2021

Politechnika Rzeszowska

Mateusz Sokal

[SPRAWOZDANIE -PROJEKT 12.11.2021]

Opis problemu, rozwiązania, schematy algorytmów z zadania projektowego.

Spis treści:

1.Opis problemu, teoria i implementacja.....2

2.Bubble Sort vs. Quicksort.....4

3.Schemat blokowy algorytmu.....6

4.Algorytm w pseudokodzie.....7

5.Kod programu.....7

6.Specyficzne przykłady.....10

7.Wnioski.....10

1. Opis problemu, teoria i implementacja

Treść: Dla zadanej tablicy liczb całkowitych znajdź maksymalny iloczyn dwóch elementów znajdujących się w tej tablicy.

Przykład:

Wejście: A = [-10,8,5,-4,1]

Wyjście: Czynniki generujące maksymalny iloczyn to: [-10,-4] i [8,5]

Tematem tego projektu było znalezienie maksymalnego iloczynu wśród n-elementowego zbioru liczb całkowitych.

Tablica to zbiór danych tego samego typu. Tablicę, tak samo jak pojedyncze zmienne deklarujemy według wzoru:

```
typ_danych nazwa_tablicy [ilość_elementów]
```

Jeżeli znamy liczbę i wartość elementów tablicy, zamiast **[ilość_elementów]** możemy je wypisać jako:

```
typ_danych nazwa_tablicy = {lista_elementów_tablicy}
```

Kolejną funkcją potrzebną do wykonania zadania jest **pętla for,** która umożliwia powtarzanie pewnych instrukcji do momentu spełnienia warunku końcowego. Pętlę tą definiujemy w ten sposób:

```
for (typ_danych wyrażenie1, wyrażenie2, wyrażenie3)
```

wyrażenie1 - instrukcja wykonana przed pierwszym obiegiem pętli,

wyrażenie2 - warunek zakończenia pętli,

wyrażenie3 - opis instrukcji wykonywanej za każdym obiegiem pętli.

Pętla while (oraz do while) wykonuje instrukcje dopóki warunek końcowy jest spełniony, od **pętli for** różni się tym, że nie trzeba definiować po ilu powtórzeniach pętla ma się zakończyć. **Pętlę while** definiujemy jako:

```
while(warunek)
 {instrukcja}

     do
 {instrukcja}
while(warunek)
```

Sortowanie jeden z podstawowych problemów informatyki, polegający na uporządkowaniu zbioru danych względem pewnych cech charakterystycznych każdego elementu tego zbioru.

Podczas pracy nad projektem stosowałem dwa rodzaje sortowania: sortowanie bąbelkowe (z ang. Bubble Sort) oraz sortowanie szybkie (z ang. Quicksort).

Sortowanie bąbelkowe jest jednym z najprostszych oraz najbardziej intuicyjnych metod posortowania zbioru liczbowego. Polega na porównywaniu dwóch kolejnych elementów i zamianie ich kolejności, jeżeli zaburza ona porządek, w jakim się sortuje tablicę. Można to zapisać jako:

```
for(int i=0; i<n; i++)
  {
        for(int j=1; j<n-i; j++)
        {
            if(tab[j-1]<tab[j])
            {
                swap(tab[j-1], tab[j]); // czyt. zamień miejscami
            }
         }
  }
```

Sortowanie szybkie to jedna z najpopularniejszych metod sortowania, działająca na zasadzie "dziel i zwyciężaj". Polega na wybraniu elementu rozdzielającego wewnątrz tablicy, po czym tablica jest dzielona na dwa fragmenty: do początkowego przenoszone są wszystkie elementy nie większe od rozdzielającego, do końcowego wszystkie większe. Potem sortuje się osobno początkową i końcową część tablicy. Proces ten kończy się po otrzymaniu tablic jednoelementowych, które nie wymagają sortowania. Taką funkcję można zapisać jako:

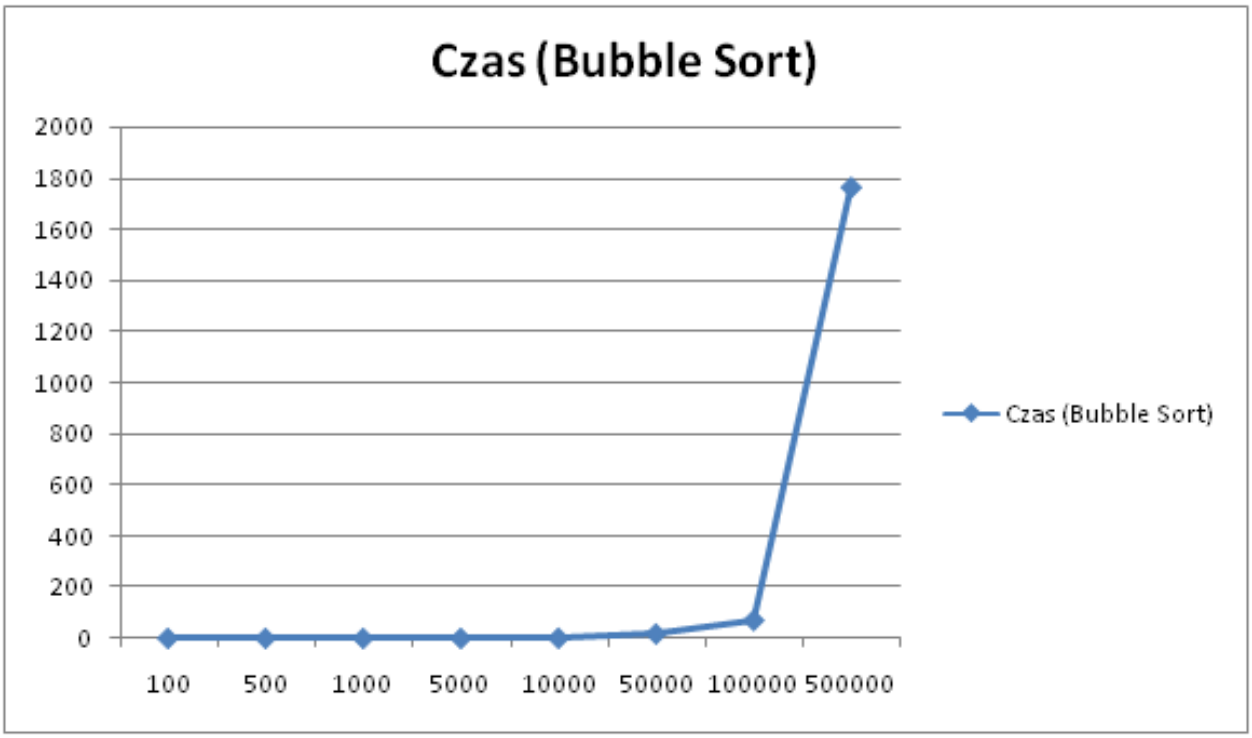
```
void quicksort(int *tablica, int lewy, int prawy)
{
    int v=tablica[(lewy+prawy)/2];
    int i,j,x;
```

```
    i=lewy;
    j=prawy;
    do
    {
        while(tablica[i]>v) i++;
        while(tablica[j]<v) j--;
        if(i<=j)
        {
            x=tablica[i];
            tablica[i]=tablica[j];
            tablica[j]=x;
            i++;
            j--;
        }
    }
    while(i<=j);
    if(j>lewy) quicksort(tablica,lewy, j);
    if(i<prawy) quicksort(tablica, i, prawy);
}
```

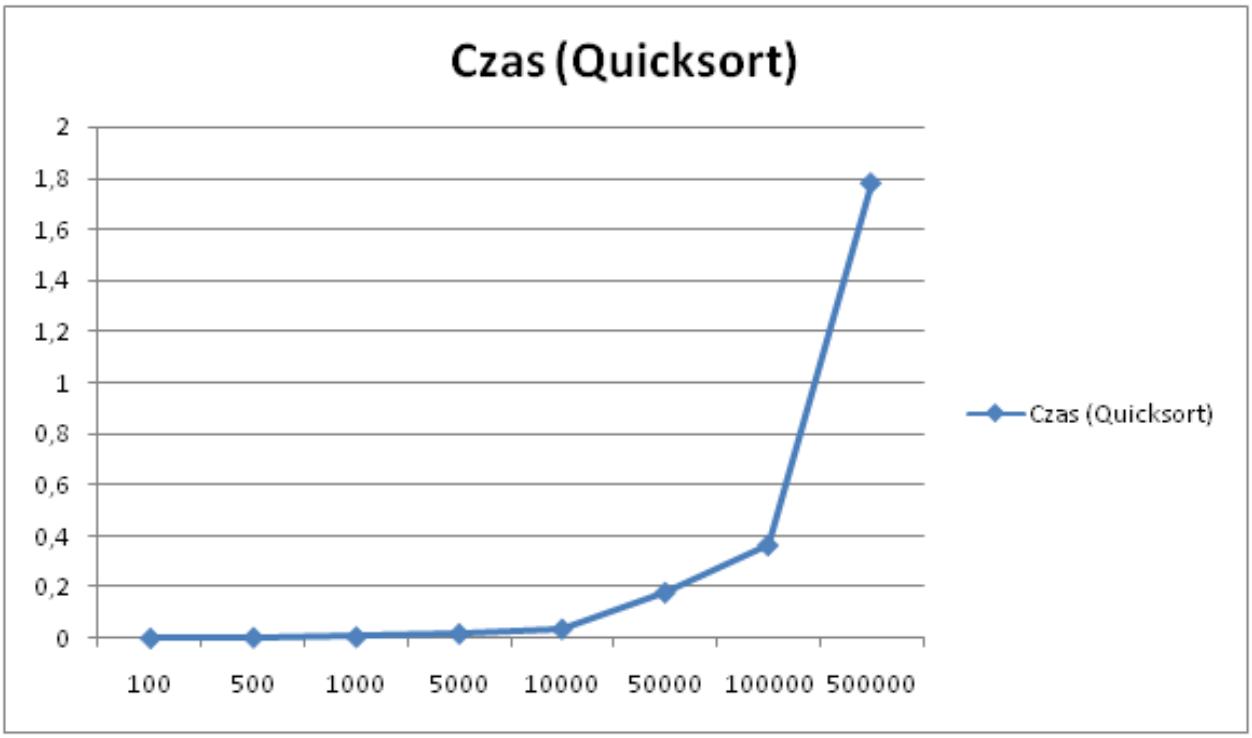
2. Bubble Sort vs. Quicksort

Główną różnicą między tymi metodami jest szybkość sortowania, którą najlepiej przedstawić na wykresie.

Czas obliczeń algorytmu (Bubble Sort)

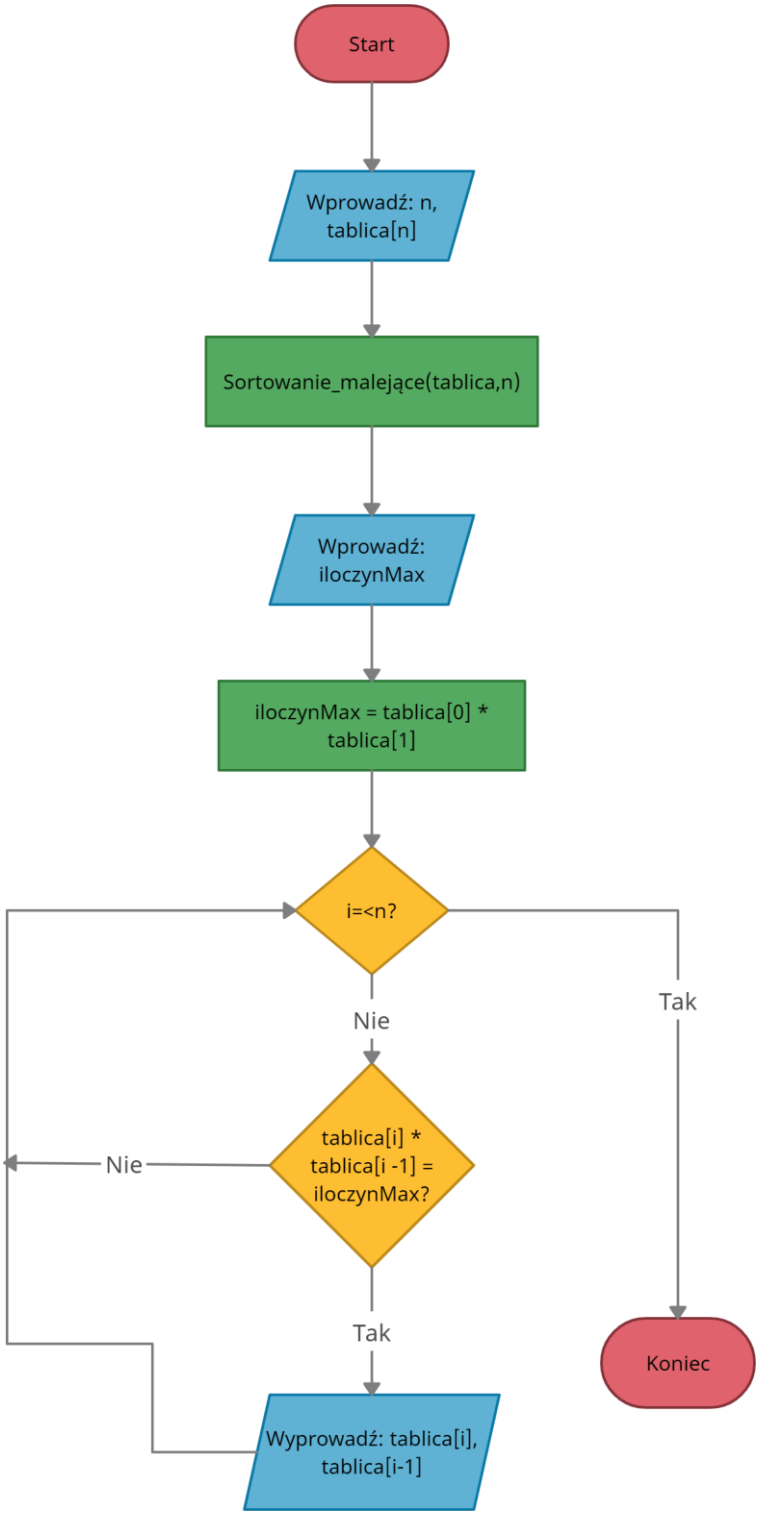


Czas obliczeń algorytmu (Quicksort)



Aby zauważyć różnicę wystarczy tylko spojrzeć na oś y, reprezentującą czas obliczeń. O ile nie ma to wielkiego znaczenia dla tablic o niewielkiej ilości elementów (do 10 tys.) tak zastosowanie szybszej metody sortowania ma zauważalne znaczenie przy dużych tablicach (100 tys. elementów i więcej)

3. Schemat blokowy algorytmu



Żródło: app.creately.com

4. Algorytm w pseudokodzie

Wczytaj n

Wczytaj a

Inicjuj tablica[n]

Jeżeli plik dane.txt **nie istnieje**

Zakończ program

W przeciwnym razie

Dla i=0 do i<n **powtarzaj**

Wczytaj a **z pliku**

tablica[i]=a

Sortuj dane malejąco z tablica[n]

Wczytaj iloczyn_max

Jeżeli tablica[0]<0 i tablica [1]<0

iloczyn_max = tablica[n-1] * tablica[n-2]

W przeciwnym razie

iloczyn_max = tablica[0] * tablica[1]

Dla i=0 do i<n **powtarzaj**

Jeżeli tablica[n] * tablica[n-1] = iloczyn_max

Wypisz tablica[n], tablica[n-1]

Zakończ program

5. Kod programu

```
#include <iostream>
#include <fstream>
```

```
#include <time.h>
#include <cstdlib>
using namespace std;

void quicksort(int *tablica, int lewy, int prawy) // funkcja sortowania danych
{
    int v=tablica[(lewy+prawy)/2];
    int i,j,x;
    i=lewy;
    j=prawy;
    do
    {
        while(tablica[i]>v) i++;
        while(tablica[j]<v) j--;
        if(i<=j)
        {
            x=tablica[i];
            tablica[i]=tablica[j];
            tablica[j]=x;
            i++;
            j--;
        }
    }
    while(i<=j);
    if(j>lewy) quicksort(tablica,lewy, j);
    if(i<prawy) quicksort(tablica, i, prawy);
}

int main()
{
    fstream dane, wyniki;
    int *tab, n;
```

```
    double a, czas;
    long iloczynMax;
    clock_t start, stop;

    dane.open("dane.txt", ios::in);
    wyniki.open("wyniki.txt", ios::out);
    if(dane.good()==false) // Sprawdzenie czy plik “dane” istnieje, bez niego
// program nie może działać
    {
        cout << "Plik dane.txt nie istnieje! Dolacz plik do folderu projektu i
sprobuj ponownie.";
        return 0;
    }
    cout<<"Ile liczb znajduje sie w tablicy? ";
    cin>>n;

    tab = new int [n];

    start = clock(); // Rozpoczęcie pomiaru czasu obliczeń
    for(int i=0; i<n; i++) // Funkcja wprowadzająca dane z pliku
    {
        dane>>a;
        tab[i]=a;
    }

    quicksort(tab, 0, n-1);

    cout << endl;
    if (tab[0]<0 && tab[1]<0) // Zapewnia poprawne działanie programu nawet gdy
    {                         // wszystkie liczby będą ujemne
        iloczynMax = tab[n-1] * tab[n-2];
    }
    else
```

```
    {
        iloczynMax = tab[0] * tab[1];
    }
    cout << "Maksymalny iloczyn: " << iloczynMax << endl;

    cout << "Pary liczb generujace najwiekszy iloczyn: " << endl;
    wyniki << "Pary liczb generujace najwiekszy iloczyn: " << endl;
    for (int i=1; i<n; i++)
    {
        if (tab[i-1]*tab[i]==iloczynMax) // Wypisywanie wyników
        {
            cout << tab[i-1] << " " << tab[i] << endl;
            wyniki << tab[i-1] << " " << tab[i] << endl;
        }
    }
    stop = clock();
    czas = (double)(stop - start)/CLOCKS_PER_SEC; // Obliczenie czasu pracy
    cout << "Czas pracy programu: " << czas << endl;
    return 0;
}
```

6. Specyficzne przykłady

Jak zachowa się program gdy:

1. **Wszystkie liczby będą miały tą samą wartość?**

Wejście: **tablica** = {10,10,10,10,10,10,10,10,10,10}

Wyjście: **Maksymalny iloczyn:** 100

Pary liczb generujące największy iloczyn: (10,10) (10,10) …

2. **Wszystkie liczby będą ujemne?**

Wejście: **tablica** = {-1,-4,-5,-9,-2,-3,-10,-6,-8,-7}

Wyjście: **Maksymalny iloczyn:** 90

Pary liczb generujące największy iloczyn: (-10,-9)

7. Wnioski

Program zdecydowanie nie jest w pełni zoptymalizowany. Bez wątpienia istnieją metody, dzięki którym bylibyśmy w stanie dalej skrócić czas pracy algorytmu, jednak uważam, że moja wersja algorytmu jest bardzo bliska maksymalnej efektywności.