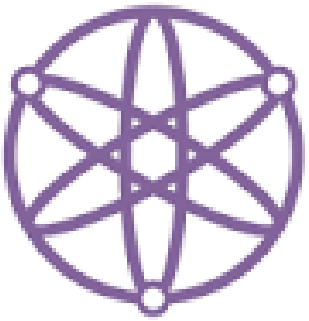
1.5em 0pt



WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Sprawozdanie z zadania programistycznego

Imię i nazwisko: Jakub Półtorak (nr indeksu: 179965)
Grupa projektowa: 6
Prowadzący: Dr.inż.Mariusz Borkowski, prof PRz

Data: Styczeń 2025

Spis treści

1 Treść zadania

Dla zadanej tablicy liczb całkowitych przesuń wszystkie elementy mniejsze od 0 na jej koniec (należy zachować kolejność występowania).

Przykład:

Wejście: A[]=$[-10, 5, 8, -4, 1, 3, 0, -7]$
Wyjscie: $[5, 8, 1, 3, 0, -10, -4 - 7]$

2 Rozwiązanie problemu

2.1 Sposób 1 - metoda brute force

2.1.1 Analiza problemu

W pierwszej metodzie rozwiązania postępuję „siłowo" implementując rozwiązanie najbardziej logiczne, które wprost wynika z treści zadania. Polega ono na wykonywaniu operacji przesuwania wartości w tablicy tak, aby na końcu umieszczać znalezione liczby ujemne. Na początku więc wyznaczam ilość liczb ujemnych w zbiorze. Jeżeli wszystkie liczby są ujemne, lub dodatnie, wynikiem działania programu jest tablica bez zmian. W przeciwnym razie sprawdzam, czy więcej liczby ujemne stanowią mniej niż połowę tablicy. Jeśli tak, przeglądam tablicę aż do napotkania liczby ujemnej. Kiedy spotykam ujemną wartość, zapisuję wartość ostatniego indeksu w zmiennej pomocniczej, na ostatnim indeksie umieszczam znalezioną wartość. Później zapamiętuję wartość na przedostatnim indeksie i wprowadzam tam dotychczasową wartość zmiennej pomocniczej, zaś zmiennej pomocniczej przypisuję zapamiętaną wartość. Analogicznie postępuje do pozycji, w której znajdowała się wcześniej liczba ujemna (nie przesuwam liczb znajdujących się przed tą pozycją). Po wykonaniu zamiany na wcześniejszej pozycji liczby ujemnej, zwiększam licznik przesunięć o 1. Czynności te powtarzam dopóki licznik przesunięć jest mniejszy od ilości liczb ujemnych. Jest to swego rodzaju zabezpieczenie przed zbyt wieloma operacjami przesunięcia. Jeżeli w tablicy jest mniej liczb większych lub równych 0 niż ujemnych wykonuje przesuwanie liczb dodatnich na początek tablicy analogicznie jak w przypadku liczb ujemnych. Tablica powstała po wykonaniu powyższych operacji**jest wynikiem działania programu**.

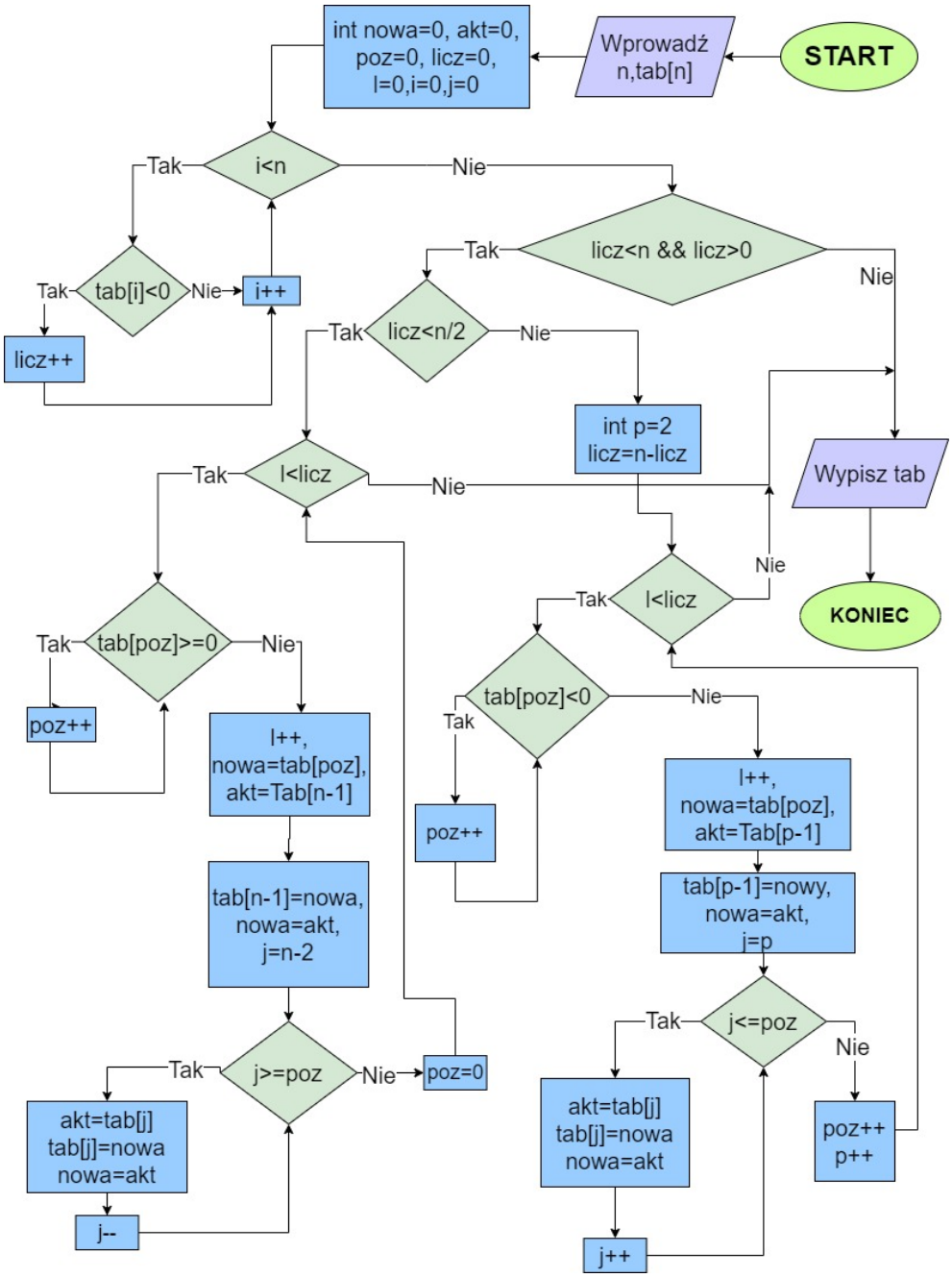
1. **Dane wejściowe:** Tablica liczb całkowitych
2. **Wynik:** Tablica liczb całkowitych z liczbami ujemnymi umieszczonymi na końcu zgodnie z ich występowaniem.

2.1.2 Schemat blokowy algorytmu

Na następnej stronie przedstawiono implementację algorytmu w postaci schamtu blokowego:
W schemacie blokowym występują zmienne pomocnicze, które umożliwiają zamianę pozycjami liczb. Są to:

- akt - wartość na indeksie przed zamianą
- nowa - nowa wartość indeksu tablicy

Należy je oczywiście opowiednio zdefiniować.



Rysunek 1: Schemat blokowy algorytmu 1

2.1.3 Zapis algorytmu w pseudokodzie

Wejście: tab[n], n

Algorithm 1 Pseudokod alogrytmu 1

```
nowa ← 0
akt ← 0
poz ← 0
licz ← 0                                   ▷ Licznik liczb ujemnych
l ← 0                                      ▷ Licznik przesuniętych liczb ujemnych
for i ← 0 to n − 1 do
    if tab[i] < 0 then licz ← licz + 1
    end if
end for
if licz ≠ n AND licz ≠ 0 then
    if licz < n/2 then
        while l < licz do
            while tab[poz] ⩾ 0 do poz ← poz + 1
            end while
            l ← l + 1
            nowa ← tab[poz]
            akt ← tab[n − 1]
            tab[n − 1] ← nowa
            nowa ← akt
            for j = {n − 2, n − 3, n − 4, ..., poz} do
                akt ← tab[j]
                tab[j] ← nowa
                nowa ← akt
            end for
            poz ← 0
        end while
    else
        p ← 1
        licz ← n − licz
        while l < licz do
            while tab[poz] < 0 do poz ← poz + 1
            end while
            l ← l + 1
            nowa ← tab[poz]
            akt ← tab[p − 1]
            tab[p − 1] ← nowa
            nowa ← akt
            for j = {p, p + 1, p + 2, ..., poz} do
                akt ← tab[j]
                tab[j] ← nowa
                nowa ← akt
            end for
            poz ← poz + 1
            p ← p + 1
        end while
    end if
    Wypisz tab
```

2.1.4 Teoretyczne szacowanie złożoności obliczeniowej

Główną operacją w tym przypadku jest zamiana miejscami liczb w tablicy. Złożoność obliczeniowa tego algorytmu będzie w dużej mierze zależeć od rozmieszczenia liczb w zbiorze.

Na początku zakładamy, że liczb ujemnych jest mniej niż dodatnich. Przyjmijmy, że na indeksie i tablicy o rozmiarze n występuje liczba ujemna. Zauważamy fakt, że przesuwamy liczby występujące tylko po napotkanej liczbie ujemnej. Jest to więc $n - i$ operacji. Zależność ta będzie jednak spełniona tylko wtedy, kiedy będzie to pierwsza w kolejności liczba ujemna. Dla następnych w kolejności liczb ujemnych musimy uwzględnić ich kolejność w tablicy, gdyż z każdym przesunięciem wcześniej napotkanych liczb ujemnych, rozpatrywana liczba przesuwa się w lewo o 1, zatem jej pozycja maleje o 1.

Uwzględniając powyższą analizę, liczba operacji przesunięcia dla liczby znajdującej się na indeksie i wynosi:

$$L_k = (n - i_k + (k - 1))$$

gdzie:
i - indeks liczby w tablicy początkowej
k - kolejność liczby w tablicy pierwotnej

Reasumując, dla tablicy o n elementach i o k liczbach ujemnych liczba operacji wynosić będzie:

$$L = L_1 + L_2 + L_3 + ... + L_k = (n - i_1) + (n - i_2 + 1) + (n - i_3 + 2) + ... + (n - i_k + (k - 1))$$

Po przekształceniach:

$$L = n \cdot k - (i_1 + i_2 + i_3 + ... + i_k) + \frac{1}{2} \cdot (k^2 - k)$$

gdzie:
n - rozmiar tablicy
k - ilość liczb ujemnych
i_n - indeks n-tej liczby ujemnej w tablicy

W przypadku wykonywania tych operacji na liczbach dodatnich, złożoność będzie prezentować się nieco inaczej. Przesuwamy wtedy liczby dodatnie na początek tablicy. Z każdym krokiem indeks na którym umieszczana jest liczba rośnie o 1. Zatem liczba operacji dla liczby dodatniej na k-tym indeksie wynosić będzie:

$$L = i_k - (k - 1)$$

gdzie: i_k - pozycja w tablicy k-tej liczby dodatniej,
k - kolejność liczby wśród wszystkich liczb ujemnych w zbiorze początkowym

Zatem dla tablicy o k elementach dodatnich i rozmiarze n, gdzie $k \leqslant n - k$, wzór na złożoność obliczeniową prezentuje się następująco:

$$L_c = L_1 + L_2 + ... + L_k = (i_1 - 1) + (i_2 - 2) + ... + (i_k - k)$$

2.2 Rozwiązanie 2 - wykorzystanie tablicy pomocniczej

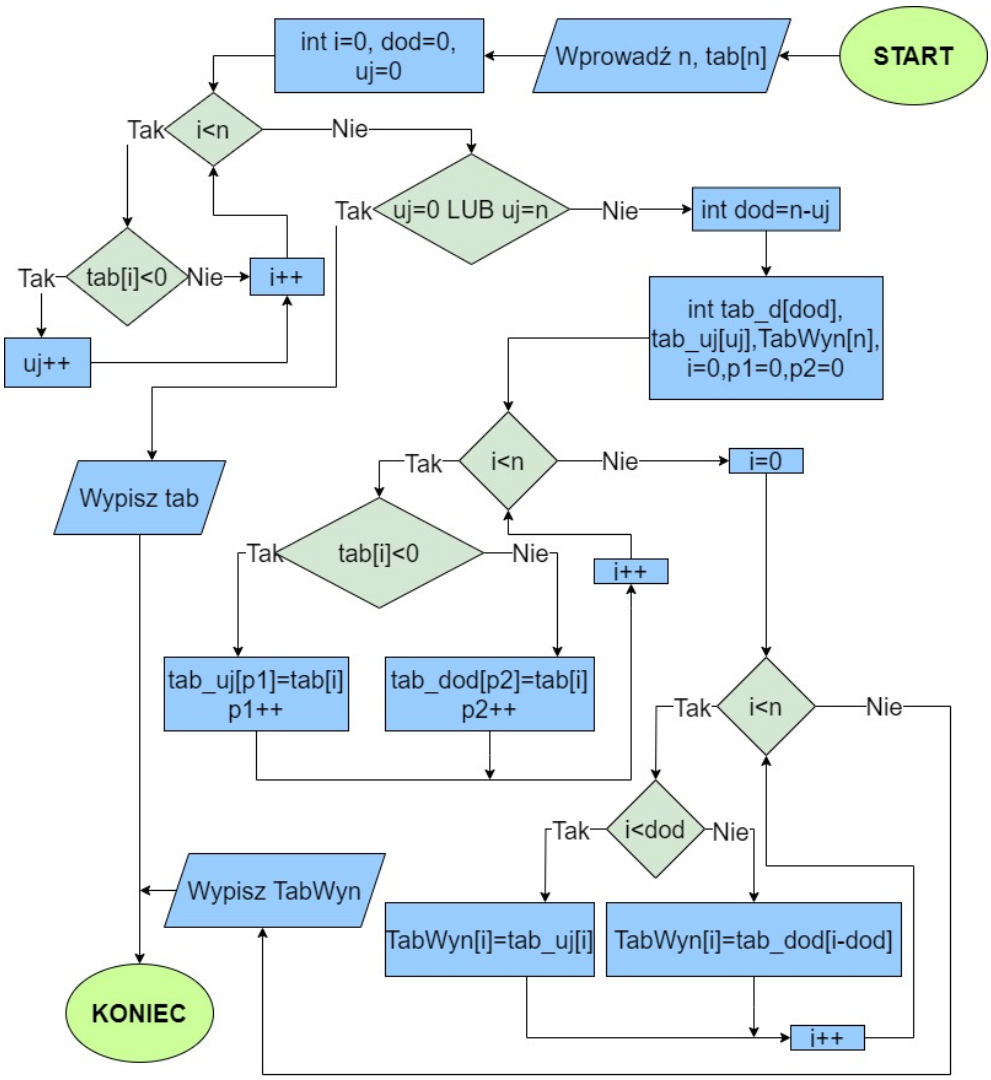
2.2.1 Analiza problemu

W metodzie tej wykorzystamy pomocniczą tablicę, do której wprowadzimy liczby dodatnie i kolejną, do której wprowadzimy liczy ujemne. Następnie przepisujemy je do tablicy wynikowej, najpierw liczby dodatnie, poźniej ujemne. Tablica, do której wprowadzamy te wartości jest **wynikiem działania programu**.

1. **Dane wejściowe:** Tablica liczb całkowitych
2. **Wynik:** Tablica liczb całkowitych z liczbami ujemnymi umieszczonymi na końcu zgodnie z ich występowaniem.

2.2.2 Schemat blokowy algorytmu

Wyżej opisany alogorytm postępowania w postaci schematu blokowego:



Rysunek 2: Schemat blokowy algorytmu 2

Z schematu blokowego wynika, że należy zdefiniować dwie tablice pomocnicze na liczby ujemne i dodatnie. Następnie zgodnie z kolejnością przypisać je do nowej tablicy, którą poźniej należy wypisać.

2.2.3 Zapis algorytmu w pseudokodzie

Dane: tab[n] - tablica liczb całkowitych, n - rozmiar tablicy
Wynik: TabWyn[n] - tablica końcowa

Algorithm 2 Pseudokod alogrytmu 2

```
1: procedure PRZESUN(tab, n)
2:     uj ← 0                                          ▷ Licznik elementów ujemnych
3:     dod ← 0                                         ▷ Licznik elementów dodatnich
4:     for i ← 0 to n − 1 do
5:         if tablica[i] < 0 then
6:             uj ← uj + 1
7:         end if
8:     end for
9:     if uj = n OR uj = 0 then
10:        WYPISZ(tab)
11:    else
12:        dod ← n − uj                                ▷ Obliczanie liczby elementów dodatnich
13:        tab_d[dod] ← 0
14:        tab_uj[uj] ← 0
15:        TabWyn[n] ← 0
16:        p1 ← 0                                      ▷ Indeks dla tablicy ujemnych
17:        p2 ← 0                                      ▷ Indeks dla tablicy dodatnich
18:        for i ← 0 to n − 1 do
19:            if tablica[i] < 0 then
20:                tab_uj[p1] ← tab[i]
21:                p1 ← p1 + 1
22:            else
23:                tab_dod[p2] ← tab[i]
24:                p2 ← p2 + 1
25:            end if
26:        end for
27:        for i ← 0 to n − 1 do
28:            if i < dod then
29:                TabWyn[i] ← tab_dod[i]
30:            else
31:                TabWyn[i] ← tab_d[i − dod]
32:            end if
33:        end for
34:        WYPISZ(TabWyn)
35:    end if
36: end procedure
```

2.2.4 Złożoność obliczeniowa

W przypadku tego alogorytmu główną operacją będzie przypisanie elementu na indeks w tablicy. Gdy w tablicy występuje n liczb, następuje n przypisań do tablicy z liczbami dodatnimi, bądź ujemnymi. Później następuje przepisanie elementów z kolejno tablicy liczb dodatnich i ujemnych do nowej tablicy wynikowej. Dochodzi do tego również n razy. Złożoność obliczeniowa tego algorytmu dla n-elementowej tablicy wyraża się wzorem:

$$L = 2n$$

Wniosek: Złożoność obliczeniowa tego algorytmu **zawsze wyraża się takim samym wzorem**.

3 Porównianie sposobów rozwiązania problemu

3.1 Wstęp

Już na etapie teoretycznej analizy złożoności obliczeniowej zauważamy, że choć efekt działania jest ten sam, między algorytmami występują istotne różnice. Porównamy więc alogrytmy na podstawie:

- Złożoności obliczeniowej - ilości operacji wykonanych przez programy.
- Złożoności czasowej - zmiany czasu wykonywania danych w zależności od rozmiaru danych
- Złożoności pamięciowej - ilości pamięci rezerwowanej przez alogrytm

Jako że na etapie analizy teoretycznej złożoności obliczeniowej w pierwszym alogrytmie zauważono zależność złożoności obliczeniowej od rozmieszczenia danych, złożoności rozptrywać będziemy w kilku przypadkach:

1. **Przypadek 1:** Występuje tylko jedna liczba ujemna pod koniec (na początku) tablicy.
2. **Przypadek 2:** Pierwsza połowa tablicy to liczby ujemne, a druga to liczby nieujemne.
3. **Przypadek 3:** Występuje tylko jedna liczba dodatnia na początku lub końcu tablicy.

Uwaga: dla alogrytmu 2 złożoność obliczeniowa będzie taka sama, **niezależnie od rozmieszczenia danych**.

3.2 Złożoność obliczeniowa programu 1

3.2.1 Przypadek 1

Do testów wykorzystam tablicę liczb pseudolosowych wypełnioną liczbami dodatnimi poza trzecią pozycją, gdzie będzie liczba ujemna. Sprawdźmy, ile operacji zostanie wykonane w zależności od rozmiaru tablicy.

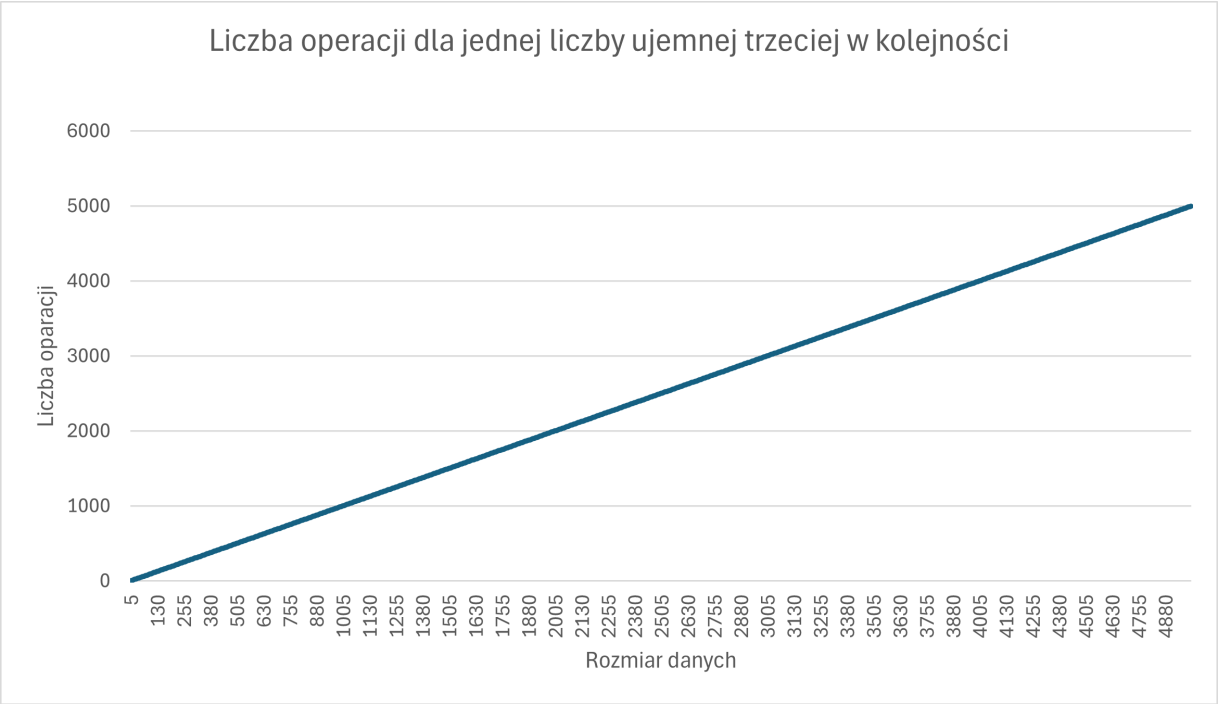
Rozmiar tablicy	Liczba operacji
5	3
10	7
15	12
30	27
50	47
100	97
200	197
1000	997
10000	9997

Tabela 1: Złożność obliczeniowa programu 1 w przypadku 1(liczba ujemna na początku)

Łatwo zauważyć, że liczba operacji jest równa rozmiarowi tablicy pomnejszonemu o wartość indeksu tablicy (przy założeniu że liczymy od 0).
Dla algorytmu 1 złożoność obliczeniowa wynosi więc:

$$O_n = n - p$$

Gdzie:
n - rozmiar tablicy
p - położenie liczby w tablicy.



Rysunek 3: Wykres złożoności obliczeniowej programu 1 w 1 przypadku(liczba ujemna na 3 pozycji)

Sprawdźmy jeszcze, jak kształtuje się złożoność obliczeniowa dla przypadku, gdy jedyna liczba ujemna znajduje się pod koniec tablicy (jest na przykład trzecia od końca).

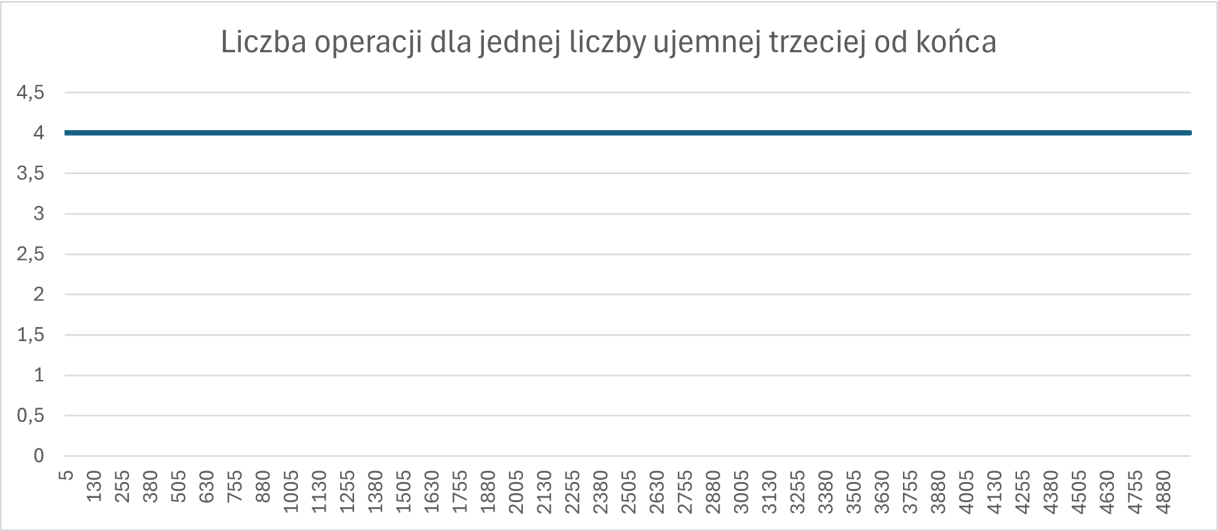
Rozmiar tablicy	Liczba operacji
5	4
10	4
15	4
30	4
50	4
100	4
200	4
1000	4
10000	4

Tabela 2: Złożność obliczeniowa programu 1 w przypadku 1 (liczba ujemna pod koniec tablicy)

W tym przypadku zauważalne jest, że złożoność obliczeniowa nie jest zależna od rozmiaru tablicy. Złożoność obliczeniowa jest równa odległości od końca tablicy powiększonej o 1. Jest ona więc dana wzorem:

$$O = i + 1$$

gdzie i to **odległość liczby od końca tablicy**.



Rysunek 4: Wykres złożoności obliczeniowej programu 1 w 1 przypadku (liczba ujemna na 3 pozycji od końca)

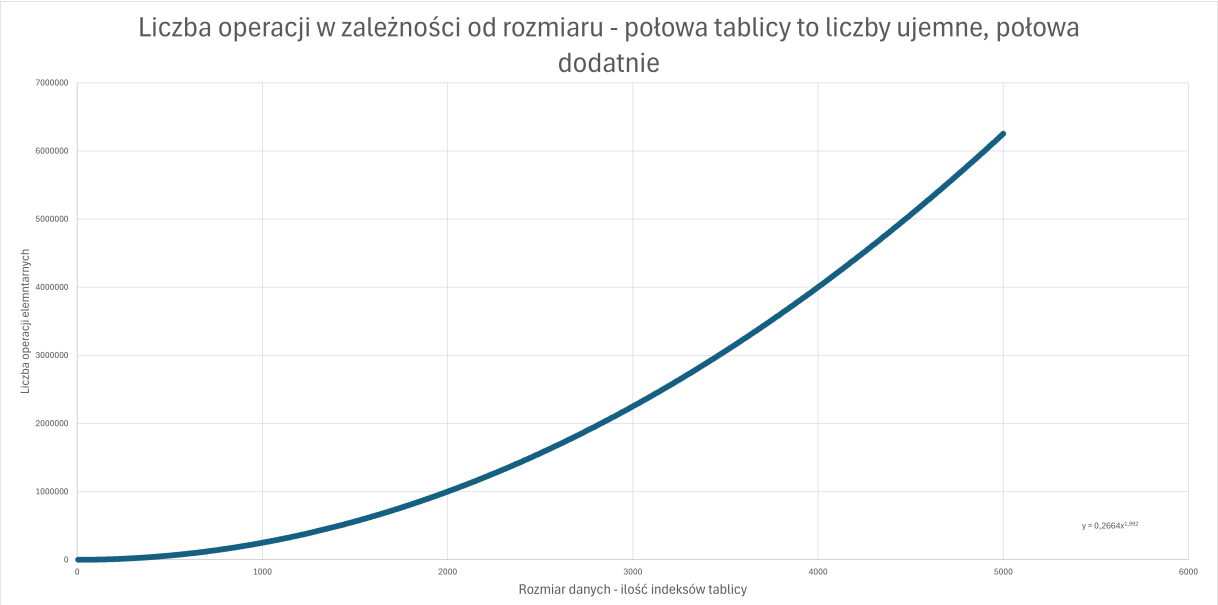
3.2.2 Przypadek 2

W tym modelu rozpatrujemy tablicę wypełnioną w połowie liczbami dodatnimi, a w połowie ujemnymi, gdzie liczby ujemne znajdują się na jej początku. Złoóżność obliczeniowa dla poszczególnych rozmiarów danych wygląda wtedy następująco:

Rozmiar tablicy	Liczba operacji
5	9
10	30
15	64
30	240
50	650
100	2550
200	10100
1000	250500
10000	24980004

Tabela 3: Złożność obliczeniowa programu 1 w przypadku 2 (połowa liczb ujemnych, połowa dodatnich)

W przypadku tym obserwujemy występowanie złożoności kwadratowej.



Rysunek 5: Wykres złożoności obliczeniowej programu 1 w 2 przypadku

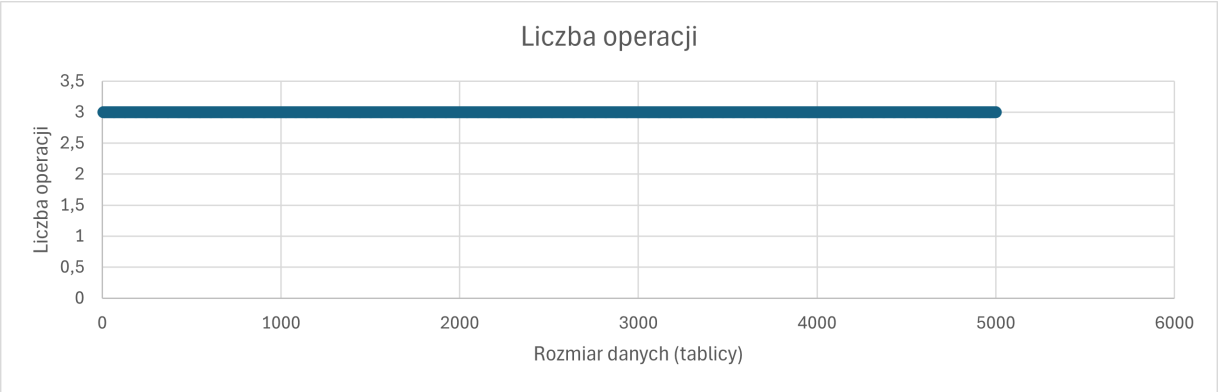
3.2.3 Przypadek 3:

Rozpatrzmy tablicę o rozmiarze n wypełnioną liczbami ujemnymi poza jedną pozycją (np.3), gdzie występuje liczba dodatnia. Poniższa tabela przedstawia liczę operacji algorytmu w zależnosci od rozmiaru danych (n).

Rozmiar tablicy (n)	Liczba operacji
5	3
10	3
15	3
30	3
50	3
100	3
200	3
1000	3
10000	3

Tabela 4: Liczba operacji dla przypadku 3 w zależności od n

W tym przypadku złożoność obliczeniowa jest analogiczna jak w przypadku, gdy występowała jedna liczba ujemna na końcu tablicy - jest ona równa **odległości od końca tablicy i jest stała, niezależnie od rozmiaru danych**.



Rysunek 6: Wykres złożoności obliczeniowej programu 1 w 3 przypadku

3.3 Złożoność obliczeniowa algorytmu 2

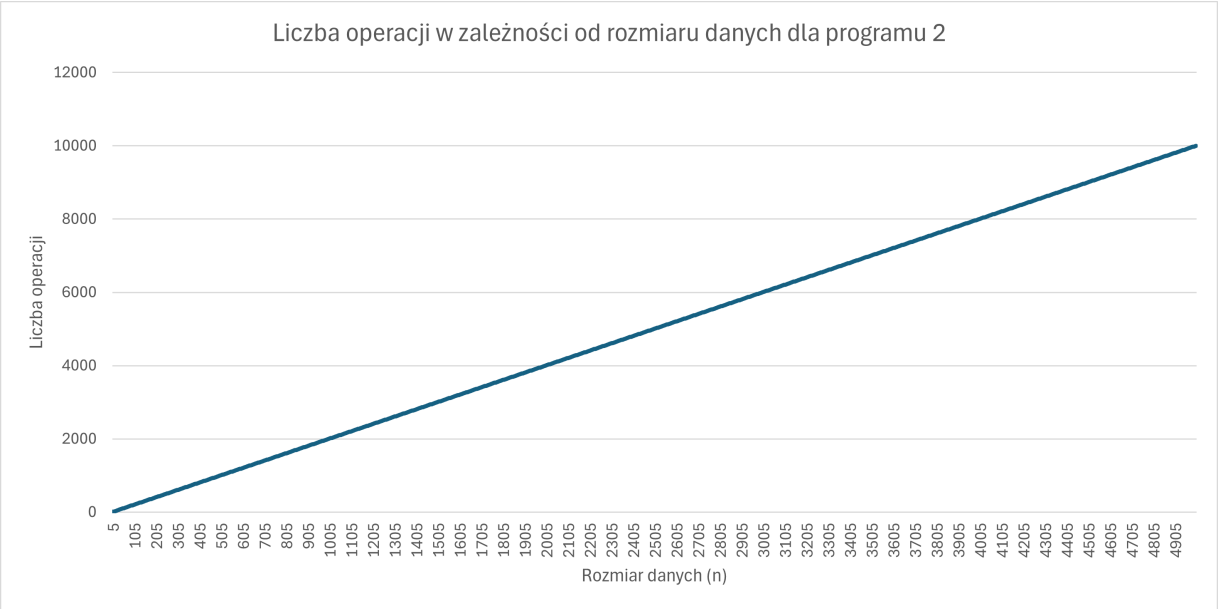
W przypadku algorytmu 2, złożoność obliczeniowa nie będzie zależna od rozmieszczenia danych, czy ich typu. Liczba operacji w zależności od rozmiaru danych będzie prezentować się następująco:

Rozmiar tablicy (n)	Liczba operacji
5	10
10	20
15	30
30	60
50	100
100	200
200	400
1000	2000
10000	20000

Tabela 5: Liczba operacji programu 2 w zależności od n

Widać, że złożoność obliczeniowa **jest zależna liniowo od rozmiaru tablicy**. Dla tablicy n-elementowej dana będzie wzorem:

$$O_n = 2n$$



Rysunek 7: Wykres złożoności obliczeniowej programu 2

3.4 Złożoność czasowa

Kolejnym ważnym aspektem wydajności programów jest ich złożoność czasowa. Pomiary wykonywałem wewnątrz programu przy pomocy biblioteki chrono. Dla mniejszych wartości, róznice w czasie wykonywania programów nie były znaczne. Dopiero przy większych rozmiarach danych można było zauważyć zmiany. Złożoności czasowe programów w zależności od rozmiaru tablicy przedstawia poniższa tabela:

Rozmiar tablicy (n)	Czas wykonania program 1 (w ms)			Czas wykonania program 2 (w ms)
	Przypadek 1	Przypadek 2	Przypadek 3	
10000	1,03	178,6	1,15	1,1
200000	1,5	23070	3,2	1,6
300000	1,5	57505	4,3	3,5
400000	2,1	91996	5,3	5,5
500000	2,7	147886	5,5	7,15

Tabela 6: Czas wykonania programów w milisekundach w zależności od n

3.5 Złożoność pamięciowa

Złożoność pamieciowa opisuje, ile zasobów pamięci potrzebuje kompilator do wykonania algorytmu. W momencie deklaracji zmiennej następuje rezerwacja pamięci. Na złożoność pamięciową składają się zmienne, tablice, zarówno statyczne i dynamiczne, jak i inne struktury danych.

3.5.1 Program 1

W programie 1 występują zmienne: **nowa, akt, poz, licz, l**. Są to zmienne typu **int**, każda z nich zajmuje więc po 4 bajty. W programie występuje również tablica **tab[N]**, też typu int, a skoro na każdy jej indeks rezerwowana jest pamięć 4 bajtów, to zajmuje ona **4N bajtów**.
Łączną pamięć rezerwowaną przez algorytm można więc opisać zależnością:

$$P(N) = 4 \cdot N + 20$$

gdzie N to rozmiar tablicy.
Zależność ta jest liniowa , zatem program ma złożoność pamięciową **O(n)**

3.5.2 Program 2

W tym przypadku występują zmienne typu int: **uj, dod, p1, p2**, których każda zajmuje po 4 bajty pamięci. Dodatkowo, występują dwie tablice statyczne o rozmiarze N - **tab[N] i nowa[N]**. Znajdują się tu również dynamicznie alokowane tablice pomocznicze - **dodatnie[N-uj] i ujemne[uj]**, łącznie o rozmiarze N. Każda z tych tablic zajmuje po **4N bajtów**. Łączna pamięć zajmowaną przez program opisuje więc wzór:

$$C(N) = 12 \cdot N + 16$$

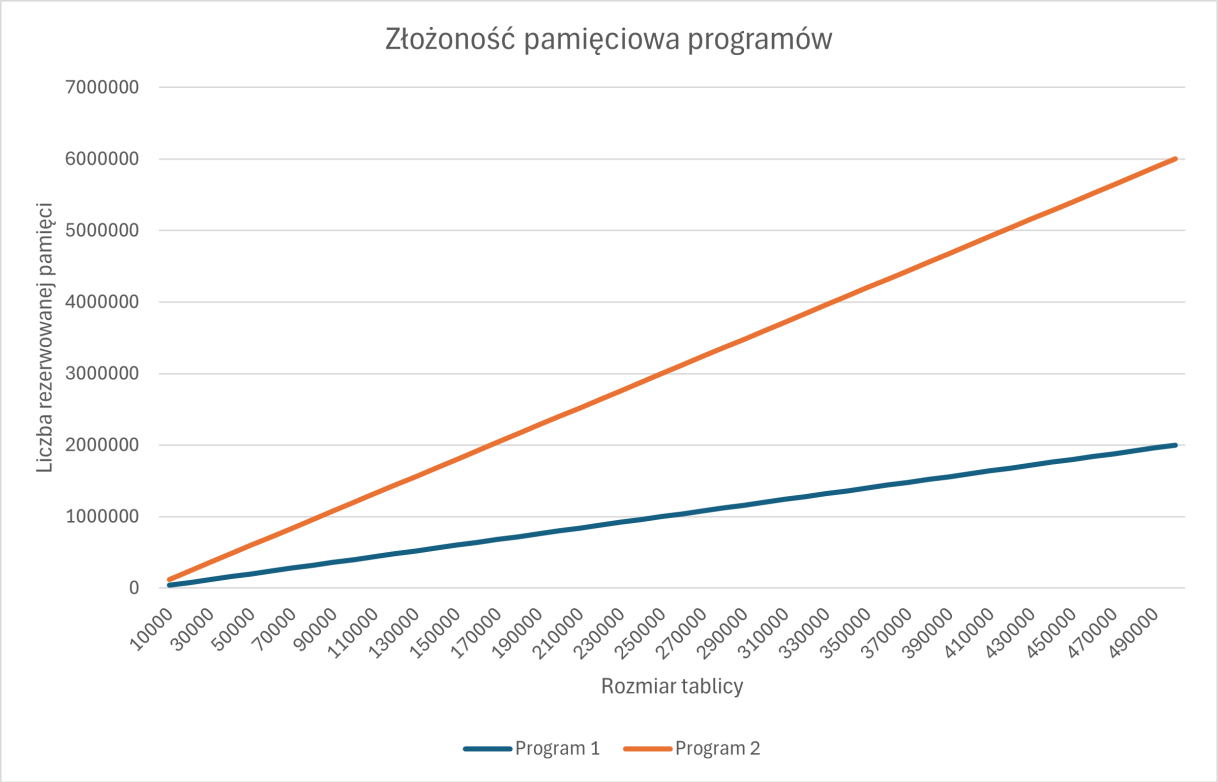
Zależność ta również jest liniowa , zatem program ma złożoność pamięciową **O(n)**, choć tutaj rezerwowana pamięć rośnie szybciej niż w przypadku przykładu 1.

3.5.3 Testowanie złożoności pamięciowej obu programów.

Porównanie złożoności pamięciowej obu algorytmów widoczne jest w poniższej tabeli:

Rozmiar tablicy (n)	Liczba rezerwowanej pamięci(w bajtach)	
	Program 1	Program2
10000	40020	120016
200000	800020	2400016
300000	1200020	3600016
400000	1600020	4800016
500000	2000020	6000016

Tabela 7: Liczba operacji programu 2 w zależności od n



Rysunek 8: Wykres złożoności pamięciowej programów

3.6 Wnioski

- Istnieją rózne sposoby rozwiązania problemu o różnych złożonościach obliczeniowych, czasowych i pamięciowych.
- Rozwiązania „siłowe" (wprost) najczęsciej są bardziej złożone obliczeniowo.
- Rozwiązania mniej złożone obliczeniowo najczęściej potrzebują większej ilości pamięci.
- Program nie zawsze musi mieć stałą złożoność obliczeniową. Zależeć ona może od różnych czynników, jak w przyapdku programu 1, gdzie złożoność wahała się od kilku operacji do złożoności kwadratowej.
- W niektórych prypadkach teoretycznie bardziej złożone funkcje wykonują mniej operacji niż algorytmy o teoretycznie niższej złożoności obliczeniowej. Dlatego aby dokładnie ocenić złożoność algorytmu, należy go analizować w różnych przypadkach.
- Różnice w liczbie wykonywanych operacji mają odzwierciedlenie w czasie wykonywania się programu.

4 Podsumowanie

Problem przesuwania liczb ujemnych na koniec tablicy ma wiele możliwości rozwiązania o różnych klasach złożoności. Alogorytm pierwszy wykonuje operacje przesuwania liczb ujemnych na koniec, lub dodatnich na początek, w zależności od tego, których jest więcej. Jest to rozwiązanie w przeważającej liczbie przypadków dość skomplikowane obliczeniowo, jednak w pewnych warunkach liczba operacji wykonywanych przez algorytm jest bardzo mała.
Z kolei drugi przedstawiony algorytm prezentuje inne podejście. Polega na podziale liczb na ujemne i większe lub równe 0. Następnie „skleja" części dodatnią i ujemną zbioru, przy czym najpierw umieszcza podzbiór liczb dodatnich. Rozwiązanie to, niezależnie od warunków ma stałą, liniową złożonosć obliczeniową. Najczęściej jest to rozwiąznie wydajniejsze od rozwiązania pierwszego, jednak dla pojedynczych ujemnych(dodatnich) elementów w tablicy, może wykonywać więcej operacji niż program pierwszy.

Najlepszym rozwiąznaiem wydaje się więc wykonywanie operacji programu pierwszego dla niewielkich ilości liczb ujemnych (dodatnich). W pozostałych przypadkach, na przykład kiedy ilość liczb ujemnych i dodatnich jest zbliżona, lub różnica między nimi nie jest tak duża, warto wykonywać operacje programu drugiego.