WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Katarzyna Sułtanowa

Projekt

PRACA INŻYNIERSKA

Opiekun pracy:
dr inż. Mariusz Borkowski prof. PRz

Rzeszów, 2025

Spis treści

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. Zdodnie z warunkami zadania ciąg dany na wejściu składa się z liczb naturalnych. W tym przypadku nie używałam tablicy `wypisano`, która śledzi, które liczby zostały już wypisane, a także złożoność algorytmu była wyższa, ponieważ nie było sprawdzania tylko tych par, które są w odległości m od siebie. Porównywałam wszystkie możliwe pary liczb w tablicy, nawet jeśli były one oddzielone o więcej niż m pozycji. W efekcie, to było dużo mniej wydajne, ponieważ sprawdzanie wszystkich par zajmowało więcej czasu. Nie miałam sprawdzenia warunku `if (Tab[w] == Tab[s])`– zamiast tego, po prostu wykonywałam porównanie dla wszystkich liczb, co zwiększało liczbę sprawdzanych par.

Zauważyłam jeszcze parę wad w podejściu Brute force dla rozwiązania tego zadania. Są to między innymi Niska efektywnośc – każda para liczb jest porównywana bez optymalizacji, więc algorytm może działać znacznie wolniej przy większej liczbie elementów w tablicy. Potencjalne powtarzanie się par – w tym podejściu każdą liczbę porównujemy z każdą inną, co prowadzi do sytuacji, w której te same liczby mogą być wypisane wielokrotnie.

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.

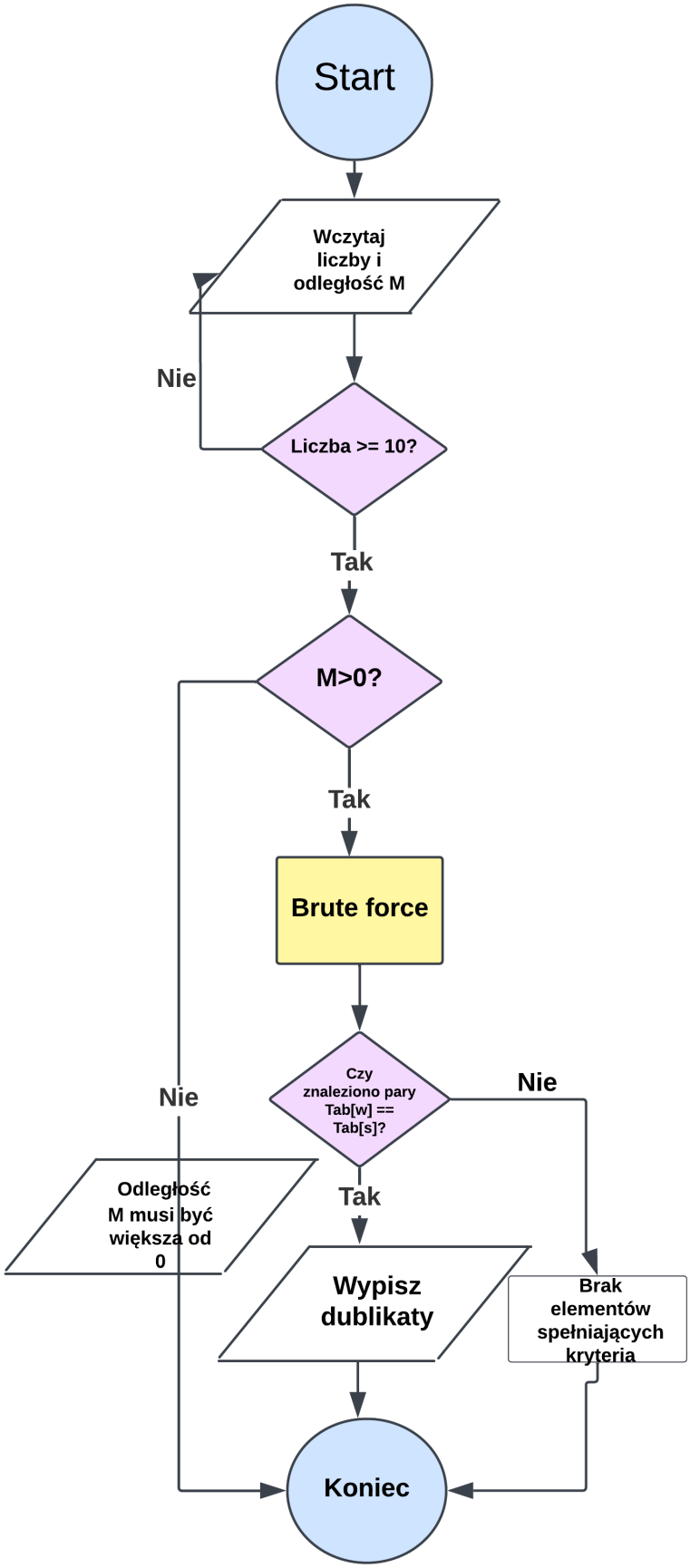
- Wartość odległości m: Program następnie prosi o wprowadzenie liczby m, która jest określoną odległością. Wartość ta musi być większa niż 0, inaczej program kończy działanie z komunikatem o błędzie.

2. Dane wyjściowe:

- Jeśli program znajdzie odpowednie 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 niewłaściwe: program wypisze komunikat, że brak jest elementów spełniających kryteria.

1.2. Schemat blokowy algorytmu



Text

1.3. Algorytm zapisany w pseudokodzie

1. Inicjalizuj tablicę o rozmiarze 100.

2. Inicjalizuj zmienne: k (do liczenia liczby wprowadzonych liczb), m (odległość), liczba (wprowadzone liczby).

3. Wprowadźliczby do tablicy:

Wyświetl komunikat "Wprowadź liczby". Powtarzaj: Wprowadź liczbę. Jeśli liczba jest większa lub równa 10, zakończ wprowadzanie liczb. W przeciwnym razie zapisz liczbę do tablicy w pozycji k i zwiększ k o 1.

4.Wprowadź wartość m (odległość):

Wyświetl komunikat "Wprowadź odległość (liczbę m)". Wprowadź wartość m.

5. Sprawdź, czy m jest większe niż 0:

Jeśli m <= 0, wyświetl komunikat "Odległość m musi być większa od 0."i zakończ działanie programu.

6. Sprawdzanie duplikatów z uwzględnieniem odległości m:

Ustaw zmienną znalazlem na fałsz. Dla każdej liczby w tablicy (od indeksu 0 do k - 1): Dla każdej liczby, która znajduje się po niej w tablicy, ale nie dalej niż o m pozycji (od w + 1 do w + m): Jeśli liczby są równe: Wyświetl komunikat "Znaleziono duplikat: <liczba> na pozycjach <w> i <s>". Ustaw znalazlem na prawdę.

7. Jeśli nie znaleziono żadnych duplikatów, wyświetl komunikat:

"Brak elementów spełniających kryteria."

8. 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]	znaleziono
0	1	2	4	0	false
0	2	2	8	0	false
0	3	2	4	0	false
0	4	2	5	0	false
0	5	2	5	0	false
0	6	2	9	0	false
1	2	4	8	0	false
1	3	4	4	1	true
1	4	4	5	0	false
1	5	4	5	0	false
1	6	4	9	0	false
2	3	8	4	0	false
2	4	8	5	0	false
2	5	8	5	0	false
2	6	8	9	0	false
3	4	4	5	0	false
3	5	4	5	0	false
3	6	4	9	0	false
4	5	5	5	1	true
4	6	5	9	0	false
5	6	5	9	0	false

Po "Ołówkowym"sprawdzeniu poprawności algorytmu otzrymamy następujące wyniki:

[4 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

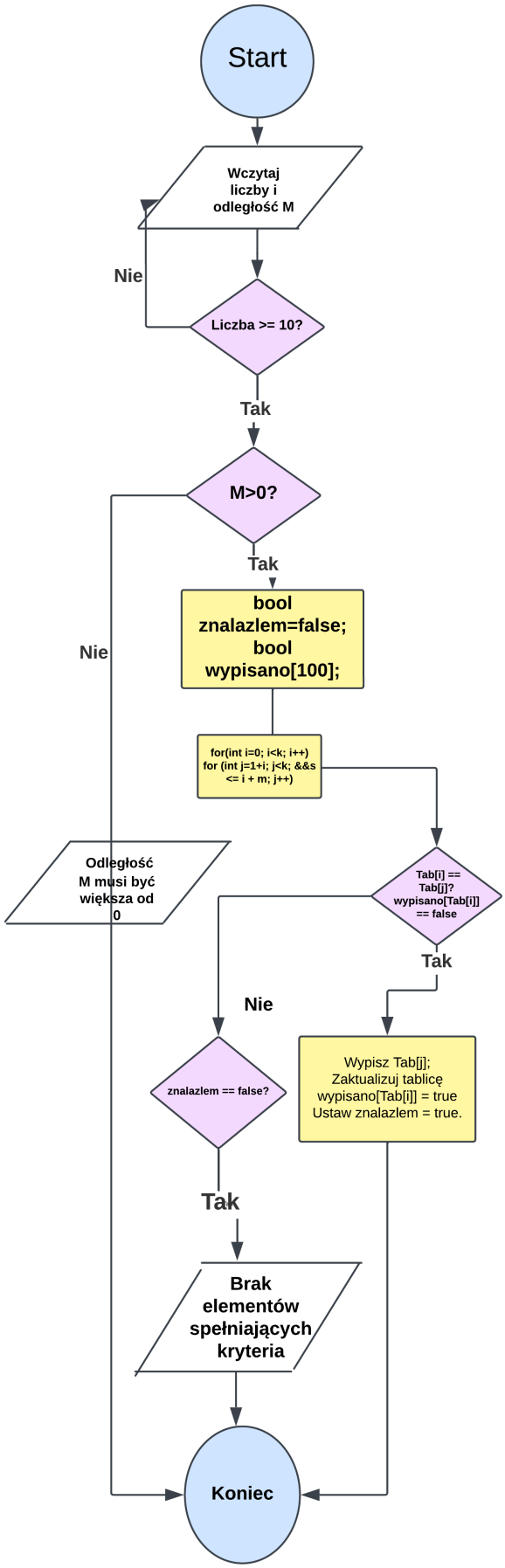
Znalezione wady kodu pierwotnego szczegółowo już są opisane w rozdziale pierwszym. Najważniejsze wnioski:

Brak kontroli nad odległością m,

Niska efektywność,

Potencjalne powtarzanie liczb,
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. Start

2.Inicjalizuj zmienne:

Tablica Tab[100]

Zmienna k = 0 (liczba wprowadzonych liczb)

Zmienna m (odległość)

Zmienna liczba (wprowadzone liczby

3. Wprowadź liczby do tablicy:

Wyświetl komunikat "Wprowadź liczby".

Powtarzaj:

Wprowadź liczbę liczba.

Jeśli liczba >= 10, zakończ wprowadzanie liczb.

W przeciwnym razie:

Zapisz liczba w tablicy Tab[k].

Zwiększ k o 1 (przechodzimy do kolejnej pozycji tablicy).

Wprowadź wartość m (odległość):

4. Wyświetl komunikat "Wprowadź odległość (liczbę m)".

Wprowadź wartość m.

5. Sprawdź, czy m jest większe niż 0:

Jeśli m <= 0, wyświetl komunikat "Odległość m musi być większa od 0"i zakończ program.

6.Inicjalizuj tablicę logiczną wypisano[100] jako fałsz (false).

Zmienna znalazlem = false (informacja, czy znaleziono duplikaty).

7.Porównuj liczby w tablicy:

Dla każdej liczby i w tablicy (od 0 do k-1):

Dla każdej liczby j w tablicy, która znajduje się po liczbie i i odległość j <= i + m:

Jeśli Tab[i] == Tab[j] i wypisano[Tab[i]] == false (czyli liczby są równe i jeszcze nie zostały wypisane):

Wypisz Tab[j].

Ustaw wypisano[Tab[i]] = true (oznacz liczbę jako wypisaną).

Ustaw znalazlem = true.

8. Sprawdzenie, czy znaleziono duplikaty:

Jeśli znalazlem == false, wyświetl komunikat "Brak elementów spełniających kryteria".

9.Koniec programu

Kluczowe elementy pseudokodu:

Pętla wewnętrzna porównuje liczby w tablicy, aby znaleźć duplikaty w zadanej odległości.

Zmienna wypisano zapewnia, że każda liczba zostanie wypisana tylko raz.

Program kończy się komunikatem, jeśli nie znaleziono żadnych duplikatów spełniających kryteria.

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	j	Tab[i]	Tab[j]	Tab[i]==Tab[j]	wypisano
0	1	2	4	0	false
1	2	4	8	0	false
2	3	8	4	0	false
3	4	4	5	0	false
4	5	5	5	1	true
5	6	5	9	0	false

Możemy zauważyć, że ten kod potrzebuje znacznie mniej porównań dla znalezienia pary.

2.5. Oszacowanie złożoności obliczeniowej dla algotytmu wydajniejszego

Wewnętrzna pętla porównuje liczbę Tab[i] z liczbami od Tab[i+1] do Tab[i+m], przy czym ograniczeniem jest m. W najgorszym przypadku, gdy odległość m jest bardzo duża, pętla wewnętrzna może przejść przez wszystkie pozostałe liczby, co daje złożoność O(k). Jednak, w ogólnym przypadku, pętla wewnętrzna ma złożoność O(m), ponieważ iteruje maksymalnie przez m elementów dla każdej liczby i. Zatem całkowita złożoność obliczeniowa tego kodu to:

O(k*m)

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

3.1. Prosta implementacja

Przykład programu pierwszego (uproszczonego) wraz z wynikiem jego działania przedstawia się następująco:

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int Tab[100];
    int k = 0, 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 kończy się, gdy liczba jest >= 10
        }
        Tab[k++] = liczba; // Zapisujemy liczbe w tablicy
    }

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

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

    // Brute-force - porównujemy każda liczbe z każda inna
    bool znalazlem = false;
    for (int w = 0; w < k; w++) {
        for (int s = w + 1; s < k; s++) {
```

```
            // Porównujemy każdą liczbę z każdą inną, bez sprawdzania odległości m
            if (Tab[w] == Tab[s]) {
                cout << Tab[s] << " ";
                znalazlem = true;
            }
        }
    }

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

    cout << endl;
    return 0;
}
```

Rys. 3.1. Kod programu prostszego

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

Rys. 3.2. Wynik działania kodu pierwszego (prostszego) dla [2, 4, 8, 4, 5, 5, 9] M=1

Zauważmy, że kod pierwszy choć i znajduje pary powtarzające się, lecz nie uwzględnia odległość pomiędzy elementami.

Przykład programu drugiego (wersja wydajniejsza) wraz z wynikiem jego działania przedstawia się następująco:

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int Tab[100];
    int k = 0, 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 kończy się, gdy liczba jest >= 10
        }
        Tab[k++] = liczba; // Zapisujemy liczbę w tablicy
    }

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

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

    // Zmienna do śledzenia, które liczby zostały już wypisane
    bool wypisano[100] = {false}; // Tablica logiczna śledząca, czy liczba została już wypisana
    bool znalazlem = false; // Zmienna do śledzenia, czy znaleziono liczby spełniające warunki

    // Przechodzimy po tablicy i porównujemy elementy
    for (int i = 0; i < k; i++) {
        // Porównujemy liczby w odległości m od siebie
        for (int j = i + 1; j < k && j <= i + m; j++) {
            // Jeśli liczby są takie same i jeszcze nie zostały wypisane
            if (Tab[i] == Tab[j] && !wypisano[Tab[i]]) {
                cout << Tab[j] << " "; // Wypisujemy liczbę
                wypisano[Tab[i]] = true; // Oznaczamy, że ta liczba już została wypisana
                znalazlem = true;
            }
        }
    }

    if (!znalazlem) {
        cout << "Brak elementow spelniajacych kryteria" << endl;
    }
    cout << endl;
    return 0;
}
```

Rys. 3.3. Kod programu(wersja wydajniejsza)

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

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

W porównaniu do pierwszego kodu, który nie uwzględniał odległość m między powtarzającymi się liczbami, ta wersja kodu jest bardziej skomplikowana i działa w sposób bardziej szczegółowy.

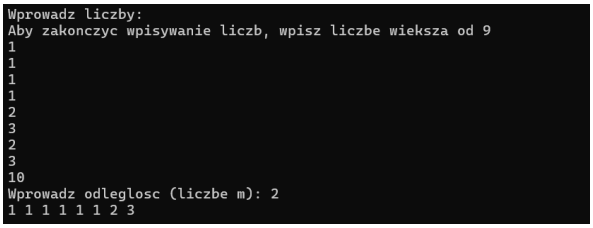
W poprzednim kodzie nie istniała tablica logiczna wypisano[], która ma na celu zapobieganie wielokrotnemu wypisywaniu tych samych liczb, jeśli występują w różnych miejscach tablicy. Program sprawdza, czy dana liczba była już wypisana i jeśli tak, to jej nie wypisywał ponownie.

3.2. Testy „niewygodnych” zestawów danych

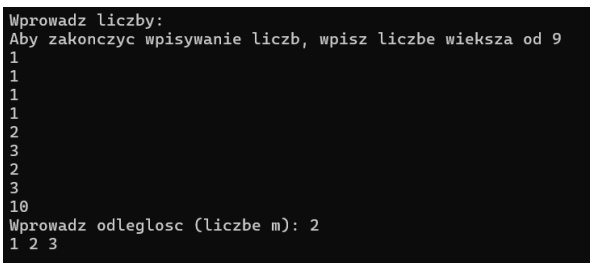
Sprawdzimy działanie kodu wersji pierwotnego oraz bardzej wydajniejszego dla następnego wejścia.

Po uruchomienu kodu i wprowadzaniu danych wejściowych, zauważmy, że piewsza prostsza wersja wypisuje nie tylko powtarzające się elementy, lecz i ich powtórzenia bez sprawdzenia, czy dany element już był wipisany. Natomiast drugi kod charakteryzuje się poprawnym działaniem. Wypisuje elementy powtarzające się na zadanej odległości oraz nie wypisuje powtórnie tych elementów, które już zostały wypisane.

Wejście: [1, 1, 1, 1, 2, 3, 2, 3] M=2

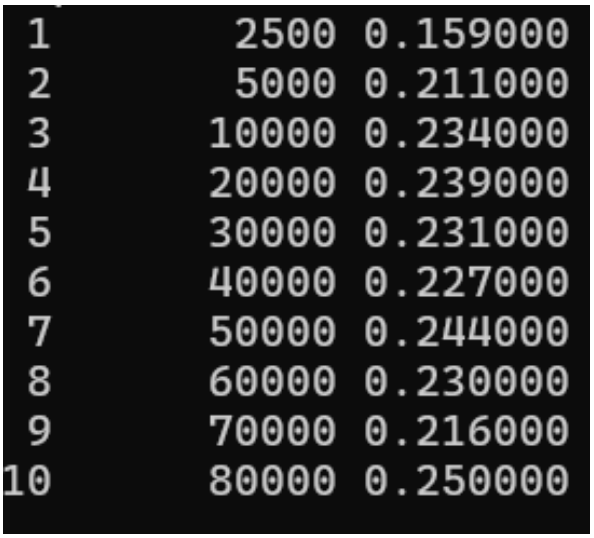


Rys. 3.5. Wynik działania kodu pierwszego dla [1, 1, 1, 1, 2, 3, 2, 3] M=2



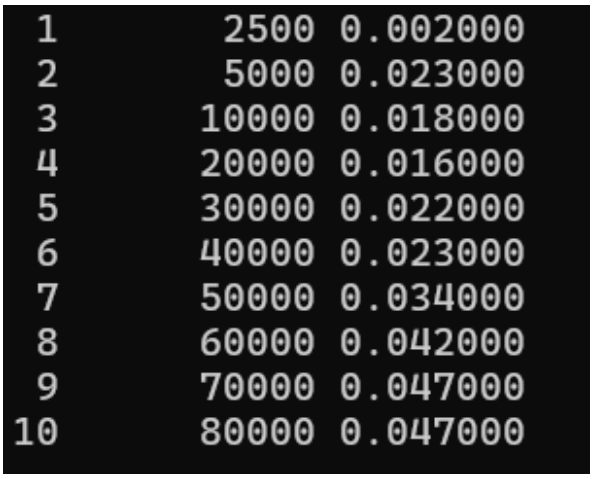
Rys. 3.6. Wynik działania kodu wydajniejszego dla [1, 1, 1, 1, 2, 3, 2, 3] M=2

3.3. Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej



1	2500	0.159000
2	5000	0.211000
3	10000	0.234000
4	20000	0.239000
5	30000	0.231000
6	40000	0.227000
7	50000	0.244000
8	60000	0.230000
9	70000	0.216000
10	80000	0.250000

Rys. 3.7. Wyniki testu czasowego dla kodu prostszego



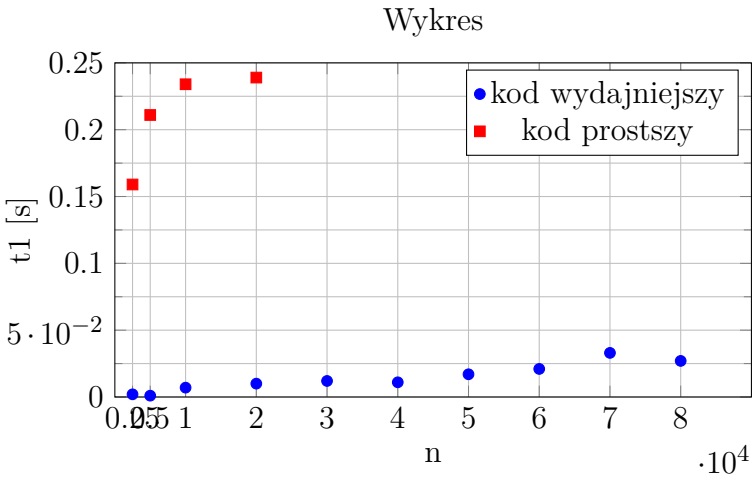
1	2500	0.002000
2	5000	0.023000
3	10000	0.018000
4	20000	0.016000
5	30000	0.022000
6	40000	0.023000
7	50000	0.034000
8	60000	0.042000
9	70000	0.047000
10	80000	0.047000

Rys. 3.8. 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.



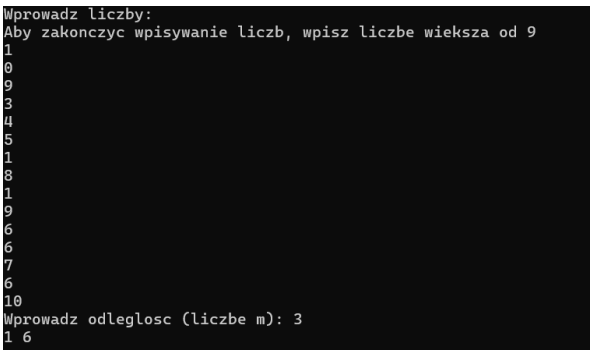
Rys. 3.9. Wykres czasu obliczeń dla kodu wydajniejszego oraz kodu prostszego

3.5. Testy

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

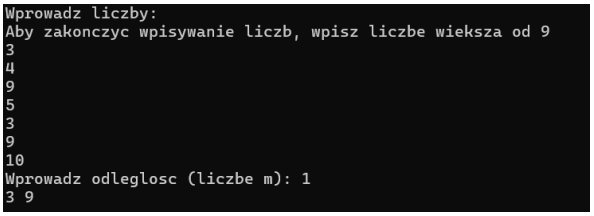
```
Wprowadz liczby:
Aby zakonczyc wpisywanie liczb, wpisz liczbe wieksza od 9
1
0
9
3
4
5
1
8
1
9
6
6
7
6
10
Wprowadz odleglosc (liczbe m): 3
1 1 9 1 6 6 6
```

Rys. 3.10. Wyniki testu pierwszego dla kodu prostszego

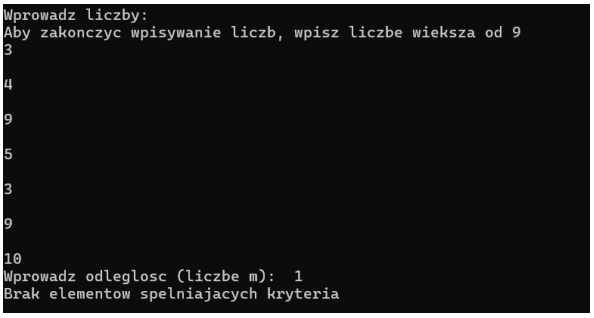


Rys. 3.11. Wyniki testu pierwszego dla kodu wydajniejszego

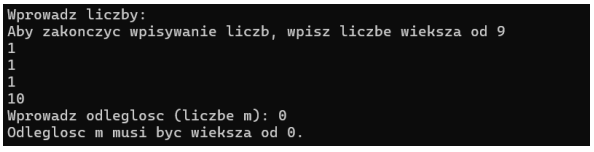
3. Wejście [3, 4, 9, 5, 3, 9] M=1



Rys. 3.12. Wyniki testu trzeciego dla kodu prostszego



Rys. 3.13. Wyniki testu trzeciego dla kodu wydajniejszego



Rys. 3.14. Wyniki testu gdzie m<1

Możemy zauważyć, że kod w prostszej wersji nie zawsze działa sprawnie. Dobrze sprawdza się, jeśli zestaw danych wejściowych składa się z elementów powtarzających się nie częściej niż jeden raz. Natomiast kod w wersji wydajniejszej zawsze działa sprawnie. Poprawność nie zależy od ilości powtórzeń lub długości M.