SYSTEMY ZAPISU LICZB

- pozycyjne
- niepozycyjne (np. system rzymski)

ZAPIS POZYCYJNY O PODSTAWIE p ($p \geqslant 2, p \in \mathbb{N}$)

$(c_n c_{n-1} \dots c_1 c_0)_p = (c_n \cdot p^n + c_{n-1} \cdot p^{n-1} + \dots + c_1 \cdot p + c_0)_{10}$

Używane znaki: +, -, . , 0,..., p-1

Przykład.
$p = 16$ $\quad 0, 1, \dots, 9, A, B, C, D, E, F$

ZMIANA PODSTAW W SYSTEMIE POZYCYJNYM

$(y)_p = (c_n \dots c_0)_p = (c_n p^n + \dots + c_0 p^0)_{10}$
$(x)_{10} = (d_n \dots d_0)_{10} = (c_k p^k + \dots + c_0 p^0)_{10} = (c_k c_{k-1} \dots c_1 c_0)_p$

Przykład.

$(1012)_3 = (1 \cdot 3^3 + 0 \cdot 3^2 + 1 \cdot 3^1 + 2 \cdot 3^0)_{10} = (27 + 0 + 3 + 2)_{10} = (32)_{10}$
$(19)_{10} = (10011)_2 = (1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (19)_{10}$

Algorytm znajdowania zapisu w układzie pozycyjnym o podstawie p liczby naturalnej x podanej w systemie dziesiętnym

$x_0 = x$	$c_0 = x_0 \% p$
$x_1 = x_0 / p$ (dzielenie całkowite)	$c_1 = x_1 \% p$
$x_2 = x_1 / p$	$c_2 = x_2 \% p$
$\vdots$	
$x_n = x_{n-1} / p$	$c_n = x_n \% p$
0	

$(x)_p = (c_n c_{n-1} \dots c_1 c_0)_p$

Przykład. $x = (195)_{10}, p = 2$

195	$c_0 = 1$
97	$c_1 = 1$
48	$c_2 = 0$
24	$c_3 = 0$
12	$c_4 = 0$
6	$c_5 = 0$
3	$c_6 = 1$
1	$c_7 = 1$
0	

$(195)_{10} = (11000011)_2$

UŁAMKI

$(0.d_{-1} d_{-2} \dots d_{-n})_{10} = d_{-1} \cdot \frac{1}{10} + d_{-2} \cdot \frac{1}{100} + \dots + d_{-n} (\frac{1}{10})^n$

$(0.c_{-1}, c_{-2} \dots c_{-n}) p = (c_{-1} \cdot \frac{1}{p} + \dots + c_{-n} \dots (\frac{1}{p})^n)_{10}$

Przykład.
$(0.121)_3 = (1 \cdot \frac{1}{3} + 2 \cdot \frac{1}{9} + 1 \cdot \frac{1}{27})_{10} = (\frac{9+6+1}{27})_{10} = (\frac{16}{27})_{10} = \ldots$

Algorytm zamiany ułamka x w systemie dziesiętnym na ułamek w systemie o podstawie p

$$\begin{array}{l|l} x_0 = x & \\ x_1 = x_0 \cdot p & c_{-1} = \lfloor x_1 \rfloor \\ x_2 = (x_1 - c_{-1})_p & c_{-2} = \lfloor x_2 \rfloor \end{array}$$

Przykład. Zamiana $(0.48)_{10}$ na zapis w systemie o podstawie $p = 4$.
$(0.48)_{10} = (0.1322\ldots)_4$.

Spostrzeżenie.
Liczba zapisana w systemie o podstawie p może być łatwo przekonwertowana do systemu o podstawie: $p^2, p^3, \ldots$.
Każde k kolejnych cyfr w systemie o podstawie p reprezentuje jedną cyfrę w systemie o podstawie p^k.

Przykład. $(00011011101110101110.101111011000)_2$ zapisać w systemie o postawie 16.
Odp. $(1BBAF.BD8)_{16}$

METODY ZAPISU LICZB $(p = 2)$

Liczby całkowite kodujemy przeznaczając pierwszy bit na znak liczby a pozostałe na rozwinięcie dwójkowe modułu liczby.

LICZBY W ZAPISIE STAŁOPOZYCYJNYM

1 bit znaku, **n** bitów na część całkowitą, **m** bitów na część ułamkową

1) KOD PROSTY (znak - moduł prosty)

dla $x \geqslant 0$ $\quad \underline{0}\ \_\_\_$ - rozwinięcie dwójkowe liczby
dla $x \leqslant 0$ $\quad \underline{1}\ \_\_\_$ - rozwinięcie dwójkowe modułu liczby

Przykład.
Zapisać w kodzie prostym liczby 6.75 i -6.75, przyjmując $n = 4, m = 3$.

$(6)_{10} = (110)_2$

$(0.75)_{10} = (0.11)_2$

$6.75 = \underline{0}\ \underline{0110}\ \underline{110}$
$-6.75 = \underline{1}\ \underline{0110}\ \underline{110}$

Niestety w kodzie prostym liczba 0 jest reprezentowana przez dwa kody:

$0 = \underline{0}\ \underline{0000}\ \underline{000}$
$0 = \underline{1}\ \underline{0000}\ \underline{000}$

Przykłady. Dodawanie liczb zapisanych w kodzie prostym (przyjmując $n = 7, m = 0$): $1+2$, $1+(-2)$, $(-1)+2$, $64+65$.

2) KOD ODWROTNOŚCIOWY (znak - moduł odwrotny)

$x \geqslant 0$ tak samo jak w kodzie prostym
$x \leqslant 0$ $\underline{1}\ \_\ \underline{|\bar{x}|}\ \_$ - powstaje przez zastąpienie zer jedynkami i jedynek zerami

Przykład.
$6.75 = \underline{0}\ \underline{0110}\ \underline{110}$
$-6.75 = \underline{1}\ \underline{1001}\ \underline{001}$

- Dokładność w kodzie prostym i odwrotnościowym: 2^{-m}
- Maksymalna liczba w kodzie prostym i odwrotnościowym:

$\underline{0}\ \underline{11\ldots1}\ \underline{1\ldots1}\ \ 2^n - 1 + 1 - 2^{-n} = 2^n - 2^{-m}$

- Minimalna liczba w kodzie prostym i odwrotnościowym:

$\underline{1}\ \underline{11\ldots1}\ \underline{1\ldots1}\ \ -(2^n - 2^{-m})$

Przykład.
Dodawanie w kodzie odwrotnościowym: $(n = 4, m = 3)$ $6.75 + (-4.75)$
$6.75 = \underline{0}\ \underline{0110}\ \underline{110}$
$-4.75 = \underline{1}\ \underline{1011}\ \underline{001}$ - w kodzie odwrotnościowym
$(4)_{10} = 100 \quad (0.75)_{10} = 0.11 \quad 4.75 = \underline{0}\ \underline{0100}\ \underline{110}$

3) KOD UZUPEŁNIENIOWY (uzupełnienie do 2)

$x \geqslant 0$ $\underline{0}\ \_\ \underline{\text{x}}\ \_$
$x \leqslant 0$ $\underline{1}\ \underline{|\bar{x}| + 0\ldots01}$ - dodawanie 1 do najmłodszego bitu

Przykład.
$6.75 = \underline{0}\ \underline{0110}\ \underline{110}$
$-6.75 = \underline{1}\ \underline{1001}\ \underline{010}$ (w kodzie uzupełnieniowym)

Przykład.
Dodawanie w kodzie uzupełnieniowym: (gdy $n = 4, m = 3$): $6.75 + (-4.75)$
$6.75 = \underline{0}\ \underline{0110}\ \underline{110}$
$-4.75 = \underline{1}\ \underline{1011}\ \underline{001}$ - w kodzie odwrotnościowym
$-4.75 = \underline{1}\ \underline{1011}\ \underline{010}$ - w kodzie uzupełnieniowym

4) KOD NADMIAROWY
Różni się od kodu uzupełnieniowego tym, że dla liczb nieujemnych bit znaku jest 1 a dla ujemnych znak bitu jest 0.

Przykład.
$6.75 = \underline{0}\ \underline{0110}\ \underline{110}$ (w kodzie uzupełnieniowym)
$-6.75 = \underline{1}\ \underline{1001}\ \underline{010}$ (w kodzie uzupełnieniowym)

$6.75 = \underline{1}\ \underline{0110}\ \underline{110}$ (w kodzie nadmiarowym)
$-6.75 = \underline{0}\ \underline{1001}\ \underline{010}$ (w kodzie nadmiarowym)

Przykład.
n=15, m=0
Dla liczby całkowitej $x \in [-2^n, 2^n - 1]$ kod nadmiarowy liczby x, to zapis dwójkowy liczby $x + 2^n \in [0, 2^{n+1} - 1]$.

LICZBY W ZAPISIE ZMIENNOPOZYCYJNYM

$l = m * p^c$
m - mantysa, $\frac{1}{p} \leqslant |m| < 1$
c - cecha
Kodowanie: bit znaku mantysy, d bitów mantysy, bit znaku cechy, w bitów cechy

Przykład.
$6.75 \quad \frac{1}{10} \leqslant m < 1$
$0.675 \cdot 10^1$
$m = 0.675$, $c = 1$.

Przykład. p=2
$(6.75)_{10} = 110\ 11 \rightarrow \quad 0.11011 \cdot 2^3$
$m = 0.11011 \quad \frac{1}{2} \leqslant m < 1$
$c = (3)_{10} = 11$

W kodzie uzupełnieniowym (przy $d = 10, w = 4$), $\underline{0}\ \underline{1101100000}\ \underline{0}\ \underline{0011}$

- Maksymalna liczba dodatnia: $2^{2^{w-1}}(1 - 2^{-d})$
- Minimalna liczba dodatnia: 2^{-2^w}

BINARNE KODY NIEPOZYCYJNE

Przykład.
Kod BCD (binary coded decimals)
$0 \rightarrow 0000$
$1 \rightarrow 0001$
$2 \rightarrow 0010$
$3 \rightarrow 0011 \qquad (625)_{10} = (0110\ 0010\ 0101)_{BCD}$
$4 \rightarrow 0100$
$5 \rightarrow 0101$
$6 \rightarrow 0110$
$7 \rightarrow 0111$
$8 \rightarrow 1000$
$9 \rightarrow 1001$

ZNAKI

Do kodowania znaków wykorzystujemy tablicę znaków ASCII. Każdemu znakowi przyporządkowana jest w sposób jednoznaczny liczba 8-bitowa tzn. liczba $\in \{0, \ldots, 255\}$.

Przykładowe znaki:

'0' - 48	'A' - 65	'a' - 97
⋮	⋮	⋮
'9' - 57	'Z' - 90	'z' - 122

Znaki sterujące:

EOF	- 27	(end of file)
LF	- 10	(przesunięcie linii)
CR	- 13	(powrót karetki)
Nul	- 0	(znak pusty)

ANALIZA ALGORYTMÓW

Analiza algorytmu polega na określeniu zasobów potrzebnych do jego wykonania (tzn. czas, pamięć).

Na ogół, szukamy kompromisu między prostotą algorytmu, łatwością zrozumienia implementacji, a efektywnością wykorzystywania zasobów.

Jeśli algorytm będzie wykorzystywany parokrotnie, to należy wybrać taki, który jest prostszy do napisania.
Jeśli zaś będzie on wykorzystywany wielokrotnie, to większą rolę odgrywa wtedy szybkość jego działania.

Czas wykonywania programu zależy od następujących czynników:

1) jakość kodu wygenerowanego przez kompilator, którego użyliśmy do stworzenia pliku wynikowego,

2) rodzaj i szybkość instrukcji maszynowych użytych do wykonania programu (zależy głównie od szybkości komputera),

3) wielkość i rodzaj danych wejściowych programu,

4) złożoność czasowa algorytmu, na podstawie którego powstał program.

Czas wykonywania algorytmu dla konkretnych danych wyrażamy **liczbą** wykonywanych tzw. **operacji elementarnych** (zakładamy ich maszynową niezależność i wykonywanie danego typu operacji w stałym czasie).

Będziemy rozważać **złożoność czasową pesymistyczną** czyli czas wykonania algorytmu dla ustalonego rozmiaru danych wejściowych w najgorszym przypadku oraz **średnią złożoność czasową**.

Obliczając złożoność czasową stosujemy najczęściej tzw. kryterium jednorodne, przyjmując założenie, że każda operacja elementarna zajmuje czas jednostkowy.

Oznaczenia:

A – algorytm

D – dziedzina algorytmu (zbiór danych wejściowych)

$D \supseteq D_n$ – zbiór danych wejściowych rozmiaru n

$D_n \ni d$ – dana wejściowa rozmiaru n

Rozmiar danej wejściowej d to sensownie określona wielkość d (np. rozmiar tablicy, liczba cyfr liczby w zapisie dziesiętnym, ...).

$t(d)$ – liczba jednostek czasowych potrzebnych do wykonania algorytmu A dla danej wejściowej d

$s(d)$ – liczba komórek pamięci potrzebnych do wykonania algorytmu A dla danej wejściowej d

Złożonością czasową pesymistyczną algorytmu A nazywamy funkcję $T : \mathbb{N} \to \mathbb{R}_+$ taką, że:

$$T(n) = \max\{t(d) : d \in D_n\}$$

Złożonością pamięciową algorytmu A nazywamy funkcję $S : \mathbb{N} \to \mathbb{R}_+$ taką, że:

$$S(n) = \max\{s(d) : d \in D_n\}$$

Złożonością czasową średnią algorytmu A nazywamy funkcję $A : \mathbb{N} \to \mathbb{R}_+$ taką, że:

$$A(n) = \sum_{d \in D_n} t(d) \cdot p_n(d)$$

gdzie $p_n(d)$ oznacza prawdopodobieństwo z jakim dana d pojawia się na wejściu (w stosunku do wszystkich danych rozmiaru n).

Przykład. Wyszukiwanie elementu w tablicy.

Dane: int T[MAX], n, x.

Cel: znaleźć taki indeks i, że T[i] jest równe x.

```
int szukaj (int T[MAX], int n, int x)
{
  int i;
  i=0;
  while (i<n)
   {
      if(T[i]==x) return i;
        else i++;
   }
  return -1; //jeśli nie znaleziono x;
}
```

I sposób.
Przyjmujemy, że operacją dominującą jest operacja porównywania x z elementami tablicy $T[i]$.

- Pesymistyczny przypadek: $T(n) = n$
- Średnia złożoność: Przyjmujemy, że prawdopodobieństwo, że element x jest na pozycji i-tej takie samo dla każdego i oraz jest równe prawdopodobieństwu, że x nie występuje w tablicy, czyli wszystkie prawdopodobieństwa są równe $\frac{1}{n+1}$.

$$A(n) = 1 \cdot \frac{1}{n+1} + 2 \cdot \frac{1}{n+1} + \cdots + n \cdot \frac{1}{n+1} + n \cdot \frac{1}{n+1} = \frac{1}{n+1} \sum_{i=1}^{n} i + \frac{n}{n+1}$$

$$= \frac{1}{n+1} \cdot \frac{1+n}{2} \cdot n + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

II sposób
Przyjmujemy, że operacjami dominującymi są wszystkie operacje porównania i operacje przypisania.

- Operacje porównywania w pesymistycznym przypadku:
  $T_1(n) = \underset{==}{n} + \underset{i<n}{n} + 1$
- Operacje podstawienia w pesymistycznym przypadku:
  $T_2(n) = 1 + n$

  $T(n) = T_1(n) + T_2(n) = 3n + 1$

- Średnia złożoność:

$$A(n) = 3 \cdot \frac{1}{n+1} + 6 \cdot \frac{1}{n+1} + 9 \cdot \frac{1}{n+1} + \cdots + 3n \cdot \frac{1}{n+1} + (3n+1) \cdot \frac{1}{n+1} =$$

$$= \frac{1}{n+1}(3 + 6 + \cdots + 3n) + \frac{3n+1}{n+1} = \frac{3}{n+1} \cdot \frac{n(n+1)}{2} + \frac{3n+1}{n+1} = \frac{3}{2}n + \frac{3n+1}{n+1}$$

Złożoność pamięciowa:
$S(n) = n + \underset{i}{1} + \underset{n}{1} + \underset{x}{1} = n + 3$

NOTACJA O, Ω, Θ

Definicja
$f, g : \mathbb{N} \to \mathbb{R}_+$
Mówimy, że $f(n) = O(g(n)) \overset{df}{\Leftrightarrow} \exists_{c>0} \exists_{n_0 \in \mathbb{N}} \forall_{n \geqslant n_0} f(n) \leqslant c \cdot g(n)$
$f(n)$ jest asymptotycznie nie większa niż $g(n)$,
$f(n)$ jest rzędu niewiększego od $g(n)$.

Przykład.

- $f(n) = n$, $g(n) = n^2$
  $\exists_{c>0} \exists_{n_0 \in \mathbb{N}} \forall_{n \geqslant n_0} n \leqslant c \cdot n^2 \quad c = 1$
- $f(n) = n$, $g(n) = n \log n$
  $n \leqslant c \cdot n \log n \quad n = 0(n \log n)$

Spostrzeżenie
$\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} < +\infty \Rightarrow f(n) = 0(g(n))$

Definicja
$f(n) = \Omega(g(n)) \overset{df}{\Leftrightarrow} \exists_{c>0} \exists_{n_0 \in \mathbb{N}} \forall_{n \geqslant n_0} f(n) \geqslant c \cdot g(n)$
$f(n)$ jest asymptotycznie nie mniejsza niż $g(n)$,
$f(n)$ jest rzędu niemniejszego niż $g(n)$.

Przykład.
$n \log n = \Omega(n)$

Spostrzeżenie
$\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$

Definicja
$f(n) = \Theta(g(n)) \overset{df}{\Leftrightarrow} \exists_{c_1>0, c_2>0} \exists_{n_0 \in \mathbb{N}} \forall_{n \geqslant n_0} c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$
$f(n)$ jest asymptotycznie równa $g(n)$,
$f(n)$ jest tego samego rzędu jak $g(n)$.

Przykład.

Czy $2^n = \Theta(3^n)$?

$$2^n = \Theta(3^n) \Leftrightarrow \exists_{\substack{c_1,c_2>0 \\ c_2=1}} \ldots c_1 \cdot 3^n \leqslant 2^n \leqslant c_2 \cdot 3^n$$

$c_1 \leqslant \frac{2^n}{3^n}$
$c_1 \leqslant (\frac{2}{3})^n$
$\Rightarrow 2^n \neq \Theta(3^n)$

Spostrzeżenie
$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < +\infty \Rightarrow f(n) = \Theta(g(n))$

Przykład. Asymptotyczna złożoność algorytmu wyszukiwania elementu w tablicy.

Pesymistyczny przypadek:

$T(n) = n$ – gdy za operację dominującą przyjęliśmy tylko porównania elementów tablicy z x-em

$T(n) = 3n + 1$ – gdy przyjęliśmy, że operacjami dominującymi są wszystkie operacje porównania i operacje przypisania.

W obu przypadkach **asymptotyczna pesymistyczna złożoność** wynosi $T(n) = \Theta(n)$.

Średnia złożoność:

$A(n) = \frac{n}{2} + \frac{n}{n+1}$
$A(n) = \frac{3}{2}n + \frac{3n+1}{n+1}$

W obu przypadkach **asymptotyczna średnia złożoność** wynosi $A(n) = \Theta(n)$.

OBLICZANIE ZŁOŻONOŚCI ASYMPTOTYCZNEJ

1) Czas wykonywania każdej instrukcji porównania, podstawienia (przypisania), czytania, pisania wynosi $\Theta(1)$ (jest stała).
2) Czas wykonywania sekwencji instrukcji oblicza się stosując **regułę sumowania**.
3) Czas wykonywania instrukcji warunkowej jest sumą czasów sprawdzenia warunku i instrukcji wykonywanej (lub większemu z czasów w instrukcji warunkowej długiej).
4) Czas wykonywania instrukcji iteracyjnej (pętli) jest sumą po wszystkich iteracjach czasów wykonywania wnętrza pętli i czasu sprawdzania warunku zakończenia pętli. Można zastosować **regułę mnożenia**.

REGUŁA SUMOWANIA

Niech sekwencja instrukcji składa się z dwóch części: P_1, P_2.
$T_1(n) = \Theta(f(n))$ - czas wykonywania P_1,
$T_2(n) = \Theta(g(n))$ - czas wykonywania P_2.

Wtedy sumaryczny czas wykonywania obu cześci P_1 i P_2 wynosi:
$T_1(n) + T_2(n) = \Theta(f(n)) + \Theta(g(n)) = \Theta(\max\{f(n), g(n)\})$.

Jeśli wiemy tylko, że $T_1(n) = O(f(n))$ i $T_2(n) = O(g(n))$, to wówczas $T_1(n) + T_2(n) = 0(f(n)) + 0(g(n)) = 0(\max\{f(n), g(n)\})$.

REGUŁA MNOŻENIA

Jeżeli część algorytmu o złożoności $T(n) = \Theta(f(n))$ jest wykonywana $\Theta(g(n))$ razy, to złożoność całości wynosi $\Theta(f(n) \cdot g(n))$.

Przykład. Szybkie oszacowanie **asymptotycznej** złożoności algorytmu wyszukiwania elementu w tablicy.

```
int szukaj (int T[MAX], int n, int x)
{
    int i;
    i=0;
    while (i<n)
     {
      if(T[i]==x) return i;
      else i++;
      }
    return -1; //jeśi nie znaleziono x;
}
```

Każda z operacji w pętli ma złożoność $\Theta(1)$. Z reguły sumowania złożoność wnętrza pętli jest równa $\Theta(\max\{1,1,1\}) = \Theta(1)$. W pesymistycznym przypadku pętla wykonywana jest n razy czyli $\Theta(n)$ razy. Zatem z reguły mnożenia wykonanie części algorytmu, w której jest pętla, ma złożoność $\Theta(1 \cdot n) = \Theta(n)$. Pozostała część algorytmu ma (z reguły sumowania) złożoność $\Theta(1)$. Zatem całość algorytmu ma (znów z reguły sumowania) złożoność $\Theta(\max\{n,1\}) = \Theta(n)$.

Przykład.
Mamy do dyspozycji pięć algorytmów $A_1, \ldots, A_5$ do rozwiązania pewnego zadania w czasie 1 minuty na komputerze o wydajności $2 \cdot 10^9$ operacji algorytmu na sekundę.

Algorytmy	Złożoność asymptotyczna algorytmu	Maksymalny rozmiar zadania n
A_1	n	$1,2 \cdot 10^{11}$
A_2	$n \log n$	$3,8 \cdot 10^9$
A_3	n^3	$4,9 \cdot 10^3$
A_4	2^n	37
A_5	$n!$	14

Czas wykonania algorytmu dla zadania o rozmiarze n

Algorytmy	Złożoność	n=10	n=20	n=25	n=50
A_1	n	$5 \cdot 10^{-9}s$	$10^{-8}s$	$1,25 \cdot 10^{-8}s$	
A_2	$n \log n$	$1,7 \cdot 10^{-8}s$	$4,3 \cdot 10^{-8}s$	$5,8 \cdot 10^{-8}$	
A_3	n^3	$3 \cdot 10^{-7}$	$4 \cdot 10^{-6}$	$8 \cdot 10^{-6}s$	$12 \cdot 10^{-2}s$
A_4	2^n	$5,1 \cdot 10^{-7}s$	$5,2 \cdot 10^{-4}s$	$1,7 \cdot 10^{-2}s$	$35,7lat$
A_5	$n!$	$1,8 \cdot 10^{-7}s$	$38,6lat$	$2,5 \cdot 10^{7}lat$	

KLASYFIKACJA PROBLEMÓW OBLICZENIOWYCH

- problemy nierozstrzygalne - np. problem stopu
- problemy trudne - problemy, dla których wiadomo, że ich złożoność nie jest wielomianowa (co najmniej wykładnicza) (np. wygenerowanie wszystkich permutacji dla danego n)
- problemy NP - problemy dla których (przynajmniej teoretycznie) mogą istnieć algorytmy wielomianowe
- problemy P (łatwe) - problemy, dla których znaleziono algorytmy wielomianowe
- problemy NP - zupełne - najtrudniejsze problemy w NP dla których do tej pory nie znaleziono algorytmów wielomianowych

Przykład.
Danych jest n miast i wszystkie odległości między parami miast.

- Problem wyznaczenia najkrótszej drogi między dwoma zadanymi miastami (ma złożoność $0(n^2)$).
- Problem komiwojażera (przedstawiciela handlowego)

Znaleźć najkrótszą drogę poprzez wszystkie miasta tak, aby:

1) wrócić do miasta, z którego rozpoczęto drogę,

2) w każdym mieście być dokładnie jeden raz.

Do tej pory nie znaleiono algorytmu rozwiązującego problem komiwojażera w czasie wielomianowym.
Problem komiwojażera jest w klasie problemów NP-zupełnych.

Hipoteza: $P = NP$?

Przykład. Wyszukiwanie elementu w tablicy uporządkowanej.
Zakładamy, że elementy w tablicy T są ustawione w ciąg niemalejący.

```
int szukaj2(int lewy, int prawy, int T[max], int x)
{
    int s;
    s=(lewy+prawy)/2;
    if(x==T[s]) return s;
    if(lewy==prawy) return -1;
    if(x<T[s]) return (szukaj2(lewy,s-1,T,x);
    else return(szukaj2(s+1,prawy,T,x);

}

printf("%d",szukaj2(0,n-1,T,x)); //wywołanie w programie
```

Złożoność pesymistyczna algorytmu:

$$\begin{cases} T(1) = \Theta(1) & (=3) \\ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) & (=4) \end{cases}$$

TW. O REKURSJI UNIWERSALNEJ

Zał: $a \geqslant 1, b > 1$
$f : \mathbb{N} \to \mathbb{R}_+$
$T(n) = a \cdot T(\frac{n}{b}) + f(n)$

Teza:

1. Jeśli $f(n) = \Theta(n^{log_b a})$, to $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
2. Jeśli $f(n) = 0(n^{log_b a - \varepsilon})$, dla pewnego $\varepsilon > 0$, to $T(n) = \Theta(n^{log_b a})$
3. Jeśli $f(n) = \Theta(n^{log_b a + \varepsilon})$ dla pewnego $\varepsilon > 0$ i istnieją takie $c < 1$ i n_0, że $a \cdot f(\frac{n}{b}) \leqslant c \cdot f(n)$ dla każdego $n \geqslant n_0$, to $T(n) = \Theta(f(n))$.

Przykład. Oszacowanie złożoności pesymistycznej algorytmu wyszukiwania elementu w tablicy uporządkowanej.

$$\begin{cases} T(1) = \Theta(1) & (=3) \\ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) & (=4) \end{cases}$$

$a = 1, b = 2$
$f(n) = \Theta(1)$
$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
$f(n)$ spełnia 1) $\Rightarrow T(n) = \Theta(1 \cdot \log_2 n) = \Theta(\log_2 n)$

Przykład.
Złożoność czasowa algorytmu rekurencyjnego dana jest równaniem rekurencyjnym: $T(n) = T(\frac{n}{2}) + f(n)$
Wyznacz asymptotyczną złożoność czasową algorytmu, jeśli:

a) $f(n) = n + 1$

b) $f(n) = \frac{n(n-1)}{2}$

c) $f(n) = n^3$

d) $f(n) = n \log n$

a) $f(n) = n + 1$

$n + 1 = 0(n)$, bo:

$\exists_{c>0} \exists_{n_0} \forall_{n \geqslant n_0} \;\; n + 1 \leqslant c \cdot n, c = 2 \;\; n + 1 \leqslant 2n$

$n + 1 = 0(n^{2-\varepsilon}), \varepsilon = 1$

$\overset{tw.2)}{\Rightarrow} T(n) = \Theta(n^2)$

b) $f(n) = \dfrac{n(n-1)}{2}$

$\dfrac{n(n-1)}{2} = \Theta(n^2)$, bo:

$\lim\limits_{n\to\infty} \dfrac{f(n)}{n^2} = \lim\limits_{n\to\infty} \dfrac{n-1}{2n} = \dfrac{1}{2}$

$0 < \dfrac{1}{2} < \infty \Rightarrow \dfrac{n(n-1)}{2} = \Theta(n^2)$

$\overset{tw.1)}{\Rightarrow} T(n) = \Theta(n^2 \cdot \log n)$

c) $f(n) = n^3$

$n^3 = \Omega(n^3)$, bo:

$\exists_{c>0} \exists_{n_0} n^3 \geqslant c \cdot n^3, c = 1$

$n^3 = \Omega(n^{2+\varepsilon}), \varepsilon = 1$

$\overset{tw.3)}{\Rightarrow} T(n) = \Theta(n^3)$

d) $f(n) = n \log n$

$n \log n = 0(n^{2-\varepsilon})$, $\varepsilon = ?$

$\varepsilon = \frac{1}{2}$, $n \log n \overset{?}{=} 0(n^{\frac{3}{2}})$

$\lim\limits_{n\to\infty} \dfrac{f(n)}{n^{\frac{3}{2}}} = \lim\limits_{n\to\infty} \dfrac{n \log n}{n^{\frac{3}{2}}} = \lim\limits_{n\to\infty} \dfrac{\log n}{\sqrt{n}} = 0$

$\Rightarrow n\log n = O(n^{\frac{3}{2}})$

$\overset{tw.2)}{\Rightarrow} T(n) = \Theta(n^2)$

SORTOWANIE

Dane: ciąg $a_0, \ldots, a_{n-1}$ o długości n $(a_i \in \mathbb{R})$

Cel: uporządkować elementy ciągu (a_i) tak, aby występowały w porządku niemalejącym: $a_{i_0} \leqslant a_{i_1} \leqslant \cdots \leqslant a_{i_{n-1}}$

Zakładamy, że elementy ciągu są umieszczone w tablicy (jednowymiarowej).

```
int A[MAX], n;
```

Na potrzeby analizy złożoności czasowej średniej zakładamy, że wszystkie elementy w ciągu są parami różne oraz że każda permutacja elementów na wejściu jest jednakowo prawdopodobna i jej prawdopodobieństwo wystąpienia jest równe $\frac{1}{n!}$.

1. SORTOWANIE PRZEZ PROSTE WYBIERANIE

5 7 2 1 3 4 6
1|7 2 5 3 4 6
1 2|7 5 3 4 6
1 2 3|5 7 4 6
1 2 3 - posortowana część ciągu
5 7 4 6 - nieposortowana część ciągu

```
ProsteWybieranie(int A[MAX], int n)
{
    int i,j,k,min;
    for(i=0;i<n-1;i++)
    {
      k=i;
      min=A[i];
      for (j=i+1;j<n;j++)
        if(A[j]<min)
          { min=A[j];
            k=i;
          }
      A[k]=A[i]; // zamiana A[k] z A[i]
      A[i]=min;
    }
}
```

Złożoność czasowa

Liczba porównań między elementami ciągu ($A[j] < min$):

$$(n-1)+(n-2)+\cdots+1 = \frac{(n-1)+1}{2}(n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Liczba wszystkich porównań również jest równa $\Theta(n^2)$.

Liczba podstawień również jest równa $\Theta(n^2)$.

Liczba zamian elementów w tablicy jest równa $n-1$ (z tym, że niekoniecznie wszystkie zamiany zamieniają dwa różne elementy)

FAKT. Złożoność czasowa pesymistyczna i średnia jest równa $T(n) = A(n) = \Theta(n^2)$.

FAKT. Złożoność pamięciowa jest równa $S(n) = n+5$.

2. SORTOWANIE BĄBELKOWE

```
Babelkowe(int A[MAX], int n)
{int i,j,x;
 for(i=0; i<n-1; i++)
  {
    for(j=n-1; j>i; j--)
      if(A[j-1]>A[j])
       {x=A[j-1]; A[j-1]=A[j]; A[j]=x;
            //zamiana A[j-1] z A[j]
       }
  }
}
```

Złożoność czasowa

Liczba porównań między elementami ciągu ($A[j-1] > A[j]$):

$$(n-1)+(n-2)+\cdots+1 = \frac{(n-1)+1}{2}(n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Liczba zamian elementów w tablicy:
a) w przypadku pesymistycznym (gdy elementy ciągu są w porządku odwrotnym np. 7 6 5 4 3 2 1) jest równa $\frac{1}{2}n(n-1)$,

b) w przypadku optymistycznym (gdy elementy ciągu już są uporządkowane np. 1 2 3 4 5 6 7) jest równa 0.

Średnia liczba zamian elementów jest $\Theta(n^2)$. Z tego też powodu sortowanie bąbelkowe ma średnią złożoność gorszą niż sortowanie przez proste wybieranie (asymptotycznie taką samą ale z większym współczynnikiem).

FAKT. Złożoność czasowa pesymistyczna i średnia jest równa $T(n) = A(n) = \Theta(n^2)$.

FAKT. Złożoność pamięciowa jest równa $S(n) = n+4$.

3. SORTOWANIE PRZEZ PROSTE WSTAWIANIE

```
ProsteWstawianie(int A[MAX], int n)
{int i,j,x;
 for(i=1;i<n;i++)
 {x=A[i];
  j=i-1;
  while(j>=0 && A[j]>x)
  {A[j+1]=A[j];
   j--;
  }
  A[j+1]=x;
 }
}
```

Złożoność czasowa

Liczba porównań między elementami ciągu ($A[j] > x$):

a) w przypadku pesymistycznym (gdy elementy ciągu są w porządku odwrotnym) jest równa $1+2+\cdots+n-1 = \frac{n(n-1)}{2}$.

b) w przypadku optymistycznym (gdy elementy ciągu już są uporządkowane) jest równa $n-1$.

Średnia liczba porównań między elementami ciągu jest równa $\Theta(n^2)$.

Średnia liczba wszystkich podstawień jest równa $\Theta(n^2)$.

FAKT. Złożoność czasowa pesymistyczna i średnia jest równa $T(n) = A(n) = \Theta(n^2)$.

FAKT. Złożoność pamięciowa jest równa $S(n) = n+4$.

Sortowanie przez wstawianie połówkowe –
modyfikacja sortowania przez proste wybieranie w którym miejsce wstawienia elementu x znajdujemy wykorzystując algorytm wyszukiwania połówkowego.

Liczba porównań w przypadku pesymistycznym jest równa $\Theta(n\log_2 n)$.

Niestety liczba podstawień jest równa $\Theta(n^2)$.

Ostatecznie, $T(n) = A(n) = \Theta(n^2)$

4. SORTOWANIE PRZEZ SCALANIE

```
Scalanie(int l, int p, int A[MAX], int n)
 {int i,j,k,m,B[MAX];
  k=(l+p)/2;
  if(k>l) scalanie(l,k,A,n);
  if(k+1<p) scalanie(k+1,p,A,n);
  i=l; j=k+1; m=0;
  while(i<=k && j<=p)
     if(A[i]<=A[j])
     {B[m]=A[i];
      i++;
      m++;
     }
     else
     {B[m]=A[j];
      j++;
      m++;
     }

     while(i<=k)
     {B[m]=A[i];
      i++;
      m++;
     }

     while(j<=p)
```

```
    {B[m]=A[j];
     j++;
     m++;
    }

  for(int i=0; i<m; i++)
  A[l+i]=B[i];
}
```

Wywołanie w programie głównym: Scalanie(0, n-1, A, n);

Asymptotyczna pesymistyczna złożoność czasowa

$$T(n) = 2 \cdot T(\tfrac{n}{2}) + n - 1$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = n - 1 = \Theta(n)$$

Z tw. o rekursji otrzymujemy, że $T(n) = \Theta(n \cdot \log_2 n)$

FAKT. Złożoność czasowa średnia również jest równa $A(n) = \theta(n \cdot \log_2 n)$

WADĄ sortowania przez scalanie jest bardzo duża złożoność pamięciowa wynikająca z użycia dodatkowej tablicy B[MAX] przy każdym kolejnym rekurencyjnym wywołaniu sortowania.

Ćwiczenie domowe. Oblicz złożoność pamięciową algorytmu sortowania przez scalanie zakładając, że $n = MAX = 2^k$.

5. SORTOWANIE SZYBKIE

```
Szybkie(int l, int p, int A[MAX], int n)
{int i,j,x,z;
 i=l; j=p;
 x=A[(l+p)/2];

 while(i<=j)
 {while (A[i]<x) i++;
  while (A[j]>x) j--;

  if(i<=j)
  {z=A[i];
   A[i]=A[j];
   A[j]=z;
   i++;
   j--;
  }
 }

 if (j>l) szybkie (l,j,A,n);
 if (i<p) szybkie (i,p,A,n );
}
```

Wywołanie w programie głównym: Szybkie (0, n-1, A, n)

Złożoność czasowa optymistyczna

Gdy w każdym kroku element x jest ”środkowym” elementem w sortowanej części (tzn. gdy następuje podział na części o tej samej liczbie elementów).

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Z twierdzenia o rekursji uniwersalnej otrzymujemy $T(n) = \Theta(n \log_2 n)$.

Złożoność czasowa pesymistyczna

Gdy x jest elementem największym lub najmniejszym w sortowanej części tablicy (tzn. gdy następuje podział w którym w jednej części tego podziału jest jeden element, a w drugiej części wszystkie pozostałe).

RÓWNANIE REKURENCYJNE LINIOWE: $T(n) = T(n-1) + \Theta(n)$

$T(n) = \Theta(n^2)$

Złożoność czasowa średnia

$A(n) = \frac{2}{log_2 e} n \log_2 n + 0(n) \Rightarrow A(n) = \Theta(n \log n)$

$\frac{2}{log_2 e} \approx 1,4$

PORÓWNANIE CZASU SORTOWANIA

n	Bąbelkowe	Scalanie
10^3	5ms	0,22ms
10^5	45s	30ms
10^6	74min	340ms
10^7	123h	4s

6. SORTOWANIE STOGOWE (kopcowe)

Niech A będzie tablicą n-elemetową (indeksowaną od 0).

Def. Mówimy, że tablica A ma własność stogu, gdy:

$$\forall\ 0 \leqslant i \leqslant \frac{n-2}{2} \quad A[i] \geqslant A[2i+1]$$

$$\forall\ 0 \leqslant i \leqslant \frac{n-3}{2} \quad A[i] \geqslant A[2i+2]$$

Element $A[2i+1]$ jest nazywany lewym synem a $A[2i+2]$ prawym synem elementu $A[i]$.

Spostrzeżenie. Element $A[0]$ (zwany korzeniem) jest elementem największym w tablicy będącej stogiem.

Załóżmy, że wszystkie elementy od A[l+1] do A[p] spełniają własność stogu (a element A[l] być może nie spełnia tej własności). Po wykonaniu funkcji przesiewania własność stogu będą spełniać elementy od A[l] do A[p].

```
Przesiewanie(int l, int p, int A[MAX])
 {int x, i, j;
   x=A[l];
   i=l;
   j=2*i+1;
   while(j<=p)
   { if (j<p && A[j]<A[j+1]) j++;
     if (x<A[j])
        { A[i]=A[j];
          i=j;
          j=2*i+1;
        }
      else break;
   }
   A[i]=x;
 }
```

```
BudujStog(int n, int A[MAX])
 {for(i=(n-2)/2; i>=0; i--)
  Przesiewanie(i,n-1,A);
 }
```

```
SortowanieStogowe(int n, int A[MAX])
 { BudujStog(n,A);
   for(i=n-1; i>=1; i--)
   { x=A[0];
     A[0]=A[i];
     A[i]=x;
    Przesiewanie(0,i-1);
   }
```

Asymptotyczna pesymistyczna złożoność czasowa

Liczba iteracji w funkcji Przesiewanie jest proporcjonalna do różnicy poziomów między elementami A[l] i A[p].
W najgorszym przypadku (gdy $l = 0$ oraz $p = n-1$) ta różnica poziomów to wysokość stogu h. Oczywiście,

$$1+2+\ldots+2^{h-1}+1 \leqslant n \leqslant 1+2+4+\ldots 2^h.$$

Stąd, $h = \theta(\log_2 n)$.

Wniosek. Przesiewanie ma złożoność pesymistyczną $\theta(\log_2 n)$.

```
BudujStog(int n, int A[MAX])
 {for(i=(n-2)/2; i>=0; i--)
  Przesiewanie(i,n-1,A);
 }
```

BudujStog: $\frac{n}{2} \cdot \theta(\log_2 n) = \theta(n \log_2 n)$

Uwaga. Można nawet pokazać, że BudujStog ma złożoność $\theta(n)$.

```
SortowanieStogowe(int n, int A[MAX])
 { BudujStog(n,A);
   for(i=n-1; i>=1; i--)
   { x=A[0];
     A[0]=A[i];
     A[i]=x;
    Przesiewanie(0,i-1);
   }
 }
```

SortowanieStogowe: $\theta(n \log_2 n) + n \cdot \theta(\log_2 n) = \theta(n \log_2 n)$

FAKT. Pesymistyczna (i średnia) złożoność czasowa sortowania stogowego jest równa $T(n) = \theta(n \cdot \log_2 n)$.

ZALETĄ sortowania stogowego jest to, że w przeciwieństwie do sortowania przez scalanie nie używamy żadnej dodatkowej tablicy.

Dolne ograniczenie złożoności czasowej problemu sortowania

Przyjmujemy, że operacją dominującą jest porównanie dwóch elementów tablicy i zakładamy, że wszystkie elementy tablicy są parami różne.

DRZEWO DECYZYJNE

To struktura przedstawiająca schemat porównań wykonywanych przez dany algorytm sortowania. Węzły wewnętrzne drzewa przedstawiają pojedyncze porównania elementów tablicy, zaś węzły końcowe (zwane również liśćmi) odpowiadają konkretnym permutacjom elementów. Wykonywanie algorytmu sortowania polega na przejściu od węzła początkowego (zwanego korzeniem) do jednego z węzłów końcowych.

Drzewo decyzyjne dla sortowania trzech elementów: a_1, a_2, a_3.

h - wysokość drzewa decyzyjnego to długość najdłuższej ścieżki od korzenia do najniżej położonego liścia.

Lemat 1.
Każde drzewo decyzyjne o wysokości h ma co najwyżej 2^h liści.

Dowód.

1° $h = 0$.
Drzewo ma dokładnie jeden węzeł (korzeń, który jest jednocześnie liściem). Czyli $2^0 = 1$. Ok.

2° Zakładamy, że twierdzenie jest prawdziwe dla h. Mamy udowodnić, że drzewo decyzyjne o wysokości $h+1$ ma co najwyżej 2^{h+1} liści.

Dow. Weźmy dowolne drzewo decyzyjne o wysokości $h+1$.
Wtedy liczba liści $\leqslant 2^h + 2^h = 2 \cdot 2^h = 2^{h+1}$. cbdu.

Lemat 2.
Każde drzewo decyzyjne sortujące n różnych elementów ma wysokość co najmniej $\log_2 n!$.

Dowód. Z lematu 1, liczba liści $n! \leqslant 2^h$.

Stąd $\log_2 n! \leqslant h$ czyli $h \geqslant \log_2 n!$. cbdu

Wzór Stirlinga:
$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

Stąd $\log_2 n! \geqslant \log_2(\frac{n}{e})^n = \log_2 n^n - \log_2 e^n = n\log_2 n - n\log_2 e$

TWIERDZENIE

- Pesymistyczna złożoność czasowa algorytmu sortującego n różnych elementów:
  $T(n) = \Omega(n\log_2 n)$.
- Średnia złożoność czasowa:
  $A(n) = \Omega(n\log_2 n)$

SORTOWANIE METODĄ ZLICZANIA

Jest to metoda dogodna do sortowania tablicy liczb całkowitych z ustalonego (niezbyt dużego) przedziału.
Główna idea algorytmu polega na wyznaczeniu dla każdego elementu tablicy ile jest w tablicy elementów nie większych od niego.

Zakładamy, że elementy tablicy mają wartości całkowite ze zbioru $\{0,1,2,\ldots,m-1\}$.

```
Sortowanie_przez_zliczanie(int A[MAX], int n)
{int C[mMAX], D[MAX];
 int i, j;
  for(i=0; i<m; i++) C[i]=0;

  for(j=0; j<n; j++) C[A[j]]++;

  for(i=1; i<m; i++) C[i]=C[i]+C[i-1];

  for(j=n-1; j>=0; j--)
   { C[A[j]]--;
     D[C[A[j]]]=A[j];
   }
  for(j=0;j<n;j++)
   A[j]=D[j];
}
```

Złożoność czasowa $\Theta(m+n)$

Gdy m jest mniejsze niż n (a nawet gdy $m = O(n)$), to otrzymujemy złożoność $\Theta(n)$, czyli złożoność liniową (od liczby sortowanych elementów).