Algorytmy i struktury danych

Jak poprawnie rozwiązać zadanie projektowe

mb MB

10 listopada 2023

Spis treści

Streszczenie

Celem tej instrukcji jest przedstawienie poprawnego sposobu rozwiązywania zadania projektowego nr 1.

Niech dane będzie następujące zadanie:

1. Treść zadania

Znajdź maksymalną różnicę pomiędzy dwoma elementami występujacymi w ciągu liczb naturalnych, pod warunkiem, ze element mniejszy występuje przed elementem większym.

Przyklad:

```
we = [10 4 2 9 5]
```

-> 9-2 = 7

2. Etapy rozwiązywania problemu.

W trakcie rozwiązywania zagadnienia należy zachować **właściwą** kolejność kolejnych podzadań, która przedstawia się następująco:

1. Analiza problemu.
2. Utworzenie schematu blokowego.
3. Utworzenie programu zapisanego w pseudokodzie.
4. Analiza działania zapisanego algorytmu na kartce papieru (Rozwiązanie zadania przykładowego).
5. Teoretyczne oszacowanie złożoności obliczeniowej.
6. Ponowne przemyślenie problemu i próba wymyślenia algorytmu wydajniejszego.
7. Powtórzenie punktów 2-5 dla algorytmu wydajnieszego.
8. Powtórzenie punktów 6-7 wybraną przez siebie liczbę razy.
9. Implementacja wymyślonych algortymów w wybranym środowisku i języku.
10. Testy numeryczne działania algorytmów i potwierdzenie doświadczalne wyników teoretycznych.

Wymagania:

- zachowanie kolejności rozwiązywania problemu
- co najmniej dwie wersje algorytmu rozwiazujacego
- schemat blokowy wraz z pokazaniem jego zastosowania na kartce papieru
- implementacja w C oraz w C++ na tablicach dynamicznych oraz na vectorach.
- teoretyczne oszacowanie złożoności obliczeniowej
- eksperymenty pokazujące złożoność czasową

2.1. Rozwiązanie - podejście pierwsze (brute force)

2.1.1. Analiza problemu

W pierwszym podejściu postąpimy „siłowo” i zaimplementujemy rozwiązanie, które „narzuca się” samo. Utworzymy zmienną pomocniczą `roznica_max`, w której będziemy przechowywać aktualną wartość maksymalnej różnicy. Następnie, metodycznie, sprawdzając parami każde dwa elementy ciągu (pod warunkiem, że mniejszy z nich znajduje się przed większym) obliczymy różnicę pomiędzy nimi i jeśli będzie ona większa od wartości przechowywanej w zmiennej `roznica_max` to przypiszemy ją do tej zmiennej.

Należy w tym miejscu zwrócić uwagę na problem inicjalizacji zmiennej `roznica_max` właściwą wartością. Jaką wartość należy przypisać tej zmiennej przed przystąpieniem do analizy zadanego ciągu?

Zgodnie z warunkami zadania, ciąg dany na wejściu składa się z liczb naturalnych. Można założyć, że w większości przypadków da się znaleźć wartość spełniającą warunki zadania, np. dla `we = [2 3 5 1]`, różnica ta wynosi $5 - 2 = 3$. Ale należy również rozważyć przypadek „nieszczęśliwej” kolejności danych wejściowych, które mogą utworzyć ciąg nierosnący np. `we = [8 5 4 2 1]`, czy `we = [2 2 2]`. Na potrzebę uwzględnienia tego typu sytuacji zmienną `roznica_max` inicjalizujemy wartością `-1`. Jeśli algorytm zwróci taką wartość, będzie to oznaczało, że w ciągu podanym na wejście nie znalazł ani jednej pary, które spełniała kryteria zadane w treści

zadania. (Zauważmy, że wartością inicjalizującą zmienną `różnica_max` może być dowolna liczba mniejsza niż 1, również 0, a przyjęta przez nas wartość jest po prostu kwestią umowy.)

Inną rzeczą, którą powinien uwzględniać algorytm, jest sytuacja, gdy podany ciąg składa się z mniej niż dwóch elementów. Na ten wypadek do algorytmu zostanie dodana instrukcja warunkowa `if`, która nie pozwoli na rozpoczęcie pętli po „zbyt krótkiej" tablicy.

1. Dane wejściowe

   Danymi wejściowymi naszego algorytmu uczynimy:

   - strukturę danych typu tablica (zmienna `tab`), przechowującą wartości zadanego ciągu, oraz
   - długość tego ciągu (zmienna `n`).

2. Dane wyjściowe

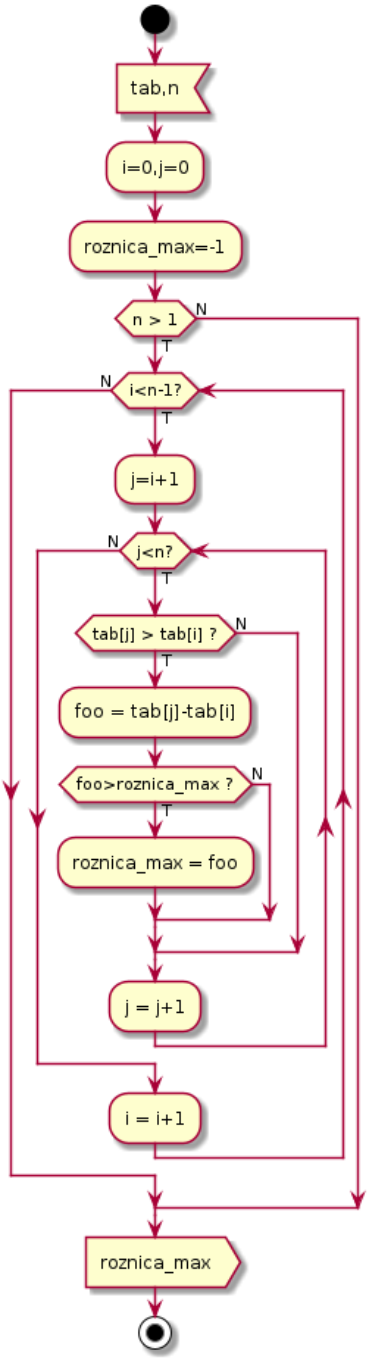
   Algorytm zwróci wartość rozwiązania w zmiennej `roznica`.

2.1.2. Schemat blokowy algorytmu

Uwaga: 2.1

W prezentowanych poniżej schematach blokowych i pseudokodzie założono, że tablice indeksowane są tak jak w języku C/C++, począwszy od wartości 0.

Algorytm zapisany w postaci schematu blokowego mógłby przedstawiać się następująco:



Rysunek 1: Schemat blokowy algorytmu - wersja naiwna

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 - `foo` to zmienna przechowująca tymczasową wartość różnicy pomiędzy `i`-tym oraz `j`-tym elementem ciągu (tablicy).

2.1.3. Algorytm zapisany w pseudokodzie

Uwaga: 2.2

Za polską wikipedią(podkreślenia - MB):
Pseudokod – sposób zapisu algorytmu, w którym, zachowując strukturę charakterystyczną dla kodu zapisanego w języku programowania, rezygnuje się ze ścisłych reguł składniowych na rzecz prostoty i czytelności. Pseudokod nie zawiera szczegółów implementacyjnych (jak np. inicjalizacja zmiennych, alokacja pamięci itp.), często też opuszcza się w nim opis działania podprocedur (jeśli powinien być on oczywisty dla czytelnika), zaś nietrywialne kroki algorytmu opisywane są z pomocą formuł matematycznych lub zdań w języku naturalnym.
Nie istnieją w chwili obecnej szerzej przyjęte standardy zapisu pseudokodu. Większość autorów używa przyjętej ad hoc składni, często opierając się na składni istniejących języków programowania (Pascal, ALGOL, C), przy czym tego typu wzorowanie się na składni konkretnego języka jest **odradzane**.

1. Pseudokod „agnostyczny składniowo”

```
 1   input: tab         // tablica przechowujaca wartosci ciagu
 2           n          // dlugosc tablicy
 3  output: roznica_max // maksymalna roznica spelniajaca kryteria zadania
 4
 5  i := 0
 6  j := 0
 7  roznica_max := -1
 8
 9  if n > 1
10   while i < n-1
11    j := j + 1
12    while j < n
13       if (tab[j] > tab[i])
14         foo := tab[j] - tab[i]
15         if foo > roznica_max
16           roznica_max := foo
17         endif
18       endif
19       j := j+1
20    endwhile
21    i := i+1
22   endwhile
23  endif
24
25  return roznica_max
```

Zauważmy, że w pseudokodzie do zapisu obu pętli użyto pętli typu `while`.

Zapis algorytmu w pseudokodzie, jest etapem pośrednim pomiędzy analizą problemu i opracowaniem algorytmu, a samą implementacją w konkretnym języku programowania, która zostanie przedstawiona w kolejnych rozdziałach.

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.

Uwaga: 2.3

Ważne jest, aby w momencie wykonywania tej części zadania na chwilę "stać się kompilatorem" i wykonywać obliczenia, aktualizować wartości zmiennych tak jak to zostało opisane przy pomocy pseudokodu, a nie tak, jak autor „chciałby" żeby program działał.

Prezentacja poprawnego wykonania tej części zadania zostanie przedstawiona na przykładzie danych wejściowych

```
we = [10 4 2 9 5]
```

Obliczenia zgodnie z algorytmem przedstawia poniższa tabela (wartości zmiennych i poszczególnych wyników pośrednich z kolejnych iteracji są umieszczone w wierszach).

i	j	tab[i]	tab[j]	tab[j]>tab[i]	foo	foo>roznica_max	roznica_max
0	1	10	4	0	_	0	-1
0	2	10	2	0	_	0	-1
0	3	10	9	0	_	0	-1
0	4	10	5	0	_	0	-1
1	2	4	2	0	_	0	-1
1	3	4	9	1	5	0	5
1	4	4	5	1	1	0	5
2	3	2	9	1	7	0	7
2	4	2	5	1	3	0	7
3	4	9	5	0	3	0	7

Dopiero teraz możemy próbować zabierać się za programowanie (!), lecz zanim do tego przystąpimy warto jeszcze zastanowić się nad złożonością opracowanego algorytmu i przeprowadzić nieskomplikowaną analizę teoretyczną.

2.1.5. Teoretyczne oszacowanie złożoności obliczeniowej.

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

- dla `i=0` będzie to `n-1` porównań
- dla `i=1` będzie to `n-2` porównań
- dla `i=2` będzie to `n-3` porównań
- ...

A więc całkowita liczba porównań to suma $(n-1)+(n-2)+(n-3)+...+2+1$.
Przypominając sobie zależność na sumę kolejnych liczb naturalnych

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

możemy policzyć, że dla ciągu 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.

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

2.1.6. Implementacja algorytmu

Tutaj moglibyśmy przystąpić do implementacji algorytmu w wybranym przez siebie języku, jednak zanim do tego przystąpimy spróbujmy wymyśleć coś lepszego niż algorytm o złożoności $O(n^2)$. Implementacją zajmiemy się w momencie gdy będziemy mieć opracowane przynajmniej dwie wersje algorytmu.

2.2. Rozwiązanie - próba druga (nieco bardziej finezyjna)

2.2.1. Ponowne przemyślenie problemu i próba wymyślenia algorytmu wydajniejszego.

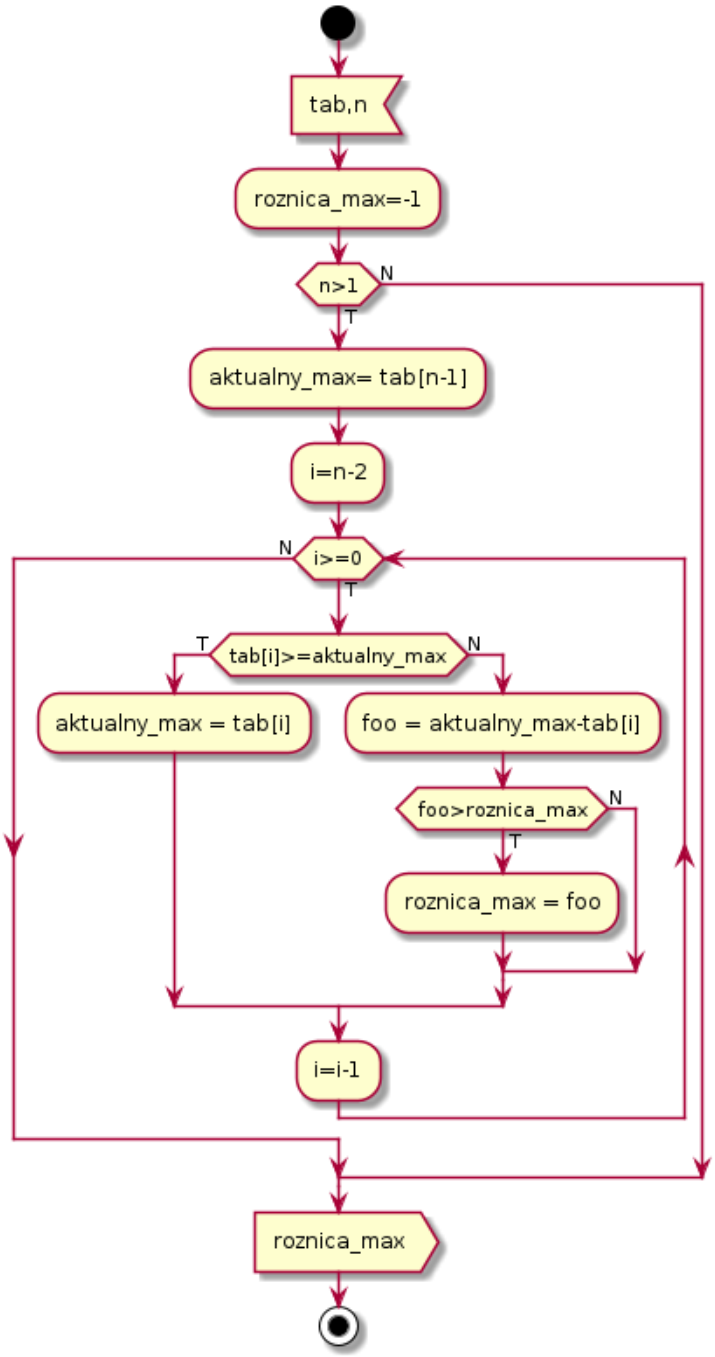
Pomyślmy czy da się rozwiązać problem w wydajniejszy sposób. Pomyślmy jeszcze trochę…

I jeszcze trochę…

Nie zrażaj się jeśli niczego nie wymyślisz w ciągu trzech minut… Nie spiesz się… Daj sobie jeszcze kilka godzin…

Dłuższa analiza problemu zwykle pozwola się przekonać, że poprawienie wydajności algorytmu jest możliwe. W tym wypadku pomysł opierać się będzie na przechodzeniu tablicy od_końca (od prawej strony) z jednoczesnym przechowywaniem wartości maksymalnego do tej pory odnalezionego elementu (nazwijmy go `aktualny_max`) oraz, tak jak poprzednio, maksymalnej roznicy przechowywanej w zmiennej `roznica_max`. Jeśli w trakcie przechodzenia natrafimy na element większy od `aktualny_max` to aktualizujemy wartość zmiennej `aktualny_max`. Jeśli element będzie mniejszy, sprawdzamy różnicę pomiędzy nim, a zmienną `aktualny_max`. Jeśli różnica ta będzie większa niż aktualna wartość maksymalnej różnicy to aktualizujemy wartość zmiennej `roznica_max`.

2.2.2. Schemat blokowy algorytmu



2.2.3. Algorytm zapisany w pseudokodzie (Powtórzenie punktów 2-5 dla algorytmu wydajnieszego).

```
1   input: tab         // tablica przechowujaca wartosci ciagu
2            n         // dlugosc tablicy
3  output: roznica_max // maksymalna roznica spelniajaca kryteria zadania
4
5  roznica_max := -1
6
7  if n > 1
8   aktualny_max := tab[n-1]
```

```
 9   i := n-2
10   while i >= 0
11       if (tab[i] > aktualny_max)
12         aktualny_max := tab[i]
13       else
14         foo := aktualny_max - tab[i]
15         if foo > roznica_max
16           roznica_max := foo
17         endif
18       endif
19     i := i-1
20   endwhile
21  endif
22
23  return roznica_max
```

2.2.4. „Ołówkowe" sprawdzenie poprawności algorytmu nr 2

Dla tego samego przypadku co poprzednio możemy sprawdzić przykładowe działanie algorytmu, w celu znalezienia w nim luk i niespójności.

Dla ciągu wejściowego
`we = [10 4 2 9 5]`
obliczenia będą się przedstawiały następująco

i	aktualny_max	tab[i]	tab[i]>aktualny_max	aktualny_max	foo	foo>roznica_max	roznica_max
3	5	9	0	9	_	0	-1
2	9	2	0	9	7	0	7
1	9	4	0	9	5	0	7
0	9	10	0	10	5	0	7

2.2.5. Teoretyczne oszacowanie złożoności obliczeniowej dla algorytmu 2.

Podobnie jak poprzednio operacją wyznaczającą złożoność algorytmu będzie operacja porównania, która musi być wykonana dla wszystkich elementów tablicy. Jak można zauważyć, tym jednak razem rozwiązanie problemu będzie wymagało jednokrotnego przejścia przez tablicę (jedna pętla `while` w pseudokodzie), a konkretnie dla ciągu o n elementach pętla `while` wykona się $n-2$ razy.

W takim wypadku mówimy, że algorytm jest o złożoności czasowej $O(n)$, czyli zależy liniowo od liczby danych wejściowych.

Uwaga: 2.4

W wyjątkowych sytuacjach (w przypadku wyjątkowo prostych problemów lub wyjątkowo genialnych programistów) może sie zdażyć, że już pierwsza wersja algorytmu będzie algorytmem optymalnym. Z reguły jednak początkowa metoda rozwiązania problemu będzie nieoptymalna i nieco bardziej dogłębna analiza pozwoli na poprawienie wydajności algorytmu. (Należy tutaj również podkreślić, że w praktyce, obiżenie złożoności do liniowej, może być dość trudne, lub w przypadkach wybranych problemów, wręcz niemożliwe).
W tej części zadania chodzi właśnie o przedstawienie i udowodnienie przez Autora trudu i czasu jaki poświęcił na dogłebne rozwiązanie swojego problemu. Może to być wykazane właśnie przez pełną dokumentację historii rozwiązywania projektu, która może/powinna zawierać przynajmniej dwie wersje algorytmu, a także ich porównanie poprzez analizę złożoności czasowej czy też pamięciowej, itp.
W przypadku problemów z teoretyczną analizą ww. zagadnień, można również w ostateczności posłużyć się właściwie przeprowadzonymi eksperymentami obliczeniowymi co zostanie przedstawione w rozdziale 2.3.

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

2.3.1. Prosta implementacja

Rozważmy na początku najprostsze podejście, w którym cały program umieszczamy w jednym pliku. Aby zachować modularność oba algorytmy zapiszemy w postaci oddzielnych funkcji. Poprawność kodu można przetestować wywołując obie funkcje dla tych samych danych.

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

```
#include <stdio.h>

int MaxRoznica(int tab[], int n)
{
  int roznica_max = -1 ;
  int foo ;

  if (n > 1) {
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        if (tab[j] > tab[i]) {
          foo = tab[j] - tab[i] ;
          if (foo > roznica_max)
             roznica_max = foo ;
        }

      }
    }
  }
  return roznica_max;
}

int MaxRoznicaWersja2(int tab[], int n)
{
  int roznica_max = -1 ;
  int foo ;
```

```
  if (n > 1) {
    int aktualny_max = tab[n-1] ;
    for (int i = n-2; i >= 0 ; i--) {
      if ( tab[i] > aktualny_max )
        aktualny_max = tab[i] ;
      else
        {
        foo = aktualny_max - tab[i] ;
          if (foo > roznica_max)
             roznica_max = foo ;
        }

    }
  }
  return roznica_max;
}

int main()
{
  int tab[] = {10, 4, 2, 9, 5} ;
  // wyznacz liczbe elementow tablicy
  int n = sizeof(tab) / sizeof(tab[0]);

  int wynik ;
  wynik = MaxRoznica(tab, n);

  if (wynik != -1) {
    printf("Maksymalna roznica uzyskana algorytmem nr 1 to %d.\n", wynik);
  }

  wynik = MaxRoznicaWersja2(tab, n);
  if (wynik != -1) {
    printf("Maksymalna roznica uzyskana algorytmem nr 2 to %d.\n", wynik);
  }


  return 0;
}
```

```
Maksymalna roznica uzyskana algorytmem nr 1 to 7.
Maksymalna roznica uzyskana algorytmem nr 2 to 7.
```

2.3.2. Testy „niewygodnych" zestawów danych

Kod powyżej przedstawia proste wywołanie dwóch funkcji w celu sprawdzenia ich działania dla niewielkiego, „ręcznie zdefiniowanego" zestawu danych testowych.

Ten program moglibyśmy uzupełnić o kilka dodatkowych testów, sprawdzających działanie algorytmów dla jakiś specyficznych (z punktu widzenia zadanego algorytmu) zestawów danych, aby przekonać się, że jest on odporny na „niewygodne" zestawy danych.

Przykładem tego typu danych w przypadku omawianego problemu mogą być wspomniane już wcześniej ciągi wartości nierosnących.

```
int main()
{
  int wynik ;

  int tab2[] = {3,2,1} ;
  int tab3[] = {7,7,7,7,7} ;

  // wyznacz liczbe elementow tablic
  int n2 = sizeof(tab2) / sizeof(tab2[0]);
  int n3 = sizeof(tab3) / sizeof(tab3[0]);

  wynik = MaxRoznica(tab2, n2);
  printf("Test1A: %d.\n", wynik);

  wynik = MaxRoznicaWersja2(tab2, n2);
  printf("Test1B: %d.\n", wynik);

  wynik = MaxRoznica(tab3, n3);
  printf("Test1A: %d.\n", wynik);

  wynik = MaxRoznicaWersja2(tab3, n3);
  printf("Test1B: %d.\n", wynik);

  return 0;
}
```

```
#include <stdio.h>

int MaxRoznica(int tab[], int n)
{
  int roznica_max = -1 ;
  int foo ;

  if (n > 1) {
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        if (tab[j] > tab[i]) {
          foo = tab[j] - tab[i] ;
          if (foo > roznica_max)
             roznica_max = foo ;
        }

      }
    }
  }
  return roznica_max;
}
```

```
int MaxRoznicaWersja2(int tab[], int n)
{
  int roznica_max = -1 ;
  int foo ;

  if (n > 1) {
    int aktualny_max = tab[n-1] ;
    for (int i = n-2; i >= 0 ; i--) {
      if ( tab[i] > aktualny_max )
        aktualny_max = tab[i] ;
      else
        {
        foo = aktualny_max - tab[i] ;
          if (foo > roznica_max)
             roznica_max = foo ;
        }

    }
  }
  return roznica_max;
}

int main()
{
  int wynik ;

  int tab2[] = {3,2,1} ;
  int tab3[] = {7,7,7,7,7} ;

  // wyznacz liczbe elementow tablic
  int n2 = sizeof(tab2) / sizeof(tab2[0]);
  int n3 = sizeof(tab3) / sizeof(tab3[0]);

  wynik = MaxRoznica(tab2, n2);
  printf("Test1A: %d.\n", wynik);

  wynik = MaxRoznicaWersja2(tab2, n2);
  printf("Test1B: %d.\n", wynik);

  wynik = MaxRoznica(tab3, n3);
  printf("Test1A: %d.\n", wynik);

  wynik = MaxRoznicaWersja2(tab3, n3);
  printf("Test1B: %d.\n", wynik);

  return 0;
}
```

```
Test1A: -1.
```

```
Test1B: -1.
Test1A: -1.
Test1B: 0.
```

Przeprowadzenie tego rodzaju testów często pozwala na znalezienie luk w opracowanych algorytmach i każe nam wrócić się do fazy wcześniejszej.

W naszym przypadku wyniki testów wskazują, że implementacja algorytmu nr 2 nie działa tak, jak byśmy się tego mogli spodziewać i aby poprawić działanie dla ciągów tych samych wartości (a więc poprawić jego ogólną niezawodność) należy wprowadzić pewne poprawki (Zastanów się jakie).

Uwaga: 2.5

Testy tego rodzaju zawieramy w sprawozdaniu z projektu!

2.3.3. Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej

O wiele ciekawsze będzie przeprowadzenie nieco bardziej zaawansowanych testów, które pozwolą potwierdzić eksperymentalnie większą wydajność algortymu nr 2 nad wersją naiwną.

Będzie to możliwe jeśli porównamy czasy działania obu algorytmów dla tych samych zestawów danych wejściowych. W ten sposób dla rosnącej liczby danych wejściowych n zgromadzimy czasy obliczeń $t(n)$. Te wyniki następnie można przedstawić w formie tabeli lub wykresu.

Uwaga! Należy przy tym pamiętać, że w celu zauważenia istotnych różnic w czasach działania algorytmów zestawy danych muszą być **naprawdę** duże.

W tym celu potrzebne będzię stworzenie odpowiedniego kodu pomocnicznego, którego zadaniem będzie:

- generowanie zestawów danych testowych
- zapamiętanie wyników testów
- wyświetlenie wyników testów
- itp.

Najwygodniej będzie zapisać poszczególne zadania w postaci oddzielnych funkcji.

W najprostszym podejściu, w przypadku omawianego zadania jako oddzielne funkcje możemy zrealizować

1. Generowanie losowych danych testowych

```
void Generuj(int *tab, int n, int nmax)
{
  for (int i=0; i<n; i++)
    tab[i] = rand()%nmax ;
}
```

2. Wyświetlanie tablicy z danymi wejściowymi

   Jest to funkcja pomocnicza, której poniżej nie używamy, ale może ona być przydatna w fazie implementacji w celu sprawdzenia, czy funkcja generująca dane testowe działa poprawnie.

```
void Wypisz(int *tab, int n)
{
  for (int i=0; i<n; i++)
    printf("%d ",  tab[i]) ;
```

```
  printf("\n") ;
}
```

W celu niekomplikowania tego dokumentu ponad miarę na razie pozostaniemy na tych dwóch funkcjach, a część kodu zawierającą testy umieścimy bezpośrednio w funkcji `main`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int MaxRoznica(int tab[], int n)
{
  int roznica_max = -1 ;
  int foo ;

  if (n > 1) {
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        if (tab[j] > tab[i]) {
          foo = tab[j] - tab[i] ;
          if (foo > roznica_max)
             roznica_max = foo ;
        }

      }
    }
  }
  return roznica_max;
}

int MaxRoznicaWersja2(int tab[], int n)
{
  int roznica_max = -1 ;
  int foo ;

  if (n > 1) {
    int aktualny_max = tab[n-1] ;
    for (int i = n-2; i >= 0 ; i--) {
      if ( tab[i] > aktualny_max )
        aktualny_max = tab[i] ;
      else
        {
        foo = aktualny_max - tab[i] ;
          if (foo > roznica_max)
             roznica_max = foo ;
        }

    }
  }
```

```
  return roznica_max;
}

void Generuj(int *tab, int n, int nmax)
{
  for (int i=0; i<n; i++)
    tab[i] = rand()%nmax ;
}

void Wypisz(int *tab, int n)
{
  for (int i=0; i<n; i++)
    printf("%d ",  tab[i]) ;
  printf("\n") ;
}

int *tab ;
float *czasy1 ;
float *czasy2 ;

clock_t begin ;
clock_t end ;

int main(){

  //int N[] = {10, 50, 100, 200, 500, 1000, 5000, 10000} ;
  int N[] = {2500, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000} ;
  int liczba_testow = sizeof(N) / sizeof(N[0]) ;
  czasy1 = (float *) malloc(liczba_testow * sizeof (float) );
  czasy2 = (float *) malloc(liczba_testow * sizeof (float) );

  for(int i=0; i<liczba_testow; i++){
    tab = (int *) malloc(N[i] * sizeof (int) );
    Generuj(tab, N[i], 1000) ;

    begin = clock();
    MaxRoznica(tab, N[i]);
    end = clock();
    czasy1[i] = (double)(end - begin) / CLOCKS_PER_SEC;

    begin = clock();
    MaxRoznicaWersja2(tab, N[i]);
    end = clock();
    czasy2[i] = (double)(end - begin) / CLOCKS_PER_SEC;
    free(tab) ;
  }
  printf("\n") ;
  printf(" L.p.      n    t1 [s]   t2 [s] \n ") ;
  for(int i=0; i<liczba_testow; i++){
    printf("%2d %10d %6.6f %6.6f\n ", i+1, N[i], czasy1[i], czasy2[i]) ;
```

```
  }

  return  0;
}
```

L.p.	n	t1 [s]	t2 [s]
1	2500	0.018269	0.000007
2	5000	0.071893	0.000015
3	10000	0.283508	0.000029
4	20000	1.147783	0.000058
5	30000	2.585719	0.000089
6	40000	4.579347	0.000118
7	50000	7.130314	0.000146
8	60000	10.336639	0.000174
9	70000	14.007204	0.000204
10	80000	18.251205	0.000232

Jak widać z tabeli wygenerowanej przy pomocy powyższego kodu dla tych samych zestawów danych wejściowych obliczenia wykonywane przy pomocy algorytmu w wersji naiwnej trwają o wiele dłużej niż w przypadku wersji ulepszonej. Dla ciągu liczącego 80000 elementów różnica w czasie działania obu algorytmów jest rzędu 10^4. Przekłada się na to, że obliczenia w jednym przypadku zajmują około 20s, podczas gdy w drugim nie trwają nawet milisekundy!

Wyniki obliczeń można również przedstawić w postaci graficznej. Na podstawie kształtu krzywych jakie tworzą wykresy punktów funkcji $t(n)$ można się przekonać, że obliczona teoretycznie złożoność obliczeniowa ma bezpośrednie odzwierciedlenie w czasach obliczeń wykonanych przy pomocy obu algorytmów.

Na rysunkach poniżej za pomocą czerwonych kółek oznaczono punkty których odcięte to liczba danych wejściowych równa odpowiednio 2500, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, a rzędne to czasy obliczeń algorytemem w wersji pierwszej i drugiej (kolumny 2, 3 oraz 4 tabeli powyżej).

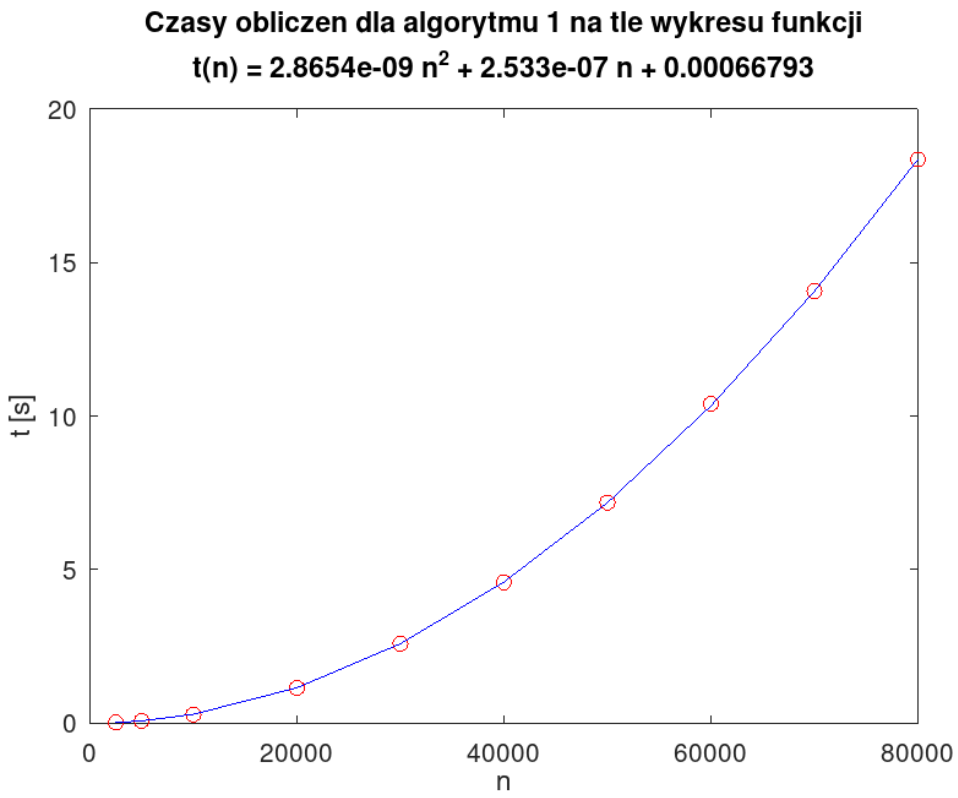
Bazując na tych punktach dokonano aproksymacji funkcji (linia ciągła niebieska). Jak można zauważyć, czasy obliczeń w pierwszym przypadku „układają się" na pewnej paraboli, a w drugim leżą wzdłuż linii prostej! (Współczynniki obu funkcji przedstawione w tytułach wykresów zapisane są w notacji naukowej, a więc zapis $2.8654e-09$ oznacza liczbę $2.8654 \cdot 10^{-9}$).

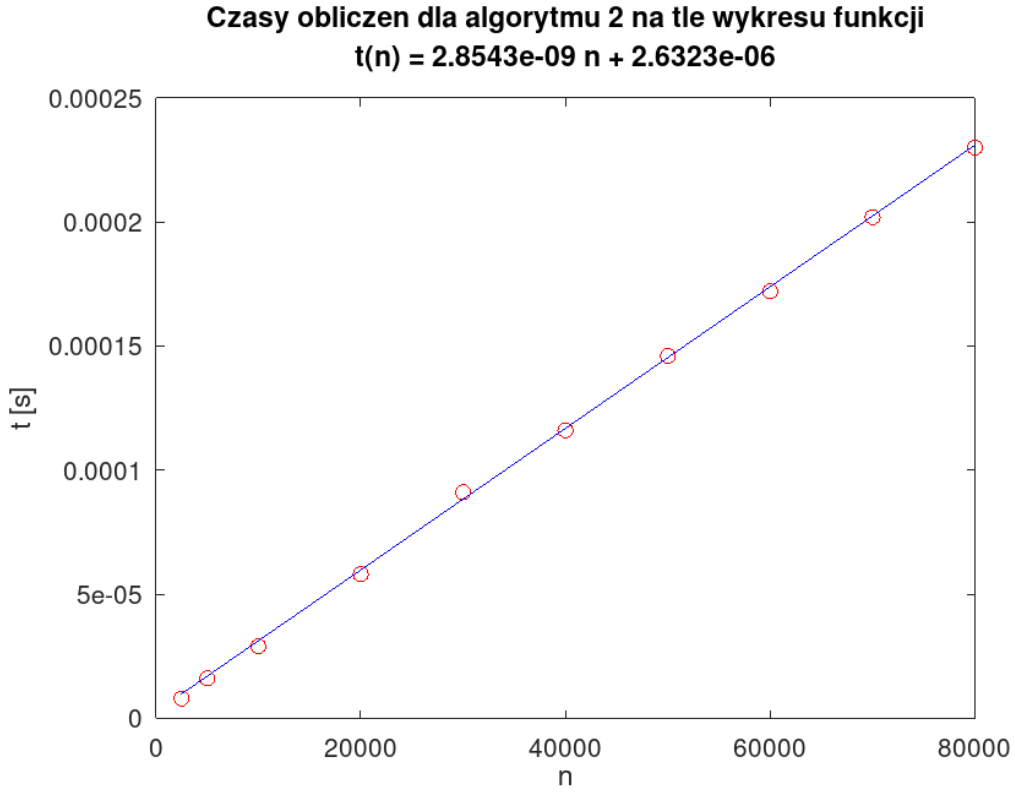
Uwaga: 2.6

Przedstawione wyniki zostały otrzymane na „nienajnowszym" laptopie autora i będą one oczywiście inne jeśli program zostanie wykonany na innym sprzęcie. Jednak w każdym przypadku charakter uzyskanych krzywych (parabola i prosta) powinny zostać zachowane.

Zadanie: 2.1

Na podstawie funkcji przedstawionych na obu wykresach oszacuj ile dni (sic!) będą trwały obliczenia algorytmem 1 jeśli tablica wejściowa będzie miała 10 milionów elementów. Ile sekund (sic!) będzie potrzebował na rozwiązanie tego samego zadania algorytm 2?





Wyniki jednego przykładowego testu zostały przedstawione powyżej. Można zauważyć wielokrotną różnicę w czasie obliczeń

Zadanie: 2.2

Uwaga: Przedstawiony w rozdziale 2.3.3 kod nie prezentuje całkowicie eleganckiego podejścia.
W celu jego poprawienia zaleca się jego reorganizację.
Proszę przepisać zaprezentowany program, tak aby, testy, które w rozdziałach 2.3.1, 2.3.2 oraz 2.3.3 były umieszczone bezpośrenio wewnątrz funkcji `main` wydzielić od osobnych funkcji. Końcowym rezultatem powinien być program, w którym ciało funkcji `main` będzie wyglądało np. tak:

```
int main(){

  test1() ;
  test2() ;
  test3() ;

  return  0;
}
```

2.3.4. Testy wydajności algorytmów - złożoności optymistyczne/pesymistyczne

W przypadku niektórych problemów może się okazać, że istnieją pewne specyficzne zestawy danych wejściowych, które będą skutkowały szybszym (lub wolniejszym) zakończeniem działania algorytmu w porównaniu do przeciętnie oczekiwanego czasu (częsty przypadek w przypadku algorytmów sortujących). Mówimy wtedy o złożoności optymistycznej (pesymistycznej).

Zastanów się czy Twój problem jest tego rodzaju problemem. Jeśli tak - skonstruuj funkcję generującą odpowiednie zestawy danych wejściowych, a następnie wykonaj serie testów (podobnie jak powyżej). Wyniki znowu przedstaw na wykresach i porównaj do siebie.