WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Radosław Szynal

Sprawozdanie do projektu

dr inż. Mariusz Borkowski prof. PRz

Rzeszów, 2025

Spis treści

1. Treść zadania

Dla zadanej tablicy liczb całkowitych wypisz wszystkie pod tablice, których suma wynosi 0.

Wejście:

[3, 4, -7, 3, 1, 3, 1, -4, -2, -2]

Wyjście

Istnieją podtablice, których suma wynosi 0.

Tablice te to:

$[3, 4, -7]$

$[4, -7, 3]$

$[-7, 3, 1, 3]$

$[3, 1, -4]$

$[3, 1, 3, 1, -4, -2, -2]$

$[3, 4, -7, 3, 1, 3, 1, -4, -2, -2]$

2. Etapy rozwiązywania problemu

1) Analiza problemu.
2) Rozpisanie i analiza rozpisanego problemu na kartce.
3) Stworzenie schematu blokowego i pseudokodu.
4) Testy "ołówkowe".
5) Próba teoretycznego oszacowania złożoności.
6) Ponowna analiza problemu i próba znalezienia wydajniejszego kodu.
7) Stworzenie schematu blokowego i pseudokodu.
8) Testy "ołówkowe".
9) Próba teoretycznego oszacowania złożoności.
10) Stworzenie wykresów.
11) Implementacja podejścia "brute force".
12) Implementacja drugiej wersji.
13) Przeprowadzenie testów jednostkowych.

2.1. Rozwiązywanie problemu – pierwsze podejście

2.1.1. Analiza problemu

Program ma za zadanie znaleźć wszystkie podciągi tablicy wejściowej, które sumują się do zera. Przy pierwszym podejściu do rozwiązania problemów postanowiłem zaimplementować dwie pętle w sobie, jedna przesuwająca początek zakresu i drugą zmieniającą jego koniec. W drugiej pętli na koniec, żeby kod robił to co ma zlecone trzeba było dodać odpowiednie if'y sprawdzające, czy suma kolejnych elementów jest zerem i je wyświetlić.

Dane wejściowe:

Danymi wejściowymi algorytmu są:

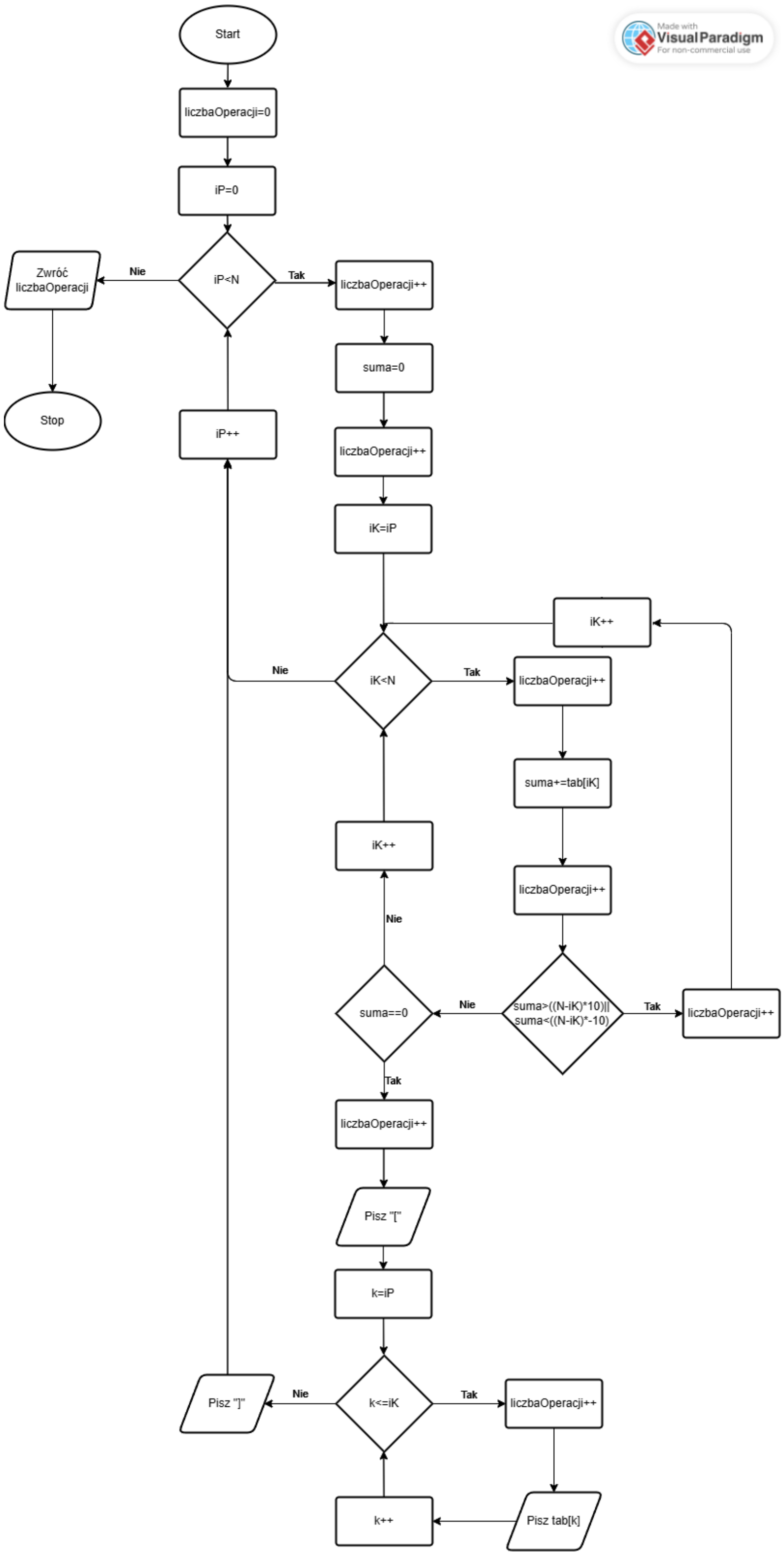
- struktura danych typu tablica (zmienna tab), przechowująca wartości zadanego ciągu.

Dane wyjściowe:

Algorytm zwraca wartość licznika operacji.

2.1.2. Schemat blokowy i pseudokod

Algorytm napisany w postaci schematu blokowego prezentowałby się następująco:



Rysunek 2.1: Schemat blokowy kodu "brute force"

Natomiast kod w pseudokodzie prezentuje się następująco:

```
liczbaOperacji = 0 // Inicjalizacja licznika operacji
Dla iP = 0 do N - 1:
    liczbaOperacji = liczbaOperacji + 1
    suma = 0  // Inicjalizacja sumy dla podciągu
    liczbaOperacji = liczbaOperacji + 1
    Dla iK = iP do N - 1:
        liczbaOperacji = liczbaOperacji + 1
        suma = suma + tab[iK]  // Sumowanie elementów podciągu
        liczbaOperacji = liczbaOperacji + 1
        Jeżeli suma > (N - iK) * 10 lub suma < (N - iK) * -10:
            liczbaOperacji = liczbaOperacji + 1
            Zakończ wewnętrzną pętlę  // Break
        Inaczej Jeżeli suma == 0:
            liczbaOperacji = liczbaOperacji + 1
            Wypisz element podciągu od iP do iK
            Wypisz nową linię
    Zwróć liczbaOperacji  // Zwróć liczbę operacji wykonanych w algorytmie
```

2.1.3. Testy "ołówkowe"algorytmu

Przeprowadźmy teraz testy ołówkowe naszego kodu:

iP	iK	tab[iK]	suma	suma > (N - iK) * 10	suma < (N - iK) * -10	suma == 0	liczbaOperacji
0	0	3	3	Nie	Nie	Nie	2
0	1	4	7	Nie	Nie	Nie	4
0	2	-7	0	Nie	Nie	Tak	6
0	3	3	3	Nie	Nie	Nie	8
0	4	1	4	Nie	Nie	Nie	10
1	1	4	4	Nie	Nie	Nie	12
1	2	-7	-3	Nie	Nie	Nie	14
1	3	3	0	Nie	Nie	Tak	16
1	4	1	1	Nie	Nie	Nie	18
2	2	-7	-7	Nie	Nie	Nie	20
2	3	3	-4	Nie	Nie	Nie	22
2	4	1	-3	Nie	Nie	Nie	24

Rysunek 2.2: Tabela przedstawiające wynik testów "ołówkowych"

2.1.4. Szacowanie złożoności

Przeprowadźmy teraz teoretyczne szacowanie złożoności obliczeniowej:

1) Pierwsza pętla:

   Pętla iteruje wszystkie możliwe wartości iP od 0 do N. Liczba iteracji wynosi dokładnie N. Złożoność tej pętli wynosi *O-(N)*.

2) Druga pętla:

   Dla każdej wartości iP, druga pętla iteruje od iP do N. Liczba iteracji zmniejsza się z każdą wartością iP (od N do 1), co prowadzi do łącznej liczby iteracji równej $\frac{N(N+1)}{2}$. Złożoność tej pętli wynosi $O(N^2)$.

3) Trzecia pętla:

   Wewnątrz drugiej pętli, jeśli zostaną spełnione odpowiednie warunki, trzecia pętla iteruje od iP do iK, wypisując wyniki. Liczba iteracji tej pętli w najgorszym przypadku wynosi N dla każdej pary iP i iK

W najlepszym przypadku kod ma złożoność $O(N^2)$, kiedy większość operacji w drugiej pętli jest przerywana $break'em$ albo suma nigdy nie osiąga wartości 0.

W najgorszym przypadku kod ma złożoność $O(N^3)$, kiedy warunek $suma == 0$ będzie spełniany cały czas i trzecia pętla będzie odpalana za każdym przejściem pętli drugiej.

Powyższe obliczenia odnosiły się do złożoności czasowej. Jeśli chodzi o złożoność przestrzenną to wynosi ona $O(1)$, ponieważ nie wykorzystujemy dynamicznej alokacji pamięci.

2.2. Rozwiązanie problemu - drugie podejście

2.2.1. Analiza problemu

Po kolejnym przemyśleniu problemu, wpadłem na pomysł wykorzystania tablicy pomocniczej, która będzie przechowywać sumy wartości kolejnych indeksów od 0 do N, dzięki czemu później wystarczy odjąć jeden podciąg od drugiego, żeby uzyskać dowolny inny podciąg. Więc w drugiej wersji kodu, dodajemy pomocniczą tablicę przechowującą sumy podciągów.

Dane wejściowe:

Danymi wejściowymi algorytmu są:

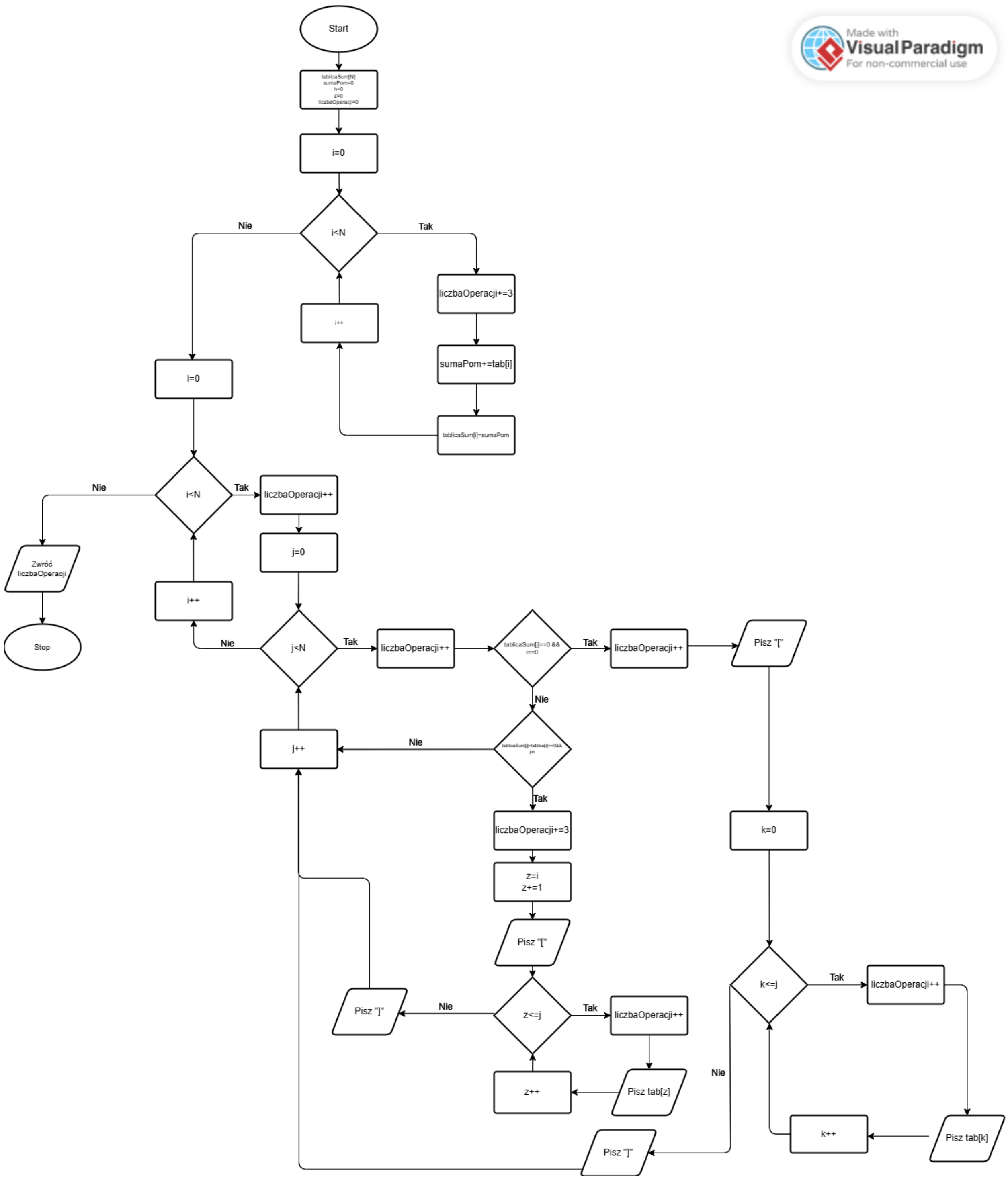
- struktura danych typu tablica (zmienna tab), przechowująca wartości zadanego ciągu.

Dane wyjściowe:

Algorytm zwraca wartość licznika operacji.

2.2.2. Schemat blokowy i pseudokod

Algorytm napisany w postaci schematu blokowego prezentowałby się następująco:



Rysunek 2.3: Schemat blokowy kodu w drugiej wersji

Natomiast kod w pseudokodzie prezentuje się następująco:

```
Funkcja WyszukiwanieDrugaWersja(tab):
    Inicjalizuj tablicaSum jako tablica o rozmiarze N
    sumaPom = 0
    liczbaOperacji = 0
    Dla i = 0 do N - 1:
        liczbaOperacji = liczbaOperacji + 1
        sumaPom = sumaPom + tab[i]  // Sumowanie elementów tablicy
       liczbaOperacji = liczbaOperacji + 1
        tablicaSum[i] = sumaPom  // Przechowywanie sumy w tablicaSum
        liczbaOperacji = liczbaOperacji + 1
     Wypisz nową linię
    // Szukanie podciągów sumujących się do 0
     Dla i = 0 do N - 1:
        liczbaOperacji = liczbaOperacji + 1
        Dla j = i do N - 1:
            liczbaOperacji = liczbaOperacji + 1
            Jeżeli tablicaSum[j] == 0 i i == 0:
                liczbaOperacji = liczbaOperacji + 1
               Wypisz podciąg od 0 do j
                Wypisz nową linię
            Inaczej Jeżeli tablicaSum[j] - tablicaSum[i] == 0 i j > i:
               liczbaOperacji = liczbaOperacji + 1
                z = i + 1  // Ustawienie początkowego indeksu dla wypisania
               liczbaOperacji = liczbaOperacji + 1
                Wypisz podciąg od z do j
                Wypisz nową linię
     Zwróć liczbaOperacji  // Zwrócenie liczby wykonanych operacji
```

2.2.3. Testy "ołówkowe"algorytmu

Przeprowadźmy teraz testy ołówkowe naszego kodu:

i	j	tablicaSum[j]	tablicaSum[j] - tablicaSum[i]	Czy spełniony warunek?	Wynik/Drukowanie
0	0	3	-	Nie	Brak
0	1	7	-	Nie	Brak
0	2	0	-	Tak (tablicaSum[j] == 0)	[3, 4, -7]
0	3	3	-	Nie	Brak
0	4	4	-	Nie	Brak
1	1	7	-	Nie	Brak
1	2	0	0 (tablicaSum[j] - tablicaSum[i] == 0)	Tak	[-7]
1	3	3	-	Nie	Brak
1	4	4	-	Nie	Brak
2	2	0	-	Nie	Brak
2	3	3	3 (tablicaSum[j] - tablicaSum[i] == 3)	Nie	Brak
2	4	4	4 (tablicaSum[j] - tablicaSum[i] == 4)	Nie	Brak

Rysunek 2.4: Tabela przedstawiające wynik testów "ołówkowych"

2.2.4. Szacowanie złożoności

Przeprowadźmy teraz teoretyczne szacowanie złożoności obliczeniowej:

1) Pierwsza pętla(zliczająca sumy pomocnicze):

   Pętla iteruje wszystkie możliwe wartości iP od 0 do N. Liczba iteracji wynosi dokładnie N. Złożoność tej pętli wynosi *O-(N)*.

2) Druga pętla(wyszukująca podciągi):

   - Zewnętrzna pętla iteruje po wszystkich elementach tablicy, co daje złożoność $0(N)$).
   - Wewnętrzna pętla dla każdego elementu zewnętrznej pętli również iteruje po elementach tablicy, co daje złożoność O(N) dla każdej iteracji zewnętrznej pętli.

   Więc cała część związana z tymi dwoma pętlami ma złożoność $0(N^2)$.

3) Trzecia pętla:

   W najgorszym przypadku wykonuje się N ilość razy.

W najlepszym przypadku kod ma złożoność $O(N^2)$, kiedy większość operacji w drugiej pętli jest przerywana $break'em$ albo suma nigdy nie osiąga wartości 0.

W najlepszym przypadku kod ma złożoność $O(N^3)$, kiedy warunek $suma == 0$ będzie spełniany cały czas i trzecia pętla będzie odpalana za każdym przejściem pętli drugiej.

Powyższe obliczenia odnosiły się do złożoności czasowej. Jeśli chodzi o złożoność przestrzenną to wynosi ona $O(N)$, wynika to z tego, że w algorytmie używamy pomocniczej tablicy, która przechowuje N zmiennych, zależnych od wejściowych parametrów.

Jest to bardzo podobny wynik jak w przypadku poprzedniego algorytmu jednak ten wykonuje mniejszą ilość operacji, co będzie pokazane na wykresach w dalszej części.

2.3. Implementacja algorytmów

2.3.1. Kod w wersji brute force

```
long long int wyszukiwanieBruteForce(int suma,int iP,int iK,int tab[N]) {
    long long int liczbaOperacji = 0; // Zmienna zliczająca operacje
    for(iP=0; iP<N; iP++) { //Pierwsza pętla zmieniająca wyszukiwany początek
        liczbaOperacji++;
        suma=0;
        liczbaOperacji++;
        for(iK=iP; iK<N; iK++) {//Drugą pętla zmieniająca szukany koniec
            liczbaOperacji++;
            suma+=tab[iK];  //Sumowanie kolejnych wartości
            liczbaOperacji++;
            if(suma>((N-iK)*10) || suma<((N-iK)*-10)) {
                liczbaOperacji++;
                break;
            } else if(suma==0) { // Sprawdzanie, czy wyszukane podciągi spełniają waruneki
                liczbaOperacji++;
                cout<<"[";
                for(int k=iP; k<=iK; k++) {
                    liczbaOperacji++;
                    cout<<setw(4)<<tab[k]<<","; // Wypisywanie
                }
                cout<<"]";
                cout<<endl;
            }

        }

    }
    return liczbaOperacji;
}
```

Rysunek 2.5: Kod w wersji "brute force"

2.3.2. Kod w wersji drugiej

```
long long int wyszukiwanieDrugaWersja(int tab[N]) {

    int tablicaSum[N];
    int sumaPom=0,h=0,z=0;
    long long int liczbaOperacji=0;

    //Generowanie tablicy pomocniczej w której przechowujemy wartości zsumowanych liczb
    for(int i=0; i<N; i++) {
        liczbaOperacji++;
        sumaPom+=tab[i];
        liczbaOperacji++;
        tablicaSum[i]=sumaPom;
        liczbaOperacji++;
    }
    cout<<endl;
    for(int i=0; i<N; i++) {
        liczbaOperacji++;
        for(int j=i; j<N; j++) {
            liczbaOperacji++;
            if(tablicaSum[j]==0 && i==0) {
                liczbaOperacji++;
                cout<<"[ ";
                for(int k=0; k<=j; k++) {
                    liczbaOperacji++;
                    cout<<setw(4)<<tab[k]<<",";
                }
                cout<<" ]";
                cout<<endl;
            } else if(tablicaSum[j]-tablicaSum[i]==0 &&  j>i) {
                liczbaOperacji++;
                z=i;
                liczbaOperacji++;
                z+=1;
                liczbaOperacji++;
                cout<<"[ ";
                for(z; z<=j; z++) {
                    cout<<setw(4)<<tab[z]<<",";
                    liczbaOperacji++;
                }
                cout<<" ]";
                cout<<endl;
            }
        }
    }
    return liczbaOperacji;
}
```

Rysunek 2.6: Kod w wersji drugiej

2.4. Testy algorytmów

2.4.1. Tabele

W celu porównania obu algorytmów przeprowadziłem testy odpalając oba algorytmy na takich samych danych i porównując ich wyniki. Prezentują się one następująco:

n	w1: czas (s)	w2: czas (s)
80	0.0948	0.1204
100	0.1296	0.1468
150	0.9028	0.9252
200	1.1390	1.1052
5000	0.0800	0.0866
7500	0.1766	0.1748
10000	0.4162	0.4086
15000	1.1796	1.1550
20000	2.4350	2.1694

(a) Tabela czasu

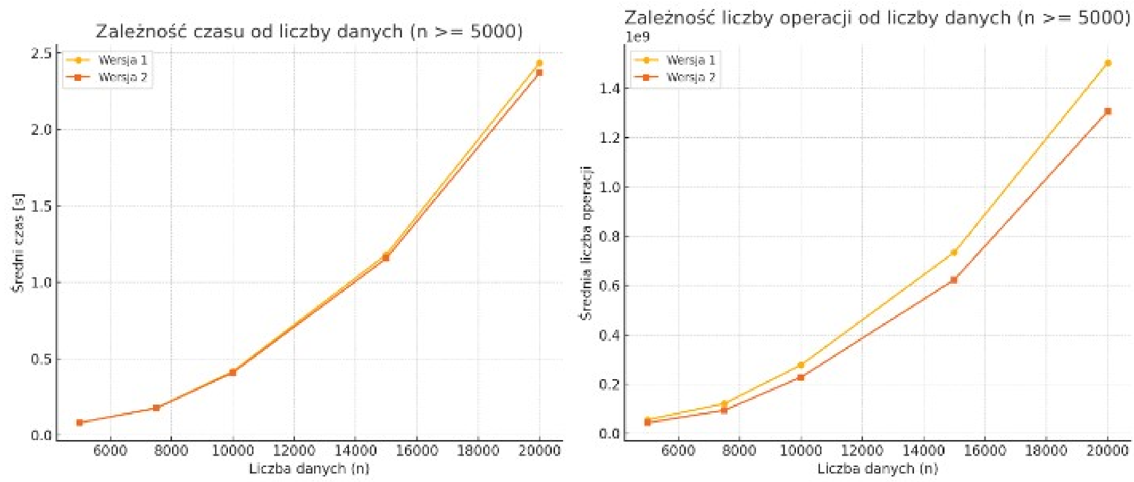
n	w1: liczba operacji	w2: liczba operacji
80	7291.6	4454.4
100	10976.4	6674.0
150	30019.0	19791.6
200	49339.2	30675.8
5000	55158370.4	42966606.4
7500	120681743.0	93713671.4
10000	277354641.2	227934971.0
15000	673840921.2	623049677.2
20000	1508369737.2	1301876717.8

(b) Tabela liczby operacji

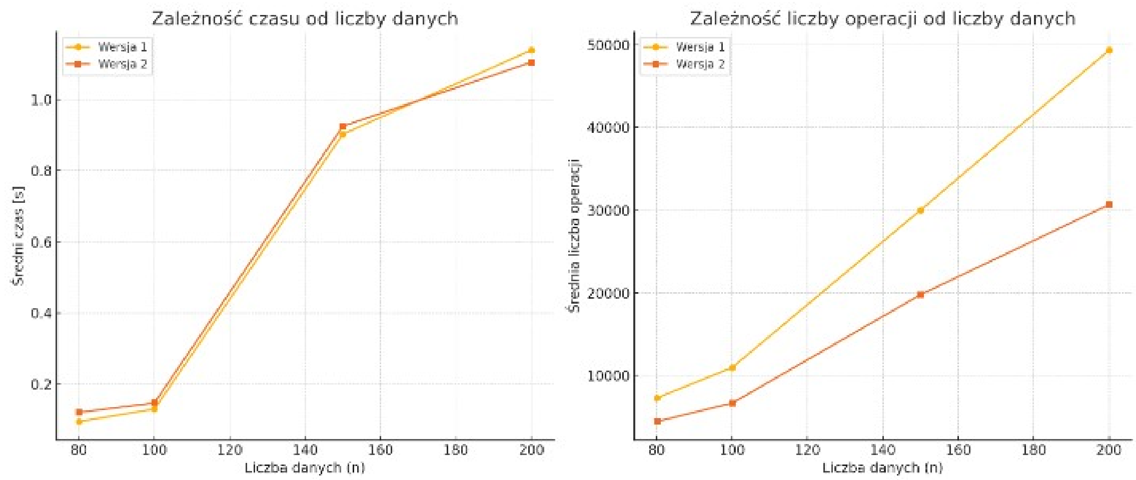
Rysunek 2.7: Porównanie tabel

2.4.2. Wykresy

Wykresy prezentujące dane z powyższych tabel:



(a) Wykres zależności czasu i ilości obliczeń (bez wyświetlania)



(b) Wykres zależności czasu i ilości obliczeń (z wyświetlaniem)

Rysunek 2.8: Porównanie wykresów zależności czasu i obliczeń

2.5. Podsumowanie

Kod w obu wersjach działa poprawnie i wyszukuje podciągi sumujące się do zera. Porównując oba programy możemy zauważyć, że są bardzo podobne do siebie natomiast przy zwiękaszających się ilościach danych wejściowych, możemy zauważyć, że na prowadzenie wychodzi drugi sposób, których wraz ze wzrostem liczby danych, wolniej od pierwszego algorytmu, zwiększa liczbę wykonywanych operacji. Podobną zależność możemy zauważyć na wykresach czasowych, natomiast wyniki tam nie są aż tak spektakularne. Myślę, że dopiero jeszcze bardziej zwiększając testowane dane byłoby lepiej, widoczne, natomiast już na takim zakresie widzimy pewną tendencję.

2.6. Appendix

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <time.h>

using namespace std;

const int N=20000;

//Deklaracja funkcji
long long int wyszukiwanieBruteForce(int suma,int iP,int iK,int tab[N]);
long long int wyszukiwanieDrugaWersja(int tab[N]);

int main(int argc, char** argv) {

    srand (time(NULL));

    //Deklarowanie zmiennych
    int tab[N];
    int iP=0,iK=0,suma=0;
    double wynik1=0.0,wynik2=0.0;
    long long int liczbaOperacji1,liczbaOperacji2;

    //Generowanie tablicy losowych liczb z przedziału od -10 do 10
    for(int i=0; i<N; i++) {
        tab[i]=rand()%21-10;
    }

    //Wyświetlanie tablicy
    for(int i=0; i<N; i++) cout<<" "<<tab[i];
    cout<<endl;


    //Wywołanie funkcji brute force
    clock_t start = clock();
    liczbaOperacji1=wyszukiwanieBruteForce(suma,iP,iK,tab);
    clock_t end = clock();
    wynik1=double(end-start)/CLOCKS_PER_SEC;
    cout<<endl<<"Drugie rozwiazanie:"<<endl;
    clock_t start1 = clock();
    liczbaOperacji2=wyszukiwanieDrugaWersja(tab);
    clock_t end1 = clock();
    wynik2  = double(end1-start1)/CLOCKS_PER_SEC;
    cout<<"Czas poswiecony na wyszukiwanie w wersji podstawowej to: "<<wynik1<<", a liczba wykoanych operacji to: "<<liczbaOperacji1<<endl;
    cout<<"Czas poswiecony na wyszukiwanie w wersji drugiej to: "<<wynik2<<", a liczba wykoanych operacji to: "<<liczbaOperacji2;
    return 0;
}
```

Rysunek 2.9: Kod cz.1

```
//Uzupełnienie zadeklarowanej wcześniej funkcji
long long int wyszukiwanieBruteForce(int suma,int iP,int iK,int tab[N]) {
    long long int liczbaOperacji = 0; // Zmienna zliczająca operacje
    for(iP=0; iP<N; iP++) { //Pierwsza pętla zmieniająca wyszukiwany początek
        liczbaOperacji++;
        suma=0;
        liczbaOperacji++;
        for(iK=iP; iK<N; iK++) {//Drugą pętla zmieniająca szukany koniec
            liczbaOperacji++;
            suma+=tab[iK];  //Sumowanie kolejnych wartości
            liczbaOperacji++;
            if(suma>((N-iK)*10) || suma<((N-iK)*-10)) {
                liczbaOperacji++;
                break;
            } else if(suma==0) { // Sprawdzanie, czy wyszukane podciągi spełniają waruneki
                liczbaOperacji++;
                cout<<"[";
                for(int k=iP; k<=iK; k++) {
                    liczbaOperacji++;
                    cout<<setw(4)<<tab[k]<<","; // Wypisywanie
                }
                cout<<"]";
                cout<<endl;
            }

        }

    }
    return liczbaOperacji;
}
```

Rysunek 2.10: Kod cz.2

```
long long int wyszukiwanieDrugaWersja(int tab[N]) {

    int tablicaSum[N];
    int sumaPom=0,h=0,z=0;
    long long int liczbaOperacji=0;

    //Generowanie tablicy pomocniczej w której przechowujemy wartości zsumowanych liczb
    for(int i=0; i<N; i++) {
        liczbaOperacji++;
        sumaPom+=tab[i];
        liczbaOperacji++;
        tablicaSum[i]=sumaPom;
        liczbaOperacji++;
    }
    cout<<endl;
    for(int i=0; i<N; i++) {
        liczbaOperacji++;
        for(int j=i; j<N; j++) {
            liczbaOperacji++;
            if(tablicaSum[j]==0 && i==0) {
                liczbaOperacji++;
                cout<<"[ ";
                for(int k=0; k<=j; k++) {
                    liczbaOperacji++;
                    cout<<setw(4)<<tab[k]<<",";
                }
                cout<<" ]";
                cout<<endl;
            } else if(tablicaSum[j]-tablicaSum[i]==0 &&  j>i) {
                liczbaOperacji++;
                z=i;
                liczbaOperacji++;
                z+=1;
                liczbaOperacji++;
                cout<<"[ ";
                for(z; z<=j; z++) {
                    cout<<setw(4)<<tab[z]<<",";
                    liczbaOperacji++;
                }
                cout<<" ]";
                cout<<endl;
            }
        }
    }
    return liczbaOperacji;
}
```

Rysunek 2.11: Kod cz.3