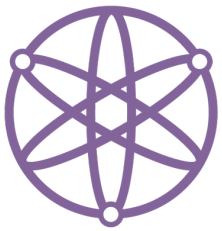


WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Daniel Czudek

Wyszukiwanie dwuelementowych podciągów

Sprawozdanie z projektu

Opiekun pracy:
dr inż. Mariusz Borkowski

Rzeszów, 2025

Spis treści

1. Treść zadania

Znajdź w ciągu liczb te podciągi dwuelementowe, których suma jest mniejsza od następującego po nich elementu.

Przyklad:

Wejście [3, 1, -5, 0, 2, 1]

wyjście [1, -5], [-5, 0]

Wejście [10, 0, 5, 3, 5, -6]

Wjście brak elementów do wyświetlenia

2. Rozwiązywanie problemu

2.1. 1 program

2.1.1. Analiza problemu

Pierwszym rozwiązaniem jest stworzenie tablicy pomocniczej ktora będzie przetrzymywać nasze sumy sąsiednich wyrazów i w następnym kroku sprawdzać które są mniejsze od ich sąsiednich wyrazów.

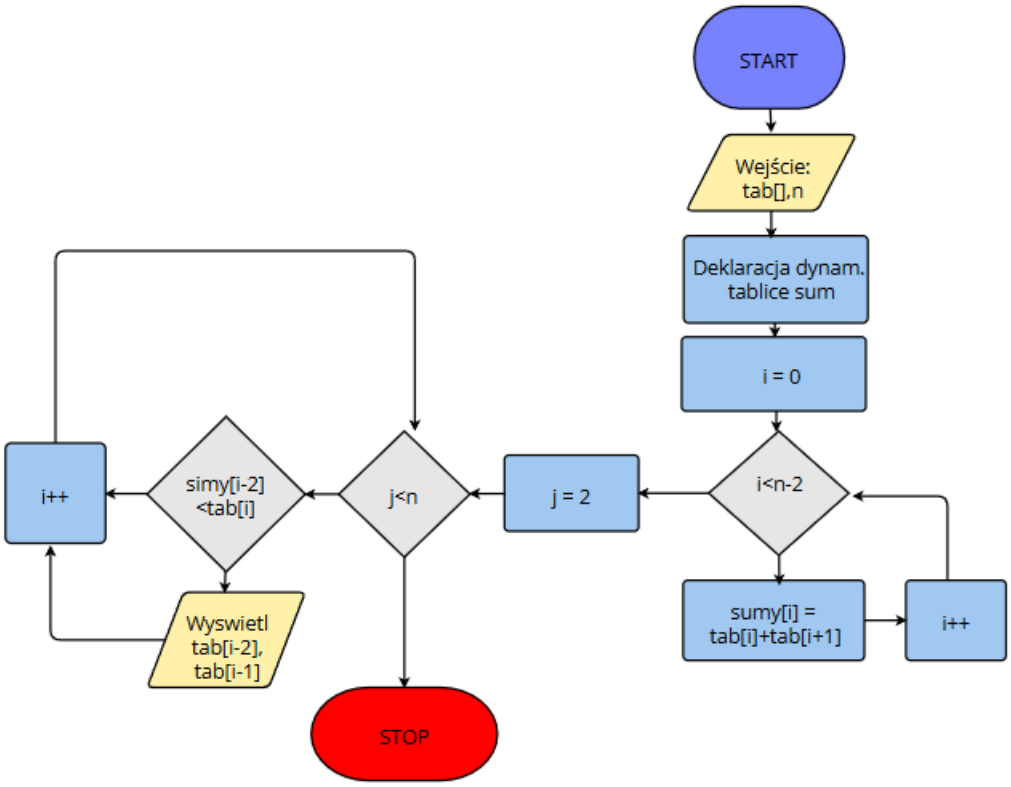
Dane wejścia

- struktura danych typu tablica przechowująca nasz ciąg liczb
- rozmiar tablicy

Dane wyjścia

- pary liczb

2.1.2. Schemat blokowy



2.1.3. Algorytm zapisany w pseudokodzie

input: tab

n

int sumy[n-2] for i:=0 i<n-2 i++

sumy[i] = tab[i]+tab[i+1]

for j: =2 j<n j++

if sumy[j-2]<tab[j]

wypisz tab[j-2], tab[j-1]

2.1.4. Teoretyczne oszacowanie złożoności obliczeniowej

Algorytm przechodzi pierwszy raz przez tablice n-2 razy przypisujac do drugiej tablicy sumy licz sąsiadujących. Drugi raz przechodzi przez tablice dając nam pary liczb. Zlozonosc obliczeniowa jest liniowa O(n)

2.1.5. Implementacja programu w C++

```cpp
void wyszukaj2(int tab[],int n) {
    int* sumy= new int[n - 2];
    for (int i = 0; i < n - 2; i++) {
        sumy[i] = tab[i] + tab[i + 1];
    }
    for (int i = 2; i < n;i++) {
        if (sumy[i - 2] < tab[i]) {
            cout << "[" << tab[i - 2] << ", " << tab[i - 1] << "]\n";
        }
    }
}
int main() {


    const int size = 100;
    int tab[size];
    srand(time(NULL));
    for (int i = 0; i < size; i++) {
        tab[i] = rand()%100;
    }
    for (int i = 0; i < size; i++) {
        cout<<tab[i]<<" ";
    }
    cout << endl;
\
    wyszukaj2(tab, size);

    return 0;
}
```

2.1.6. Przykładowe wyniki

```
39 38 53 73 97 2 20 61 66 3 72 81 85 23 71 24 66 48 63 45 44 8 7 82 88 16 34 97 87 43 47 14 92 93
 80 30 93 13 23 84 43 7 82 14 56 61 78 22 73 8 26 93 93 25 17 63 45 34 93 29 79 51 26 72 96 45 70
 91 56 62 82 54 93 76 65 15 25 90 67 45 79 69 41 32 5 8 67 57 13 18 26 10 61 55 7 7 70 21 39 74
[2, 20]
[66, 3]
[3, 72]
[8, 7]
[16, 34]
[47, 14]
[13, 23]
[43, 7]
[8, 26]
[25, 17]
[45, 34]
[15, 25]
[5, 8]
[26, 10]
[7, 7]
[21, 39]
```

```
89 45 5 73 16 91 21 78 36 4 68 7 66 90 2 44 10 40 72 36 69 35 61 73 97 32 87 14 11 84 18 32 27 24
 64 43 53 57 50 11 62 78 1 47 29 99 65 96 23 54 66 72 21 60 80 89 40 51 16 25 81 81 51 32 94 99 1
5 20 8 59 4 47 27 11 77 50 87 31 94 33 85 56 76 59 89 30 36 23 45 3 48 99 42 93 72 0 12 90 91 67
[45, 5]
[73, 16]
[36, 4]
[7, 66]
[10, 40]
[14, 11]
[27, 24]
[50, 11]
[11, 62]
[47, 29]
[16, 25]
[51, 32]
[20, 8]
[27, 11]
[3, 48]
[0, 12]
```

2.2. 1 program

2.2.1. Analiza problemu

Pierwszym rozwiązaniem narzucającym się jest przejście przez nasz ciąg liczb i sumowanie elementów sąsiadujących i porównywanie ich z elementem następującym po nich. Pary liczb można przechować w tablicach 2-wymiarowych. Aspekty które trzeba rozważyć to wielkość tablicy danych wejścia, która powinna mieć przynajmniej 2 elementy. Najwięcej podciągów w najlepszym przypadku mozemy znaleźć n-2, gdzie n jest wielkością(libczą elementów w tablicy wejścia). Dodatkową zmienną która będzie nam potrzebna nazwiemy licznik, posłuży ona nam do zliczania par, i ich ewentualnego wypisania.

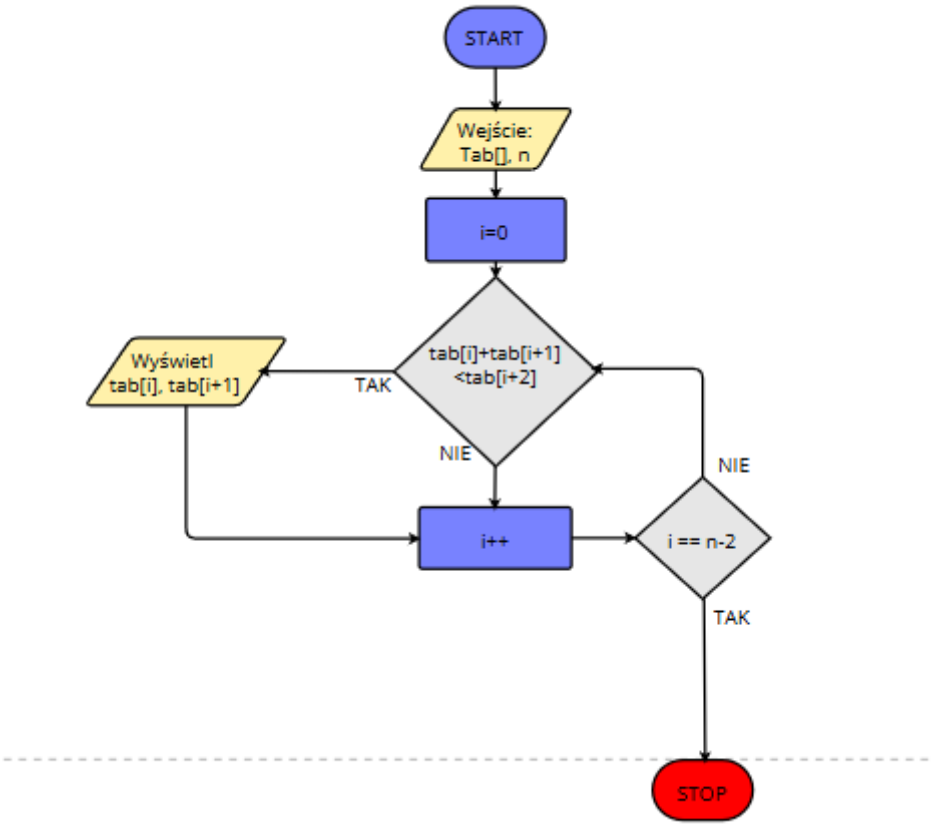
Dane wejścia

- struktura danych typu tablica przechowująca nasz ciąg liczb
- rozmiar tablicy

Dane wyjścia

- vector par

2.2.2. Schemat blokowy



2.2.3. Algorytm zapisany w pseudokodzie

input: tab

n

for i:=0 i<n-2 i++

if (tab[i]+tab[i+1]<tab[i+2])

wyswietl tab[i], tab[i+1]

2.2.4. Teoretyczne oszacowanie złożoności obliczeniowej

Algorytm wykonuje pętlę for od 0 do n-2 (czyli dla n elementów w tablicy, pętla iteruje n-2 razy). W każdej iteracji wykonuje operacje dodawania i porównania (które są operacjami o stałej złożoności). Kluczowe jest tutaj jednak to, że operacje w pętli są wykonywane liniowo. Co daje nam złożoność obliczeniową O(n). Jednak w porównaniu do porzedniego programu przechodzi przez tablice o polowe mniej.

2.2.5. Implementacja programu w C++

```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <vector>
using namespace std;

void wyszukaj(int tab[], int n) {
    for (int i = 0; i < n - 2; i++) {
        if (tab[i] + tab[i + 1] < tab[i + 2]) {
            cout << "[" << tab[i] << ", " << tab[i + 1] << "]\n";
        }
    }
}
int main() {


    const int size = 100;
    int tab[size];
    srand(time(NULL));
    for (int i = 0; i < size; i++) {
        tab[i] = rand()%100;
    }
    for (int i = 0; i < size; i++) {
        cout<<tab[i]<<" ";
    }
    cout << endl;
    wyszukaj(tab, size);

    return 0;
```

2.2.6. Przykładowe wyniki

```
4 63 73 19 26 83 11 24 59 96 24 12 97 31 51 12 5 11 38 12 48 99 44 65 74 9 79 74 18 29 48 16 50 58
 65 47 11 70 74 69 16 37 27 69 15 48 93 77 16 93 3 59 85 74 39 30 61 49 85 66 25 87 75 61 48 86 34
 30 26 47 24 27 55 72 22 79 13 64 19 31 95 45 42 21 64 87 5 30 59 89 71 64 88 61 3 24 31 83 69 32
[4, 63]
[19, 26]
[11, 24]
[24, 59]
[24, 12]
[5, 11]
[12, 48]
[18, 29]
[47, 11]
[37, 27]
[15, 48]
[3, 59]
[24, 27]
[19, 31]
[42, 21]
[21, 64]
[5, 30]
[3, 24]
[24, 31]
```

```
36 36 78 78 65 69 19 64 2 35 23 39 93 40 31 32 38 82 47 96 48 33 52 82 9 50 58 99 91 77 7 97 13
6 49 18 29 96 4 29 1 42 38 62 45 30 82 74 5 35 47 76 7 77 99 38 27 4 64 55 29 29 53 15 3 36 8 32
 52 78 19 66 20 93 96 81 73 74 73 69 65 66 92 95 37 93 99 46 69 78 60 88 53 83 11 2 16 55 52 93
[36, 36]
[23, 39]
[32, 38]
[77, 7]
[13, 6]
[18, 29]
[29, 1]
[45, 30]
[5, 35]
[7, 77]
[27, 4]
[15, 3]
[8, 32]
[66, 20]
[11, 2]
[2, 16]
```

2.3. Testy wydajności algorytmów - sprawdzenie złożoności czasowej

```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <vector>
#include <chrono>
using namespace std;

void wyszukaj(int tab[], int n) {
    for (int i = 0; i < n - 2; i++) {
        if (tab[i] + tab[i + 1] < tab[i + 2]) {
            // Komentarz: cout zablokowany, aby pomiar czasu był bardziej miarodajny
            //cout << "[" << tab[i] << ", " << tab[i + 1] << "]\n";
        }
    }
}

void wyszukaj2(int tab[], int n) {
    int* sumy = new int[n - 2];
    for (int i = 0; i < n - 2; i++) {
        sumy[i] = tab[i] + tab[i + 1];
    }
    for (int i = 2; i < n; i++) {
        if (sumy[i - 2] < tab[i]) {
            // Komentarz: cout zablokowany, aby pomiar czasu był bardziej miarodajny
            //cout << "[" << tab[i - 2] << ", " << tab[i - 1] << "]\n";
        }
    }
    delete[] sumy;  // Zwolnienie pamięci
}

void zmierzCzas(void (*fun)(int*, int), int tab[], int n) {
    auto start = chrono::high_resolution_clock::now();
    fun(tab, n);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> czas_trwania = end - start;
    cout << "Czas: " << czas_trwania.count() << " sekund\n";
}

int main() {
    const vector<int> sizes = { 40000, 60000, 100000, 150000, 500000, 1000000,10000000 };

    // Wylosowanie liczb pseudolosowych jeden raz
    srand(time(NULL));

    for (int size : sizes) {
        int* tab = new int[size];

        for (int i = 0; i < size; i++) {
            tab[i] = rand() % 101;
        }

        cout << "\nWielkość tablicy: " << size << "\n";

        cout << "Funkcja wyszukaj:\n";
        zmierzCzas(wyszukaj, tab, size);

        cout << "Funkcja wyszukaj2:\n";
        zmierzCzas(wyszukaj2, tab, size);

        delete[] tab;  // Zwolnienie pamięci
    }

    return 0;
}
```

L.p	N	f1	f2
1	40000	2.97e-05	0.00030204
2	60000	5.537e-05	0.00045366
3	100000	7.372e-05	0.00061066
4	150000	0.00010689	0.00084854
5	500000	0.00040771	0.0030164
6	1000000	0.00074981	0.0052392
7	10000000	0.00726241	0.0488528