (warstwa transportowa) w ogóle tego nie zauważą. Upraszcza to też implementacje, bo na przykład w HTTP nie musimy programować mechanizmów korekcji błędów radiowych czy trasowania pakietów – polega się na gotowych usługach TCP i IP. Każda warstwa rozwiązuje inne problemy i korzysta z działania pozostałych warstw. HTTP odpowiada za przesyłanie dokumentów, TCP zapewnia niezawodną transmisję, IP odpowiada za routing w dużych sieciach, Ethernet za komunikację lokalną, a warstwa fizyczna za bezpośredni przesył sygnału.

Wadą takiego podejścia jest narzut wynikający z tego, że każda warstwa dokłada swój własny nagłówek. Do tego czas przetwarzania jednego pakietu jest większy ze względu na konieczność wielokrotnego przetworzenia go przez protokoły działające na kolejnych warstwach sieci.

Podczas przesyłania danych przez sieć następuje enkapsulacja – dane z warstwy wyższej stają się ładunkiem (payload) dla warstwy niższej.

1. Protokół HTTP w warstwie aplikacji generuje pobierany plik (lub jego fragment), tworzy odpowiedź HTTP składającą się z nagłówka i przesyłanych danych.
2. Strumień HTTP jest dzielony na mniejsze części, które mieszczą się w Maximum Segment Size protokołu TCP. Do każdej porcji doklejany jest nagłówek TCP, zawierający m.in. port źródłowy (80 lub 443), port docelowy klienta oraz numery SEQ i ACK.

3. Segment TCP trafia do warstwy IP, która dokleja nagłówek IP. Zawiera on adres IP serwera, adres IP klienta oraz flagę określającą, że ładunkiem jest protokół TCP.
4. Pakiet IP trafia do sterownika karty sieciowej. Dokładany jest nagłówek Ethernet zawierający adres MAC źródła oraz docelowy adres MAC (następnego routera). Na koniec dokładane jest pole FCS (Frame Check Sequence), które zawiera kod CRC-32.
5. Gdy ramka Ethernet jest gotowa, karta sieciowa zamienia bity na fizyczne sygnały elektryczne, które zostaną wysłane przez kabel. Zanim karta wyśle właściwą ramkę, wysyła sekwencję synchronizacyjną znaną jako preambuła. Jest to ciąg naprzemiennych jedynek i zer przez 8 bajtów, który kończy się dwukrotnym wystąpieniem 1. Pozwala to karcie odbiorcy „wstroić się" w zegar nadawcy, tak by był w stanie odróżnić na przykład ciąg 11 od pojedynczego 1.
6. Następuje scrambling – dane są xorowane z pewną wartością ustaloną tak, aby uniknąć długich ciągów samych jedynek lub zer w wyniku, co zapobiega problemowi DC-bias, który polega na tym, że jeśli odbiornik otrzyma za dużo jedynek (sygnałów z energią) naraz, to może to zmienić stan jego odczytu. Do tego jeśli przesyłane sygnały zmieniają się, to łatwiej jest zachować synchronizację z nadawcą. Wartość, z którą xorujemy, jest generowana przez rejestr LFSR (Linear Feedback Shift Register) w sposób powtarzalny i taki, że odbiorca jest w stanie odtworzyć pierwotną wiadomość (w momencie wysyłania preambuły rejestry nadawcy i odbiorcy synchronizują się).
7. Następnie grupy bitów są mapowane na 5-poziomowe napięcia elektryczne (kodowanie PAM-5; Pulse Amplitude Modulation). Każdy stan napięcia przenosi 2 bity informacji równocześnie. Stosujemy do tego TCM (Trellis Coded Modulation), które pozwala nam wykorzystać fakt, że mamy 5 możliwych wartości sygnałów do naprawiania zniekształconych sygnałów.
8. Sygnał jest przesyłany jednocześnie przez wszystkie 4 pary skrętki w kablu. Sygnał jest jednocześnie nadawany i odbierany (full-duplex). Patrząc na odczyt z kabla i wiedząc, co nadawaliśmy, możemy odtworzyć wysłany nam sygnał (tak zwany mechanizm echo cancellation). To pozwala nam przesyłać sygnał z prędkością 1Gb na sekundę.

III.5.2 Błędy transmisji

> Błędy transmisji. Podaj przykłady technologii sieciowych wykrywających/korygujących błędy komunikacji. Z jakich algorytmów korzystają? W jaki sposób protokół TCP wykrywa błędy transmisji i jak na nie reaguje?

W zależności od warstwy modelu OSI i medium transmisyjnego (kable, światłowody, fale radiowe) stosuje się dwie strategie obsługi błędów.

- **EDC (Error Detection Codes)** służą do wykrywania błędów. Po wykryciu anomalii następuje odrzucenie ramki/pakietu z prośbą o retransmisję. Stosuje się je zwykle w systemach przewodowych, gdzie błędy zdarzają się rzadko.
- **FEC (Forward Error Correction)** polega na naprawianiu uszkodzonych bitów na podstawie nadmiarowych informacji bez ponownego przesyłania danych. Tę strategię stosuje się w systemach bezprzewodowych, gdzie błędy i zakłócenia są częste.

W Ethernecie (warstwa łącza danych) stosuje się algorytm CRC-32 (Cyclic Redundancy Check), który służy do wykrywania błędów. Ramka Ethernet jest traktowana jak wielomian nad $\{0, 1\}$. Dzielimy ją przez pewien ustalony wielomian (w przypadku CRC-32 jest to wielomian stopnia 32), a otrzymaną resztę umieszczamy w polu FCS (Frame Check Sequence) na końcu ramki. Odbiorca wykonuje to samo działanie; jeśli reszta się nie zgadza, ramka jest bezpowrotnie odrzucana. Ethernet nie naprawia ramek – ponowne przesłanie to zadanie wyższych warstw (np. TCP).

W sieciach komórkowych i WiFi stosuje się do tego tak zwane kody LDPC (Low-density Parity Check), które służą nie tylko do wykrywania błędów, ale też do naprawiania ich. W dużym skrócie dzielą one strumień danych na bloki o określonej długości, a następnie generują bity parzystości (nadmiarowe) na

podstawie operacji XOR. Dekodowanie nie operuje na twardych zerach i jedynkach, ale na prawdopodobieństwach. Proces dekodowania tworzy dwudzielny graf zwany grafem Tannera. Po jednej stronie są bity wiadomości i ich wartości (prawdopodobieństwa wartości), a po prawej bity kontrolne. W grafie istnieje krawędź, jeśli bit uczestniczył w tworzeniu bitu kontrolnego. Bity wiadomości przekazują dane o swojej domniemanej wartości do bitów kontrolnych, a bity kontrolne decydują, czy przekazane im wartości są takie, jakie powinny być. Informacja zwrotna jest przekazana do bitów wiadomości, które aktualizują swoje wartości na tej podstawie. Po pewnej liczbie takich iteracji znajdziemy się w sytuacji, w której bity kontrolne się zgadzają.

W nagłówku IP umieszczana jest checksuma nagłówka. Checksuma ma 16 bitów. Obliczana jest poprzez zsumowanie wszystkich 16-bitowych fragmentów nagłówka ze sobą (zakładamy tu, że fragment z checksumą jest wyzerowany). Jeśli wyjdzie nam więcej niż 16-bitów wyniku, to dodajemy nadmiarowe bity do najmniej znaczących 16. W ten sposób dostajemy mniejszą liczbę i powtarzamy tę operację, aż dostaniemy liczbę 16-bitową. Następnie flipujemy bity tak uzyskanej liczby. Jeśli dla otrzymanego nagłówka IP policzymy checksumę, to powinno nam wyjść 0. Jeśli tak nie jest, to pakiet jest odrzucany.

W wersji IPv6 nie występuje checksuma – uznano, że detekcja błędów stosowana w protokołach niższego i wyższego poziomu jest wystarczająca. Zrobiono to w celu przyspieszenia routingu – każdy router IPv4 przy zmianie pola TTL (Time to Live) musiał przeliczać sumę kontrolną na nowo. W IPv6 założono, że warstwa 2 (np. Ethernet z CRC) oraz warstwa 4 (TCP/UDP z checksumą) zapewniają wystarczającą kontrolę błędów.

W UDP checksuma jest obliczana w podobny sposób jak w IP. Obejmuje pseudo-nagłówek IP (część danych z nagłówka IP, używana tylko do celów liczenia tej checksumy), nagłówek UDP oraz dane. Jeśli stosujemy IPv4, to checksuma jest opcjonalna. Przy IPv6 jest obowiązkowa (bo IPv6 nie ma swojej). Datagramy o niewłaściwej checksumie są odrzucane.

W TCP checksuma również jest obliczana tak samo, na podstawie pseudo-nagłówka IP, nagłówka TCP i danych. Jeśli checksuma jest niepoprawna, to pakiet jest odrzucany.

TCP wykrywa błędy transmisji poprzez mechanizm potwierdzeń – odbiorca wysyła nadawcy wiadomości potwierdzające, że otrzymał wiadomość (i do jakiego miejsca ją otrzymał). Jeśli nadawca przez długi czas nie dostanie potwierdzenia, to zakłada, że pakiet został zgubiony (lub odrzucony jako błędny) i przesyła go raz jeszcze.

Naprawa błędów w TCP opiera się na strategii ARQ (Automatic Repeat reQuest) i realizowana jest poprzez mechanizmy RTO (Retransmission Timeout) i Fast Retransmit (omówione dokładniej przy innym pytaniu). W nowoczesnym TCP działa też rozszerzenie **SACK (Selective Acknowledgement)**. W klasycznym TCP, jeśli nadawca wysłał pakiety 1-10, a zaginęły pakiety 2 i 5, to skumulowane ACK utknęło na żądaniu pakietu 2. Nadawca musiałby retransmitować albo wszystkie pakiety od 2 do 10, albo retransmitować pakiet 2, czekać na ACK żądające pakietu 5 i dopiero wtedy słać resztę. Rozszerzenie SACK (obecnie powszechnie stosowane) pozwala odbiorcy w specjalnym polu nagłówka TCP przekazać informację w postaci jawnych zakresów: „Otrzymałem pakiety 3-4 oraz 6-10, ale wciąż brakuje mi 2 i 5". Dzięki temu nadawca ponownie wysyła tylko te pakiety, które faktycznie zaginęły

III.5.3 Protokół IP

IP. Opisz schemat działania tablic trasowania pakietów IP na przykładzie systemu Linux. Dlaczego tablice trasowania w ogóle mają szanse działać?

Protokół IP tworzy warstwę internetu – za jego pomocą budujemy bardzo duże sieci. Dwa komputery połączone ze sobą bezpośrednio (np. przez Ethernet, WiFi) przesyłają do siebie pakiety IP za pomocą tych niskopoziomowych technologii. Aby przesłać pakiet IP między komputerami, które nie są bezpośrednio połączone, trzeba się będzie bardziej wysilić.

Pakiet IP składa się z przesyłanych danych oraz tak zwanego nagłówka IP. Zawiera on wiele ważnych informacji (takich jak wersja protokołu, checksuma, długość pakietu), ale przede wszystkim adresy IP nadawcy i adresata wiadomości. Adresy IP to 4-bajtowe liczby (w wersji IPv4, współcześnie stosuje się

też IPv6 zajmujące 16 bajtów), które jednoznacznie identyfikują maszyny, którym są przypisane (stąd potrzeba stosowanie IPv6; 4 bajty to za mało, by ponumerować wszystkie komputery na świecie).

Nie jest możliwe, aby dany komputer miał fizyczne połączenie z każdym innym komputerem, z którym potencjalnie chciałby się komunikować. Dlatego przesyłanie pakietów IP odbywa się za pośrednictwem innych maszyn – nadawca przesyła pakiet IP do pobliskiej maszyny, o której spodziewa się, że będzie mogła dostarczyć jego wiadomość do adresata, ta maszyna przekazuje ten pakiet dalej, do jakiegoś swojego sąsiada i tak dalej, aż pakiet dotrze do celu. Skąd jednak maszyny wiedzą, do kogo powinny przesyłać pakiety, aby zbliżyły się do odpowiedniego adresata? Otóż adresy IP nie są losowe. Przydziela się je w taki sposób, aby komputery o tym samym prefiksie adresu IP były położone blisko siebie (im dłuższy wspólny prefiks, tym większa bliskość).

Pomysł na efektywny routing (czyli znajdowanie ścieżek w sieci) jest taki, że tworzymy hierarchię sieci – dzielimy ją na fragmenty, które odpowiadają prefixom numerów IP. Znacznie ogranicza to rozmiar problemu – jeśli na przykład dzielimy po bajtach, to na najwyższym poziomie mamy 256 podsieci, a przy takiej liczbie już da się zapewnić, aby w każdej z nich znajdowała się centralna maszyna, która jest w stanie skomunikować się z centralnymi maszynami innych podsieci. Następnie te podsieci dzielimy na jeszcze mniejsze sieci w ten sam sposób. Po ustaleniu takiej struktury jeśli maszyna chce przesłać pakiet IP do jakiegoś adresata, który nie jest w jej lokalnej sieci, to wysyła go do odpowiedniej maszyny, która odpowiada za komunikację z zewnętrznymi sieciami. Ta maszyna z kolei albo już wie, do której lokalnej sieci przesłać pakiet, albo przesyła go jeszcze wyżej w hierarchii sieci. W pewnym momencie pakiet trafi na tak wysoki poziom hierarchii, że aktualnie posiadająca go maszyna będzie mogła przesłać go w dół hierarchii, do jakiejś mniejszej sieci, w której wie, że adresat się w niej znajduje. Następnie pakiet schodzi w głąb hierarchii, aż ostatecznie trafia do adresata.

W praktyce nie stosuje się zaproponowanego przez nas podziału adresów po bajtach (czyli hierarchii o 4 poziomach). Zamiast tego istnieje pojęcie Systemu Automonicznego (Autonomous System, AS) – sporego zbioru adresów IP, który jest kontrolowanych przez jedną organizację (lub kilka powiązanych), która ma niezależny dostęp do wielu sieci. Typowo takimi organizacjami są na przykład dostawcy usług internetowych (Internet Service Provider, ISP).

Wewnątrz jednego AS routing przebiega w sposób zdefiniowany przez właściciela AS. Ogólna idea jest taka, że komputery w sieci starają się znaleźć jak najkrótszą ścieżkę, która doprowadzi dany pakiet od celu (więcej szczegółów za moment). Pomiędzy ASami sposób routingu jest jasno określony przez odpowiednie protokoły routingu.

Aby móc znajdować efektywne ścieżki w sieci, maszyny przechowują tak zwane tablice trasowania (routingu), które mówią im, jakim maszynom mają przekazywać pakiety o danych adresatach. Tablica routingu w systemie Linux może wyglądać na przykład tak.

```
default via 195.114.0.1 dev eth0 proto static metric 10
default via 10.0.0.1 dev wlan0 proto dhcp metric 100
10.0.0.0/24 dev wlan0 proto kernel scope link src 10.0.0.15 metric 100
192.168.1.0/24 dev eth1 proto kernel scope link src 192.168.1.1
192.168.1.0/26 dev eth2 proto kernel scope link src 192.168.1.65
192.168.2.0/24 via 192.168.1.2 dev eth1 proto static
10.8.0.0/24 dev tun0 proto static
195.114.0.0/30 dev eth0 proto kernel scope link src 195.114.0.2
```

Wartość po ukośniku oznacza maskę podsieci – to, jak długi prefiks danego adresu jest ustalony. Na przykład `10.0.0.0/24` oznacza, że dana reguła trasowania obowiązuje dla adresów, których pierwsze 24 bity pokrywają się z podanymi. Ogólna zasada jest taka, że zawsze wybierana jest najbardziej konkretna reguła (czyli ta o największej liczbie ustalonych bitów), do której pasuje IP adresata. W przypadku remisów decyduje spodziewany koszt połączenia (mniejsza wartość pola `metric`). W przypadku braku konkretnej reguły pasującej do adresata wykonywana jest jedna z reguł `default`.

Reguły w tablicach trasowania mogą być ustalane na sztywno przez administratora sieci lub tworzone dynamicznie za pomocą odpowiedniego protokołu trasowania (co ma sporo sensu, bo w końcu maszyny potrafią dołączać się do sieci i odłączać, a istniejące połączenia mogą zmieniać swoją przepustowość). Wewnątrz jednego AS stosuje się protokoły z rodziny Interior Gateway Protocols (IGP). Jednym z nich jest Routing Information Protocol (RIP). Opiera się on na wzajemnej współpracy maszyn, które

komunikują sobie, jakie połączenia widzą. Każda z nich tworzy distance vector – zapamiętuje najmniejsze odległości do wszystkich innych. W każdym kroku maszyna przesyła swój distance vector do sąsiadów, na podstawie otrzymanej informacji poznawane są ścieżki do nowych maszyn (lub krótsze do już wcześniej widocznych), które zapamiętujemy. Innym przykładem IGP jest Open Shortest Path First (OSPF), który odtwarza całą topologię sieci (nie tylko odległości; tak zwany Link State) poprzez przesyłanie jej sobie między sąsiadami. Oba te protokoły wymagają nakładu czasu i pamięci. Dlatego właśnie są stosowane w relatywnie małych sieciach – wewnątrz jednego AS.

Do komunikacji pomiędzy ASami stosuje się Exterior Gateway Protocols. Przykładami takich protokołów są Exterior Gateway Protocol (EGP; przestarzały) oraz Border Gateway Protocol (BGP; następca EGP). BGP jest dominującą strategią trasowania wykorzystywaną obecnie w Internecie.

III.5.4 Protokół TCP

TCP. Opisz technikę Sliding Window. Opisz w jaki sposób algorytmy TCP sterują prędkością transmisji. Opisz API do obsługi połączeń TCP w bibliotece standardowej (moduł socket) w języku Python.

TCP (Transmission Control Protocol) jest protokołem transportowym zapewniającym niezawodność transmisji, zachowanie kolejności danych, wykrywanie utraty pakietów, retransmisję, kontrolę przepływu (flow control), kontrolę przeciążenia (congestion control).

TCP to próba uzyskania niezawodnego połączenia. Chcemy nie tracić przekazywanych pakietów – stosujemy potwierdzenia transmisji i retransmisje w przypadku ich braku. Pakiety TCP mają pole SEQ (sequence number) oznaczające numer pierwszego bajtu aktualnego pakietu (bajty w TCP są numerowane) i ACK (accept) oznaczające numer bajtu, na którego przyjęcie jesteśmy gotowi (czyli przyjęliśmy bajt o numerze o jeden mniejszym).

W najprostszej wersji możemy po wysłaniu pakietu czekać na potwierdzenie i dopiero po nim wysyłać kolejny. To jednak jest wolne i słabo wykorzystuje sieć, szczególnie gdy opóźnienia są duże. Dlatego w TCP istnieje pojęcie Sliding Window – nadawca może wysłać kilka pakietów bez oczekiwania na ACK. Ma jednak ograniczenie na liczbę bajtów wysłanych bez potwierdzenia. Gdy dostanie potwierdzenie przyjścia pierwszych z wysyłanych pakietów, to wie, że może wysyłać kolejne – okno numerów SEQ, które może wysłać przesuwa się (stąd nazwa).

Pojęcie Sliding Window umożliwia zarówno kontrolę przepływu, jak i kontrolę przeciążenia. Kontrola przepływu służy do dopasowania szybkości nadawcy do możliwości odbiorcy. Odbiorca informuje nadawcę, jak wiele bajtów może zostać wysłanych bez potwierdzenia (tak zwana wartość Advertised Window). W ten sposób jeśli odbiorca nie nadąża z procesowaniem pakietów i jego bufor przepełnia się, to może poprosić o przesyłanie mniejszej ilości danych naraz.

Poza Advertised Window na wielkość Sliding Window liczy się też wartość Congestion Window, która zależy od aktualnego stanu sieci i tego, jak bardzo jest przeciążona. Przesyłanie dużej ilości danych w sieci przeciąża ją, więc TCP chce zmniejszyć liczbę wysyłanych bajtów, gdy wykryje przeciążenie. Dla TCP sygnałem przeciążenia sieci jest gubienie się pakietów. Pakiet zostaje uznany za zgubiony, gdy potwierdzenie jego przyjścia nie przychodzi przez bardzo długi czas (mierzony względem średniego RTT, czyli Round Trip Time).

Na początku połączenia Congestion Window jest ustawione na małą wielokrotność maksymalnego rozmiaru pakietu (Slow Start). Następnie rośnie o 1 MSS (maximum segment size) za każde otrzymane ACK, czyli może podwoić się w ciągu jednego RTT. Po osiągnięciu ustalonej wartości (slow start threshold, ssthresh) zaczyna rosnąć o co najwyżej 1 MSS w ciągu jednego RTT.

Po wykryciu utraty pakietu następuje retransmisja, a rozmiar Congestion Window zostaje zmniejszony. W przypadku stosowania algorytmu kontroli przeciążenia TCP Tahoe wartość ssthresh jest ustawiana na połowę aktualnego Congestion Window, a samo Congestion Window wraca do początkowej wartości – Slow Start zaczyna się od nowa. W przypadku algorytmu TCP Reno połowa wartości Congestion Window

staje się nową wartością ssthresh i jednocześnie nową wartością Congestion Window – pomijamy fazę Slow Start.

Może się zdarzyć tak, że tylko jeden pakiet z ciągu pakietów zostanie zgubiony. Odbiorca przyjmuje wtedy kolejne pakiety. Odsyła potwierdzenia, w których wartość ACK jest pierwszym jeszcze nie przyjętym bajtem, a więc pierwszym bajtem zgubionego pakietu. Nadawca dostaje wtedy wielokrotne potwierdzenia z tą samą wartością ACK. Po otrzymaniu trzeciego zduplikowanego potwierdzenia nadawca ma sporą pewność, że jeden pakiet się zgubił. Wysyła go więc ponownie, nie czekając na upłynięcie odpowiedniego czasu. Przyspiesza to transmisję w sytuacjach, gdy gubią się tylko pojedyncze pakiety. Takie zachowanie protokołu TCP nazywa się szybką retransmisją (Fast Retransmit). Szybka retransmisja nie powoduje spadku wartości Congestion Window.

W Pythonie obsługę TCP zapewnia moduł `socket`. Klient używa `socket()`, `connect()`, `sendall()` i `recv()`, natomiast serwer używa `socket()`, `bind()`, `listen()`, `accept()`, `sendall()` i `recv()`. Gniazdo TCP tworzy się jako `socket(AF_INET, SOCK_STREAM)`.

```python
import socket

SERVER = "example.org"
PORT = 50007

# socket IPv4 po TCP
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((SERVER, PORT)) # nawiązanie połączenia

    s.sendall(b"Hello!") # wysyłanie danych
    data = s.recv(1024) # odbiór co najwyżej 1024 bajtów danych
    print(data.decode("utf-8"))
```

```python
import socket

HOST = "" # wszystkie hosty
PORT = 50007

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT)) # powiązanie socketa z hostem i portem
    # nasłuchiwanie, argument to maksymalna długość kolejki nieprzyjętych połą
        czeń, po której system automatycznie odrzuci kolejne
    s.listen(1)
    conn, addr = s.accept() # przyjmowanie połączenia, nowy socket dedykowany
        do niego
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:  # pusty pakiet to zamknięte połączenie
                break
            conn.sendall(data)
```

III.5.5 Protokół HTTP

HTTP. Omów zawartość strumienia TCP podczas prostej komunikacji HTTP zwracając szczególną uwagę na sposób wykorzystania nagłówków. Skomentuj różnice w wykorzystaniu połączenia TCP w różnych wersjach protokołu HTTP. Jakie są powody i konsekwencje tych różnic?

HTTP (HyperText Transfer Protocol) działa w warstwie aplikacji i wykorzystuje usługi warstwy transportowej do prowadzenia komunikacji serwera webowego z klientem. W pierwszej, najprostszej wersji HTTP/0.9 klient przesyłał serwerowi żądanie (request) udostępnienia danego pliku, a serwer odsyłał go

w swojej odpowiedzi (response). W wersji HTTP/1.0 wprowadzono nagłówki – pary składające się z klucza i wartości, które opisują różne parametry requestu HTTP.

Każda linia w nagłówkach jest zakończona znakami `\r\n` (CRLF), a nagłówki od ciała wiadomości (body) oddziela pusta linia. Przykładowy request może wyglądać tak.

```
GET /index.html HTTP/1.1\r\n
Host: www.przyklad.pl\r\n
User-Agent: Mozilla/5.0\r\n
Accept: text/html\r\n
Connection: keep-alive\r\n
\r\n
[Ewentualne ciało requestu, np. dane z formularza]
```

Najpierw mamy użytą metodę, URI żądanego pliku i wersję HTTP. Następne są nagłówki, po których następuje ciało requestu. Odpowiedź serwera może wyglądać tak.

```
HTTP/1.1 200 OK\r\n
Date: Tue, 01 Jan 2137 12:00:00 GMT\r\n
Server: Apache/2.4\r\n
Content-Type: text/html; charset=UTF-8\r\n
Content-Length: 154\r\n
\r\n
<!DOCTYPE html><html>...[154 bajty kodu HTML]...</html>
```

Najpierw mamy wersję HTTP, kod statusu i jego komunikat, następnie nagłówki i na koniec ciało odpowiedzi.

Kilka możliwych typów requestów (metod):

- **GET** – zwykłe pobieranie pliku
- **HEAD** – tylko nagłówki
- **PUT** – zmiana danych na serwerze
- **POST** – umieszczenie nowych danych na serwerze
- **DELETE** – usuniecie danych z serwera
- **OPTIONS** – zapytanie o parametry komunikacji
- **TRACE** – pętla zwrotna, serwer zwraca to, co otrzymał
- **PATCH** – częściowa modyfikacji istniejącego zasobu

Przykładowe nagłówki żądania:

- **Host** – domena, do której chcemy się dostać; pozwala jednemu serwerowi obsługiwać wiele domen, od HTTP/1.1 obowiązkowy
- **User-Agent** – informacja o typie klienta
- **Accept** – oczekiwany typ danych
- **Cookie** – dane sesji
- **Authorization** – dane uwierzytelniające
- **If-Modified-Since** – prosi o przesłanie danych tylko jeśli były zmodyfikowane od podanej daty, używany do cache'owania
- **Connection** – czy utrzymywać połączenie po odpowiedzi na to żądanie

Przykładowe nagłówki odpowiedzi:

- **Content-Type** – w jakim formacie jest dokument (html, pdf itd.)

- **Content-Length** – długość w bajtach przesyłanej wiadomości, dla danych przesyłanych z serwera obowiązkowy
- **Content-Encoding** – podaje format kodowania treści
- **Set-Cookie** – ustawia ciasteczko u klienta
- **Date** – aktualny czas
- **Location** – przekierowanie w inne miejsce
- **Server** – identyfikuje serwer i użyte w nim oprogramowanie
- **WWW-Authenticate** – określa sposób w jaki ma zostać przeprowadzone uwierzytelnienie użytkownika
- **Expires** – data, po której dokument jest nieaktualny, używany do cache'owania
- **Last-Modified** – data ostatniej modyfikacji, używany do cache'owania
- **Allow** – określa metody HTTP obsługiwane przez serwer

W HTTP/1.0 było jedno połączenie TCP na request, po wysłaniu danych serwer natychmiast zamykał połączenie TCP. Powodowało to ogromny narzut na ponowne nawiązywanie połączeń (każdorazowy TCP 3-way handshake). Dlatego w HTTP/1.1 wprowadzono trwałe połączenia i pipelining. Można zastosować nagłówek `Connection: keep-alive` aby jedno połączenie TCP pozostało otwarte na wiele requestów. Wprowadzono też pipelining (wysłanie kolejnych requestów przed odebraniem poprzednich odpowiedzi). Nie był on jednak popularny z powodu tak zwanego blokowania HoL (Head-of-Line): odpowiedzi w strumieniu TCP musiały wracać dokładnie w takiej kolejności, w jakiej wysłano requesty. Jeśli pierwszy request (np. ciężki skrypt PHP) generował się długo, to blokował pobieranie kolejnych zasobów (np. lekkich obrazków), więc pipelining nie bardzo miał sens. Aby sobie z tym poradzić w HTTP/2.0 wprowadzono multiplexing – serwer może wysyłać odpowiedzi równocześnie, w dowolnej kolejności, kawałek po kawałku, co rozwiązuje problem HoL. Dodatkowo zmieniono format przesyłanych danych z tekstowego na binarny.

W HTTP/2.0 dalej występował tak zwany problem TCP HoL. Jeżeli zgubi się jeden segment TCP, to TCP nie udostępni aplikacji danych po zgubionym segmencie, dopóki nie zostanie on retransmitowany. Jest to ograniczenie wynikające wprost z zastosowania protokołu TCP. Dlatego standard HTTP/3.0 odszedł od TCP na rzecz warstwy QUIC implementowanej za pomocą protokołu UDP.

III.5.6 TLS i SSL

> Transport Layer Security. Opisz jak TLS (SSL) używa kryptografii klucza publicznego i kryptografii klucza symetrycznego. Opisz schemat certyfikacji kluczy stosowany w TLS. Opisz API do obsługi TLS w bibliotece standardowej (moduł ssl) w języku Python.

TLS (Transport Layer Security) to protokół zapewniający bezpieczną komunikację w sieci komputerowej. Jest następcą protokołu SSL (Secure Sockets Layer). Często stosuje się te dwa określenia wymiennie mając na myśli protokół TLS – faktyczny SSL jest stary i nieużywany (w 1999 roku TLS1.0 powstał jako następca SSL3.0). TLS zapewnia poufność danych (nie mogą zostać odczytane przez osoby trzecie), ich integralność (dane nie mogą zostać zmienione w sposób niezauważony) oraz uwierzytelnienie stron komunikacji (jedna strona może zweryfikować tożsamość drugiej). TLS jest używany m.in. w HTTPS, SMTP, IMAP i wielu innych protokołach.

TLS korzysta zarówno z szyfrowania asymetrycznego (klucza publicznego), jak i symetrycznego. Szyfrowanie asymetryczne jest wykorzystywane głównie podczas nawiązywania połączenia (TLS Handshake). Każda strona posiada klucz prywatny (tajny) i klucz publiczny (jawny). Za ich pomocą strony są w stanie nadawać do siebie wiadomości, które tylko one nawzajem mogą odczytać. Stosuje się tutaj algorytmy takie jak RSA czy protokół Diffie'ego-Hellmana. Te algorytmy są skomplikowane obliczeniowo (wymagają przeprowadzenia pewnych operacji na bardzo dużych liczbach), ale pozwalają na bezpieczną komunikację w środowisku, które stale podsłuchuje komunikujące się strony.

Za pomocą szyfrowania asymetrycznego strony ustalają wspólny klucz symetryczny, który służy do dalszej komunikacji – za jego pomocą można szyfrować oraz odszyfrowywać wiadomości. Stosuje się w tym celu algorytmy szyfrowania symetrycznego (np. AES), które są bardzo szybkie i dlatego dobrze nadają się do szyfrowania i przesyłania dużej ilości danych (ale wymagają ustalenia na wstępie klucza symetrycznego w jakiś bezpieczny sposób tak, by nikt inny go nie znał).

Algorytm RSA (Rivest-Shamir-Adleman) działa w następujący sposób. Niech p, q będą pierwsze, $N = pq$. Mamy $\varphi(N) = (p-1)(q-1)$. Dla $e, d \in \mathbb{Z}$ takich, że $ed \equiv 1 \pmod{\varphi(N)}$ możemy szyfrować $x \to x^e \pmod{N}$ i deszyfrować $y \to y^d \pmod{N}$. Liczba e jest kluczem publicznym, a d kluczem prywatnym.

Protokół Diffie'ego-Hellmana działa w następujący sposób. Mamy nieszyfrowany kanał komunikacji. Chcemy w bezpieczny sposób ustalić wartość, która będzie kluczem symetrycznym służącym do dalszej bezpiecznej komunikacji. W tym celu wybieramy grupę cykliczną G i jej generator g (typowo G to grupa pewnego ciała skończonego, w najprostszym wariancie $\mathbb{Z}_p$ dla pierwszego p; można też stosować bardziej wyszukane grupy, na przykład krzywe eliptyczne). Jedna strona komunikacji wybiera klucz prywatny $a \in \mathbb{Z}$, a druga $b \in \mathbb{Z}$. Następnie publikują g^a (odpowiednio g^b). Ustalana wartość to g^{ab} – obie strony umieją ją wyznaczyć, a strona przechwytująca komunikację musiałaby wyznaczyć g^{ab} znając g^a i g^b, co jest trudnym problemem znanym jako problem Diffie'ego-Hellmana.

Algorytm AES (Advanced Encryption System) polega na mieszaniu bitów szyfrowanej wiadomości w sposób zadany przez klucz (i taki, że mała zmiana w kluczu lub wiadomości powoduje duże zmiany w zaszyfrowanym tekście). Jest on zaprojektowany w taki sposób, aby wykonanie go od tyłu pozwoliło na odszyfrowanie wiadomości (oczywiście przy znajomości klucza).

Przedstawiony system działa, o ile zakładamy, że nikt nie zmienia przesyłanych komunikatów. Jeśli tak nie jest, to ktoś inny może przechwycić komunikację podczas TLS Handshake i przesłać obu stronom swój klucz publiczny. One następnie ustalą z oszustem klucz symetryczny tak jakby ustaliły go między sobą i zaczną komunikację, którą oszust może odczytywać. Takiej sytuacji (Man in the Middle Attack) ciężko uniknąć bez wprowadzenia dodatkowych mechanizmów kontroli. W tym celu stosuje się tak zwane certyfikaty SSL (a właściwie TLS, ale używa się tych pojęć wymiennie), które umożliwiają potwierdzenie tożsamości drugiej strony.

Podczas zawierania połączenia TLS strony przesyłają sobie klucze publiczne. Aby dowieść swojej tożsamości przesyłają dodatkowo certyfikat SSL (zazwyczaj robi to tylko jedna strona komunikacji – serwer, z którym jakiś użytkownik chce się skomunikować). Taki certyfikat poświadcza o tożsamości posiadacza i jest podpisany przez odpowiednią instytucję wydającą (Certificate Authority, CA), to znaczy jest zaszyfrowany kluczem prywatnym tej instytucji (a konkretniej hasz certyfikatu jest tak zaszyfrowany). Strona komunikacji może odszyfrować tak podpisany certyfikat za pomocą klucza publicznego tej instytucji, który jest dołączony do certyfikatu i w ten sposób potwierdzić jego oryginalność. Skąd wiemy, czy ten certyfikat nie został sfałszowany? Otóż certyfikat ma swój certyfikat, podpisany przez inne, większe CA. Ten certyfikat również jest certyfikowany i tak dalej, aż ostatecznie trafimy na certyfikat podpisany przez tak zwane Root CA – instytucję na tyle dużą, że jej klucze publiczne znajdują się w magazynie zaufanych certyfikatów systemu operacyjnego. Certyfikatom wystawianym przez Root CA ufamy. Powstały w ten sposób łańcuch zależności nazywamy Infrastrukturą Klucza Publicznego (Public Key Infrastructure, PKI).

W Pythonie obsługę TLS zapewnia moduł `ssl`. Kluczową klasą jest `SSLContext`, służąca do konfiguracji połączenia. Funkcja `create_default_context()` tworzy bezpieczną konfigurację klienta, `wrap_socket()` zamienia zwykły socket w połączenie TLS, `load_cert_chain()` ładuje certyfikat serwera, a `getpeercert()` pozwala odczytać certyfikat drugiej strony. Dzięki temu moduł umożliwia implementację klientów i serwerów korzystających z TLS.

```python
import socket
import ssl

hostname = "example.org"
port = 443 # standardowy port HTTPS

# domyślna konfiguracja
context = ssl.create_default_context()

with socket.create_connection((hostname, port)) as sock:
```

```
11      # opakowanie socketu w warstwę TLS
12      with context.wrap_socket(sock, server_hostname=hostname) as ssock:
13          print(f"Protocol version: {ssock.version()}. Certificate data: {ssock.
                getpeercert()}")
14          # tutaj dalsze działania jak na zwykłym obiekcie socket
```

```
1  import socket
2  import ssl
3
4  HOST = "example.org"
5  PORT = 443
6
7  # konfiguracja TLS dla serwera
8  context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
9
10 # wczytanie certyfikatu i klucza prywatnego
11 context.load_cert_chain(
12     certfile="server.crt",
13     keyfile="server.key"
14 )
15
16 with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
17     sock.bind((HOST, PORT))
18     sock.listen(5)
19     while True:
20         client_socket, address = sock.accept()
21         with context.wrap_socket(client_socket, server_side=True) as tls_socket
               :
22             # tutaj dalsze działania jak na zwykłym obiekcie socket
```

III.6 Systemy Operacyjne

III.6.1 Komunikacja międzyprocesowa

Opisz mechanizmy komunikacji międzyprocesowej standardu POSIX.

Są cztery podstawowe mechanizmy komunikacji.

1. Sygnały – jeden z najprostszych systemów komunikacji międzyprocesowej, pozwala na wysyłanie małej ilości informacji do innych procesów, których identyfikator (PID) znamy.[4]

   Sygnały wysyłamy korzystając z funkcji `kill(2)`, podając numer sygnału (w kodzie zwykle jako makro zdefiniowane w headerze `signal.h`) oraz numer procesu. Procesy otrzymujące sygnał rejestrują funkcje obsługujące sygnał za pomocą funkcji wyższego rzędu `sigaction(2)`. Dana funkcja zostaje wykonana przez kernel, podczas gdy pozostałe akcje zostają przerwane i przywrócone w momencie zakończenia pracowania nad sygnałem. Pracując nad sygnałem mamy ograniczony zbiór dozwolonych operacji (jako że mogliśmy brutalnie przerwać np. `printf`'a, to nie możemy stwierdzić że jego bufory są w odpowiednim stanie aby z niego korzystać i co za tym idzie nie możemy z niego korzystać), które nazywamy funkcjami asynchronicznie bezpiecznymi (async-signal-safe). Istnieje też funkcja `signal(2)`, jednak jej semantyka jest bardzo błędogenna i przestarzała, więc z niej nie korzystamy.

   Jeżeli sygnał nie zostanie złapany (nie zarejestrujemy handlera sigaction), to wykonuje się jego domyślna akcja. Zazwyczaj jest to zakończenie programu, czasem jego zapauzowanie lub zignorowanie sygnału.

   W podstawowym standardzie zdefiniowane jest całkiem sporo różnych sygnałów, w tym dwa przewidzane do użytku przez aplikacje w celu przekazywania „generycznego" sygnału. Poniżej jest kilka ważniejszych z nich.

   - INT / QUIT – przerwij/zatrzymaj wykonanie programu; wysyłane przez emulator terminala po wciśnięciu `CTRL+C` / `CTRL+\`; proszą program o zatrzymanie, INT uprzejmiej (często programy go przechwytują), QUIT lekko mniej uprzejmie (rzadko przechwytywany, ale da się) i domyślnie generuje dodatkowo core dump procesu, czyli zrzut jego pamięci wirtualnej w momencie zatrzymania na potrzeby debugowania.
   - ABRT – abort, wysyłany przez np. niezłapane wyjątki C++'sa czy funkcję `abort(3)`, analogicznie do QUIT zabija proces i generuje coredump o ile nie jest przechwycony sigaction;
   - KILL – natychmiast zatrzymaj proces; niemożliwy do przechwycenia, kernel z góry zatrzymuje proces i zabija;
   - TSTP / STOP – zatrzymaj proces, zapisując jego stan w pamięci do późniejszego wznowienia za pomocą sygnału CONT; TSTP generowany przez terminal po kliknięciu `CTRL-Z`, STOP jest analogiem do pary INT/KILL, czyli zatrzymuje program bez możliwości reakcji / złapania;
   - CHLD – wysyłany rodzicowi procesu aby poinformować go, że jedno z jego bezpośrednich dzieci zakończyło działanie – domyślnie ignorowany, użyteczny do przechwycenia np. przy pisaniu interaktywnego shella aby wiedzieć, kiedy można kontynuować swoje wykonywanie;
   - USR1 i USR2 – wspomniane wcześniej sygnały „generyczne" tj. użytkownika; domyślnie kończą program (jak INT).

   Ważną cechą sygnałów jest, że nie kolejkują się oraz nie mamy pewności w jakiej kolejności zostaną rozpatrzone – jeżeli procesy A, B, C wyślą sygnały USR1, USR1 i USR2 do procesu D (w tej kolejności chronologicznej, nawet zakładając pojedynczy rdzeń procesora), to proces D może

[4] W teorii możemy też wysłać go do całej grupy procesów; aby to zrobić, w funkcji kill podajemy numer grupy jako zanegowaną wartość.

rozpatrzyć je jako USR2 i USR1 (bez zachowania kolejności i gubiąc jedno powtórzenie) – wynika to po części z faktu, że standard został napisany na podstawie istniejących implementacji, które dla prostoty i szybkości wprowadziły taką semantykę. W ten sposób lista sygnałów dla procesu może być przechowywana dla każdego procesu jako liczba, której i-ty bit oznacza, że proces jeszcze nie rozpatrzył sygnału nr. i i musi to zrobić tak szybko jak to możliwe (kernel wybiera pierwszy zapalony, bit więc tracimy porządek przychodzenia sygnałów, a zapalenie dwa razy tego samego bitu pozostaje niezauważone).

W celu uproszczenia i zwiększenia użyteczności sygnałów, do standardu POSIX dodano również rozszerzenie (które np. Linux stara się wspierać) sygnałów Real-Time. Sygnały te różnią się tym, że:

- są wysyłane przez `sigqueue(3)` a nie kill;
- są kolejkowane (stąd nazwa), czyli przychodzą w poprawnej kolejności i się nie gubią;
- mogą zawierać dodatkową informację w formie jednej liczby lub wskaźnika.

2. Strumienie (pipe/FIFO) – traktujemy je jako pseudo-plik, który kolejkuje zapisy do siebie w ramach pewnego buforu w pamięci, a następnie wypisuje je w analogicznej kolejności (czyli jak kolejka first-in-first-out) przy odczycie przez ten sam lub inny proces. Na strumieniach operuje się w sposób analogiczny do plików – otrzymujemy file descriptor, z którego możemy korzystać podobnie do wszelkich innych (metody `read(2)` i `write(2)` działają jakbyśmy się spodziewali).

   Dzielimy je na dwa główne rodzaje: anonimowe i nazwane.

   Strumienie anonimowe możemy utworzyć korzystając m.in. z metody `pipe(2)`, która zwraca dwa nowe deskryptory – jeden „otwarty" w trybie wejścia, drugi wyjścia. Te deskryptory możemy następnie bezpośrednio przekazać innym programom (np. dzieciom), lub skorzystać z nich aby podmienić znaczenie istniejącego deskryptora typu `1`, tj. `stdout`.

   Strumienie nazwane możemy stworzyć np. za pomocą funkcji `mkfifo(3)` – podajemy mu ścieżkę pod którą utworzy się obiekt w systemie plików (nie jest to plik *per se*, jednak możemy go otworzyć i operować na nim w dużej mierze analogicznie). W ten sposób wiele programów może otworzyć danego pipe'a i równolegle z niego korzystać, np. do pisania logów, które jeden proces-odbiorca zbiera i przekazuje dalej.

   Ważną rzeczą w przypadku pipe'ów jest, analogicznie do plików, brak domyślnej synchronizacji – podczas gdy operacje write mają zagwarantowaną atomowość w przypadku małych zapisów (do rozmiaru `PIPE_BUF` zadeklarowanego przez system, co najmniej 512 byte'ów), to odczyty nie mają takiej gwarancji. Jednocześnie bardzo łatwo o problem, gdy odczytamy jedynie część wiadomości – nasza będzie bezużyteczna, a kolejna, być może nie odczytana przez ten sam proces, zostanie skoruptowana.

   Warto zaznaczyć, że **sockety** możemy interpretować jako specjalny rodzaj pipe'ów, który system operacyjny dodatkowo podpina pod odpowiednie części jądra, aby móc komunikować się w bardziej skomplikowany sposób, np. przez urządzenia sieciowe; jest to trochę inne, jednak kluczowe jest podobieństwo interfejsu – zarówno metody na socketach jak i lokalnych strumieniach zwracają/przyjmują/operują na deskryptorach plików (FD).

3. Pamięć współdzielona (shared memory) – najbardziej efektywny sposób komunikacji, pozwalający na przekazywanie informacji nie przechodząc przez jądro (oczywiście po pierwotnym otwarciu). Główną wadą w porównaniu do innych metod jest potrzeba utrzymywania stałego modelu danych oraz konieczność synchronizacji w celu zabezpieczenia danych przed korupcją.

   Standard POSIX udostępnia zbiór funkcji, które pozwalają na manipulację blokami pamięci współdzielonej – działają one również na deskryptorach plików, jednak korzystają z mechanizmu mapowania plików (tj. `mmap(2)`) a nie z operacji read/write jak pipe'y. Funkcja `shm_open(3)` pozwala na otwarcie/utworzenie nowego bloku pamięci o wspólnym identyfikatorze (jak `open(2)` z flagą create), a `shm_unlink(3)` na jego zamknięcie (jak `close(2)`).

   Alternatywnym sposobem w Linuxie jest bezpośrednie użycie funkcji `mmap(2)` do alokowania anonimowego bloku pamięci współdzielonej, którą możemy następnie przekazać naszym bezpośrednim dzieciom (`fork(2)` zachowuje te mapowania).

W standardzie POSIX istnieje też dużo mechanizmów synchronizacji (mutex'y, semafory, bariery) – one również mają dwa główne interfejsy; nazwane (wtedy otwieramy je po ścieżce w analogiczny sposób do podobnego API dla pamięci) **lub poprzez wspólne miejsce w pamięci współdzielonej (dlatego uwzględniamy je w tej sekcji)**. Mechanizmy te są przydatne najbardziej przy korzystaniu z pamięci współdzielonej, ale potrafią przydawać się też przy dostępach do plików/pipe'ów.

4. Message queues[5] – bliski odpowiednik message passing'u w Minixie, rozwiązuje problem pipe'ów, tj. fakt, że dane są nieustrukturyzowane i możemy przeczytać tylko część wiadomości, desynchronizując/koruptując kanał. Jest cały rząd metod do operacji na nich (patrz: `mq_overview(7)`), podstawowe to otwarcie kolejki (po nazwie, zwraca identyfikator), zamknięcie kolejki , wysłanie wiadomości (być może z timeoutem, aby nie blokować się na za długo jeżeli kolejka jest pełna), odebranie wiadomości (być może z timeoutem, aby nie blokować się na za długo jeżeli kolejka jest pusta).

Na koniec fun fact – dla dużej liczby rzeczy (typu message queue, shared memory i mutexy) istnieje równoległy, niekompatybilny system metod zachowanych na potrzeby „backwards compatibility" z systemem UNIX w wersji piątej (zazwyczaj znanym jako SysV). W opinii autora tego opracowania, API to jest w wielu aspektach lepiej przemyślane od POSIX'owego (dokładniej ma mniej „undefined behaviour" w sobie), ale bywa mniej popularne i nie jest rozwijane (ale jest stabilne).

III.6.2 Szeregowanie procesów

Porównaj mechanizmy szeregowania zadań na przykładzie serwera przetwarzającego zadania w trybie wsadowym oraz systemu interaktywnego.

Szeregowanie (*scheduling*) to decyzja ile czasu procesora (i kiedy) dostaje każdy proces.

Podstawową osią dzielącą systemy szeregowania jest pytanie, czy jeżeli daliśmy procesorowi procesor to czy możemy mu go odebrać (pauzując egzekucję); dzielimy więc metody szeregowania na *z wywłaszczaniem* (możemy zabrać procesor) i *bez wywłaszczania* (sam proces musi ustąpić, np. kończąc się lub blokując na syscall'u).

Podczas porównywania systemów szeregowania interesują nas podstawowe kryteria:

- przepustowość (*throughput*), czyli liczba zadań, którą możemy ukończyć na jednostkę czasu.
- czas obrotu (*turnaround time*), czyli ile czasu mija od powstania zadania do jego ukończenia.
- czas oczekiwania (*waiting time*), czyli ile czasu proces łącznie spędza będąc gotowym do wykonania i czekając na procesor.
- czas odpowiedzi (*response time*), czyli ile czasu mija od gotowości procesu/zadania do pierwszego przydziału procesora na jego rzecz.
- sprawiedliwość (*fairness*); cięższe do skwantyfikowania, ogólnie chodzi o to aby nie głodzić niektórych zadań.[6]

Każdy scheduler chciałby oczywiście optymalizować wszystkie z tych metryk, jednak świat nie jest taki piękny i trzeba wybrać które kryteria są dla nas najważniejsze; każdy ma swoje priorytety, niestety.

1. Przy systemach wsadowych zadania są kolejkowane i wykonywane do końca. Czas odpowiedzi jest niezbyt istotny; optymalizujemy przepustowość (i co za tym idzie wykorzystanie procesora) oraz średni czasu obrotu. Możemy (ale nie musimy) sobie pozwolić na szeregowanie bez wywłaszczania (mniejszy narzut na przełączanie kontekstu, prostsza implementacja).

   Najbardziej klasyczne metody określania kolejności w tej rodzinie to:

[5] O tym opowiemy najkrócej, bo chyba nie było to wspomniane na Systemach Operacyjnych

[6] Terminów procesy/zadania używamy względnie wymiennie – w związku z ich zastosowaniami termin „zadanie" stosuje się zwykle dla systemów wsadowych, a „proces" przy systemach interaktywnych.

- *First-In-First-Out*, czyli standardowa kolejka; najprostszy sposób, delegujemy zadania po czasie ich zgłoszenia. Efektywny sposób, jednak mamy problem w postaci tzw. efektu konwoju, gdzie jedno długie zadanie może blokować całą resztę przez co znacząco cierpi średni czas obrotu.
- *Shortest Job First*, czyli zamiast kolejki używamy kolejki priorytetowej (wybieramy najkrótsze zadanie); dowodliwie optymalny sposób dla średniego czasu oczekiwania oraz obrotu. Problematyczny, ponieważ trzeba umieć szacować długość zadania oraz możemy zagłodzić długie zadania (nigdy nie dając im się wykonać). Ten pierwszy problem można rozwiązać dając wskazówki (np. przy dużych zadaniach w ramach sieci obliczeniowej możemy zwykle oszacować tę długość całkiem dobrze); ten drugi też się da się rozwiązać, ale wtedy uznajemy model za interaktywny.
- *Shortest Remaining Time First*, czyli podobne do powyższego, tylko jeżeli przychodzi nowe zadanie krótsze od czasu, który został do zakończenia wykonywania obecnego, to uruchamiamy je natychmiast, wywłaszczając procesor.

2. Systemy interaktywne zamiast optymalizować pierwszorzędnie przepustowość, optymalizują fundamentalnie czas odpowiedzi oraz wariancję. Ludzie przy terminalach oczekują szybkich i przewidywalnych reakcji. Systemy te wymagają wywłaszczania, gdyż inaczej nieposłuszny proces mógłby zawłaszczyć procesor *ad infinitum* i nie dać innym się wykonać.

   Kilka algorytmów przydziału z tej dziedziny to:

   - *Round Robin*; każdy proces dostaje stały kwant czasu (*quantum*), po czym trafia na koniec kolejki. Wspomagając się przerwaniami zegarowymi wywłaszczamy procesy trwające więcej niż ich kwant, wrzucając je na koniec kolejki i zastępując górą kolejki. Dobór kwantu jest istotny, ponieważ zmiany kontekstu są całkiem drogie (nie może być za mały), a przy dużym kwancie zachowujemy się jak wsadowa kolejka FIFO (czas odpowiedzi cierpi).
   - *Szeregowanie Priorytetowe z Postarzaniem*; priotytetyzujemy procesy według jakiejś metryki, ale dodatkowo zwiększamy priorytet dla procesów które czekają już długo (postarzamy je), aby ich nie zagłodzić; priorytetyzując czas wykonania dostajemy Shortest Job First, jednak bez wady zagłodzenia;
   - *Multilevel Feedback Queue*, czyli pięknie spolszczone wielopoziomowe kolejki ze sprzężeniem zwrotnym; względnie zaawansowany system rozdzielania zadań, używany np. w Minixie 3. Robimy wiele kolejek, każda mająca jakiś priorytet – jeżeli zadania czekają na kilku z nich wybieramy tę z największym priorytetem, a wewnątrz jednej kolejki robimy Round Robin z jakimś kwantem czasu. Dla większości zadań (tj. nie procesów systemowych, które mają stale wysoki priorytet) możemy zmieniać ich priorytet w zależności od ich zachowania (to jest „feedback" z nazwy). Jeżeli proces zużywa cały kwant czasu bez blokowania na IO, tymczasowo obniżamy jego priorytet, tj. przerzucamy do „gorszej" kolejki; z czasem przywracamy priorytet aby nie karać go w nieskończoność za dane przewinienie. Zaletą tego systemu jest to, że efektywnie przybliża on Shortest Job First (co daje nam jego fajne własności bycia optymalnym pod względem czasu oczekiwania/obrotu) bez znania długości każdego z zadań *a priori*. Równolegle do tego dostajemy zalety metody Round Robin w zakresie małego czasu odpowiedzi i wariancji.

   Ważna uwaga: przy metodach szeregowania z priorytetem może nastąpić *inwersja priorytetów*. Dzieje się to gdy proces wysokopriorytetowy czeka na zasób (np. zwolnienie mutexa) zajęty przez proces o niższym priorytecie. Jeżeli powstanie proces z priorytetem gdzieś pomiędzy ich priorytetami, to wywłaszczy on proces o niskim priorytecie, efektywnie blokując ten o wysokim priorytecie (a tak nie chcemy). Rozwiązaniem jest dziedziczenie priorytetów: jeżeli jako proces trzymamy mutex na którym blokuje się proces o wyższym priorytecie, tymczasowo dziedziczymy jego priorytet, aby móc dokończyć działanie i pozwolić mu na dalsze wykonanie. W ten sposób procesy o priorytecie „pośrednim" nie mogą zablokować działania tego o wyższym. Co ważne, dziedziczenie trwa tylko na czas gdy blokujemy zasób potrzebny procesowi o wyższym priorytecie (po jego zwolnieniu zostajemy wywłaszczeni a owy wysokopriorytetowy proces zaczyna egzekucję). Alternatywnym rozwiązaniem jest ustalenie dla każdego zasobu zakresu priorytetów jakiego muszą być dane procesy w celu korzystania z niego; w ten sposób wiemy, że dany zasób nie będzie mógł być wykorzystany przez zbyt ważny proces, ergo nie mamy jak go zablokować.

Podsumowując, porównując metody możemy zobaczyć następującą zależność: systemy interaktywne zachowują się znacząco lepiej przy procesach ograniczonych IO i, jak nazwa wskazuje, interaktywnych (krótkie fazy obliczeń, potem czekanie na użytkownika/dysk; dlatego Linux/Minix je implementują). Tymczasem metody wsadowe lepiej utylizują zasoby w przypadku procesów ograniczonych mocą procesora. Korzystając z nich mamy większy throughput, czyli przerabiamy więcej zadań przez dany czas, jednak nie jesteśmy bardzo responsywni. Dlatego używane są np. w sieciach obliczeniowych, gdzie zmiany kontekstu są znacząco droższe niż na zwykłym procesorze.

III.6.3 Deadlock

Wyjaśnij pojęcie deadlock. Opisz metody wykrywania i zapobiegania powstawaniu deadlocku w kontekście współdzielonych zasobów.

Deadlock (zakleszczenie) to sytuacja w systemie współbieżnym, w której grupa procesów lub wątków czeka na zasoby zajmowane przez siebie nawzajem, przez co żaden z nich nie może kontynuować działania. W efekcie procesy nie kończą swojej pracy, zasoby pozostają zablokowane, a system nie robi postępu.

Aby doszło do zakleszczenia, w tym samym czasie muszą zostać spełnione cztery warunki znane jako warunki Coffmana.

1. **Wzajemne wykluczenie (Mutual Exclusion)**. Zasoby są niepodzielne – w danym momencie tylko jeden proces może korzystać z danego zasobu.
2. **Przetrzymywanie i oczekiwanie (Hold and Wait)**. Proces, który już posiada jakiś zasób, może prosić o kolejne zasoby i czekać na nie, nie zwalniając tych, które już trzyma.
3. **Brak wywłaszczania (No Preemption)**. Zasób nie może zostać odebrany procesowi siłą. Może zostać zwolniony tylko dobrowolnie przez proces, który go posiada.
4. **Cykliczne oczekiwanie (Circular Wait)**. Istnieje zamknięty łańcuch procesów, w którym każdy proces czeka na zasób trzymany przez kolejny proces w łańcuchu.

Aby wykryć deadlock buduje się tak zwany graf alokacji zasobów, w którym wierzchołkami są procesy (P) oraz zasoby (R). Krawędź $P \to R$ oznacza, że proces żąda zasobu, a krawędź $R \to P$ oznacza, że zasób jest przydzielony procesowi. Istnienie cyklu w takim grafie oznacza, że wystąpił deadlock (zakładając, że każdy zasób ma dokładnie jedną instancję; jeśli tak nie jest, to możemy wykrywać deadlocki za pomocą algorytmu działającego tak jak opisany niżej algorytm bankiera).

Istnieje kilka metod radzenia sobie z deadlockami. Najprostsza to ignorowanie problemu – zakładamy, że deadlocki zdarzają się rzadko i jeśli już taki się pojawi, to można zacząć obliczenia od nowa.

Inną metodą jest okresowe wykrywanie deadlocku za pomocą wyżej opisanych metod i reagowanie, gdy deadlock się pojawi. Jednym ze sposobów reagowania jest zakończenie zakleszczonych procesów. Możemy zabić je wszystkie lub tylko jeden, wybrany losowo lub za pomocą jakiejś metryki (na przykład czas działania, stopień zaawansowania wykonywanego zadania). Jest to prosta metoda, ale jej wadą jest utrata wykonanej pracy. Innym pomysłem jest wywłaszczanie zasobów – możemy odebrać zasoby jednemu procesowi i przekazać je innemu, tym samym rozbijając cykl. Wymaga to cofnięcia wywłaszczanego procesu do bezpiecznego punktu kontrolnego sprzed zajęcia zasobu, co niekoniecznie jest łatwe. Zaletą takiego podejścia jest to, że nie tracimy aż tyle wykonanej pracy co przy zabijaniu procesów.

Klasycznym przykładem deadlocku jest problem ucztujących filozofów, dokładniej opisany w III.6.8.

Można też całkowicie zapobiegać deadlockom, to znaczy projektować systemy współbieżne w taki sposób, aby co najmniej jeden warunek Coffmana nigdy nie był spełniony.

1. **Wzajemne wykluczenie**. Dla znaczącej większości zasobów nie da się go wyeliminować, gdy programy na przykład piszą po tej samej pamięci, to zezwolenie na równoległe pisanie może spowodować niewłaściwe zapisy.

2. **Przetrzymywanie i oczekiwanie**. Możemy wymusić, aby proces pobierał wszystkie potrzebne mu zasoby jednocześnie. Uniemożliwia to wystąpienie deadlocków, ale powoduje słabe wykorzystanie zasobów, bo procesy zajmują zasoby mimo, że nie potrzebują ich w danym momencie.
3. **Brak wywłaszczania**. Procesy mogą zwalniać zajęte zasoby, jeśli nie dostały kolejnych. Teoretycznie nie wystąpią wtedy deadlocki, ale mogą wystąpić tak zwane livelocki, czyli sytuacje, gdy procesy pracują, ale żaden z nich nie dokonuje postępów (na przykład wszystkie procesy na zmianę zajmują zasoby i je zwalniają, bo nawzajem je sobie zajmują).
4. **Cykliczne oczekiwanie**. Najprostszą metodą eliminacji cykli jest ponumerowanie zasobów i przyznawanie ich tylko w kolejności rosnących indeksów. Wtedy nie może powstać cykl.

Do unikania deadlocków służy tak zwany algorytm bankiera, zaproponowany przez Edsgera Dijkstrę. Zakładamy, że system wie, ile zasobów jest dostępnych w systemie i ile z nich w danym momencie posiada każdy proces. Do tego dla każdego procesu znana jest maksymalna ilość danego zasobu, jakiej może zażądać. System przydziela procesowi zasoby tylko wtedy, gdy wie, że istnieje taka kolejność przydzielania zasobów do obecnie działających procesów, że każdy z nich jest w stanie zakończyć działanie korzystając tylko z nieprzydzielonych w tym momencie zasobów. W praktyce sprawdza się ten warunek iterując się po wszystkich procesach i wybierając jeden, który jest w stanie się skończyć, a następnie powtarzając taką iterację przy założeniu, że zasoby zajęte przez ten pierwszy proces są już zwolnione.

III.6.4 Spooling

> Wyjaśnij mechanizm spooling, omów wady i zalety tego mechanizmu.

Spooling (Simultaneous Peripheral Operations On-Line)[7] to technika zarządzania urządzeniami wejścia-/wyjścia, która polega na wykorzystaniu nośnika pamięci masowej jako tymczasowego bufora pośredniczącego między powolnymi urządzeniami peryferyjnymi (np. drukarka, skaner, czytnik taśm magnetycznych) a procesorem/programami.

Jeśli proces chce się skomunikować z pewnym wolnym urządzeniem zewnętrznym, to standardowo robi to za pomocą odpowiednich portów (port-mapped IO), a urządzenie informuje proces o gotowości do działania za pomocą pollingu, to znaczy proces musi bezpośrednio prosić je o raport gotowości. Jeśli proces chce na przykład skorzystać z drukarki, to musi wysłać jej część danych do drukowania za pomocą odpowiedniego portu, a następnie co jakiś czas odpytywać, czy drukarka skończyła fizycznie przetwarzać te dane. Dopiero wtedy proces może wysłać kolejną część danych. Jest to ogromne marnowanie czasu procesu, bo jak wiemy drukarki bywają wolne nawet dla ludzi, a co dopiero dla procesorów. Proces przez cały czas drukowania musi co chwilę czekać na drukarkę i nie może wykonywać żadnych innych operacji, aż drukowanie się nie skończy. Ponadto, często chcemy ograniczyć port-mapped I/O dla niektórych urządzeń do działania jedynie w trybie uprzywilejowanym procesora, co również lekko utrudnia nam sytuację w przypadku aktywnego czekania.

Rozwiązaniem tego problemu jest spooling: zamiast wysyłać dane bezpośrednio z programu do drukarki, system operacyjny (pamiętajmy, że to on pośredniczy w komunikacji procesu z urządzeniem) zapisuje całe drukowane dane na dysku w specjalnym obszarze zwanym spoolem (kolejką). Natychmiast informuje program, że zadanie zostało zakończone, dzięki czemu ten może działać. W tle działa dedykowany proces (tak zwany spooler; w Linux'ie tę rolę dla drukarki spełnia system drukujący CUPS), który pobiera dane ze spoola i powoli, w tempie drukarki, przesyła jej kolejne dane do przetworzenia.

Taki mechanizm zwiększa wydajność systemu i procesora oraz upłynnia działanie programu, który zleca drukowanie – nie trzeba marnować cykli procesora na czekanie w pętli aż powolna operacja I/O zakończy się. Do tego rozwiązuje on problem urządzeń dedykowanych (takich, które nie mogą przerwać pracy nad jednym zadaniem, by zająć się innym) polegający na tym, że jeśli wiele procesów chce korzystać z drukarki jednocześnie, to drukarka zajmie się jednym zadaniem, a pozostałe procesy będą musiały czekać na zakończenie tego zadania, nim ich zadanie będzie mogło zacząć się wykonywać. Spooling pozwala wielu

[7] Poprawność tego rozwinięcia nie jest pewna, istnieje spora szansa, że jest to akronim wtórny, czyli najpierw pojawiło się słowo „spooling”, a dopiero potem zostało rozwinięte jako akronim. Istnieje teoria mówiąca, że nazwa „spooling” wzięła się od słowa „spool” oznaczającego szpulę (na przykład taśmy magnetycznej/nici)

procesom „drukować” jednocześnie poprzez wirtualizację zasobu, jakim jest drukarka. Proces przejmujący zasób drukarki musi tylko przesłać odpowiednie dane na dysk, po czym zwalnia go i pozwala zająć go innym procesom. Zapewnia to asynchroniczność działania wolnego urządzenia zewnętrznego – dostajemy więc zalety urządzeń asynchronicznych z punktu widzenia systemu operacyjnego (np. przez sygnały / sampling statusu) nie zależnie od faktycznego rodzaju komunikacji z fizycznym urządzeniem. Do tego mamy możliwość zarządzania kolejką – zadania znajdują się na dysku, więc system (lub administrator) może zarządzać kolejnością ich wykonywania, zmieniać priorytety, wstrzymywać lub usuwać zadania przed ich fizycznym przetworzeniem.

Jedną z wad spoolingu jest zużycie przestrzeni dyskowej. Programy zlecające drukowanie przestają być ograniczone prędkością działania drukarki, przez co mogą zlecać bardzo dużo zadań w krótkim czasie (chociaż oczywiście można to ograniczyć). Przez to kolejka może urosnąć na dysku do bardzo dużych rozmiarów, bo drukarka cały czas działa w swoim wolnym tempie. Jeśli proces spoolera ulegnie awarii, niektóre z zadań mogą zostać skoruptowane. Jeżeli dysk zostanie uszkodzony, to zakolejkowane zadania mogą zostać utracone (normalnie byłyby trzymane w pamięci, co jest jednocześnie bardziej i mniej wrażliwe). Niezależnie, w związku z tymi błędami może nastąpić sytuacja, w której daemon spoolera się pomyli i procesy albo zostaną poinformowane o udanym drukowaniu pomimo jego braku, albo druk zostanie zlecony kilkukrotnie (dobry design potrafi temu częściowo zapobiec, ale z problemu dwóch generałów zawsze jest taka możliwość). Do tego potrzeba obsługiwania spoolingu zwiększa złożoność samego systemu, gdyż musimy zlecać i opiekować się stabilnością dodatkowego zbioru daemonów (spoolerów). Co więcej, potrzeba przesyłania danych do kolejki przed wysłaniem ich do urządzenia powoduje, że łączna liczba operacji wykonanych przez procesor staje się większa (ale to nie jest duży problem jako, że i tak pracujemy z urządzeniami wolnymi).

Warto zaznaczyć, że pojęcia spoolingu i buforowania nie są ze sobą tożsame. Buforowanie odnosi się zazwyczaj do wykorzystania (niezbyt dużego) obszaru pamięci RAM w celu wyrównywania różnic prędkości między dwoma urządzeniami podczas transferu. Spooling z kolei wykorzystuje dysk twardy do kolejkowania całych zadań pochodzących z wielu różnych procesów jednocześnie.

III.6.5 Segmentacja i stronicowanie

Segmentacja i stronicowanie - porównaj mechanizmy. Opisz jak te mechanizmy są wykorzystywane na przykładzie wybranego systemu operacyjnego.

Aby sprawnie zarządzać pamięcią, system operacyjny nie udostępnia działającym procesom adresów w pamięci RAM wprost. Zamiast tego działa mechanizm pamięci wirtualnej. We współczesnych architekturach procesora, funkcjonuje on dzięki tak zwanemu **stronicowaniu**. Pamięć (zarówno wirtualna, jak i rzeczywista) jest podzielona na bloki (zwane stronami w przypadku pamięci wirtualnej i ramkami dla pamięci rzeczywistej; zazwyczaj o rozmiarze 4kB). Dokładny sposób działania tego mechanizmu jest opisany przy odpowiednim pytaniu z Programowania Niskopoziomowego. Dzięki stronicowaniu programy nie muszą być trzymane w RAM-ie w ciągłym bloku, co daje systemowi operacyjnemu większą swobodę w zarządzaniu pamięcią. Do tego przesuwanie danych procesu przez system operacyjny jest proste, bo nie wymaga to żadnej zmiany po stronie procesu – wystarczy zmienić mapowanie. Jeśli w RAM-ie brakuje pamięci dla wszystkich działających procesów, to system operacyjny może przenieść część stron tych procesów na dysk. Dzięki temu możliwe jest sprawniejsze radzenie sobie z sytuacjami, gdy pamięć operacyjna jest bardzo silnie wykorzystywana.

W dawnych systemach operacyjnych przestrzenie adresowe były 16-bitowe. To powodowało, że już przy 1MB pamięci RAM brakowało adresów do jej adresowania. Ten problem rozwiązuje się poprzez **segmentację**. Istnieje kilka przestrzeni adresowych (tak zwanych segmentów). Adres (nazywany adresem logicznym) składa się z numeru segmentu, po którym następuje offset od początku tego segmentu. Te dwie informacje pozwalają nam wyznaczyć tak zwany adres liniowy (być może jest to adres wirtualny – dalej możemy stosować stronicowanie).

Segmentacja pozwala na podział pamięci procesu na segmenty odpowiadające logicznym częściom programu (w przeciwieństwie do stronicowania, które wprowadza podział techniczny). Może istnieć na przykład segment kodu, stosu, sterty. Umożliwia to łatwą ochronę pamięci – dla każdego segmentu możemy

określić, czy znajdujące się na nim dane można czytać, pisać do nich, czy może wykonywać je jako kod. W ten sposób możemy na przykład ustalić, że segment kodu jest tylko od odczytu i wykonywania, podczas gdy do stosu i sterty można pisać, ale nie można wykonywać z nich kodu. Wadą segmentacji jest tak zwana fragmentacja zewnętrzna – jeśli alokujemy spójne segmenty (w przypadku nie stosowania stronicowania) różnej długości dla wielu różnych procesów, to w alokacjach powstają „dziury". Wtedy może dojść do sytuacji, w której w RAM-ie jest dużo wolnej pamięci, ale nie ma dużego spójnego fragmentu. Wtedy system operacyjny musi przestawiać procesy, aby zaalokować pamięć dla nowego.

Zaletą stronicowania jest brak fragmentacji zewnętrznej – pamięć może być rozproszona pomiędzy różne ramki. Mamy łatwą realokację procesów i możliwość przenoszenia stron na dysk. Wadą stronicowania jest tak zwana fragmentacja wewnętrzna – ostatnia strona, która należy do danego programu może być wykorzystana w bardzo niedużym stopniu. Wolna pamięć na tej stronie marnuje się – nikt inny nie może z niej skorzystać. Do tego pojawia się narzut związany z translacją adresów wirtualnych na rzeczywiste (z czym pomaga mechanizm cache'u TLB). Duże przestrzenie adresowe wymagają rozbudowanych wielopoziomowych tablic stron, aby sensownie funkcjonować (które same zajmują pamięć).

We współczesnych systemach operacyjnych (na przykład Linux) stosuje się rozwiązania hybrydowe – mamy segmentację, adresy logiczne wskazują na odpowiednie adresy liniowe, które są adresami wirtualnymi tłumaczonymi przez stronicowanie na adresy fizyczne. Linux kładzie nacisk na stronicowanie, segmentacja jest sprowadzona do minimum. Funkcjonuje tak zwany model płaski – wszystkie segmenty mają tę samą, maksymalną możliwą długość, a offset do początku każdego segmentu wynosi 0. To powoduje, że adres liniowy jest identyczny z logicznym. Występują segmenty CS (Code Segment), DS (Data Segment), SS (Stack Segment) oraz ES, FS, GS. FS i GS są używane do danych specyficznych dla wątku (Thread Local Storage) i są wyjątkami od podanej wcześniej reguły (offset do początku segmentu nie musi wynosić 0). Poza tym segmentacja w Linux'ie służy głównie do określenia poziomu uprawnień i rozróżniania kodu jądra (Ring 0) i kodu użytkownika (Ring 3). W procesorze istnieją rejestry segmentowe, np. CS. Dwa najniższe bity rejestru CS określają CPL (Current Privilege Level), czyli bieżący poziom uprawnień procesora. Kod, dla którego CPL jest równe 3 nie może działać na danych z segmentów, które są oznaczone jako 0 (tryb jądra).

Stronicowanie jest podstawowym mechanizmem zarządzania pamięcią w Linux'ie. W danych konkretnej strony trzymane są bity, które określają, jakie uprawnienia ma dana strona (czy można po niej pisać, czytać z niej, wykonywać z niej kod). Dodatkowo strony są dzielone na te należące do jądra i do użytkownika – użytkownik nie ma dostępu do stron jądra. Ten system zapewnia większą precyzję ustawień ochrony niż segmentacja. Dzięki stronicowaniu Linux realizuje koncepcję pamięci na żądanie (demand paging). Jeśli brakuje RAM-u, rzadko używane strony są zrzucane na dysk (do partycji SWAP), a w ich miejsce ładowane są inne. Strona z dysku zostanie z powrotem przeniesiona do RAM-u, gdy program będzie chciał z niej skorzystać.

Aby obsłużyć ogromną 64-bitową przestrzeń adresową bez marnowania pamięci na gigantyczne tablice stron, Linux używa stronicowania wielopoziomowego (w architekturze x86_64 jest to zazwyczaj 5- lub 4-poziomowe stronicowanie). Wielopoziomowa tablica stron jest strukturą drzewiastą, w której tworzymy i przechowujemy tylko te gałęzie, które są aktualnie używane przez program. W systemach 64-bitowych do adresowania pamięci wirtualnej wykorzystuje się de facto 48 bitów (lub 57 w przypadku tablic 5-poziomowych). Te 48 bitów nie jest jednolitym ciągiem – procesor dzieli je na mniejsze kawałki, z których każdy odpowiada za wskazanie indeksu na odpowiednim poziomie tabeli stron. Najbardziej znaczące 9 bitów odpowiada za wskazanie tabeli najwyższego poziomu. Taka tabela może trzymać $2^9 = 512$ wpisów określających tabele niższego poziomu. Każdy wpis ma 8 bajtów, dzięki czeku cała tabela zajmuje 4kB i idealnie mieści się na jednej stronie. We wskazanej przez tabelę pierwszego poziomu tabeli drugiego poziomu wyszukujemy tabelę trzeciego poziomu (na podstawie kolejnych 9 bitów adresu) i tak dalej, aż dojdziemy do konkretnej strony, którą znajdujemy w tabeli najniższego poziomu. Ostatnie 12 bitów adresu to offset na tej stronie.

III.6.6 Mikrojądro a architektura monolityczna

Porównaj monolityczną architekturę systemu operacyjnego z architekturą opartą na mikro jądrze.

W systemie operacyjnym procesor może pracować w **trybie użytkownika** lub w **trybie jądra**.

- Gdy procesor pracuje w **trybie użytkownika**, może wykonywać operacje arytmetyczne i logiczne, pisać oraz czytać z pamięci przypisanej do obecnie wykonywanego procesu. *Nie może robić nic więcej.* Nie ma bezpośredniego dostępu do sprzętu ani pamięci innych procesów.
- Gdy procesor pracuje w **trybie jądra**, może więcej, między innymi obsługiwać wejście/wyjście (I/O), alokować/zwalniać pamięć, komunikować się ze sprzętem (hardware).

Istnieją dwie architektury systemów operacyjnych, które różnią się właśnie podejściem do tego, jak dużo kodu systemu operacyjnego wykonuje się w trybie jądra.

1. **Mikro jądro** to architektura, w której kod wykonywany w trybie jądra odpowiada za najbardziej kluczowe aspekty systemu operacyjnego *i nic więcej.* Przykładem systemu opartego na tej architekturze jest Minix. W Minix'ie jądro realizuje komunikację międzyprocesową (IPC), scheduling procesów, mapowanie pamięci wirtualnej oraz bezpośrednią komunikację z hardware'em.

   Pozostałe funkcjonalności systemu operacyjnego, takie jak zarządzenia procesami, obsługa pamięci, czy wszelkiego rodzaju sterowniki, realizowane są przez tak zwane *serwery* (na przykład PM zarządza pamięcią, VFS obsługuje pamięć). Serwery to specjalne procesy, które odpowiadają za logikę tego co się dzieje w systemie operacyjnym. Pracują one jednak w trybie użytkownika. Aby wykonać niezbędne dla ich działania operacje dostępne tylko w trybie jądra, muszą poprosić o to kod jądra. Ogólny schemat funkcjonowania systemu wygląda tak.

   (a) Użytkownik wysyła syscall do odpowiedniego serwera. Serwer dostaje wiadomość o tym przez IPC (w Minixie realizowane przez message passing). Proces użytkownika jest blokowany w oczekiwaniu na odpowiedź serwera.

   (b) Serwer realizuje żądanie. Gdy potrzebuje wykonać jakieś działanie, za które odpowiada jądro, to wysyła do niego *kernel calla*, który jest odpowiednikiem *syscalla* wykonywanego przez użytkownika. Procesor wtedy wchodzi w tryb uprzywilejowany i wykonuje kod jądra, potem wraca do wykonywania kodu serwera.

   (c) Po skończeniu realizacji żądania serwer wysyła odpowiedź do procesu i zajmuje się realizacją kolejnych żądań. Proces jest odblokowywany (przez jądro – to ono zajmuje się komunikacją między procesami).

2. **Jądro monolityczne** to architektura, w której wszystkie funkcjonalności systemu operacyjnego wykonywane są przez jądro. Istnieją procesy jądra, które domyślnie pracują w trybie jądra. Natomiast procesy użytkownika mogą przełączać się w tryb jądra i wykonywać potrzebny im kod systemu operacyjnego. W ten sposób wszystko funkcjonuje jako jeden organizm, odpowiedzialny za obsługę systemu operacyjnego. Przykładem systemu, który realizuje tę architekturę jest Linux. Jądro monolityczne systemu Linux podzielone jest na subsystemy:

   - I/O subsystem: system plików, komunikacja z hardware'm, komunikacja sieciowa.
   - Memory Management subsystem: pamięć wirtualna, cache, paging.
   - Process management subsystem: scheduling, zarządzanie procesami i wątkami.

   Użytkownik, chcąc wykonać działanie, do którego potrzebny jest tryb uprzywilejowany, wykonuje syscall. Działa to w następujący sposób.

   (a) Proces pracuje sobie i nagle potrzebuje czegoś od systemu operacyjnego. W tym momencie wykonuje instrukcję `syscall`.

   (b) Wykonanie tej instrukcji sprawia, że procesor sprzętowo przełącza się w **tryb jądra** i przekazuje sterowanie do z góry określonego miejsca w kodzie systemu operacyjnego.

(c) System operacyjny (jądro) przejmuje kontrolę i wykonuje swój własny, zaufany kod zadanego syscalla w imieniu tego procesu.

(d) Po zakończeniu obsługi system operacyjny wywołuje instrukcję powrotu, co powoduje sprzętowe przełączenie w **tryb użytkownika** i proces dalej wykonuje swój kod.

Obydwa podejścia są bardzo różne od siebie. Porównując:

- **Mikro jądro** pozwala na bardziej *rozdrobnioną* pracę systemu, co ułatwia obsługę awarii. Jeśli do takiej dojdzie, to w obrębie jednego z serwerów, zatem bez wpływu na pozostałe komponenty systemu. Do tego w trybie jądra pracuje tylko bardzo konkretny, napisany z wielką ostrożnością kod jądra. Programy takie jak instalowane z zewnątrz sterowniki do hardware'u nie mają dostępu do trybu uprzywilejowanego, więc nie mogą aż tak wiele zepsuć w przypadku wystąpienia w nich błędów.

  Głównym minusem takiego podejścia jest to, że rozproszenie wymaga np. udziału jądra w IPC przy realizacji syscalli. Przez to jest większy narzut na system operacyjny, a co za tym idzie niższa wydajność w jego działaniu.

- **Jądro monolityczne** natomiast może efektywnie realizować syscalle (przełączenie w tryb jądra to szybka operacja sprzętowa, nie ma narzutu spowodowanego IPC), kosztem tego, że awarie poszczególnych komponentów dotyczą całego jądra. Błąd (np. Kernel Panic) w sterowniku karty sieciowej powoduje crash całego systemu operacyjnego, ponieważ wszystko działa w jednej przestrzeni pamięci.

W 1992 roku wywiązała się internetowa debata między **Linusem Torvaldsem**, twórcą używanej do dziś rodziny systemów operacyjnych **Linux**, a **Andrew Tanenbaumem**, twórcą systemu **Minix**. Tannenbaum twierdził, że mikro jądro jest lepszą architekturą od jądra monolitycznego i nowoczesne systemy operacyjne powinny ją implementować. Torvalds z kolei obstawał przy architekturze monolitycznej, którą implementował wtedy w Linux'ie. Podczas gdy teoretycy raczej zgadzają się, że mikro jądro jest lepszą architekturą, współczesne systemy operacyjne dalej bazują na architekturze monolitycznej. Historycznie wygrał Linux – głównie dlatego, że ktoś go zaimplementował.

III.6.7 Współdzielenie bibliotek

> Przedstaw mechanizm współdzielenia bibliotek programistycznych (w systemie Linux), uwzględniając odpowiednie metody adresowania.

W przypadku bibliotek programistycznych mamy dwie opcje.

1. **Biblioteki statyczne.** Kod biblioteki jest dołączany do pliku wykonywalnego w fazie linkowania (łączenia), która następuje po kompilacji kodu źródłowego. Program jest scalany z biblioteką w jeden duży plik binarny. Nie zawsze jest to efektywne, bo jak mamy wiele programów korzystających z tej samej biblioteki, to ich kod jest duplikowany w każdym z nich.

2. **Biblioteki dynamiczne.** W przeciwieństwie do bibliotek statycznych, których kod jest duplikowany, biblioteki dynamiczne są skompilowane osobno. Program w fazie linkowania otrzymuje jedynie referencje (symbole), a właściwe łączenie kodu następuje dopiero podczas uruchamiania lub działania programu. Ma to zaletę taką, że oszczędzamy zarówno pamięć na dysku, jak i pamięć operacyjną naszych programów.

Biblioteki dynamiczne mają rozszerzenie `.so`. Za ich przyłączanie do programów odpowiada system operacyjny, który ładuje do pamięci operacyjnej bibliotekę współdzieloną w momencie, gdy zostanie uruchomiony program, który z niej korzysta. Dokładniej, za ładowanie bibliotek dynamicznych i ich przyłączanie do przestrzeni adresowej procesu odpowiada specjalny program systemowy – linker dynamiczny (zazwyczaj `ld.so` lub `ld-linux.so`), który jest uruchamiany przez jądro systemu operacyjnego przed przekazaniem kontroli do właściwego programu.

Procesy w systemie operacyjnym nie pracują na rzeczywistej pamięci RAM, tylko na tak zwanej pamięci wirtualnej. Dzieje się to za pomocą mechanizmu **stronicowania**. Pamięć (zarówno wirtualna, jak i

rzeczywista) jest podzielona na bloki (zwane stronami). W procesorze istnieje jednostka hardware'owa Memory Management Unit (MMU), która dokonuje mapowania bloków wirtualnych na rzeczywiste. Dzięki temu dane procesu nie muszą być trzymane w RAMie w ciągłym bloku, co daje systemowi operacyjnemu większą swobodę w zarządzaniu pamięcią. Do tego przesuwanie danych procesu przez system operacyjny jest proste, bo nie wymaga to żadnej zmiany po stronie procesu – wystarczy zmienić mapowanie w MMU.

Ten mechanizm umożliwia też korzystanie z bibliotek współdzielonych (dynamicznych). W rzeczywistym RAMie istnieje tylko jedna kopia takiej biblioteki. Strony ją zawierające zostają przypisane do pewnych stron w pamięci wirtualnej procesu, dzięki czemu proces ten może korzystać z kodu takiej biblioteki.

Segment kodu (`.text`) oraz danych tylko do odczytu (`.rodata`) biblioteki są mapowane z uprawnieniami Read-Only oraz Execute. Dzięki temu wiele programów może czytać ten sam kod naraz i nie ma żadnych race conditions ani nieprzewidzianych zachowań. Segment danych modyfikowalnych (`.data` oraz `.bss`) biblioteki zawiera zmienne globalne i statyczne. Strony te są mapowane jako Copy-on-Write (CoW). Dopóki procesy tylko czytają zmienne globalne, współdzielą te same fizyczne strony. Gdy któryś proces spróbuje zmodyfikować zmienną, MMU zgłasza wyjątek, a system operacyjny tworzy prywatną kopię tej strony RAM dla danego procesu. Dzięki temu zmienne globalne biblioteki pozostają odizolowane pomiędzy procesami.

Mechanizm stronicowania powoduje, że kod biblioteki współdzielonej znajduje się pod różnymi adresami (wirtualnymi) dla różnych procesów. Dlatego biblioteka nie może stosować bezwzględnych adresów w swoim kodzie. Taka architektura wymusza na bibliotece **adresowanie względne**. Pisząc bibliotekę współdzieloną możemy założyć o niej jedynie, że zostanie zmapowana na spójny fragment pamięci wirtualnej procesu. Dlatego wewnątrz biblioteki (przy wywoływaniu funkcji/skokach warunkowych) nie odwołujemy się do konkretnych adresów, zamiast tego przesuwamy adres z rejestru `RIP` o określoną odległość – skaczemy po kodzie korzystając z faktu, że znamy relatywne offsety pomiędzy instrukcjami naszego kodu. W ten sposób piszemy *Position Independent Code* (PIC), który działa niezależnie od tego, na jakie adresy biblioteka zostanie zmapowana.

Kod binarny programów korzystających z bibliotek ładowanych dynamicznie nie wie, pod jakie adresy ma skoczyć, aby skorzystać z funkcji bibliotecznej. Aby temu zaradzić istnieje **PLT (Procedure Linkage Table)**. Jest to sekcja kodu naszego programu, która zawiera wskaźniki do funkcji importowanych dynamicznie. Pod tymi wskaźnikami znajduje się kod linkera dynamicznego, który w razie potrzeby ładuje bibliotekę i uzupełnia referencje, a potem skacze do odpowiedniego kodu (znajdującego się w opisanej niżej GOT). Aby wywołać funkcję z biblioteki dynamicznej w kodzie Assemblera piszemy na przykład `call printf@plt`, co znaczy, że szukany symbol jest w sekcji PLT.

Podobnie wygląda sytuacja ze zmiennymi globalnymi bibliotek. Istnieje **GOT (Global Offset Table)** – tablica wskaźników umieszczona w prywatnym segmencie danych procesu, trzymająca faktyczne adresy zmiennych. W czasie ładowania programu linker dynamiczny wyznacza rzeczywiste adresy zmiennych globalnych i zapisuje je w tablicy GOT. Nasz kod (a w zasadzie zazwyczaj kod biblioteki, bo przeważnie to on odwołuje się do zmiennych globalnych biblioteki), chcąc odczytać zmienną globalną, używa adresowania względnego i odwołuje się do tablicy GOT. W Assemblerze wygląda to tak: `mov rax, [rip + zmienna@GOTPCREL]`.

III.6.8 Problem ucztujących filozofów

Na przykładzie problemu ucztujących filozofów przedyskutuj pojęcia poprawności pod względem bezpieczeństwa i żywotności. Zaproponuj rozwiązanie spełniające oba te warunki.

Problem ucztujących filozofów wygląda następująco. Pięciu filozofów siedzi przy okrągłym stole. Każdy z nich naprzemiennie myśli i je. Między filozofami leży 5 widelców (po jednym z lewej i prawej strony każdego filozofa). Do zjedzenia posiłku filozof potrzebuje dwóch widelców jednocześnie. Zastanawiamy się, jak zsynchronizować dostęp do ograniczonych, dzielonych zasobów (widelców), aby nie doszło do konfliktów.

Naszym celem jest zapewnienie bezpieczeństwa systemu, to znaczy zagwarantowanie, że żaden zasób (widelec) nie będzie używany przez dwie osoby jednocześnie (w przypadku procesów pracujących na danych taką sytuację nazywamy race condition, może ona spowodować naruszenie integralności danych i niepoprawność programu). Jednocześnie chcemy utrzymać żywotność naszego systemu – chcemy, aby system robił postępy w działaniu (posiłki były zjadane) i żaden proces (filozof) nie czekał w nieskończoność. Nie chcemy dopuścić do zakleszczeń (deadlock), czyli sytuacji, gdy procesy czekają na siebie nawzajem w cyklu. Chcemy też uniknąć zagłodzenia (starvation) procesu, czyli sytuacji, gdy proces nigdy nie otrzyma zasobów, jakich potrzebuje do wykonania swojej pracy.

Zabraniamy sytuacji, w których różni filozofowie trzymają ten sam widelec (to założenie wydaje się być oczywiste, ale przy implementowaniu odpowiedniego kodu wcale takie nie jest; rozwiązujemy ten problem stosując locki/mutexy). Teraz musimy tylko zapewnić żywotność systemu. W najprostszym rozwiązaniu filozof podnosi kolejno lewy i prawy widelec, a jeśli nie może podnieść widelca, to czeka, aż będzie mógł. To rozwiązanie nie działa – wszyscy filozofowie mogą podnieść swój lewy widelec, wtedy każdy czeka na widelec po swojej prawej i występuje deadlock. Innym pomysłem jest ponumerowanie widelców i zaczynanie zawsze od podniesienia widelca o mniejszym numerze. Wtedy nie może zajść sytuacja, że każdy filozof ma po jednym widelcu, a więc ktoś będzie miał dwa i system będzie działał. Mimo to może dojść do zagłodzenia – istnieje dwóch sąsiednich filozofów, którzy zaczynają od podniesienia tego samego widelca. Może się zdarzyć tak, że zawsze to jeden z nich będzie dostawał ten widelec, a drugi nie dostanie go nigdy.

Poprawne rozwiązanie tego problemu wygląda tak.

- Każdy widelec jest w każdym momencie przez kogoś trzymany.
- Zaczynamy od stanu, który nie jest symetryczny, to znaczy takiego, w którym co najmniej jeden filozof trzyma oba swoje widelce.
- Każdy filozof po zakończeniu jedzenia przekazuje oba swoje widelce odpowiednim sąsiadom.

Powyższe rozwiązanie utrzymuje niezmiennik, że stan nie jest symetryczny – po pozbyciu się widelców przez filozofa mamy stan, w którym pewien filozof ma 0 widelców, a więc pewien musi mieć 2. Zatem nigdy nie nastąpi deadlock, bo zawsze ktoś jest w stanie jeść. Nie nastąpi też zagłodzenie – gdyby pewien filozof X miał nigdy nie dostać swojego lewego widelca to znaczy, że jego lewy sąsiad Y trzyma ten widelec (dla niego prawy) i nigdy nie dostanie swojego lewego widelca od swojego lewego sąsiada. Kontynuując to rozumowanie dostajemy cykl, w którym każdy filozof trzyma swój prawy widelec – sprzeczność z tym, że stan nie jest symetryczny. Podobnie sprawa wygląda dla prawego widelca X.

Inne pomysły: można wprowadzić „kelnera”, czyli osobny proces, który komunikuje się z filozofami i przekazuje im oba potrzebne im widelce, nie dając tego samego widelca różnym filozofom jednocześnie. Jeśli kelner trzyma filozofów w kolejce i przekazuje widelce filozofowi na przodzie kolejki, to nie dojdzie do zagłodzenia. Można też rozważać system, w którym istnieje „token” przekazywany cyklicznie na okręgu. Tylko filozof posiadający token może w danym momencie jeść. Po zjedzeniu przekazuje token dalej. To rozwiązanie jest poprawne, ale znacząco zmniejsza wydajność systemu – tylko jeden proces (filozof) pracuje w danym momencie.

Posłowie

Nauka na egzamin licencjacki nie jest zadaniem łatwym ani przyjemnym. Sama ilość materiału, który należy przed takim egzaminem opanować, potrafi być przytłaczająca. Sporządzenie notatek wspomagających taką naukę, mimo że samo w sobie jest bardzo skuteczną metodą przyswajania wiedzy, wymaga jeszcze więcej wysiłku i czasu spędzonego zgłębiając dany temat, zwłaszcza, gdy naszym celem jest stworzenie opracowania w miarę kompletnego, nieodwołującego się do zewnętrznych źródeł i zrozumiałego dla sporego grona odbiorców.

Z tego miejsca chciałbym podziękować wszystkim moim Współautorom. To dzięki Waszemu zaangażowaniu i ciężkiej pracy udało się doprowadzić ten projekt do końca. Wiem, że pracując w pojedynkę nie miałbym czasu, siły ani chęci, aby stworzyć coś podobnego. Mówi się, że uprawianie matematyki jest zajęciem społecznym, u którego podstaw leży wzajemna współpraca i ciągła wymiana pomysłów oraz idei. Moim zdaniem możemy być z siebie dumni jako matematycy (choć niektórzy z nas wolą się tytułować informatykami). Wierzę, że to opracowanie przysłuży się wielu naszym kolegom i koleżankom, którzy będą zmagać się z tymi samymi zagadnieniami w przyszłości.

Gdy pytają mnie, czy dobrze jest studiować TCS, odpowiadam, że moim zdaniem to nie ma tak, że dobrze albo że niedobrze. Gdybym miał powiedzieć, co cenię w życiu najbardziej, powiedziałbym, że ludzi. Otóż ludzi poznanych na TCSie bardzo sobie cenię. Teraz, gdy ten rozdział naszego życia się zamyka, chciałbym Wam wszystkim podziękować za te wspólnie spędzone trzy lata. Wiem, że wiele się od Was nauczyłem i uważam, że stałem się dzięki Wam lepszą osobą. Dziękuję za wszystkie uśmiechy, dobre słowa i radosne chwile, które na pewno pozostaną ze mną na długo. Dziękuję też za wszystkie godziny (żeby tylko) spędzone wspólnie walcząc ze studiami, załamując się i kwestionując swoje wybory życiowe. Liczę na to, że patrząc wstecz nie będziemy myśleć o tych trudach, a jedynie wspominać to, co było dobre. Szczególnie chciałbym podziękować mieszkańcom pewnego fikcyjnego miasta leżącego na granicy województwa małopolskiego i zachodniopomorskiego (ciężko powiedzieć, po której stronie granicy dokładnie). Wasza obecność i wsparcie były naprawdę nieocenione. Powodzenia i do zobaczenia.

Maciej Mikołajczak