Algorytmy i struktury danych

Karol Garbacz

29 stycznia 2025

Zadanie projektowe

Spis treści

1. Treść zadania

Zastąp każdy element występujący w zadanej tablicy liczbą równą liczbie elementów większych od niego i leżących na lewo od niego.

Przykład

Wejście

$[4, 5, 0, 2, 4, 9]$

Wyjście

$[0, 0, 2, 2, 1, 0]$

2. Rozwiązywanie zadania

2.1 Rozwiązanie

2.1.1 Analiza problemu

Zaczniemy od utworzenia kopii tablicy, po to aby po zastąpieniu któregoś elementu można było odnosić się oryginalnych danych. Następnie sprawdzimy dla każdego elementu, ile jest liczb większych od niego i znajdujących się na lewo od niego. W tym celu zapuścimy sobie pętle która iteruje przez tablicę, a wewnątrz niej zainicjalizujemy zmienną ll i przypiszemy jej wartość 0. Zmienna ta tymczasowo będzie przechowywała ilość liczb, która znajduje się na lewo oraz jest większa od bieżącego elementu. Następnie zapuścimy drugą pętlę, która również będzie iterowała przez tą tablicę, lecz z taką różnicą, że będzie sprawdzała czy kolejne elementy tablicy znajdujące się na lewo od rozpatrywanego elementu są większe od niego. Jeśli tak, to inkrementujemy licznik o jeden. Po zakończeniu pętli wewnętrznej przypisujemy wartość licznika elementowi tablicy o indeksie takim jak w pętli zewnętrznej.

Dodatkowo, w sytuacji, gdy podany ciąg składa się z jednego elementu, to wartość po zastąpieniu wyniesie 0, a gdy ciąg będzie składał się z mniej niż jednego elementu, to pętla nie zostanie rozpoczęta.

1. Dane wejściowe

   Danymi wejściowymi naszego algorytmu uczynimy:

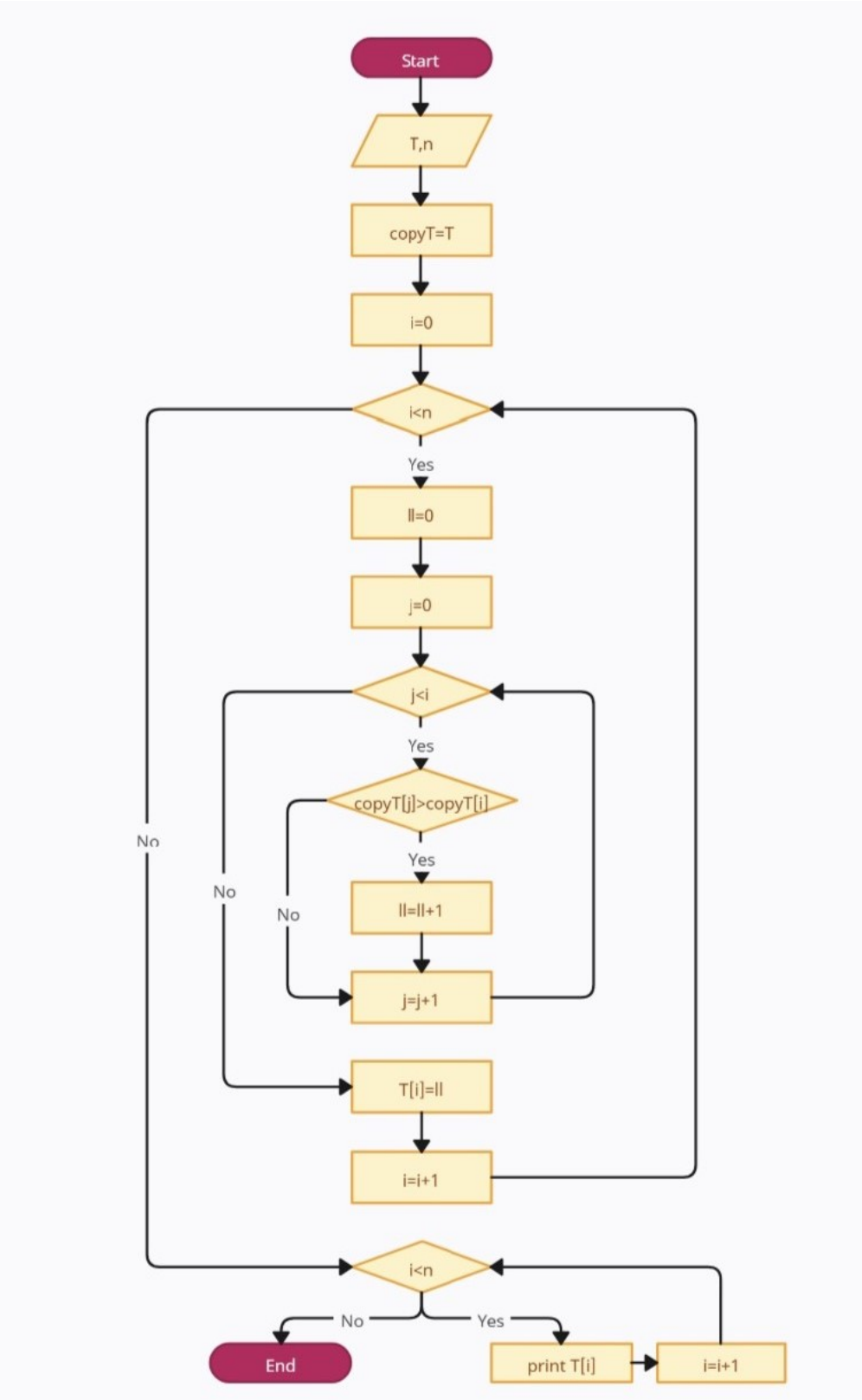
   - strukturę danych typu tablica (zmienna T), przechowującą wartości zadanego ciagu, oraz
   - długość tego ciagu (zmienna n).

2. Dane wyjściowe

Algorytm zwróci wszytkie elementy tablicy T.

2.1.2 Schemat blokowy algorytmu

Algorytm zapisany w postaci schematu blokowego Rysunek (1) mógłby przedstawiać się następująco:



Rysunek 1: Schemat blokowy algorytmu

Zauważmy, że w algorytmie pojawiają się trzy zmienne pomocnicze, które należy odpowiednio zainicjalizować. Dwie z nich: i oraz j będą wskaźnikami ułatwiającymi poruszanie się po tablicy. Trzecia zmienna - ll to zmienna przechowująca tymczasową wartość licznika liczb stojących na lewo oraz większych od rozpatrywanej.

2.1.3 Algorytm zapisany w pseudokodzie

```
 1  input: T    //tablica przechowujaca wartosci ciagu
 2         n    //dlugosc tablicy
 3  output: T   //tablica po zastąpieniu oryginalnych elementów
 4  copyT := T  // Kopiowanie oryginalnej tablicy do copyT
 5  i:=0 do n-1
 6  j:=0
 7
 8  while i<n
 9      ll=0
10      while j<i
11          if( copyT[j] > copyT[i] )
12              ll=ll+1
13              j=j+1
14          endif
15      endwhile
16      T[i]:=ll
17      i=i+1
18  endwhile
19  while i<n
20      print T[i]
21  endwhile
```

2.1.4 Sprawdzenie poprawności algorytmu poprzez „ołówkowe"rozwiązanie problemu

Aby przekonać się, że zapronowany algorytm rzeczywiście jest w stanie rozwiązać zadany problem wystarczy „,kartka i ołówek". Przeanalizowanie jego działania, krok po kroku, zgodnie tym co zapisane zostało w postaci pseudokodu pozwala wykryć trywialne błędy i niespójności jeszcze na początkowym etapie pracy nad projektem i ograniczyć czas spędzony nad samą implementacją.

Prezentacja poprawnego wykonania tej części zadania zostanie przedstawiona na przykładzie danych:
Wejście $= \begin{bmatrix} 4 & 5 & 0 & 2 & 4 & 9 \end{bmatrix}$
Obliczenia zgodnie z algorytmem przedstawia poniższa tabela (1) (wartości zmiennych i poszczególnych wyników pośrednich z kolejnych iteracji są umieszczone w wierszach).

i	j	copyT[i]	copyT[j]	copyT[j] > copyT[i]	ll	T[i]
0	-	-	-	-	0	0
1	0	5	4	0	0	0
2	0	0	4	1	1	-
2	1	0	5	1	2	2
3	0	2	4	1	1	-
3	1	2	5	1	2	-
3	2	2	0	0	2	2
4	0	4	4	0	0	-
4	1	4	5	1	1	-
4	2	4	0	0	1	-
4	3	4	2	0	1	1
5	0	9	4	0	0	-
5	1	9	5	0	0	-
5	2	9	0	0	0	-
5	3	9	2	0	0	-
5	4	9	4	0	0	0

Tabela 1: Tabela z "ołówkowym"rozwiązaniem

2.1.5 Teoretyczne oszacowanie złożoności obliczeniowej.

Analizując algorytm można zauważyć, że podstawową jego operacją będzie porównanie i-tej i j-tej wartości ciagu/tabeli. Łatwo policzyć ile operacji tego rodzaju zostanie wykonanych:

1. Kopiowanie wektora.
   Operacja kopiowania wektora T do copyT ma złożoność $O(n)$, ponieważ musimy skopiować każdy element z T do copyT.

2. Pętla zewnętrzna.
   Ta pętla wykonuje iteracje n razy i wykonuje operację wypisywania każdego elementu z T. Złożoność tej pętli to $O(n)$.

3. Pętla zewnętrzna i wewnętrzna.

   - Zewnętrzna pętla działa n razy (od $i = 0$ do $i = n - 1$).
   - Wewnętrzna pętla działa dla każdego i od j = 0 do j = i-1. Oznacza to, że liczba iteracji wewnętrznej pętli zależy od wartości i. Dla każdego i pętla wewnętrzna będzie wykonywać i iteracji.

   Łączna liczba operacji w pętli wewnętrznej i zewnętrznej wynosi:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

   możemy policzyć, że dla ciagu o n elementach algorytm będzie musiał wykonać

$$\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

   porównań.

4. Druga pętla.
   Ta pętla działa identycznie jak pierwsza pętla i ma złożoność $O(n)$.

   Powiemy więc, że złożoność czasowa algorytmu wynosi $O\left(n^2\right)$, a więc czas jego wykonania będzie proporcjonalny do kwadratu liczby danych wejściowych.

2.2 Implementacja algorytmu

2.2.1 Prosta implementacja

Przykład tego rodzaju programu wraz z wynikiem jego działania przedstawia się następująco:

```
#include <iostream>
#include <vector>

void funkcja(std::vector<int>& T, int& n)
{
 std::vector<int> copyT = T;// Kopia oryginalnej tablicy

 for (int i = 0; i < n; i++)
 {
  int ll = 0; //Licznik liczb leżących na lewo od rozpatrywanej
  for (int j = 0; j < i; j++)
  {
   if (copyT[j] > copyT[i])
    ll++;
  }
  T[i] = ll;
 }
 for (int i = 0; i < n; i++)
  std::cout << T[i] << " ";
}

int main()
{
 int n = 0;
 int value = 0;
 std::vector<int> T;
 std::cout << "Podaj liczbe elementow: \n";
 std::cin >> n;
 for (int i = 0; i < n; i++)
 {
  std::cout << "Podaj element: \n";
  std::cin >> value;
  T.push_back(value);
 }
 for (int i = 0; i < n; i++)
  std::cout << T[i] << " ";
 std::cout << "\n";
 funkcja(T,n);
 return 0;
}
```

2.2.2 Testy "niewygodnych"zestawów danych

Analizując podany kod, nie ma w nim bezpośrednich instrukcji powodujących awarię programu przy standardowych danych wejściowych.

2.2.3 Testy wydajności algorytmów

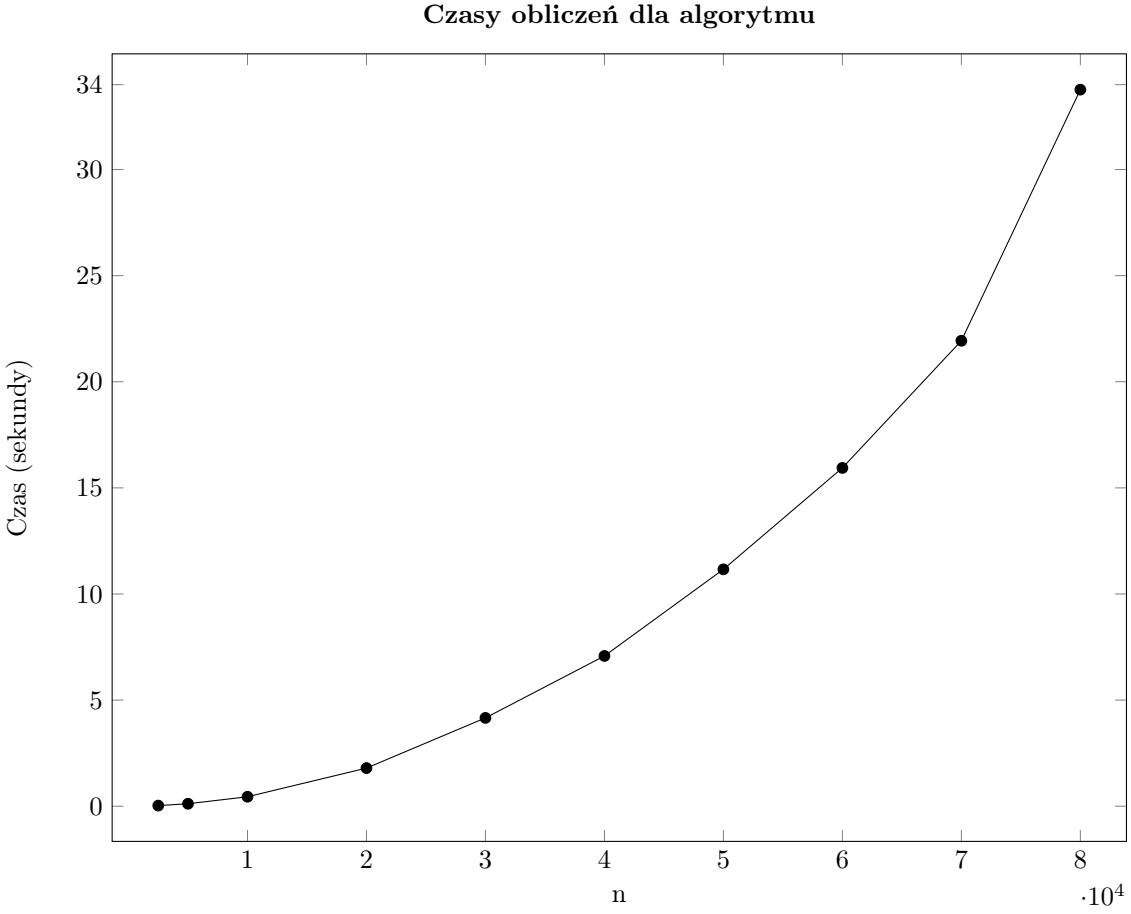
Zaczniemy od zmodyfikowania funkcji main tak, aby wygenerować liczby losowe, dla różnych wielkości zestawów danych.

```
#include <iostream>
#include <vector>
#include <chrono>

void funkcja(std::vector<int>& T, int n)
{
    std::vector<int> copyT = T; // Kopia oryginalnej tablicy

    for (int i = 0; i < n; i++)
    {
        int ll = 0; // Licznik liczb leżących na lewo od rozpatrywanej
        for (int j = 0; j < i; j++)
        {
            if (copyT[j] > copyT[i])
                ll++;
        }
        T[i] = ll;
    }
}

int main()
{
    // Lista wartości n
    std::vector<int> sizes = { 2500, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000 };

    for (int n : sizes)
    {
        // Wypełniamy wektor T liczbami losowymi
        std::vector<int> T;
        for (int i = 0; i < n; i++)
        {
            T.push_back(rand()%1000);
        }
        // Mierzenie czasu
        auto start = std::chrono::high_resolution_clock::now();
        funkcja(T, n);
        auto end = std::chrono::high_resolution_clock::now();

        std::chrono::duration<double> duration = end - start;
        std::cout << "Czas dla n = " << n << ": " << duration.count() << " sekund" << std::endl;
    }
    return 0;
}
```

Program wygenerował nam wyniki:

```
Czas dla n = 2500: 0.0286573 sekund
Czas dla n = 5000: 0.115216 sekund
Czas dla n = 10000: 0.445449 sekund
Czas dla n = 20000: 1.79555 sekund
Czas dla n = 30000: 4.16089 sekund
Czas dla n = 40000: 7.07953 sekund
Czas dla n = 50000: 11.1622 sekund
Czas dla n = 60000: 15.9408 sekund
Czas dla n = 70000: 21.9326 sekund
Czas dla n = 80000: 33.7645 sekund
```

Rysunek 2: Wyniki wydajności algorytmu



2.3 Wnioski

Złożoność czasowa tego algorytmu wynosi $O(n)$, ponieważ dla każdego elementu w wektorze (w pętli zewnętrznej) sprawdzamy wszystkie poprzednie elementy (w pętli wewnętrznej). Złożoność kwadratowa sprawia, że program może działać wolno dla dużych rozmiarów wektora.

Algorytm jest mało wydajny przy dużych danych wejściowych (np. dla $n > 1000$), ze względu na złożoność $O(n)$.

3. Podsumowanie

Próby optymalizacji nie przyniosły oczekiwanego rezultatu. Program poprawnie realizuje cel obliczeniowy, ale nie jest odpowiedni dla bardzo dużych danych wejściowych ze względu na złożoność kwadratową.

A. Kod programu

```
#include <iostream>
#include <vector>

void funkcja(std::vector<int>& T, int& n)
{
 std::vector<int> copyT = T;// Kopia oryginalnej tablicy

 for (int i = 0; i < n; i++)
 {
  int ll = 0; //Licznik liczb leżących na lewo od rozpatrywanej
  for (int j = 0; j < i; j++)
  {
   if (copyT[j] > copyT[i])
    ll++;
  }
  T[i] = ll;
 }

 for (int i = 0; i < n; i++)
  std::cout << T[i] << " ";
}

int main()
{
 int n = 0;
 int value = 0;

 std::vector<int> T;

 std::cout << "Podaj liczbe elementow: \n";
 std::cin >> n;
 for (int i = 0; i < n; i++)
 {
  std::cout << "Podaj element: \n";
  std::cin >> value;
  T.push_back(value);
 }

 for (int i = 0; i < n; i++)
  std::cout << T[i] << " ";


 std::cout << "\n";
 funkcja(T,n);
 return 0;
}
```