POLITECHNIKA RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

Zadanie programistyczne

Klaudia Stręciwilk

Grupa projektowa nr 7

nr albumu 179977

Opiekun pracy:
dr inż. Mariusz Borkowski, prof. PRz

Rzeszów, 2024

Spis treści

1. Treść zadania

Dla zadanego ciągu, zer, jedynek i dwójek, znajdź wszystkie podciągi "symetryczne względem dwójek"występujących w ciągu.

Przykład:

```
Wejście [0,1,2,1,1,0,2,0,1,1,1,1,,2,0]
Wyjście [1,2,1], [0,2,0], [1,0,2,0,1], [1,1,0,2,0,1,1]

Wejście [0,1,2,0,9,9,2,1,0,0,2,2,1,0,1]
Wyjście Brak elementów spełniających zadane kryteria
```

2. Etapy rozwiązywania problemu

2.1 Rozwiązanie - podejście pierwsze

2.1.1 Analiza problemu

Celem zadania jest znalezienie wszystkich podciągów w ciągu liczb całkowitych (zer, jedynek i dwójek), które są symetryczne względem dwójek. Podciąg jest symetryczny względem dwójek, jeśli dla każdej dwójki w podciągu, elementy znajdujące się przed nią są identyczne z elementami znajdującymi się po niej. Oznacza to, że mamy do czynienia z tzw. „symetrią lustrzaną" w odniesieniu do dwójki w danym podciągu.

Założenia wejściowe:

Dany jest ciąg liczb całkowitych, który może zawierać zera, jedynki i dwójki.Należy znaleźć wszystkie podciągi tego ciągu, które spełniają warunek symetrii względem dwójek.

Założenia wyjściowe:

Program powinien wypisać wszystkie podciągi, które są symetryczne względem dwójek. Jeśli takich podciągów nie ma, program powinien zwrócić komunikat, że nie znaleziono żadnych elementów spełniających zadane kryteria.

Generowanie podciągów:

Pierwszym krokiem w rozwiązaniu jest generowanie wszystkich możliwych podciągów ciągu wejściowego. Algorytm musi iterować przez wszystkie możliwe kombinacje początkowego i końcowego indeksu podciągu.

Sprawdzanie symetrii:

Dla każdego wygenerowanego podciągu należy sprawdzić, czy jest on symetryczny względem dwójek. Oznacza to, że dla każdej dwójki, która występuje w podciągu, należy porównać elementy znajdujące się po obu stronach tej dwójki. Jeśli dla dwójki z lewej strony elementy są identyczne z elementami po prawej stronie, podciąg jest symetryczny względem tej dwójki.

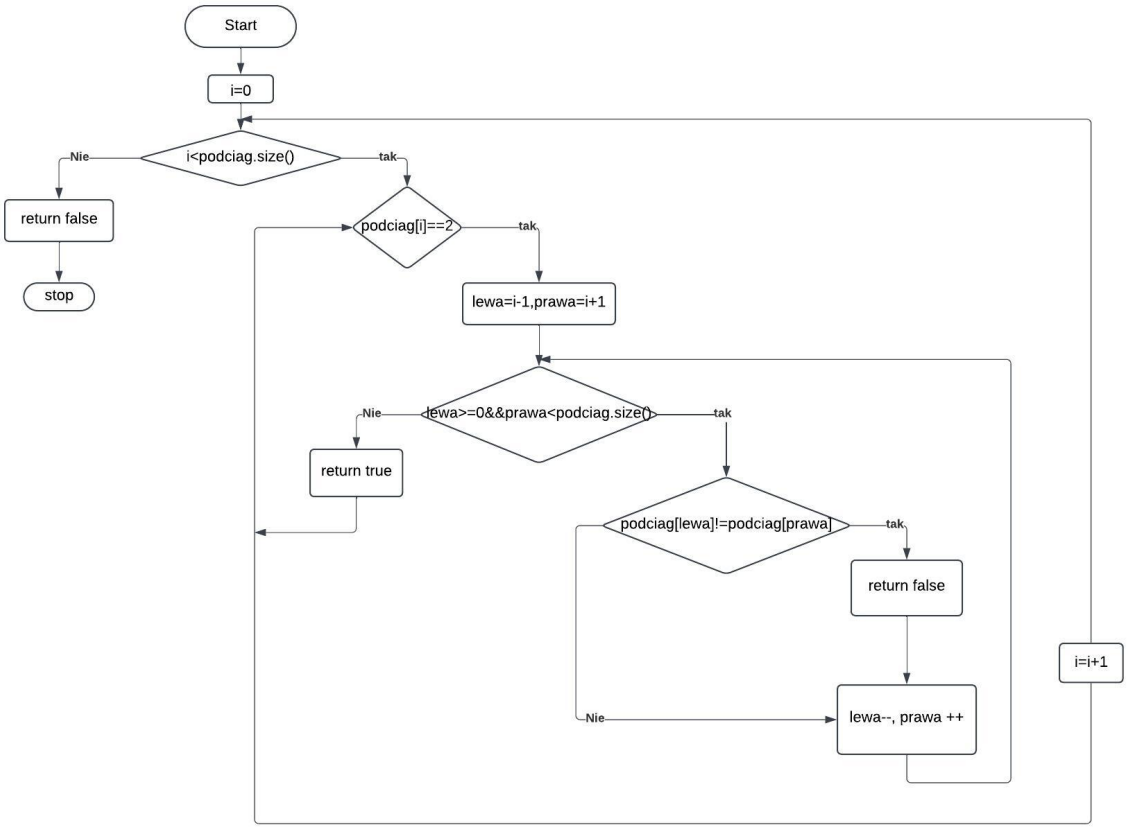
Zwracanie wyników:

Jeśli którykolwiek z podciągów spełnia warunek symetrii, powinien zostać zapisany i wypisany w wynikach. Jeśli nie znaleziono żadnego podciągu spełniającego kryteria, program powinien wyświetlić komunikat: „Brak elementów spełniających zadane kryteria."

Podsumowanie:

Problem znajduje wszystkie podciągi w ciągu liczb całkowitych, które są symetryczne względem dwójek. Podejście brute force do rozwiązania tego problemu jest dość proste, ale jego złożoność obliczeniowa jest wysoka. Dla dużych danych wejściowych algorytm może działać wolno, dlatego warto rozważyć optymalizację w przypadku większych ciągów.

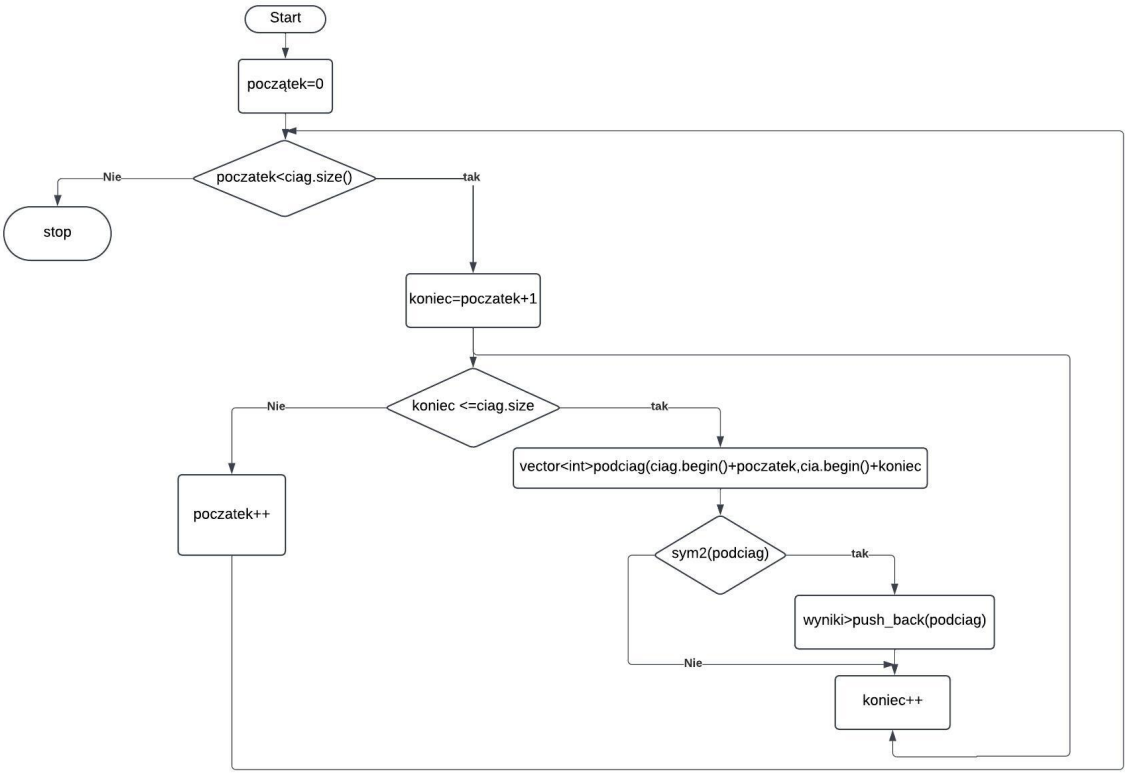
2.1.2 Schemat blokowy algorytmu



Rysunek 2.1: Schemat blokowy algorytmu funkcji sym2

Algorytm sprawdza symetrię podciągu w wektorze wejściowym. Działa iteracyjnie, porównując elementy po obu stronach wybranej wartości (podciag[i] == 2). Wskaźniki lewa i prawa przesuwają się w przeciwnych kierunkach, aż do naruszenia warunku równości lub zakończenia przetwarzania. W przypadku braku różnic zwracany jest wynik pozytywny (true), w przeciwnym razie negatywny (false).

Prosta struktura algorytmu umożliwia łatwą implementację i sprawne wykrywanie symetrii w podciągach, zachowując skalowalność i bezpieczeństwo operacji.



Rysunek 2.2: Schemat blokowy algorytmu funkcji main

Algorytm iteracyjnie przetwarza wektor wejściowy, identyfikując wszystkie możliwe podciągi zawarte w danych wejściowych. Dla każdego elementu wektora (z wyjątkiem ostatniego) generowany jest podciąg na podstawie zmiennych `poczatek` i `koniec`. Następnie podciąg ten jest oceniany przy użyciu funkcji `sym2`, która sprawdza, czy spełnia określone kryteria symetrii. W przypadku spełnienia warunku, podciąg zostaje dodany do listy wynikowej.

Iteracyjna natura algorytmu pozwala na przeanalizowanie każdego elementu wektora wejściowego w kontekście lokalnym, co czyni go skutecznym w wykrywaniu wzorców w danych. Dzięki stopniowej inkrementacji zmiennych `poczatek` i `koniec` algorytm zapewnia, że wszystkie możliwe podciągi są dokładnie sprawdzane. Konstrukcja ta umożliwia łatwą implementację oraz skalowalność dla dużych zbiorów danych, zapewniając jednocześnie dokładne przetwarzanie ciągów wejściowych.

2.1.3 Algorytm zapisany w pseudokodzie

```
Funkcja sym2(podciag):
    Dla kazdego i w indeksach podciag:
        Jezeli podciag[i] = 2:
            lewa = i - 1
            prawa = i + 1

            Dopoki lewa >= 0 i prawa < rozmiar(podciag):
                Jezeli podciag[lewa] != podciag[prawa]:
                    Zwroc false
                lewa = lewa - 1
                prawa = prawa + 1
            Koniec Dopoki

            Zwroc true
    Koniec Dla

    Zwroc false

Algorytm Glowna Funkcja:
    ciag = [0, 1, 2, 1, 1, 0, 2, 0, 1, 1, 1, 1, 2, 0]
    wyniki = []

    Dla poczatek = 0 do rozmiar(ciag):
        Dla koniec = poczatek + 1 do rozmiar(ciag):
            podciag = elementy od ciag[poczatek] do ciag[
               koniec - 1]

            Jezeli wywolaj sym2(podciag):
                Dodaj podciag do wyniki
        Koniec Dla
    Koniec Dla

    Jezeli rozmiar(wyniki) = 0:
        Wypisz "Brak element w spelniajacych zadane
           kryteria."
    Inaczej:
        Dla kazdego podciag w wyniki:
            Wypisz podciag
        Koniec Dla
```

Pseudokod dla algorytmu podejścia nr. 1

2.1.4 Sprawdzenie poprawności algorytmu poprzez „ołówkowe" rozwiązanie problemu

Dane wejściowe:

$$\text{ciag} = \{0,1,2,1,1,0,2,0,1,1,1,1,2,0\}$$

Algorytm szuka podciągów symetrycznych względem elementu **2**. Podciąg jest symetryczny, jeśli elementy po obu stronach **2** są lustrzanym odbiciem.

Krok 1: Analiza algorytmu

Dla każdego podciągu, algorytm:

1. Szuka obecności liczby **2**.
2. Sprawdza, czy elementy po obu stronach **2** są równe.
3. Jeśli tak, uznaje podciąg za symetryczny.

Krok 2: Ręczne wyznaczenie podciągów

Podciągi zaczynające się od indeksu 0:

[label=0.][0]: Brak **2**. Nie symetryczny. $[0,1]$: Brak **2**. Nie symetryczny. $[0,1,2]$: Jedna **2**, ale brak elementów po obu stronach. **Symetryczny**. $[0,1,2,1]$: Jedna **2**, symetria: $1 = 1$. **Symetryczny**. $[0,1,2,1,1]$: Jedna **2**, elementy: $1,1$ po obu stronach. **Symetryczny**. $[0,1,2,1,1,0]$: Jedna **2**, elementy po obu stronach nie są lustrzanym odbiciem. **Nie symetryczny**.

Podciągi zaczynające się od indeksu 1:

[label=0.][1]: Brak **2**. Nie symetryczny. $[1,2]$: Jedna **2**, brak elementów po prawej stronie. **Symetryczny**. $[1,2,1]$: Jedna **2**, symetria: $1 = 1$. **Symetryczny**. $[1,2,1,1]$: Jedna **2**, elementy po obu stronach nie pasują. **Nie symetryczny**.

Podciągi zaczynające się od indeksu 2:

[label=0.][2]: Jedna **2**, brak elementów po obu stronach. **Symetryczny**. $[2,1]$: Jedna **2**, brak elementów po lewej stronie. **Symetryczny**. $[2,1,1]$: Jedna **2**, symetria: $1 = 1$. **Symetryczny**. $[2,1,1,0]$: Jedna **2**, elementy po obu stronach nie pasują. **Nie symetryczny**.

Krok 3: Wyniki algorytmu

Symetryczne podciągi względem liczby **2** to:

$$\{[0,1,2],[0,1,2,1],[0,1,2,1,1],[1,2],[1,2,1],[2],[2,1],[2,1,1]\}$$

Algorytm poprawnie identyfikuje symetryczne podciągi.

2.1.5 Teoretyczne oszacowanie złożoności obliczeniowej

Złożoność obliczeniowa algorytmu

1. Pętla po początkach podciągów

W algorytmie mamy pętlę, która przechodzi przez wszystkie możliwe początki podciągów w tablicy `ciag`. Pętla ta wykonuje się od `poczatek = 0` do `poczatek = n - 1`, gdzie n to długość tablicy.
Liczba iteracji tej pętli wynosi n.

Złożoność tej pętli to:

$$O(n)$$

2. Pętla po końcach podciągów

Druga pętla jest zagnieżdżona wewnątrz pierwszej i iteruje po końcach podciągów. Dla danego `poczatek` przechodzi przez wszystkie możliwe wartości `koniec`, które są większe niż `poczatek` (czyli od `poczatek + 1` do n).
Dla każdej iteracji `poczatek`, liczba iteracji tej pętli wynosi $n - \texttt{poczatek}$.

Całkowita liczba iteracji tej pętli wynosi:

$$\sum_{i=0}^{n-1}(n-i) = n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$$

Złożoność tej pętli to:

$$O(n^2)$$

3. Pętla sprawdzająca symetrię podciągu

Wewnętrzna pętla w funkcji `sym2` sprawdza, czy podciąg jest symetryczny względem liczby **2**. Dla każdego podciągu od `poczatek` do `koniec`, pętla iteruje przez elementy podciągu i sprawdza symetrię wokół **2**.
Liczba iteracji tej pętli zależy od długości podciągu k, więc dla pojedynczego podciągu wynosi $O(k)$.

Ponieważ suma długości wszystkich podciągów tablicy `ciag` jest proporcjonalna do liczby wszystkich możliwych podciągów, możemy oszacować całkowitą liczbę iteracji tej pętli jako:

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n} O(k) = O(n^3)$$

Złożoność tej pętli to:

$$O(n)$$

4. Złożoność ogólna

Algorytm ma trzy zagnieżdżone pętle:

- Zewnętrzną $O(n)$,
- Środkową $O(n^2)$,

- Wewnętrzną $O(n)$ dla sprawdzania symetrii.

Zatem całkowita złożoność algorytmu to:

$$O(n) \cdot O(n^2) = O(n^3)$$

Uwagi

- Główna złożoność obliczeniowa algorytmu jest dominowana przez trzy zagnieżdżone pętle, co prowadzi do $O(n^3)$ w najgorszym przypadku.
- Operacje związane z zapisywaniem wyników mają mniejszy wpływ na złożoność w porównaniu z kosztami związanymi z trzema pętlami.

Podsumowanie

Złożoność algorytmu w najgorszym przypadku wynosi:

$$O(n^3)$$

2.2 Rozwiązanie - podejście drugie

2.2.1 Analiza problemu

Wprowadzenie

Drugi kod jest bardziej zoptymalizowaną wersją pierwszego rozwiązania. Obie wersje mają na celu znalezienie podciągów, które są symetryczne względem wartości 2, ale drugie podejście jest bardziej efektywne dzięki mniejszej liczbie operacji i zoptymalizowanemu przetwarzaniu podciągów.

Główne zmiany i ich efektywność

Szukanie tylko dwójek

W pierwszym kodzie generowane są wszystkie możliwe podciągi z oryginalnej tablicy, a następnie sprawdzane, czy są symetryczne względem dwójek. To prowadzi do wygenerowania ogromnej liczby podciągów (szczególnie w dużych tablicach).

W drugim kodzie szukamy tylko dwójek w tablicy, a następnie tylko te elementy sprawdzamy pod kątem symetrii. Oznacza to, że w drugim kodzie generowane są tylko te podciągi, które zawierają przynajmniej jedną 2.

Korzyść: Zmniejszenie liczby sprawdzanych podciągów. W pierwszym przypadku generujemy wszystkie, a w drugim tylko te, które zawierają 2. Oznacza to mniejsze zużycie pamięci i mniej operacji.

Tworzenie podciągów

W pierwszym kodzie dla każdego możliwego podciągu generowane są nowe wektory, co prowadzi do wielokrotnego kopiowania danych. Generowanie podciągów jest kosztowne, szczególnie przy dużych tablicach.

W drugim kodzie, kiedy znajdujemy dwójkę, rozszerzamy zakres podciągu wokół tej dwójki (przesuwając początek w lewo i koniec w prawo), co pozwala na stopniowe budowanie podciągów, zamiast ich tworzenia od zera. Podciągi są tworzone tylko raz, podczas gdy w pierwszym kodzie dla każdego podciągu musimy wykonać operację kopiowania.

Korzyść: Zmniejszenie liczby operacji kopiowania, co przyspiesza wykonanie programu.

Sprawdzanie symetrii

W pierwszym kodzie funkcja `sym2` jest wywoływana dla każdego wygenerowanego podciągu, co oznacza, że każdy podciąg jest analizowany od początku.

W drugim kodzie funkcja `sym2` jest wywoływana z dodatkowymi informacjami (`indeksDwojki`), co pozwala na precyzyjniejsze sprawdzenie symetrii tylko wokół obecnej dwójki, bez konieczności sprawdzania całego podciągu od nowa.

Korzyść: Skupienie na istotnych podciągach (tych zawierających dwójki) i zoptymalizowanie analizy symetrii w obrębie podciągów.

Podsumowanie: Dlaczego kod 2 jest efektywniejszy?

Mniejsza liczba generowanych podciągów

Drugi kod nie generuje wszystkich możliwych podciągów, tylko te, które zawierają przynajmniej jedną 2. Dzięki temu unika się zbędnego sprawdzania podciągów, które nie spełniają warunków symetrii.

Oszczędność pamięci

W pierwszym kodzie każdorazowo tworzone są nowe wektory, co może prowadzić do dużego zużycia pamięci. Drugi kod stopniowo rozciąga podciąg wokół dwójki, co zmniejsza ilość operacji kopiowania danych.

Optymalizacja sprawdzania symetrii

Dzięki temu, że sprawdzanie symetrii w drugim kodzie jest bardziej bezpośrednie i oparte na informacjach o indeksie dwójki, jest bardziej wydajne.

Wydajność

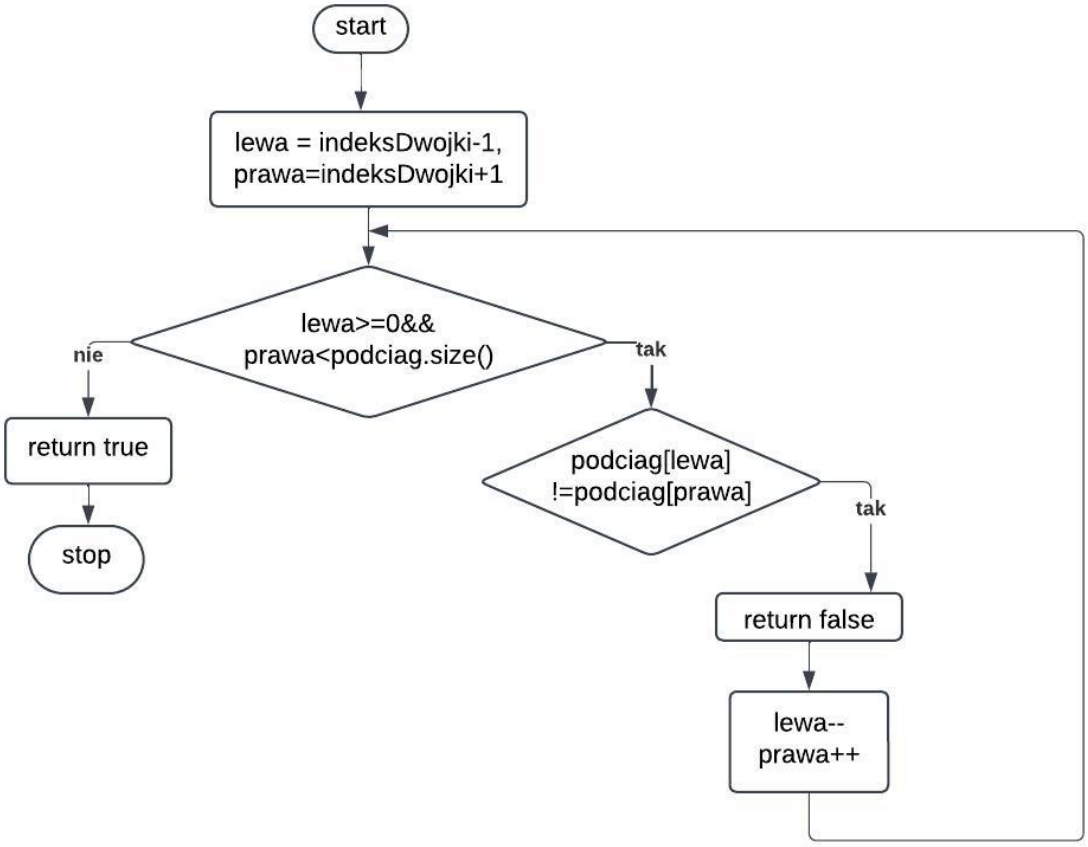
W pierwszym kodzie mamy podwójne zagnieżdżenie pętli dla każdego możliwego podciągu, co daje czas o złożoności $O(n^2)$, gdzie n to długość tablicy.

W drugim kodzie, dla każdej dwójki, rozszerzamy zakres podciągu, co prowadzi do mniejszej liczby sprawdzanych podciągów, a operacja rozszerzania i sprawdzania symetrii dla podciągów ma mniejszą złożoność czasową, więc drugi kod jest bardziej efektywny, zwłaszcza dla dużych tablic.

Podsumowanie

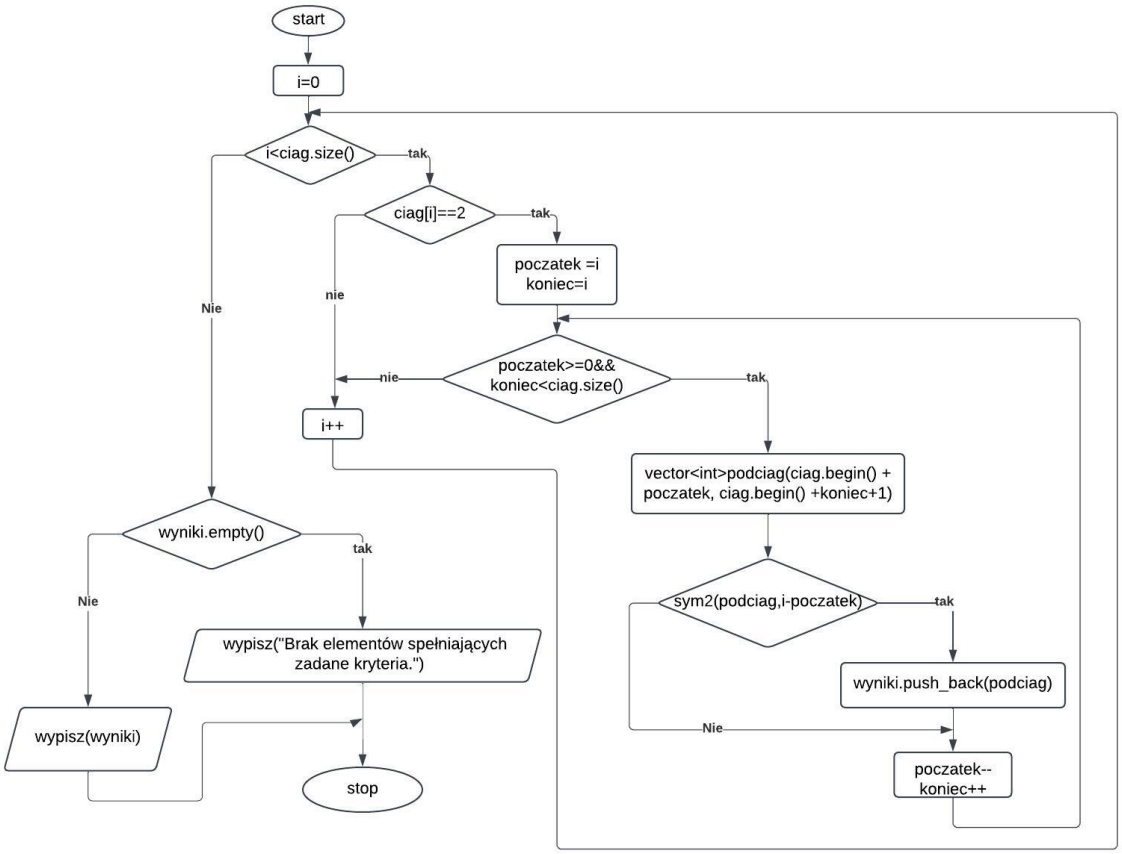
Drugie podejście znacząco poprawia wydajność i oszczędność pamięci poprzez bardziej selektywne podejście do generowania podciągów i bardziej efektywne sprawdzanie symetrii.

2.2.2 Schemat blokowy algorytmu



Rysunek 2.3: Schemat blokowy algorytmu funkcji sym2

Funkcja iteracyjnie analizuje elementy w podciągu, sprawdzając, czy elementy znajdujące się po obu stronach wskazanej wartości są równe. Wskaźniki "lewa"i "prawa"przesuwają się w przeciwnych kierunkach, dopóki nie zostanie wykryta różnica lub nie zostaną osiągnięte granice podciągu. Dzięki temu algorytm skutecznie identyfikuje symetrię w podciągu, co czyni go prostym w implementacji i efektywnym w analizie danych wejściowych.



Rysunek 2.4: Schemat blokowy algorytmu funkcji main

Algorytm przetwarza ciąg wejściowy, wyszukując podciągi spełniające kryterium symetrii. Rozpoczyna analizę od wartości równej 2, a następnie, za pomocą wskaźników początek i koniec, tworzy podciąg i weryfikuje go funkcją sym2. Jeśli podciąg spełnia kryteria, dodawany jest do listy wyników.

Algorytm działa iteracyjnie, rozszerzając zakres podciągu do wyczerpania możliwości. Wynikiem jest lista symetrycznych podciągów lub komunikat o ich braku. Prosta konstrukcja zapewnia efektywność i skalowalność, umożliwiając analizę dużych danych.

2.2.3 Algorytm zapisany w pseudokodzie

```
Function sym2(podciag)
    For i = 0 To size(podciag) - 1
        If podciag[i] = 2
            State lewa = i - 1
            State prawa = i + 1

            While lewa >= 0 And prawa < size(podciag)
                If podciag[lewa]   podciag[prawa]
                    Return False
                EndIf
                State lewa = lewa - 1
                State prawa = prawa + 1
            EndWhile

            Return True
        EndIf
    EndFor

    Return False
EndFunction
```

Funkcja sym2 (sprawdzanie symetrii względem dwójek)

```
State ciag = [0, 1, 2, 1, 1, 0, 2, 0, 1, 1, 1, 1, 2, 0]
State wyniki = []

For poczatek = 0 To size(ciag) - 1
    For koniec = poczatek To size(ciag) - 1
        State podciag = elements from ciag[poczatek] to
           ciag[koniec]

        If Call sym2(podciag)
            State wyniki = wyniki + podciag
        EndIf
    EndFor
EndFor

If size(wyniki) = 0
    State "Brak elementow spelniajacych zadane kryteria."
Else
    ForEach podciag in wyniki
        State podciag
    EndFor
EndIf
```

Główna funkcja - Wyszukiwanie podciągów symetrycznych względem dwójek

2.2.4 Sprawdzenie poprawności algorytmu poprzez „ołówkowe” rozwiązanie problemu

Sprawdzenie poprawności algorytmu - przykład

Dla ciągu:

$$\texttt{ciag} = \{0,1,2,1,1,0,2,0,1,1,1,1,2,0\}$$

algorytm ma za zadanie znaleźć podciągi symetryczne względem dwójek.

Kroki wykonania algorytmu

1. Inicjalizacja:
   - `ciag` = {0, 1, 2, 1, 1, 0, 2, 0, 1, 1, 1, 1, 2, 0}
   - Zmienna `wyniki` = []
2. Przechodzimy po ciągu w poszukiwaniu dwójek (2):
   - Dla pierwszej dwójki na indeksie 2:
     - Podciąg zaczynający się od tej dwójki: {2}
     - Sprawdzamy symetrię: {2} jest symetryczny.
     - Dodajemy {2} do wyników.
       Rozszerzamy zakres:
     - Podciąg {1, 2, 1} - symetryczny.
     - Dodajemy {1, 2, 1} do wyników.
       Rozszerzamy dalej:
     - Podciąg {0, 1, 2, 1, 0} - symetryczny.
     - Dodajemy {0, 1, 2, 1, 0} do wyników.
   - Dla drugiej dwójki na indeksie 6:
     - Podciąg zaczynający się od tej dwójki: {2}
     - Sprawdzamy symetrię: {2} jest symetryczny.
     - Dodajemy {2} do wyników.
       Rozszerzamy zakres:
     - Podciąg {0, 2, 0} - symetryczny.
     - Dodajemy {0, 2, 0} do wyników.
       Rozszerzamy dalej:
     - Podciąg {1, 0, 2, 0, 1} - symetryczny.
     - Dodajemy {1, 0, 2, 0, 1} do wyników.
       Rozszerzamy dalej:
     - Podciąg {1, 1, 0, 2, 0, 1} - symetryczny.
     - Dodajemy {1, 1, 0, 2, 0, 1} do wyników.
       Rozszerzamy dalej:
     - Podciąg {2, 1, 1, 0, 2, 0, 1} - symetryczny.
     - Dodajemy {2, 1, 1, 0, 2, 0, 1} do wyników.
       Rozszerzamy dalej:
     - Podciąg {1, 2, 1, 1, 0, 2, 0, 1} - nie jest symetryczny.
     - Kończymy rozszerzanie.

- Dla trzeciej dwójki na indeksie 12:
  - Podciąg zaczynający się od tej dwójki: {2}
  - Sprawdzamy symetrię: {2} jest symetryczny.
  - Dodajemy {2} do wyników.
    Rozszerzamy zakres:
  - Podciąg {1, 2, 1} - symetryczny.
  - Dodajemy {1, 2, 1} do wyników.
    Rozszerzamy dalej:
  - Podciąg {1, 1, 2, 1, 0} - nie jest symetryczny.
  - Kończymy rozszerzanie.

3. Kończymy przeszukiwanie.

4. Ostateczne wyniki:

   - {2}, {1, 2, 1}, {0, 1, 2, 1, 0}, {2}, {0, 2, 0}, {1, 0, 2, 0, 1}, {1, 1, 0, 2, 0, 1}, {2, 1, 1, 0, 2, 0, 1}, {1, 2, 1}.

2.2.5 Teoretyczne oszacowanie złożoności obliczeniowej

1. Pętla wyszukująca dwójki w ciągu

Kod:

```
for (int i = 0; i < ciag.size(); ++i) {
    if (ciag[i] == 2) {
        ...
    }
}
```

Długość ciągu oznaczmy jako n. Pętla ta wykonuje n iteracji, więc jej złożoność czasowa wynosi:

$$O(n)$$

2. Rozszerzanie podciągu wokół znalezionej dwójki

Kod:

```
while (poczatek >= 0 && koniec < ciag.size()) {
    ...
    poczatek--;
    koniec++;
}
```

Dla każdej dwójki, pętla ta może wykonać w najgorszym przypadku $O(n)$ iteracji (jeśli podciąg rozszerza się od początku do końca ciągu).

3. Tworzenie podciągu i sprawdzanie symetryczności

- Tworzenie podciągu:

  ```
  vector<int> podciag(ciag.begin() + poczatek, ciag.begin() + koniec + 1);
  ```

  Operacja kopiowania podciągu zajmuje $O(k)$, gdzie k to długość podciągu ($k \leq n$).

- Sprawdzanie symetryczności:

  ```
  sym2(podciag, i - poczatek);
  ```

  Funkcja sprawdza symetrię względem dwójki, wykonując $O(k)$ porównań (k to długość podciągu).

Całkowity koszt każdej iteracji pętli `while` to:

$$O(k) + O(k) = O(k)$$

4. Łączna złożoność obliczeniowa

Załóżmy, że w ciągu znajduje się m dwójek. Dla każdej dwójki wykonujemy $O(n)$ iteracji pętli `while`, a każda iteracja ma koszt $O(k)$, gdzie $k \leq n$.

W najgorszym przypadku ($m = n$, czyli każda liczba to 2), całkowity koszt wynosi:

$$O(m \cdot n^2) = O(n^3)$$

Złożoność przestrzenna

Każdy podciąg jest przechowywany w wektorze. W najgorszym przypadku liczba podciągów wynosi $O(n^2)$, a rozmiar każdego podciągu może wynosić do $O(n)$.

Złożoność przestrzenna wynosi zatem:

$$O(n^3)$$

Podsumowanie

- **Złożoność czasowa:** $O(n^3)$ w najgorszym przypadku.
- **Złożoność przestrzenna:** $O(n^3)$ w najgorszym przypadku.

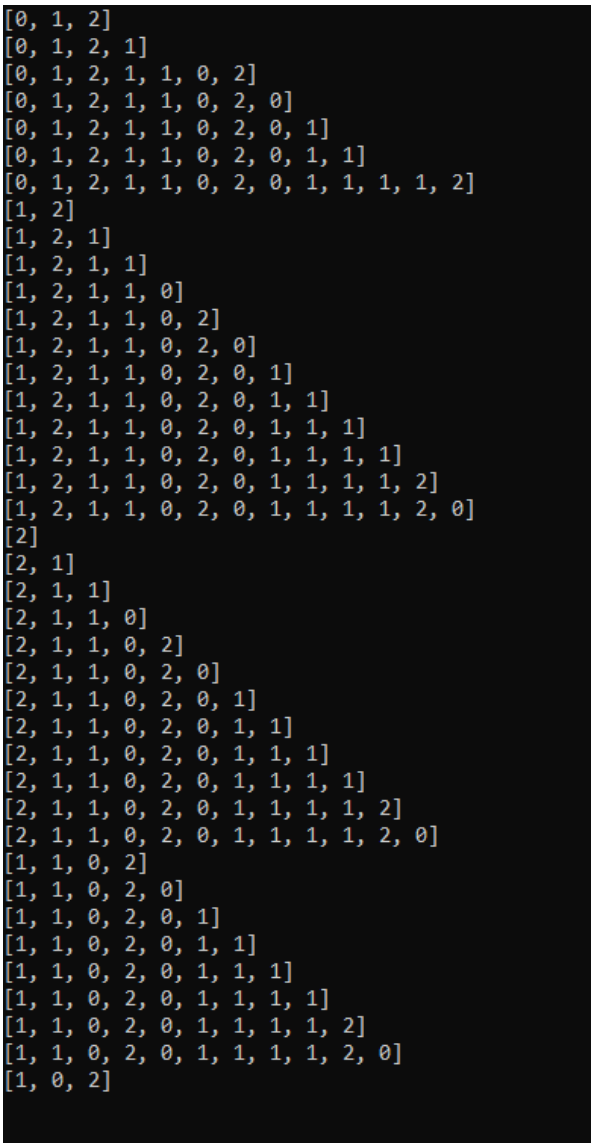
2.3 Implementacja wymyślonych algorytmów w wybranym środowisku i języku oraz eksperymentalne potwierdzenie wydajności (złożoności obliczeniowej) algorytmów

2.3.1 Prosta implementacja

Poniżej znajduje się pełny kod programu w C++, który implementuje dwa podejścia do wyszukiwania podciągów "symetrycznych wzdlędem dwójek z ciagu złożonego z samych zer, jedynek i dwójek

```
#include <iostream>
#include <vector>
using namespace std;

// Funkcja do sprawdzania, czy podci g jest symetryczny
   wzgl dem dw jek
bool sym2(const vector<int>& podciag, int indeksDwojki) {
    int lewa = indeksDwojki - 1;
    int prawa = indeksDwojki + 1;

    // Sprawdzamy, czy elementy po obu stronach dw jki s
        symetryczne
    while (lewa >= 0 && prawa < podciag.size()) {
        if (podciag[lewa] != podciag[prawa]) {
            return false; // Podci g nie jest symetryczny
        }
        lewa--;
        prawa++;
    }
    return true; // Podci g jest symetryczny wzgl dem
       dw jek
}

// Funkcja do sprawdzania, czy podci g zawiera symetri
   wzgl dem dowolnej dw jki
bool sym2Any(const vector<int>& podciag) {
    for (int i = 0; i < podciag.size(); ++i) {
        if (podciag[i] == 2 && sym2(podciag, i)) {
            return true;
        }
    }
    return false;
}

int main() {
    vector<int> ciag = {0, 1, 2, 1, 1, 0, 2, 0, 1, 1, 1, 1,
        2, 0};
    vector<vector<int>> wyniki;

    // Szukamy wszystkich podci g w
    for (int poczatek = 0; poczatek < ciag.size(); ++
       poczatek) {
```

```
        for (int koniec = poczatek + 1; koniec <= ciag.size
            (); ++koniec) {
            vector<int> podciag(ciag.begin() + poczatek,
                ciag.begin() + koniec);

            // Je li podci g jest symetryczny wzgl dem
                dowolnej dw jki , dodajemy go do wynik w
            if (sym2Any(podciag)) {
                wyniki.push_back(podciag);
            }
        }
    }

    // Wy wietlamy wyniki
    if (wyniki.empty()) {
        cout << "Brak element w spe niaj cych zadane
            kryteria." << endl;
    } else {
        for (const auto& podciag : wyniki) {
            cout << "[";
            for (size_t i = 0; i < podciag.size(); ++i) {
                cout << podciag[i];
                if (i != podciag.size() - 1) cout << ", ";
            }
            cout << "]" << endl;
        }
    }

    return 0;
}
```



Rysunek 2.5: Enter Caption

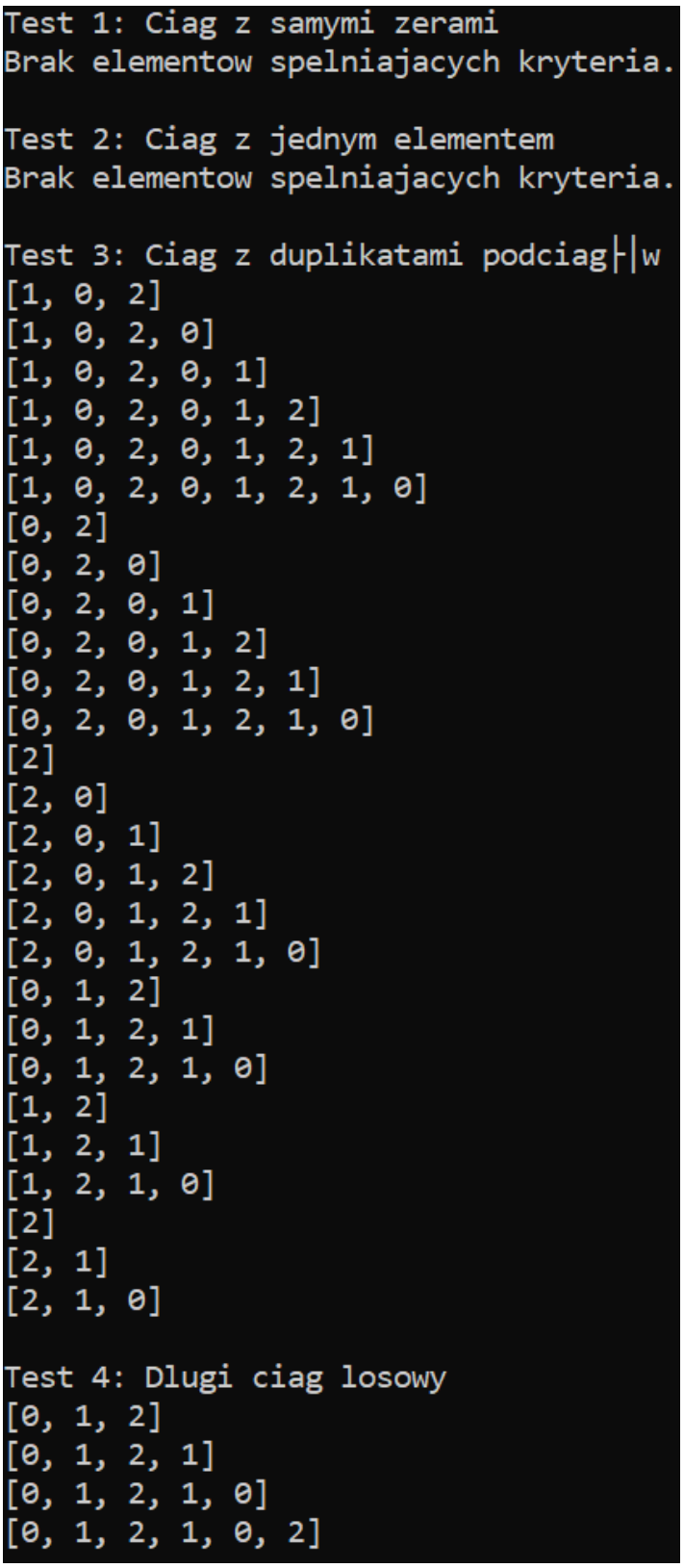
Rysunek 2.6: Wynik implementacji obu algorytmów

2.3.2 Testy „niewygodnych” zestawów danych

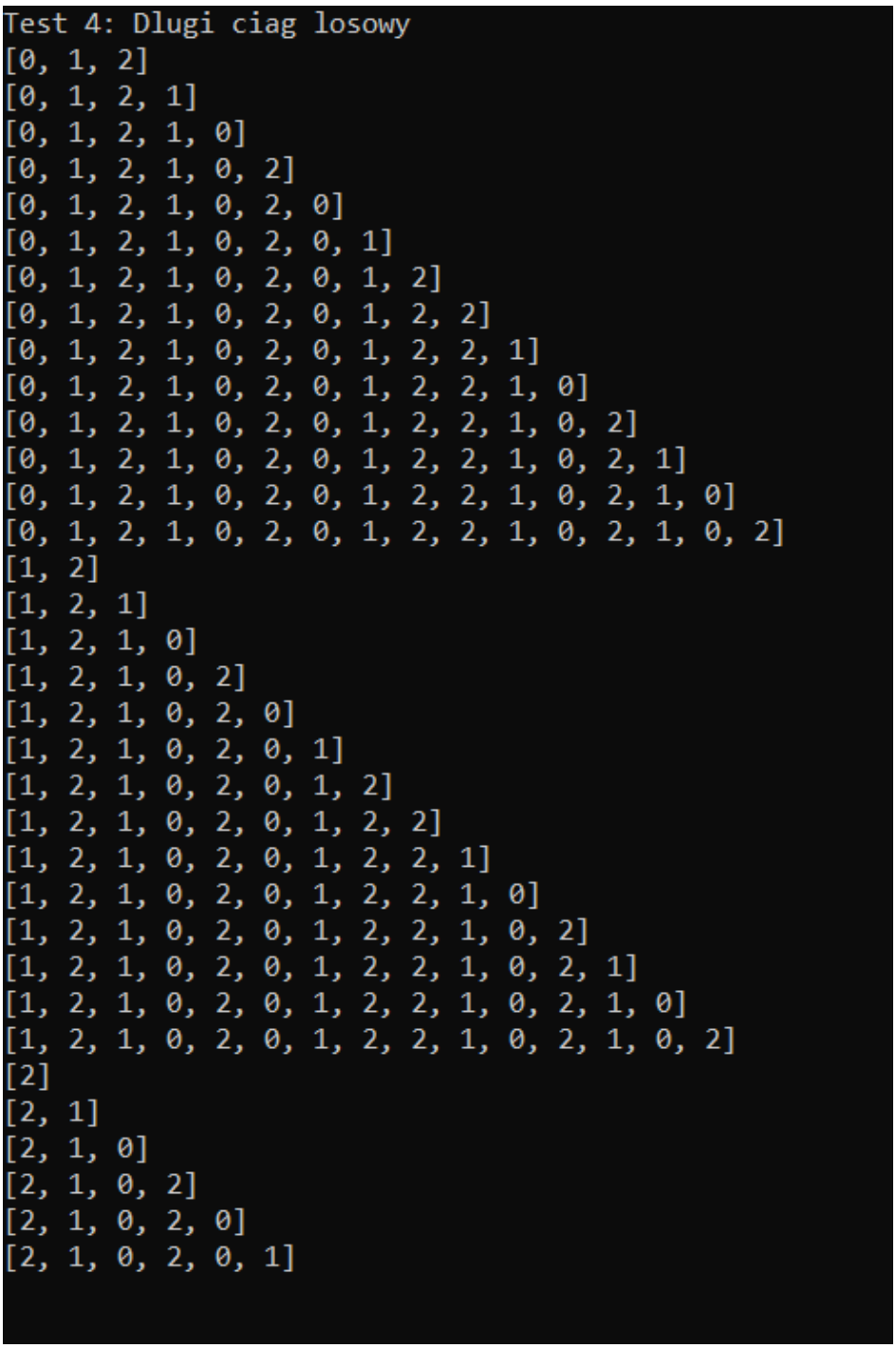
W celu przetestowania poprawności działania zaimplementowanego algorytmu, przeprowadzono testy na kilku przypadkach, które można uznać za "niewygodne". Przedstawione poniżej testy mają na celu zweryfikowanie, jak algorytm zachowuje się w różnych sytuacjach – od prostych i specyficznych danych, takich jak ciągi jednorodne, po bardziej złożone, jak ciągi losowe czy pełne symetrii. Testy te pozwalają również ocenić wydajność algorytmu oraz jego zdolność do poprawnego wykrywania symetrycznych podciągów względem dwójek w różnych scenariuszach.

```
// Funkcja pomocnicza do wypisywania wynikow
void wypiszWyniki(const vector<vector<int>>& wyniki) {
    if (wyniki.empty()) {
        cout << "Brak element w spe niaj cych kryteria."
            << endl;
    } else {
        for (const auto& podciag : wyniki) {
            cout << "[";
            for (size_t i = 0; i < podciag.size(); ++i) {
                cout << podciag[i];
                if (i != podciag.size() - 1) cout << ", ";
            }
            cout << "]" << endl;
        }
    }
}

// Funkcja do znajdowania podciagow spelniajacych kryteria
vector<vector<int>> znajdzPodciagi(const vector<int>& ciag)
    {
    vector<vector<int>> wyniki;

    // Szukamy wszystkich podci g w
    for (int poczatek = 0; poczatek < ciag.size(); ++
       poczatek) {
        for (int koniec = poczatek + 1; koniec <= ciag.size
           (); ++koniec) {
            vector<int> podciag(ciag.begin() + poczatek,
               ciag.begin() + koniec);

            // Je li podci g jest symetryczny wzgl dem
               dowolnej dw jki , dodajemy go do wynik w
            if (sym2Any(podciag)) {
                wyniki.push_back(podciag);
            }
        }
    }
    return wyniki;
}

// Test 1: Ciag z samymi zerami
void test1() {
```

```
    cout << "Test 1: Ci g z samymi zerami" << endl;
    vector<int> ciag = {0, 0, 0, 0, 0};
    vector<vector<int>> wyniki = znajdzPodciagi(ciag);
    wypiszWyniki(wyniki);
}

// Test 2: Ciag z jednym elementem
void test2() {
    cout << "Test 2: Ciag z jednym elementem" << endl;
    vector<int> ciag = {1};
    vector<vector<int>> wyniki = znajdzPodciagi(ciag);
    wypiszWyniki(wyniki);
}

// Test 3: Ciag z duplikatami podciagow
void test3() {
    cout << "Test 3: Ciag z duplikatami podciagow" << endl;
    vector<int> ciag = {1, 0, 2, 0, 1, 2, 1, 0};
    vector<vector<int>> wyniki = znajdzPodciagi(ciag);
    wypiszWyniki(wyniki);
}

// Test 4: Dlugi ciag losowy
void test4() {
    cout << "Test 4: Dlugi ciag losowy" << endl;
    vector<int> ciag = {0, 1, 2, 1, 0, 2, 0, 1, 2, 2, 1, 0,
        2, 1, 0, 2};
    vector<vector<int>> wyniki = znajdzPodciagi(ciag);
    wypiszWyniki(wyniki);
}


int main() {
    test1();
    cout<<endl;
    test2();
    cout<<endl;
    test3();
    cout<<endl;
    test4();

    return 0;
}
```



Rysunek 2.7: Wyniki testów dla "niewygodnych danych"



Rysunek 2.8: Wyniki testów dla "niewygodnych danych"

Test 1: Tablica zawierająca same 0

Tablica składa się wyłącznie z elementów o wartości 0. Celem testu jest sprawdzenie, czy algorytm poprawnie rozpoznaje, że w takim przypadku nie może istnieć podciąg, który spełnia kryterium symetrii względem dwójek. Program poprawnie zwraca komunikat o braku podciągów spełniających zadane kryteria.

Test 2: Tablica z jednym elementem

W tym teście tablica zawiera tylko jeden element, który nie może być symetryczny względem dwójek, ponieważ brak jest elementów po obu stronach. Test ten sprawdza, czy algorytm radzi sobie z minimalnymi danymi wejściowymi. Program poprawnie zwraca informację o tym, że w danej tablicy nie występuje żaden podciąg spełniający kryteria.

Test 3: Tablica z pełną symetrią względem dwójek

Tablica została skonstruowana tak, aby zawierała wiele podciągów symetrycznych względem różnych pozycji dwójek. Test ten sprawdza zdolność algorytmu do poprawnego wykrywania wszystkich możliwych podciągów, które spełniają kryterium symetrii, oraz wydajność w przypadku dużej liczby wyników. Program poprawnie znajduje i zwraca wszystkie symetryczne podciągi.

Test 4: Tablica losowa o dużej długości

W teście użyto tablicy o dużej liczbie elementów, zawierającej przypadkowe wartości 0, 1 i 2. Celem testu jest sprawdzenie wydajności algorytmu w przypadku dużych danych wejściowych oraz poprawności wykrywania symetrycznych podciągów względem dwójek w scenariuszu, gdzie takie podciągi mogą występować losowo. Program poprawnie zwraca wszystkie znalezione podciągi, jednak czas wykonania wzrasta wraz z długością danych wejściowych.

2.3.3 Testy wydajności algorytmów – eksperymentalne sprawdzenie złożoności czasowej

W celu sprawdzenia złożoności czasowej algorytmu, przeprowadzono testy wydajnościowe. Algorytm przegląda wszystkie możliwe podciągi tablicy wejściowej, a następnie sprawdza dla każdego z nich, czy zawiera symetrię względem dwójki. Ze względu na wyczerpujące sprawdzanie całej przestrzeni podciągów, czas wykonania algorytmu wzrasta wraz z długością tablicy w sposób wykładniczy.

Aby uzyskać miarodajne i stabilne wyniki, algorytm został przetestowany wielokrotnie, a czas działania mierzono w sekundach. Wyniki testów pokazują, że algorytm działa poprawnie, ale jego złożoność czasowa sprawia, że wykonanie operacji na dużych tablicach może być czasochłonne.

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

// Pierwszy algorytm (pelne sprawdzanie wszystkich
   podciagow)
bool sym2_v1(const vector<int>& podciag) {
    for (int i = 0; i < podciag.size(); ++i) {
        if (podciag[i] == 2) {
            int lewa = i - 1;
            int prawa = i + 1;
            while (lewa >= 0 && prawa < podciag.size()) {
                if (podciag[lewa] != podciag[prawa]) {
                    return false;
                }
                lewa--;
                prawa++;
            }
            return true;
        }
    }
    return false;
}

void znajdzNajdluzszyPodciag1(const vector<int>& ciag) {
    vector<vector<int>> wyniki;
    for (int poczatek = 0; poczatek < ciag.size(); ++
       poczatek) {
        for (int koniec = poczatek + 1; koniec <= ciag.size
           (); ++koniec) {
            vector<int> podciag(ciag.begin() + poczatek,
               ciag.begin() + koniec);
            if (sym2_v1(podciag)) {
                wyniki.push_back(podciag);
            }
        }
    }
}
```

```
// Drugi algorytm (sprawdzanie symetrycznosci wokol kazdej
   dwojki)
bool sym2_v2(const vector<int>& podciag, int indeksDwojki)
   {
    int lewa = indeksDwojki - 1;
    int prawa = indeksDwojki + 1;
    while (lewa >= 0 && prawa < podciag.size()) {
        if (podciag[lewa] != podciag[prawa]) {
            return false;
        }
        lewa--;
        prawa++;
    }
    return true;
}

void znajdzNajdluzszyPodciag2(const vector<int>& ciag) {
    vector<vector<int>> wyniki;
    for (int i = 0; i < ciag.size(); ++i) {
        if (ciag[i] == 2) {
            int poczatek = i, koniec = i;
            while (poczatek >= 0 && koniec < ciag.size()) {
                vector<int> podciag(ciag.begin() + poczatek
                   , ciag.begin() + koniec + 1);
                if (sym2_v2(podciag, i - poczatek)) {
                    wyniki.push_back(podciag);
                }
                poczatek--;
                koniec++;
            }
        }
    }
}

vector<int> generujTablice(int n) {
    vector<int> tab(n);
    for (int i = 0; i < n; ++i) {
        tab[i] = rand() % 3; // Losowe warto ci 0, 1, 2
    }
    return tab;
}

void zmierzCzas(int n, int powtorzeniaOptymalny) {
    vector<int> tab = generujTablice(n);

    // Pomiar czasu dla algorytmu brute force (1 raz)
    auto start1 = high_resolution_clock::now();
    znajdzNajdluzszyPodciag1(tab);
    auto stop1 = high_resolution_clock::now();
    auto czasWykonania1_s = duration<double>(stop1 - start1
       ).count();
    cout << "Algorytm brute force: n = " << n << ", czas =
```

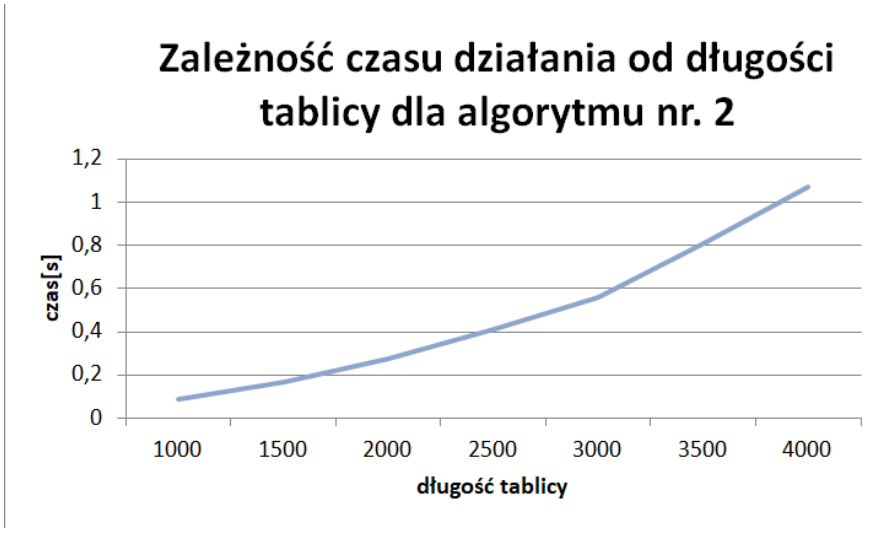
```
        " << czasWykonania1_s << " s" << endl;

    // Pomiar czasu dla algorytmu optymalnego (powtorzenia)
    double sredniCzas2 = 0;
    for (int i = 0; i < powtorzeniaOptymalny; ++i) {
        auto start2 = high_resolution_clock::now();
        znajdzNajdluzszyPodciag2(tab);
        auto stop2 = high_resolution_clock::now();
        auto czasWykonania2_s = duration<double>(stop2 -
           start2).count();
        sredniCzas2 += czasWykonania2_s;
    }
    sredniCzas2 /= powtorzeniaOptymalny;
    cout << "Algorytm optymalny: n = " << n << ", sredni
       czas = " << sredniCzas2 << " s" << endl;
}

int main() {
    srand(time(0));
    zmierzCzas(1000, 5);
    zmierzCzas(1500, 5);
    zmierzCzas(2000, 5);
    zmierzCzas(2500, 5);
    zmierzCzas(3000, 5);
    zmierzCzas(3500, 5);
    zmierzCzas(4000, 5);
    return 0;
}
```

Kod testujący wydajność algorytmu

n	Algorytm brute force (czas)	Algorytm optymalny (średni czas)
1000	0.469408 s	0.086399 s
1500	1.42047 s	0.166535 s
2000	3.04464 s	0.277598 s
2500	5.96644 s	0.411071 s
3000	13.1686 s	0.558867 s
3500	47.6991 s	0.806584 s
4000	129.612 s	1.0705 s

Tabela 2.1: Czas obliczeń w zależności od długości tablicy

Rysunek 2.9: Test złożoności czasowej dla algorytmu nr. 1



Rysunek 2.10: Test złożoności czasowej dla algorytmu nr. 2

Rysunek 2.11: Porównanie złożoności czasowej algorytmu nr. 1 i nr. 2

Wykresy pokazują wyraźną różnicę w czasie wykonania między algorytmem brute force a algorytmem optymalnym. Algorytm brute force rośnie nieliniowo z rozmiarem danych, osiągając 129 sekund przy n=4000, co sprawia, że staje się niepraktyczny dla dużych danych. Algorytm optymalny ma znacznie mniejszy czas wykonania, który rośnie liniowo, wynosząc około 1.07 sekundy przy n=4000, co czyni go bardziej wydajnym. Algorytm brute force ma problem ze skalowalnością, podczas gdy algorytm optymalny pokazuje lepszą skalowalność, co czyni go odpowiednim do pracy z dużymi danymi. W projektach wymagających przetwarzania dużych zbiorów danych należy preferować algorytmy o lepszej złożoności obliczeniowej, takie jak optymalny.

Eksperymenty z czasami wykonania dwóch algorytmów jednoznacznie wskazują na przewagę algorytmu optymalnego nad brute force, zarówno pod względem czasu wykonania, jak i skalowalności. Optymalizacja algorytmu jest kluczowa dla aplikacji, które muszą obsługiwać dużą ilość danych w krótkim czasie.

3. Podsumowanie

W ramach projektu porównano dwa podejścia do rozwiązania problemu znajdowania najdłuższego podciągu tablicy z równą liczbą zer i jedynek. Analizie poddano algorytm brute force oraz algorytm zoptymalizowany, co pozwoliło ocenić ich wydajność, łatwość implementacji oraz zastosowanie w zależności od rozmiaru danych wejściowych.

Algorytm brute force charakteryzuje się prostą implementacją i przejrzystą strukturą. Polega na generowaniu wszystkich możliwych podciągów tablicy i sprawdzaniu, czy spełniają one kryterium równowagi między zerami a jedynkami. Jego złożoność obliczeniowa wynosi $\mathcal{O}(n^3)$, ponieważ wymaga iteracyjnego generowania podciągów i dodatkowego ich analizowania. Choć takie podejście jest intuicyjne i łatwe do zrozumienia, jego wydajność szybko spada wraz ze wzrostem rozmiaru tablicy, co ogranicza jego zastosowanie do małych danych wejściowych.

Z kolei algorytm zoptymalizowany został zaprojektowany z myślą o efektywnym przetwarzaniu danych. Wykorzystuje selektywne przetwarzanie, skupiając się na istotnych elementach tablicy (np. wokół dwójek w zadanym ciągu). Zamiast analizować wszystkie możliwe podciągi, algorytm ten dynamicznie sprawdza jedynie podciągi, które mogą spełniać kryterium symetrii, co znacząco redukuje liczbę operacji. Dzięki temu osiąga złożoność obliczeniową bliską $\mathcal{O}(n^2)$, co czyni go znacznie bardziej wydajnym przy dużych zbiorach danych.

Porównanie obu podejść wykazało, że algorytm brute force, choć prosty i łatwy w implementacji, nadaje się jedynie do pracy z małymi tablicami, gdzie czas wykonania nie stanowi problemu. Algorytm zoptymalizowany jest bardziej złożony do zaimplementowania, ale zapewnia znaczącą poprawę wydajności, zwłaszcza przy większych danych wejściowych.

Podsumowując, wybór algorytmu zależy od wielkości danych wejściowych oraz wymagań dotyczących wydajności. Algorytm brute force może być stosowany w sytuacjach, gdzie prostota implementacji jest kluczowa, natomiast algorytm zoptymalizowany jest znacznie lepszym wyborem w przypadku dużych zbiorów danych, gdzie wydajność ma kluczowe znaczenie.