



WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Katarzyna Sułtanowa

Projekt

PRACA INŻYNIERSKA

Opiekun pracy:

dr inż. Mariusz Borkowski prof. PRz

Rzeszów, 2025

Spis treści

Zadanie nr 16	5
1. Rozwiązanie - podejście pierwsze(Brute force)	6
1.1. Analiza problemu	6
1.2. Schemat blokowy algorytmu	7
1.3. Algorytm zapisany w pseudokodzie	8
1.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe” rozwiązanie problemu	9
1.5. Teoretyczne oszacowanie złożoności obliczeniowej	10
2. Rozwiązanie- próba druga	11
2.1. Ponowne przemyślenie problemu i próba wymyślenia algorytmu wydajniejszego	11
2.2. Schemat Blokowy (wersja wydajniejsza)	12
2.3. Algorytm zapisany pseudokodzie(wersja wydajniejsza)	14
2.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe” rozwiązanie problemu (wersja wydajniejsza)	15
2.5. Oszacowanie złożoności obliczeniowej dla algorytmu wydajniejszego	16
3. Implementacja wymyślonych algorytmów w wybranym środowisku i języku oraz eksperymentalne potwierdzenie wydajności (złożoności obliczeniowej) algorytmów.	17
3.1. Prosta implementacja	17
3.2. Testy „niewygodnych” zestawów danych	19
3.3. Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej	21
3.4. Wykresy	21
3.5. Testy	22
3.5.1. 1. Wejście: [1, 0, 9, 3, 4, 5, 1, 8, 1, 9, 6, 6, 7, 6] M=3	22
3.5.2. 2. Wejście [2, 4, 8, 4, 5, 5, 9] M=1	23
3.5.3. 3. Wejście [3, 4, 9, 5, 3, 9] M=1	23

Treść zadania

Dla zadanego ciągu liczb wypisz wszystkie elementy powtarzające się, pod warunkiem że elementy te występują w ciągu w odległości nie większej niż M .

Przykład:

Wejście: [1, 0, 9, 3, 4, 5, 1, 8, 1, 9, 6, 6, 7, 6] $M=3$

Wyjście: 1, 6

Wejście: [2, 4, 8, 4, 5, 5, 9] $M=1$

Wyjście: 5

Wejście: [3, 4, 9, 5, 3, 9] $M=1$

Wyjście: Brak elementów spełniających zadane kryteria.

1. Rozwiązanie - podejście pierwsze(Brute force)

1.1. Analiza problemu

Pierwsze podejście siłowe do rozwiązania tego zadania polegało na brute force, czyli sprawdzeniu każdej pary liczb w tablicy i porównaniu ich, bez optymalizacji. Zgodnie z warunkami zadania ciąg dany na wejściu składa się z liczb naturalnych. Najpierw musimy wczytać dane do tablicy. Tablica `vector<int> Tab` jest zaalokowana dynamicznie, więc nie musimy podawać jej rozmiaru na samym początku. Podejdźmy do napisania algorytmu sposobem Brute Force, czyli sprawdzamy każdą możliwą parę, a potem sprawdzamy czy znajdują się te elementy na potrzebnej nam długości.

1. Dane wejściowe:

- Liczby do tablicy: Program prosi użytkownika o wprowadzanie liczb do tablicy. Wprowadzenie liczby większej lub równej 10 kończy proces wprowadzania liczb. Wprowadzone liczby (mniejsze niż 10) są zapisywane do tablicy `Tab`.

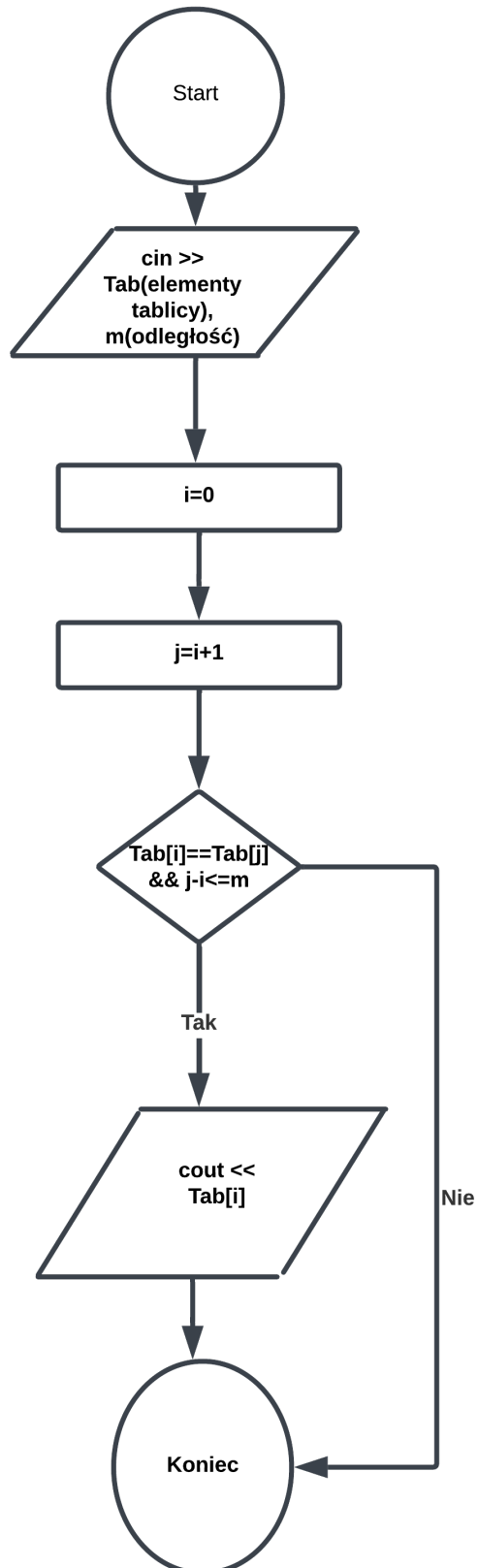
- Wartość odległości `m`: Program następnie prosi o wprowadzenie liczby `m`, która jest określoną odległością. Wartość ta musi być większa niż 0, inaczej program kończy działanie z komunikatem o błędzie.

2. Dane wyjściowe:

- Jeśli program znajdzie odpowiednie pary: program wypisuje te liczby na ekranie, po jednym dla każdej liczby, która występuje więcej niż raz.

- Jeśli program nie znalazł par, spełniających podane kryteria lub odległość podana niewłaściwie: program wypisze komunikat, że brak jest elementów spełniających kryteria.

1.2. Schemat blokowy algorytmu



1.3. Algorytm zapisany w pseudokodzie

1 Input:

Tab[i] // elementy tablicy

m // odległość

3 Output:

Tab[i] // element powtarzający się na zadanej odległości

4 if $m > 0$

5 bool znalazlem = false; // Zmienna do śledzenia, czy znaleziono jakąkolwiek

liczbę

6 int i=0

7 int j=i+1

8 if ($\text{Tab}[i] == \text{Tab}[j]$ $j-i \leq m$)

9 znalazlem=true

10 cout « Tab[i]

11 Zakończ program.

1.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe” rozwiązanie problemu

Wejście: [2, 4, 8, 4, 5, 5, 9] M=1

i	j	Tab[i]	Tab[j]	Tab[i]==Tab[j]	j-i <= m	znalazlem
0	1	2	4	0	0	false
0	2	2	8	0	0	false
0	3	2	4	0	0	false
0	4	2	5	0	0	false
0	5	2	5	0	0	false
0	6	2	9	0	0	false
1	2	4	8	0	0	false
1	3	4	4	1	0	false
1	4	4	5	0	0	false
1	5	4	5	0	0	false
1	6	4	9	0	0	false
2	3	8	4	0	0	false
2	4	8	5	0	0	false
2	5	8	5	0	0	false
2	6	8	9	0	0	false
3	4	4	5	0	0	false
3	5	4	5	0	0	false
3	6	4	9	0	0	false
4	5	5	5	1	1	true
4	6	5	9	0	0	false
5	6	5	9	0	0	false

Po "Ołówkowym" sprawdzeniu poprawności algorytmu otrzymamy następujące wyniki: [5]

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 $i=0$: ilość porównań jest równa $k-1$;

dla $i=1$: ilość porównań jest równa $k-2$;

itd.

Zewnętrzna pętla (pierwsza pętla `for`) iteruje przez wszystkie liczby w tablicy, czyli od 0 do $k-1$. Czas działania tej pętli to $O(k)$. Wewnętrzna pętla (druga pętla `for`) porównuje liczbę w tablicy z wszystkimi kolejnymi liczbami w tablicy. Czas działania tej pętli to $O(k)$ w najgorszym przypadku. W związku z tym, złożoność tych dwóch zagnieżdżonych pętli to $O(k^2)$.

2. Rozwiązanie- próba druga

2.1. Ponowne przemyślenie problemu i próba wymyślenia algorytmu wydajniejszego

Chociaż wyniki ołówkowego sprawdzania programu zgadzają się z przykładowymi wynikami, można jeszcze zmodyfikować ten kod tak, aby wyeliminować następne wady danego kodu:

Niska efektywność, złożoność obliczeniowa tego kodu jest $O(k^2)$,

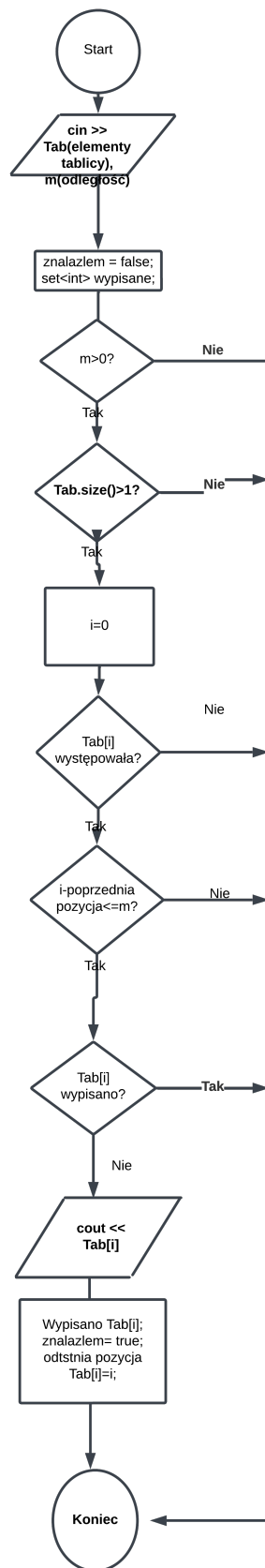
związane to jest z tym, że program przechodzi przez całą pętlę, sprawdza każdy element.

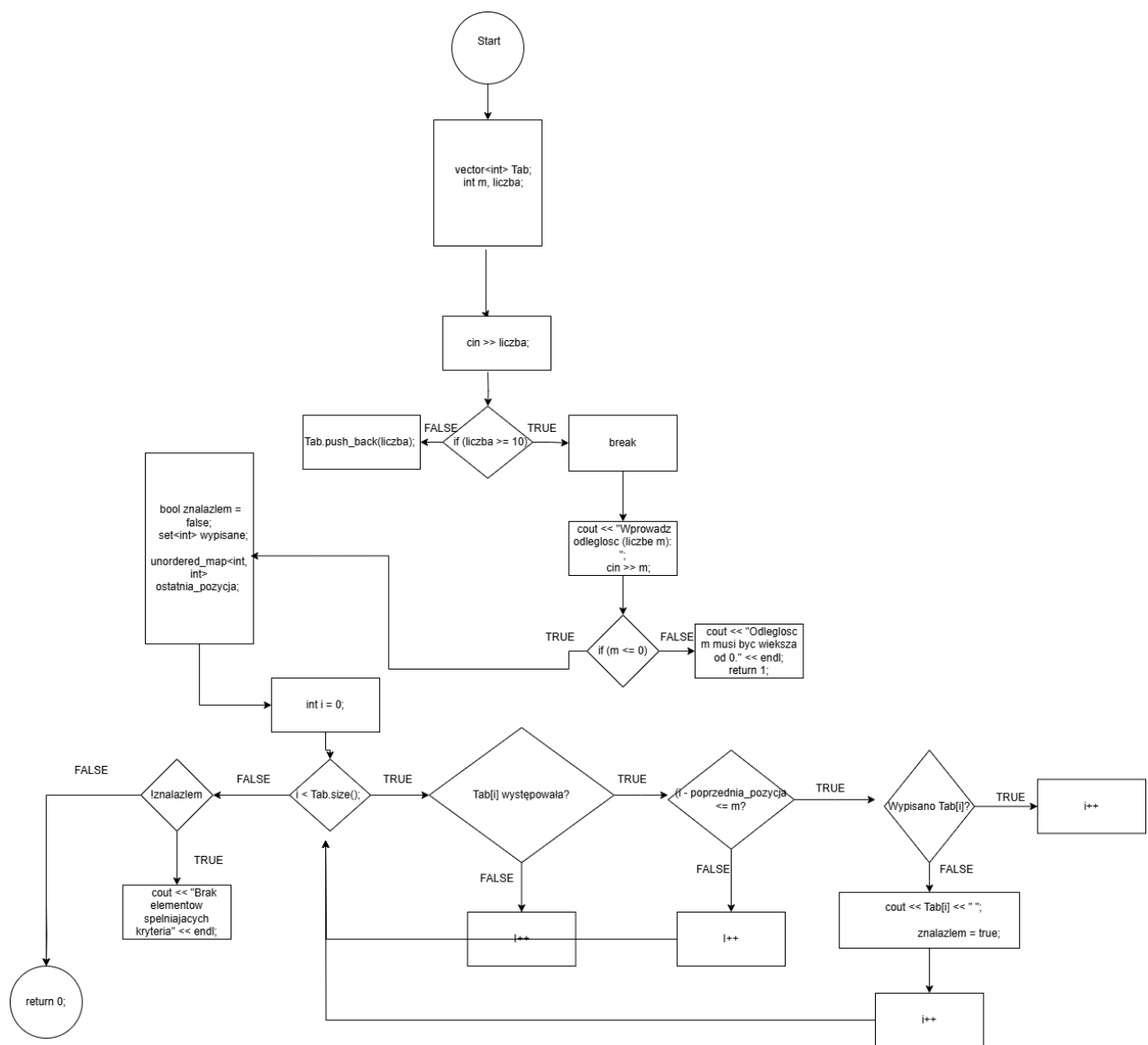
Algorytm niepotrzebnie sprawdza powtarzające się więcej niż 2 razy elementy, bo wynik będzie identyczny.

Brak wcześniejszego przerywania pętli, jeśli liczba już została znaleziona. Wiele z następujących obliczeń jest niepotrzebne.

Musimy zmodyfikować ten algorytm tak, aby porównywał tylko te elementy, które są w odległości m od siebie, co ograniczało by liczbę sprawdzanych par. Zaimplementować nową tablicę, która by pomagała zapobiegać wielokrotnemu wypisywaniu tego samego elementu.

2.2. Schemat Blokowy (wersja wydajniejsza)





2.3. Algorytm zapisany pseudokodzie(wersja wydajniejsza)

1 Input:

Tab[i] // elementy tablicy

m // odległość

3 Output:

Tab[i] // element powtarzający się na zadanej odległości

4 if m>0

5 bool znalazlem = false; // Zmienna do śledzenia, czy znaleziono jakąkolwiek liczbę

6 wypisane := pusty zbiór

7 ostatnia pozycja := pusta mapa

8 for i := 0 ; i <= długość(tab) - 1

9 liczba := tab[i]

10 if liczba jest w mapie ostatnia pozycja

11 poprzednia pozycja := ostatnia pozycja[liczba]

12 if i - poprzednia pozycja <= m

13 if liczba nie jest w zbiorze wypisane

14 wypisz liczba // wypisanie elementu

15 dodaj liczba do zbioru wypisane

16 ustaw znalazlem na prawda

17 endif

18 endif

19 endif

20 zaktualizuj mapę ostatnia pozycja[liczba] = i

21 endfor

22 if znalazlem == fałsz

23 wypisz "Brak elementów spełniających kryteria"

24 endif

2.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe” rozwiązanie problemu (wersja wydajniejsza)

wejście = [2, 4, 8, 4, 5, 5, 9] M=1 obliczenia będą się przedstawiały następująco

i	Tab[i]	ostatnia_pozycja	odl. pom. el.	znalazlem
0	2	0	∅	false
1	4	1	∅	false
2	8	2	∅	false
3	4	3	2	false
4	5	4	∅	false
5	5	5	1	true
6	9	6	∅	false

Możemy zauważyć, że ten kod potrzebuje znacznie mniej porównań dla znalezienia pary. Przy ołówkowym sprawdzaniu poprawności używano mapę ostatnia pozycja po aktualizowaniu, żeby można było sprawdzić czy odległość pomiędzy dublikatami jest mniejsza lub w tym przypadku równa 1.

2.5. Oszacowanie złożoności obliczeniowej dla algorytmu wydajniejszego

Pętla (`for int i=0; i < Tab.size(); i++`) przechodzi przez całą tablicę ($O(k)$), gdzie k to ilość elementów tablicy.

Algorytm wykonuje poszczególne operacje związane z mapami: sprawdzanie i dodawanie elementów do map. Są to między innymi takie operacje jak na przykład sprawdzenie, czy liczba już występuje w mapie ostatnia pozycja (`if (ostatnia pozycja.find(Tab[i]) != ostatnia pozycja.end())`), porównanie pozycji liczby w mapie, obliczenie odległości i porównanie z m , sprawdzenie, czy liczba została już wypisana i tak dalej.

Te operacje mają $O(1)$ złożoność obliczeniową. Dlatego w wyniku otrzymujemy, że złożoność obliczeniowa tego algorytmu jest równa $O(k)$.

3. Implementacja wymyślonych algorytmów w wybranym środowisku i języku oraz eksperymentalne potwierdzenie wydajności (złożoności obliczeniowej) algorytmów.

3.1. Prosta

implementacja

Przykład programu pierwszego (uproszczonego)

```
#include <iostream>
#include <vector>
#include <set>
#include <unordered_map>
using namespace std;

int main() {

    vector<int> Tab;
    int m, liczba;

    // Wprowadzanie liczb do tablicy
    cout << "Wprowadz liczby: " << endl;
    cout << "Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9" << endl;
    while (true) {
        cin >> liczba;
        if (liczba >= 10) {
            break; // Petla konczy sie, gdy liczba jest >= 10
        }
        Tab.push_back(liczba); // Dodajemy liczbe do wektora
    }

    // Wprowadzamy wartosc m
    cout << "Wprowadz odleglosc (liczbe m): ";
    cin >> m;

    // Sprawdzamy, czy m jest wieksza niz 0
    if (m <= 0) {
        cout << "Odleglosc m musi byc wieksza od 0." << endl;
        return 1; // Konczymy, jezeli m nie jest poprawne
    }
    bool znalazlem = false;
    set<int> wypisane; // Zbiór do przechowywania już wypisanych liczb

    // Zagnieżdżona petla do porównania każdej liczby z pozostałymi
    for (int i = 0; i < Tab.size(); i++) {
        for (int j = i + 1; j < Tab.size(); j++) {
            if (Tab[i] == Tab[j] && j - i <= m) {
                // Sprawdzamy, czy ta liczba już była wypisana
                if (wypisane.find(Tab[i]) == wypisane.end()) {
                    cout << Tab[i] << " ";
                    wypisane.insert(Tab[i]); // Dodajemy liczbę do zbioru
                    znalazlem = true;
                    break; // Nie musimy szukać dalej, bo wystarczy pierwszy wynik
                }
            }
        }
    }

    if (!znalazlem) {
        cout << "Brak elementow spelniajacych kryteria" << endl;
    }
    // Wywołanie funkcji
    // funkcja1(Tab, m);
    //cout << endl; // Dla lepszej czytelności wycięcia
    // funkcja2(Tab, m);

    return 0;
}
```

Rys. 3.1. Kod programu

Przykład programu wydajniejszego

```
#include <unordered_map>
using namespace std;

int main() {

    vector<int> Tab;
    int m, liczba;

    // Wprowadzanie liczb do tablicy
    cout << "Wprowadz liczby: " << endl;
    cout << "Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9" << endl;
    while (true) {
        cin >> liczba;
        if (liczba >= 10) {
            break; // Petla konczy sie, gdy liczba jest >= 10
        }
        Tab.push_back(liczba); // Dodajemy liczbe do wektora
    }

    // Wprowadzamy wartosc m
    cout << "Wprowadz odleglosc (liczbe m): ";
    cin >> m;

    // Sprawdzamy, czy m jest wieksza niz 0
    if (m <= 0) {
        cout << "Odleglosc m musi byc wieksza od 0." << endl;
        return 1; // Konczymy, jesli m nie jest poprawne
    }

    bool znalazlem = false;
    set<int> wypisane; // Zbiór do przechowywania już wypisanych liczb
    unordered_map<int, int> ostatnia_pozycja; // Mapa przechowująca ostatnia pozycje liczby

    // Przechodzimy przez tablice
    for (int i = 0; i < Tab.size(); i++) {
        // Sprawdzamy, czy ta liczba już wystąpiła w odległości m
        if (ostatnia_pozycja.find(Tab[i]) != ostatnia_pozycja.end()) {
            int poprzednia_pozycja = ostatnia_pozycja[Tab[i]];

            // Jeśli odległość między poprzednia a obecna pozycja jest mniejsza lub równa m, wypisujemy liczbę
            if (i - poprzednia_pozycja <= m) {
                // Sprawdzamy, czy ta liczba już była wypisana
                if (wypisane.find(Tab[i]) == wypisane.end()) {
                    cout << Tab[i] << " ";
                    wypisane.insert(Tab[i]); // Dodajemy liczbę do zbioru
                    znalazlem = true;
                }
            }
        }
        // Zapisujemy bieżącą pozycję liczby w mapie
        ostatnia_pozycja[Tab[i]] = i;
    }

    if (!znalazlem) {
        cout << "Brak elementow spelniajacych kryteria" << endl;
    }

    return 0;
}
```

Rys. 3.2. Kod programu

3.2. Testy „niewygodnych” zestawów danych

Sprawdzimy działanie kodu wersji pierwotnego oraz bardziej wydajniejszego dla następnego wejścia.

Przykład programu pierwszego (uproszczonego) i wydajniejszego

```
#include <iostream>
#include <vector>
#include <set>
#include <unordered_map>
using namespace std;

void funkcja1(const vector<int>& Tab, int m) {
    bool znalazlem = false;
    set<int> wypisane; // Zbiór do przechowywania już wypisanych liczb

    // Zagnieżdżona petla do porównania każdej liczby z pozostałymi
    for (int i = 0; i < Tab.size(); i++) {
        for (int j = i + 1; j < Tab.size(); j++) {
            if (Tab[i] == Tab[j] && j - i <= m) {
                // Sprawdzamy, czy ta liczba już była wypisana
                if (wypisane.find(Tab[i]) == wypisane.end()) {
                    cout << Tab[i] << " ";
                    wypisane.insert(Tab[i]); // Dodajemy liczbę do zbioru
                    znalazlem = true;
                    break; // Nie musimy szukać dalej, bo wystarczy pierwszy wynik
                }
            }
        }
    }

    if (!znalazlem) {
        cout << "Brak elementow spelniajacych kryteria" << endl;
    }
}

void funkcja2(const vector<int>& Tab, int m) {
    bool znalazlem = false;
    set<int> wypisane; // Zbiór do przechowywania już wypisanych liczb
    unordered_map<int, int> ostatnia_pozycja; // Mapa przechowująca ostatnia pozycję liczby

    // Przechodzimy przez tablice
    for (int i = 0; i < Tab.size(); i++) {
        // Sprawdzamy, czy ta liczba już wystąpiła w odległości m
        if (ostatnia_pozycja.find(Tab[i]) != ostatnia_pozycja.end()) {
            int poprzednia_pozycja = ostatnia_pozycja[Tab[i]];
            // Jeśli odległość między poprzednią a bieżącą pozycją jest mniejsza lub równa m, wypisujemy liczbę
            if (i - poprzednia_pozycja <= m) {
                // Sprawdzamy, czy ta liczba już była wypisana
                if (wypisane.find(Tab[i]) == wypisane.end()) {
                    cout << Tab[i] << " ";
                    wypisane.insert(Tab[i]); // Dodajemy liczbę do zbioru
                    znalazlem = true;
                }
            }
        }
        // Zapisujemy bieżącą pozycję liczby w mapie
        ostatnia_pozycja[Tab[i]] = i;
    }

    if (!znalazlem) {
        cout << "Brak elementow spelniajacych kryteria" << endl;
    }
}
```

```

int main() {
    vector<int> Tab;
    int m, liczba;

    // Wprowadzanie liczb do tablicy
    cout << "Wprowadz liczby: " << endl;
    cout << "Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9" << endl;
    while (true) {
        cin >> liczba;
        if (liczba >= 10) {
            break; // Petla konczy sie, gdy liczba jest >= 10
        }
        Tab.push_back(liczba); // Dodajemy liczbe do wektora
    }

    // Wprowadzamy wartosc m
    cout << "Wprowadz odleglosc (liczbe m): ";
    cin >> m;

    // Sprawdzamy, czy m jest wieksze niz 0
    if (m <= 0) {
        cout << "Odleglosc m musi byc wieksza od 0." << endl;
        return 1; // Konczymy, jezeli m nie jest poprawne
    }

    // Wywołanie funkcji
    cout << "wynik programu naiwnego:" << endl;
    funkcja1(Tab, m);
    cout << endl << "wynik programu wydajniejszego:" << endl; // Dla lepszej czytelności wyśledzia
    funkcja2(Tab, m);

    return 0;
}

```

Rys. 3.3. Kod programu

```

Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
2
4
8
4
5
5
9
10
Wprowadz odleglosc (liczbe m): 1
wynik programu naiwnego:
5
wynik programu wydajniejszego:
5

```

Rys. 3.4. Wynik działania kodu dla [2, 4, 8, 4, 5, 5, 9] M=1

Wejście: [1, 1, 1, 1, 2, 3, 2, 3] M=2

```

Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
1
1
1
1
2
3
2
3
10
Wprowadz odleglosc (liczbe m): 2
wynik programu naiwnego:
1 2 3
wynik programu wydajniejszego:
1 2 3

```

Rys. 3.5. Wynik działania kodu dla [1, 1, 1, 1, 2, 3, 2, 3] M=2

3.3. Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej

```
Wprowadz odleglosc (liczbe m): 2

L.p. n t1 [s] t2 [s]
1 2500 0.024 0.001
2 5000 0.05 0
3 10000 0.188 0.001
4 20000 0.725 0.003
5 30000 1.605 0.003
6 40000 2.85 0.005
7 50000 4.51 0.005
8 60000 6.492 0.008
9 70000 9.257 0.011
10 80000 12.32 0.009
```

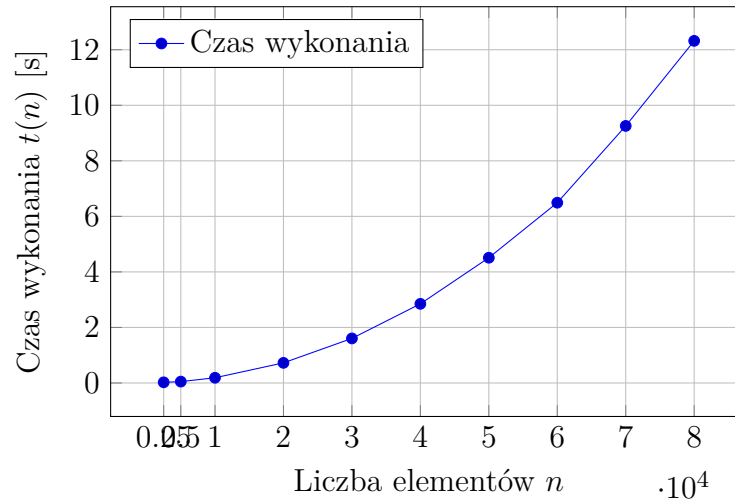
Rys. 3.6. Wynik testu czasowego dla kodu wydajniejszego

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. 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

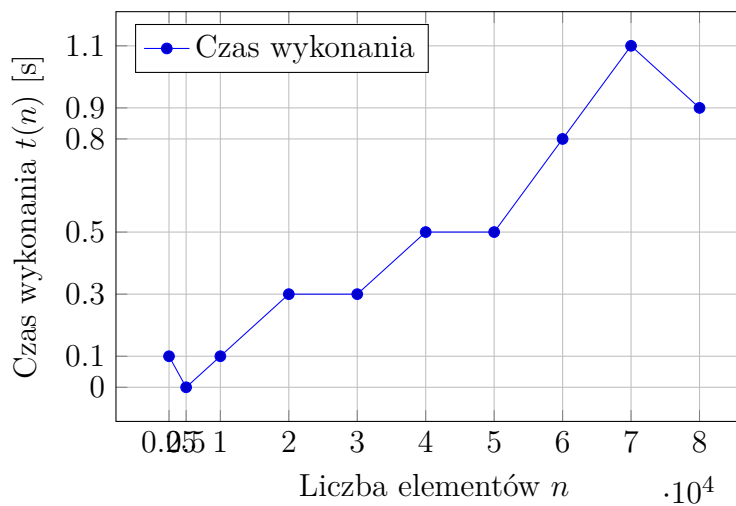
3.4. Wykresy

Z podanego niżej wykresu możemy zauważyć, że czasy obliczeń dla kodu prostszego oraz wydajniejszego różnią się. Kod wydajniejszy potrzebuje znacznie mniej czasu. Natomiast kod prostszy potrzebuje bardzo dużo czasu, żeby przejść przez każdy element.

Wykres czasów dla wesji naiwnej



Wykres czasów dla wesji wydajnej



3.5. Testy

3.5.1. 1. Wejście: [1, 0, 9, 3, 4, 5, 1, 8, 1, 9, 6, 6, 7, 6] M=3

```

Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
1
0
9
3
4
5
1
8
1
9
6
6
7
6
10
Wprowadz odleglosc (liczbe m): 3
wynik programu naiwnego:
1 6
wynik programu wydajniejszego:
1 6
    
```

Rys. 3.7. Wyniki testu pierwszego dla kodu

3.5.2. 2. Wejście [2, 4, 8, 4, 5, 5, 9] M=1

```
Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
2
4
8
4
5
5
9
10
Wprowadz odleglosc (liczbe m): 1
wynik programu naiwnego:
5
wynik programu wydajniejszego:
5
```

Rys. 3.8. Wyniki testu drugiego dla kodu

3.5.3. 3. Wejście [3, 4, 9, 5, 3, 9] M=1

```
Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
3
4
9
5
3
9
10
Wprowadz odleglosc (liczbe m): 1
wynik programu naiwnego:
Brak elementow spelniajacych kryteria
wynik programu wydajniejszego:
Brak elementow spelniajacych kryteria
```

Rys. 3.9. Wyniki testu trzeciego dla kodu

```
Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
3
4
9
4
5
10
Wprowadz odleglosc (liczbe m): 0
Odleglosc m musi byc wieksza od 0.
```

Rys. 3.10. Wyniki testu dla $m < 1$