Lista 0
3.

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

- Rozkład danych: BubbleSort za każdym razem wykonuje kwadrat operacji ($\Theta(n^2)$ porównań), zatem rozłożenie danych nie ma znaczenia. Lepiej wybrać Insert.

- Złożoność czasowa: zawsze rzędu n^2
  Instrukcje pętli wewnętrznej wykonują się zawsze $\Theta(n^2)$ razy.
  SWAP oraz IF zawsze wykonują stałą liczbę instrukcji.
  Każda z ich iteracji to stałe liczby instrukcji - $\Theta(n^2)$.

- Wielkość rekordów:
  w najgorszym przypadku wykonamy $\Omega(n^2)$ zamian,
  w tym wypadku SELECT byłby lepszym wyborem

- Stabilność:
  ten rodzaj sortowania jest stabilny (ostre nierówności)

- Intensywność wykorzystania:
  - dodanie flagi, która w przypadku posortowanej tablicy wychodzi z „fora" (gdy nie będzie żadnych „swapów")
  - w wewnętrznym forze dodanie odjęcie od warunku stopu j: $n-1-j$

31 32
15 16
7 8
3 4
1 2
1

4.

Weźmy dowolne liczby a i $b \in \mathbb{R}$

$a_1, a_2, \dots, a_{m+1} \rightarrow a_1 = a,\ a_{m+1} = 1,\ a_{i+1} = \left\lfloor \frac{a_i}{2} \right\rfloor$, $b_1, b_2, \dots, b_{m+1} \rightarrow b_1 = b,\ b_{i+1} = 2b_i$

Dzieląc a całkowicie przez 2 i patrząc na jej reszty otrzymamy binarny zapis tej liczby.

$$a = x_0 2^0 + x_1 2^1 + \dots + x_m 2^m \quad , \quad x_i \in \{0, 1\}$$

gdzie $x_0 = 1 \Leftrightarrow a_1 \equiv_2 1$, $x_1 = 1 \Leftrightarrow a_2 \equiv_2 1 \dots$

$x_0 = 0 \Leftrightarrow a_1 \equiv_2 0$, $x_1 = 0 \Leftrightarrow a_2 \equiv_2 0 \dots$

czyli bit i-ty jest „zapalony" gdy $a_i \equiv_2 1$

(wtedy wg. algorytmu powinniśmy dodać b_i)

$$a \cdot b = x_0 \cdot 2^0 \cdot b + x_1 2^1 \cdot b + \dots + x_m 2^m \cdot b$$

algorytm dodaje $\sum_{\substack{i=1 \\ a_i \text{ nieparzyste}}}^{m+1} b_i = \sum_{i=1}^{m+1} x_{i-1} b_i = \sum_{i=1}^{m+1} x_{i-1} \cdot 2^{i-1} b_1 = x_0 \cdot 2^0 \cdot b + x_1 2^1 \cdot b + \dots =$

$= a \cdot b$, zatem algorytm jest poprawny.

- jednorodne kryterium kosztów
- Pętla w algorytmie wykonuje się $O(\log_2 a)$ razy, ponieważ w niej dzielimy całk. a, mnożymy b oraz wykonujemy możliwe dodawanie
- Podczas pętli wystarczy nam pamiętać tylko a, b, wynik, także zł. pamięciowa stała - $O(1)$

- kryterium logarytmiczne
- wykonanie pętli - $O(\log_2 a)$, wykonanie operacji (dodawanie) w najgorszym przypadku to $(\log_2 a \cdot b)$. Zatem złożoność to $O(\log_2 a \cdot \log_2 a \cdot b)$
- wynik zajmować może $\max \lfloor \log_2 a \cdot b \rfloor + 1$ bity, więc $O(\log_2 ab)$

5.

a) wystarczy zmienić macierz przekształcenia

Weźmy dowolny ciąg f t. że $f_1, \dots, f_n$ ustalone, $f_{n+1} = \sum_{i=n+1-n}^{n} a_i \cdot f_i$

$$\begin{bmatrix} 0 & 1 & & & 0 \\ & 0 & 1 & & \\ & & 0 & 1 & \\ & & & \ddots & 0 \\ & & & & 1 \\ 0 & & & & \\ a_1 & a_2 & \dots & & a_n \end{bmatrix} \cdot \begin{bmatrix} f_i \\ f_{i+1} \\ \\ \\ f_{i+n-1} \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ f_{i+2} \\ \\ \\ f_{i+n} \end{bmatrix}$$

w oczywisty sposób ta równość zachodzi (ostatni wiersz po wymnożeniu z wektorem jest komb. liniową n el. wcześniejszych)

Wystarczy podnieść macierz do odp. potęgi

b) musimy dodatkowo zawrzeć w macierzy inkrementację wielomianu

$$(*)\quad \begin{bmatrix} 0 & 1 & & & & 0 & 0 & \dots & 0 \\ & 0 & 1 & & & & 0 & & 0 \\ & & & 1 & & & & & \\ & & & & \ddots & & & & \\ & & & & & 0 & 0 & & \\ & & & & & 1 & 0 & & 0 \\ a_1 & a_2 & \dots & & & a_n & x_0 & \dots & x_k \\ 0 & - & - & - & - & 0 & 1 & 0 & 0 \\ & & & & & & 1 & 1 & 0 \\ & & & & & & 1 & 2 & 1 \\ & & & & & & 1 & 3 & 3 \\ & & & & & & \vdots & & \\ 0 & - & - & - & - & 0 & 1 & \binom{k}{1} & \binom{k}{2} \end{bmatrix} \cdot \begin{bmatrix} f_i \\ f_{i+1} \\ \\ \\ f_{i+n-1} \\ 1 \\ n \\ n^2 \\ n^3 \\ \\ n^k \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ f_{i+2} \\ \\ \\ f_{i+n} \\ 1 \\ n+1 \\ (n+1)^2 \\ \\ (n+1)^k \end{bmatrix}$$

w wektorze wynikowym dostajemy ciąg przesunięty o 1, wiersz (*) po wymnożeniu jest sumą komb. liniowej n elementów oraz wielomianu stopnia k.

Zaznaczony kwadrat jest trójkątem Pascala. Zera przed nim nie wpływają na wartości ciągu, a jego kolejne wiersze są współczynnikami z dwumianu Newtona. $(n+1)^k = \sum_{i=0}^{k} \binom{k}{i} n^i$ ← wymnożenie współczynniki przez potęgi n-ów, po czym zwijamy do postaci $(n+1)^k$.

6. W algorytmie w zadaniu interesuje nas tylko parzystość (końcowy bit) ostatniego elementu.

Zauważmy, że operacje wykonywane w pętli:

$a - b \equiv_2 a \bmod 2 - b \bmod 2$, ponieważ interesuje nas tylko końcowy bit ($-1 \equiv_2 1$) to wszystkie odejmowanie możemy zamienić na $|a \bmod 2 - b \bmod 2| \equiv a \text{ xor } b$ #

Możemy stąd wywnioskować, że cały algorytm sprowadza się do obliczenia xor'a wszystkich liczb (różnie ponumerowanych). Ponieważ xor jest łączny i przemienny to końcowy wynik będzie postaci $a_0 \text{ xor } a_1 \text{ xor } \dots \text{ xor } a_m$

PSEUDOKOD: (złożoność pamięciowa stała $O(1)$)

```
wynik ← 0
for i ← 0 to |A| do:
    wczytaj a
    a %= 2          ← możemy też wziąć jednego % na końcu
    wynik xor= a
output(wynik)
```

7.

Ponieważ liczba zapytań może być długa warto zrobić preprocesing

//zamiana listy krawędzi na listę sąsiedztwa:

```
G ← lista sąsiedztwa
for k in lista_krawędzi:
    if k[0] not in G:
        G[k[0]] ← lista (słownik)
    G[k[0]][k[1]] = 1
czas = 0
czas_wejścia ← [0]*n
czas_wyjścia ← [0]*n
```

```
odwiedzony ← [0]*
Procedure DFS ( wierzchołek)
    czas_wejścia ← [wierzchołek] = czas
    odwiedzony [wierzchołek] = 1
    czas +=1
    for sąsiad in graf[wierzchołek]:
        if not odwiedzony[sąsiad]
            DFS (sąsiad)
    czas_wyjścia [wierzchołek] = czas
    czas++
DFS(korzeń)
Procedure is_good (v, u)
    if czas_wejścia[u] < czas_wyjścia[v] and
        czas_wejścia[u] > czas_wejścia[v]
            output (true)
    output(false)
```

Ponieważ DFS działa jednowątkowo i odwiedza wszystkie wierzchołki w grafie - czas wejścia ojca będzie zawsze mniejszy niż czas wejścia syna oraz czas wyjścia ojca będzie zawsze mniejszy niż czas wyjścia ojca będzie większy niż wejście/wyjścia syna. Stąd is_good zwróci prawdę jeśli v_i leży na ścieżce do korzenia z u_i.