LISTA nr. 1

z3]

```
procedure bubble (T[1...n])
    for i ← 1 to n-1 do
        for j ← 1 to n-1 do
            if T[j] > T[j+1] then
                swap (T[j], T[j+1])
```

Złożoność czasowa - zawsze rzędu n^2

Zarówno if jak i swap sprowadzają się do wykonania stałej liczby instrukcji maszyny RAM. Instrukcje wewnętrznej pętli wykonują się $\Theta(n^2)$ razy. Czyli całość $\Theta(n^2)$

- Rozkład danych

U przypadku bubble sort'a rozkład danych nie wpływa na czas jego działania. Zawsze wykonujemy $\Theta(n^2)$ operacji, zatem insert byłby lepszym wyborem.

- Wielkość rekordów

U najgorszym przypadku mamy $\Omega(n^2)$ swapów, co przy dużych rekordach jest kosztowne. Z tego względu select byłby lepszym wyborem.

- Stabilność

Mamy ostrą nierówność przy porównywaniu, więc rekordy o jednakowych kluczach pozostają w takim samym względnym porządku. Zatem insert i bubble spełniają ten warunek.

- Intensywność wykorzystania algorytmu

Różnią przerwać wykonywanie algorytmu jeśli po całym przejściu pętli wewnętrznej nie zostanie wykonana żadna podmiana.

Z4

Zauważmy, że w kolejnym kroku dzielenia liczby a przez 2 przesuwamy ją binarnie w lewo o 1.

Sprawdzenie czy a w danym kroku jest parzyste $\Leftrightarrow$ sprawdzenie czy ostatni bit jest zapalony.

Dodatkowo możemy zauważyć, że w kroku i mamy liczbę $b \cdot 2^i$.

Czyli nasz algorytm oblicza $\sum_{i=1}^{\log_2 a} 2^i \cdot m \cdot k_i$, gdzie k_i to i-ty bit liczby a.

$$\Updownarrow$$

$$m \sum_{i=1}^{\log_2 a} 2^i \cdot k_i = m \cdot n$$

- Jednorodne kryterium:

  W każdym kroku algorytmu wykonujemy stałą liczbę operacji jednostkowych, a kroków wykonamy $\log_2 a$.
  Zatem złożoność to $O(\log_2 a)$.

- Logarytmiczne kryterium:

  W najgorszym wypadku w danym kroku algorytmu będziemy operować na liczbie o $\log_2 a \cdot b$ bitach, czyli
  złożoność algorytmu to $O(\log_2 a \cdot \log_2 a \cdot b)$.

Z5)

Punkt pierwszy:

$$\begin{bmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & 0 & 1 & \\ & & & \ddots & 1 \\ m_1 & m_2 & \dots & & m_n \end{bmatrix} \begin{bmatrix} f_i \\ \vdots \\ f_{i+n} \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ \vdots \\ f_{i+n+1} \end{bmatrix}$$

gdzie $f_{i+n+1} = m_1 \cdot f_i + m_2 \cdot f_{i+1} + \dots + m_n \cdot f_{i+n}$

Punkt drugi:

$$\begin{bmatrix} 0 & 1 & & & & & & \\ & 0 & 1 & & & & & \\ & & 0 & 1 & & & & \\ & & & 0 & 1 & & & \\ & & & & \ddots & 1 & & \\ m_0 & m_1 & \dots & & m_n & a_0 & \dots & a_k \\ & & & & & 1 & & \\ & & & & & 1 & 1 & \\ & & & & & 1 & 2 & 1 \\ & & & & & 1 & 3 & 3 & 1 \\ & & & & & \vdots & & & \ddots \\ & & & & & 1 & \binom{k}{1} & \binom{k}{2} & \dots & \binom{k}{k-1} & 1 \end{bmatrix} \begin{bmatrix} f_i \\ f_{i+1} \\ \vdots \\ f_{i+n-1} \\ 1 \\ n \\ n^2 \\ \vdots \\ n^k \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ f_{i+2} \\ \vdots \\ f_{i+n} \\ 1 \\ n+1 \\ (n+1)^2 \\ \vdots \\ (n+1)^k \end{bmatrix}$$

gdzie $f_{i+n} = m_0 \cdot f_i + m_1 \cdot f_{i+1} + \dots + m_n \cdot f_{i+n-1} + a_0 + a_1 \cdot n + \dots + a_k \cdot n^k$

- dowolen Newtona - $(x+y)^n = \binom{n}{0} x^n + \binom{n}{1} \cdot x^{n-1} y + \dots + \binom{n}{n} \cdot y^n$

26

Zauważmy, że algorytm ten wrzuca liczbę $x \% 2$, czyli interesuje nas tylko ostatni bit tej liczby.

Dodatkowo zauważmy, że $a - b \equiv_2 a \bmod 2 - b \bmod 2$

a to z kolei $\equiv_2 a \text{ xor } b$.

a	b	a xor b	a mod 2 − b mod 2
1	1	0	0
1	0	1	1
0	1	1	1
0	0	0	0

Wiemy, że xor jest operacją łączną oraz przemienną, zatem kolejność wykonania elementów nie ma znaczenia

Przykładowy wynik działania:

$x = (a_1 \text{ xor } a_2) \text{ xor } a_3) \text{ xor } a_4) \bmod 2$

Algorytm:

```
wynik ← 0
for i ← 1 to |A| do
    'wczytaj a
    wynik ← wynik ^ a
output (wynik mod 2)
```

Z7)

Zamieniamy nasz graf na listę sąsiedztwa.

Preprocessing: $O(n)$

```
czas_wejscia ← [0]*n
czas_wyjscia ← [0]*n
visited ← [0]*n
time ← 0
def dfs(v):
    visited[v] = 1
    czas_wejscia[v] = time
    time += 1
    for w in G[v]:
        if visited[w] == 0:
            dfs(w)
    czas_wyjscia[v] = time
    time += 1

dfs(korzeń)
```

Zapytania: $O(1)$

```
if czas_wejscia[ui] > czas_wejscia[vi] && czas_wejscia[ui] < czas_wyjscia[vi]
    return true
return false
```