Algorytmy i Struktury Danych
Zadanie Projektowe

Norbert Tabisz
Inżynieria i Analiza Danych, grupa: 9
nr. indeks (184309)
Politechnika Rzeszowska

28 stycznia 2026

Spis treści

1 Problematyka

1.1 Treść zadania

Dla zadanej tablicy liczb całkowitych znajdź te pary, których różnica jest równa zadanej liczbie k.

Wejscie: tablica liczb całkowitych A[] , liczba całkowita k
Wyjście: wszystkie nie powtarzające się pary liczb (a,b) takie, że |a - b| = k
Przykład:
Wejście:
A[] = [1,5,2,2,2,5,5,4]
k = 3
Wyjścię: [2,5] oraz [1,4]

1.2 Analiza problemu

Danym wejściem jest tablica liczb całkowitych i liczba całkowita k. Celem jest znalezienie wszystkich par elementów tablicy, których różnica wynosi k.

Najprostszym podejściem do rozwiązania problemu jest sprawdzenie wszystkich możliwych par elementów tablicy poszczególno ze sobą i obliczenie różnicy dla każdej z nich.

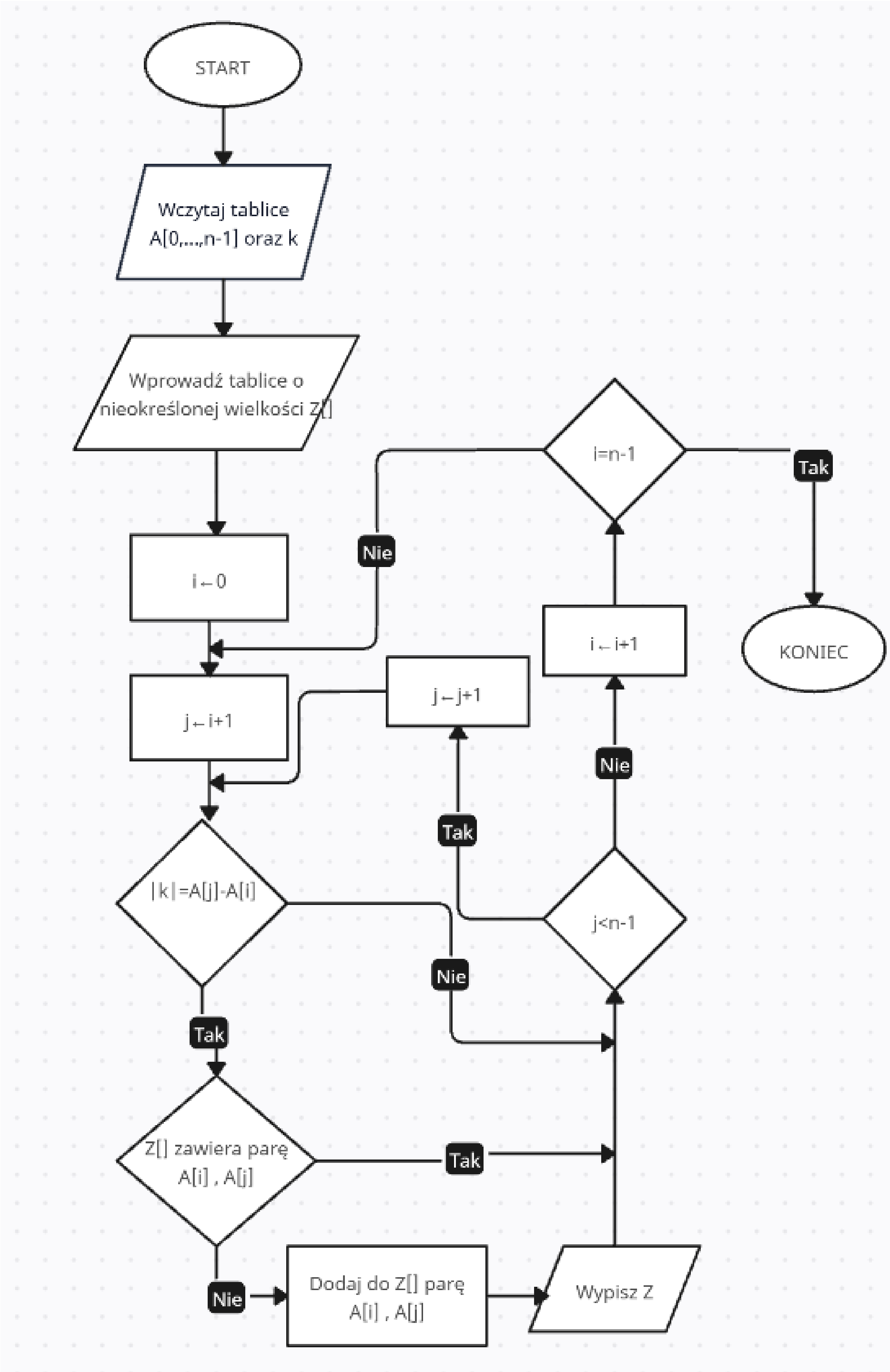
Analizując ten problem można zauważyć że kolejnosc elementow w parze ma znaczenie, ponieważ a - b != b - a. Znaczy to tyle że trzeba każdą pare rozpatrywac osobno. Mimo najprostszego działania metoda brute-force nie zawsze jest tą najoptymalniejszą dlatego warto rozważyć rozwiązanie tego problemu inną metodą przedstawioną w dalszym etapie.

2 Rozwiązanie problemu metodą brute-force

2.1 Schemat blokowy

W poniższym schemacie blokowym oraz pseudokodzie przyjęto założenie, że tablice indeksowane są od 0.

Algorytm zapisany w postaci schematu blokowego przedstawia się następująco:



Rysunek 1: Schemat blokowy algorytmu brute-force

Algorytm wykorzystuje trzy zmienne pomocnicze wymagające wstępnej inicjalizacji. Liczniki i oraz j służą jako indeksy do nawigacji po strukturze tablicy, natomiast tablica Z[] przechowuje pasujące wartosci o podanej różnicy k.

2.2 Pseudokod

Tworze pseudokod programu.

```
1. Array A, k                    // A - tablica z wartosciami, k-dana roznica
2. n                             // wielkosc tablicy
3. Z                             // pusta tablica par

4. for i <- 0 to  n 1
5.     for j <- i+1 to  n 1
6.         if |A[j] - A[i]| = |k|
               if A[j] - A[i] = k
                   x<-A[i]
                   y<-A[j]
               else
                   x<-A[j]
                   y<-A[i]
7.             end if
9.             if pair (x, y) is not in Z
10.                add (x, y) to Z
11.            end if
12.        end if
13.    end for
14. end for

15. if Z empty
16.     print "Brak par o roznicy k"
17. else
18.     print Z
19. end if
```

Listing 1: Pseudokod metodą brute-force

2.3 "Ołówkowe"sprawdzenie

Poniżej jest przedstawione "ołówkowe"sprawdzenie aby wychwycić potencjalne błędy lub niezgodności algorytmu.Dla przykładowych wartości w tablicy A oraz różnicy k między wartościami.

A=[1,5,2,4,3] oraz k=3

k	i	j	A[i]	A[j]	A[j]-A[i] =k
3	0	1	1	5	0
3	0	2	1	2	0
3	0	3	1	4	1
3	0	4	1	3	0
3	1	2	5	2	0
3	1	3	5	4	0
3	1	4	5	3	0
3	2	3	2	4	0
3	2	4	2	3	0
3	3	4	4	3	0

Jedynym prawidłowem wynikiem w tym przypadku jest $[1, 4]$.

2.4 Złożoność obliczeniowa algorytmu brute-froce

Aby obliczyć złożoność obliczeniową tego algorytmu, musimy przeanalizować liczbę operacji dominujących(tych wykonujących się wewnątrz pętli). Algorytm opiera się na dwóch zagnieżdżonych pętlach, które porównują pary elementów tablicy o rozmiarze n.Algorytm porównuje i-tą i j-tą, wartości tabeli. Liczba i-ta zaczyna się od i=0 do i=n-1 , więc wykonuje się n-1 razy.Liczba j-ta natomiast zaczyna się od j=i+1 do j=n-1.Liczba iteracji pętli wewnętrznej zmniejsza razem ze wzrostem i.

- dla $i = 0$,j idzie od 1 do $n - 1$ ($n - 1$ porównań).
- dla $i = 1$, j idzie od 2 do $n - 1$ ($n - 2$ porównań).
- dla $i = 2$, ($n - 3$ porównań)
- ...

Łączną liczbę porównań ($k = A[j] - A[i]$) określa suma ciągu arytmetycznego:

$$(n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2}$$

Możemy go przekształcić w taki sposób aby dostać ilość porównań:

$$\frac{1}{2}n^2 - \frac{1}{2}n$$

Dominującym skłądnikiem jest n^2. W notacji dużego O pomijamy stałe oraz składniki niższych rzędów co daje nam złożoność obliczeniową $O(n^2)$

2.5 Implementacja metodą brute-force w C++

Na podstawie powyższego schematu blokowego i pseudokodu napisałem kod programu w języku C++ Podany kod podzieliłem na funkcje dla łatwiejszego wyjaśnienie działania programu.

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <utility>
using namespace std;

void wyswietlTablice(int tab[], int rozmiar) {
    cout << "Przykladowa␣tablica␣A:␣[";
    for (int i = 0; i < rozmiar; i++) {
        cout << tab[i];
        if (i < rozmiar - 1) cout << ",␣";
    }
    cout << "]" << endl;
}
```

Listing 2: Funckcja odpowiedzialna za wyswietlanie tablicy

Zaimplementowałem algorytm sprawdzajacy czy 2 pary zawieraja te same elementy nie zaleznie od kolejnosci.

```cpp
bool czyTeSameElementy(const pair<int,int> para1, const pair<int,int> para2) {
    return (para1.first == para2.first && para1.second == para2.second) || (para1.first == para2.second && para1.second == para2.first);
}
```

Listing 3: Funckja logiczna czyTeSameElementy sprawdzającą czy para sie powtarza

Poniższa funkcja sprawdza czy kolejne wartosci z tablicy sa rowne podanej roznicy k. Ponadto dba o poprawne i zrozumiałe wyswietlenie danej pary w zaleznosci od podanego znaku roznicy. Funkcja znajdzPary jest też odpowiedzialna za sprawdzanie czy dana para już wcześniej nie wystapila aby uniknąć powtórzeń. W tym fragmencie kodu użyto wbudowany typ danych, wektor który jest rodzajem tablicy dynamicznej.

```
   vector<pair<int,int>> znajdzPary(int A[], int n, int k) {
    vector<pair<int,int>> Z;

    for (int i = 0; i < n-1; i++) {
        for (int j = i + 1; j < n; j++) {

            if (abs(A[j] - A[i]) == abs(k)) {
                pair<int,int> potencjalnaPara;

                if (A[j] - A[i] == k) {
                    potencjalnaPara.first = A[i];
                    potencjalnaPara.second = A[j];
                } else {
                    potencjalnaPara.first = A[j];
                    potencjalnaPara.second = A[i];
                }

                bool juz_jest = false;
                for (int m = 0; m < Z.size(); m++) {
                    if (czyTeSameElementy(potencjalnaPara, Z[m])) {
                        juz_jest = true;
                        break;
                    }
                }

                if (!juz_jest) {
                    Z.push_back(potencjalnaPara);
                }
            }
        }
    }
    return Z;
}
```

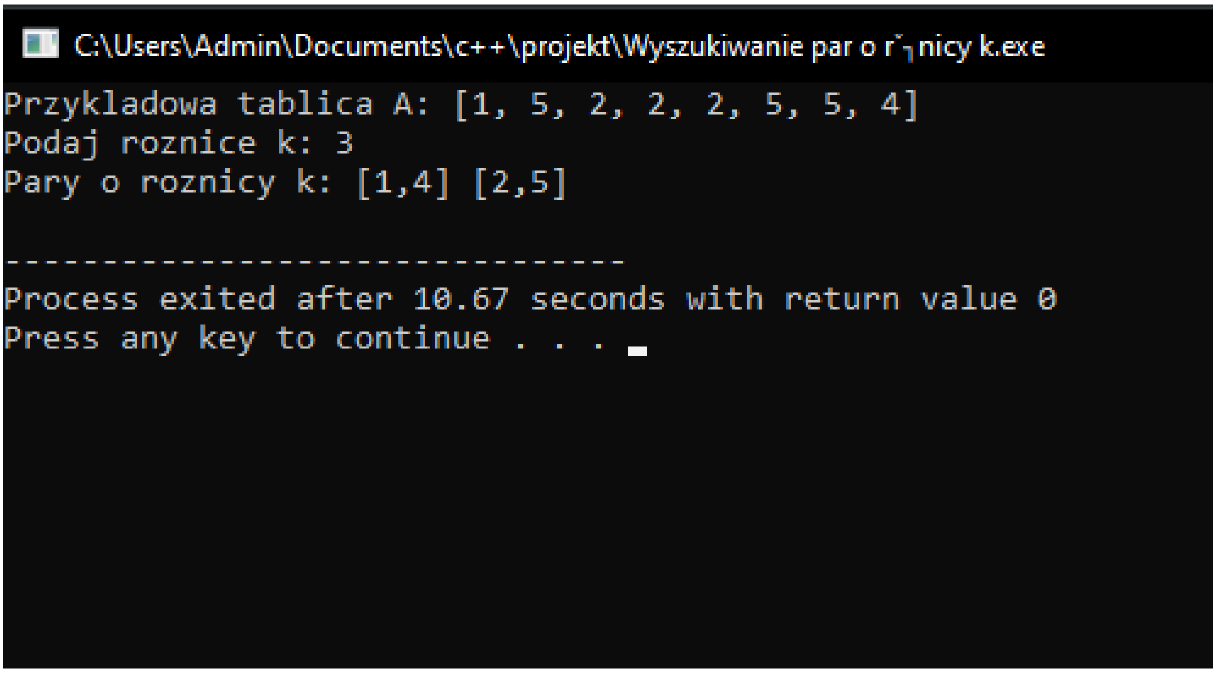
Listing 4: Funkcja znajdujaca pary o roznicy k

W funkcji main pozostawiono Wczytywanie liczby k z klawiatury, sprawdzenie czy tablica odpowiedzialna za przechowywanie poprawnych par nie jest pusta oraz wyswietlanie wyniku.

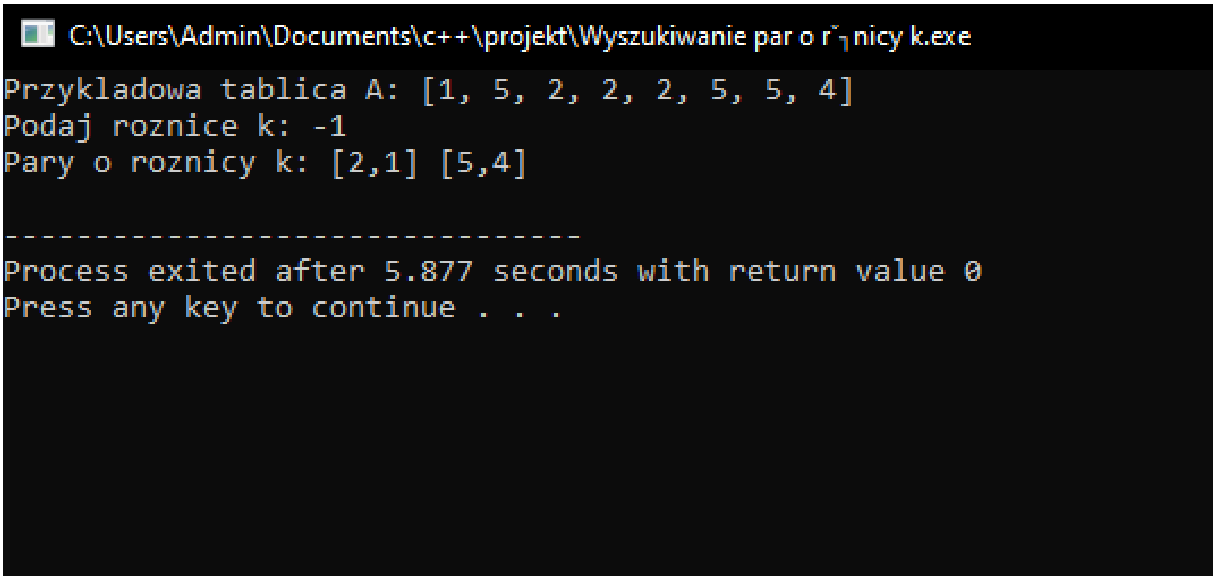
```
int main() {
    int A[] = {1, 5, 2, 2, 2, 5, 5, 4};
    int n = sizeof(A) / sizeof(A[0]);
    int k;

    wyswietlTablice(A, n);
    cout << "Podaj roznice k: ";
    cin >> k;

    vector<pair<int,int>> Z = znajdzPary(A, n, k);

    if (Z.empty()) {
        cout << "Brak par o roznicy k." << endl;
    } else {
        cout << "Pary o roznicy k: ";
        for (int i = 0; i < Z.size(); i++) {
            cout << "[" << Z[i].first << "," << Z[i].second << "]";
        }
        cout << endl;
    }

    return 0;
}
```

Listing 5: Funckja main

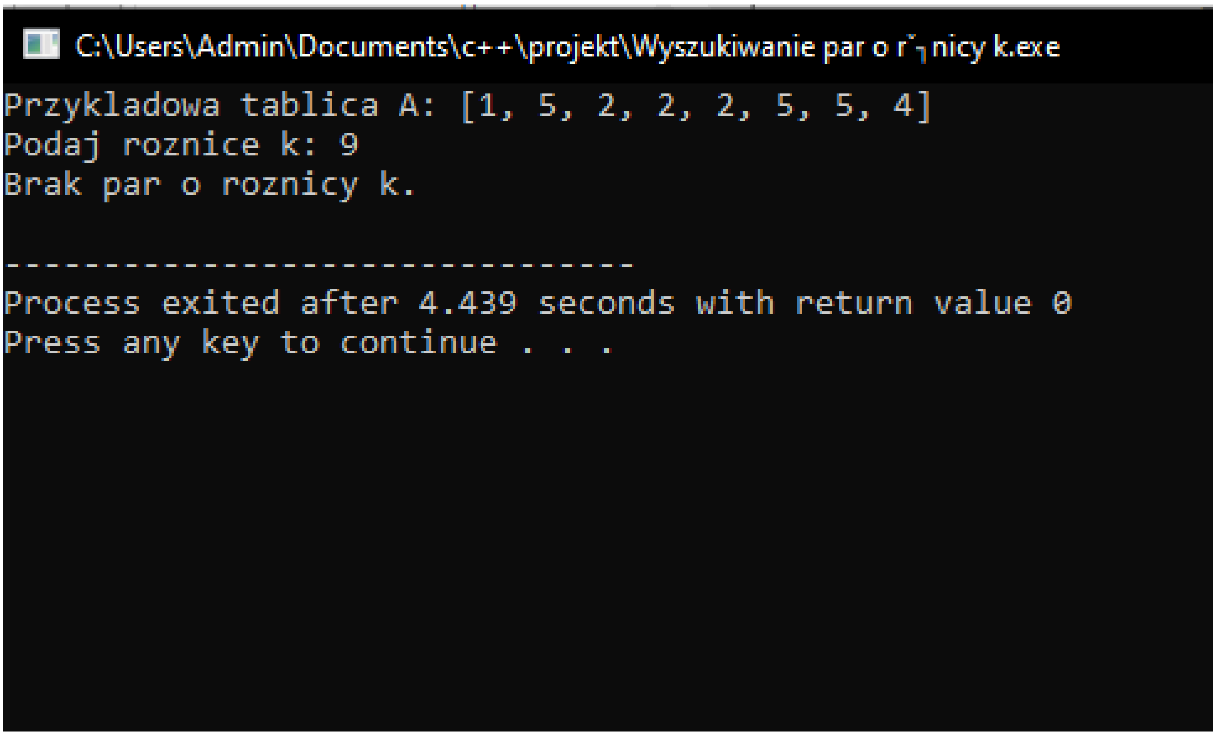
2.6 Wynik działania programu



Rysunek 2: Przykładowe działanie dla dodatniej różnicy k



Rysunek 3: Przykładowe działanie dla ujemnej różnicy k



Rysunek 4: Przykładowe działanie dla braku par o roznicy k

Ważnym elementem jest to że różnice k można przedstawic jako wartość bezwzgledna lecz pozniej dla trafniejszego wyswietlania wprowadzono warunek sprawdzajacy znak roznicy liczb A[i] oraz A[j].

3 Rozwiązanie metodą sortowania

Po głębszym zastanowieniu się nad przedstawionym problemem można znaleźć inne, efektywniejsze oraz mniej obciążające metody rozwiązania go. Jedną z takich metod jest metoda z sortowaniem tablicy która różni sie od wyżej przedstawionej metody brute-force oraz znacząco zmniejsza złożoność obliczeniową programu.

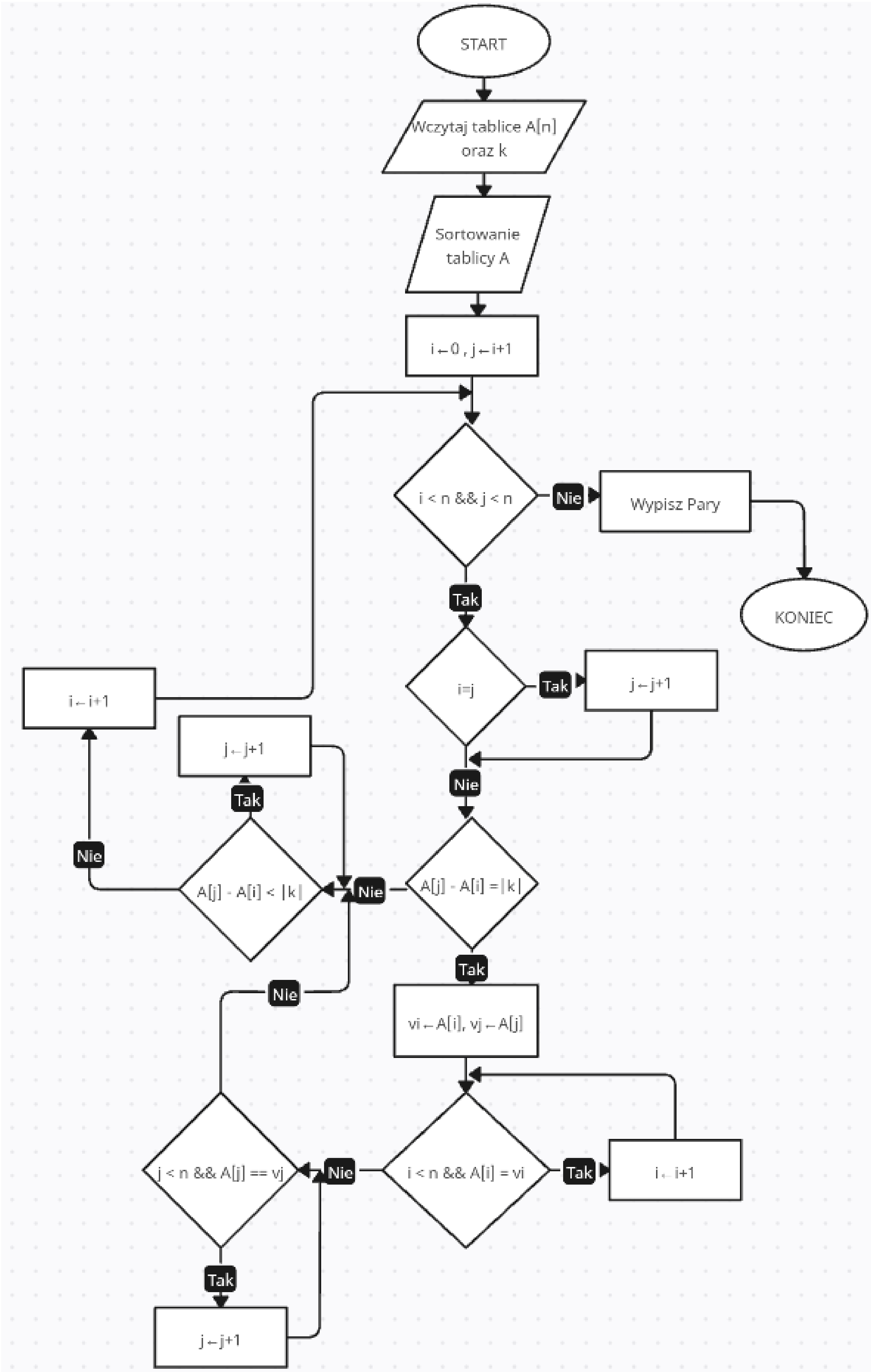
Ideą tej metody jest to że znając różnice k i pierwszą liczbe x=A[i], możemy wyznaczyć drugą liczbę y=A[j].Tablice sortujemy rosnąco za pomoca gotowej funkcji. Dzięki temu nie jest konieczne sprawdzanie wielu pozycji dwukrotnie.Zarówno wartosci i sprawdzamy n razy a także j, n razy co zamiast n^2 daje nam n.

$$y - x = k \rightarrow y = k + x \tag{1}$$

Ważnym elementem tej metody jest to że, sortujemy podana tablice i z powtorzonych wartosci bierzemy tylko tę pierwszą. Sortowanie jest kluczowe, bo umożliwia użycie dwóch wskaźników. Pętla działa, dopóki oba wskaźniki i oraz j są w granicach tablicy.Ponieważ tablica zostaje posortowana to zawsze roznica j-i bedzie wieksza od 0. Kod w swojej usprawnionej wersji zarówno jak i poprzedni pomija wypisywanie duplikatów par.

3.1 Schemat blokowy

Na poniższym schemacie blokowym przedstawiono działanie algorytmu dla metody z sortowaniem:



Rysunek 5: Schemat blokowy drugiej metody

3.2 Pseudokod programu zapisanego metodą z sortowaniem

```
A[]                                // tablica A z podanymi liczbami
k                                  // roznica miedzy dwoma wartosciami
n                                  // wielkosc tablicy A
print "Tablica␣przed␣sortowaniem:"
Print A

sort array A

print Sorted A

i <- 0
j <- i + 1

while i < n and j < n
    if i = j
        j <- j + 1
    end if

    if A[j] - A[i] = |k|
        if k > 0
            print (A[i], A[j])
        else
            print (A[j], A[i])
        end if

        vi <- A[i]
        vj <- A[j]

        while i < n and A[i] = vi
            i <- i + 1
        end while
        while j < n and A[j] = vj
            j <- j + 1
        end while

    else if A[j] - A[i] < |k|
        j <- j + 1
    else
        i <- i + 1
    end if
end while
```

Listing 6: Pseudokod drugiej metody

3.3 Złożoność obliczeniowa drugiej metody

Rozważmy sortowanie tablicy zawierającej n elementów.

Złożoność obliczeniową programu wyznaczono poprzez analizę liczby operacji wykonywanych w zależności od rozmiaru danych wejściowych n, gdzie n oznacza liczbę elementów tablicy. Dla notacji O uwzględniane są jedynie operacje dominujące, bo decydują o czasie działania algorytmu dla dużych wartości n.

Wyświetlanie tablicy i wypisywanie znalezionych par ma złożoność $O(n)$. Dominujacym elementem jest sortowanie tablicy przy użyciu funkcji sort() posiadajacej złożoność $O(nlogn)$. (Wytłumaczenie tej funkcji zawarte jest pod fragmentem kodu "Funkcja main"). Po posortowaniu danych wykonywany jest algorytm wyszukiwania par liczb o zadanej różnicy, oparty na technice dwóch wskaźników. Każdy wskaźnik przesuwa się wyłącznie w jednym kierunku, dzięki czemu każdy element tablicy jest odwiedzany co najwyżej jeden raz co daje złożoność $O(n)$.

Ostatecznie otrzymujemy złożoność obliczeniową $O(nlogn)$.

3.4 Kod programu metodą sortowania

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>
using namespace std;

void wyswietlTablice(const vector<int>& A) {
    for (int x : A) {
        cout << x << " ";
    }
}
```

Listing 7: Funkcja wyswietlTablice

Funkcja wyswietlTablice jest odpowiedzialna za wyświetlanie tablicy A.

```cpp
   vector<pair<int,int>> znajdzPary(const vector<int>& A, int k) {
    int n = A.size();
    int i = 0, j = i + 1;
  vector<pair<int,int>> pLiczb;
    while (i < n && j < n) {
        if (i == j) {
            j++;
        }

        if (A[j] - A[i] == abs(k)) {

            if (k > 0) {
              pair<int,int> para(A[i],A[j]);
                pLiczb.push_back(para);
              }else {
                pair<int,int> para(A[j],A[i]);
                pLiczb.push_back(para);
              }
```

```
            int vi = A[i], vj = A[j];
            while (i < n && A[i] == vi) i++;
            while (j < n && A[j] == vj) j++;

        }
        else if (A[j] - A[i] < abs(k)) {
            j++;
        }
        else {
            i++;
        }
    }
    return pLiczb;
}
```

Listing 8: Funkcja znajdzPary

Funkcja znajdzPary przyjmuje jako argument vector liczb spośrod których ma znaleźć pary o różnicy k.

```
    int main() {
    vector<int> A = {1, 5, 2, 2, 2, 5, 5, 4};
    int k;
    cout << "Tablica przed sortowaniem:\n";
    cout << "[ ";
    wyswietlTablice(A);
    cout << "]";

    sort(A.begin(), A.end());

    cout << "\nPodaj k: ";
    cin >> k;
    cout << endl;

    cout << "Tablica po posortowaniu:\n";
    cout << "[ ";
    wyswietlTablice(A);
    cout << "]"<<endl<<endl;


  vector<pair<int,int>> pary=znajdzPary(A, k);

  if(pary.size()==0){
    cout<<"Brak par o podanej roznicy k"<<endl;
    return 0;
  }else{
        cout << "Znalezione pary: "<<endl;
      for(int i=0;i<pary.size();i++){
    pair<int,int> para=pary.at(i);
    cout<<"[ "<<para.first<<","<<para.second<<" ]  ";
     }
    }

```

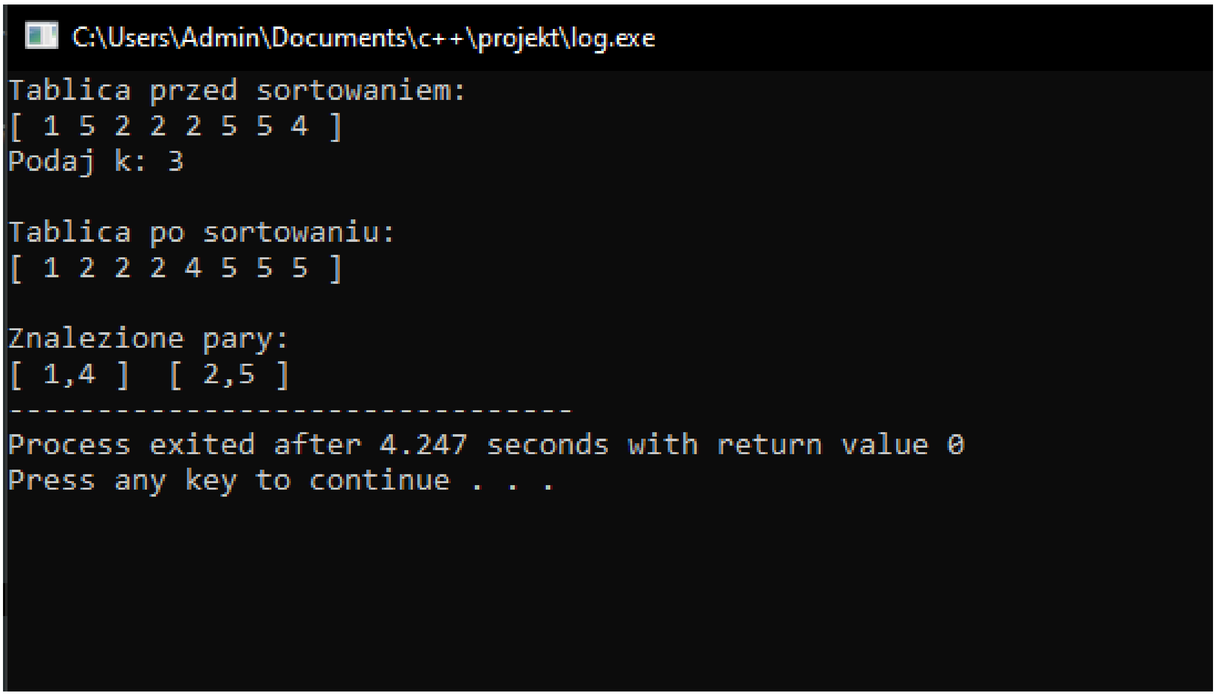
```
    return 0;
}
```

Listing 9: Funkcja main

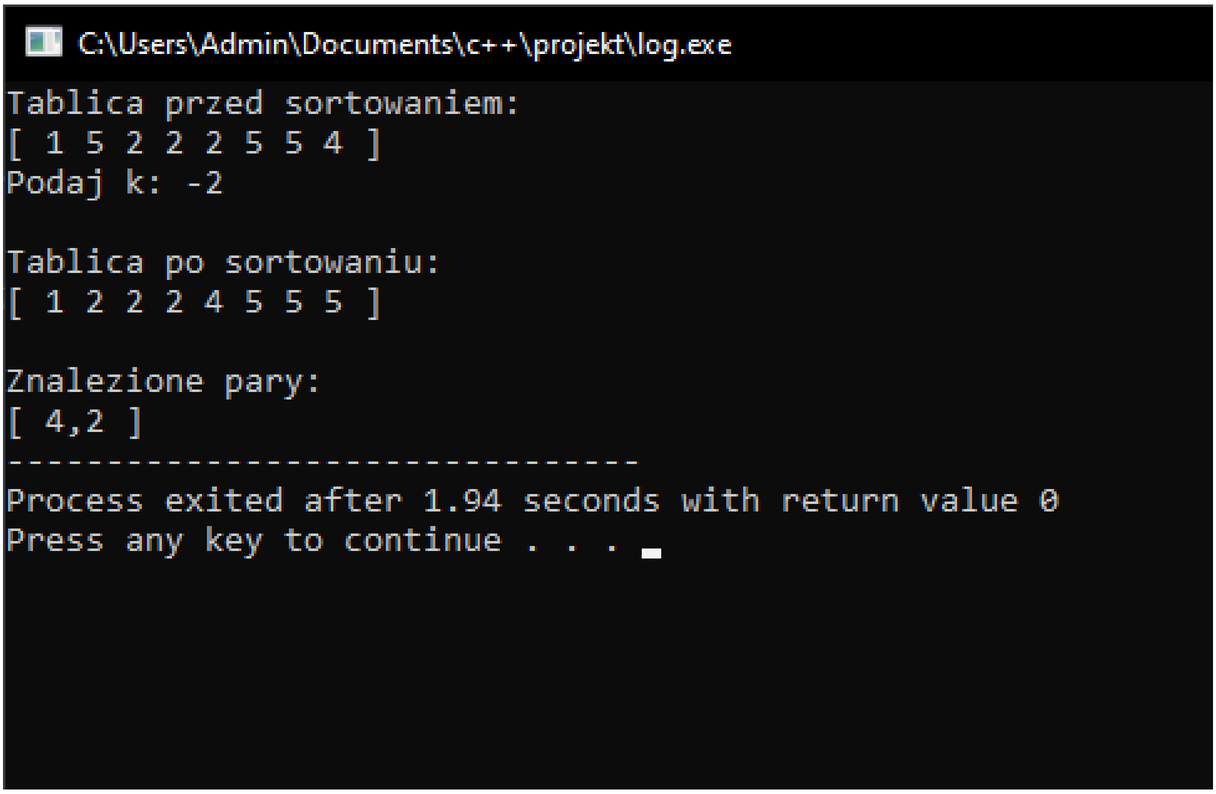
Zainportowanie biblioteki <algorithm> umożliwia użycie funkcji sortującej sort(). Funkcja ta wykorzystuje algorytm typu Introsort, będący połączeniem sortowania szybkiego, sortowania przez kopcowanie oraz sortowania przez wstawianie, co zapewnia złożonośćO ($nlogn$) dla małych jak i dużych liczb.

Introsort to algorytm hybrydowy który w zależności od ilosci sortowanych elementów przełącza się miedzy algorytmami QuickSort, HeapSort i InsertionSort. QuickSort jest najszybszy lecz jego złożoność może spaść do $O(n^2)$, gdy spadek złożoności jest to możliwy, typ sortowania zostaje zmieniony na HeapSort który zawsze posiada złożoność $O(nlogn)$.

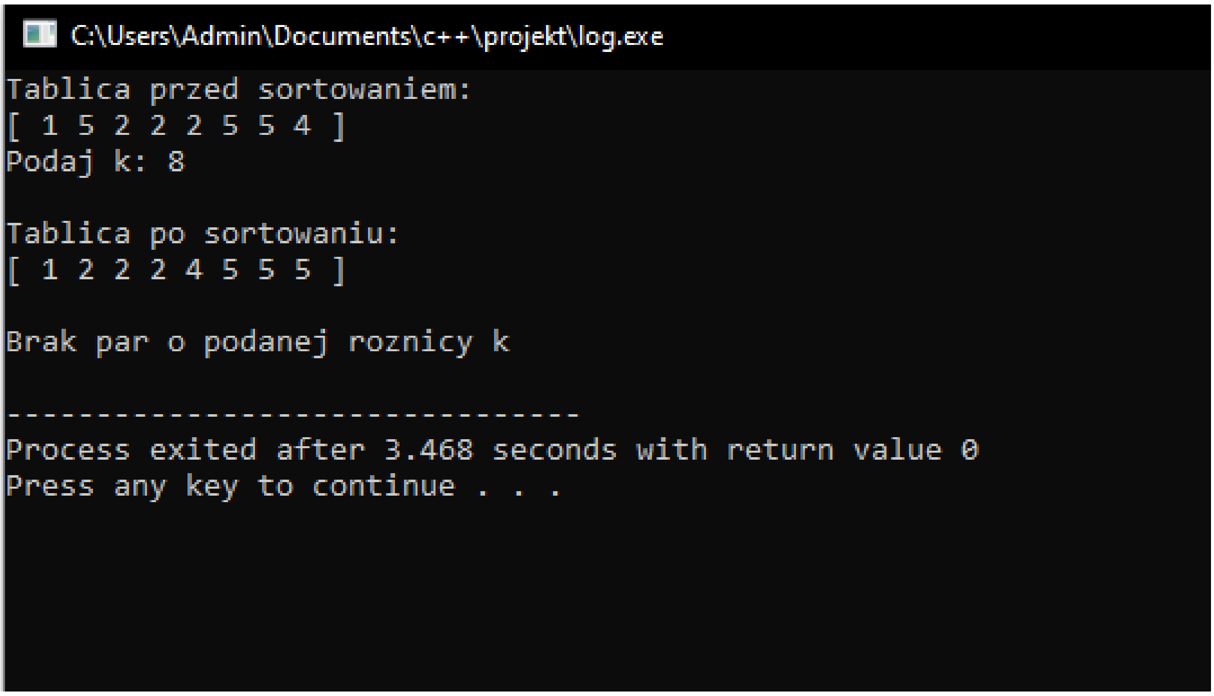
3.5 Wynik działania drugiego programu



Rysunek 6: Przykładowe działanie dla dodatniej różnicy k



Rysunek 7: Przykładowe działanie dla ujemnej różnicy k

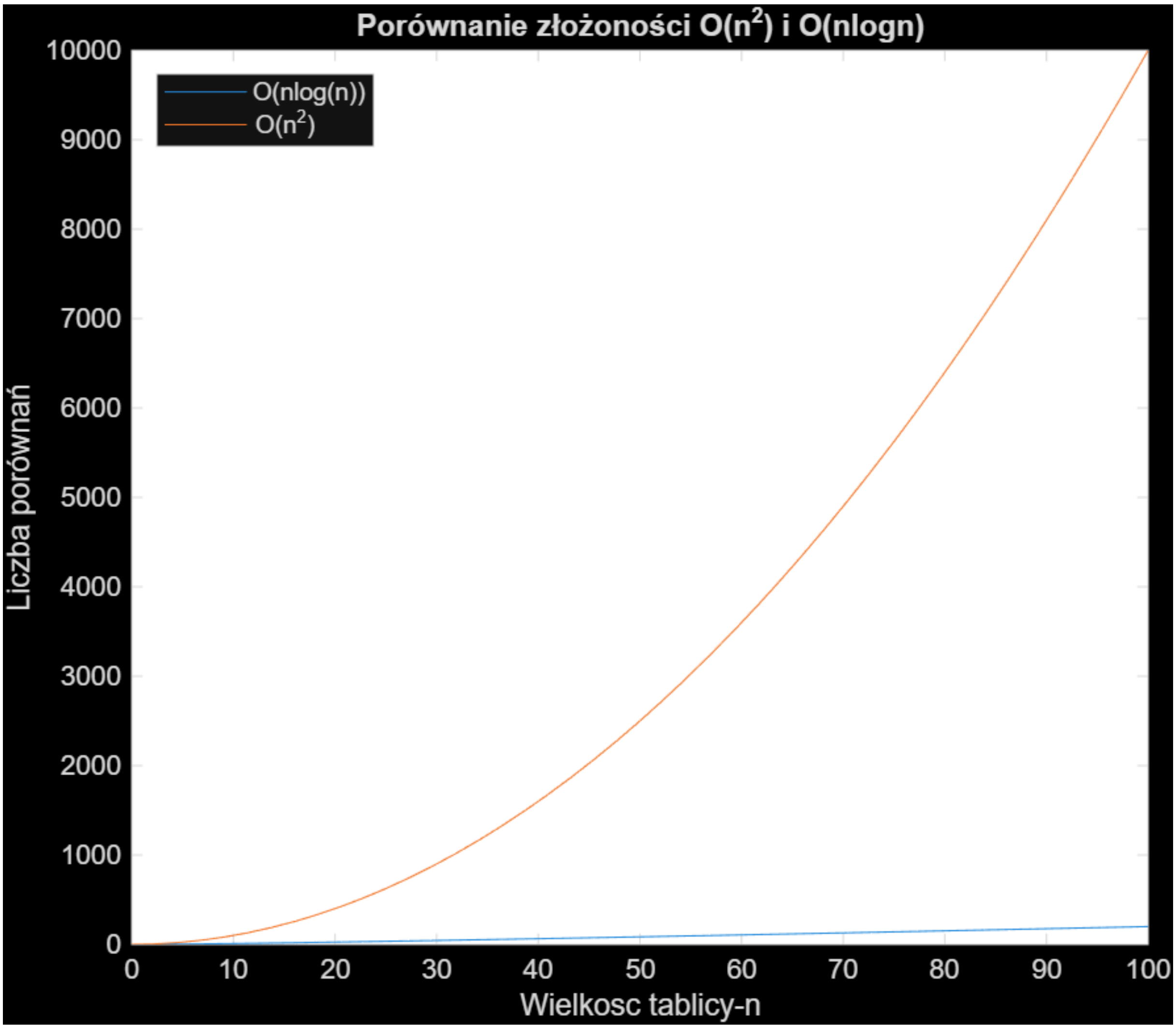


Rysunek 8: Przykładowe działanie dla braku par o roznicy k

4 Podsumowanie

4.1 Porównanie obu złożoności

Na poniższym wykresie można zauważyć jak wielką różnice daje nam zoptymalizowanie poczatkowaego pomysłu o złożoności $O(n^2)$ do $O(nlogn)$. Wraz z zwiekszeniem wielkosci tablicy liczba porównań diametralnie wzrasta dla sposobu pierwszego który był początkowym zamysłem programu.Natomiast liczba porównań dla drugiego sposobu wzrasta w znacznie powolniejszym tempie.



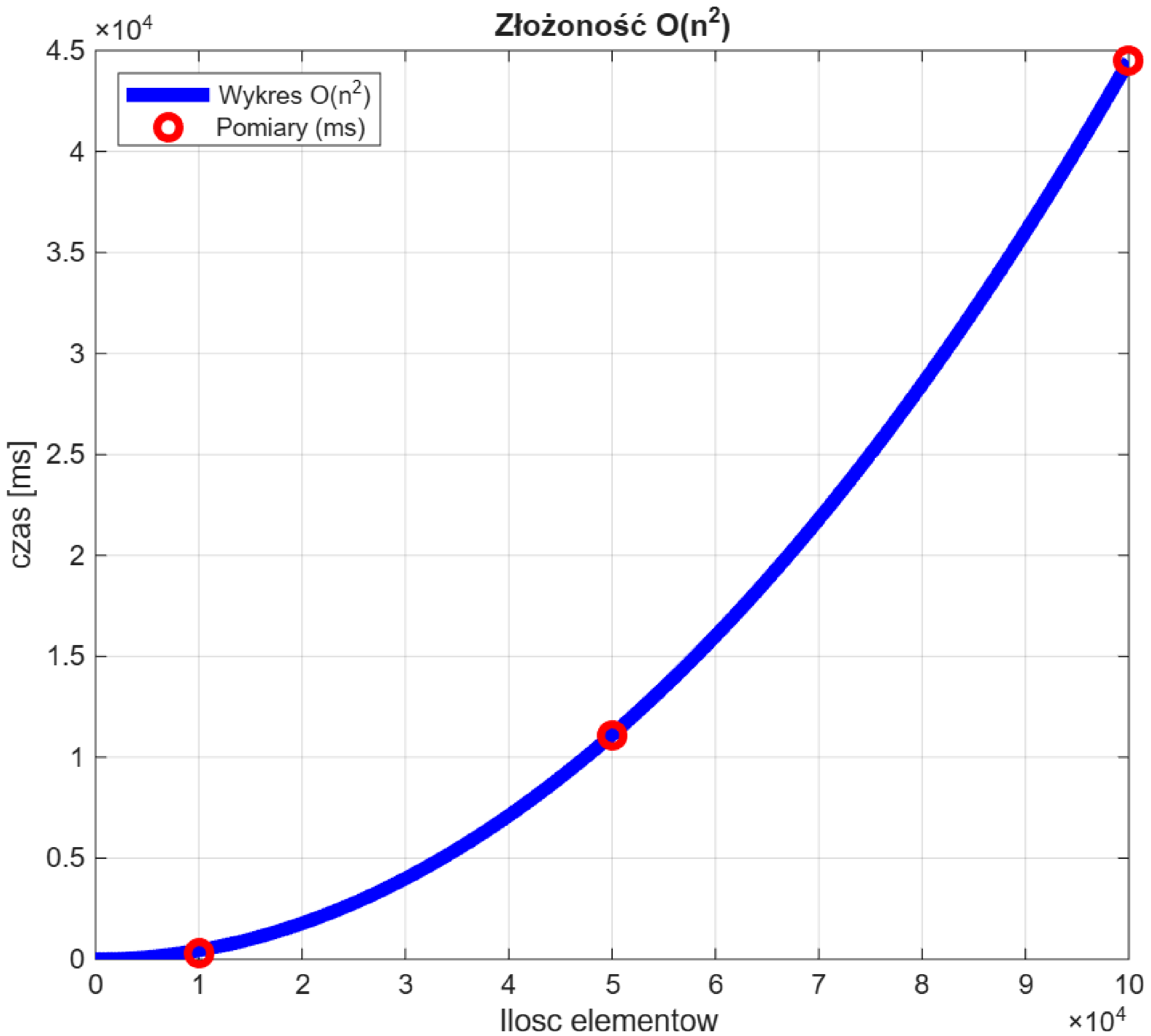
Rysunek 9: Porównanie ilosci operacji

4.2 Testy wydajnościowe

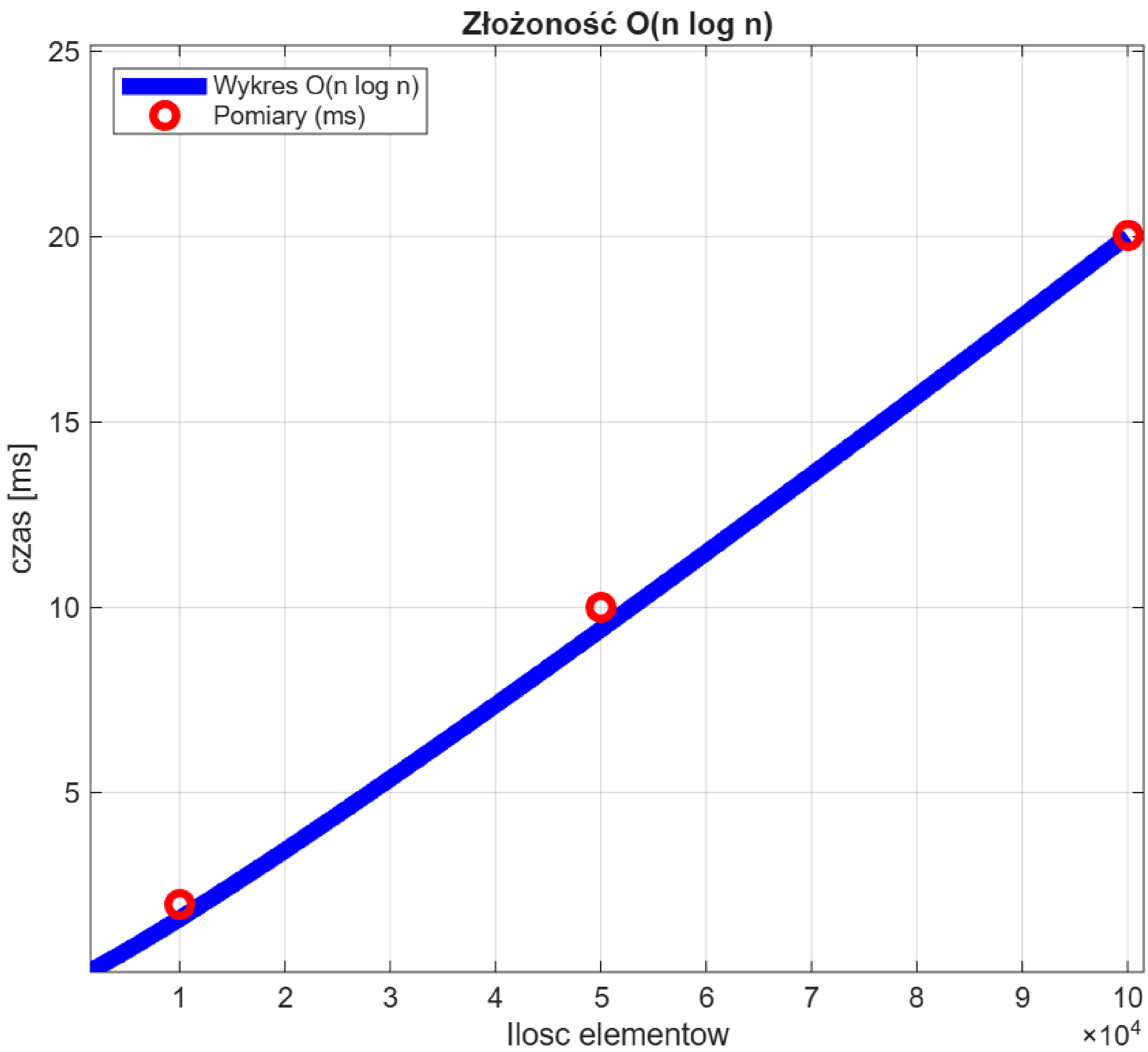
Do policzenia czasu wykonywania obu programów użyłem funkcji wbudowanej w C++ biblioteki <chrono> zawierajacej funkcje do operacji na czasie systemowym.

Ilość elementów	$O(n^2)$	O(nlog n)
10000	277.253 ms	2.001 ms
50000	11068.8 ms	10.011 ms
100000	44491.5 ms	20.019 ms

Tabela 1: Porównanie czasu wykonywania dla poszczególnych wielkosci tablicy



Rysunek 10: Czas wykonań obliczeń dla pierwszego algorytmu o złożoności $O(n^2)$



Rysunek 11: Czas wykonań obliczeń dla drugiego algorytmu o złożoności O(nlogn)

Jak widać na powyższych wykresach czas obliczeń poszczególnych algorytmów pokrywa się z wykresem ich złożoności.

4.3 Wnioski

Ostatecznie udało się zoptymalizować program do znacznie wydajniejszej wersji. Dokładna analiza problemu pozwoliła stworzyć program o mniejszym czasie wykonywania.