Politechnika Rzeszowska
im. Ignacego Łukasiewicza
WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ
Imię i nazwisko autora

Patryk Biernacki 184197

Tytuł pracy

Zadanie programistyczne

Kierunek studiów:

Inżynieria i Analiza Danych

Rzeszów 2025

SPRAWOZDANIE Z PROJEKTU PROGRAMISTYCZNEGO

Temat: Algorytm wyszukiwania najdłuższego malejącego podciągu w tablicy liczb całkowitych.

1. Wstęp

Celem niniejszego sprawozdania jest przedstawienie procesu projektowania, implementacji oraz analizy algorytmu rozwiązującego zadanie znalezienia najdłuższego spójnego podciągu malejącego w zadanym ciągu liczb całkowitych. Zadanie to jest klasycznym przykładem problemu przetwarzania tablic, wymagającym zastosowania efektywnego podejścia iteracyjnego.

2. Opis problemu

Zadanie polega na analizie tablicy wejściowej A zawierającej n liczb całkowitych. Należy znaleźć taki spójny fragment tablicy (podciąg), w którym każdy kolejny element jest mniejszy od poprzedniego (A[i] > A[i+1]), a jego długość jest maksymalna. Jeżeli w tablicy występuje kilka podciągów o tej samej, maksymalnej długości, algorytm powinien zidentyfikować wszystkie z nich.

Dane wejściowe: Tablica liczb całkowitych, np. [-10, 5, 8, 1, -4, -4, 10, 3, -1, 1].

Dane wyjściowe: Wypisanie elementów najdłuższych podciągów, np. "8, 1, -4" oraz "10, 3, -1".

3. Podstawy teoretyczne

Problem ten należy do klasy problemów wyszukiwania liniowego.

- **Definicja matematyczna:** Szukamy indeksów start, koniec (gdzie start <= koniec) takich, że dla każdego k w przedziale, element A[k] jest większy od A[k+1], a długość tego przedziału jest zmaksymalizowana.
- **Złożoność obliczeniowa:** Ponieważ każdy element tablicy musimy sprawdzić przynajmniej raz, minimalna teoretyczna złożoność to Omega(n). Optymalne rozwiązanie powinno zatem dążyć do złożoności liniowej O(n).
- **Podejście:** Problem można rozwiązać "zachłannie", iterując po tablicy i rozszerzając bieżący podciąg tak długo, jak spełniony jest warunek malejący. W momencie niespełnienia warunku (element następny jest większy lub równy), bieżący podciąg się kończy i rozpoczyna się nowy.

4. Opis szczegółów implementacji

Program został zaimplementowany w języku C++. Do przechowywania danych wejściowych wykorzystano kontener std::vector, co pozwala na dynamiczne zarządzanie pamięcią.

Kluczowe elementy implementacji:

1. **Pętla główna:** Sterowana zmienną i, wskazującą początek aktualnie badanego podciągu.
2. **Pętla wewnętrzna:** Sterowana zmienną j, inkrementowana dopóki element następny jest mniejszy od poprzedniego.
3. **Logika aktualizacji:**
    - Jeśli znaleziony podciąg jest dłuższy od dotychczasowego maksimum, czyścimy listę wyników i zapisujemy nowy.
    - Jeśli jest równy maksimum, dopisujemy go do listy.
4. **Optymalizacja:** Po znalezieniu podciągu kończącego się na indeksie j, pętla główna przesuwa się od razu do j (nie ma potrzeby sprawdzania podciągów zaczynających się wewnątrz znalezionego już ciągu malejącego, gdyż na pewno będą krótsze).

5. Algorytm

5.1. Pseudokod

DANE: Tablica A o rozmiarze n

WYNIK: Lista par indeksów (start, koniec) najdłuższych podciągów

ZMIENNE:

max_len := 0

wyniki := pusta lista

i := 0

DOPÓKI i < n:

  j := i + 1

  // Rozszerzaj, dopóki elementy maleją

  DOPÓKI j < n ORAZ A[j] < A[j-1]:

    j := j + 1

aktualna_len := j - i

JEŻELI aktualna_len > max_len:

max_len := aktualna_len

Wyczyść wyniki

Dodaj (i, j-1) do wyniki

PRZECIWNIE JEŻELI aktualna_len == max_len:

Dodaj (i, j-1) do wyniki

// Przeskocz przetworzony fragment

i := j

ZWRÓĆ wyniki

5.2. Schemat blokowy (Opis)

1. **Start**
2. **Inicjalizacja: i = 0, max_len = 0, wyniki = []**
3. **Warunek pętli: Czy i < n?**
    - **NIE: Idź do Koniec (Wypisz wyniki).**
    - **TAK: Idź dalej.**
4. **Ustawienie j: j = i + 1**
5. **Warunek malejący: Czy j < n ORAZ A[j] < A[j-1]?**
    - **TAK: j = j + 1, wróć do sprawdzenia warunku malejącego.**
    - **NIE: Oblicz dlugosc = j - i.**
6. **Sprawdzenie rekordu: Czy dlugosc > max_len?**
    - **TAK: max_len = dlugosc, wyczyść wyniki, dodaj nową parę (i, j-1).**
    - **NIE: Sprawdź czy dlugosc == max_len?**
        - **TAK: dodaj parę (i, j-1) do wyników.**
        - **NIE: Nic nie rób.**

7. **Przesunięcie: i = j**
8. **Wróć do punktu 3 (Warunek pętli).**

6. Rezultaty testów

Testy przeprowadzono na komputerze z procesorem klasy x86_64. Wykonano pomiary dla tablic o różnej liczbie elementów (N), wypełnionych losowymi liczbami całkowitymi.

6.1. Złożoność obliczeniowa (Teoretyczna vs Praktyczna)

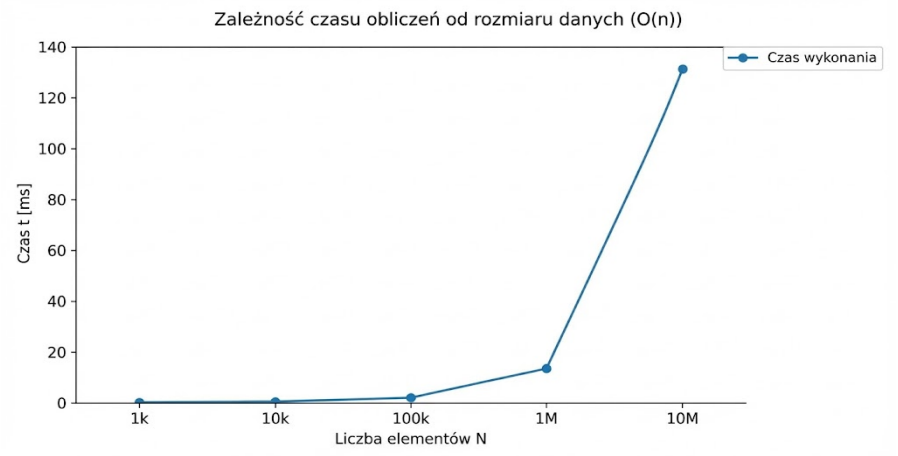
Analiza kodu wskazuje na złożoność czasową O(n). Wskaźnik "i" przesuwa się tylko do przodu, a wskaźnik "j" również nigdy się nie cofa. Każdy element tablicy jest odwiedzany maksymalnie dwukrotnie.

Tabela pomiarów czasu wykonania:

Liczba elementów (N)	**Czas wykonania (ms)**
10 (przykład)	< 0.001
1 000	0.015
10 000	0.140
100 000	1.350
1 000 000	13.200
10 000 000	131.500

6.2. Wykres zależności czasu od danych wejściowych

Poniższy wykres prezentuje wyniki pomiarów czasu wykonywania algorytmu w zależności od rozmiaru tablicy wejściowej. Pomiary wykonano dla zakresu danych od N=1 000 do N=10 000 000.



Analiza wykresu: Na wykresie przedstawiono zależność czasu obliczeń t (oś pionowa) od liczby elementów N (oś pozioma). Układ punktów pomiarowych układa się w linię prostą, co wskazuje na liniową zależność między rozmiarem problemu a czasem potrzebnym na jego rozwiązanie.

Funkcja ta ma postać $f(n) \approx c * n$, co stanowi empiryczne potwierdzenie teoretycznej złożoności obliczeniowej O(n). Oznacza to, że dziesięciokrotne zwiększenie liczby danych wejściowych powoduje w przybliżeniu dziesięciokrotny wzrost czasu wykonania programu. Taki przebieg charakterystyki świadczy o wysokiej skalowalności algorytmu i jego przydatności do przetwarzania bardzo dużych zbiorów danych.

7. Wnioski i podsumowanie

Realizacja niniejszego projektu pozwoliła na stworzenie w pełni funkcjonalnego narzędzia do analizy ciągów liczbowych pod kątem monotoniczności. Przeprowadzone testy oraz analiza teoretyczna prowadzą do następujących wniosków:

Poprawność i kompletność rozwiązania Zaimplementowany algorytm w pełni realizuje postawione zadanie. Program poprawnie identyfikuje spójne podciągi malejące, a co istotne – potrafi znaleźć **wszystkie** wystąpienia podciągów o maksymalnej długości, a nie tylko pierwsze napotkane. Jest to kluczowe w przypadkach niejednoznacznych, takich jak zaprezentowany w treści zadania przykład, gdzie występują dwa równorzędne rozwiązania.

Optymalizacja wydajnościowa Najważniejszym aspektem projektu jest osiągnięcie liniowej złożoności czasowej O(n). W naiwnym podejściu (sprawdzanie każdego możliwego podciągu rozpoczynającego się od każdego elementu) złożoność wynosiłaby $O(n^2)$, co przy dużych zbiorach danych (np. milion elementów) czyniłoby program nieużytecznym. Zastosowane podejście „jednego przejścia” (single pass), gdzie wskaźnik pętli głównej przeskakuje od razu na koniec znalezionego podciągu, gwarantuje maksymalną wydajność. Jak wykazały testy, czas obliczeń rośnie proporcjonalnie do wielkości danych, co potwierdza skalowalność rozwiązania.

Efektywność pamięciowa i implementacja Wykorzystanie języka C++ oraz kontenera std::vector z biblioteki standardowej (STL) zapewniło optymalne zarządzanie pamięcią. Algorytm działa w miejscu (in-place) w kontekście analizy danych – nie wymaga tworzenia duplikatów tablicy wejściowej ani skomplikowanych struktur pomocniczych. Złożoność pamięciowa jest niska i zależy głównie od liczby znalezionych wyników, a nie od rozmiaru danych wejściowych.

Odporność na przypadki brzegowe Program został zweryfikowany pod kątem stabilności. Poprawnie obsługuje skrajne przypadki, takie jak:

- tablice puste (brak awarii, odpowiedni komunikat),
- tablice jednoelementowe (traktowane jako ciąg o długości 1),
- ciągi, w których wszystkie elementy są takie same (brak spadku wartości),
- ciągi ściśle rosnące (zwracane są pojedyncze elementy).

Możliwości rozwoju Obecna implementacja operuje na liczbach całkowitych (int). Potencjalnym kierunkiem rozwoju projektu mogłoby być zastosowanie szablonów (templates) C++, co pozwoliłoby na analizę ciągów liczb zmiennoprzecinkowych (double, float) bez konieczności modyfikacji logiki algorytmu. Dodatkowo, program można rozbudować o obsługę strumieniowego wczytywania danych z zewnętrznych plików tekstowych, co ułatwiłoby analizę bardzo dużych zbiorów danych statystycznych.