Algorytmy i struktury danych

Zadanie projektowe

P02 Jakub Goleń
Inżynieria i analiza danych

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

Użyte biblioteki

W całym programie użyto następujące biblioteki:

- ➔ **vector** - wyposaża klasyczną tablicę w kilka mechanizmów.
- ➔ **chrono** - oblicza czas.
- ➔ **random** - losuje liczbę.
- ➔ **sstream** - ułatwia dodawanie ze sobą obiektów typu String.
- ➔ **bits/stdc++.h** - wyposaża wszystkie standardowe biblioteki c++ (w programie wykorzystano fstream)

Inne funkcje w programie

- ➔ **timer**(**arr**, **algorithm**, **count**);
  - ◆ Funkcja liczy średni czas (suma czasów/**count**) wykonywania algorytmu (**algorithm**) w tablicy (**arr**).
- ➔ **array_generator**(**count**);
  - ◆ Funkcja tworzy **count**-elementową tablicę wypełnioną losowymi liczbami z zakresu (-100;100).
- ➔ **array_display**(**arr**)
  - ◆ Funkcja wyświetla tablicę (**arr**) w konsoli.
- ➔ **array_by_user**()
  - ◆ Funkcja tworzy tablicę n-elementową na życzenie użytkowania. Użytkownik następnie musi podać wszystkie elementy.
- ➔ **result_display**(**arr**)
  - ◆ Funkcja wyświetla 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.

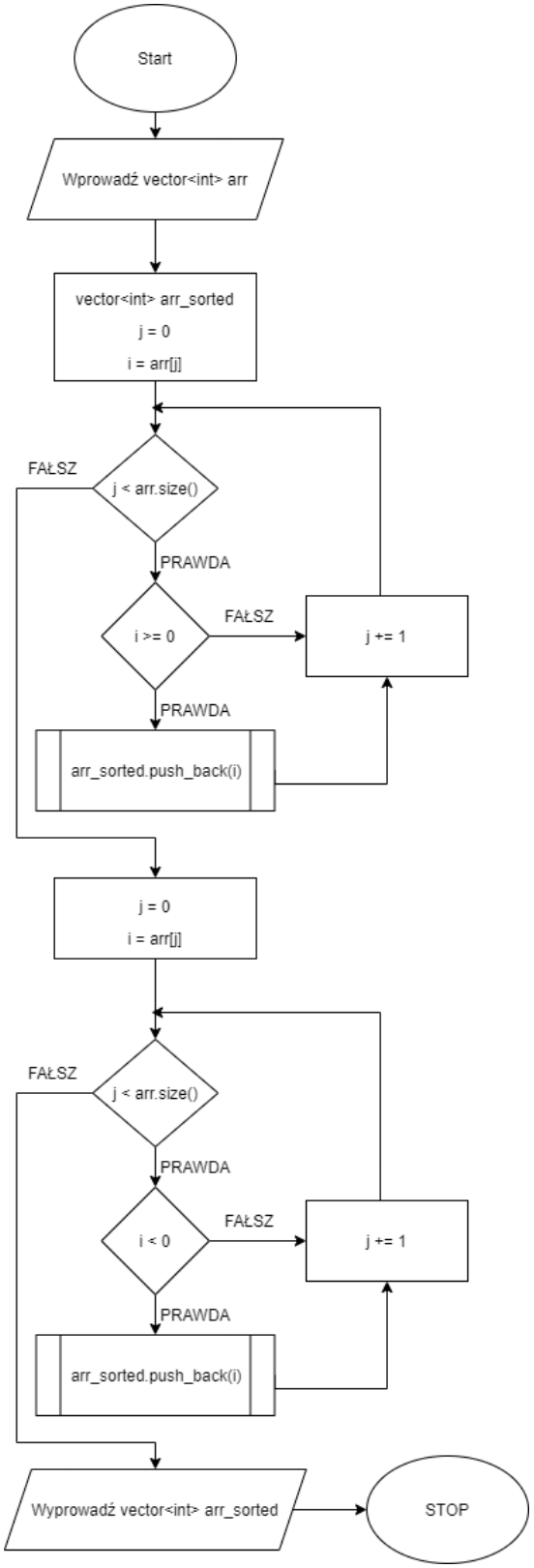
Algorytm nr. 1

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 blokowy[2]



[2] Schemat blokowy algorytmu 1 - autorskie

Pseudokod

```
vector<int> algorithm_v1(vector<int> arr){
        vector<int> arr_sorted

        dla każdego( int i w arr ){
                jeżeli( i >= 0 ) {
                        arr_sorted.push_back(i)
                }
        }
        dla każdego( int i w arr ){
                jeżeli( i < 0 ) {
                        arr_sorted.push_back(i)
                }
        }

        zwróć arr_sorted
}
```

Algorytm nr. 2

Opis działania

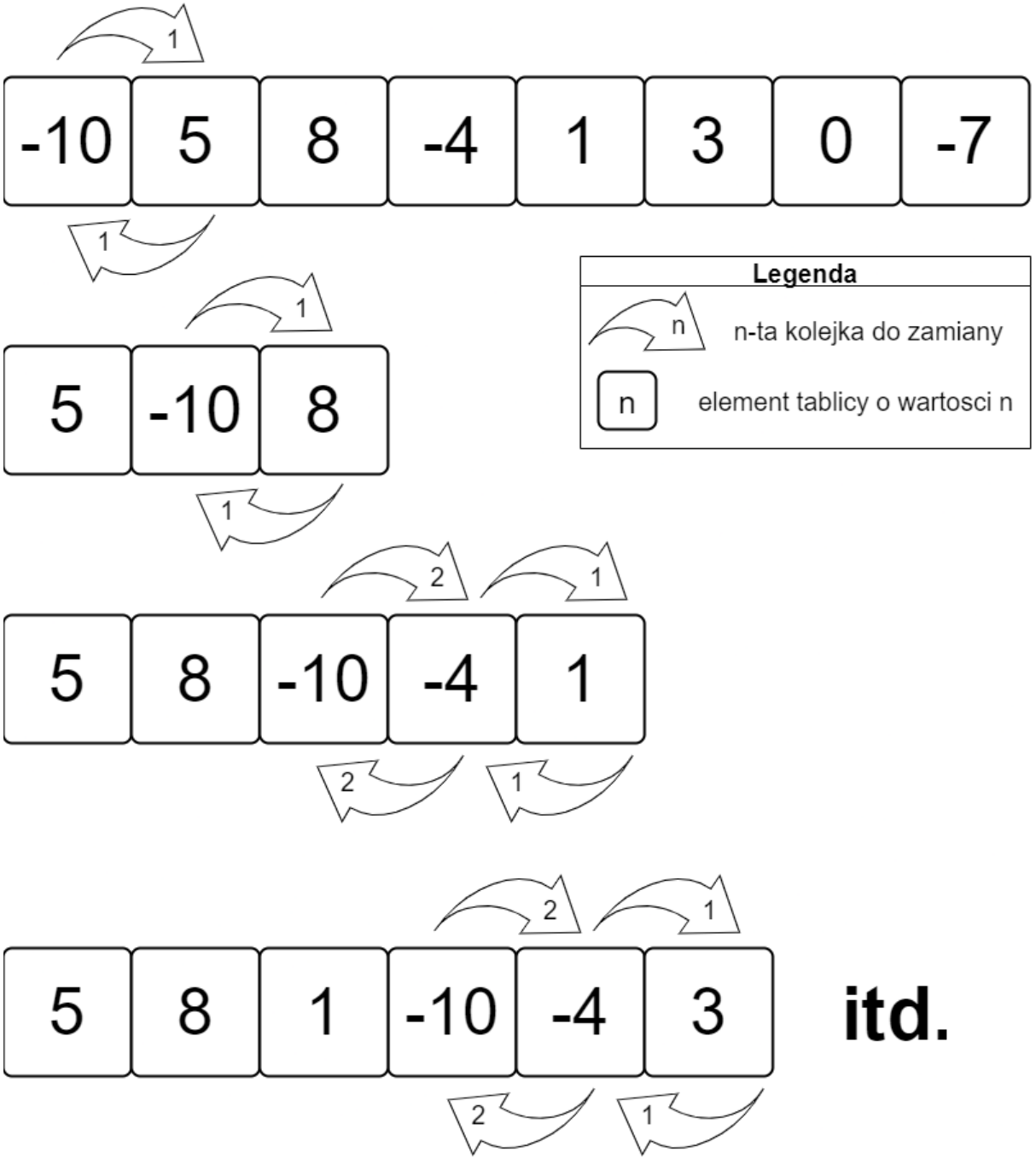
Druga funkcja o nazwie **algorithm_v2** działa zupełnie inaczej niż pierwsza. Zaczyna się od zainicjowania zmiennej pomocniczej **count** o typie **int** i wartości 1. Służy ona później do oznaczania ilości liczb ujemnych jak i ilości wykonań pętli do funkcji **swap**. Następnie funkcja rozpoczyna pętlę 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**[3] 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**.

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

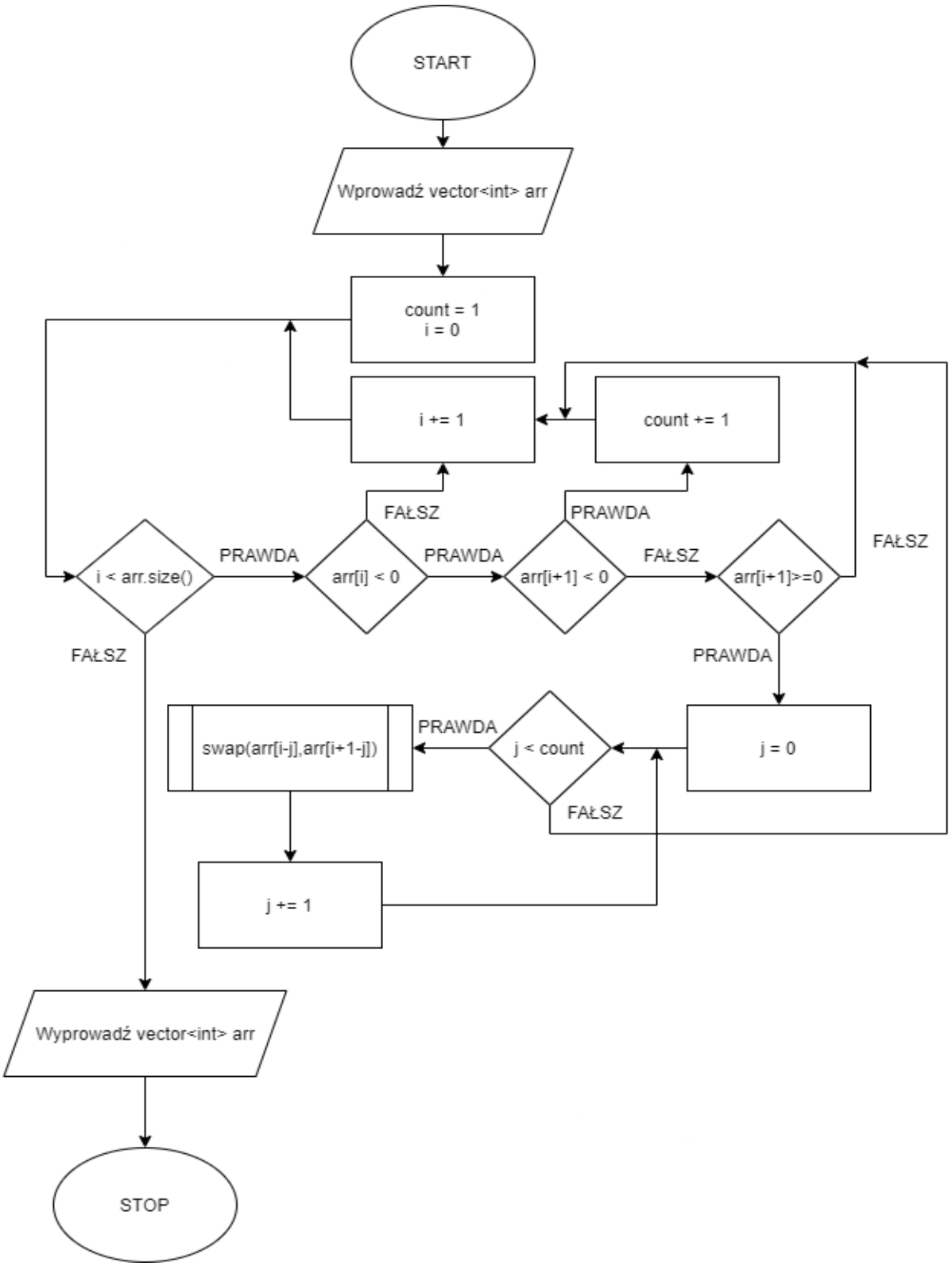
Graficzna reprezentacja algorytmu[4]



itd.

[4] Graficzna reprezentacja algorytmu 2 - autorskie

Schemat blokowy[5]



[5] Schemat blokowy algorytmu 2 - autorskie

Pseudokod

```
vector<int> algorithm_v2(vector<int> arr){

  int count = 1;

  dla ( int i = 0; i < arr.size()-1; i++ ){
     jeżeli ( arr[i] < 0 ){
        jeżeli ( arr[i+1] < 0 ) {
           count += 1;
        }
        lub jeżeli ( arr[i+1] >= 0 ){
           for ( int j = 0; j < count; j++ ){
              swap( arr[i-j],arr[i+1-j] );
           }
        }
     }
  }
  return arr;
}
```

Algorytmy - porównanie

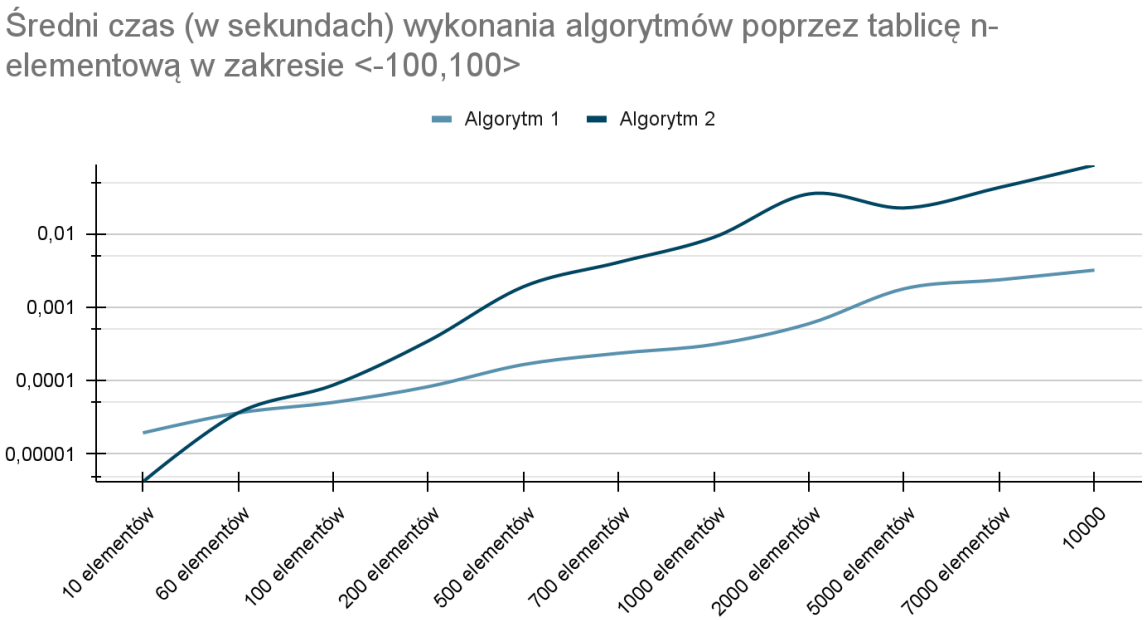
Tabela średniego czasu wykonania algorytmu (w sekundach)[6]

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

	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

[6] Tabela 1 - porównanie czasów algorytmów - autorskie
Dokument bez tytułu - Wykres liniowy 1

Wykres średniego czasu wykonania algorytmu (w sekundach)[7]



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.

[7] Wykres 2 - porównanie czasów algorytmów - autorskie
Dokument bez tytułu - Wykres liniowy 1