WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Barbara Gładysz

Algorytmy i struktury danych

SPRAWOZDANIE

Rzeszów, 2025

Spis treści

1. Wstęp

W sprawozdaniu proponuję dwa sposoby na poszukiwanie powielających się elementów w tablicy n-elementowej, gdzie wartości zawierają się w przedziale od 1 do n-1.

Mimo że jest to dość proste zagadnienie, pokażę, że odpowiednie podejście może znacznie ograniczyć złożoność czasową algorytmu.

2. Etapy rozwiązania problemu

2.1. Podejście pierwsze

2.1.1. Analiza problemu

Pierwsze rozwiązanie problemu będzie opierać się na metodycznym porównywaniu ze sobą elementów tablicy.

Aby to zrobić, utworzymy pomocniczą tablicę L typu bool, która będzie zawierała n elementów, czyli tyle, ile wartości może przyjmować tablica główna(od 1 do n-1) plus 1 (0 nie należy do rozpatrywanego przedziału, więc L[0] jest nam niepotrzebne). Moglibyśmy zredukować liczbę elementów o jeden i przypisywać wartości do indeksów o jeden mniejszych, jednak wpłynęłoby to na czytelność kodu, a nie kosztuje wiele zasobów.

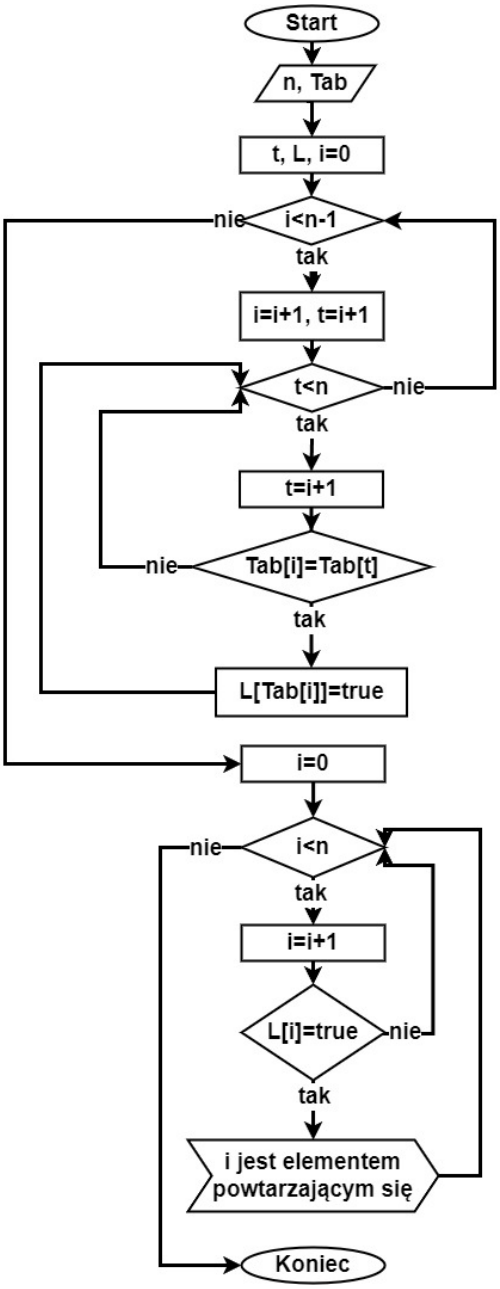
Każdemu elementowi w tablicy L przypisujemy wartość false.

Następnie sprawdzamy każdy element głównej tablicy z każdym elementem występującym po nim (kolejność porównywania elementów jest nieistotna; porównanie elementu 1 z 3 jest równoznaczne z porównaniem 3 z 1).

Jeżeli znajdziemy dwa elementy, które będą sobie równe, oznacza to, że jest to powielająca się wartość. W tablicy L na indeksie równym powtarzającemu się elementowi przypisujemy wartość true (co możemy zinterpretować jako "prawda, że dany element się powtarza").

Jest to bardziej efektywny sposób rejestrowania powielających się elementów, który opracowałam. Na początku mojej pracy nad algorytmem, znalezione powielające się wartości były zapisywane kolejno w tablicy L typu int. Jednak wymagało to zagnieżdżenia kolejnej (trzeciej) pętli for, aby sprawdzić, czy dana wartość nie znajdowała się już w tablicy pomocniczej, co znacząco obniżało wydajność funkcji. Bez tego, element pojawiający się trzykrotnie w tablicy głównej, w tablicy L znajdowałby się dwukrotnie.

2.1.2. Schemat blokowy algorytmu



Zauważmy, że w algorytmie pojawiły się dwie dodatkowe zmienne pomocnicze typu int: i oraz t. Są to wskaźniki ułatwiające poruszanie się w tablicy. Zmienną i wykorzystujemy ponownie podczas przeszukiwania tablicy pomocniczej L.

2.1.3. Algorytm zapisany w pseudokodzie

Pseudokod opisujący algorytm Brute Force:

Listing 2.1. Pseudokod funkcji BruteForce

```
1. input: Tab  // tablica przechowujaca wartosci ciagu
2.        n    // dlugosc tablicy
4.
5. L := new bool[n]
6. for i := 0 to n-1 do
7.     L[i] := false
9.
10. for i := 0 to n-2 do
11.     for t := i+1 to n-1 do
12.         if Tab[i] = Tab[t] then
13.             L[Tab[i]] := true
14.         endif
15.     endfor
16. endfor
17.
18. print "elementy powtarzajace sie: "
19. for i := 0 to n-1 do
20.     if L[i] = true then
21.         print i, " "
22.     endif
23. endfor
24.
25. delete[] L
```

2.1.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe" rozwiązanie problemu

Proces ten przeprowadzimy na przykładzie danych wejściowych: [2,4,2,3,4]. Jest to dość mały zbiór danych, lecz pozwoli nam przeanalizować kolejne kroki algorytmu.

Tabela 2.1. Pierwsze przejście pętli

i	j	Tab[i]	Tab[j]	L[Tab[i]]
0	1	2	4	false
0	2	2	2	true
0	3	2	3	false
0	4	2	4	false

Tabela 2.2. Drugie przejście pętli

i	t	Tab[i]	Tab[t]	L[Tab[i]]
1	2	4	2	false
1	3	4	3	false
1	4	4	4	true

Tabela 2.3. Trzecie przejście pętli

i	t	Tab[i]	Tab[t]	L[Tab[i]]
2	3	2	3	false
2	4	2	4	false

Tabela 2.4. Czwarte przejście pętli

i	t	Tab[i]	Tab[t]	L[Tab[i]]
2	3	2	3	false
2	4	2	4	false

Tabela 2.5. Piąte przejście pętli

i	t	Tab[i]	Tab[t]	L[Tab[i]]
3	4	3	4	false

Ostatecznie uzyskujemy dwie wartości true - dla L[2] oraz L[4]

2.1.5. Teoretyczne oszacowanie złożoności obliczeniowej

Inicjalizacja tablicy L - n-1 iteracji, co daje złożoność czasową:

$$O(n)$$

Dwie, zagnieżdżone funkcje for, porównujące elementy tablicy:

Zewnętrzna pętla wykonuje n-1 iteracji.

Wewnętrzna pętla for(int t=i+1; t<n; t++) wykonuje maksymalnie n-i-1 iteracji dla każdej iteracji zewnętrznej pętli. Całkowita liczba iteracji wewnętrznej pętli to:

$$\sum_{i=0}^{n-2}(n-i-1) = (n-1)+(n-2)+...+1 = \frac{n(n-1)}{2}$$

Stąd wiemy, że złożoność czasowa tej części wynosi:

$$O(n^2)$$

Sumaryczna złożoność czasowa algorytmu BruteForce to suma złożoności poszczególnych części:

$$O(n) + O(n^2) + O(n) = O(n^2)$$

3. Rozwiązanie - próba druga

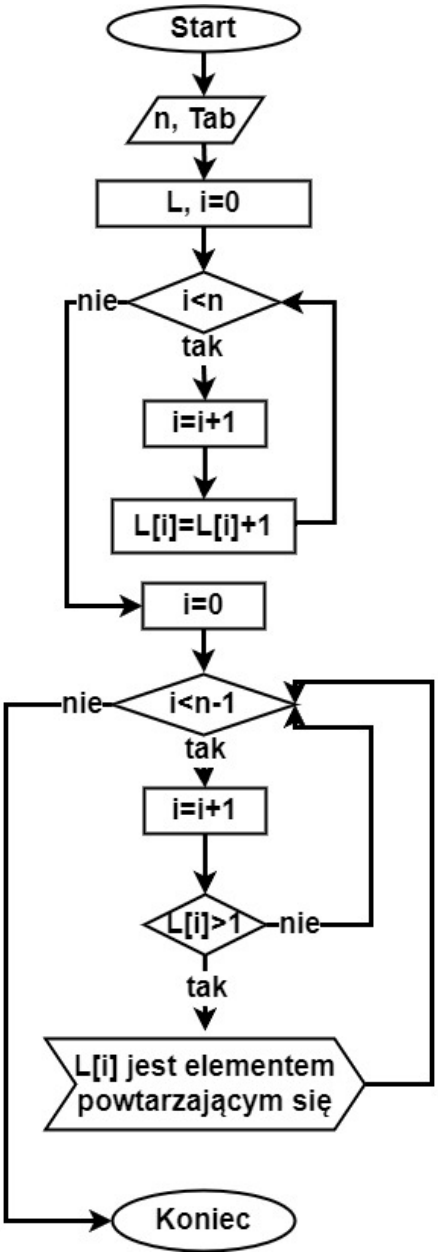
3.1. Ponowna analiza problemu

Drugi algorytm wykorzystuje fakt, że znamy zakres w którym losowane są liczby. Dzięki temu możemy przejść przez każdy element tablicy głównej i w tablicy pomocniczej L zapisać liczbę wystąpień każdej wartości. Następnie wystarczy przeszukać tablicę L oraz znaleźć elementy, które występują co najmniej dwa razy.

W ten sposób nasz algorytm zostanie znacząco uproszczony logicznie oraz stanie się zdecydowanie bardziej wydajny czasowo w porównaniu do poprzedniego rozwiązania.

Tablica L będzie podobna do tablicy L z funkcji BruteForce. Również będzie miała n-1 elementów, a jej indeksy będą odpowiadać wartościom, które mogą pojawiać się w tablicy głównej. W funkcji zoptymalizowanej tablica L będzie jednak typu int.

3.1.1. Schemat blokowy algorytmu



3.1.2. Algorytm zapisany w pseudokodzie

Pseudokod opisujący algorytm zoptymalizowany:

Listing 3.1. Pseudokod

```
1. input: Tab  // tablica przechowujaca wartosci ciagu
2.        n    // dlugosc tablicy
3.
4. L := new int[n]
5. for i := 0 to n-1 do
6.     L[i] := 0
7. endfor
8.
9. for i := 0 to n-1 do
10.     L[Tab[i]] := L[Tab[i]] + 1
11. endfor
12.
13. print "elementy powtarzajace sie: "
14. for i := 0 to n-1 do
15.     if L[i] > 1 then
16.         print i, " "
```

3.1.3. Sprawdzenie poprawności algorytmu poprzez „ołówkowe" rozwiązanie problemu

Podobnie jak w algorytmie Brute Force użyjemy danych wejściowych: [2,4,2,3,4].

Tabela 3.1. Przejście pętli zoptymalizowanego algorytmu

i	Tab[i]	L[Tab[i]]
0	2	L[2]=1
1	4	L[4]=1
2	2	L[2]=2
3	3	L[3]=1
4	4	L[4]=2

Ostatecznie uzyskujemy w tablicy L dwie wartości większe od 1 - L[2] oraz L[4]. Jest to zgodne z poprzenimi wynikami.

3.1.4. Teoretyczne oszacowanie złożoności obliczeniowej

Każda z trzech głównych części algorytmu wykonuje się w czasie O(n), więc całkowita złożoność czasowa algorytmu jest sumą tych trzech części:

$$O(n) + O(n) + O(n) = O(n)$$

4. Implementacja oraz eksperymentalne potwierdzenie wydajności algorytmów

4.1. Kod programu

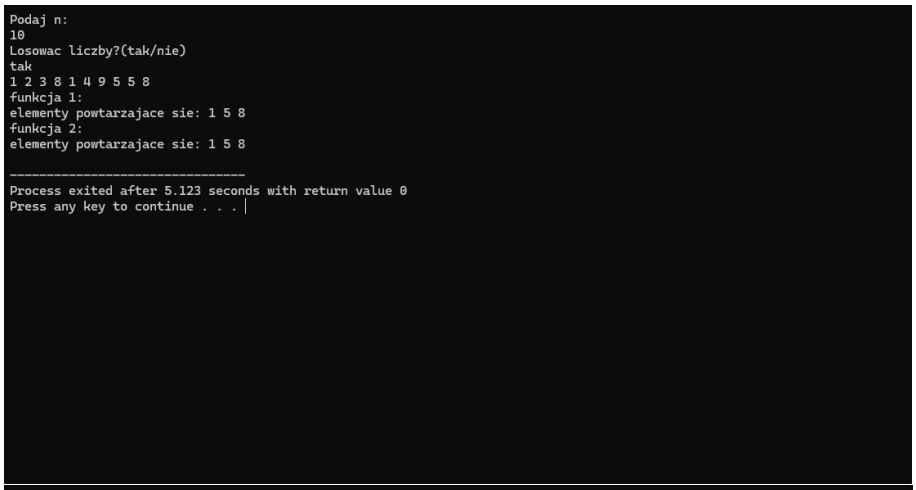
```
#include <iostream>
using namespace std;
#include <ctime>
#include <iomanip>
#include <limits>
#include <chrono>
using namespace std::chrono;
int *Tab;

void GetRandomNumbers(int n)
{//losuje elementy do tablicy
   srand(time(NULL));
   for(int i=0; i<n;i++){
      Tab[i]=rand()%(n-1)+1;//losuje liczby od 1 do n-2(rand()%(n) zaczyna
    losowanie od 0, stad powiekszenie wylosowanej liczby o 1)
      cout<<Tab[i]<<" ";//wypisuje wylosowane elementy
   }
   cout<<endl;
}

void GetNumbersFromUser(int n)
{//elementy do tablicy przypisuje uzytkownik
   int number;
   for(int i=0; i<n; i++)
   {
      cout<<"Podaj odpowiednia liczbe(od 1 do "<<n-1<<")"<<endl;
      cin>>number;
      while(number>=n||number<1){//jezeli zle podano liczbe, uzytkownik
    musi podac ja ponownie
         cout<<"zle podano liczbe, podaj jeszcze raz"<<endl;
```

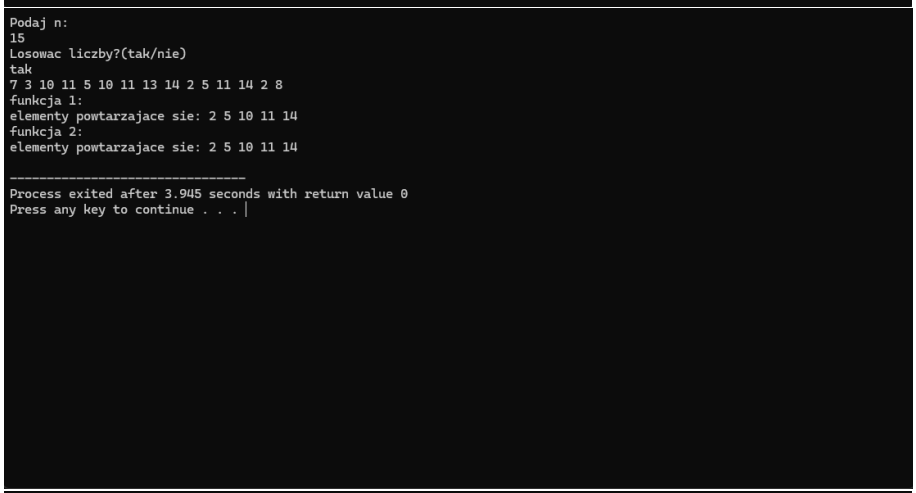
```
        cin>>number;
      }
      Tab[i]=number;
   }
}

void BruteForce(int n)
{//pierwszy algorytm
   bool* L = new bool[n];//tablica zawierajaca wartosc na indeksie n rowna
    true, jesli liczba n sie powtarza
   for(int i=0; i<n; i++)
   {
      L[i]=false;//poczatkowo wszystkie elementy tablicy L sa rowne false
   }

   for(int i=0; i<n-1; i++)//sprawdzam kazdy element z tablicy Tab(nie
    sprawdzam ostatniego elementu przez i, jest ono sprawdzane przez t.
    Jezeli i=n, j=n+1, co wychodzi poza wartosc tablicy)
   {
      for(int j=i+1; j<n; j++)//z kazdym elementem ktory wystepuje dalej w
    tablicy(np. element 1 z 2,3 i 4, nie ma potrzeby sprawdzania 1 z 1, czy
    2 z 1(jest tym samym co 1 z 2))
      {
         if(Tab[i]==Tab[j])// jezeli elementy sa sobie rowne
         {
            L[Tab[i]]=true;//zapisuje wartosc true na pozycji rownej
    wartosci sprawdzonego elementu
         }
      }
   }
   cout<<"elementy powtarzajace sie: ";
   for(int i=0; i<n; i++)
   {
      if(L[i]==true) cout<<i<<" ";
   }
```

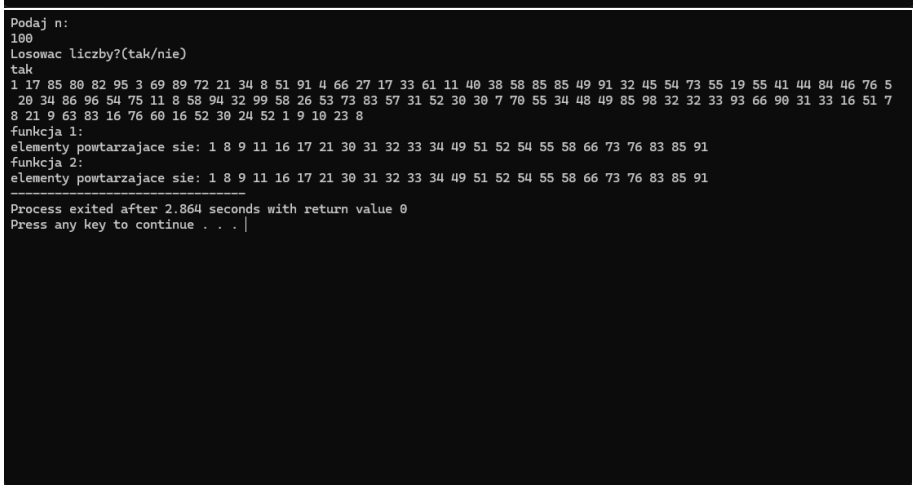
```
   cout<<endl;
   delete [] L;
}

void OptimizedFunction(int n){//drugi algorytm
   int* L = new int[n];//tablica L ma tyle elementow, ile roznych wartosci
    moga przyjmowac elementy tablicy Tab(od 0 do n-1)
   for(int i=0; i<n; i++) L[i]=0;//poczatkowo, kazdy element L jest rowny 0
   //L[0] zawsze bedzie rowne 0, poniewaz wartosci Tab zaczynaja sie od 1
    (mozna wartosci przypisywac do indeksu o jeden mniejszy od wartosci, ale
    przez to kod bylby mniej przejrzysty)



   for(int i=0; i<n; i++)
   {
      L[Tab[i]]++;//indeksy tablicy L odpowiadaja wartosciom jakie moga
    przyjmowac elementy Tab. Sprawdzam jakie elementy pojawiaja sie w Tab i
    zapisuje ich czestosc wystepywania w tablicy L
   }

   cout<<"elementy powtarzajace sie: ";
   for(int i=0; i<n; i++)
   {
      if(L[i]>1)  cout<<i<<" ";//jezeli wartosc w L jest wieksza od jeden,
    to znaczy ze wartosc w Tab rowna indeksowi pojawiala sie wielokrotnie
   }
   cout<<endl;
   for(int i=0; i<n;i++) cout<<L[i];
   delete [] L;
}

int main(){
   string answer;
   int n=0;
   cout<<"Podaj n: "<<endl;
```

```
    cin>>n;
    Tab = new int [n];
    cout<<"Losowac liczby?(tak/nie)"<<endl;
    cin>>answer;
    if(answer=="tak") GetRandomNumbers(n);
    else GetNumbersFromUser(n);

    cout<<"funkcja 1:"<<endl;
    BruteForce(n);
    cout<<"funkcja 2: "<<endl;
    OptimizedFunction(n);

    delete [] Tab;
    return 0;
}
```

4.2. Testy na małych zbiorach danych







4.3. Testy na dużych zbiorach danych(wydajnościowe)

W tabelach jako t1 oznaczono czas wykonywania funkcji BruteForce, natomiast jako t2 funkcję zoptymalizowaną

Tabela 4.1. test wydajnościowy obu fukncji

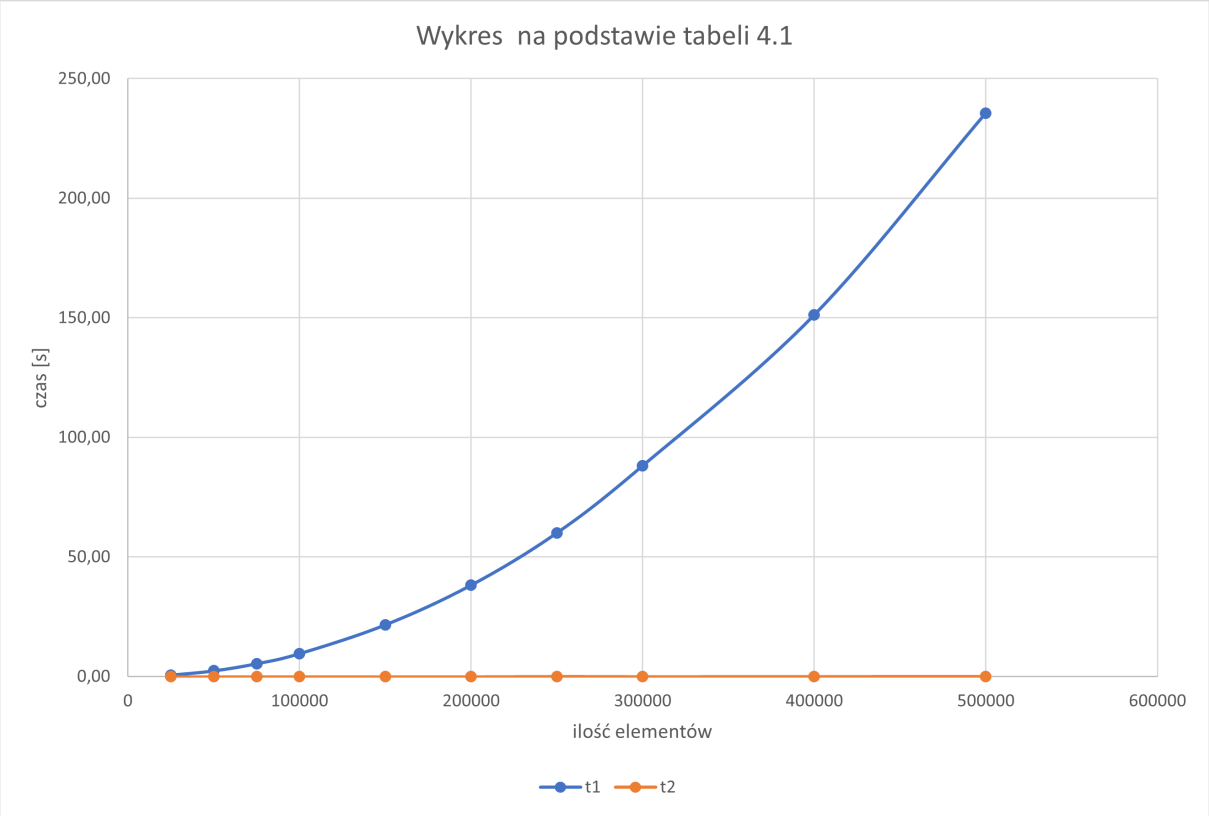
lp	rozmiar zbioru n	t1[s]	t2[s]
1	25000	0.596831	0.001537
2	50000	2.36766	0.002009
3	75000	5.35681	0.001311
4	100000	9.5698	0.001633
5	150000	21.5857	0.002166
6	200000	38.1887	0.002169
9	250000	60.0333	0.006125
7	300000	88.1083	0.004019
8	400000	151.148	0.00499
10	500000	235.522	0.006943

Ponieważ wydajność funkcji zoptymalizowanej jest zdecydowanie większa niż funkcji BruteForce, przeprowadzimy dla niej osobny test wydajnościowy, na 100-krotnie większej ilości elementów.

Tabela 4.2. test wydajnościowy funkcji zoptymalizowanej

lp	rozmiar zbioru n	t2s]
1	2500000	0.024325
2	5000000	0.04515
3	7500000	0.070451
4	10000000	0.095735
5	15000000	0.147471
6	20000000	0.196643
9	25000000	0.242369
7	30000000	0.28158
8	40000000	0.402963
10	50000000	0.48233

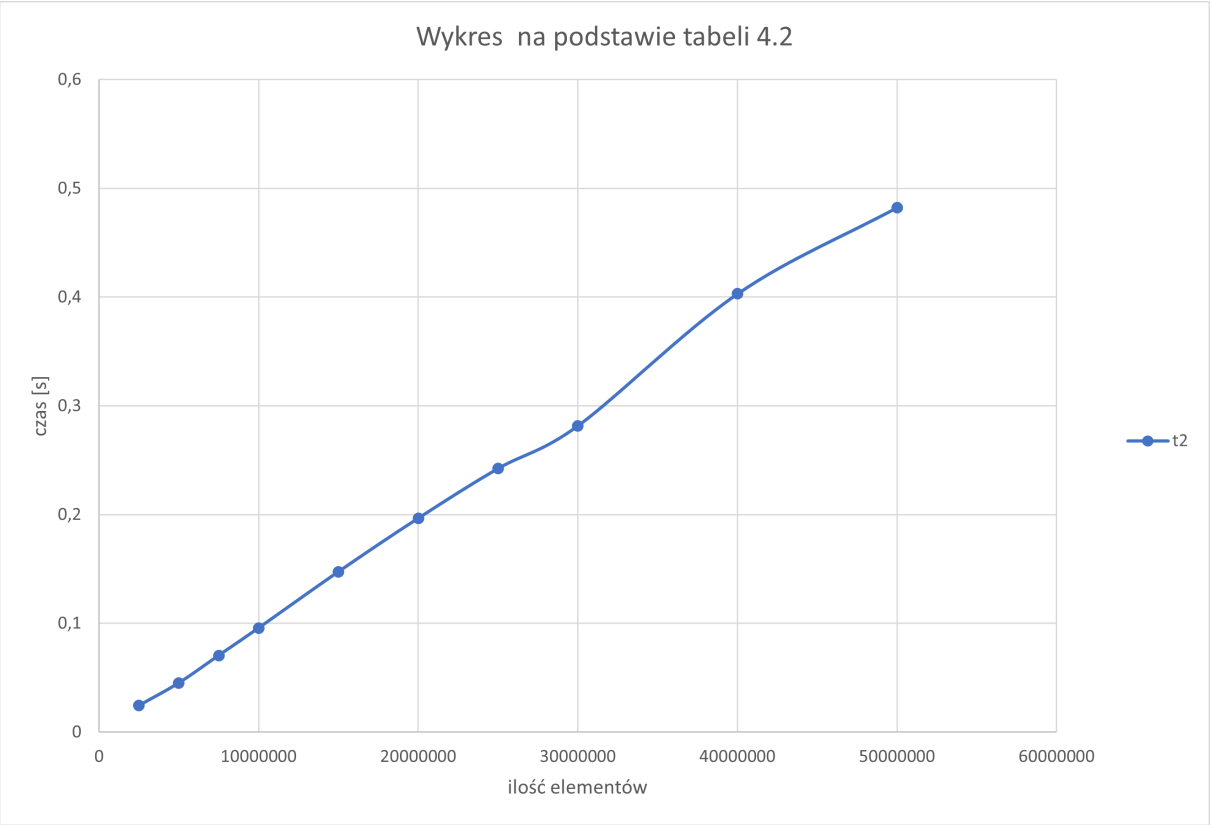
4.4. Wykresy wydajności algorytmów



Na wykresie wyraźnie widzimy, jak różne wydajnościowo są oba algorytmy. Możemy również zauważyć, że nasze obliczenia złożoności czasowej algorytmu BruteForce są poprawne, ponieważ łącząc uzyskane punkty, otrzymujemy kształt paraboli.

Nie jesteśmy jednak w stanie wyciągnąć wielu wniosków o na temat funkcji zoptymalizowanej. Ilości danych, które funkcji BruteForce zajmowały ponad trzy minuty, nie zajęły jej nawet sekundy. Jest więc zdeycowanie mniej czasochłonna, lecz żeby zaobserwować jej złożoność czasową, musimy przeprowadzić więcej testów.

W związku z tym narysujemy nowy wykres, korzystając z danych z tabeli 4.2.



Dzięki zwiększeniu ilości elementów 100-krotnie, możemy wyraźnie zauważyć kształt wykresu, który również zgadza się z naszymi obliczeniami.

5. Zakończenie

5.1. Podsumowanie

W sprawozdaniu rozpatrzyliśmy dwa możliwe sposoby na odszukanie elementów powielających się w tablicy o n elementach z zakresu (1, n-1). Pierwszy z nich zawierał dwie funkcje for, wewnętrzną i zewnętrzną, co znacznie zmniejszało jego wydajność czasową. Drugi w tablicy pomocniczej zapisywał informacje o ilości występowania wartości, co znacznie odciążyło algorytm. Z $O(n^2)$ uzyskaliśmy $O(n)$, co później potwierdziliśmy testując algorytmy.

5.2. Wnioski

Nawet w przypadku dość prostych algorytmów warto poświęcić czas na odszukanie jak najwydajniejszego rozwiązania, ponieważ nawet jeżeli różnice w wydajności na małych zbiorach były niewielkie, to im bardziej zwiększymy ilość rozpatrywanych elementów, tym bardziej je odczuwamy.