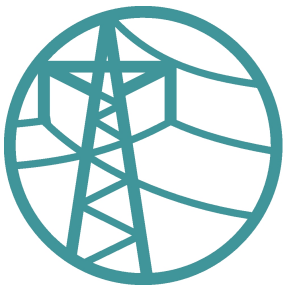


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Dawid Kurzępa 179946

Dla zadanego ciągu liczb naturalnych, znajdź długość najkrótszego ciągu, którego suma jest większa niż zadana liczba.

ZADANIE PROGRAMISTYCZNE

Opiekun pracy:
dr inż. Mariusz Borkowski prof. PRz

Rzeszów, 2025

Spis treści

1. Wstęp

1.1. Cel

Celem zadania jest opracowanie, implementacja oraz analiza algorytmów znajdujących najkrótszy podciąg, którego suma jest większa niż zadana liczba.

1.2. Zakres problemu

Zadanie obejmuje trzy podejścia algorytmiczne rozwiązujące zadany problem oraz ich analizę.

1.3. Omówienie metod

W pracy wykorzystano trzy metody rozwiązania problemu : Sortowanie i sumowanie największych liczb, Sliding Window (okno przesuwne), Dynamic Programming (brute-force optymalizowane)

2. Opis problemu

2.1. Przedstawienie problemu

Dla danego ciągu liczb naturalnych oraz liczby, należy znaleźć najkrótszy ciąg liczb z tego zbioru, którego suma jest większa niż zadana liczba k.

2.2. Dane wejściowe i wyjściowe

Dane wejściowe: Ciąg liczb naturalnych oraz liczba k.

Dane wyjściowe: Najkrótszy ciąg liczb, którego suma jest większa niż liczba k.

2.3. Przykładowe dane i oczekiwany wynik

Wejście: {1, 2, 3, 4, 5, 6, 7, 8}, k = 20, – Wyjście: {6, 7, 8}

Wejście: {1, 2, 3, 4, 5, 6, 7, 8}, k = 7, – Wyjście: {8}

Wejście: {1, 2, 3, 4, 5, 6, 7, 8}, k = 21, – Wyjście: {5, 6, 7, 8}

wejście: {1,2,3,4,5,6,7,8}, k = 40, - Wyjście: Brak odpowiedniego podciągu!

3. Podejścia do rozwiązania problemu

3.1. Sortowanie i sumowanie największych liczb

3.1.1. Opis podejścia

Metoda Sortowanie i sumowanie największych liczb polega na posortowaniu ciągu liczb w kolejności malejącej, a następnie sumowaniu największych elementów, aż suma przekroczy zadaną wartość. Po znalezieniu takiego podciągu, algorytm zwraca jego długość jako rozwiązanie. To podejście jest proste do zaimplementowania, ale wymaga sortowania, co wpływa na jego złożoność obliczeniową.

3.1.2. Podstawy teoretyczne

Liczba wszystkich podciągów dla ciągu o długości n wynosi:

$$2^n \tag{3.1}$$

Metoda sortuje liczby za pomocą algorytmu **Bubble Sort** i sumuje największe, aż przekroczy zadaną wartość. Dzięki temu ogranicza liczbę sprawdzanych elementów, osiągając złożoność $O(n^2)$ przez użycie algorytmu Bubble Sort.

3.1.3. Złożoność czasowa i obliczeniowa

Złożoność czasowa:

- **Sortowanie (Bubble Sort):** Bubble sort ma złożoność czasową $O(n^2)$, ponieważ dla każdego elementu porównuje się go z innymi elementami w tablicy w najgorszym przypadku.
- **Iteracja przez posortowaną listę:** Po posortowaniu listy, iteracja przez elementy listy w celu znalezienia najkrótszego podciągu ma złożoność $O(n)$.

Złożoność całkowita:

- **Złożoność czasowa:** $O(n^2)$ (dominująca część to sortowanie)
- **Złożoność pamięciowa:** $O(n)$, ponieważ potrzebujemy tylko jednej dodatkowej zmiennej do przechowywania sumy i długości podciągu (pomijając samą tablicę liczb).

3.1.4. Wady i zalety

Zalety:

- Prosta koncepcja – metoda polega na posortowaniu liczb i sumowaniu największych z nich, co jest łatwe do zrozumienia.
- Łatwość implementacji – w porównaniu do bardziej skomplikowanych algorytmów, metoda sortowania i sumowania jest stosunkowo łatwa do zaimplementowania.

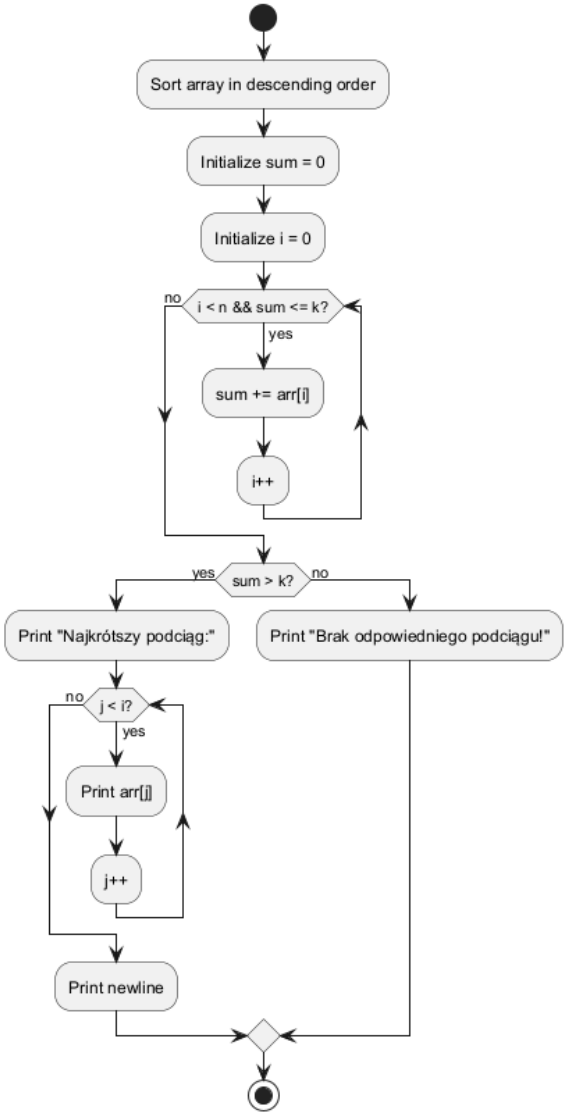
Wady:

- Złożoność czasowa w zależności od algorytmu sortowania – jeśli użyjemy algorytmu o złożoności $O(n^2)$, jak bubble sort, cała metoda będzie miała złożoność $O(n^2)$.
- Może być nieoptymalna, gdy dane wejściowe są już posortowane lub gdy mamy do czynienia z dużymi zestawami danych.
- Sortowanie w celu znalezienia najkrótszego podciągu może być zbędne, jeśli wystarczyłaby inna metoda optymalizacyjna.

3.1.5. Implementacja w pseudokodzie

- Funkcja FindShortestSubsequence:
- Wejście: lista liczb, k
- Posortuj listę liczb w porządku malejącym
- Inicjalizuj sumę na 0
- Inicjalizuj zmienną dla długości podciągu
- Dla każdego elementu w posortowanej liście:
  - Dodaj element do sumy
  - Zwiększ długość podciągu
  - Jeśli suma > k, zakończ
- Jeśli nie znaleziono odpowiedniego podciągu:
- Zwróć komunikat o braku rozwiązania
- Zwróć długość podciągu

3.1.6. Implementacja w schemacie blokowym



Rys. 3.1. schemat blokowy Sortowanie i Sumowanie

3.2. Sliding Window (okno przesuwne)

3.2.1. Opis podejścia

Metoda okna przesuwnego (sliding window) polega na iteracyjnym sprawdzaniu podciągów o rosnącym rozmiarze w danej tablicy, gdzie "okno"przesuwa się po elementach w sposób ciągły. Początkowo okno obejmuje początkowe elementy, a następnie stopniowo rozszerza się lub przesuwa, aż spełni warunki zadania. Jest to efektywna metoda o złożoności czasowej $O(n)$, idealna do rozwiązywania problemów z ciągłymi fragmentami danych.

3.2.2. Podstawy teoretyczne

Metoda okna przesuwnego jest techniką wykorzystywaną do rozwiązywania problemów związanych z przetwarzaniem ciągów danych w sposób efektywny. Główna idea polega na tym, że rozpatrujemy tylko część ciągu, zwaną "oknem", i przesuwamy to okno po ciągu, przy czym w każdym kroku obliczamy wynik dla bieżącego podciągu. Okno ma zazwyczaj stałą wielkość, ale może się zmieniać w zależności od wymagań problemu.

Jeśli mamy ciąg liczb o długości n, to metoda okna przesuwnego pozwala na rozwiązywanie problemów, takich jak znajdowanie największej sumy podciągu o zadanej długości, bez konieczności ponownego przetwarzania całego ciągu w każdym kroku. Działanie algorytmu polega na dodaniu nowego elementu do okna i usunięciu elementu, który wychodzi poza zakres okna. Dzięki temu, operacje są wykonywane w czasie $O(1)$ w każdym kroku, a cała złożoność czasowa algorytmu wynosi $O(n)$.

Metoda okna przesuwnego jest szczególnie użyteczna w problemach, gdzie trzeba obliczać wynik dla wielu podciągów ciągu w sposób efektywny, bez konieczności rozważania każdego możliwego podciągu.

3.2.3. Złożoność czasowa i obliczeniowa

Złożoność czasowa metody okna przesuwnego wynosi $O(n)$, gdzie n to liczba elementów w ciągu danych wejściowych. Jest to możliwe dzięki temu, że w każdym kroku operacje związane z dodaniem nowego elementu do okna i usunięciem elementu wychodzącego z okna wykonują się w czasie stałym $O(1)$. W rezultacie cała metoda wymaga tylko jednego przejścia przez dane wejściowe.

Złożoność obliczeniowa tej metody jest bardzo efektywna, ponieważ jej zapotrzebowanie na pamięć wynosi $O(1)$, co oznacza, że nie potrzebujemy dodatkowych struktur danych o rozmiarze zależnym od wielkości wejścia. Okno przesuwne przetwa-

rza dane w sposób ciągły, zmieniając tylko kilka zmiennych pomocniczych, co zapewnia minimalne zużycie pamięci.

3.2.4. Wady i zalety

Zalety:

- Złożoność czasowa $O(n)$ – bardzo szybka metoda, działająca w czasie liniowym.
- Złożoność pamięciowa $O(1)$ – minimalne zużycie pamięci, ponieważ wykorzystujemy tylko kilka zmiennych pomocniczych.
- Efektywna przy problemach wymagających ciągłych obliczeń dla fragmentów danych.

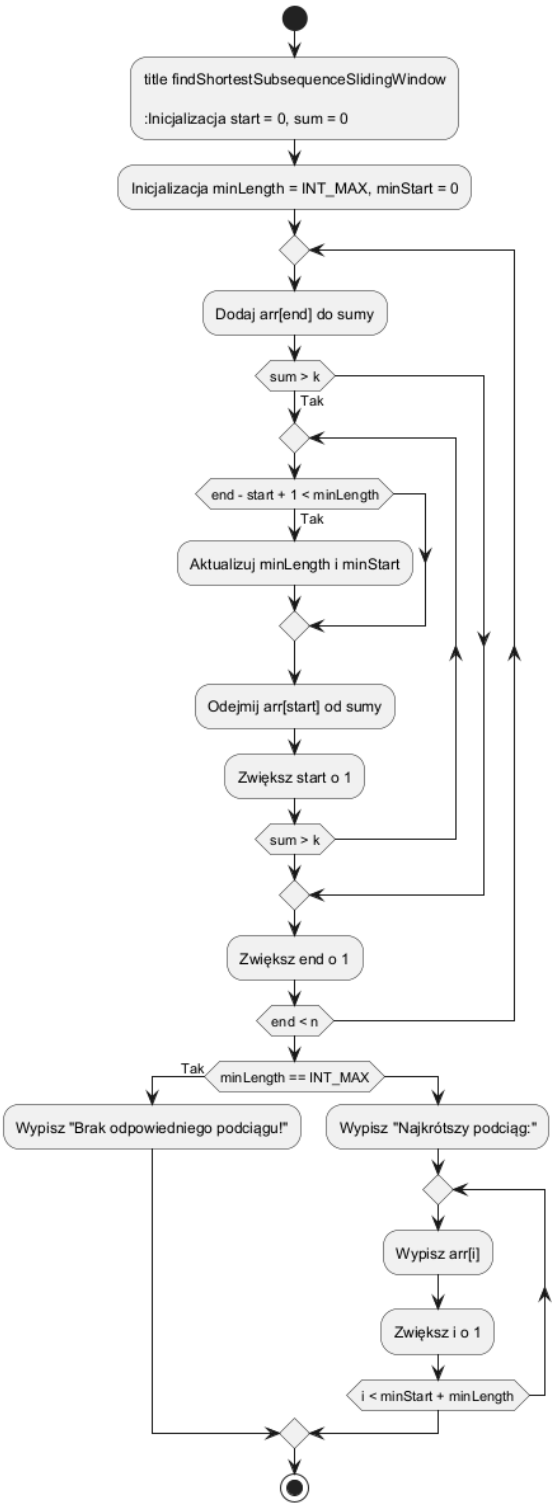
Wady:

- Wymaga przetworzenia danych w sposób iteracyjny, co może być trudne do zaimplementowania w porównaniu do prostszych algorytmów.
- Działa głównie na problemach, w których okno ma stałą lub zmieniającą się wielkość, co może ograniczać zastosowanie w bardziej złożonych przypadkach.

3.2.5. Implementacja w pseudokodzie

- Funkcja FindShortestSubsequence:
- Wejście: lista liczb, k
- Inicjalizuj sumę jako 0
- Inicjalizuj minimalną długość podciągu jako nieskończoność
- Inicjalizuj indeks początkowy podciągu (lewy) jako 0
- Dla każdego elementu (prawy) w liście:
  - Dodaj bieżący element do sumy
  - Dopóki suma jest większa niż k:
    - Zaktualizuj minimalną długość podciągu
    - Odejmij element na pozycji lewy od sumy
    - Zwiększ wartość lewy
- Jeśli nie znaleziono odpowiedniego podciągu:
- Zwróć komunikat o braku rozwiązania
- W przeciwnym razie zwróć minimalną długość podciągu

3.2.6. Implementacja w schemacie blokowym



Rys. 3.2. schemat blokowy Okno przesuwne

3.3. Metoda brute-force optymalizowane

3.3.1. Opis podejścia

Metoda brute-force optymalizowane opiera się na sprawdzeniu wszystkich możliwych podciągów w sposób systematyczny, ale z wykorzystaniem optymalizacji przyspieszających działanie algorytmu. Zamiast sprawdzać każdy podciąg niezależnie, algorytm eliminuje niepotrzebne obliczenia, np. przerywa iterację, gdy suma podciągu przekroczy wartość progową.

3.3.2. Podstawy teoretyczne

W najgorszym przypadku liczba wszystkich podciągów w ciągu o długości n wynosi:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{3.2}$$

Dzięki zastosowaniu optymalizacji, takich jak wcześniejsze zakończenie przetwarzania, rzeczywista liczba analizowanych podciągów jest mniejsza, co poprawia wydajność w porównaniu do klasycznej metody brute-force.

3.3.3. Złożoność czasowa i obliczeniowa

Podstawowa wersja brute-force ma złożoność czasową $O(n^2)$, jednak dzięki wczesnemu zakończeniu analizy niektórych podciągów rzeczywista wydajność jest lepsza w praktyce. Złożoność pamięciowa tej metody wynosi $O(1)$, ponieważ algorytm używa jedynie zmiennych pomocniczych do przechowywania sumy i długości podciągu.

3.3.4. Wady i zalety

Zalety:

- Łatwiejsza implementacja w porównaniu do bardziej złożonych metod.
- Działa dla dowolnych danych wejściowych bez specjalnych założeń.
- Optymalizacja znacząco przyspiesza działanie w porównaniu do klasycznej wersji brute-force.

Wady:

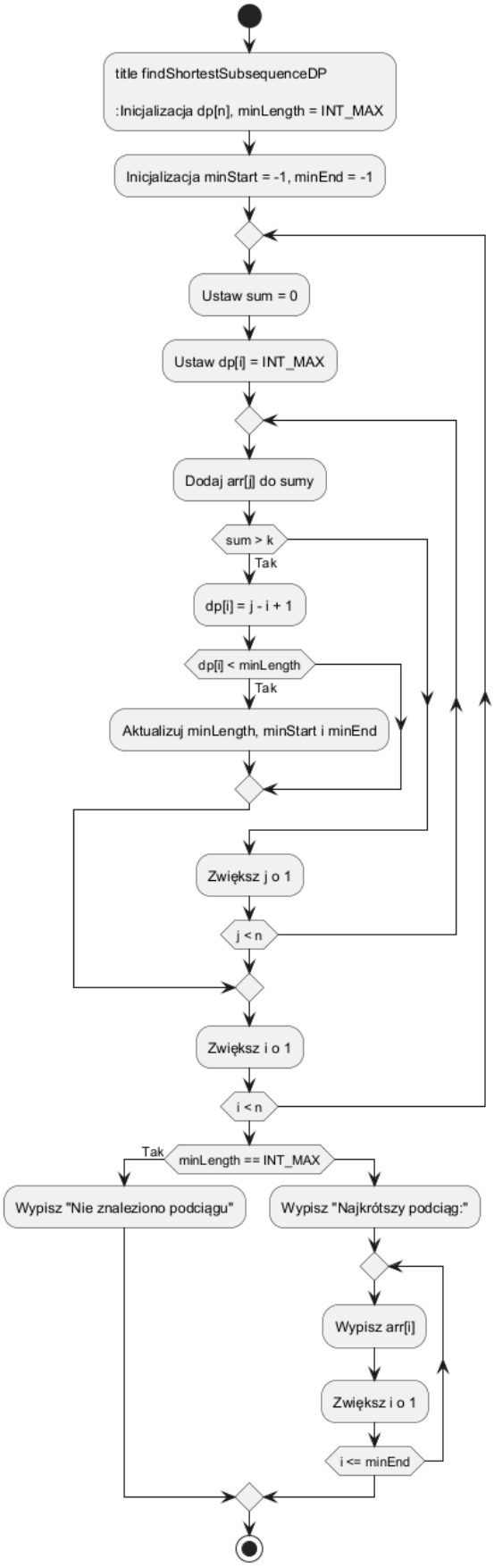
- Nadal gorsza wydajność niż metody oparte na oknie przesuwnym.
- W najgorszym przypadku nadal może wymagać sprawdzenia wszystkich podciągów.

3.3.5. Implementacja w pseudokodzie

- Funkcja FindShortestSubsequence:

- Wejście: lista liczb, k
- Inicjalizuj minimalną długość podciągu jako nieskończoność
- Dla każdego indeksu początkowego w liście:
  - Inicjalizuj sumę jako 0
  - Dla każdego indeksu końcowego od punktu początkowego:
    * Dodaj bieżący element do sumy
    * Jeśli suma $> k$, zaktualizuj minimalną długość i przerwij pętlę
- Jeśli nie znaleziono odpowiedniego podciągu:
- Zwróć komunikat o braku rozwiązania
- W przeciwnym razie zwróć minimalną długość podciągu

3.3.6. Implementacja w schemacie blokowym



Rys. 3.3. Schemat blokowy Brute force

4. Wyniki testów i wykres czasu

4.1. Wyniki testów czasowych

Przykładowy wynik dla tablicy 1-n dla n = 10000 oraz k = 500000

4.1.1. Sortowanie i sumowanie

```
wersja 1 (sortowanie i sumowanie):
Najkrotszy podciąg: 10000 9999 9998 9997 9996 9995 9994 9993 9992 9991 9990 9989 9988 9987 9986 9985 9984 9983 9982 9981
 9980 9979 9978 9977 9976 9975 9974 9973 9972 9971 9970 9969 9968 9967 9966 9965 9964 9963 9962 9961 9960 9959 9958 9957
 9956 9955 9954 9953 9952 9951 9950
Czas wykonania: 205821 mikroS
```

Rys. 4.1. pomiar czasu sortowanie sumowanie

4.1.2. Okno przesuwne

```
wersja 2 (okno przesuwne):
Najkrotszy podciąg: 10000 9999 9998 9997 9996 9995 9994 9993 9992 9991 9990 9989 9988 9987 9986 9985 9984 9983 9982 9981
 9980 9979 9978 9977 9976 9975 9974 9973 9972 9971 9970 9969 9968 9967 9966 9965 9964 9963 9962 9961 9960 9959 9958 9957
 9956 9955 9954 9953 9952 9951 9950
Czas wykonania: 6798 mikroS
```

Rys. 4.2. pomiar czasu okno przesuwne

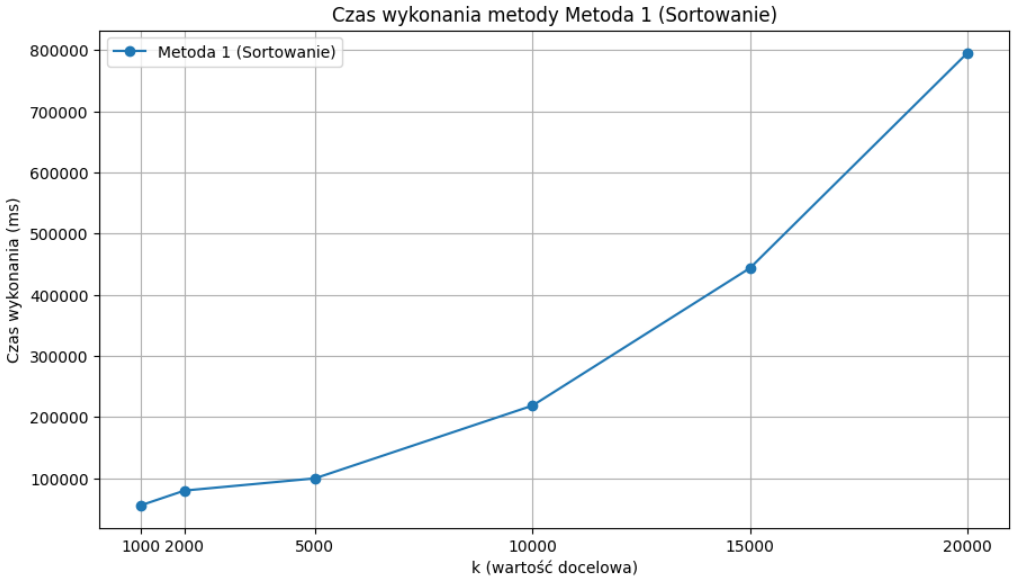
4.1.3. Brute force

```
wersja 3 (brute force):
Najkrotszy podciąg: 10000 9999 9998 9997 9996 9995 9994 9993 9992 9991 9990 9989 9988 9987 9986 9985 9984 9983 9982 9981
 9980 9979 9978 9977 9976 9975 9974 9973 9972 9971 9970 9969 9968 9967 9966 9965 9964 9963 9962 9961 9960 9959 9958 9957
 9956 9955 9954 9953 9952 9951 9950
Czas wykonania: 13268 mikroS
```

Rys. 4.3. pomiar czasu brute force

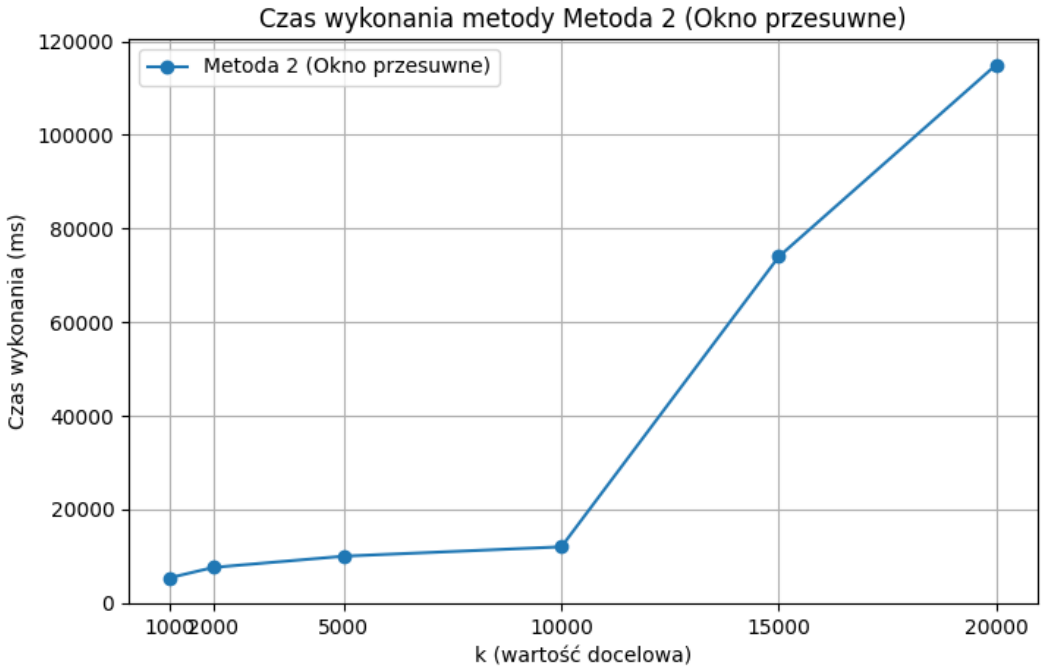
4.2. Wykresy czasu obliczeniowego

4.2.1. Sortowanie i sumowanie



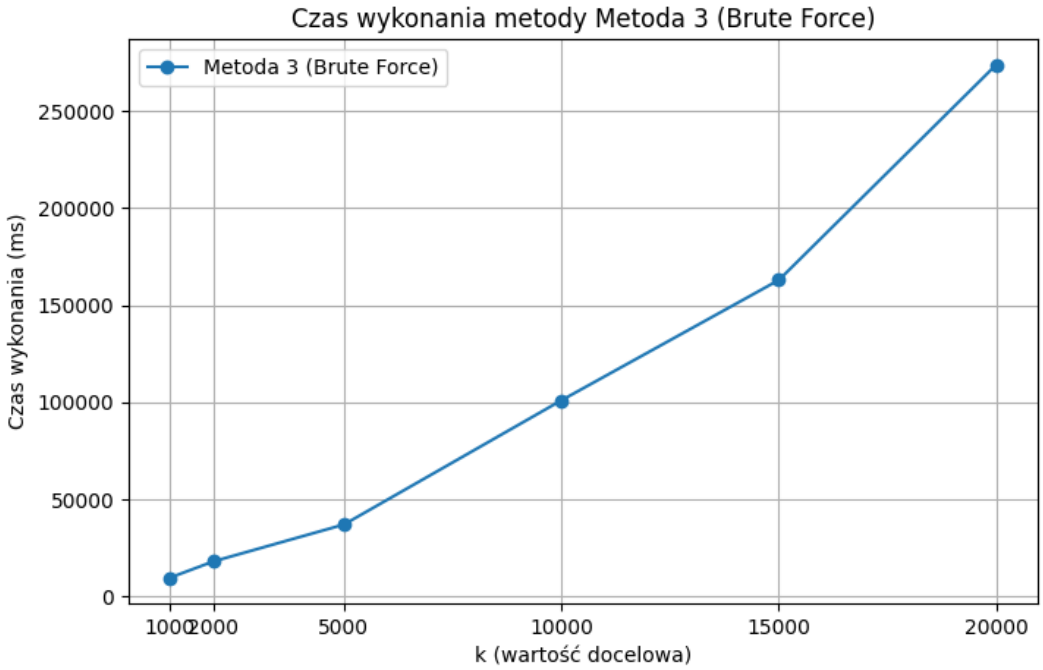
Rys. 4.4. Wykres czasu sortowanie i sumowanie

4.2.2. Okno przesuwne



Rys. 4.5. Wykres czasu okno przesuwne

4.2.3. Brute force



Rys. 4.6. Wykres czasu brute force

5. Wnioski i podsumowanie

- Analizując wykresy z przeprowadzonych badań dla dobranych zakresów wartości widać, że sortowanie wypada najsłabiej niezależnie od wielkości tablicy - nawet jeśli różnice są niewielkie.
- Okno przesuwne sprawdza się najlepiej dla każdych testowanych liczb ze znaczną przewagą
- Metoda brute force jest uśrednieniem czasu potrzebnego do wykonania zadania względem poprzednich prób

6. Załączniki

Listing 6.1. main.cpp

```cpp
#include <iostream>
#include <windows.h>
#include <chrono>
using namespace std;
using namespace std::chrono;

// bubble sort O(n^2)
void sortDescending(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] < arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Funkcja znajdujca najkrtszy  podcig  o sumie wikszej  ni  k
void findShortestSubsequence(int arr[], int n, int k) { // caa funkcja ma  zoono  O(n^2) przez sortowanie, gdyby zastosowa quicksort O(n log n)
    // Wywoanie funkcji sortujcej  tablic  w  porzdku  malejcym
    sortDescending(arr, n); // Funkcja sortujca tablic arr o dugoci n w porzdku malejcym

    // Inicjalizacja zmiennych pomocniczych
    int sum = 0; // Zmienna do przechowywania sumy elementw w podcigu
    int i = 0;   // Indeks, ktry bdzie wskazywa  liczb  elementw  w najkrtszym podcigu

    // Szukamy najkrtszego podcigu o sumie wikszej  ni  k
    // Ptla przetwarza tablic i sumuje elementy, dopki suma nie przekroczy k
    while (i < n && sum <= k) {
        sum += arr[i]; // Dodajemy element z tablicy do sumy
        i++; // Przechodzimy do kolejnego elementu tablicy
    }

    // Sprawdzamy, czy znaleziono podcig
    if (sum > k) {
        // Jeli suma jest wiksza  ni  k, wypisujemy najkrtszy podcig
        cout << "Najkrotszy podciag: ";
        for (int j = 0; j < i; j++) {  // Wypisujemy elementy podcigu
            cout << arr[j] << " "; // Wypisujemy element na pozycji j
        }
        cout << endl; // Koniec wypisywania
    } else {
        // Jeli suma nigdy nie przekroczya k
        cout << "Brak odpowiedniego podciagu!"<< endl;
    }
}

void findShortestSubsequenceSlidingWindow(int arr[], int n, int k) { // caa funkcja ma zoono O(n) - kady element jest przetwarzany maksymalnie raz
    // Inicjalizacja zmiennych pomocniczych
    int start = 0, sum = 0, minLength = INT_MAX, minStart = 0;
    // 'start' wskazuje pocztek okna, 'sum' to suma elementw w oknie
    // 'minLength' to  dugo  najkrtszego  podcigu ,  ktry  spenia  warunek, domylnie  maksymalna  warto  typu int
    // 'minStart' to indeks pocztkowy tego podcigu

    // Przesuwanie okna po tablicy
    for (int end = 0; end < n; end++) {
        sum += arr[end]; // Dodajemy element z pozycji 'end' do sumy okna
```

```
60
61          // Zmniejszamy okno, gdy suma przekroczy warto k
62          while (sum > k) {
63              // Sprawdzamy, czy obecne okno jest najkrtszym, ktre spenia warunek
64              if (end - start + 1 < minLength) {
65                  minLength = end - start + 1; // Aktualizujemy dugo najkrtszego
    podcigu
66                  minStart = start; // Zapamitujemy pocztek tego podcigu
67              }
68
69              // Usuwamy element od pocztku okna (przesuwamy wskanik 'start' w prawo)
70              sum -= arr[start]; // Odejmujemy element, ktry zostaje usunity z okna
71              start++; // Przesuwamy pocztek okna w prawo
72          }
73      }
74
75      // Sprawdzamy, czy znaleziono jakikolwiek podcig
76      if (minLength == INT_MAX) {
77          // Jeli    dugo    pozostaa maksymalna, to znaczy, e nie znaleziono podcigu
78          cout << "Brak odpowiedniego podciagu!" << endl;
79      } else {
80          // Wypisujemy najkrtszy podcig
81          cout << "Najkrotszy podciag: ";
82          for (int i = minStart; i < minStart + minLength; i++) {
83              cout << arr[i] << " "; // Wypisujemy elementy najkrtszego podcigu
84          }
85          cout << endl; // Koniec wypisywania
86      }
87  }
88
89
90  void findShortestSubsequenceDP(int arr[], int n, int k) { // w najgorszym przypadku
        O(n^2)
91      // Tablica do przechowywania minimalnej dugoci podcigu dla danego indeksu
92      int dp[n];
93      int minLength = INT_MAX; // Przechowuje globaln minimaln  dugo  podcigu
         speniajcego  warunek
94      int minStart = -1, minEnd = -1; // Indeksy pocztku i koca znalezionego podcigu
95
96      // Przechodzimy przez wszystkie indeksy i obliczamy najkrtszy podcig
97      for (int i = 0; i < n; i++) {
98          int sum = 0; // Resetujemy sum dla kadego indeksu
99          dp[i] = INT_MAX; // Pocztkowo zakadamy, e nie ma podcigu
100
101         for (int j = i; j < n; j++) {
102             sum += arr[j]; // Dodajemy kolejne liczby do sumy
103
104             if (sum > k) { // Jeli suma przekroczya k
105                 dp[i] = j - i + 1; //  Dugo  podcigu  to  rnica  indeksw  + 1
106
107                 // Sprawdzamy, czy znaleziony podcig jest krtszy ni poprzednie
108                 if (dp[i] < minLength) {
109                     minLength = dp[i];
110                     minStart = i;
111                     minEnd = j;
112                 }
113                 break; // Przerywamy, bo znalelimy minimalny podcig dla tego i
114             }
115         }
116     }
117
118     // Wypisywanie wyniku
119     if (minLength == INT_MAX) {
120         cout << "Nie znaleziono podciagu, ktorego suma jest wiksza niz " << k <<
    endl;
121     } else {
122         cout << "Najkrotszy podciag: ";
123         for (int i = minStart; i <= minEnd; i++) {
124             cout << arr[i] << " ";
125         }
```

```
126         cout << endl;
127     }
128 }
129 
130 int main() {
131     // Tworzenie tablicy liczb od 1 do SIZE
132     const int SIZE = 10000; // n
133     int numbers[SIZE];
134     for (int i = 0; i < SIZE; i++) {
135         numbers[i] = i + 1;
136     }
137 
138     int k = 3000;  // liczba do przewazenia podciagiem
139 
140     // Pomiar czasu dla wersji 1
141     cout << "wersja 1 (sortowanie i sumowanie):" << endl;
142     auto start1 = high_resolution_clock::now();
143     findShortestSubsequence(numbers, SIZE, k);
144     auto stop1 = high_resolution_clock::now();
145     auto duration1 = duration_cast<microseconds>(stop1 - start1);
146     cout << "Czas wykonania: " << duration1.count() << " mikroS\n" << endl; //1
    sekunda = 1 000 000 mikrosekund
147 
148     // Pomiar czasu dla wersji 2
149     cout << "wersja 2 (okno przesuwne):" << endl;
150     auto start2 = high_resolution_clock::now();
151     findShortestSubsequenceSlidingWindow(numbers, SIZE, k);
152     auto stop2 = high_resolution_clock::now();
153     auto duration2 = duration_cast<microseconds>(stop2 - start2);
154     cout << "Czas wykonania: " << duration2.count() << " mikroS\n" << endl;
155 
156     // Pomiar czasu dla wersji 3
157     cout << "wersja 3 (brute force):" << endl;
158     auto start3 = high_resolution_clock::now();
159     findShortestSubsequenceDP(numbers, SIZE, k);
160     auto stop3 = high_resolution_clock::now();
161     auto duration3 = duration_cast<microseconds>(stop3 - start3);
162     cout << "Czas wykonania: " << duration3.count() << " mikroS\n" << endl;
163 
164     return 0;
165 }
```

Literatura

[1] https://www.bigocalc.com/

[2] https://www.geeksforgeeks.org/window-sliding-technique/

[3] https://en.wikipedia.org/wiki/Dynamic_programming

[4] https://www.korepetycjezinformatyki.pl/pseudokod/

[5] https://matplotlib.org/stable/index.html

[6] https://plantuml.com/sequence-diagram

[7] https://plantuml.com/activity-diagram-beta

[8] https://www.geeksforgeeks.org/bubble-sort-algorithm/

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza Rzeszów, 2025
Wydział Elektrotechniki i Informatyki

STRESZCZENIE ZADANIA PROGRAMISTYCZNEGO

DLA ZADANEGO CIĄGU LICZB NATURALNYCH, ZNAJDŹ DŁUGOŚĆ NAJKRÓTSZEGO CIĄGU, KTÓREGO SUMA JEST WIĘKSZA NIŻ ZADANA LICZBA.

Autor: Dawid Kurzępa 179946, nr albumu: 179946
Opiekun: dr inż. Mariusz Borkowski prof. PRz
Słowa kluczowe: (C++, podciągi, algorytmy, analiza złożoności, schematy blokowe)

W pracy zaprezentowano trzy metody znajdowania najkrótszego podciągu liczb naturalnych, którego suma przekracza zadaną wartość: sortowanie i wybór największych elementów, algorytm sliding window oraz podejście dynamiczne. Porównano ich efektywność oraz złożoność obliczeniową, przedstawiając wykresy i schematy blokowe. Implementacja została wykonana w C++.

RZESZOW UNIVERSITY OF TECHNOLOGY Rzeszow, 2025
Faculty of Electrical and Computer Engineering

PROGRAMMING TASK ABSTRACT

FOR A GIVEN SEQUENCE OF NATURAL NUMBERS, FIND THE LENGTH OF THE SHORTEST SEQUENCE WHOSE SUM IS GREATER THAN THE SPECIFIED VALUE NUMBER.

Author: Dawid Kurzępa 179946, Student's ID: 179946
Supervisor: Prof. Imię i nazwisko opiekuna, (academic degree)
Key words: (C++, subsequences, algorithms, complexity analysis, flowcharts)

This paper presents three methods for finding the shortest subsequence of natural numbers whose sum exceeds a given value: sorting and selecting the largest elements, the sliding window algorithm, and a dynamic approach. Their efficiency and complexity were compared using graphs and flowcharts. The implementation was done in C++.