Zad. 3

```
Procedure bubble(T[1...n])
    for i ← n to 2 do
        for j ← 1 to i-1
            if T[j] > T[j+1]
                temp ← T[j]
                T[j] ← T[j+1]
                T[j+1] ← temp
```

Idea algorytmu: przechodzimy po tablicy T przepychając największy element na koniec tablicy. W k-tej iteracji zewnętrznej pętli na swoim miejscu są elementy $T[n], T[n-1], \ldots T[n-k+1]$.

Złożoność: zawsze n^2. Instrukcje wewnętrznej pętli wykonują się $\Theta(n^2)$ razy i wykorzystują stałą liczbę operacji maszyny RAM.

Algorytm jest stabilny. Jednakże wykonuje bardzo wiele porównań i potencjalnie bardzo wiele zamian wartości zmiennych.

Zad. 4

```
Procedure russian_mult (A, B)
    result ← 0
    while A > 0
        if A % 2 = 1
            result ← result + B
        B ← B · 2
        A ← ⌊A/2⌋
```

Spójrzmy na binarny zapis A oraz B.

```
    1100101
  ·   11010
  ---------
    1100101
   1100101
  0000000
 1100101
---------
1010000010
```

Zauważamy, że w algorytmie z treści notatki a_i niepamiętamy iff i-ty bit w A jest zapalony. Ponadto

$$A \cdot B = \sum_{\substack{i \text{ t.że} \\ i\text{-ty bit w } A \\ \text{zapalony}}} B \cdot 2^{i-1} =$$

$$= \sum_{i : a_i \text{ niepamiętamy}} B \cdot 2^{i-1} =$$

$$= \sum_{i : a_i \text{ niep.}} B_i$$

Zatem algorytm realizuje pisemne mnożenie liczb w jedn. kryt. kosztów w zapisie binarnym. Jego złożoność to $\log A$, pamięciowe $O(1)$

W logarytmicznym kryterium
złożoności czasowa wymosi $\log A \cdot \log AB$
złożoności pamięciowa to $\log A + \log B$.

Zad. 5

Mamy dany ciąg rekurencyjny wzorem:

$$\begin{cases} a_1 = c_1 \\ a_2 = c_2 \\ \vdots \\ a_k = c_k \\ a_{n+1} = b_k a_n + b_{k-1} a_{n-1} + \dots + b_1 a_{n-k+1}, & n \geq k \end{cases}$$

Gdzie c_i, b_i to pewne stałe.

Wtedy

$$\begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & b_3 & b_4 & \dots & b_k \end{bmatrix} \begin{bmatrix} a_n \\ a_{n+1} \\ a_{n+2} \\ \vdots \\ a_{n+k-1} \end{bmatrix} = \begin{bmatrix} a_{n+1} \\ a_{n+2} \\ a_{n+3} \\ \vdots \\ a_{n+k} \end{bmatrix}$$

Macierz M

Zatem jeśli weźmiemy $V_1 = [c_1, \dots, c_k]^T$, to pierwsza wartość w wektorze $M^{n-1} V_1$ określa a_n.

Kiedy rekurencyjna zależność wyraża się wzorem

$$a_{n+1} = b_k a_n + \dots + b_1 a_{n-k+1} + w(n),$$

gdzie $w(n) = w_0 + w_1 n + \dots + w_l n^l$,

to algorytm możemy przeprowadzić w analogiczny sposób.

$$\begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 \\ b_1 & b_2 & b_3 & \dots & b_k & w_0 & w_1 & \dots & w_l \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 0 & \binom{l}{0} & \binom{l}{1} & \dots & \binom{l}{l} \end{bmatrix} \begin{bmatrix} a_n \\ a_{n+1} \\ \vdots \\ a_{n+k-2} \\ a_{n+k-1} \\ 1 \\ n+k \\ \vdots \\ (n+k)^l \end{bmatrix} = \begin{bmatrix} a_{n+1} \\ a_{n+2} \\ \vdots \\ a_{n+k-1} \\ a_{n+k} \\ 1 \\ n+k+1 \\ \vdots \\ (n+k+1)^l \end{bmatrix}$$

Obliczamie kolejnych wyrazów $a_n, \dots, a_{n+k}$

Obliczamie $1, n+k+1, \dots, (n+k+1)^l$

Złożoność pierwszego algorytmu to $O(\log n \cdot k^\omega)$, drugiego $O(\log n \cdot (k+1)^\omega)$.
(z wykorzystaniem algorytmu szybkiego potęgowania)

Zad. 6

Pokażemy za pomocą indukcji matematycznej, że wynikiem procedury jest $(a_1 + \dots + a_n) \bmod 2$.

Dla $n=1$ teza oczywiście zachodzi. Załóżmy, że teza zachodzi dla $1, \dots, n$. Pokażemy jej prawdziwość dla $n+1$.

Niech $A = \{a_1, \dots, a_{n+1}\}$.

BSO w pierwszym kroku wybrałiśmy liczby a_n oraz a_{n+1}. Wtedy liczby te wypadają ze zbioru i trafia tam liczba $(a_n - a_{n+1})$, czyli $f' = \{a_1, \dots, a_{n-1}, a_n - a_{n+1}\}$.

Z założenia indukcyjnego wiemy, że wynikiem wykonania procedury na zbiorze A' jest $(a_1 + \ldots + a_{n-2} + (a_n - a_{n+1})) \% 2$

$= (a_1 + \ldots + a_{n-1} + a_n + a_{n+1}) \% 2$. Ponadto wynik ten jest równy wynikowi wykonania naszej wyjściowej procedury, co dowodzi tezę. ∎

Korzystając z naszego twierdzenia z łatwością prezentujemy następujący równoważny algorytm:

```
result ← 0
for a in A
    result ← result + a
return (result % 2)
```

Zad. 7

Przy pomocy listy krawędzi konstruujemy tablicę par[1...n], gdzie par[u] = v ⇔ (v,u) ∈ E (czyli u jest ojcem v).

Ponadto zakł. że 1 jest korzeniem, par[1] = 1.

```
Procedure is_ancestor (u , v)
    if u = 1 or par[v] = u
        return true
    return is_ancestor (u, par[v])
```

(u jest przodkiem v ⇔ u jest ojcem v lub u jest przodkiem ojca v).

Złożoność takiego algorytmu to $O(n \cdot m)$. Da się to zrobić lepiej.

Niech $G[1 \dots n]$ będzie listową reprezentacją grafu (przy czym rozkładem otoczenie krawędzi od ojców do synów). Definiujemy procedurę obliczania tablic $pre[1 \dots n]$ oraz $post[1 \dots n]$:

```
Procedure compute_prepost ( G[1 ... n] ):
    cnt ← 0

    Procedure DFS (v):
        pre [v] ← cnt
        cnt ← cnt + 1
        for u in G[v]
            DFS(u)
        post [u] ← cnt
    DFS(1)
```

Zauważmy, że u jest przodkiem v
$\Leftrightarrow$ v jest w poddrzewie u.

Nietrudno zauważyć, że

v jest w poddrzewie u $\Leftrightarrow$

$pre[v] \geq pre[u]$ oraz $post[v] \leq pre[u]$.

Wtedy procedura is_ancestor

znacząco się upraszcza:

Procedura is_ancestor (u, v):

return $pre[v] \geq pre[u]$ and $post[v] \leq post[u]$

Złożoność czasowa: $O(n)$

Złożoność pamięciowa: $O(n)$

(dla obu algorytmów).