WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Sprawozdanie z projektu programistycznego

Ciągi liczbowe z podciągami malejącymi

Imię i Nazwisko Daniel Olejasz
Numer grupy: P05
Data: 15 grudnia 2024

Środowisko: Code::Blocks IDE, język C++

GitHub jako repozytorium kodu

Spis treści

1 Treść zadania

Dla zadanego ciągu liczbowego liczb całkowitych (w postaci tablicy) znajdź liczbę wszystkich podciągów malejących (Podciąg musi składać się z przynajmniej dwóch wartości).

Przykład:

Wejście: A[] = [5, 4, 2, 2, 1]
Wyjście: Liczba wszystkich podciągów malejących to 4
[5, 4], [5, 4, 2], [4, 2], [2, 1]

Wejście: A[] = [2, 5, 3]
Wyjście: Liczba wszystkich podciągów malejących to 1
[5, 3]

Wejście: A[] = [5, 4, 2, 2, 1]
Wyjście: Liczba wszystkich podciągów malejących to 0

2 Rozwiązywanie problemu

2.1 Podejście pierwsze - stałe tablice

2.1.1 Opis problemu

Program rozwiązuje problem znajdowania wszystkich malejących podciągów w zadanych tablicach liczb całkowitych oraz ich liczby. Każdy podciąg musi składać się z co najmniej dwóch liczb i musi spełniać warunek, że każdy kolejny element jest mniejszy od poprzedniego. Dane wejściowe są pobierane z pliku tekstowego, a wyniki (lista malejących podciągów i ich liczba) są zapisywane do innego pliku tekstowego.

2.1.2 Struktura danych wejściowych i wyjściowych

Wejście (plik `input.txt`):

1. Liczba tablic (`numArrays`)
2. Dla każdej tablicy:
    - Liczba elementów tablicy (`n`)
    - `n` liczb całkowitych, stanowiących elementy tablicy

Wyjście (plik `output.txt`):

1. Lista wszystkich malejących podciągów dla każdej tablicy
2. Łączna liczba tych podciągów

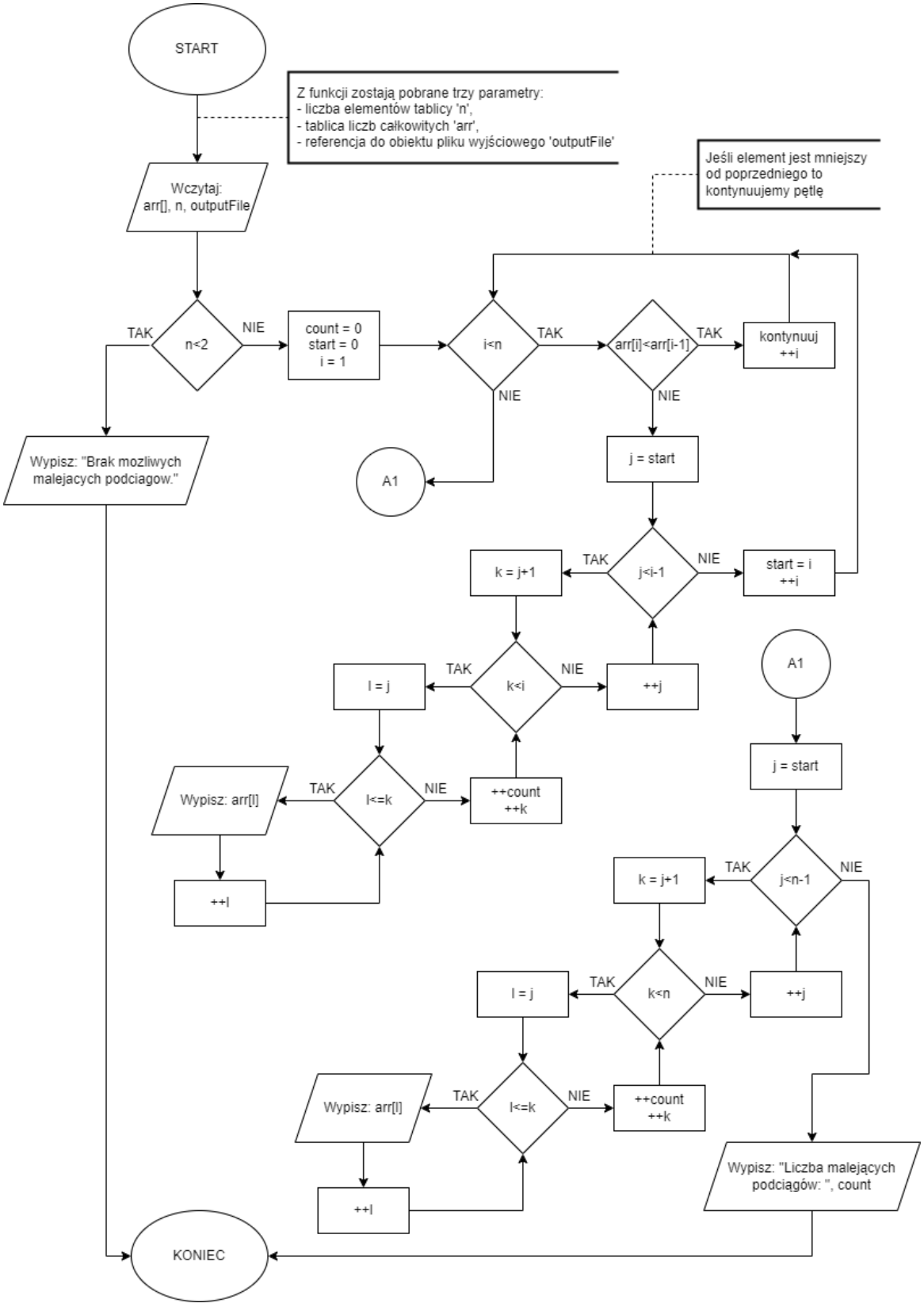
2.1.3 Opis działania programu

1. Wczytywanie danych z pliku
   - Program otwiera plik wejściowy `input.txt`. Jeśli plik nie istnieje lub wystąpi problem z jego otwarciem, program zgłasza błąd i kończy działanie.
   - Z pierwszej linii pliku wczytywana jest liczba tablic (`numArrays`)
   - Następnie dla każdej tablicy:
     - Wczytywany jest jej rozmiar (`n`)
     - Wczytywane jest `n` elementów tablicy do lokalnej tablicy `arr`
2. Przetwarzanie danych

   Dla każdej tablicy program:
   - Wypisuje jej zawartość do pliku wyjściowego `output.txt`.
   - Przetwarza tablice, szukając wszystkich malejących podciągów za pomocą funkcji `Subarrays`.
3. Funkcja `Subarrays`
   - Funkcja przyjmuje:
     - Tablicę liczb całkowitych `arr`.
     - Liczbę elementów tablicy `n`.
     - Referencję do obiektu pliku wyjściowego `outputFile`
4. Działanie funkcji
   - Sprawdza, czy długość tablicy jest mniejsza niż 2. Jeśli tak, wypisuje komunikat, że nie ma możliwych malejących podciągów.
   - Zlicza liczbę malejących podciągów:
     - Utrzymuje indeks `start` jako początek aktualnej malejącej sekwencji.
     - Iteruje po tablicy sprawdzając warunek `arr[i]<arr[i-1]`.
     - Gdy napotka element, który nie spełnia tego warunku, wypisuje wszystkie możliwe podciągi wynikające z bieżącej sekwencji do pliku wyjściowego.
     - Po zakończeniu iteracji przetwarza ostatnią sekwencję, jeśli zakończyła się na końcu tablicy.
   - Zapisuje liczbę znalezionych podciągów do pliku.
5. Zapisywanie wyników do pliku
   - Wszystkie malejące podciągi są zapisywane do pliku wyjściowego, każdy w formacie `[a,b,c,...]`.
   - Po każdym zestawie podciągów dla tablicy program dodaje liczbę znalezionych podciągów.
6. Zamknięcie plików
   - Program zamyka pliki wejściowy i wyjściowy po zakończeniu działania.

2.1.4 Schemat blokowy algorytmu

W tym miejscu znajduję się schemat blokowy dla funkcji `Subarrays`. Ilustruje on zasadę jej działania:



Rysunek 1: Schemat blokowy funkcji `Subarrays`.

2.1.5 Pseudokod algorytmu

Pseudokod dla funkcji `Subarrays` rozwiązującej zadany problem:

```
Funkcja Subarrays(arr, n, outputFile):
    Jeśli n < 2:
        Wypisz "Brak mozliwych malejacych podciagow" do pliku
        Zakończ funkcję
    Zainicjalizuj zmienną "count" na 0
    Zainicjalizuj zmienną "start" na 0
    Dla i od 1 do n-1:
        Jeśli arr[i] < arr[i-1]:
            Kontynuuj pętlę
        Inaczej:
            Dla j od start do i-2:
                Dla k od j+1 do i-1:
                    Wypisz podciąg [arr[j], arr[j+1], ..., arr[k]] do pliku
                    Zwiększ count o 1
            Ustaw start na i
    Dla j od start do n-2:
        Dla k od j+1 do n-1:
            Wypisz podciąg [arr[j], arr[j+1], ..., arr[k]] do pliku
            Zwiększ count o 1
    Wypisz "Liczba malejacych podciagow: count" do pliku
```

2.2 Drugie podejście - dynamiczne tablice

2.2.1 Opis problemu - pytanie o usprawnienie programu

W tym miejscu warto zadać sobie pytanie, czy jest możliwe napisanie kodu programu, który będzie zawierał mniej pętli `for`, a co za tym idzie będzie skutkował niższą złożonością obliczeniową? Odpowiedź brzmi: **Tak, ale nie do końca.**

Obniżenie złożoności czasowej programu wymaga optymalizacji działania funkcji `Subarrays`, ponieważ jest to główny element wpływający na czas wykonania programu. Funkcja ta generuje wszystkie malejące podciągi w sposób nieefektywny, przeszukując wszystkie możliwe kombinacje, co powoduje złożoność $O(n^3)$ (więcej o złożoności obliczeniowej w kolejnym rozdziale).

W jaki więc sposób można częściowo rozwiązać to zadanie tak, aby złożoność obliczeniowa była mniejsza? Poniżej przedstawiam sposób na rozwiązanie tego problemu opisując samą funkcję:

2.2.2 Opis działania omawianej funkcji

1. Wczytywanie danych
   - Funkcja otrzymuję tablicę `arr` oraz liczbę jej elementów `n`.
   - Wewnątrz funkcji tworzone są zmienne `licznik = 0` oraz `dlugosc = 1`.
2. Przetwarzanie danych

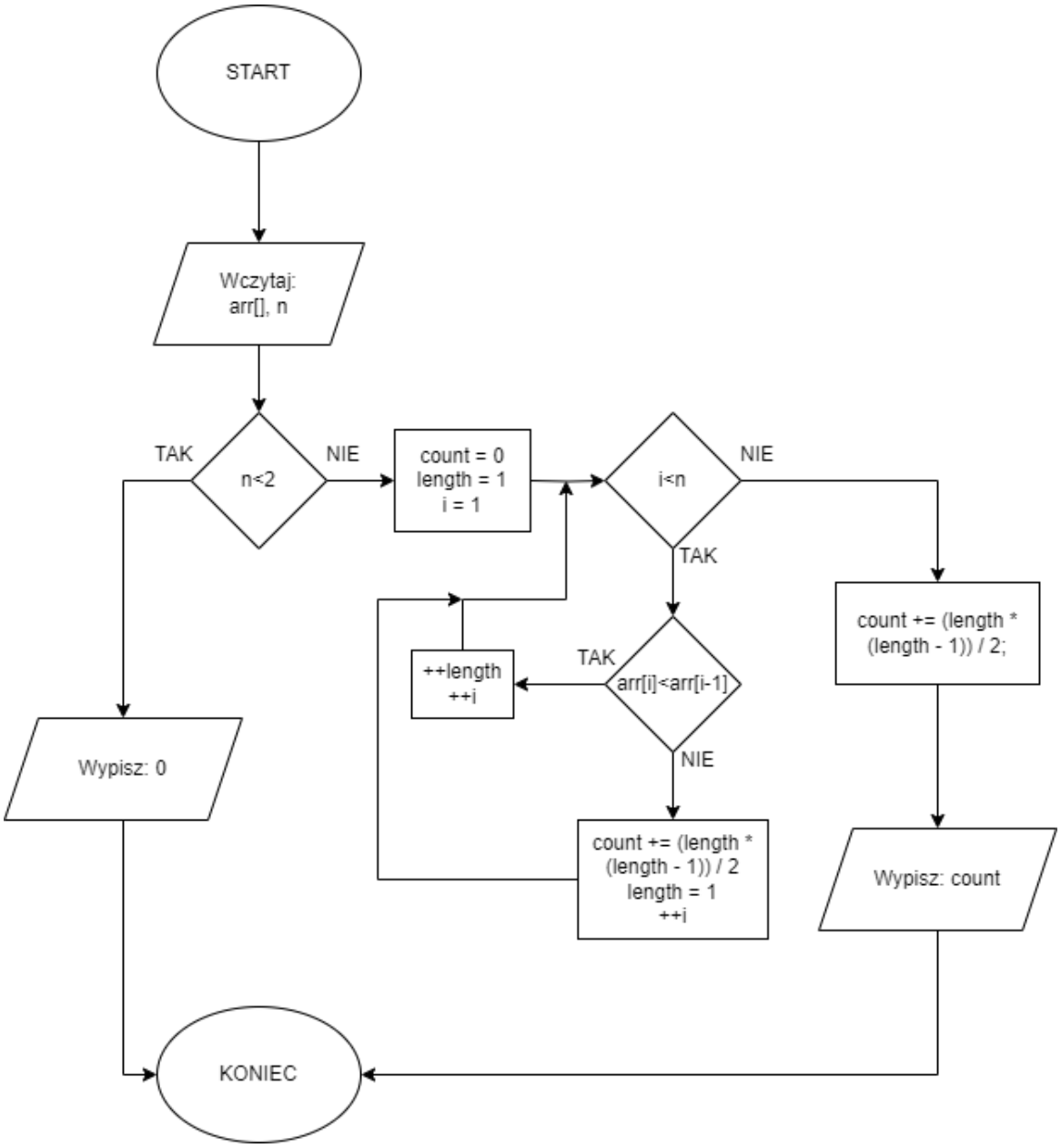
   Dla każdej tablicy funkcja (CountDecreasingSubarrays):
   - Sprawdza, czy długość tablicy jest mniejsza od 2. Jeśli tak, to zwraca liczbę 0 (ponieważ nie znajdują się tam żadne podciągi malejące).
   - Wypisuje liczbę podciągów.
3. Funkcja `CountDecreasingSubarrays`
   - Funkcja przyjmuje:
     - Tablicę liczb całkowitych `arr`.
     - Liczbę elementów tablicy `n`.
   - Następnie funkcja iteruję przez tablicę (porównuje kolejne elementy tablicy):
     - Jeśli element jest mniejszy od poprzedniego, wydłużamy bieżącą sekwencję malejącą `(dlugosc)`.
     - Jeśli nie jest, obliczamy liczbę podciągów dla zakończonego fragmentu malejącego i resetujemy długość.
     - Po zakończeniu pętli dodajemy podciągi dla ostatniego fragmentu malejącego.
4. Wyświetlanie wyników
   - Na koniec funkcja zwraca łączną liczbę malejących podciągów.

Zatem częściowym rozwiązaniem programu byłaby taka funkcja, która **tylko** zlicza ilość malejących podciągów, bez ich wypisania. Zmniejsza to czytelność wyników, gdyż nie otrzymujemy wypisanych podciągów tylko ich liczbę, ale skorzystanie z takiej funkcji ma też swoje zalety. Ta funkcja ma złożoność obliczeniową `O(n)`, co jest bardzo dużą różnicą względem poprzedniej funkcji.

Zatem w przypadku implementacji takiej funkcji w innym programie warto się zastanowić nad jej użyciem zamiast funkcji `Subarrays`, w przypadku gdy nie potrzebujemy znać podciągów, lecz tylko ich liczbę.

2.2.3 Schemat blokowy funkcji CountDecreasingSubarrays

W tym miejscu znajduję się schemat blokowy omawianej funkcji `CountDecreasingSubarrays`. Przedstawia on zasadę jej działania:



Rysunek 2: Schemat blokowy funkcji `CountDecreasingSubarrays`.

2.2.4 Pseudokod funkcji CountDecreasingSubarrays

Pseudokod dla funkcji `CountDecreasingSubarrays` rozwiązującej zadany problem:

```
Funkcja CountDecreasingSubarrays(arr, n):
    Jeśli n < 2:
        Wypisz 0
        Zakończ funkcję
    Zainicjalizuj zmienną "count" na 0
    Zainicjalizuj zmienną "length" na 1
    Dla i od 1 do n-1:
        Jeśli arr[i] < arr[i-1]:
            Zwiększ length o 1
        Inaczej:
            Wykonaj count += (length * (length - 1)) / 2;
            Ustaw length na 1
    Wykonaj count += (length * (length - 1)) / 2;
    Wypisz "Liczba malejacych podciagow: count"
```

2.3 Implementacja obu podanych funkcji

2.3.1 Pierwsza wersja

Zaimplementujemy teraz nasze pierwsze opisane podejście. Mamy już schemat tego jak funkcja powinna wyglądać oraz jej zarys w pseudokodzie, zatem zostaję ją teraz zapisać w języku `C++`.

```
void Subarrays(int arr[], int n, ofstream &outputFile) {
    if (n < 2) {
        outputFile << "Brak mozliwych malejacych podciagow." << endl;
        return; // Jeśli długość tablicy jest mniejsza niż 2, nie ma podciągów
    }

    int count = 0; // Zmienna przechowująca liczbę malejących podciągów
    int start = 0; // Początek aktualnego malejącego podciągu

    // Zliczamy podciągi
    for (int i = 1; i < n; ++i) {
        if (arr[i] < arr[i - 1]) {
            continue; // Jeśli element jest mniejszy od poprzedniego, kontynuujemy podciąg
        } else {
            // Jeśli sekwencja się kończy, wypisz wszystkie podciągi wynikające z aktualnej sekwencji
            for (int j = start; j < i - 1; ++j) {
                for (int k = j + 1; k < i; ++k) {
                    // Zwiększamy licznik i wypisujemy podciąg do pliku
                    outputFile << "[";
                    for (int l = j; l <= k; ++l) {
                        outputFile << arr[l];
                        if (l < k) outputFile << ", ";
                    }
                    outputFile << "]" << ", ";
                    ++count;
                }
            }
            start = i; // Reset początku nowej sekwencji
        }
    }
```

Rysunek 3: Implementacja pierwszej metody.

```
    // Przetwarzamy ostatni podciąg, jeśli zakończył się na końcu tablicy
    for (int j = start; j < n - 1; ++j) {
        for (int k = j + 1; k < n; ++k) {
            // Zwiększamy licznik i wypisujemy podciąg do pliku
            outputFile << "[";
            for (int l = j; l <= k; ++l) {
                outputFile << arr[l];
                if (l < k) outputFile << ", ";
            }
            outputFile << "]" << endl;
            ++count;
        }
    }

    // Zapisz liczbę malejących podciągów do pliku
    outputFile << "Liczba malejacych podciagow: " << count << endl;
}
```

Rysunek 4: Implementacja pierwszej metody.

Funkcja `Subarrays` po utworzeniu w języku `C++` wygląda następująco.

Tworzymy teraz nasz plik wejściowy i dodajemy do niego dane. Zgodnie z oczekiwanym przez program wejściem, plik `input.txt` będzie wyglądał następująco:

```
4
5 5 4 2 2 1
3 2 5 3
5 1 2 4 6 7
4 8 6 4 2
```

Następnie w głównej funkcji `main` programu wywołujemy naszą utworzoną funkcję `Subarrays`.

```
int main() {
    // Otwieramy plik do odczytu
    ifstream inputFile("input.txt");
    if (!inputFile) {
        cout << "Nie mozna otworzyc pliku wejsciowego!" << endl;
        return 1; // W przypadku bledu otwarcia pliku, kończymy program
    }

    // Otwieramy plik do zapisu
    ofstream outputFile("output.txt");
    if (!outputFile) {
        cout << "Nie mozna otworzyc pliku wyjsciowego!" << endl;
        return 1; // W przypadku bledu otwarcia pliku, kończymy program
    }

    // Wczytanie liczby tablic i ich elementów z pliku
    int numArrays;
    inputFile >> numArrays;

    // Dla każdej tablicy wczytujemy dane
    for (int i = 0; i < numArrays; ++i) {
        int n;
        inputFile >> n; // Wczytujemy rozmiar tablicy
        int arr[n]; // Deklaracja tablicy

        // Wczytanie elementów tablicy
        for (int j = 0; j < n; ++j) {
            inputFile >> arr[j];
        }
```

Rysunek 5: Wywoływanie pierwszej metody.

```
        // Wypisujemy dane na temat tablicy do pliku wyjściowego
        outputFile << "Podciagi malejace w tablicy [";
        for (int j = 0; j < n; ++j) {
            outputFile << arr[j];
            if (j != n - 1) {
                outputFile << ", ";
            }
        }
        outputFile << "]: " << endl;

        // Wywołujemy funkcje dla każdej tablicy
        Subarrays(arr, n, outputFile);
        outputFile << endl;
    }

    inputFile.close();  // Zamykanie pliku wejściowego
    outputFile.close(); // Zamykanie pliku wyjściowego

    return 0;
}
```

Rysunek 6: Wywoływanie pierwszej metody.

2.3.2 Druga wersja

Zaimplementujemy teraz nasze drugie opisane podejście. Tutaj także mamy już schemat tego jak funkcja powinna wyglądać oraz jej zarys w pseudokodzie, zatem zapisujemy ją teraz w języku `C++`. Posłużę się tutaj dodatkową funkcją `RunTests`, która odpowiedzialna jest za testy poprawności naszych funkcji.

```cpp
// Funkcja testujaca
void RunTests() {
    // Tablica testów: {tablica, rozmiar tablicy, oczekiwany wynik}
    struct TestCase {
        int arr[10]; // Maksymalny rozmiar tablicy
        int size;
        int expected;
    };

    // Przykładowe testy
    TestCase tests[] = {
        {{5, 4, 2, 2, 1}, 5, 4},       // 4 malejace podciagi: [5,4], [5,4,2], [4,2], [2,1]
        {{2, 5, 3}, 3, 1},             // 1 malejacy podciag: [5,3]
        {{1, 2, 4, 6, 7}, 5, 0},       // Brak malejacych podciagów
        {{9, 7, 5, 3, 1}, 5, 1},      // 10 malejacych podciagów
        {{1}, 1, 0},                   // Tablica jednoelementowa: brak podciagów
        {{1, 1, 1}, 3, 0},             // Wszystkie elementy równe: brak malejacych podciagów
        {{8, 6, 6, 5, 3, 1}, 6, 7},    // Malejace podciagi z powtórzeniami
        {{10, 9}, 2, 1},               // Jeden malejacy podciag
        {{3, 2, 1, 2, 1, 0}, 6, 7},    // Kombinacja malejacych i rosnacych sekwencji (6 malejacych)
    };

    // Liczba testów
    int numTests = sizeof(tests) / sizeof(tests[0]);

    // Wykonanie testów
    for (int i = 0; i < numTests; ++i) {
        int result = CountDecreasingSubarrays(tests[i].arr, tests[i].size);
        cout << "Test " << (i + 1) << ": ";
        if (result == tests[i].expected) {
            cout << "PASSED (expected " << tests[i].expected << ", got " << result << ")" << endl;
        } else {
            cout << "FAILED (expected " << tests[i].expected << ", got " << result << ")" << endl;
        }
    }
}
```

Rysunek 7: Funkcja pomocnicza `RunTests` sprawdzająca poprawność działania programu.

```cpp
// Funkcja liczaca liczbe malejacych podciagów
int CountDecreasingSubarrays(int arr[], int n) {
    if (n < 2) return 0; // Brak malejacych podciagów

    int count = 0; // Liczba podciagów
    int length = 1; // Długość bieżacego malejacego fragmentu

    for (int i = 1; i < n; ++i) {
        if (arr[i] < arr[i - 1]) {
            ++length; // Kontynuujemy malejacy fragment
        } else {
            // Dodajemy liczbe podciagów dla zakończonego fragmentu
            count += (length * (length - 1)) / 2;
            length = 1; // Resetujemy długość fragmentu
        }
    }

    // Dodajemy podciagi z ostatniego fragmentu
    count += (length * (length - 1)) / 2;

    return count;
}
```

Rysunek 8: Funkcja `CountDecreasingSubarrays` zliczająca ilość podciągów malejących.

Funkcja `main` zawierająca wywołanie wszystkich innych funkcji prezentuje się następująco:

```
int main() {
    ifstream inputFile("input.txt");
    ofstream outputFile("output.txt");

    // Sprawdzamy, czy pliki zostały poprawnie otwarte
    if (!inputFile || !outputFile) {
        cout << "Blad otwarcia pliku!" << endl;
        return 1;
    }

    // Pierwsza liczba w pliku oznacza ilość tablic do stworzenia
    int numArrays;
    inputFile >> numArrays;

    for (int i = 0; i < numArrays; ++i) {
        int n;
        inputFile >> n;

        // Sprawdzamy, czy ta liczba jest 0. Jeśli tak to tablica jest pusta
        if (n <= 0) {
            outputFile << "Tablica " << i + 1 << " jest pusta." << endl;
            continue;
        }

        int *arr = new int[n];  // Dynamiczna alokacja pamięci dla tablicy

        for (int j = 0; j < n; ++j) {
            inputFile >> arr[j];
        }
```

Rysunek 9: Wywołanie funkcji w `main`.

```
        // Wypisanie podanej przez użytkownika tablicy
        outputFile << "Podciagi malejace w tablicy [";
        for (int j = 0; j < n; ++j) {
            outputFile << arr[j];
            if (j < n - 1) {
                outputFile << ", ";
            }
        }
        outputFile << "]:" << endl;

        // Wyznaczenie podciągów tablicy
        Subarrays(arr, n, outputFile);

        delete[] arr;  // Zwolnienie pamięci
        outputFile << endl;
    }

    cout << "Uruchamianie testow:" << endl;
    RunTests(); // Wywołanie funkcji testującej

    inputFile.close();
    outputFile.close();

    return 0;
}
```

Rysunek 10: Wypisanie podciągów przed wyliczeniem i wyznaczeniem podciągów malejących.

3 Podstawy teoretyczne

3.1 Złożoność czasowa, pamięciowa i obliczeniowa

Przejdźmy teraz do złożoności obliczeniowej. Złożoność ta określa jak wielką ilość zasobów potrzeba do rozwiązania problemu obliczeniowego. Rozważmy teraz te kwestię dla obu wersji programu.

3.1.1 Złożoności dla pierwszej wersji programu

1. Złożoność czasowa:

   Główna część obliczeń znajduje się w funkcji `Subarrays`. Przeanalizujmy ją szczegółowo:

   - Pętla zewnętrzna ze zmienną `i` iteruje przez elementy tablicy, czyli O(n).
   - Wewnętrzne przetwarzanie podciągów (gdy kończy się sekwencja malejąca):
     - Pierwsza pętla: `j` iteruje od `start` do `i-1`, czyli w najgorszym przypadku O(n).
     - Druga pętla: `k` iteruje od `j+1` do `i`, co daje `O(n-j)` iteracji, ale w najgorszym przypadku również O(n).
     - Trzecia pętla: `l` iteruje od `j` do `k`, czyli O(n) w najgorszym przypadku.

   Łączna liczba iteracji w najgorszym przypadku dla wszystkich trzech pętli to $O(n^3)$.

   Ostatni podciąg (po zakończeniu głównej pętli): Podobnie jak poprzedni fragment, jego złożoność w najgorszym przypadku wynosi $O(n^3)$.

   Podsumowując:

   Dla każdej tablicy, złożoność funkcji `Subarrays` wynosi $O(n^3)$. Zakładając, że mamy `m` tablic, każda o maksymalnym rozmiarze `n`, złożoność wynosi: $O(m*n^3)$.

2. Złożoność pamięciowa

   Główne aspekty pamięci programu:

   - Tablica wejściowa `arr`: Dla każdej tablicy deklarowanej dynamicznie w pętli `n` elementów, potrzebujemy O(n) pamięci.
   - Alokacja w funkcji `Subarrays`: Żadne dodatkowe struktury nie są tworzone poza zmiennymi lokalnymi `start`, `count`, itp., więc zużycie pamięci dla funkcji jest O(1).
   - Łączne zużycie pamięci: Zużycie pamięci przez funkcję dla każdej tablicy to O(n) (dane wejściowe). Przy `m` tablicach, całkowite zużycie pamięci to: O(n+m).

3. Złożoność obliczeniowa

   Złożoność obliczeniowa jest zgodna z analizą czasową, ponieważ każda iteracja lub operacja przyczynia się do złożoności czasowej. Podstawowa operacja to sprawdzenie warunku i zapis do pliku. Podsumowanie:

   - Główna złożoność: $O(m*n^3)$.
   - Każda iteracja i zapis do pliku stanowi jednostkę obliczeniową.

Wnioski:

1. Czasowa: $O(m*n^3)$ – dominuje analiza podciągów.
2. Pamięciowa: O(n+m) – wynika z tablicy wejściowej i liczby tablic.
3. Obliczeniowa: $O(m*n^3)$ – proporcjonalna do czasu.

3.1.2 Złożoności dla drugiej wersji programu

Jako iż program korzysta z funkcji `Subarrays` jego złożoność obliczeniowa nie uległa zmianie. Natomiast gdyby zmienić program tak, aby wypisywał tylko i wyłącznie liczbę malejących podciągów, a co za tym idzie korzystał tylko z funkcji `CountDecreasingSubarrays`, to jego złożoność wyglądałaby następująco:

1. Złożoność czasowa: Zamiast generować wszystkie malejące podciągi i wypisywać je osobno, możemy iterować po tablicy i zapisywać tylko informacje o liczbie podciągów malejących. Wystarczy przechodzić po tablicy i dla każdej sekwencji malejącej obliczyć liczbę podciągów za pomocą wzoru:

   $$Liczba\_podciagow = \frac{length \cdot (length - 1)}{2}$$

   gdzie length to długość bieżącej sekwencji malejącej. Iterujemy przez tablicę raz, więc złożoność wynosi: O(n).

2. Złożoność pamięciowa: Nie tworzymy żadnych dodatkowych struktur, poza zmiennymi licznikami, zatem złożoność pamięciowa wynosi: O(1).

3. Złożoność obliczeniowa: Podobnie jak poprzednio, w tym przypadku złożoność obliczeniowa odpowiada złożoności czasowej, zatem wynosi ona: O(n)

Wnioski:

1. Czasowa: O(n) – jednorazowe przejście pętli.

2. Pamięciowa: O(1) – nie tworzymy żadnych dodatkowych zmiennych

3. Obliczeniowa: O(n) – proporcjonalna do czasu.

Jak widzimy sama funkcja `CountDecreasingSubarrays` ma znacznie mniejszą złożoność, a co za tym idzie, w przypadku pliku z większą ilością tablic skończy działanie szybciej (chociaż zrobi to bez wypisania malejących podciągów).

4 Szczegóły implementacji

Programy zostały zaimplementowane w dwóch oddzielnych plikach:

- Program `Podciagi_file.cpp` - implementuje pierwszą wersję kodu (funkcja `Subarrays`).
- Program `Podciagi_Optymalizacja_1.cpp` - implementuje drugą wersję kodu (funkcja `Subarrays`, `RunTests` oraz `CountDecreasingSubarrays`).

Oba te programy zostały przedstawione wcześniej przy pomocy opisów, schematów blokowych, pseudokodu oraz zrzutów ekranu po utworzeniu kodu w języku C++, zatem przejdźmy teraz do testowania tych programów. Posłużymy się wcześniej zadeklarowanym plikiem `input.txt`, który wygląda następująco:

```
4
5 5 4 2 2 1
3 2 5 3
5 1 2 4 6 7
4 8 6 4 2
```

Po uruchomieniu pierwszego programu `Podciagi_file.cpp` otrzymujemy taki rezultat:

```
Podciagi malejace w tablicy [5, 4, 2, 2, 1]:
[5, 4] [5, 4, 2] [4, 2] [2, 1]
Liczba malejacych podciagow: 4

Podciagi malejace w tablicy [2, 5, 3]:
[5, 3]
Liczba malejacych podciagow: 1

Podciagi malejace w tablicy [1, 2, 4, 6, 7]:

Liczba malejacych podciagow: 0

Podciagi malejace w tablicy [8, 6, 4, 2]:
[8, 6] [8, 6, 4] [8, 6, 4, 2] [6, 4] [6, 4, 2] [4, 2]
Liczba malejacych podciagow: 6
```

Test został przeprowadzony na trzech przykładowych tablicach podanych w treści zadania oraz na tak zwanym "niewygodnym zestawie danych", czyli w naszym wypadku tablicy, która zawiera elementy malejące na każdym miejscu względem poprzedniego. W tym miejscu pojawia się sens zapisania funkcji `Subarrays` oraz `CountDecreasingSubarrays` w taki a nie inny sposób. Ostatni warunek oraz ostatnia pętla są odpowiedzialne za przypadek, w którym tak właśnie będą ułożone elementy w tablicy.

Uruchomimy teraz drugi program `Podciagi_Optymalizacja_1.cpp` i zobaczymy działanie wszystkich funkcji w praktyce:

```
Uruchamianie testow:
Test 1: PASSED (expected 4, got 4)
Test 2: PASSED (expected 1, got 1)
Test 3: PASSED (expected 0, got 0)
Test 4: FAILED (expected 1, got 10)
Test 5: PASSED (expected 0, got 0)
Test 6: PASSED (expected 0, got 0)
Test 7: PASSED (expected 7, got 7)
Test 8: PASSED (expected 1, got 1)
Test 9: FAILED (expected 7, got 6)

Process returned 0 (0x0)   execution time : 0.103 s
Press any key to continue.
```

Rysunek 11: Pierwsze testy drugiego programu (funkcja `RunTests`).

Na widocznym zrzucie ekranu z okna konsoli można zaobserwować wyniki przeprowadzonych testów, które zostały wcześniej utworzone w kodzie. Dla utworzonych tablic sprawdzamy, czy obliczona wartość malejących podciągów jest równa wartości oczekiwanej, czyli obliczonej przez nas liczbie. Jeśli te wartości się pokryją to program wypisze na ekranie `PASSED` wraz z oczekiwaną i otrzymaną wartością, ale jeśli obliczy wartość inną niż oczekiwaliśmy, to wyświetli `FAILED` wraz z obliczoną i oczekiwaną wartością (na potrzeby pokazania tej funkcji we wcześniejszej deklaracji testów dwie oczekiwane wartości zostały ustawione na niepoprawne).

```
// Przykładowe testy
TestCase tests[] = {
    {{5, 4, 2, 2, 1}, 5, 4},      // 4 malejace podciagi: [5,4], [5,4,2], [4,2], [2,1]
    {{2, 5, 3}, 3, 1},            // 1 malejacy podciag: [5,3]
    {{1, 2, 4, 6, 7}, 5, 0},      // Brak malejacych podciagów
    {{9, 7, 5, 3, 1}, 5, 1},     // 10 malejacych podciagów
    {{1}, 1, 0},                  // Tablica jednoelementowa: brak podciagów
    {{1, 1, 1}, 3, 0},            // Wszystkie elementy równe: brak malejacych podciagów
    {{8, 6, 6, 5, 3, 1}, 6, 7},   // Malejace podciagi z powtórzeniami
    {{10, 9}, 2, 1},              // Jeden malejacy podciag
    {{3, 2, 1, 2, 1, 0}, 6, 7},   // Kombinacja malejących i rosnacych sekwencji (6 malejacych)
};
```

Rysunek 12: Przypomnienie deklaracji przykładowych testów w funkcji `RunTests`.

Widać, że tam gdzie w komentarzu wypisana jest informacja o 10 malejących podciągach, w deklaracji oczekiwanej wartości jest wpisane 1, stąd pojawia się komunikat `FAILED` w oknie konsoli (analogicznie w przypadku testu numer 9).

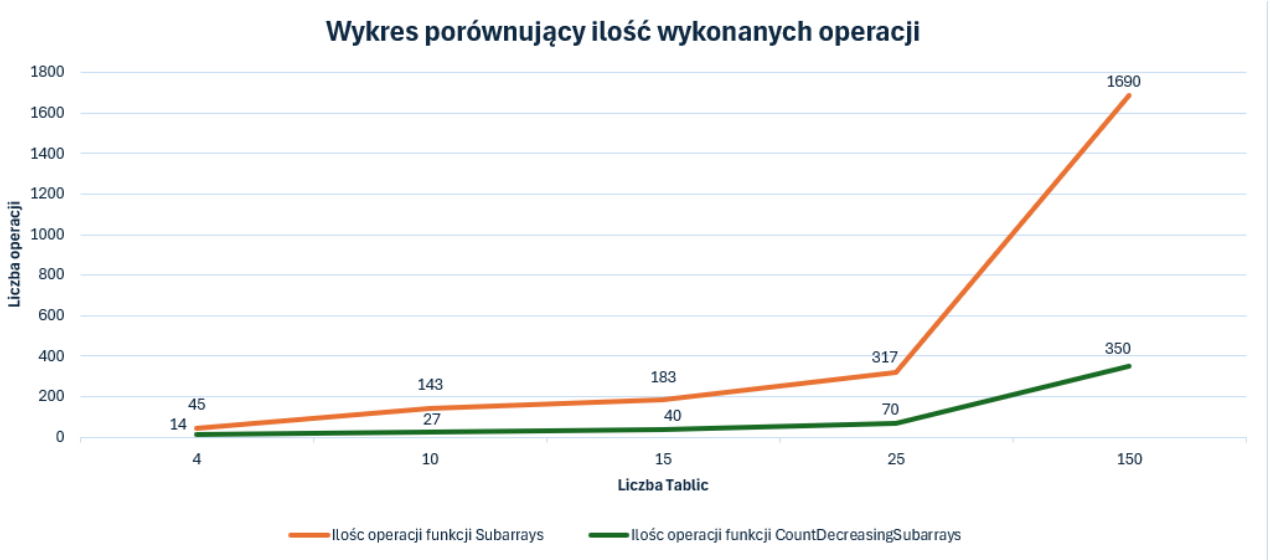
5 Testowanie

Na poniższym wykresie przedstawię porównanie liczby operacji w dwóch przypadkach:

1. Wykonanie zadanie w pełni tak, jak jest to oczekiwane (wypisanie wszystkich podciągów malejących oraz ich liczby).

2. Wykonanie zadania częściowo przy użyciu funkcji `CountDecreasingSubarrays` (wypisanie tylko liczby malejących podciągów.

   (**UWAGA:** Liczba operacji zostanie zliczona poprzez dodanie zmiennej pomocniczej, która zwiększy się o 1 przy każdym wykonaniu się danego fragmentu kodu).



Rysunek 13: Wykres porównujący ilość wykonanych operacji w zależności od liczby tablic.

Można zaobserwować, że funkcja `CountDecreasingSubarrays` odpowiedzialna tylko za obliczenie liczby podciągów malejących wykonała mniej operacji aniżeli funkcja `Subarrays`.

6 Podsumowanie i wnioski

Podczas realizacji tego projektu zobaczyliśmy jak można rozwiązać problem wyszukiwania malejących podciągów w tablicy oraz porównaliśmy dwa sposoby rozwiązywania jednego polecenia. Mimo tego, że złożoność obliczeniowa wyszła $O(n^3)$ to niestety bardzo trudno byłoby osiągnąć mniejszą złożoność, zakładając że chcemy, aby program spełniał wszystkie wymogi. Z tego też powodu program może się nie nadawać do pracy z dużą ilością tablic. Z naszych testów także wynika, że jeśli zależy nam tylko na uzyskaniu odpowiedzi na pytanie "Ile jest malejących podciągów?", to warto jest zastosować funkcje, które są bardziej wydajne i mniej złożone.