Te materiały można znaleźć pod adresem:

`http://www.is.umk.pl/~kg/zajecia/WdI.pdf`

Bibliografia

- ★ Richard Neapolitan, Kumars Naimipour. Podstawy algorytmów z przykładami w C++. Helion 2004.
- ★ Niklaus Wirth. Algorytmy + struktury danych = programy. WNT 1980-2004. Siedem wydań.
- ★ David Harel. Rzecz o istocie informatyki. WNT 1992-2001. Cztery wydania.
- ★ Jon Bentley. Perełki oprogramowania. WNT 1992.

Plan

1. Trochę historii 3
2. Reprezentacja danych 4
3. Maszyna Turinga 19
4. Algorytmy, schematy blokowe 18, 22
5. Złożoność obliczeniowa 32
6. Różne języki programowania i techniki programowania 45
7. Metody sortowania 67
8. Struktury danych 84
9. Grafy i algorytmy oparte na grafach 92

Historia informatyki

- http://www.is.umk.pl/~duch/books-fsk/historia/historia.html
- http://www.is.umk.pl/~kg/zajecia/Wstep/HIST-slajdy.pdf

Reprezentacja liczb

★ Systemy liczenia: dwójkowy (binarny), ósemkowy, dziesiętny, szesnastkowy.

system	cyfry	przykład	dziesiętnie
dwójkowy	0 1	101	$1*2^2+1*2^0=$ $1*4+1=5$
ósemkowy	0 1 2 3 4 5 6 7	270	$2*8^2+7*8^1=$ $2*64+7*8=184$
dziesiętny	0 1 2 3 4 5 6 7 8 9	309	$3*10^2+9*10^0=$ $3*100+9=309$
szesnastkowy	0 1 2 3 4 5 6 7 8 9 A B C D E F	10A	$1*16^2+10*16^0=$ $1*256+10=266$

Rozmiary danych

- ★ bit (b) - ang. binary unit, przechowuje wartość 0 lub 1
- ★ bajt (B) = 8 bitów; $2^8 = 256$ różnych reprezentacji,
  np liczby 0, 1, . . . , 255 lub liczby -128, . . . , 127
- ★ kilobajt: 1 kB = 2^{10} B = 1024 B
- ★ megabajt: 1 MB = 2^{10} kB = 2^{20} B = 1 048 576 B
- ★ gigabajt: 1 GB = 2^{10} MB = 2^{30} B = 1 073 741 824 B
- ★ terabajt: 1 TB = 2^{10} GB = 2^{40} B = 1 099 511 627 776 B
- ★ 1 Mb = 128 kB (= 1024/8 kB)
- ★ 1 MB = 8 Mb

Inne proste typy danych

- ★ Typ logiczny: {fałsz, prawda}, {0, 1}
  - ▶ niestety więcej niż 1 bit
- ★ Typ wyliczeniowy:
  - ▶ Kształt = {okrągły, trójkątny, prostokątny, . . . }
  - ▶ Dział = {Produkcja, Techniczny, Laboratorium, . . . }

Reprezentacja tekstu

- ★ ASCII - 7 bitów, rozszerzenia do 8 bitów
  - ► znaki przejścia do nowej linii: CR (carriage return), LF (line feed)
  - ► odstęp, znak tabulacji
- ★ strony kodowe dla znaków narodowych DOS CP852, Windows cp1250, ISO-8859-2 . . .
- ★ Unicode, Unicode Transformation Format (UTF)
  - ► UTF-16
  - ► UTF-8
- ★ typ string w różnych językach programowania

Dane ponadjednobajtowe stwarzają dylemat **big-endian/little-endian**, czyli od której strony zacząć.

Systemy stałopozycyjne

- ★ ściśle określona liczba cyfr przed i po przecinku
- ★ $123.45 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2}$
- ★ $101.01 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2}$
- ★ w praktyce tylko dla liczb całkowitych

Systemy zmiennopozycyjne (zmiennoprzecinkowe)

- ★ mantysa i cecha, liczba $=$ mantysa $* 2^{\text{cecha}}$ $18.5 = 0.185 * 10^2 = 0.100101 * 2^{101}$
- ★ rozdzielczość i skończoność komputerowych liczb rzeczywistych
- ★ typy 4-bajtowe (3 bajty mantysy i 1 bajt cechy), 6-bajtowe, 8-bajtowe (podwójnej precyzji)
- ★ należy być ostrożnym przy dodawaniu małej liczby zmiennoprzecinkowej do dużej (mała może zniknąć)
- ★ wartości, które nie są poprawnymi liczbami (NAN - not a number)
- ★ standard IEEE 754 – dobry opis pod http://research.microsoft.com/∼hollasch/cgindex/coding/ieeefloat.html

Standard IEEE 754

Reprezentacja bitowa

precyzja	znak	wykładnik	część ułamkowa	przesunięcie
pojedyncza	1 [31]	8 [30-23]	23 [22-00]	127
podwójna	1 [63]	11 [62-52]	52 [51-00]	1023

- ★ reprezentacja binarna – podstawa równa 2
- ★ znak – 0 - liczba dodatnia, 1 - liczba ujemna
- ★ wykładnik i przesunięcie – rzeczywista cecha = wykładnik - przesunięcie
- ★ mantysa
  - ► *postać znormalizowana liczby* – w mantysie przecinek po pierwszej niezerowej cyfrze ($1 \leqslant m < 10$)
    Przykład: $1.25 * 10^2$, nie $0.125 * 10^3$ ani $125 * 10^0$
  - ► w systemie dwójkowym niezerowa znaczy jedynka
  - ► pierwszy bit mantysy zawsze równy 1 – ukryty, reszta to część ułamkowa

Standard IEEE 754 – przykłady

23.75

- ★ postać binarna: $10111.11 = 2^4 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2}$
- ★ po normalizacji: $1.011111 * 2^4 = 1.011111 * 2^{100}$
- ★ znak 0
  wykładnik z przesunięciem $4 + 127 = 131 = 10000011$
  część ułamkowa 011111
- ★ ostatecznie mamy $0\ 10000011\ 01111100000000000000000$
  czyli $01000001\ 10111110\ 00000000\ 00000000$
  szesnastkowo $0x41BE0000$

-0.7

- ★ postać binarna: $-0.1011001100110011...$
- ★ znak = 1, wykładnik $= -1$, $127 - 1 = 126 = 01111110$
- ★ ostatecznie mamy $1\ 01111110\ 01100110011001100110011$
  czyli $10111111\ 00110011\ 00110011\ 00110011$
  szesnastkowo $0xBF333333$

Standard IEEE 754 – c.d.

Wartości specjalne – wykładniki z samych jedynek bądź z samych zer

znak	wykładnik (w)	cz. ułamkowa (u)	wartość
0	00..00	00..00	+0
0	00..00	00..01 – 11..11	dodatnia zdenorm. $0.u * 2^{-p+1}$
0	00..01 – 11..10	00..00 – 11..11	dodatnia znorm. $1.u * 2^{w-p}$
0	11..11	00..00	+Infinity (nieskończoność)
0	11..11	00..01 – 01..11	SNaN (Signalling Not A Number)
0	11..11	10..00 – 11..11	QNaN (Quiet Not A Number)
1	00..00	00..00	-0
1	00..00	00..01 – 11..11	ujemna zdenorm. $-0.u * 2^{-p+1}$
1	00..01 – 11..10	00..00 – 11..11	ujemna znorm. $-1.u * 2^{w-p}$
1	11..11	00..00	-Infinity (nieskończoność)
1	11..11	00..01 – 01..11	SNaN
1	11..11	10..00 – 11..11	QNaN

Standard IEEE 754 – c.d.

Operacje specjalne:

Operacja	Wynik
n / ±Infinity	0
±Infinity x ±Infinity	±Infinity
±nonzero / 0	±Infinity
Infinity + Infinity	Infinity

Operacja	Wynik
±0 / ±0	NaN
Infinity - Infinity	NaN
±Infinity / ±Infinity	NaN
±Infinity x 0	NaN

Zakresy dwójkowo:

precyzja	zdenormalizowane	znormalizowane
pojedyncza	$\pm 2^{-149} \ldots (1-2^{-23}) * 2^{-126}$	$\pm 2^{-126} \ldots (2-2^{-23}) * 2^{127}$
podwójna	$\pm 2^{-1074} \ldots (1-2^{-52}) * 2^{-1022}$	$\pm 2^{-1022} \ldots (2-2^{-52}) * 2^{1023}$

Zakresy dziesiętnie:

precyzja	dziesiętnie
pojedyncza	$\pm \sim 10^{-44.85} \ldots \sim 10^{38.53}$
podwójna	$\pm \sim 10^{-323.3} \ldots \sim 10^{308.3}$

Reprezentacja obrazu

- ★ kodowanie kolorów
  - ► RGB - red, green, blue
  - ► HSV - hue, saturation, value (barwa, nasycenie, jaskrawość)
  - ► CMYK - cyan, magenta, yellow, key (black)
- ★ mapy bitowe
- ★ kompresja obrazów (GIF, JPEG, . . .)
- ★ obrazy ruchome
  - ► zbiór niezależnych klatek (np. MotionJPEG)
  - ► różnice pomiędzy klatkami

Grafika wektorowa

Przykład SVG (Scalable Vector Graphics)

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <svg xmlns="http://www.w3.org/2000/svg" version="1.0"
3    width="400" height="250" viewBox="0 0 800 500">
4       <rect style="fill:white;stroke:#000000;stroke-width:6;"
5          width="800" height="500" x="0" y="0"/>
6       <rect fill="crimson" width="800" height="250"
7          x="0" y="250"/>
8 </svg>
```

Reprezentacja dźwięku

- ★ również mnóstwo formatów
- ★ również konwersja sygnału analogowego na cyfrowy
- ★ PCM (Pulse-code modulation)
  - ► próbkowanie (*sampling*) z pewną (najczęściej stałą) częstotliwością - 48kHz w DVD, 44.1 kHz w CD
  - ► kwantyzacja - 16 bitowa to 65536 różnych poziomów
  - ► modulacja i demodulacja
  - ► DPCM - modelowanie różnic
  - ► ADPCM - zmienny krok kwantyzacji
- ★ RIFF (Resource Interchange File Format) - meta format z kawałków (*chunks*) wykorzytany w formatach WAV, AU
- ★ kompresja stratna (np. MP3, ACC, Ogg Vorbis)
- ★ kodeki - kodowanie i dekodowanie strumieni danych

Informacja i jej reprezentacje

Ta sama informacja może mieć wiele różnych reprezentacji, np:

- ★ liczba $12_D = C_H = 1100_B$ (D - decimal, dziecietnie, H - hexadecimal, szesnastkowo, B - binary, dwójkowo)
- ★ „temperatura wynosi 22° C” $\equiv$ „temperatura wynosi $71,6^\circ$ F”
- ★ „prędkość 120 km/h” $\approx$ „prędkość 74,6 mil/h”

Informacja to pojęcie ogólne obejmujące nie tylko **dane** ale i **metody przetwarzania danych** (**algorytmy**)

Każdy algorytm może być zapisany na wiele różnych sposóbów:

- ★ różne języki naturalne
- ★ różne języki formalne (w tym języki programowania)
- ★ różne środki tego samego języka programowania

Algorytmy

Algorytm to przepis na uzyskanie odpowiednich danych (**wyjściowych**) startując z odpowiednich danych (**wejściowych**).

Każdy algorytm powinien precyzować:

- ★ wymagane dane wejściowe (*input*)
- ★ dane wyjściowe (*output*)
- ★ sposób generowania danych wyjściowych z danych wejściowych

Obowiązuje pełna precyzja!

Maszyna Turinga

Warto odwiedzić:

- ★ http://edu.i-lo.tarnow.pl/inf/prg/003_mt/0001.php
- ★ http://pl.wikipedia.org/wiki/Maszyna_Turinga
- ★ http://olszewski.info.pl/files/prezentacje/informatyczne/MaszynaTuringa.pdf

Nieformalny opis:

- ★ ***Maszyna Turinga*** to prosty teoretyczny model uniwersalnej maszyny liczącej.
- ★ Zbudowana jest z **(nieskończonej) taśmy**, z której odczytuje się i na której zapisuje się dane oraz **głowicy** poruszającej się po taśmie i odczytującej z niej oraz zapisującej na niej symbole zgodnie z realizowanym programem.
- ★ ***Program maszyny Turinga*** to zbiór instrukcji informujących o operacjach jakie należy wykonać w zależności od aktualnego stanu maszyny i odczytu głowicy.

Maszyna Turinga formalnie

Maszyna Turinga to trójka $MT = \langle Q, \Gamma, \delta \rangle$, gdzie:

- ★ Q to skończony zbiór stanów,
- ★ Γ to skończony zbiór dopuszczalnych symboli,
- ★ $\delta : D \to Q \times \Gamma \times \{L, P, -\}$, gdzie $D \subseteq Q \times \Gamma$ to funkcja będąca realizowanym programem. W zależności od stanu bieżącego maszyny oraz odczytanego z taśmy symbolu, maszyna przyjmuje nowy stan, zapisuje odpowiedni symbol oraz przesuwa się w lewo, prawo lub pozostaje bez przesunięcia.

Czasami, już w definicji, wyróżnia się dodatkowo:

- ★ $q_0 \in Q$ - stan początkowy maszyny,
- ★ $F \subset Q$ - zbiór stanów końcowych,
- ★ $b \in \Gamma$ - tzw. symbol pusty,
- ★ $\Sigma \subset \Gamma - \{b\}$ - zbiór symboli końcowych.

Maszyna Turinga – przykłady

★ Odwracanie bitów (zakładamy, że głowica stoi na końcu ciągu danych)

$MT = \langle \{q_0\}, \{0, 1, ?\}, \delta \rangle$

$$\langle q_0, 0 \rangle \overset{\delta}{\mapsto} \langle q_0, 1, L \rangle$$
$$\langle q_0, 1 \rangle \overset{\delta}{\mapsto} \langle q_0, 0, L \rangle$$

★ Dodawanie 1 (zakładamy, że głowica stoi na końcu liczby)

$MT = \langle \{q_0, q_1\}, \{0, 1, ?\}, \delta \rangle$

$$\langle q_0, 0 \rangle \overset{\delta}{\mapsto} \langle q_1, 1, L \rangle$$
$$\langle q_0, 1 \rangle \overset{\delta}{\mapsto} \langle q_0, 0, L \rangle$$
$$\langle q_0, ? \rangle \overset{\delta}{\mapsto} \langle q_1, 1, L \rangle$$

Interpretacja stanów:

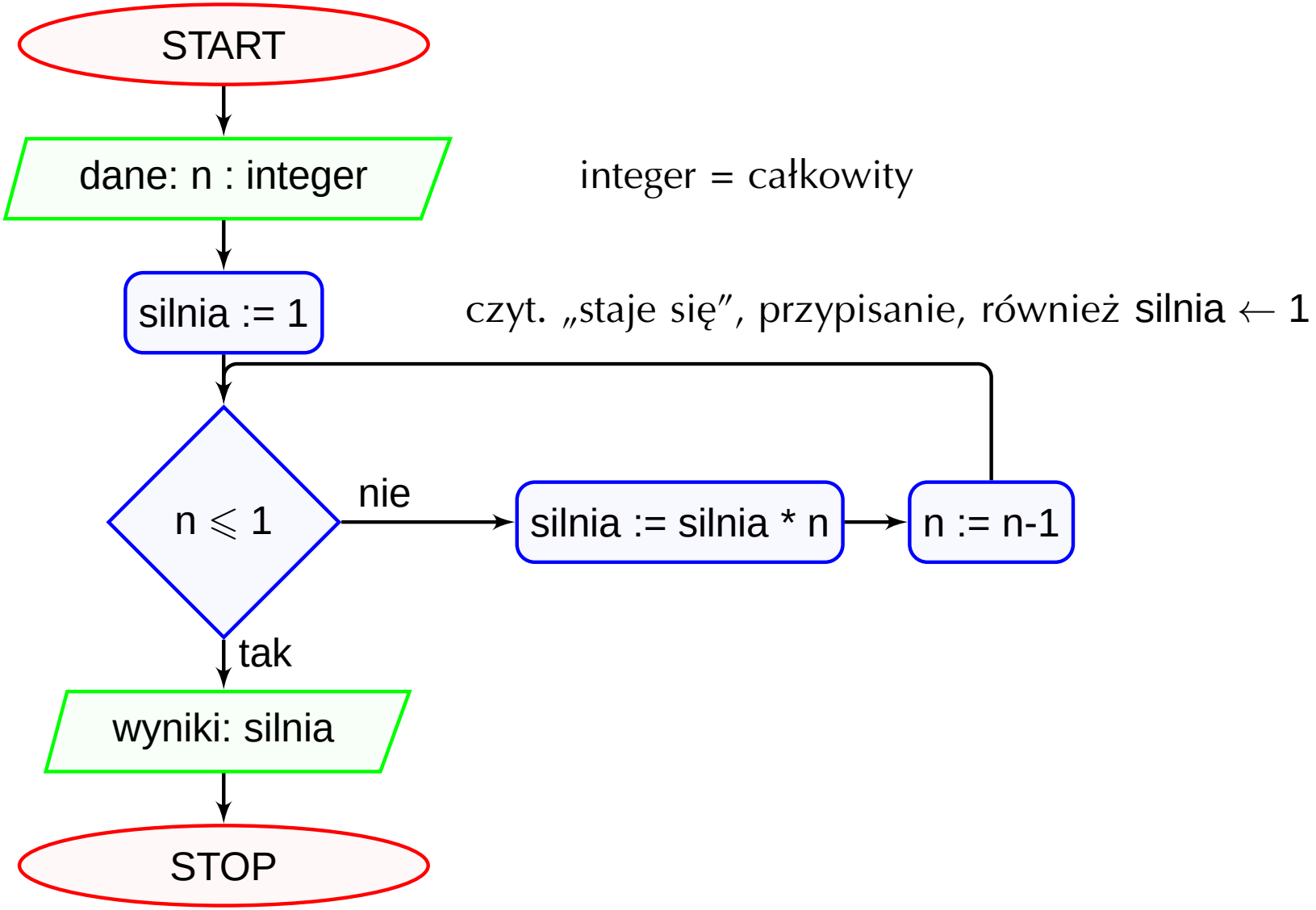
- q_0 – dodajemy jedynkę do bieżącej pozycji
- q_1 – zadanie zakończone

★ Sprawdzenie parzystości i dopisanie bitu parzystości

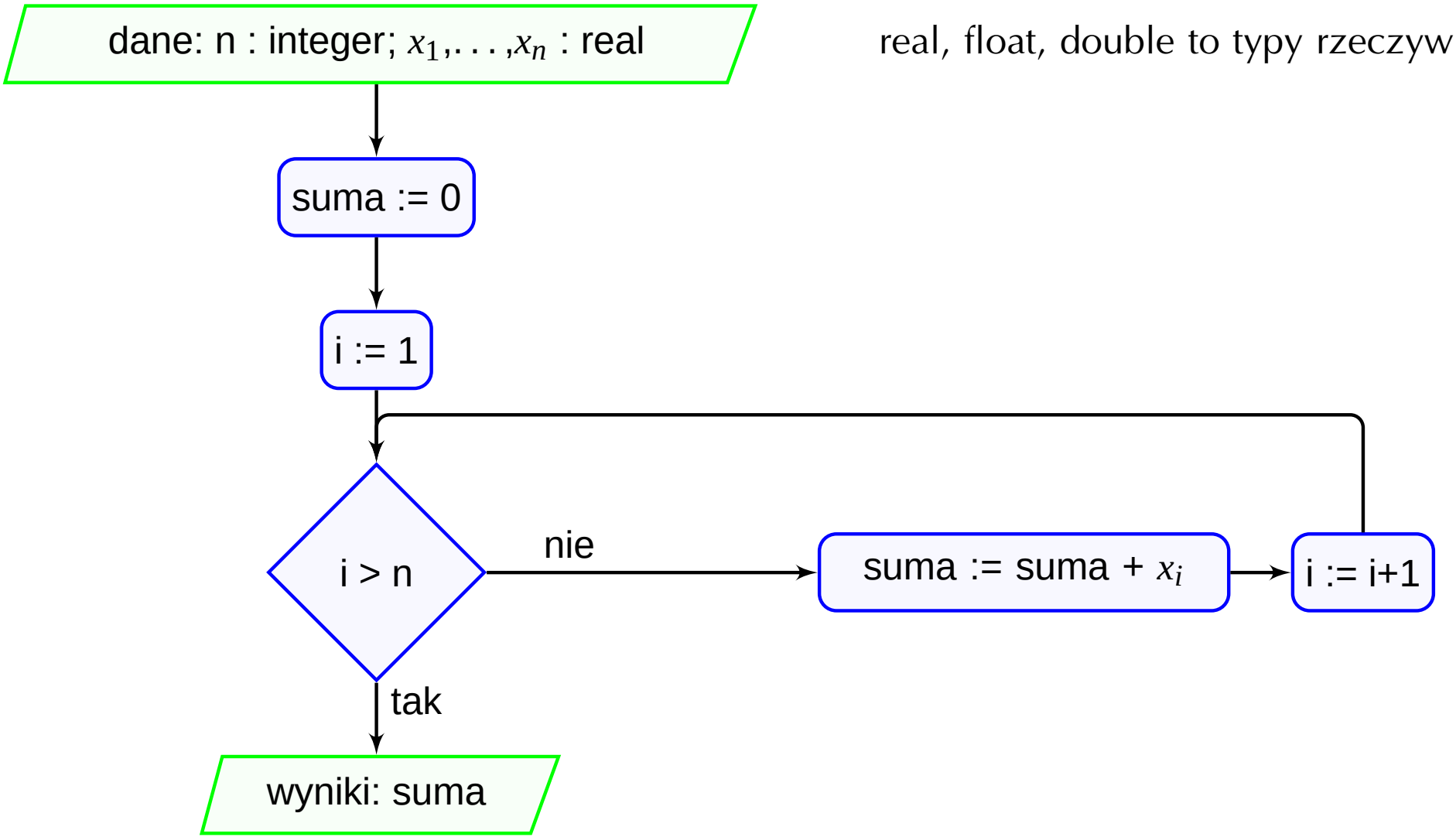
★ Podwojenie ciągu

Schematy blokowe

Jeden ze sposobów przedstawiania algorytmów (język wizualny).

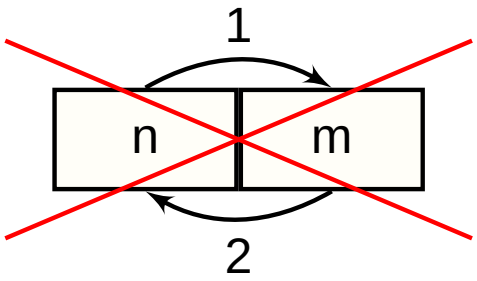


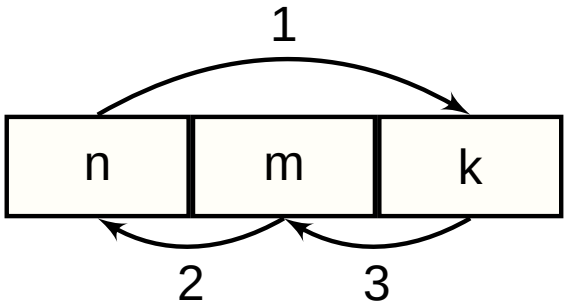
Schemat blokowy – suma n liczb

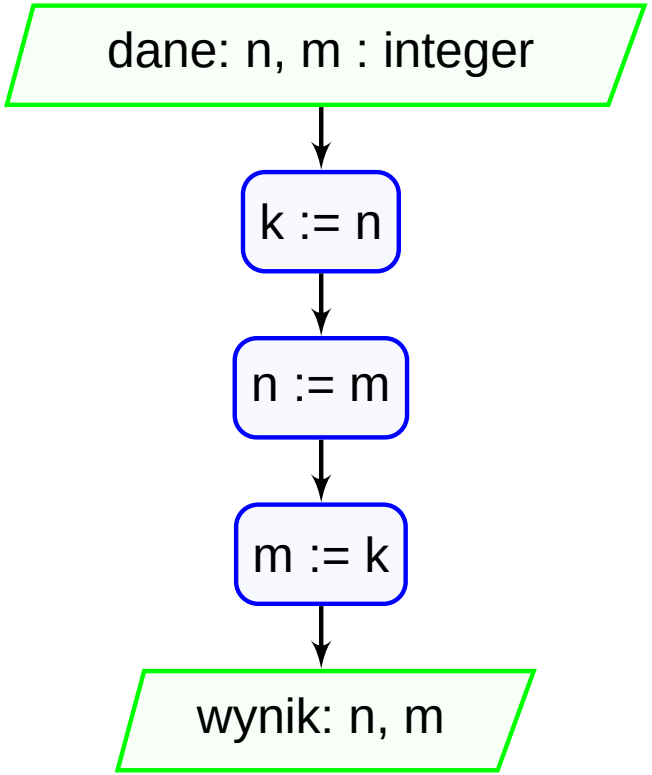


real, float, double to typy rzeczywiste

Zamiana wartości zmiennych








```
1 procedure zamiana(var n, m : integer);
2 var k : integer;
3 begin
4    k := n;
5    n := m;
6    m := k;
7 end;
```

Największy wspólny dzielnik (NWD)

Dzielenie całkowite i reszta z dzielenia

Jeśli $n, m \neq 0$ są całkowite, to istnieją (jednoznacznie) liczby całkowite d oraz $0 \leqslant r < m$ takie, *że*

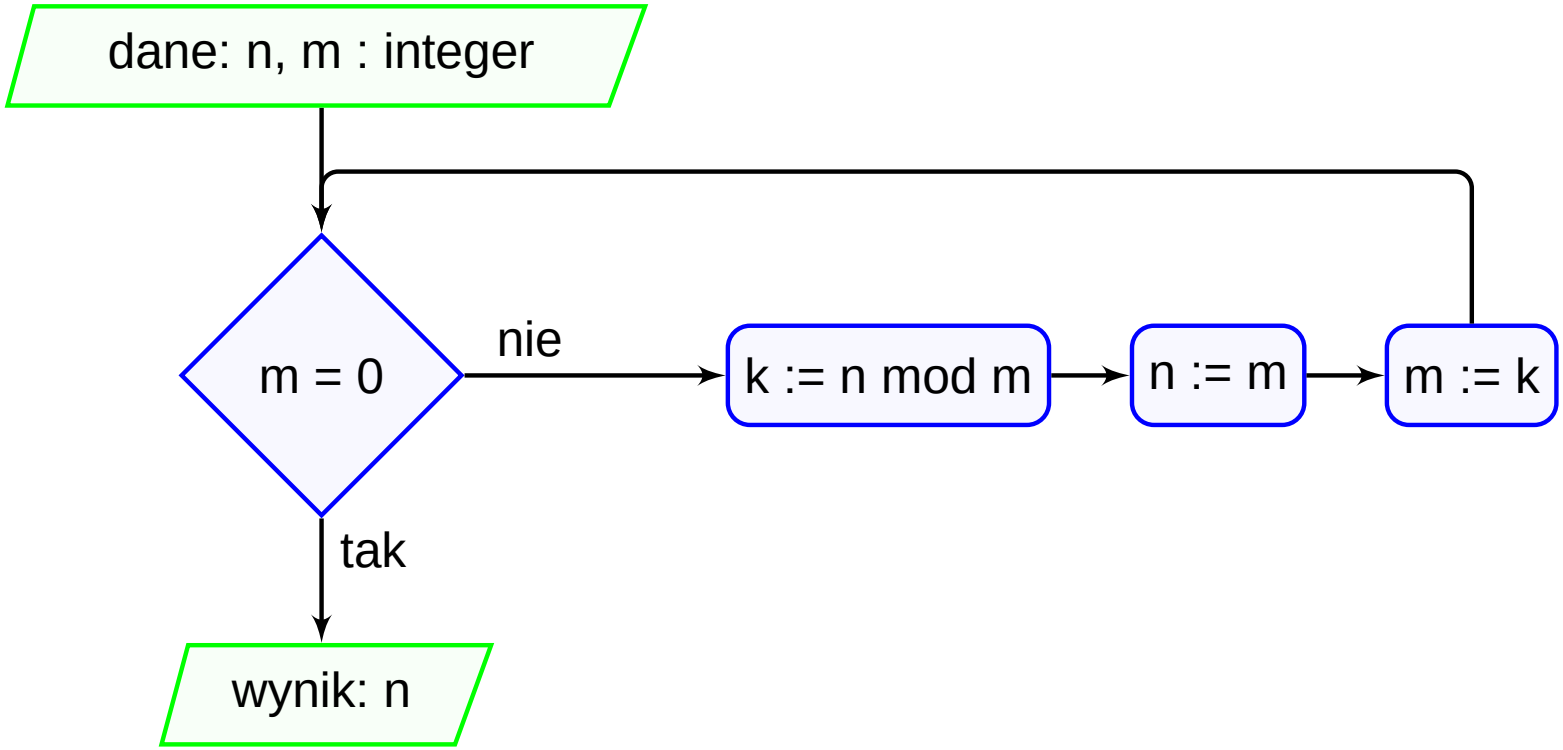
$$n = m * d + r.$$

Mówimy wówczas, *że* d jest wynikiem dzielenia całkowitego n przez m ($d = n \operatorname{div} m$) oraz, *że* r jest resztą z dzielenia n przez m ($r = n \bmod m$, czyt. *modulo*).

Algorytm Euklidesa wyznaczający największy wspólny dzielnik (NWD) dwóch liczb naturalnych wykorzystuje następującą własność:

> Jeśli n, m są całkowite, to
>
> $$\mathrm{NWD}(n,m) = \begin{cases} n & \text{jeśli } m = 0, \\ \mathrm{NWD}(m, n \bmod m) & \text{jeśli } m \geqslant 1, \end{cases}$$
>
> gdzie $n \bmod m$ to reszta z dzielenia n przez m.

NWD – schemat blokowy



Konwersja liczby do innej bazy

Zadanie:

Szukamy reprezentacji liczby naturalnej n w systemie o bazie b, czyli liczby k oraz cyfr $c_0, \ldots, c_k \in \{0, \ldots, b-1\}$, takich, że $n = (c_k \ldots c_0)_b$, czyli

$$n = c_k * b^k + \cdots + c_1 * b^1 + c_0 * b^0.$$

Podpowiedź:

Zauważmy, że

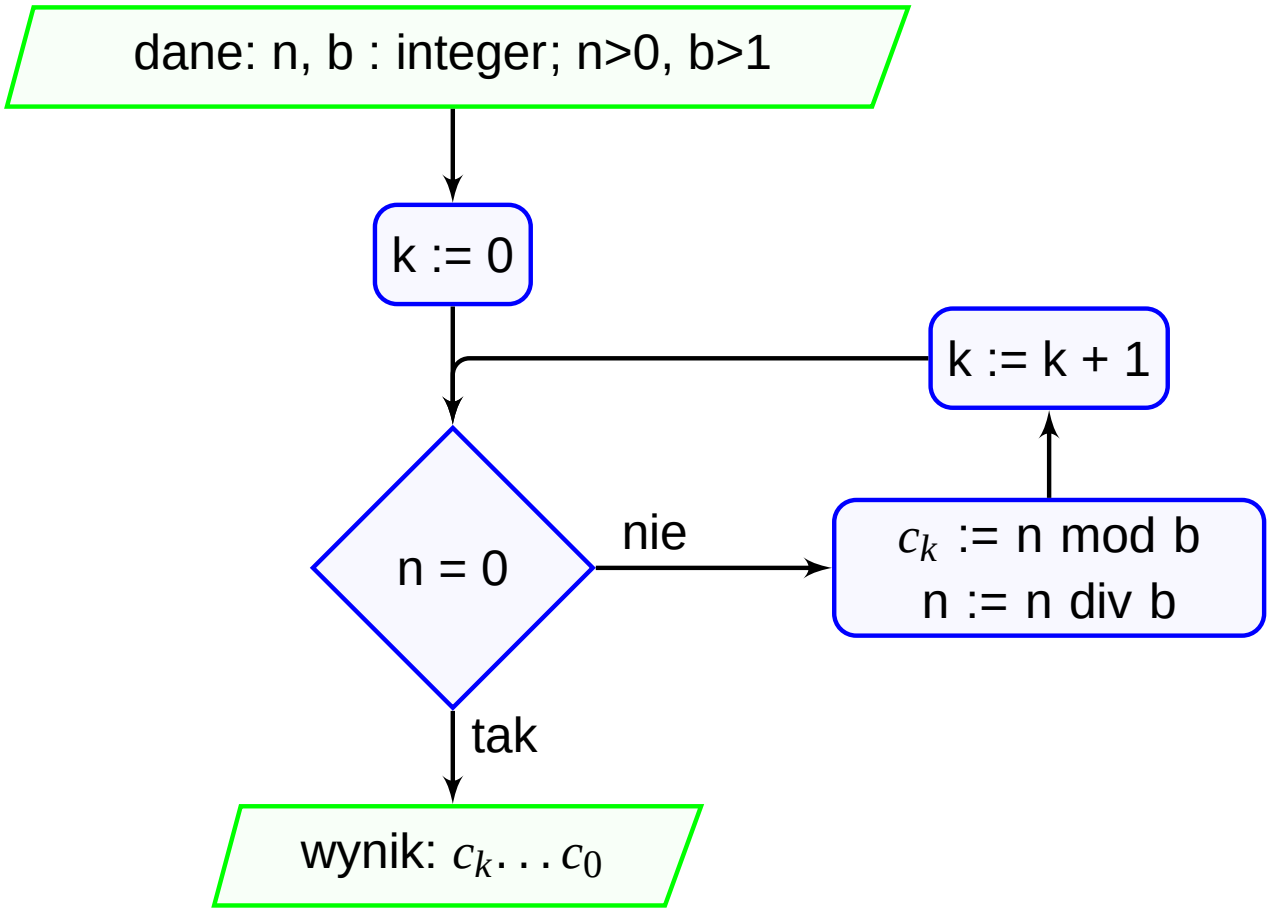
$$n = (c_k * b^{k-1} + \cdots + c_1 * b^0) * b + c_0 * b^0,$$

czyli c_0 **jest resztą z dzielenia** n **przez** b**, a** $c_k \ldots c_1$ **jest reprezentacją w bazie** b **wyniku dzielenia całkowitego** n **przez** b.

Przykład:

$$5_D = \mathbf{101}_B = 2 * 2 + \mathbf{1} = \mathbf{10}_B * 2 + 1$$

Konwersja liczby do innej bazy – schemat blokowy



Przykład:

n = 273, b = 2

n =	273	1	$= c_0$
n =	136	0	$= c_1$
n =	68	0	$= c_2$
n =	34	0	$= c_3$
n =	17	1	$= c_4$
n =	8	0	$= c_5$
n =	4	0	$= c_6$
n =	2	0	$= c_7$
n =	1	1	$= c_8$
n =	0		

Konwersja ułamka do innej bazy

Zadanie:

Szukamy maksymalnie k pierwszych cyfr reprezentacji liczby rzeczywistej $x \in (0,1)$ w systemie o bazie b, czyli cyfr $c_1, \ldots, c_k \in \{0, \ldots, b-1\}$, takich, że $x = (0.c_1 \ldots c_k \ldots)_b$, czyli

$$x = c_1 * b^{-1} + \cdots + c_k * b^{-k} + \cdots .$$

Podpowiedź:

Niech

$$\overline{x} = c_1 * b^{-1} + \cdots + c_k * b^{-k}.$$

Zauważmy, że

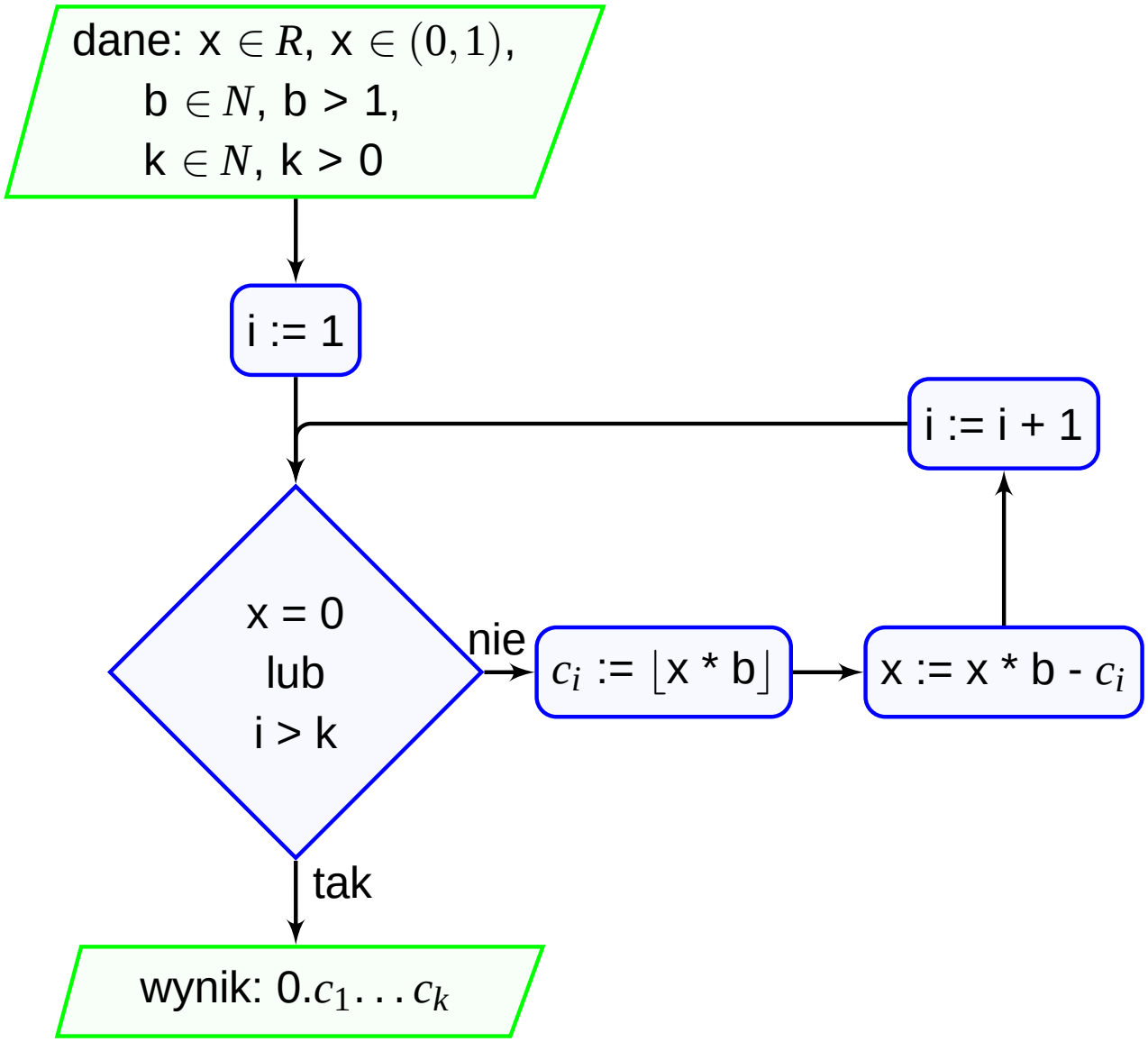
$$\overline{x} * b = c_1 + c_2 * b^{-1} + \cdots + c_k * b^{-(k-1)},$$

czyli $c_1 = \lfloor \overline{x} * b \rfloor$ **(jest częścią całkowitą liczby** $\overline{x} * b$**), a** $c_2 \ldots c_k$ **jest reprezentacją w bazie** b **(maksymalnie** $k-1$ **cyfr) liczby** $\overline{x} * b$.

Przykład:

$$0,625_D = 0,\mathbf{101}_B; \qquad 0,625_D * 2 = \mathbf{1},25_D; \qquad 0,25_D = \tfrac{1}{4} = 0,\mathbf{01}_B$$

Konwersja ułamka do innej bazy – schemat blokowy



Przykład:

x = 0.7, b = 2, k = 6

0.7 * 2	=	1	.4
0.4 * 2	=	0	.8
0.8 * 2	=	1	.6
0.6 * 2	=	1	.2
0.2 * 2	=	0	.4
0.4 * 2	=	0	.8

wynik = 0.101100

Schematy blokowe – zadania

- ★ suma n początkowych liczb naturalnych
- ★ sprawdzenie czy liczba jest pierwsza, wynik: 0 = nie, 1 = tak
- ★ rozkład liczby naturalnej na iloczyn liczb pierwszych
- ★ n początkowych wartości ciągu Fibonacciego

$$x_n = \begin{cases} 1 & \text{o ile } n < 3, \\ x_{n-1} + x_{n-2} & \text{w przeciwnym wypadku.} \end{cases}$$

$$(x_n) = (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots)$$

Złożoność obliczeniowa – intuicje

- ★ Jak liczyć efektywnie?
- ★ **Złożoność czasowa** określa rząd liczby operacji (czasu) niezbędnego do wykonania zadania
- ★ Operacje elementarne – podstawowe operacje wykonywane przez rozkazy procesora
- ★ **Złożoność pamięciowa** określa rząd ilości pamięci niezbędnej do wykonania zadania
- ★ Złożoność średnia, pesymistyczna (= w najgorszym przypadku)
- ★ Notacja $O(f(n))$, np. $O(1), O(n), O(n^2), O(n!)$

Jak efektywnie potęgować?

Zadanie: Podnieść 31 do potęgi 33.

★ Przez kolejne przemnażanie?

$$31^{33} = \underbrace{31 * \cdots * 31}_{33 \text{ razy}}$$

★ Do wykonania 32 mnożenia.

★ Można sprytniej!

Jak efektywnie potęgować?

Wykorzystajmy własność, że

$$(x^n)^m = x^{n*m}$$

Czyli można oszczędniej:

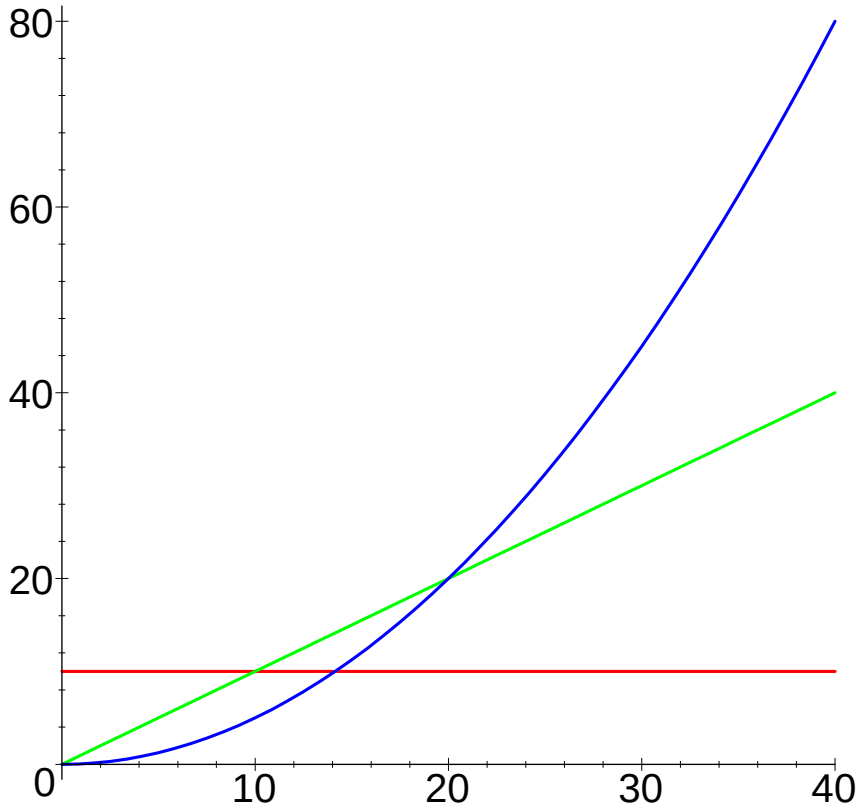
$$31^{33} = 31^{16*2+1} = 31 * \left(31^{16}\right)^2 = 31 * \left(\left(31^8\right)^2\right)^2 = \cdots = 31 * \left(\left(\left(\left(31^2\right)^2\right)^2\right)^2\right)^2$$

Każde podniesienie do kwadratu to jedno mnożenie, co daje łącznie 6 mnożeń!

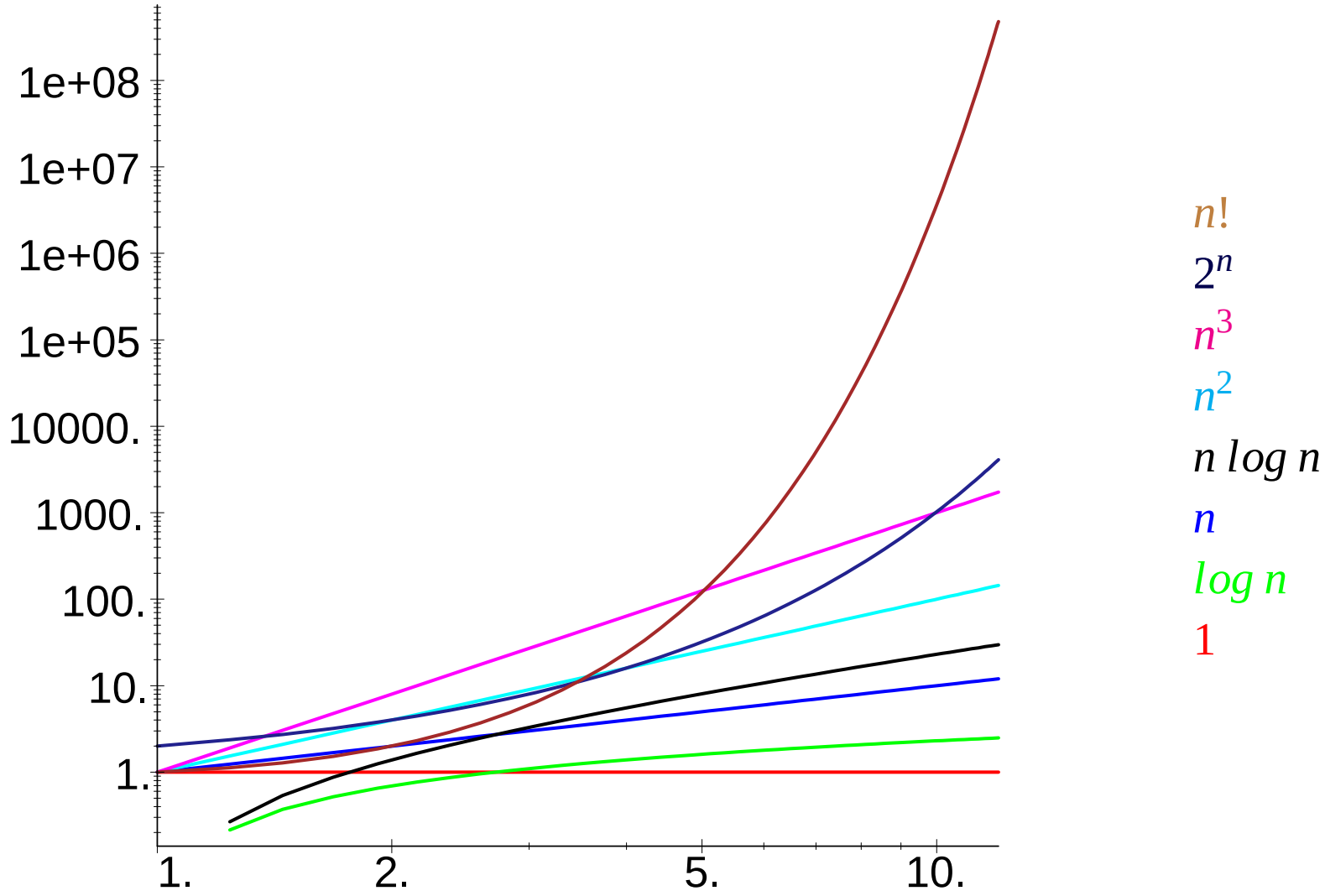
Składniki dominujące w funkcjach rzeczywistych

Przykład: $\frac{1}{20}n^2 + n + 10$

zakres	zależności
$n < 10$	$\frac{1}{20}n^2 < n < 10$
$n = 10$	$\frac{1}{20}n^2 < n = 10$
$10 < n < 20$	$10 < n \wedge \frac{1}{20}n^2 < n$
$n = 20$	$10 < n \wedge \frac{1}{20}n^2 = n = 20$
$20 < n$	$10 < n < \frac{1}{20}n^2$



Zmienność podstawowych funkcji złożoności



Porównanie czasu działania algorytmów

Zakładamy, że:

- ★ liczba działań elementarnych w algorytmie $O(f(n))$ jest dokładnie równa $f(n)$,
- ★ procesor wykonuje miliard operacji elementarnych na sekundę.

Innymi słowy: komputer obsługuje pojedynczy przypadek przez **1 ns**.

Przybliżone czasy działania poszczególnych algorytmów:

	n=1	n=10	n=50	n=100	n=1000	n=10000
$O(1)$	1 ns	1 ns	1 ns	1 ns	1 ns	1 ns
$O(log\ n)$	—	3.3 ns	5.6 ns	6.6 ns	10 ns	13.3 ns
$O(n)$	1 ns	10 ns	50 ns	100 ns	1 μs	10 μs
$O(n\ log\ n)$	—	33 ns	282 ns	664 ns	10 μs	133 μs
$O(n^2)$	1 ns	100 ns	2.5 μs	10 μs	1 ms	100 ms
$O(n^3)$	1 ns	1 μs	125 μs	1 ms	1 s	17 min
$O(2^n)$	2 ns	1 μs	13 dni	4e+13 lat	3.4e+284 lat	∞
$O(n!)$	1 ns	3.6 ms	9.6e+47 lat	3e+141 lat	∞	∞

Porównanie czasu działania algorytmów

Bardziej realnie: zakładamy, że komputer obsługuje **tysiąc przypadków w ciągu 1 sekundy**.

	n=1	n=10	n=50	n=100	n=1000	n=10000
$O(1)$	1 ms	1 ms	1 ms	1 ms	1 ms	1 ms
$O(log\ n)$	—	3.3 ms	5.64 ms	6.64 ms	10 ms	13.3 ms
$O(n)$	1 ms	10 ms	50 ms	100 ms	1 s	10 s
$O(n\ log\ n)$	—	33.3 ms	282 ms	664 ms	10 s	2.2 min
$O(n^2)$	1 ms	100 ms	2.5 s	10 s	16.7 min	1.2 d
$O(n^3)$	1 ms	1 s	2.1 min	16.7 min	11.6 d	32 lata
$O(2^n)$	2 ms	1 s	35702 lata	4e+19 lat	3.4e+290 lat	∞
$O(n!)$	1 ms	1 h	9.6e+53 lat	3e+147 lat	∞	∞

Złożoność czasowa potęgowania

★ Metoda naiwnego przemnażania:

$$x^n = \underbrace{x * \cdots * x}_{n \text{ razy}}$$

$n-1$ mnożeń – złożoność $O(n)$

★ Metoda dzielenia potęgi na pół: dla k kroków możemy wyliczyć potęgę 2^k, czyli dla x^n rząd liczby mnożeń to $\log_2 n$.
Zatem, uzyskujemy złożoność $O(\log n)$.

Wyznaczanie wartości wielomianu

Zadanie: Wyznaczyć wartość wielomianu

$$a_0 + a_1 * x + a_2 * x^2 + \cdots + a_n * x^n$$

w punkcie $x = x_0$.

★ Podejście naiwne:

$$a_0 + a_1 * x_0 + a_2 * x_0 * x_0 + \cdots + a_n * \underbrace{x_0 * \cdots * x_0}_{nrazy}$$

Liczba mnożeń = $0 + 1 + \cdots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$
Liczba dodawań = n
Złożoność $\mathbf{O(n^2)}$

★ **Schemat Hornera:**

$$((\ldots(a_n * x_0 + a_{n-1}) * x_0 \ldots) * x_0 + a_1) * x_0 + a_0$$

Liczba mnożeń = n
Liczba dodawań = n
Złożoność $\mathbf{O(n)}$

Wyznaczanie reprezentacji dziesiętnej liczby

Zadanie: Liczbę zapisaną w systemie o bazie b jako $c_n c_{n-1} \ldots c_1 c_0$ przedstawić w postaci dziesiętnej.

Rozwiązanie: Dokładnie tak samo jak przy wyznaczaniu wartości wielomianu, bo szukana liczba to

$$c_0 + c_1 * b + c_2 * b^2 + \cdots + c_n * b^n.$$

Liczymy więc wg schematu Hornera:

$$((\ldots(c_n * b + c_{n-1}) * b \ldots) * b + c_1) * b + c_0.$$

Mnożenie dwóch liczb całkowitych

★ przez sumowanie:

$$m * n = \underbrace{m + \cdots + m}_{n \text{ razy}}$$

Liczba dodawań = $n - 1$
Złożoność $O(n)$

★ mnożenie pisemne:
Warunek: znajomość tabliczki mnożenia (mnożenie jednocyfrowe). Liczba cyfr w liczbie $n \approx \log_{10} n$
Liczba mnożeń liczb jednocyfrowych = $\log m * \log n$
Liczba dodawań = $\log m * \log n - 1$
Złożoność $O(\log m * \log n)$

Rozwiązywanie równań kwadratowych

Zadanie: Rozwiązać równanie

$$ax^2 + bx + c = 0.$$

Rozwiązanie:

$$\Delta = b^2 - 4ac,$$

$$x_{1/2} = \frac{-b \pm \sqrt{\Delta}}{2a}, \text{ o ile } \Delta \geqslant 0.$$

Liczenie Δ: 4 operacje podstawowe
Liczenie $x_{1/2}$: 5 operacji (w tym 1 pierwiastkowanie)
Złożoność $O(1)$ – liczba operacji nie zależy od danych wejściowych.

Sito Eratostenesa

Zadanie: Wyznaczyć liczby pierwsze nie większe niż $n \in N$.

★ Algorytm naiwny: sprawdzamy po kolei liczby i oceniamy czy są pierwsze.
Koszt sprawdzenia czy liczba $k \in N$ jest pierwsza, gdy sprawdzamy po kolei wszystkie potencjalne dzielniki od 2 do $\sqrt{k}$: $O(k^{\frac{1}{2}}) = O(\sqrt{k})$.
Łączny koszt $\approx \sqrt{2} + \sqrt{3} + \cdots + \sqrt{n}$.

★ Sito Eratostenesa:

- Ze zbioru wartości $\{k \in N; 2 \leqslant k \leqslant n\}$ „wykreślamy" z niego liczby, które nie są pierwsze w następujący sposób:

  bierzemy po kolei liczby od najmniejszej do największej i jeśli wybrana liczba nie jest wykreślona, to wykreślamy wszystkie jej wielokrotności.
- Liczby pierwsze pozostają nie wykreślone.
- **Uwaga:** Wykreślanie wielokrotności k można zacząć od k^2, bo wszystkie wcześniejsze wielokrotności zostały już wcześniej wykreślone.

Złożoność pamięciowa: $O(n)$.
Złożoność czasowa: $O(n \log n \log\log n)$.

Języki programowania

★ różne podejścia = różne oczekiwania

★ różne poziomy abstrakcji

- ▶ niski poziom – blisko instrukcjom procesora

```
1 AND AX, 00FFH
2 MOV CL, 12
3 SHR BX, CL
4 SHL AL, CL
5 NEG CL
```

- ▶ wysoki poziom – blisko językowi naturalnemu

```
1 WydrukujOryginał;
2 if liczbaKopii > 0 then
3    for i := 1 to liczbaKopii do
4       WydrukujKopię;
```

★ meta-kod – nie konkretny język a precyzyjne wyrażenia niemal w języku naturalnym

Języki imperatywne i deklaratywne

- ★ kompletnie różne sposoby myślenia
- ★ podejścia imperatywne
  - ► „liniowe” (Basic – pierwsze wersje)
  - ► strukturalne (Pascal, C)
  - ► obiektowe (SmallTalk, C++, Object Pascal)
- ★ podejścia deklaratywne
  - ► logiczne (Prolog)
  - ► funkcyjne (ML, Lisp, Scheme)

Silnia w różnych językach

Schemat blokowy – strona 22

Basic

```
10 PRINT "Policzymy silnię. Zaczynamy."
20 INPUT "Podaj argument"; arg
30 LET silnia = 1
40 IF arg < 2 THEN GOTO 80
50 LET silnia = silnia * arg
60 LET arg = arg - 1
70 GOTO 40
80 PRINT "Wynik = "; silnia
```

Silnia w różnych językach

Pascal

```
1  function Silnia(arg : integer) : longint
2  var s : longint
3  begin
4     s := 1;
5     while arg > 1 do
6     begin
7        s = s ∗ arg;
8        arg := arg − 1
9     end;
10    Silnia := s
11 end;
```

Silnia w różnych językach

Scheme

```
1 (define silnia
2   (lambda (n)
3     (if (< n 2) 1
4       (* n (silnia (- n 1))))))
```

Prolog

```
1 silnia(0,1) :- !.
2 silnia(N,S) :- N1 is N-1, silnia(N1,S1), S is S1*N.
```

Technika *dziel i zwyciężaj*

- ★ Problemy warto rozkładać na prostsze podproblemy.
- ★ Ang. **divide and conquer**, pol. **dziel i zwyciężaj**.
- ★ Ogólny sposób rozwiązywania problemów, doskonale przydający się w programowaniu. Mnóstwo przeróżnych rozwiązań szczegółowych.
- ★ Technika niezależna od języków programowania.
- ★ Przykłady z życia:
  - ▶ wyszukiwanie słowa w słowniku,
  - ▶ budowa domu – podział na mniejsze podzadania, różni wykonawcy specjalizujący się w różnych pracach: murarz, cieśla, stolarz, kowal, ...
  - ▶ rodzina – podział obowiązków,
  - ▶ budowa komputera – procesory, karty graficzne, monitory, sieci, audio, ...
- ★ Przykłady z programowania:
  - ▶ mnożenie przez dodawanie,
  - ▶ mnożenie pisemne przez dodawanie i tabliczkę mnożenia do 10,
  - ▶ liczenie potęgi przez redukowanie do mniejszych potęg, ogólniej rekurencja,
  - ▶ metody sortowania (o nich później),

Rekurencja

★ **Rekurencja** (z łac. *recurrere*, przybiec z powrotem) to odwołanie się definicji do samej siebie.

★ Definicje rekurencyjne:

- Zbiór liczb naturalnych N:
  1. $0 \in N$,
  2. $n \in N \Rightarrow s(n) \in N$.

  W myśl tej definicji 1 = s(0), 2 = s(s(0)) itd. s to operator następnika.
- Ciąg Fibonacciego

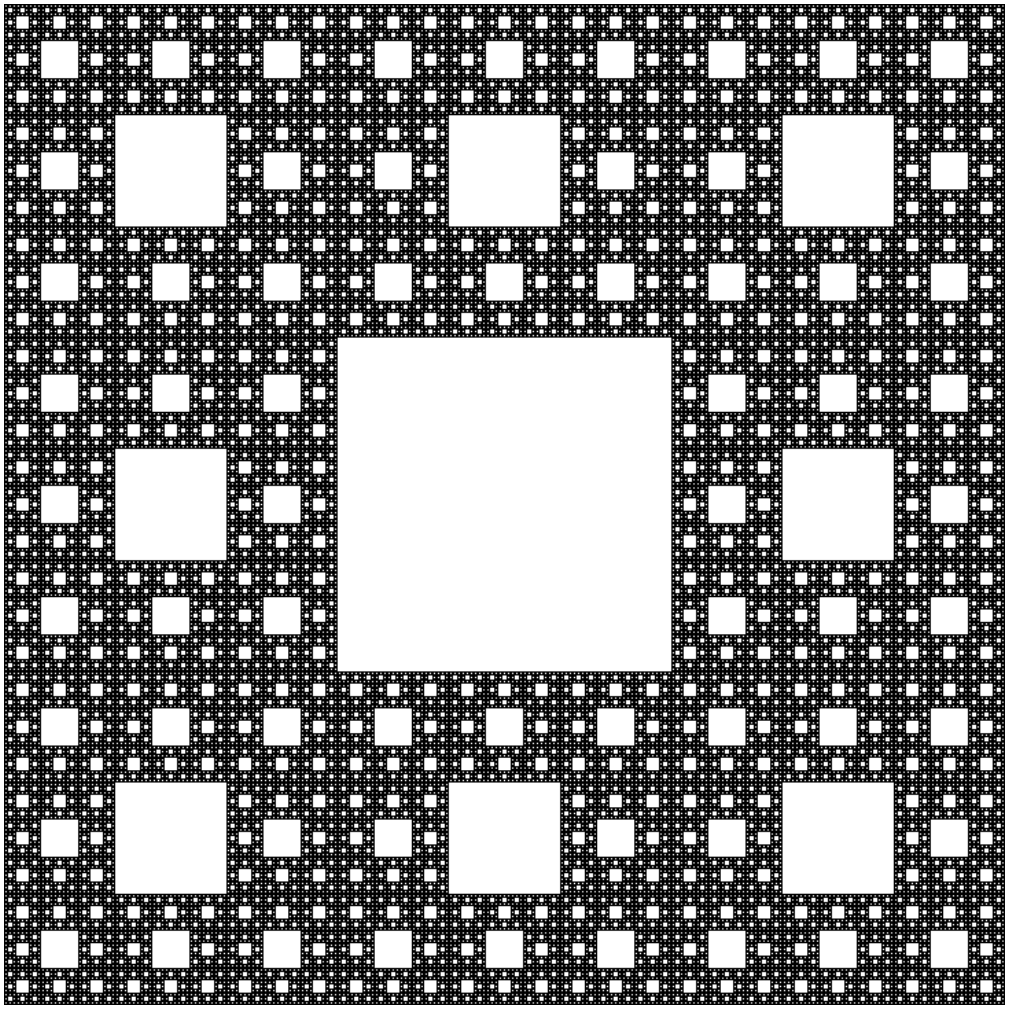
$$x_n = \begin{cases} 1 & \text{o ile } n < 3, \\ x_{n-1} + x_{n-2} & \text{w przeciwnym wypadku.} \end{cases}$$

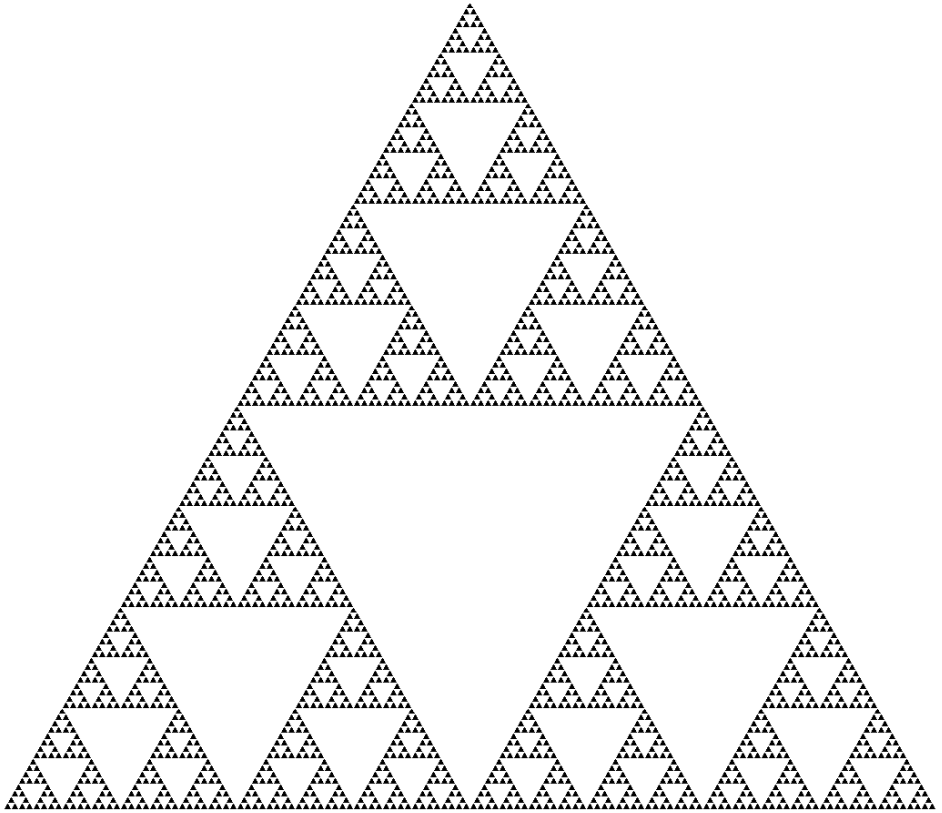
- Silnia $n \in N$:

$$n! = \begin{cases} 1 & \text{o ile } n < 2, \\ n * (n-1)! & \text{w przeciwnym wypadku.} \end{cases}$$

Obrazy rekurencyjne – fraktale

Dwa proste przykłady: dywan i trójkąt Sierpińskiego





Rekurencja w programowaniu

Silnia rekurencyjnie

```
1 function Silnia(n : integer) : longint
2 begin
3   if n < 2 then
4     Silnia := 1;
5   else
6     Silnia := n * Silnia(n-1);
7 end
```

Stos wywołań funkcji

- ★ Przy wywołaniu funkcji wewnątrz innej funkcji należy zapamiętać stan tej przerywanej, by przejść do wykonywania tej wywołanej.
- ★ Uwaga na nieskończone wywołania – powodują przepełnienie stosu.
- ★ Obsługa stosu kosztuje - zarówno czasowo jak i pamięciowo.
- ★ Optymalnie napisane funkcje nierekurencyjne są zawsze szybsze niż rekurencyjne.

Rekurencja w programowaniu – c.d.

★ Kiedy stosować rekurencję? Gdy spełnione są dwa warunki:

- ► natura problemu jest rekurencyjna (wtedy czytelność kodu jest bardzo dobra),
- ► nie ponosimy istotnych strat obliczeniowych, np:
  - złożoność metody się nie pogorszy,
  - funkcja nie jest wołana setki/tysiące razy na sekundę.

★ Złożoność w rekurencji - zwykle liczba wywołań jest najistotniejsza.

★ Złożoność silni: $O(n)$. Przykładowy stos wywołań:

```
1        Silnia(4)
2        Silnia(3)
3        Silnia(2)
4        Silnia(1)
```

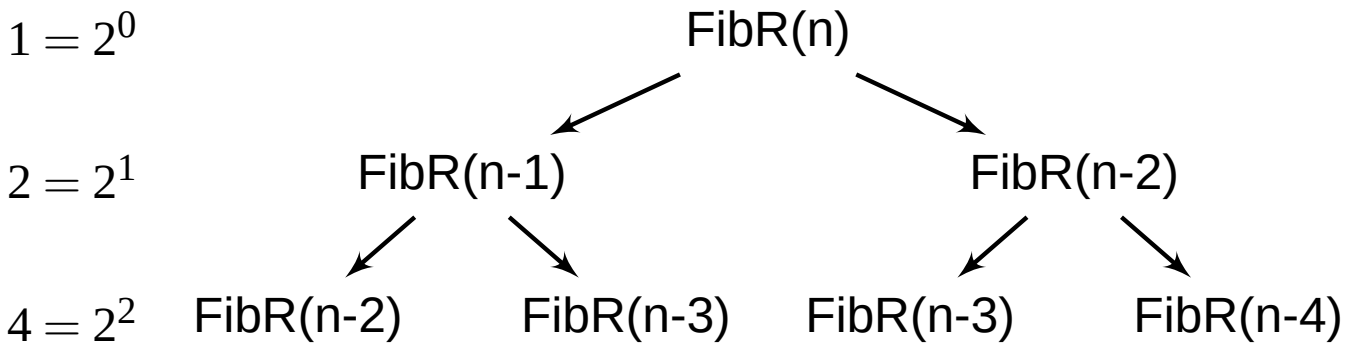
★ Uwaga na lawinowe narastanie wywołań!

Rekurencja a ciąg Fibonacciego

Przykład **absolutnie niewłaściwej** implementacji:

```
1 function FibR(n : integer) : longint;
2 begin
3   if n < 3 then
4     FibR := 1
5   else
6     FibR := FibR(n−1) + FibR(n−2)
7 end;
```

Hierarchia wywołań:



Można udowodnić, że liczba wywołań $> 2^{\frac{n}{2}}$. **Koszmarnie dużo!**
Komputer liczący **miliard** wywołań na sekundę liczyłby **FibR(120) ponad 36 lat**.

Ciąg Fibonacciego iteracyjnie

Uwaga: W kodzie poniżej, **array** oznacza tzw. tablicę n elementów typu **longint** tzn. n ponumerowanych wartości tego samego typu.

```
1 function Fib(n : integer) : longint;
2 var f : array[1..n] of longint; i : integer; { prawie Pascal! }
3 begin
4   f[1] := 1; f[2] := 1
5   for i := 3 to n do
6     f[i] := f[i−1] + f[i−2];
7   Fib := f[n];
8 end;
```

Złożoność czasowa $O(n)$, pamięciowa $O(n)$.

To już co innego, ale jeszcze ta pamięć!

Ciąg Fibonacciego iteracyjnie i oszczędniej z pamięcią

```
1  function Fib(n : integer) : longint;
2  var pop, ppop, nast : longint; i : integer;
3  begin
4    if n < 3 then
5      Fib := 1
6    else
7    begin
8      pop := 1; ppop := 1;
9      for i := 3 to n do
10     begin
11       nast := pop + ppop;
12       ppop := pop;
13       pop := nast;
14     end;
15     Fib := nast;
16   end
17 end;
```

Złożoność czasowa $O(n)$, pamięciowa $O(1)$. **To lubimy!**

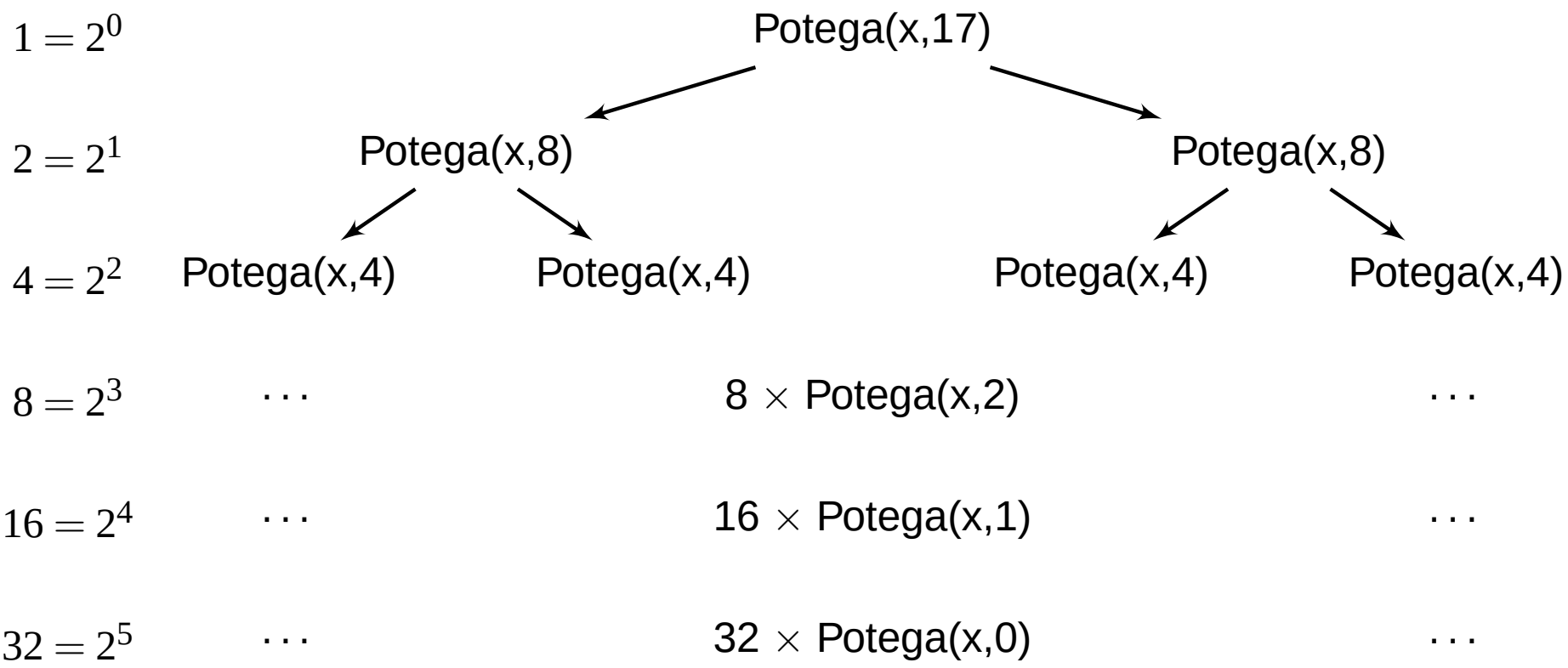
Jak zepsuć potęgowanie?

Na przykład tak:

```
1  function Potega(x : real; n : integer) : real;
2  begin
3    if n = 0 then
4      Potega := 1
5    else
6      if n mod 2 = 0 then
7        Potega := Potega(x, n div 2) * Potega(x, n div 2)
8      else
9        Potega := x * Potega(x, n div 2) * Potega(x, n div 2)
10 end;
```

Spartaczone potęgowanie

Przykładowa hierarchia wywołań:



Razem wywołań 1 + 2 + 4 + 8 + 16 + 32 = 2 * 32 - 1 = 63
Złożoność $O(2^k)$, gdzie k to głębokość wywołań (rzędu $\log_2 n$, gdzie n to wykładnik).
Ostatecznie $2^{\log_2 n} = n$, czyli złożoność $O(n)$. Jak na potęgowanie to i tak **fatalnie!**

Potęgowanie poprawnie

```
1  function Potega(x : real; n : integer) : real;
2  var pom : real;
3  begin
4    if n = 0 then
5      Potega := 1
6    else
7    begin
8      pom := Potega(x, n div 2);
9      if n mod 2 = 0 then
10       Potega := pom * pom
11     else
12       Potega := x * pom * pom
13   end
14 end;
```

Maksymalnie jedno wywołanie wewnątrz, więc lawiny nie będzie! Wywołania dla $n = 17 \to n = 8 \to n = 4 \to n = 2 \to n = 1 \to n = 0$. Złożoność $O(\log n)$.

Indukcja matematyczna

Zasada domina: Z faktu, że każda kostka przewrócona przewraca następną oraz że pierwsza została przewrócona wynika gwarancja przewrócenia wszystkich kostek.

W matematyce:
Aby udowodnić, że prawdziwe jest stwierdzenie Z(n) dla każdej liczby naturalnej n (n = 1,2, . . .), potrzeba i wystarcza by:

1. prawdziwe było zdanie Z(1),
2. z prawdziwości zdania Z(n) wynikała prawdziwość zdania Z(n+1).

Dowód przez indukcję matematyczną

Przykład:

Udowodnić, że dla każdego $n = 1, 2, \ldots$ liczba $10^n - 4$ jest podzielna przez 6.

Dowód:

Z(n) $\equiv$ „liczba $10^n - 4$ jest podzielna przez 6”

1. Dla n = 1 mamy Z(1) $\equiv$ „$10^1 - 4 = 6$ jest podzielna przez 6”. Prawda.
2. Z(n) $\Rightarrow$ Z(n+1).
   Zakładamy, że „liczba $10^n - 4$ jest podzielna przez 6” i dowodzimy, że „liczba $10^{n+1} - 4$ jest podzielna przez 6”.
   Zauważmy, że

$$10^{n+1} - 4 = 10 * 10^n - 10 * 4 + 9 * 4 = 10 * (10^n - 4) + 36.$$

   Oba składniki dzielą się przez 6, więc ich suma też. □

[koniec dowodu]

Szukanie w tablicach

Dane: Tablica obiektów.
Cel: Czy tablica zawiera pewien obiekt? Pobranie tego obiektu.

Przykłady:

- ★ Wyszukiwanie słów w słowniku.
- ★ Wyszukiwanie liczb w tablicy liczb.
- ★ Szukanie osoby urodzonej 1.04.2000 w kolekcji danych o osobach.
- ★ Szukanie działki o powierzchni 10 arów w kolekcji działek na sprzedaż.

Uwaga: Całkiem inne podejście gdy dane są odpowiednio **uporządkowane** niż kiedy porządek jest losowy.

Szukanie liniowe (sekwencyjne)

Gdy porządek jest **nieokreślony**, trzeba przeglądać **całą tablicę** element po elemencie.

```
1  function SzukajLiniowo(x : array[1..n] of Typ; klucz : TypKlucza) : Typ;
2  var i : integer;
3  begin
4    for i := 1 to n do
5      if KluczObiektu(x) = klucz then
6      begin
7        SzukajLiniowo := x;
8        koniec
9      end;
10   SzukajLiniowo := nie_znaleziono
11 end;
```

Jeśli szukanego elementu nie ma, przejrzymy całą tablicę. Jeśli obiekt w tablicy istnieje, możemy skończyć wcześniej, ale nie zmienia to średniej złożoności $O(n)$, gdzie n jest liczbą elementów w tablicy.

Szukanie binarne

Jeśli dane są uporządkowane wg klucza wyszukiwania, to można szukać znacznie krócej niż liniowo.

Dane: tablica x, klucz, zakres szukania [początek,koniec].
Procedura szukania:

- ★ Jeśli klucz $<$ KluczObiektu(x[początek]) lub klucz $>$ KluczObiektu(x[koniec]) to nie znajdziemy.
- ★ środek := (początek + koniec) div 2;
- ★ Jeśli klucz $<$ KluczObiektu(x[środek]) to szukamy w podprzedziale [początek, środek], w przeciwnym przypadku w przedziale [środek, koniec].

Uwaga: Sprawdzanie warunku zawierania się klucza w zakresie, być może warto wykonać raz przed przystapieniem do szukania, ale nie w każdym (rekurencyjnym) wywołaniu.

Złożoność $O(logn)$, gdzie n to długość tablicy.

Na przykład: szukanie wśród 1000 obiektów wymaga ok. 10 porównań, wśród miliarda obiektów – ok. 30 porównań.

Szukanie heurystyczne

Metody **heurystyczne** to takie, które próbują odgadnąć rozwiązanie (nie dają gwarancji dobrego rozwiązania, ale działają w krótkim czasie).

Jeśli dane są uporządkowane wg klucza wyszukiwania, i znamy mniej-więcej rozkład wartości klucza to można szukać jeszcze krócej niż binarnie!

Podpowiedź: Szukając w słowniku słowa „binarny” od razu próbujemy szukać w miejscu, gdzie się tego słowa spodziewamy.

Dane: tablica x, klucz, zakres szukania [początek,koniec].
Procedura szukania:

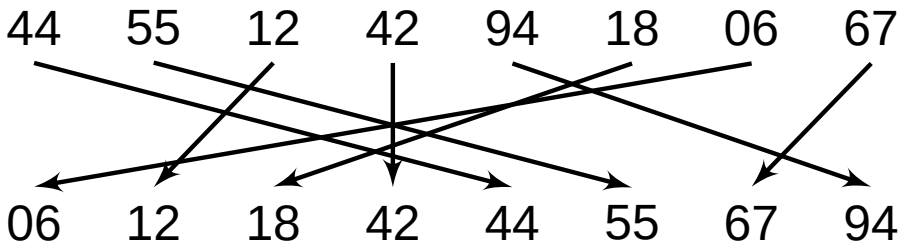
- ★ próg := ZgadnijGdzie(klucz, początek, koniec);
- ★ Jeśli klucz $<$ KluczObiektu(x[próg]) to szukamy w podprzedziale [początek, próg], w przeciwnym przypadku w przedziale [próg, koniec].

Złożoność: nawet $O(1)$, przy odpowiednio dobrej ocenie rozkładu statystycznego kluczy.

Uwaga: Szukanie binarne jest przypadkiem szczególnym szukania heurystycznego (z dość kiepską heurystyką).

Problem sortowania

- ★ Sortowanie = układanie w odpowiednim porządku.
- ★ Co i jak porządkujemy?
  - ► liczby: rosnąco, malejąco, . . .
  - ► napisy: alfabetycznie, wg kodów ASCII, wg długości, . . .
  - ► dane o osobach: wg daty urodzenia, wg wzrostu, wg nazwiska i imienia, wg wyników w nauce, . . .
  - ► . . .
- ★ Przykład:



- ★ Można abstrahować od typu obiektów sortowanych oraz relacji porządku.

Metody sortowania

- ★ Prosty wybór
- ★ Bąbelkowe
- ★ Szybkie
- ★ Łączenie
- ★ Zliczanie
- ★ i wiele innych

Uwaga na precyzję określeń!

element minimalny $\stackrel{def}{=}$ *nie większy niż którykolwiek inny*

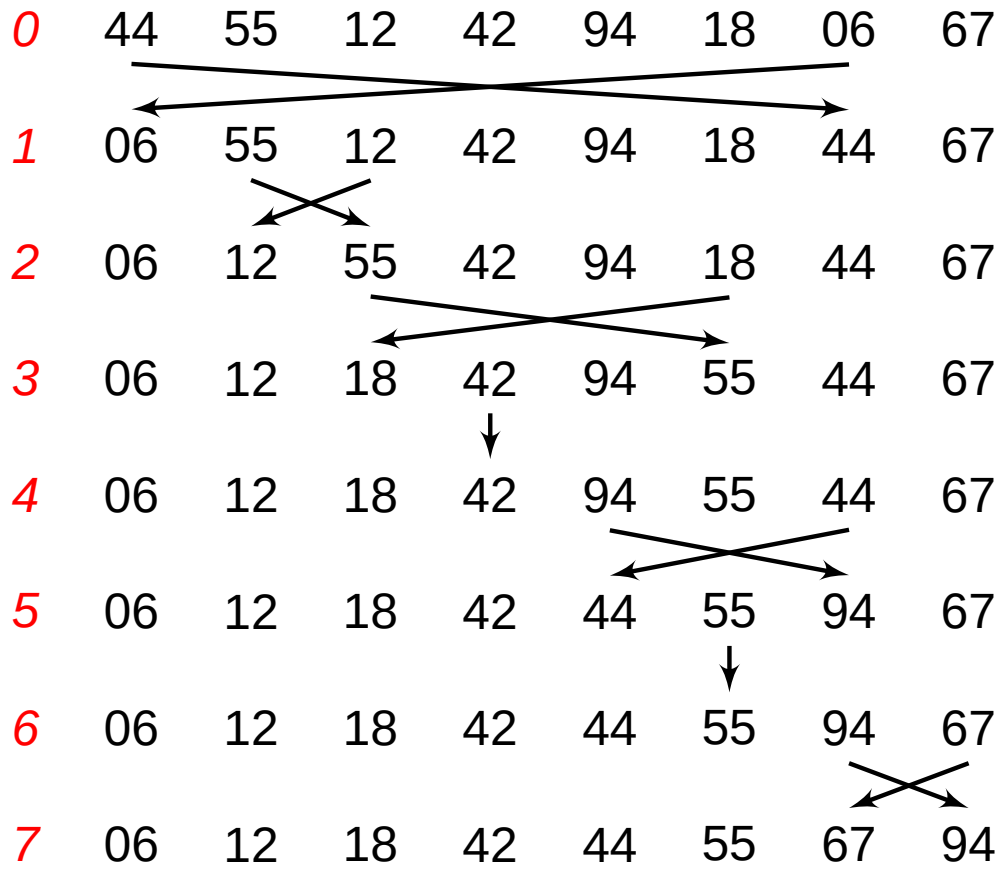
element najmniejszy $\stackrel{def}{=}$ *mniejszy od każdego innego*

Algorytmy *in situ* *(łac. „w miejscu”)* to takie, które nie wymagają dodatkowej pamięci zależnej od rozmiaru danych wejściowych (złożoność pamięciowa $O(1)$). Tutaj: trzy pierwsze są in situ, kolejne dwa nie.

Sortowanie przez prosty wybór

Idea: Znajdujemy element minimalny i zamieniamy go z pierwszym, a potem w ten sam sposób zajmujemy się pozostałymi (pomijając pierwszy).

Przykład:

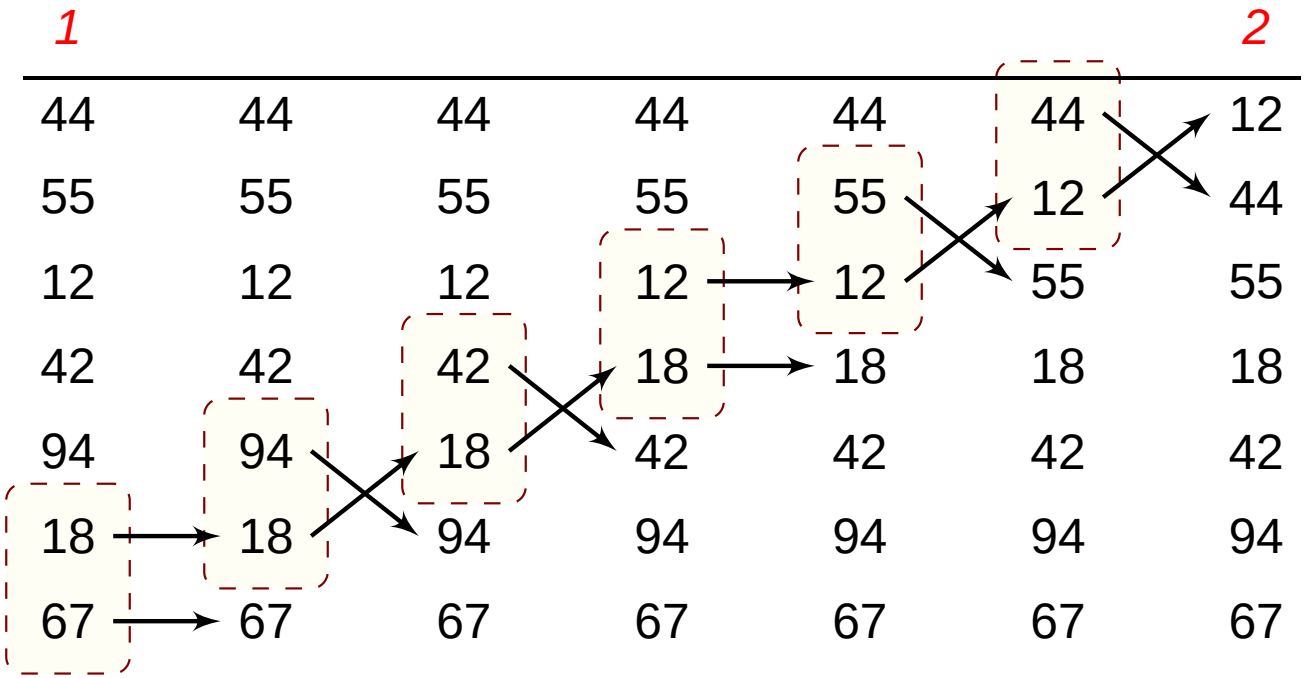


Liczba porównań: $n-1+n-2+\ldots+1$. Złożoność czasowa $O(n^2)$.

Sortowanie bąbelkowe

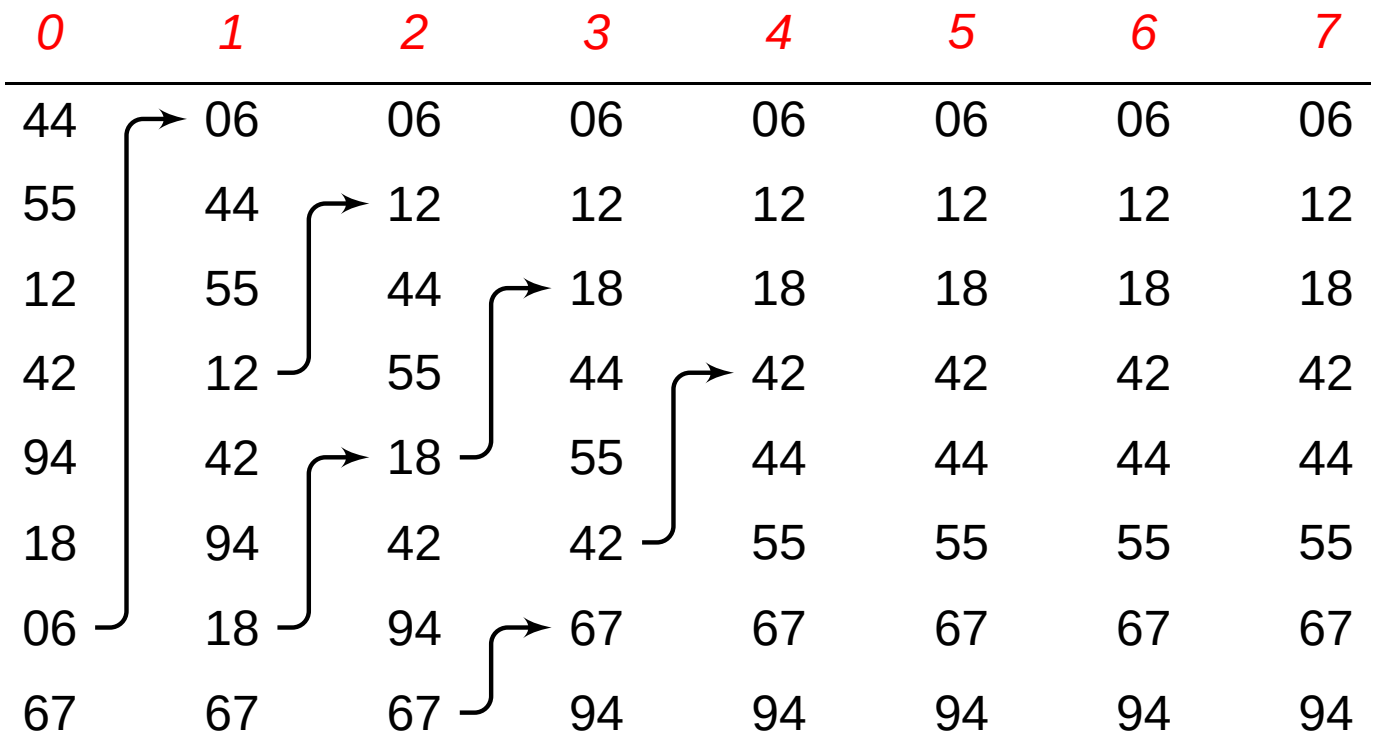
Idea: Porównujemy i zamieniamy tylko sąsiednie elementy. Po przebiegu przez całą tablicę, jeden element jest na pewno na swoim miejscu.

Przykład: Pojedynczy przebieg – porównujemy parami od dołu:



Sortowanie bąbelkowe

Przykład:



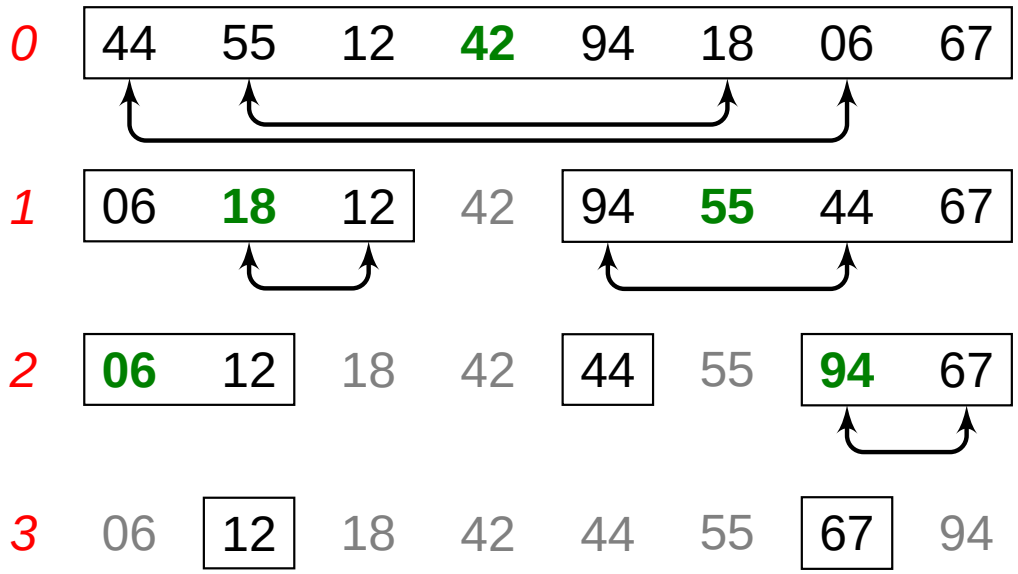
Złożoność czasowa $O(n^2)$.
Jeśli w danym przebiegu nie dokonano żadnej zamiany, to można kończyć. Czyli w najlepszym przypadku (wartości posortowane na wejściu) mamy złożoność $O(n)$.

Sortowanie szybkie (ang. quicksort)

- ★ Typowe podejście typu **dziel i zwyciężaj**.
- ★ Dążymy do podziału na dwie części takie, że każdy element pierwszej części jest nie większy niż każdy element drugiej części.
- ★ Wybieramy wartość graniczną (między dwiema częściami) i szukamy od obu stron wartości, które mogłyby być po drugiej stronie
  - ► kiedy znajdziemy z obu stron kandydatów do przeniesienia na drugą stronę zamieniamy ich miejscami i kontynuujemy poszukiwania.
  - ► miejsce, w którym poszukiwania z obu ston „spotkają się”, to miejsce podziału
- ★ Dokonujemy podziału na dwie części i każdą sortujemy tym samym algorytmem (rekurencja!).
- ★ Jeśli do posortowania jest tylko jeden element (lub zero), to nic nie trzeba robić.

Sortowanie szybkie

Przykład:

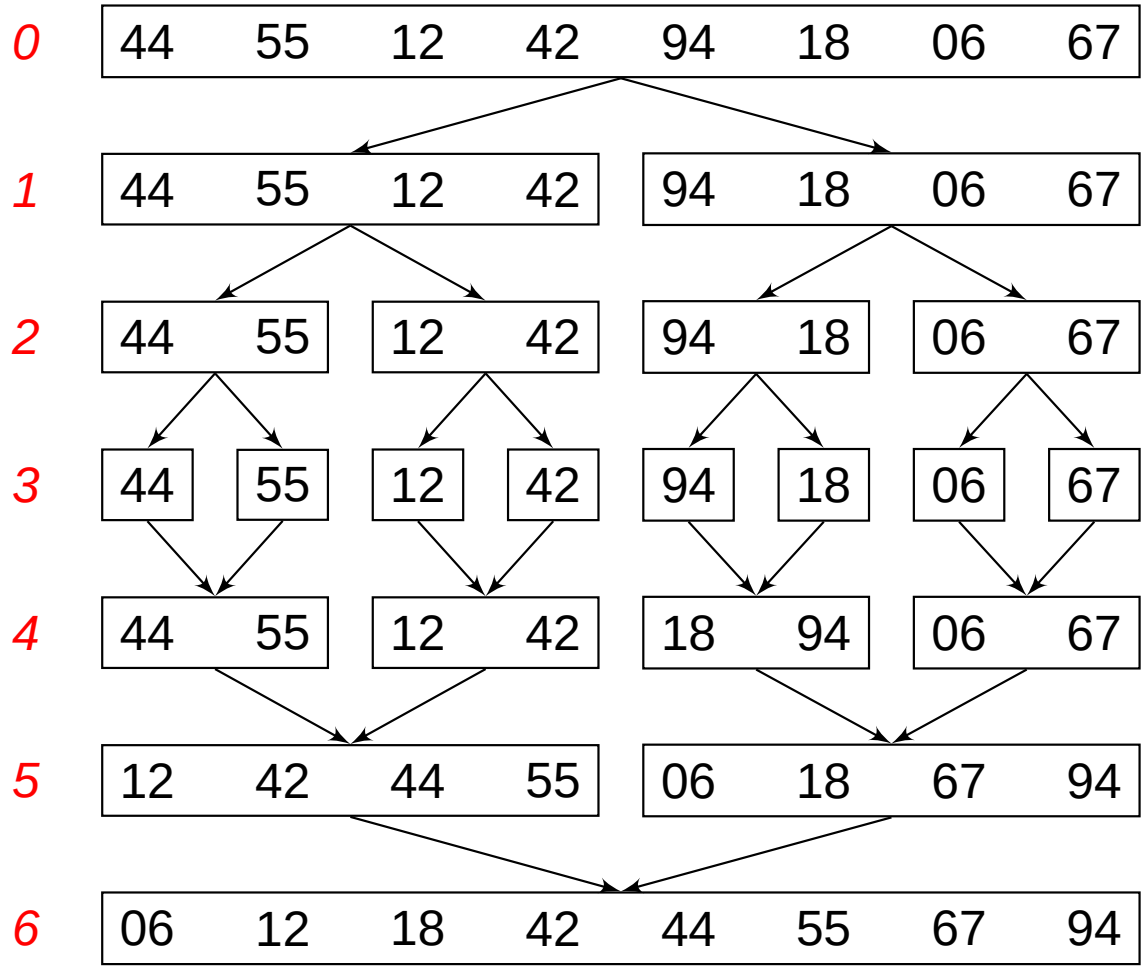


- ★ Średnia złożoność czasowa $O(n \log n)$.
- ★ Złożoność w najgorszym przypadku $O(n^2)$ (gdy z jednej strony podziału zawsze zostaje tylko jeden element).
- ★ Wybór wartości podziału – różne warianty, różne efekty
  - ▶ losowo – kiepsko, bo można w ogóle nie podzielić tablicy,
  - ▶ środek zakresu kluczy z tablicy,
  - ▶ pierwszy/ostatni/**środkowy** element z tablicy.

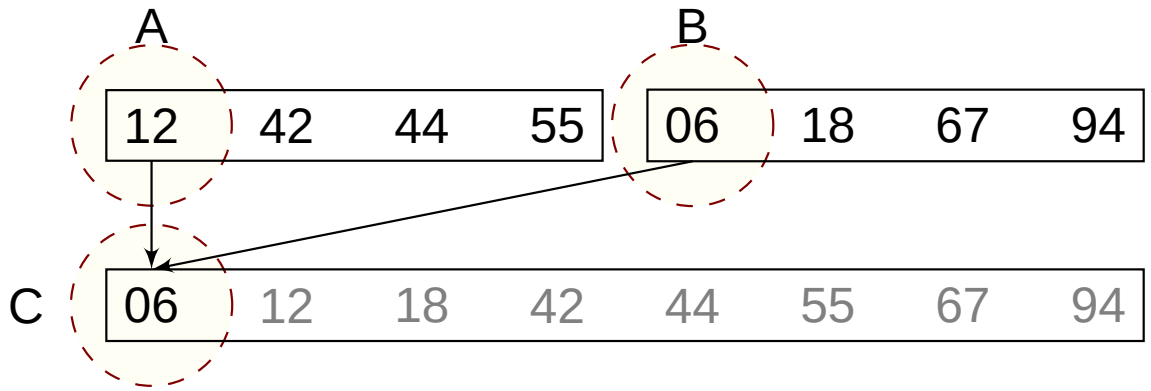
Sortowanie przez łączenie (scalanie, *ang. mergesort*)

Idea: Dzielimy tablicę na dwie części, sortujemy każdą z osobna a potem łączymy dwie uporządkowane tablice w jedną.

Przykład:



Algorytm łączenia dwóch posortowanych tablic

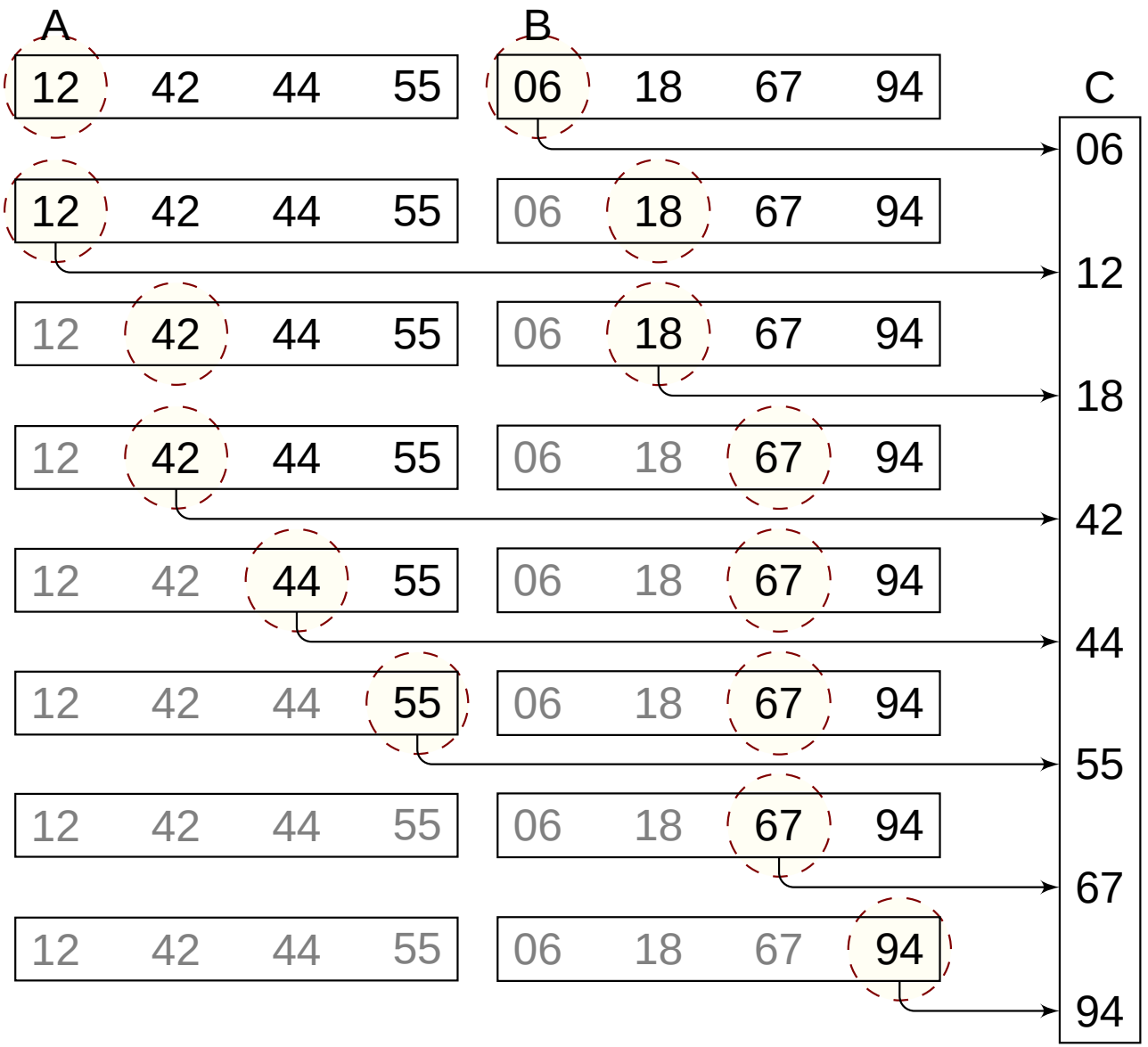


Idea: Zaczynamy od początku posortowanych tablic i przesuwamy się do końca przepisując do docelowej tablicy dane o niższej (lub równej) wartości klucza.

Uwagi:

- ★ Żeby wyznaczyć wartość minimalną wystarczy sprawdzić minimalne (pierwsze) wartości łączonych tablic.
- ★ Po przepisaniu wartości z jednej z tablic następna wartość jako minimalna z pozostałych jest kandydatem do kolejnego miejsca.

Algorytm łączenia dwóch posortowanych tablic



Sortowanie przez łączenie

Algorytm łączenia: Dane: posortowane tablice A i B o długościach d_A i d_B.
Wynik: Tablica C długości $d_A + d_B$.

- ★ $i_A \leftarrow 1$; $i_B \leftarrow 1$; $i_C \leftarrow 1$;
- ★ Tak długo jak $i_A \leqslant d_A$ lub $i_B \leqslant d_B$:
  - ► Jeśli $i_A \leqslant d_A$ oraz ($i_B > d_B$ lub $A[i_A] \leqslant B[i_B]$), to
    $C[i_C] \leftarrow A[i_A]$
    $i_A \leftarrow i_A + 1$,
    w przeciwnym przypadku:
    $C[i_C] \leftarrow B[i_B]$
    $i_B \leftarrow i_B + 1$.
  - ► $i_C \leftarrow i_C + 1$

Interpretacja symboli:

- ★ i_A, i_B, i_C – indeksy elementów poszczególnych tablic, którym się aktualnie zajmujemy, $A[i_A], B[i_B], C[i_C]$ – owe elementy,
- ★ $i_A \leqslant d_A$ – „nie skończylismy jeszcze z tablicą A”,
- ★ $i_B > d_B$ – „skończylismy już z tablicą B”.

Sortowanie przez łączenie

Algorytm główny: *SortujPrzezŁączenie(A)*
Dane: tablica A o długości d_A.
Wynik: Tablica A posortowana.

- ★ $B \leftarrow$ kopia A;
- ★ wynik $\leftarrow$ *SortujPrzezŁączenie*$(A, 1, d_A, B)$;

Funkcja pomocnicza: *SortujPrzezŁączenie(A, pocz, kon, B)*
Dane: tablice A i B identyczne w zakresie do posortowania $(pocz, kon)$.
Wynik: Tablica A posortowana.

- ★ Jeśli $pocz = kon$ to koniec;
- ★ $kon1 \leftarrow \left\lfloor \frac{pocz+kon}{2} \right\rfloor$;
- ★ *SortujPrzezŁączenie*$(B,\ pocz,\ kon1,\ A)$;
- ★ *SortujPrzezŁączenie*$(B,\ kon1+1,\ kon,\ A)$;
- ★ *Połącz*$(B,\ pocz,\ kon1,\ kon,\ A)$;

Złożoność sortowania przez łączenie

Procedura łączenia:

- ★ Złożoność czasowa: $O(n)$.
- ★ Złożoność pamięciowa: $O(n)$. Nie w miejscu.

Całość sortowania przez łączenie:

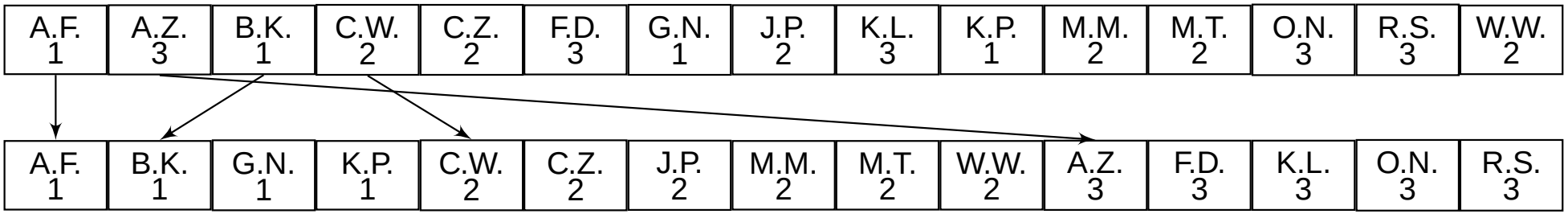
- ★ Złożoność czasowa: $O(n \log n)$. Również w najgorszym przypadku.
- ★ Złożoność pamięciowa: $O(n)$. Nie w miejscu.

Sortowanie przez zliczanie

- ★ Bardzo atrakcyjna złożoność $O(n)$, ale metoda stosowalna tylko w specyficznych okolicznościach.
- ★ Tylko dla kluczy naturalnych (np. w przedziale $1,\ldots,n$).
- ★ Nie w miejscu, choć można w miejscu, ale **niestabilnie**.

Sortowanie **stabilne** to takie, które zachowuje porządek pomiędzy obiektami o tym samym kluczu.

Przykład: Mamy tablicę danych o studentach posortowanych wg. nazwisk i chcemy posortować ją według przynależności do grup laboratoryjnych tak, by w każdej grupie pozostał alfabetyczny porządek nazwisk.



Sortowanie przez zliczanie

Idea algorytmu:

★ Liczymy ile kolejnych kluczy występuje w tablicy – w ten sposób wyznaczamy, które części tablicy będą zajmowane przez dane o kolejnych wartościach kluczy.

★ Przepisujemy wartości po kolei, od końca, na odpowiednie miejsca w kopii tablicy.

Przykład krok po kroku:

★ Dane wejściowe:

A.F. 1	A.Z. 3	B.K. 1	C.W. 2	C.Z. 2	F.D. 3	G.N. 1	J.P. 2	K.L. 3	K.P. 1	M.M. 2	M.T. 2	O.N. 3	R.S. 3	W.W. 2

★ Zliczamy wystąpienia kluczy:

Numer grupy:	1	2	3
Liczba wystąpień:	4	6	5

★ Wyznaczamy pozycje końcowe grup:

Numer grupy:	1	2	3
Pozycja:	4	10	15

★ Przepisujemy na swoje miejsca:

A.F. 1	B.K. 1	G.N. 1	K.P. 1	C.W. 2	C.Z. 2	J.P. 2	M.M. 2	M.T. 2	W.W. 2	A.Z. 3	F.D. 3	K.L. 3	O.N. 3	R.S. 3

Sortowanie przez zliczanie – formalnie

```
1  function SortujPrzezZliczanie(A : array[1..n] of Typ, k : integer)
2           : array[1..n] of Typ
3  var B : array[1..n] of Typ, { posortowana tablica }
4      C : array[1..k] of integer, { tablica pomocnicza }
5      i, j : integer;
6  begin
7      { zerujemy tablicę pomocniczą (nie trzeba w pewnych językach) }
8      for i := 1 to k do
9          C[i] := 0;
10     { zliczamy wystąpienia }
11     for i := 1 to n do
12     begin
13         j := Klucz(A[i]);
14         C[j] := C[j] + 1;
15     end;
```

```
16      { liczymy indeksy końcowe grup }
18      for i := 2 to k do
19          C[i] := C[i] + C[i−1];
20      { przepisujemy od końca z tablicy A do tablicy B }
21      for i := n downto 1 do
22      begin
23          j := Klucz(A[i]);
24          B[C[j]] := A[i];
25          C[j] := C[j] − 1;
26      end;
27      SortujPrzezZliczanie := B;
28  end
```

Struktury danych

★ Algorytmy + struktury danych = programy.

★ Struktury **statyczne** mają stały rozmiar (zajmują stały obszar pamięci) przez cały czas pracy.

- ► **tablice** – kolekcje elementów tego samego typu z dostępem swobodnym
  Deklaracja w Pascalu: C : **array**[1..n] **of integer**;
  Dostęp do elementów: C[i]
- ► **rekordy (struktury)** – grupują nazwane elementy różnych typów

Przykład w Pascalu:

```
1 type Osoba = record
2       Imie : string;
3       Nazwisko : string;
4       DataUr : Data;
5       RokUkonczenia : integer;
6     end;
```

Korzystanie:

```
1 var ja : Osoba;
2 begin
3    ja.Imie := 'Krzysztof';
4    ja.DataUr.Dzien := 7;
5    ...
6 end;
```

★ Struktury **dynamiczne** – zmienna „objętość” danych w trakcie pracy.

Pliki

Pliki lub **strumienie** (ang. **file**, **stream**) to struktury o dostępie sekwencyjnym a nie swobodnym (jak np. tablice)

- ★ różnica jak pomiędzy zapisem na taśmach a zapisem na dyskach,
- ★ w danej chwili dostęp tylko do jednego miejsca,
- ★ zmiana bieżącego elementu z pierwszego na ostatni wymaga przejścia przez wszystkie elementy pliku.

Specyfika struktury danych narzuca odpowiednie rozwiązania algorytmiczne i odwrotnie: dla sprawnego wykonania stosownych algorytmów należy dysponować odpowiednimi strukturami danych.

Dynamiczne kolekcje

Przeróżne kolekcje na różne okoliczności - warto znać ich specyfikę, by trafnie dobierać struktury danych do rozwiązywanych problemów.

Najważniejsze funkcje kolekcji: **wstawianie** (dodawanie), **usuwanie**, **wyszukiwanie**.

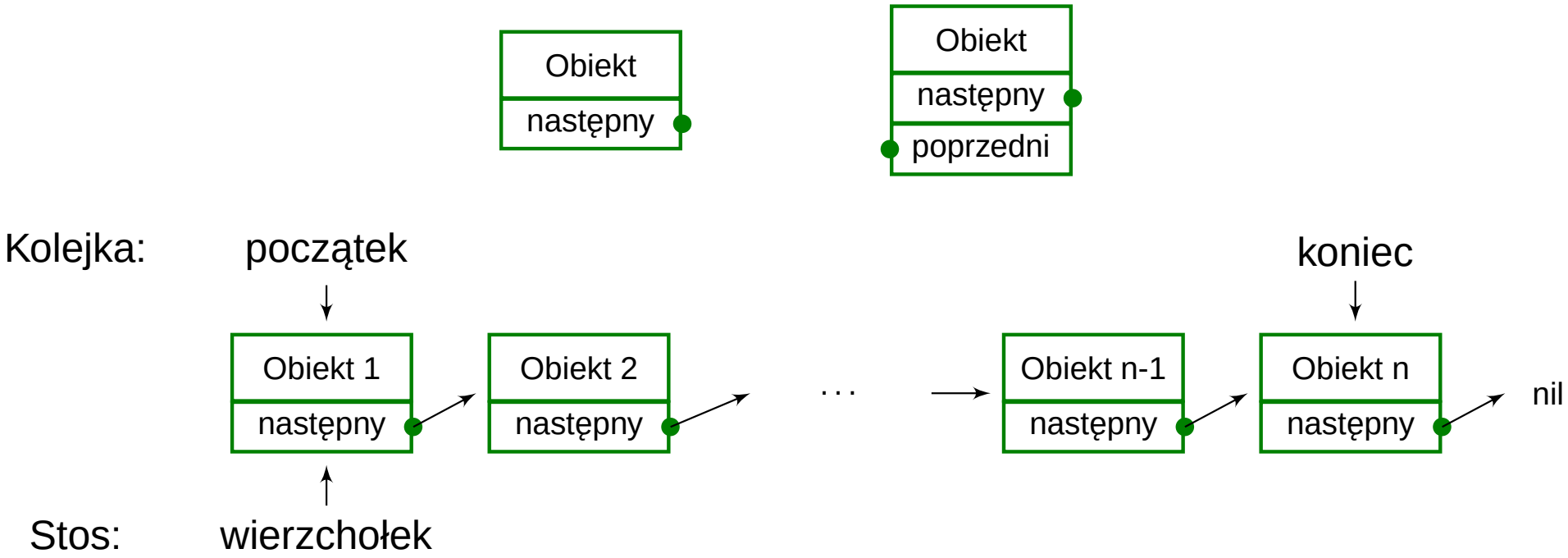
Kolekcje:

- ★ kolejki (FIFO - first-in-first-out)
- ★ stosy (LIFO - last-in-first-out)
- ★ listy
  - ► jednokierunkowe i dwukierunkowe
  - ► cykliczne
- ★ drzewa
  - ► binarne i niebinarne (wielokierunkowe)
  - ► zbalansowane (wyważone, AVT), dokładnie wyważone
  - ► czerwono-czarne
- ★ grafy

Specyfika kolekcji, złożoności operacji

Dla sprawnej obsługi kolekcji, zwykle z każdym elementem pamiętamy wskaźnik do następnego/poprzedniego elementu.

Kolejki i stosy



★ dodawanie i usuwanie w O(1),

★ wyszukiwanie w zasadzie nie stosowane, ale jeśli trzeba to O(n).

Operacje na kolejkach i stosach

Wstawienie obiektu o do kolejki

```
1 nowy.obiekt := o;
2 nowy.następny := nil;
3 koniec.następny := nowy;
4 koniec := nowy;
```

Odłożenie obiektu o na stos

```
1 nowy.obiekt := o;
2 nowy.następny := wierzchołek;
3 wierzchołek := nowy;
```

Pobranie obiektu z kolejki

```
1 pobrany := początek.obiekt;
2 początek := początek.następny;
```

Pobranie obiektu ze stosu

```
1 pobrany := wierzchołek.obiekt;
2 wierzchołek := wierzchołek.następny;
```

Wyszukiwanie w kolejkach i stosach (i listach jednokierunkowych) jest jak poruszanie się po pomieszczeniach muzeum zgodnie z określonym kierunkiem zwiedzania. Żeby wrócić po poprzedniego pomieszczenia, trzeba dojść do końca i zacząć od nowa.

Listy jednokierunkowe

Listy jednokierunkowe mają strukturę wewnętrzną taką jak kolejki (i stosy), ale z założenia pozwalają na dodatkowe operacje. Przede wszystkim:

- ★ dodawanie za wskazanym elementem,
- ★ usuwanie wybranego elementu.

Dodanie elementu za wskazanym

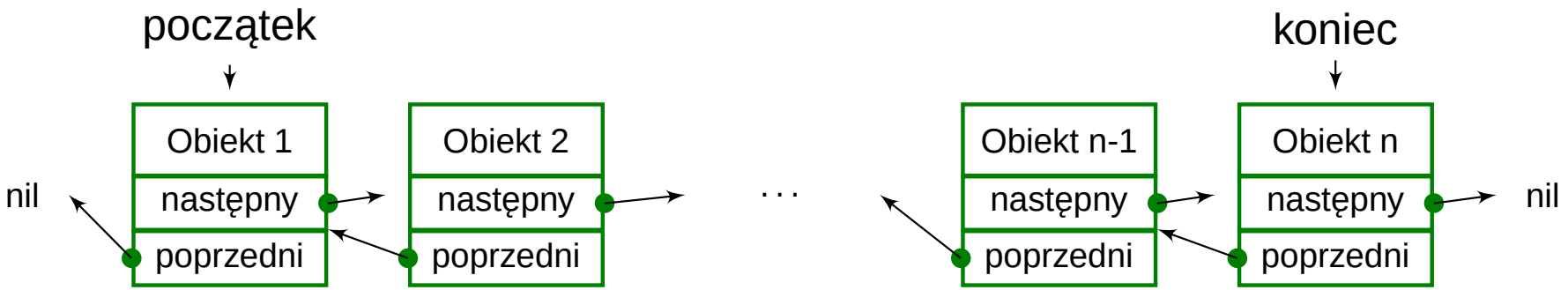
```
1 nowy.następny := wskazany.następny;
2 wskazany.następny := nowy;
```

Dodanie elementu przed wskazanym kosztuje O(n) (za dużo, by używać), bo najpierw trzeba znaleźć element poprzedzający. Jeśli wstawiamy przed pierwszy, to trzeba również zweryfikować wskaźnik początku listy.

Usunięcie elementu wymaga wskazania elementu poprzedniego w stosunku do usuwanego

```
1 usuwany := wskazany.następny;
2 wskazany.następny := usuwany.następny;
```

Listy dwukierunkowe



Dodanie o za wskazanym elementem

```
1 nowy.obiekt := o;
2 nowy.poprzedni := wskazany;
3 nowy.następny := wskazany.następny;
4 wskazany.następny := nowy;
5 nowy.następny.poprzedni := nowy;
```

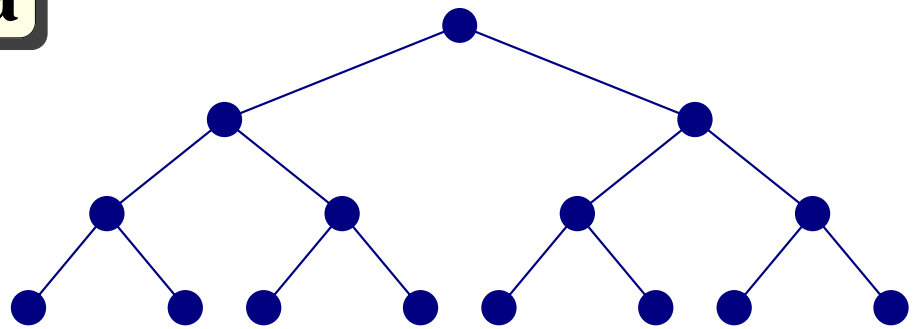
Usunięcie elementu e

```
1 if e.poprzedni != nil
2    then e.poprzedni.następny := e.następny;
3    else początek := e.następny;
4 if e.następny != nil
5    then e.następny.poprzedni := e.poprzedni;
6    else koniec := e.poprzedni;
```

★ dodawanie i usuwanie w O(1),

★ wyszukiwanie w O(n) - nie po to ta struktura.

Drzewa



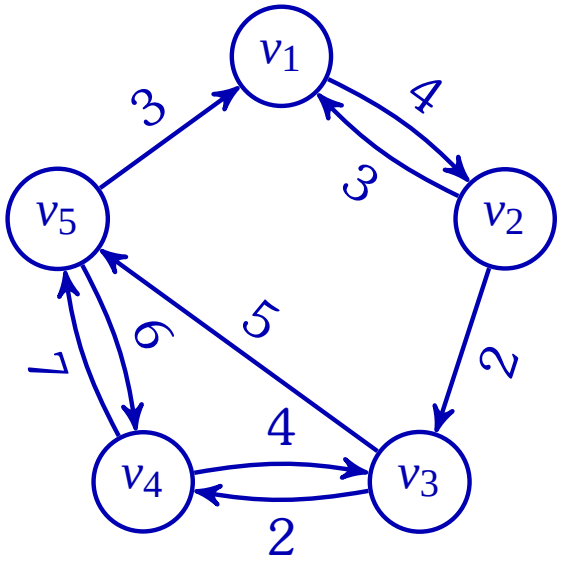
Podstawowe definicje/założenia:

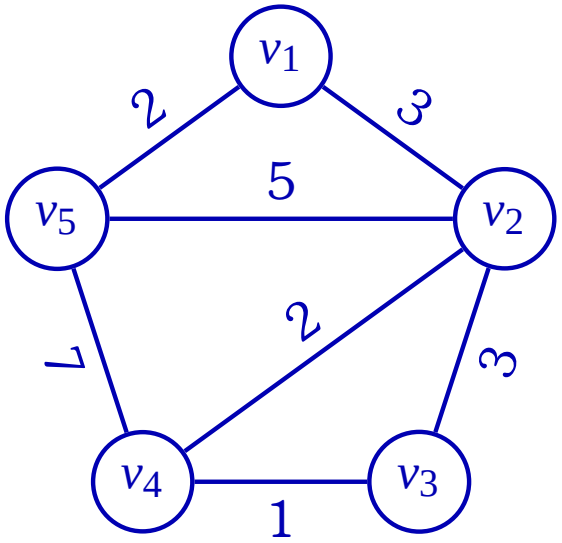
- ★ Drzewa to rekurencyjne kolekcje (podobnie jak kolejki, stosy czy listy), składające się z węzłów **etykietowanych kolekcjonowanymi obiektami**.
- ★ Drzewo jest puste bądź składa się z **korzenia** (*węzeł główny*) i kolekcji jego **poddrzew**.
- ★ Drzewa **binarne** to takie, w których każdy węzeł ma co najwyżej dwa podwęzły (lewy, prawy) a dokładniej **poddrzewa**.
- ★ Drzewa **uporządkowane** (inaczej niezbyt użyteczne): obiekt w danym węźle ma klucz nie mniejszy niż jakikolwiek węzeł jego lewego poddrzewa i nie większy niż jakikolwiek z prawego poddrzewa.
- ★ Drzewo jest **dokładnie wyważone** wtw, gdy dla każdego węzła, liczby węzłów w jego lewym i prawym poddrzewie różnią się co najwyżej o 1.
- ★ Drzewo jest **wyważone** wtw, gdy dla każdego węzła, wysokości dwóch jego poddrzew różnią się co najwyżej o 1.

Grafy

Podstawowe definicje:

- ★ **Graf skierowany (digraf)** – para (V,E), gdzie V to dowolny zbiór (zbiór **wierzchołków**) a E to zbiór **krawędzi** (par wierzchołków, $E \subseteq V \times V$).
- ★ **Graf nieskierowany** – jak skierowany, ale w krawędziach nieistotna jest kolejność wierzchołków.
- ★ **Graf ważony** – graf, w którym dodatkowo każdej krawędzi przypisana jest wartość rzeczywista zwana **wagą**: (V,E,w), $w : E \to R$.





Grafy – reprezentacja tablicowa

Ważony graf skierowany $G = (V, E, w)$ najczęściej reprezentujemy **tablicą** W (**macierzą przyległości**, ang. adjacency matrix) taką, *że*:

$$W[i,j] = \begin{cases} 0 & \text{jeśli } i = j, \\ w(v_i, v_j) & \text{jeśli } (v_i, v_j) \in E \text{ (w grafie istnieje krawędź } (v_i, v_j)), \\ \infty & \text{jeśli } (v_i, v_j) \notin E. \end{cases}$$

Graf nieskierowany można traktować jako skierowany o symetrycznych połaczeniach.

Tablice dla grafów z poprzedniego slajdu:

	1	2	3	4	5
1	0	4	∞	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	∞	∞	6	0

	1	2	3	4	5
1	0	3	∞	∞	2
2	3	0	3	2	5
3	∞	3	0	1	∞
4	∞	2	1	0	7
5	2	5	∞	7	0

Grafy – definicji c.d.

- ★ **Droga** – ciąg wierzchołków taki, że istnieje krawędź z każdego wierzchołka do jego następnika, **droga prosta** – droga, która nie przechodzi dwa razy przez ten sam wierzchołek.
- ★ **Długość drogi** – suma wag krawędzi, z których składa się droga. Dla grafów nieważonych – liczba krawędzi.
- ★ **Cykl** – droga, w której wierzchołek początkowy jest również końcowym, **cykl prosty** – cykl będący drogą prostą.
- ★ **Graf cykliczny/acykliczny** – graf, w którym istnieje/nie istnieje cykl.
- ★ Graf nieskierowany jest **spójny**, gdy między każdymi dwoma wierzchołkami istnieje droga.
- ★ **Drzewo** – graf nieskierowany, acykliczny i spójny.
- ★ **Drzewo rozpinające** graf (ang. spanning tree) – drzewo, które zawiera wszystkie wierzchołki grafu.
- ★ **Minimalne drzewo rozpinające** – drzewo rozpinające o minimalnej sumie wag krawędzi.

Problemy grafowe

Najpopularniejsze problemy:

- ★ Szukanie drogi o minimalnej długości.
- ★ Wyznaczanie minimalnego drzewa rozpinającego:
  - ► jak najmniej asfaltu, by połączyć określone miasta,
  - ► jak najmniej kabli w sieci komputerowej, itp.

Podstawowe algorytmy:

- ★ Szukanie dróg:
  - ► Dijkstry,
  - ► Floyda (alg. programowania dynamicznego).
- ★ Wyznaczanie minimalnego drzewa rozpinającego:
  - ► Prima,
  - ► Kruskala.

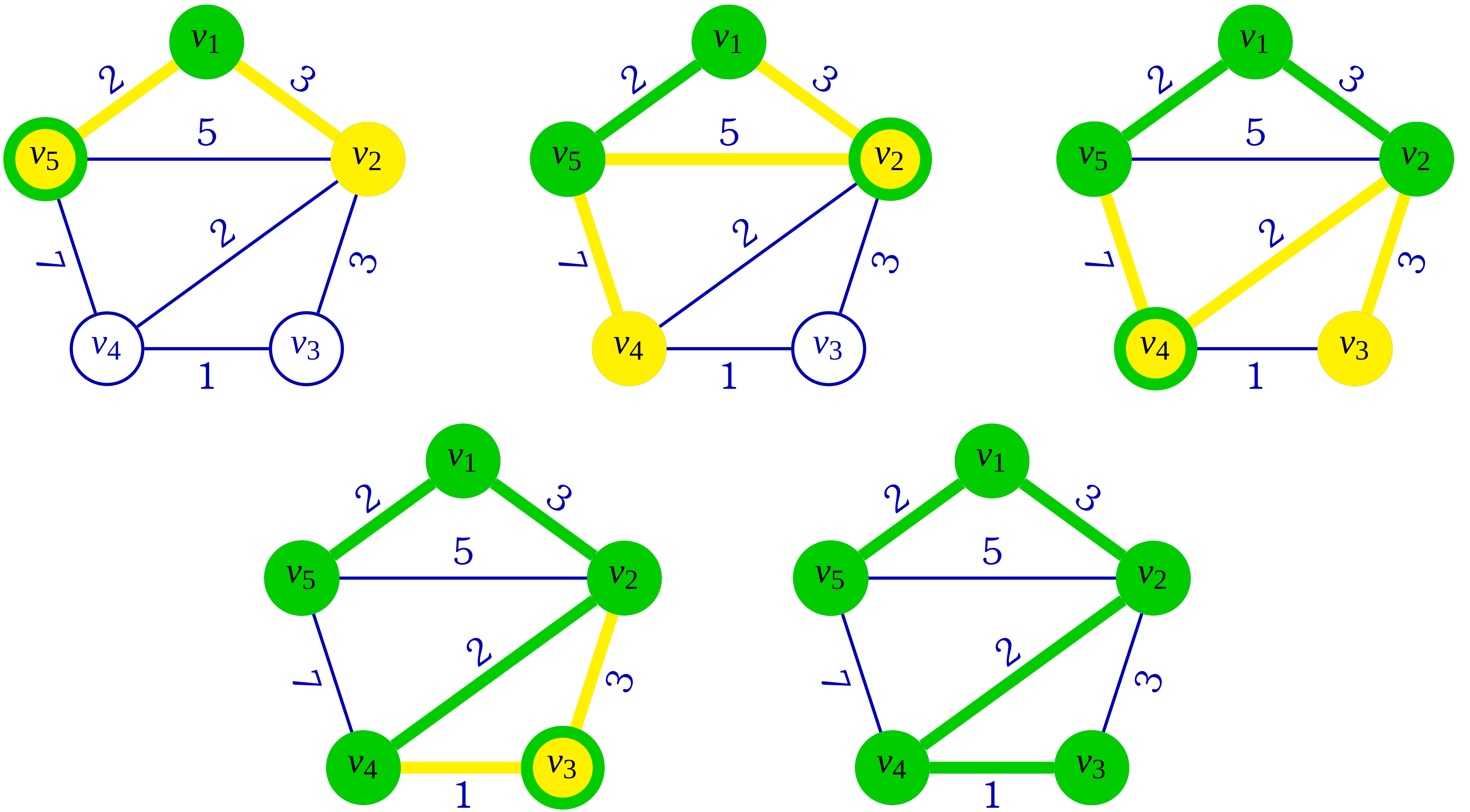
Algorytm Prima

Dane: Graf nieskierowany $G = (V, E, w)$.
Wynik: Minimalne drzewo rozpinające $D = (V, F, w|F)$ grafu G.

- ★ $Y \leftarrow \{v_1\}$
- ★ $F \leftarrow \emptyset$
- ★ Aż do uzyskania drzewa rozpinającego ($Y = V$):
  - ▶ wyznaczamy wierzchołek y z $V - Y$ o minimalnej odległości do Y (jednocześnie zapamiętujemy krawędź f, która dała minimalną odległość),
  - ▶ $Y \leftarrow Y \cup \{y\}$ tzn. dodajemy wierzchołek y do zbioru Y,
  - ▶ $F \leftarrow F \cup \{f\}$ tzn. dodajemy krawędź f do zbioru F,
- ★ Wynik: drzewo $D = (V, F, w|F)$.

Złożoność czasowa: $O(n^2)$ (n to liczba wierzchołków).

Algorytm Prima – przykład



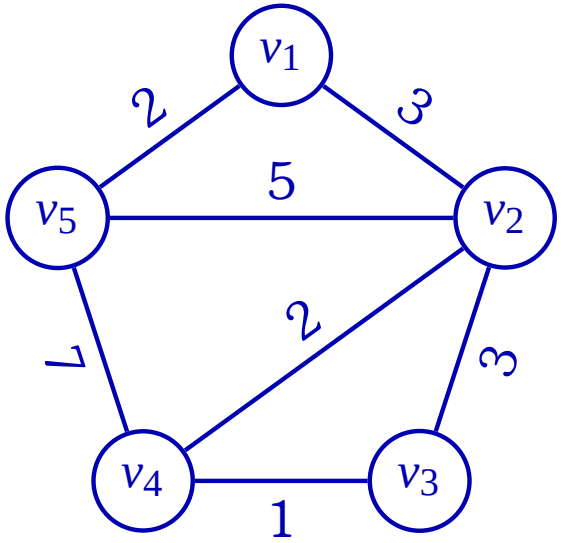
Algorytm Kruskala

Dane: Graf nieskierowany $G = (V, E, w)$.
Wynik: Minimalne drzewo rozpinające $D = (V, F, w|F)$ grafu G.

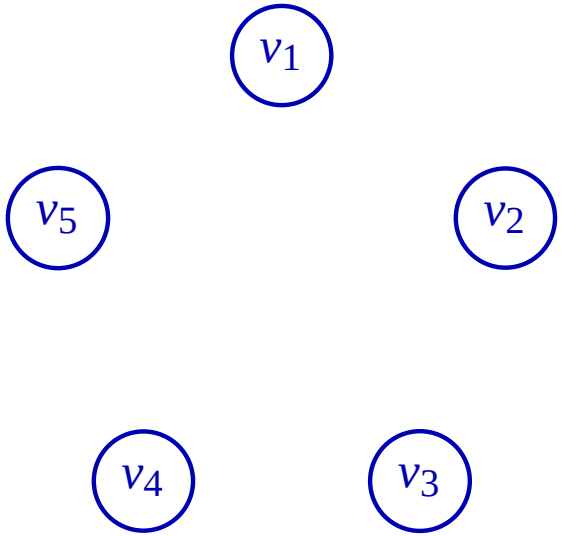
- ★ $V_i \leftarrow \{v_i\}$ – rozłączne jednoelementowe zbiory (dla każdego wierzchołka),
- ★ $F \leftarrow \emptyset$
- ★ Sortujemy krawędzie ze zbioru E niemalejąco względem wag,
- ★ Aż do uzyskania drzewa rozpinającego (wszystkie podzbiory V zostały scalone):
  - ▶ bierzemy kolejną krawędź $f \in E$ wg. ustalonego porządku,
  - ▶ jeśli krawędź f łączy wierzchołki z różnych podzbiorów to:
    - scalamy podzbiory zawierające wierzchołki,
    - $F \leftarrow F \cup \{f\}$ tzn. dodajemy krawędź f do zbioru F,
- ★ Wynik: drzewo $D = (V, F, w|F)$.

Złożoność czasowa: (n – liczba wierzchołków, m – liczba krawędzi)
Inicjowanie zbiorów: $O(n)$, sortowanie: $O(m \log m)$, pętla w najgorszym przypadku $O(m \log m)$ (każda krawędź).
W najgorszym przypadku $m \approx n^2$, więc $m \log m = n^2 \log n^2 = n^2 2 \log n$, czyli $O(n^2 \log n)$.

Algorytm Kruskala - przykład



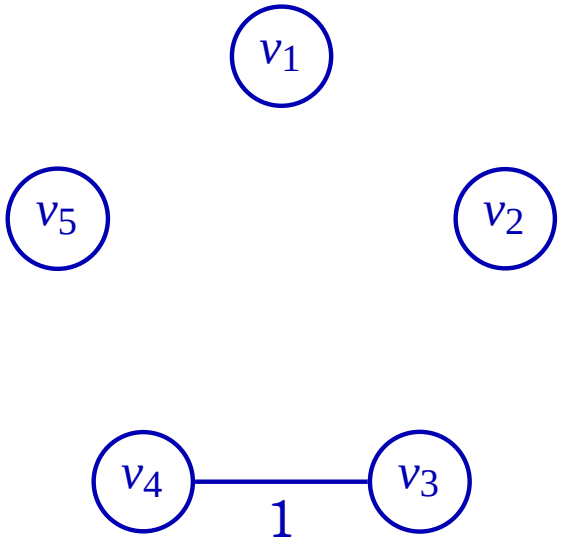
1	(v_3, v_4)	1
2	(v_1, v_5)	2
3	(v_2, v_4)	2
4	(v_2, v_3)	3
5	(v_1, v_2)	3
6	(v_2, v_5)	5
7	(v_4, v_5)	7



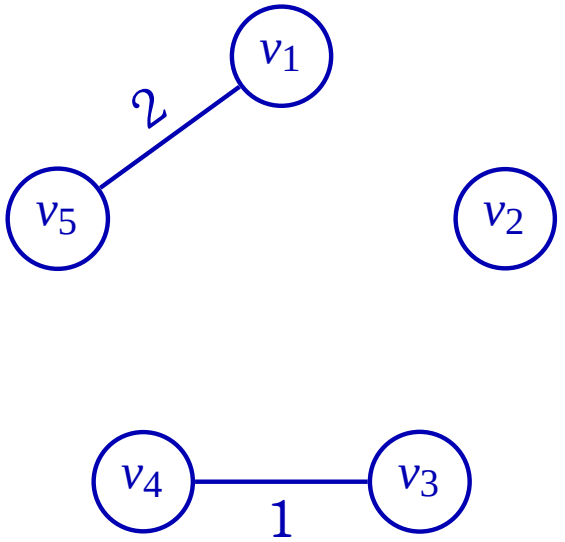
Algorytm Kruskala - przykład c.d.

1	(v_3, v_4)	1
2	(v_1, v_5)	2
3	(v_2, v_4)	2
4	(v_2, v_3)	3
5	(v_1, v_2)	3
6	(v_2, v_5)	5
7	(v_4, v_5)	7

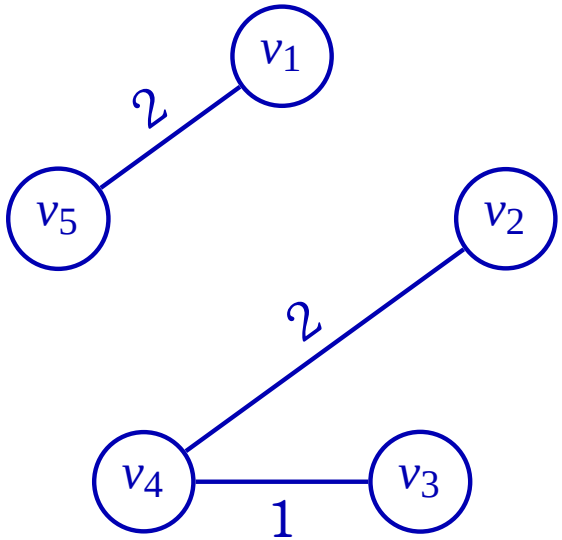
1. Dodajemy (v_3, v_4).



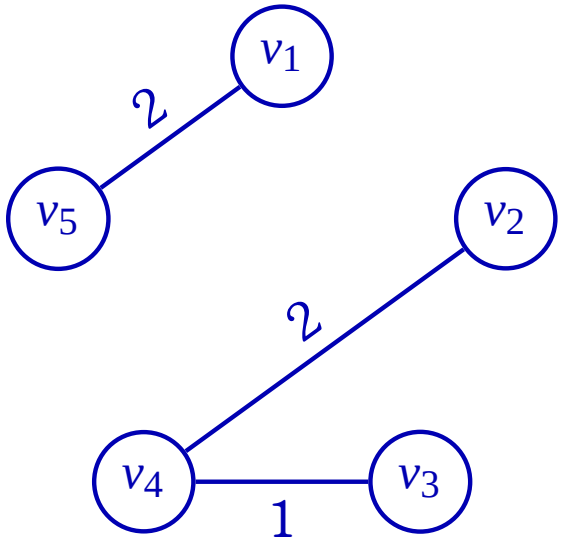
2. Dodajemy (v_1, v_5).



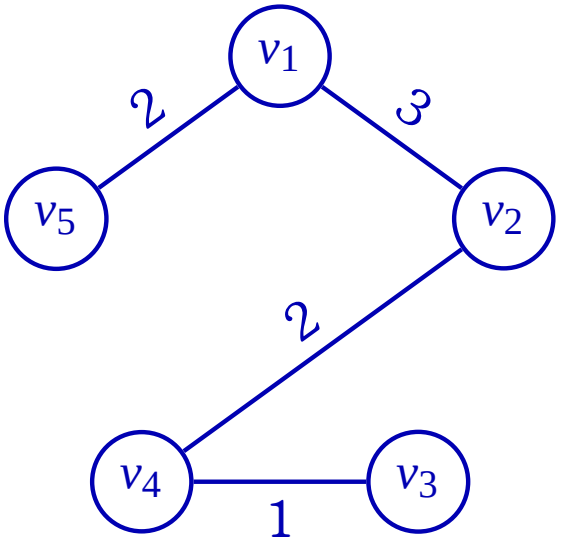
3. Dodajemy (v_2, v_4).



4. Nie dodajemy (v_2, v_3).



5. Dodajemy (v_1, v_2).



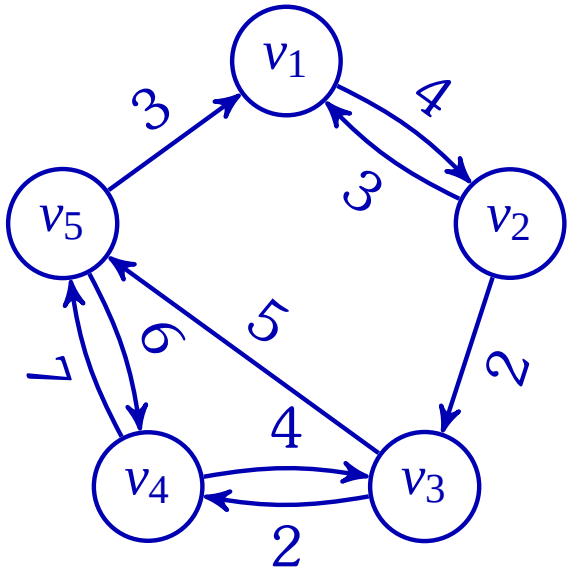
Algorytm Dijkstry

Dane: Graf $G = (V, E, w)$.
Wynik: Najkrótsze (dokładniej: minimalnej długości) drogi z wierzchołka $v \in V$ do wszystkich pozostałych, długości dróg; drzewo rozpinające zawierające najkrótsze drogi z wierzchołka v.

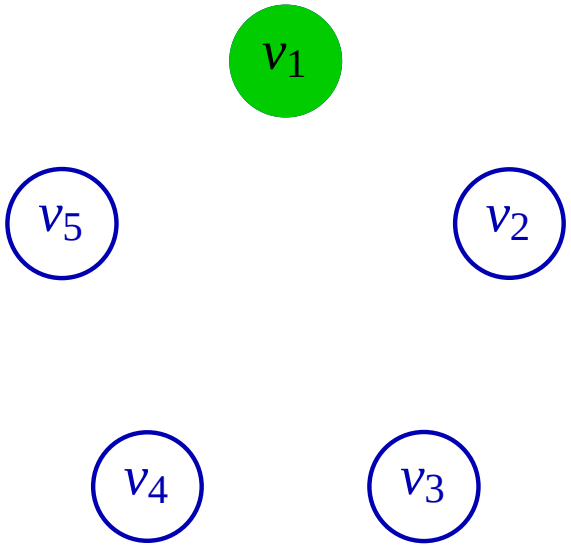
- ★ $Y \leftarrow \{v\}$
- ★ $F \leftarrow \emptyset$
- ★ Tak długo jak $Y \neq V$:
  - ▶ wyznaczamy wierzchołek $y \in V - Y$, o minimalnej długości najkrótszej drogi z v poprzez wierzchołki z Y.
  - ▶ $Y \leftarrow Y \cup \{y\}$,
  - ▶ $F \leftarrow F \cup \{f\}$.

Złożoność czasowa: $O(n^2)$ (n to liczba wierzchołków).

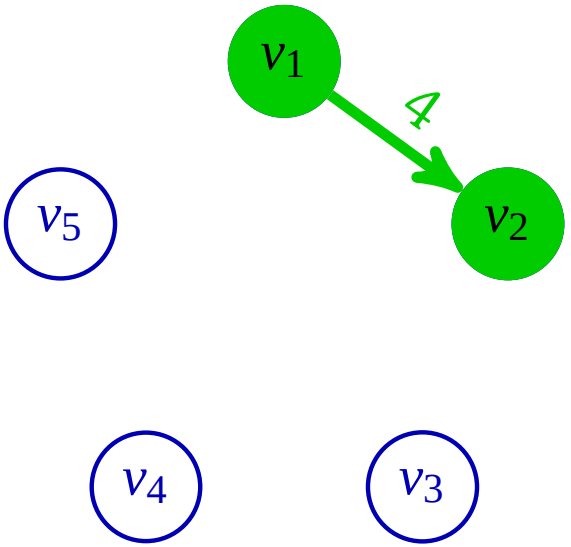
Algorytm Dijkstry - przykład



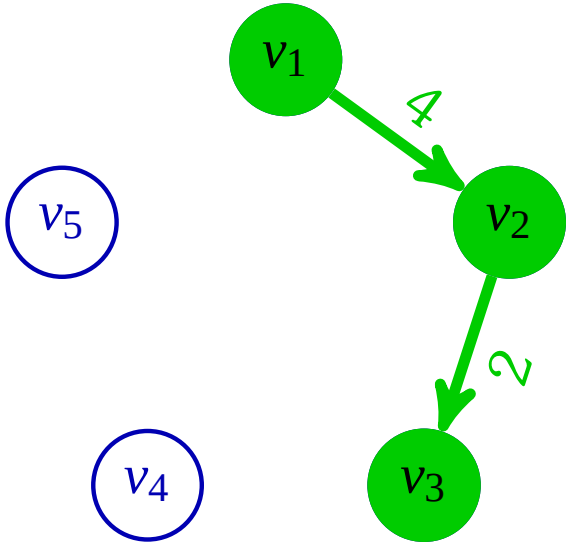
$Y = \{v_1\}$



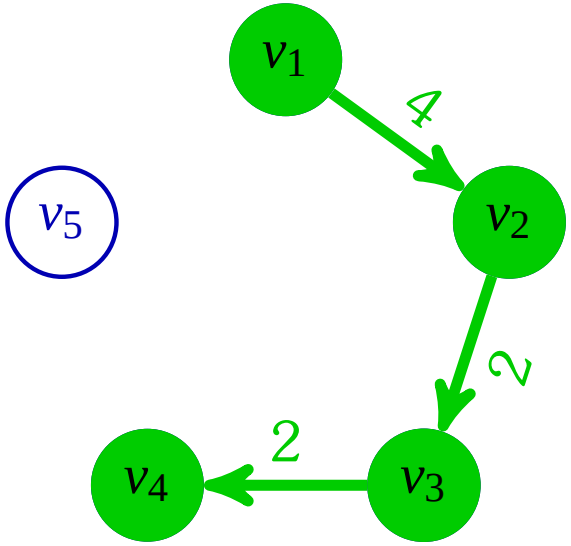
$v_2 \to Y,\ d(v_1, v_2) = 4$



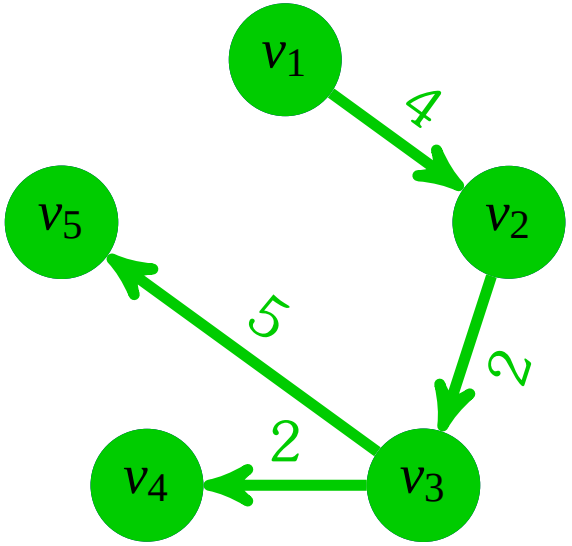
$v_3 \to Y,\ d(v_1, v_3) = 6$



$v_4 \to Y,\ d(v_1, v_4) = 8$



$v_5 \to Y,\ d(v_1, v_5) = 11$



Algorytm Floyda

Idea: Najkrótsza droga z v_i do v_j wiodąca przez v_k składa się z najkrótszej drogi z v_i do v_k oraz najkrótszej drogi z v_k do v_j.

Dane: Graf $G = (V, E, w)$.
Wynik: Długości najkrótszych (dokładniej: minimalnej długości) dróg pomiędzy każdą parą wierzchołków.

```
1  function Floyd(W : array [1..n,1..n] of real) : array [1..n,1..n] of real
2  var D : array [1..n,1..n] of real; i, j, k : integer;
3  begin
4    D := W;
5    for k := 1 to n do
6      for i := 1 to n do
7        for j := 1 to n do
8          D[i,j] := minimum(D[i,j], D[i,k] + D[k,j];
9    Floyd := D;
10 end;
```

Złożoność czasowa: $O(n^3)$ (n to liczba wierzchołków).

Algorytm Floyda

Ważna własność: W k-tym kroku wartość $D[i,j]$ to długość najkrótszej drogi z v_i do v_j o wierzchołkach pośrednich ze zbioru $\{v_1,\ldots,v_k\}$.

Przykład:

D^0	1	2	3	4	5
1	0	4	∞	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	∞	∞	6	0

D^1	1	2	3	4	5
1	0	4	∞	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	∞	6	0

D^2	1	2	3	4	5
1	0	4	6	∞	∞
2	3	0	2	∞	∞
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	9	6	0

D^3	1	2	3	4	5
1	0	4	6	8	11
2	3	0	2	4	7
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	9	6	0

D^4	1	2	3	4	5
1	0	4	6	8	11
2	3	0	2	4	7
3	∞	∞	0	2	5
4	∞	∞	4	0	7
5	3	7	9	6	0

D^5	1	2	3	4	5
1	0	4	6	8	11
2	3	0	2	4	7
3	8	12	0	2	5
4	10	14	4	0	7
5	3	7	9	6	0