

Algorytmy i struktury danych

Zadanie projektowe

P02 Jakub Goleń
Inżynieria i analiza danych

Temat zadania

Dla zadanej tablicy liczb całkowitych przesun wszystkie elementy mniejsze od 0 na jej koniec (nalezy zachowac kolejnosc wystepowania !).

Przyklad:

Wejscie: A[] = [-10, 5, 8, -4, 1, 3, 0, -7]

Wyjscie: [5, 8, 1, 3, 0, -10, -4, -7]

Uzyte biblioteki

W calym programie uzyto nastepujace biblioteki:

- **vector** - wyposaza klasyczna tablice w kilka mechanizmow.
- **chrono** - oblicza czas.
- **random** - losuje liczbe.
- **sstream** - ulatwia dodawanie ze soba obiektow typu String.
- **bits/stdc++.h** - wyposaza wszystkie standardowe biblioteki c++ (w programie wykorzystano fstream)

Inne funkcje w programie

- **timer(arr, algorithm, count);**
 - ◆ Funkcja liczy sredni czas (suma czasow/**count**) wykonywania algorytmu (**algorithm**) w tablicy (**arr**).
- **array_generator(count);**
 - ◆ Funkcja tworzy **count**-elementowa tablice wypelniona losowymi liczbami z zakresu (-100;100).
- **array_display(arr)**
 - ◆ Funkcja wyswietla tablice (**arr**) w konsoli.
- **array_by_user()**
 - ◆ Funkcja tworzy tablice n-elementowa na zyczenie uzytkownika. Uzytkownik nastepnie musi podac wszystkie elementy.
- **result_display(arr)**
 - ◆ Funkcja wyswietla wszystkie potrzebne informacje na temat tablicy (**arr**):
 - zawartość tablicy,
 - wyniki algorytmów,
 - średnie czasy wykonania algorytmów.
- **write_to_file(arr)**
 - ◆ Funkcja zapisuje wyniki takie jak w **result_display(arr)** do pliku output.txt.

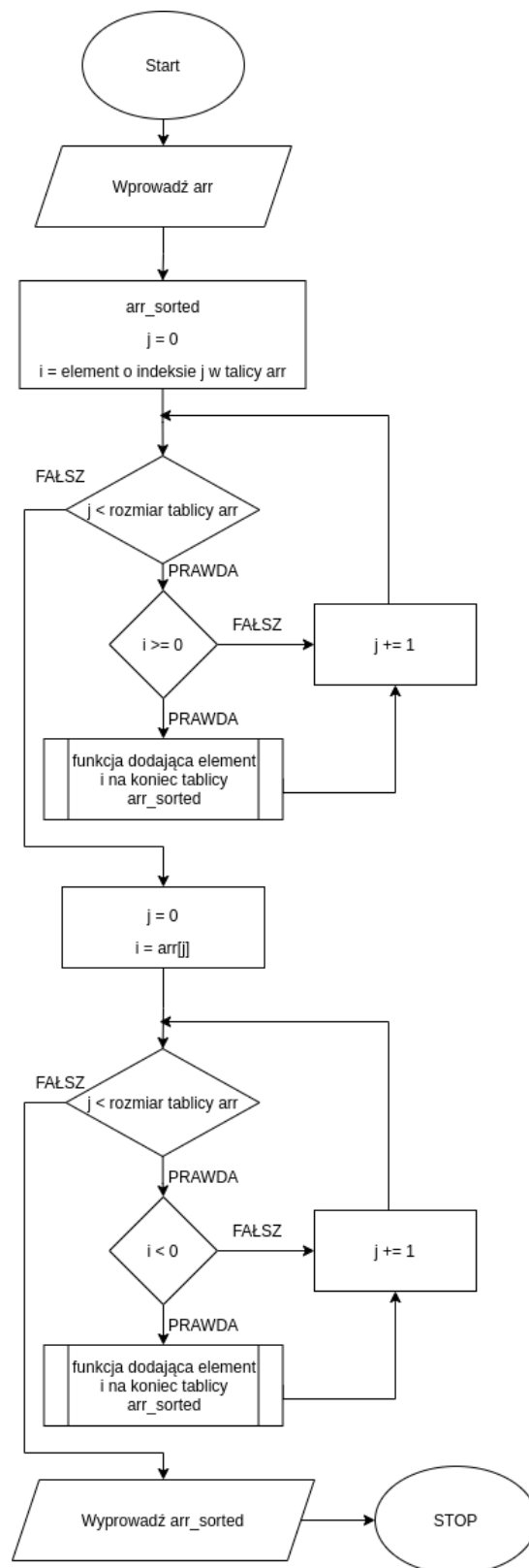
Algorytm nr. 1

Opis działania

Pierwsza funkcja o nazwie **algorithm_v1** zaczyna się od zainicjowania tablicy wyjściowej **arr_sorted**, która jest obiektem klasy **vector**. Następnie przetwarza (za pomocą zmiennej *i*) podaną tablicę na wejściu **arr** poprzez pętlę **for**. W pętli algorytm sprawdza czy element tablicy jest nieujemny. Jeżeli warunek jest spełniony za pomocą metody **push_back()**¹ element jest dodawany na sam koniec tablicy **arr_sorted**. Po zakończeniu pętli **for**, zostaje rozpoczęta kolejna pętla **for**, w której w przeciwieństwie do pierwszej pętli, sprawdzane jest czy element tablicy jest ujemny. Jeżeli element jest ujemny zostaje on dodany na koniec tablicy **arr_sorted**. Funkcja kończy się zwracaniem tablicy **arr_sorted**.

¹ **push_back(n)** - funkcja dodająca podany element *n* na koniec tablicy. Metoda ta pochodzi z klasy **vector**. Przykład: `arr.push_back(n) -> arr == [a,b,c,d,...,n]`.

Schemat blokowy²



² Schemat blokowy algorytmu 1 - autorskie

Pseudokod

```
algorithm_v1(arr){  
    arr_sorted  
  
    dla każdego( i w arr ){  
        jeżeli( i >= 0 ) {  
            funkcja dodająca na koniec tablicy arr_sorted element i3  
        }  
    }  
    dla każdego( i w arr ){  
        jeżeli( i < 0 ) {  
            funkcja dodająca na koniec tablicy arr_sorted element i  
        }  
    }  
  
    zwróć arr_sorted  
}
```

Kod

```
//algorytm 1 -realizacja poprzez nowa tablice  
vector<int> algorithm_v1(vector<int> arr){  
    vector<int> arr_sorted; //tablica wyjsciowa  
  
    for (int i : arr) { //dodawanie elementów nieujemnych do tablicy  
        if (i >= 0) {  
            arr_sorted.push_back(i);  
        }  
    }  
    for (int i : arr) { //dodawanie elementów ujemnych do tablicy  
        if (i < 0) {  
            arr_sorted.push_back(i);  
        }  
    }  
    return arr_sorted; //zwracanie tablicy wyjsciowej  
}
```

³ W kodzie C++ jest to funkcja `push_back()`

Algorytm nr. 2

Opis działania

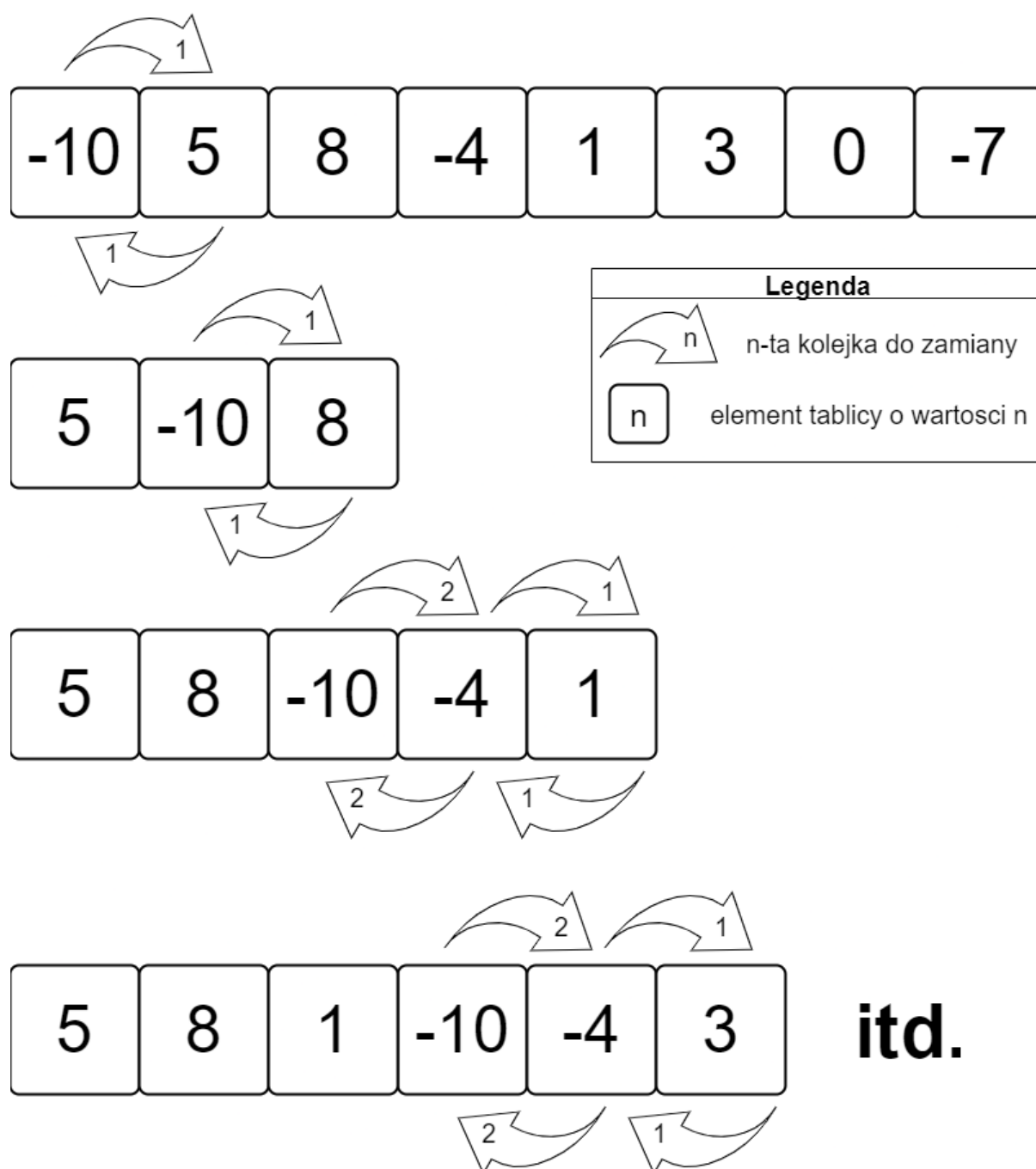
Druga funkcja o nazwie **algorithm_v2** działa zupełnie inaczej niż pierwsza. Zaczyna się od zainicjowania zmiennej pomocniczej **count** o typie **int** i wartości 1. Służy ona później do oznaczania ilości liczb ujemnych jak i ilości wykonań pętli do funkcji **swap**. Następnie funkcja rozpoczyna pętlę for gdzie używa zmiennej **i** typu **int** o wstępnej wartości **0**, jako indeks wstępnie podanej tablicy **arr**. W pierwszym warunku w pętli sprawdzane jest czy wartość elementu pod indeksem **i** jest mniejsza od **0**. Jeżeli warunek jest spełniony przedstawione są warunki:

1. Warunek 1 - czy następny element tablicy jest mniejszy od 0. Jeżeli warunek jest spełniony następuje inkrementacja zmiennej **count**.
2. Warunek 2 - czy następny element tablicy jest większy bądź równy 0. Jeżeli warunek jest spełniony następuje pętla for, w której dla zmiennej **j** o typie **int** i wstępnej wartości **0**, w czasie gdy **j** jest mniejsze od **count**, wykonywana jest funkcja **swap**⁴ która zamienia element o indeksie **i-j** z elementem o indeksie **i+1-j**. Zmienna **j** następnie się inkrementuje.

Na koniec funkcja zwraca tablicę **arr**.

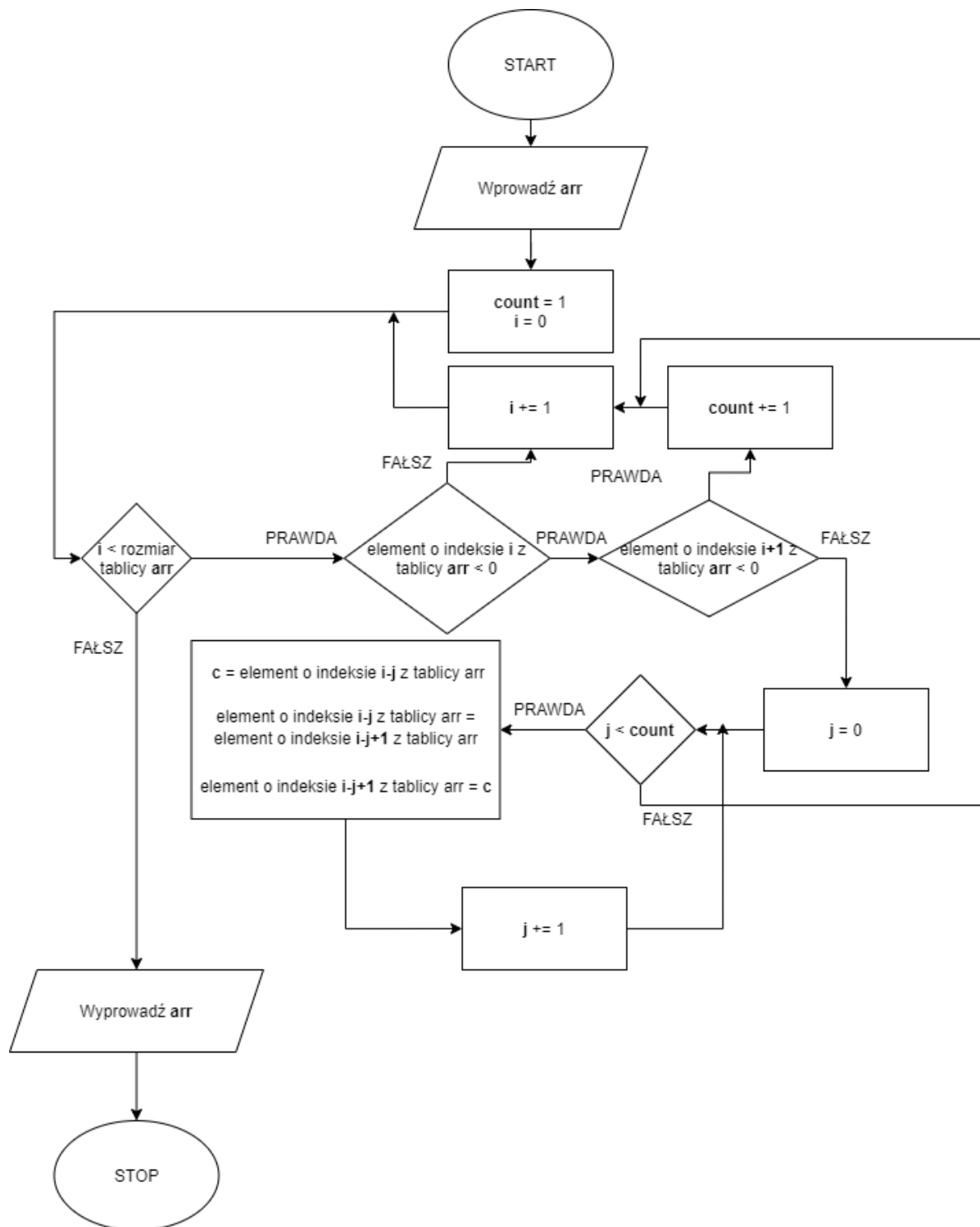
⁴ **swap(x,y)** - funkcja zamieniająca wartości dwóch obiektów ze sobą.

Graficzna reprezentacja algorytmu⁵



⁵ Graficzna reprezentacja algorytmu 2 - autorskie

Schemat blokowy⁶



⁶ Schemat blokowy algorytmu 2 - autorskie

Pseudokod

```
algorithm_v2(arr){  
  
    count = 1  
  
    dla (i = 0; i < (rozmiar tablicy arr)-1; i++){  
        jeżeli ( element o indeksie i z tablicy arr < 0 ){  
            jeżeli ( element o indeksie i+1 z tablicy arr < 0 ){  
                count = count + 1  
            }  
            lub jeżeli ( element o indeksie i+1 z tablicy arr >= 0 ){  
                for ( j = 0; j < count; j++){  
                    c = element o indeksie i-j z tablicy arr  
                    element o indeksie i-j z tablicy arr = element o indeksie i-j+1 z tablicy arr  
                    element o indeksie i-j + 1 z tablicy arr = c7  
                }  
            }  
        }  
    }  
    zwróć arr  
}
```

Kod

```
//algorytm 1 - realizacja poprzez swap()  
vector<int> algorithm_v2(vector<int> arr){  
    int count = 1; // ilosc wykonan funkcji swap()  
    for ( int i = 0; i < arr.size()-1; i++){  
        if ( arr[i] < 0 ){  
            if ( arr[i+1] < 0 ) {  
                count += 1;  
            }  
            else{  
                for ( int j = 0; j < count; j++){  
                    swap( &arr[i-j], &arr[i+1-j] );  
                }  
            }  
        }  
    }  
    return arr;  
}
```

⁷ W kodzie C++ jest to funkcja **swap()**

Algorytmy - porównanie

Wyniki przykładów

```
Przed podziałem:
[-10, 5, 8, -4, 1, 3, 0, -7]

ALGORYTM 1

Po podziale poprzez algorytm 1:
[5, 8, 1, 3, 0, -10, -4, -7]
Sredni czas wykonania algorytmu 1: 1354 nanosekund.

ALGORYTM 2

Po podziale poprzez algorytm 2:
[5, 8, 1, 3, 0, -10, -4, -7]
Sredni czas wykonania algorytmu 2: 373 nanosekund.

Przed podziałem:
[92, -32, 8, -121, 55, 32, 34, -23, 0]

ALGORYTM 1

Po podziale poprzez algorytm 1:
[92, 8, 55, 32, 34, 0, -32, -121, -23]
Sredni czas wykonania algorytmu 1: 1482 nanosekund.

ALGORYTM 2

Po podziale poprzez algorytm 2:
[92, 8, 55, 32, 34, 0, -32, -121, -23]
Sredni czas wykonania algorytmu 2: 448 nanosekund.
```

```
Przed podziałem:  
[-29, -30, 20, 10, 60, -38, -489, 892, 829, -894, 2516, 72637, -9400]
```

ALGORYTM 1

```
Po podziale poprzez algorytm 1:  
[20, 10, 60, 892, 829, 2516, 72637, -29, -30, -38, -489, -894, -9400]  
Średni czas wykonania algorytmu 1: 1630 nanosekund.
```

ALGORYTM 2

```
Po podziale poprzez algorytm 2:  
[20, 10, 60, 892, 829, 2516, 72637, -29, -30, -38, -489, -894, -9400]  
Średni czas wykonania algorytmu 2: 568 nanosekund.
```

Tabela średniego czasu wykonania algorytmu (w sekundach)⁸

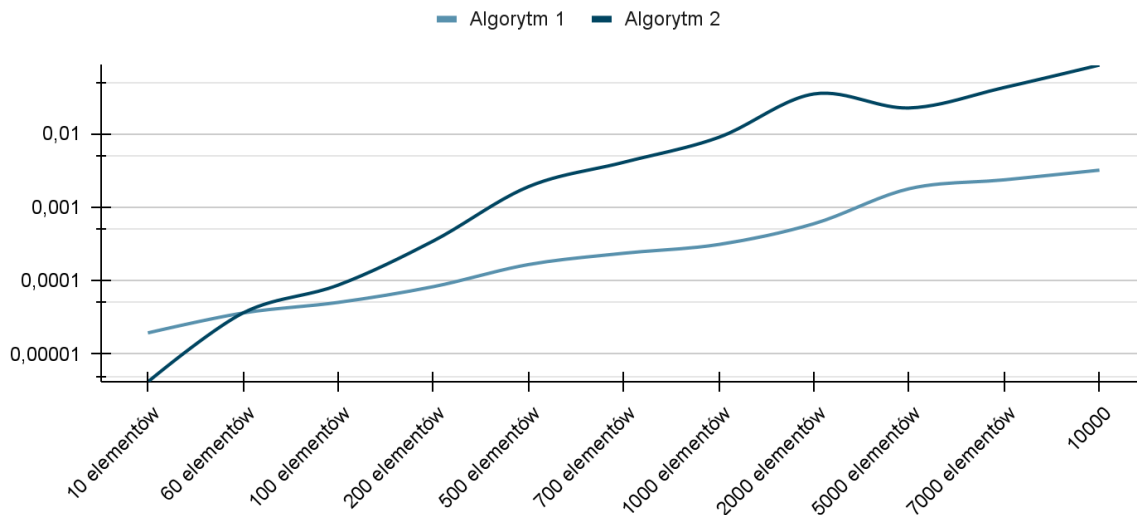
Średnia czasu z 100 testów na losowo generowanych tablicach 10,100,1000,10000 elementów:

	Algorytm 1	Algorytm 2
10 elementów	0,000001940 s	0,000000418 s
60 elementów	0,00003623 s	0,00003623 s
100 elementów	0,000005046 s	0,000008678 s
200 elementów	0,00008264 s	0,00034608 s
500 elementów	0,00016520 s	0,00189764 s
700 elementów	0,00023534 s	0,00407049 s
1000 elementów	0,000030995 s	0,000889798 s
2000 elementów	0,000594300 s	0,034659900 s
5000 elementów	0,00177763 s	0,022343900 s
7000 elementów	0,00234917 s	0,042439500 s
10000 elementów	0,000318152 s	0,085705000 s

⁸ Tabela 1 - porównanie czasów algorytmów - autorskie

Wykres średniego czasu wykonania algorytmu (w sekundach)⁹

Średni czas (w sekundach) wykonania algorytmów poprzez tablicę n-elementową w zakresie $<-100, 100>$



Podsumowanie

Porównując te dwa algorytmy ze sobą można wyciągnąć bardzo proste wnioski. Przy tablicach w okolicach 10 elementów algorytm drugi wygrywa z pierwszym mając prawie pięciokrotnie mniejsze czasy. Sytuacja się zmienia w okolicach 60 elementów, gdzie algorytmy zaczynają sobie dorównywać. Diametralnie zmienia się wszystko w momencie gdy tablice osiągają 1000 czy 10000 elementów. Przy 1000 elementowych tablicach różnica pomiędzy algorytmami jest 30-krotna, a przy 10000 elementowych tablicach różnica jest 270-krotna. Różnice te wynikają z podejścia algorytmu do problemu. W przypadku pierwszego algorytmu ilość liczb ujemnych, a ilość nieujemnych jest praktycznie obojętna. Algorytm i tak dodaje je do nowej tablicy niezależnie od ich typu. Dla drugiego algorytmu ilość liczb ujemnych jest ważna, gdyż od ilości zależy ilość zamian w funkcji. Skutkuje to innymi czasami. Podobnie jest z położeniem pierwszej liczby ujemnej w tablicy, aczkolwiek wpływ tego jest mniej znaczący.

⁹ Wykres 2 - porównanie czasów algorytmów - autorskie