WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Kacper Łuszcz

Wyszukiwanie trójelementowych kombinacji o zadanej sumie z zadanego ciągu liczbowego

Sprawozdanie z projektu programistycznego

Rzeszów, 2025

Spis treści

1. Treść zadania

Znajdź liczbę trójelementowych kombinacji liczb z zadanego ciągu, których suma jest równa zadanej liczbie M.

Przykład:

Wejście

we = [1,2,5,1,2,1,2,4]

M=6

Wyjście

Liczba kombinacji wynosi 2: [2 2 2], [1 1 4]

2. Rozwiązanie - podejście pierwsze

2.1. Analiza problemu

Celem niniejszego zadania jest znalezienie trójelementowych kombinacji liczb z zadanego ciągu, których suma jest równa zadanej liczbie M. Pierwszym rozwiązaniem, które nasuwa się samo, jest zastosowanie metody brute force. Na początku sprawdzimy, czy długość ciągu wynosi przynajmniej 3, ponieważ poszukiwane kombinacje muszą mieć długość 3. Algorytm nie wykona się, jeżeli tablica będzie zbyt krótka. Następnie, za pomocą trzech zagnieżdżonych pętli, sprawdzimy wszystkie możliwe trójelementowe kombinacje liczb w ciągu. Kolejno, dla każdej kombinacji, sprawdzimy, czy jej suma jest równa zadanej liczbie M. Jeżeli ten warunek zostanie spełniony, dodajemy tę kombinacje do tablicy wynikowej przechowującej znalezione trójki.

W tym miejscu pojawia się jednak problem. Co w przypadku, gdy dwukrotnie znajdziemy tę samą kombinację liczb? Czy w takim przypadku powinniśmy dodać ją do tablicy wynikowej? Rozważono dwie sytuacje. Jedną z nich jest sprawdzanie unikalności kombinacji. Załóżymy, że takie same kombinacje (na przykład [1, 1, 4] oraz [1, 1, 4]) będą traktowane jako jedna kombinacja. W związku z tym dana trójka liczb zostanie dodana do tablicy wynikowej tylko raz. Podejście ze sprawdzaniem unikalności może znacznie obniżyć efektywność algorytmu, jednak sprawi, że wyniki będą bardziej wiarygodne oraz przejrzyste. Drugą rozważaną sytuacją będzie brak sprawdzania unikalności trójek.

W dwóch przedstawionych sytuacjach, trójki składające się z tych samych liczb, ale w różnej kolejności, będą traktowane jako różne kombinacje (na przykład [1, 1, 4] oraz [4, 1, 1]). Aby traktować takie trójki jako identyczne, konieczne byłoby wprowadzenie sortowania. Jednak w przypadku tego zadania zakładamy, że algorytmy sortujące nie są dozwolone.

1. Dane wejściowe

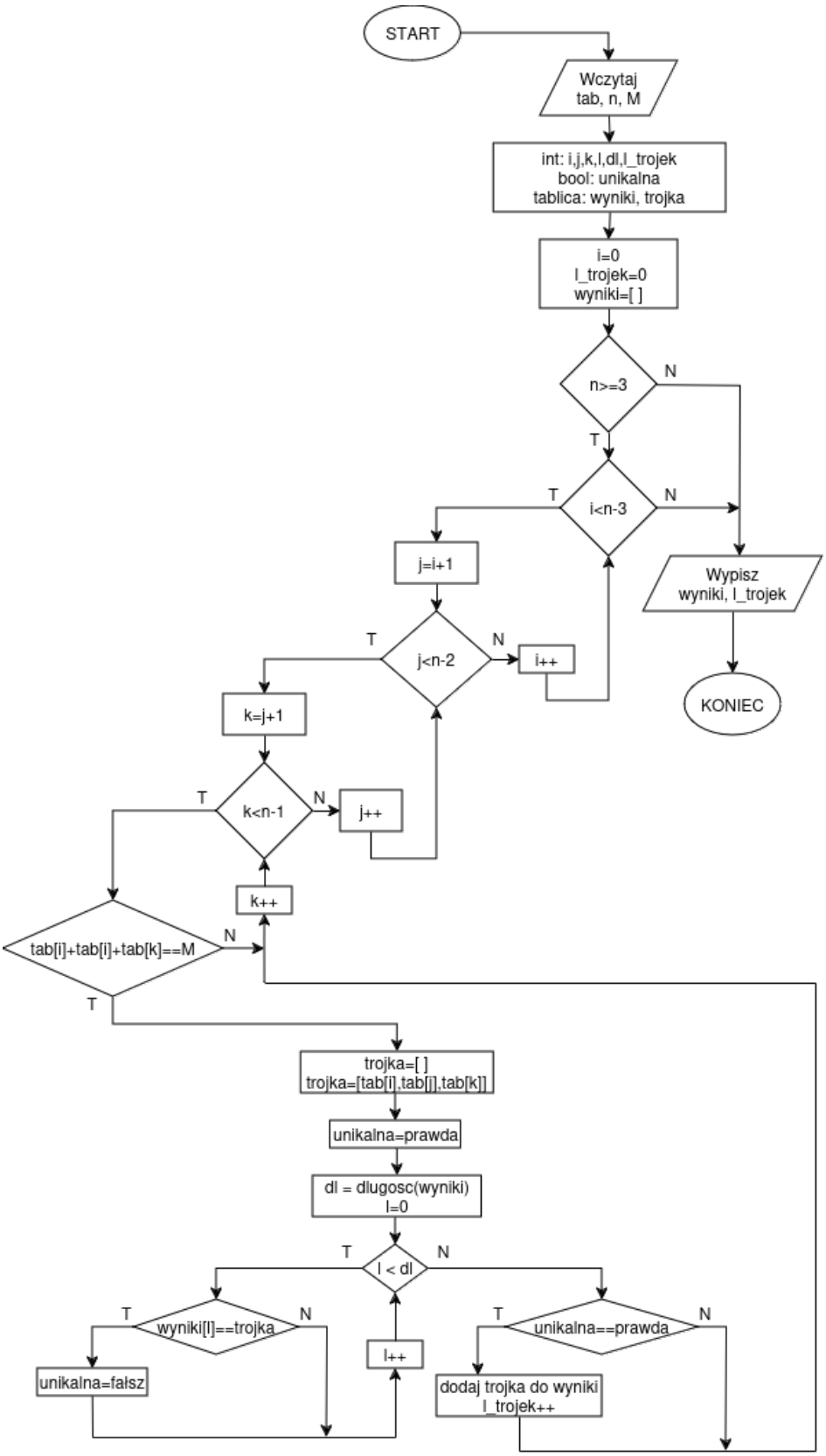
Danyme wejściowe algorytmu to:

- struktura danych typu tablica (`tab`), przechowująca wartości zadanego ciągu.
- długość tego ciągu (`n`).
- Zadana suma (`M`).

2. Dane wyjściowe

Algorytm wyświetli znalezione kombinację oraz ich liczbę.

2.2. Schemat blokowy algorytmu



Rysunek 2.1: Schemat blokowy algorytmu w wersji naiwnej

2.3. Algorytm zapisany w pseudokodzie

```
wejście:
tab // tablica początkowa
n // długość tablicy
M // podana suma szukanych trójek

wyjście: wyświetlone trójki z tablicy wynikowej oraz liczba trójek

wyniki = pusty zbiór
l_trójek = 0

jeżeli n >= 3:
    dla i = 0 do n-3 wykonuj
        dla j = i+1 do n-2 wykonuj
            dla k = j+1 do n-1 wykonuj
                jeżeli (tab[i] + tab[j] + tab[k] == M):
                    trójka = {tab[i], tab[j], tab[k]}
                    jeżeli trójka nie znajduje się w wyniki
                        dodaj trójka do wyniki
                        zwieksz l_trójek o 1

dla każdej trójki w wyniki:
    wypisz trójkę
wypisz liczba_trójek
```

2.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe” rozwiązanie problemu

Sprawdzenie poprawności algorytmu dla danych wejściowych `tab=[3,1,4,2,5]` oraz `M=7` przedstawiono w tablei poniżej. Możemy zaobserwować, że algorytm poprawnie znalazł trójkę `[1 4 2]`. Suma elementów tej trójki wynosi `M=7`.

i	j	k	tab[i]	tab[j]	tab[k]	tab[i]+tab[j]+tab[k]	suma==M	unikalna
0	1	2	3	1	4	8	Fałsz	Prawda
0	1	3	3	1	2	6	Fałsz	Prawda
0	1	4	3	1	5	9	Fałsz	Prawda
0	2	3	3	4	2	9	Fałsz	Prawda
0	2	4	3	4	5	12	Fałsz	Prawda
0	3	4	3	2	5	10	Fałsz	Prawda
1	2	3	1	4	2	7	Prawda	Prawda
1	2	4	1	4	5	10	Fałsz	Prawda
1	3	4	1	2	5	8	Fałsz	Prawda
2	3	4	4	2	5	11	Fałsz	Prawda

Tabela 2.1: Tabela przedstawiająca sprawdzenie wersji 1 algorytmu

2.5. Teoretyczne oszacowanie złożoności obliczeniowej

Złożoność obliczeniowa mojej wersji algorytmu, czyli ze sprawdzaniem unikalności danej trójelementowej kombinacji jest bardzo duża. Mamy 3 zagnieżdżone pętle:

- Zewnętrzna: i od 0 do $n-3$,
- Środkowa: j od $i+1$ do $n-2$,
- Wewnętrzna: k od $j+1$ do $n-1$.

W najgorszym wypadku mamy do wykonania $O(n^3)$ iteracji, zatem złożoność obliczeniowa samego wyszukiwania trójek wynosi $O(n^3)$. Implementacja sprawdzania czy trójka jest unikalna jeszcze bardziej podnosi złożoność. Musimy przejrzeć wszystkie dotychczasowe trójki w `wyniki`. Załóżmy, że maksymalnie l trójek już istnieje w `wyniki`, gdzie l jest proporcjonalne do $O(n^3)$, czyli liczby możliwych kombinacji trzech elementów. Podsumowując, algorytm przetwarza $O(n^3)$ trójek, a dla każdej z nich wykonuje $O(l)$, czyli $O(n^3)$, sprawdzeń unikalności. Całkowita złożoność wynosi $O(n^3 \cdot n^3) = O(n^6)$

Podsumowując:

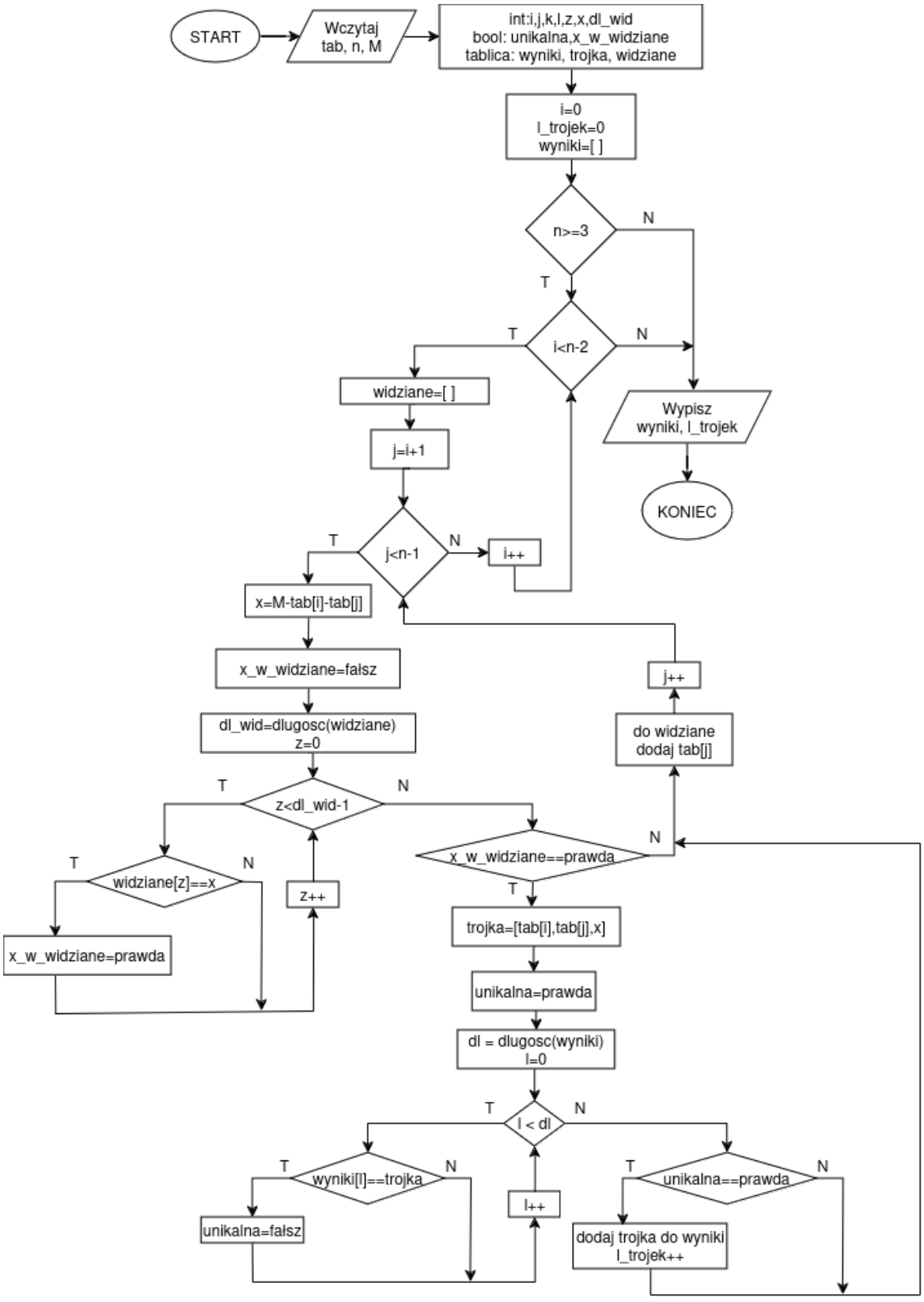
- Bez sprawdzenia unikalności $O(n^3)$
- Z iteracyjnym sprawdzaniem unikalności $O(n^6)$

3. Rozwiązanie - podejście drugie

3.1. Ponowna analiza problemu

Lepszym rozwiązaniem problemu będzie następujące podejście. Będziemy potrzebować pomocniczej tablicy, w której będziemy przechowywać widziane liczby z ciągu. Użyjemy dwóch zagnieżdżonych pętli. Dla każdej znalezionej pary obliczymy, jakiej wartości brakuje do zadaniej liczby M. Załóżmy, że brakującą liczbą będzie `x`. Wtedy dla każdej pary `i` oraz `j`: `x=M-tab[i]-tab[j]`. Następnie przeszukujemy tablicę z widzianymi liczbami. Jeżeli znajdziemy w niej x, nasza trójka będzie składać się z wartości: `[tab[i], tab[j], x]`

3.2. Schemat blokowy algorytmu



Rysunek 3.2: Schemat blokowy algorytmu w wersji nieco bardziej wydajnej

3.3. Algorytm zapisany w pseudokodzie

```
wejście:
tab // tablica początkowa
n // długość tablicy
M // podana suma szukanych trójek

wyjście: wyświetlone trójki z tablicy wynikowej oraz liczba trójek

wyniki = pusty zbiór
l_trójek = 0

jeżeli n>=3
    dla i=0 do n-2 wykonuj
    widziane = pusty zbiór
     dla j=i+1 do n-1
        x = M - tab[i] - tab[j]
      jeżeli x znajduję się w widziane
                trójka = {tab[i], tab[j], do_znalezienia}
                jeżeli trojka nie znajduje się w wyniki
                    dodaj trójka do wyniki
                    zwiększ l_trójek o 1
     do widziane dodaj tab[j]

dla każdej trójki w wyniki
    wypisz trójkę
wypisz l_trójek
```

3.4. Sprawdzenie poprawności algorytmu poprzez „ołówkowe" rozwiązanie problemu

Sprawdzenie poprawności algorytmu dla danych wejściowych `tab=[3,1,4,2,5]` oraz `M=7` przedstawiono w tablei poniżej. Możemy zaobserwować, że algorytm poprawnie znalazł trójkę `[1 4 2]`. Suma elementów tej trójki wynosi `M=7`.

i	j	tab[i]	tab[j]	x	widziane	x w widziane	unikalna
0	1	3	1	3	{}	fałsz	prawda
0	2	3	4	0	{1}	fałsz	prawda
0	3	3	2	2	{1, 4}	fałsz	prawda
0	4	3	5	-1	{1, 4, 2}	fałsz	prawda
1	2	1	4	2	{}	fałsz	prawda
1	3	1	2	4	{4}	prawda	prawda
1	4	1	5	1	{4, 2}	fałsz	prawda
2	3	4	2	1	{}	fałsz	prawda
2	4	4	5	-2	{2}	fałsz	prawda
3	4	2	5	0	{}	fałsz	prawda

Tabela 3.2: Tabela przedstawiająca sprawdzenie wersji 2 algorytmu

3.5. Teoretyczne oszacowanie złożoności obliczeniowej

W przypadku tej wersji algorytmu w najgorszym wypadku złożoność obliczeniowa teoretycznie nie będzie znacznie się różnić od złożoności obliczeniowej wersji 1, również będzie wynosić $O(n^6)$. Jednak podczas implementacji tego algorytmu możemy znacznie zoptymalizować przeszukiwanie tablicy `widziane`, dzięki czemu wyszukiwanie trójki będzie mieć złożoność $O(n^2)$, sprawdzanie unikalności równierz $O(n^2)$, co łącznie daje nam złożoność $O(n^4)$. Podsumowując:

- Bez sprawdzenia unikalności $O(n^2)$
- Z iteracyjnym sprawdzaniem unikalności $O(n^4)$

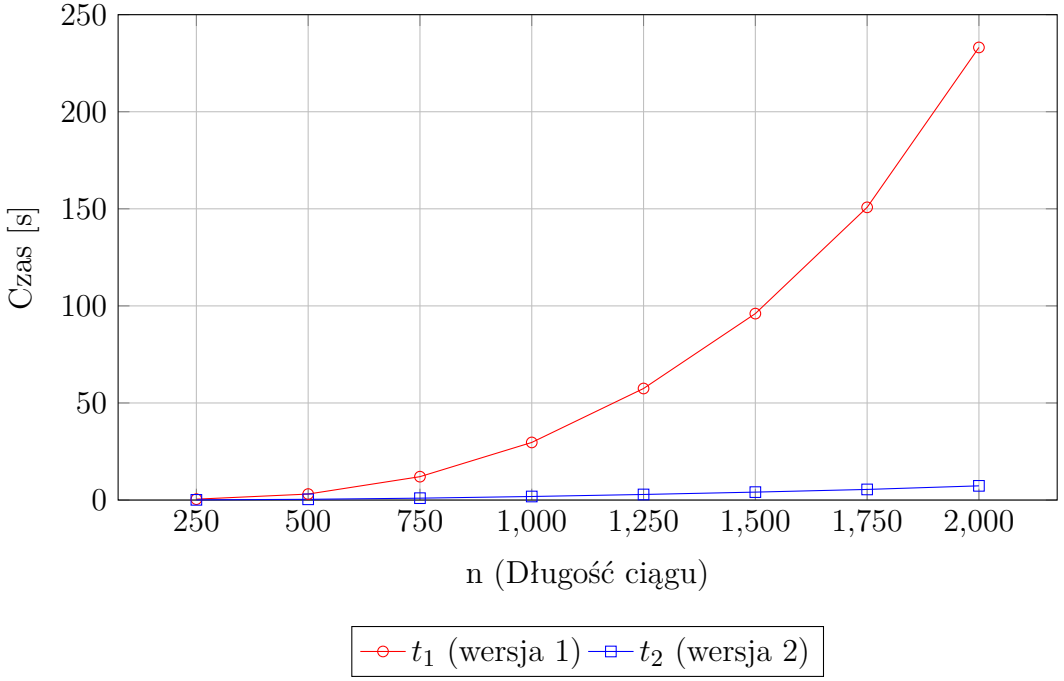
4. Implementacja algorytmów w języku programowania oraz testy wydajności

4.1. Szczegóły implementacji

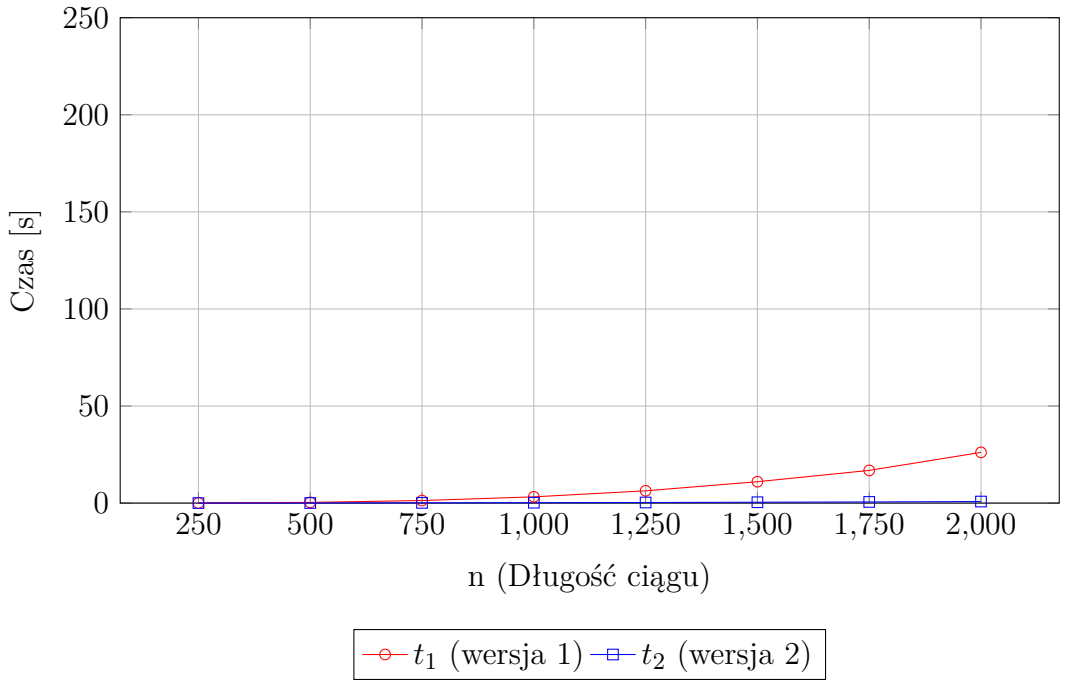
Program został napisiany w środowisku Code::Blocks IDE w języku C++. Kod źródłowy został przedstawiony w załączniku A. Logika całego programu została podzielona na funkcje: `wyswietl_wyniki_i_zapisz` - funkcja odpowiedzialna za wyświetlenie wyników z wersji 1 oraz wersji 2 algorytmu. Wyniki są wyświetlane w konsoli po czym program zapisuje je do pliku tekstowego. Następnie mamy 2 funkcje: `wersja_1` oraz `wersja_2`. Są to funkcje z implementacjami dwóch różnych wersji algorytmu. Kolejną funkcją jest funkcja `wczytaj_z_pliku`, ta funkcja jest odpowiedzialna za wczytywnaie danych wejściowych z pliku oraz za wywołanie funkcji: `wersja_1` i `wersja_2` z danymi z pliku wejsciowego. Program zawiera 2 funkcje testujące, które weryfikują poprawnośc działania wersji 1 oraz wersji 2 algorytmów. Dodatkowo zaimplementowałem mierzenie czasu wykonania danych wersji algorytmu, oraz funkcję która generuje tabelkę porównującą czasy wykoniania algorytmów dla dużych wielkości tablicy początkowej.

4.2. Wykresy porównujące czasy wykonania algorytmów dla poszczególnych wersji

Porównanie czasów wykonania wersji 1 i 2 z uwzględnieniem sprawdzania unikalności

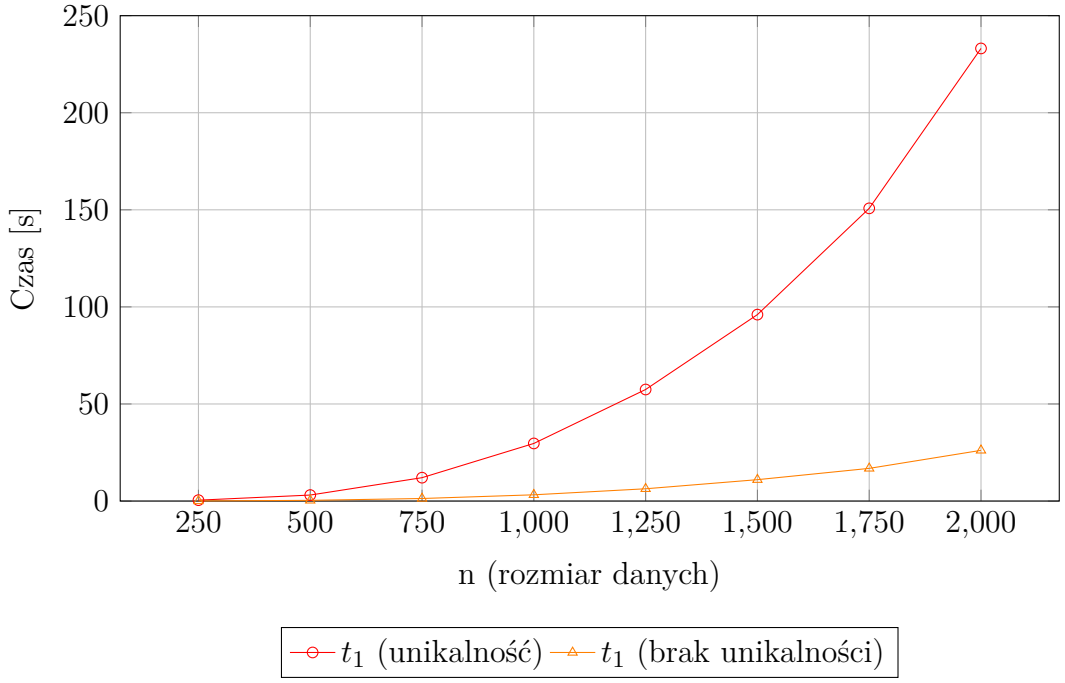


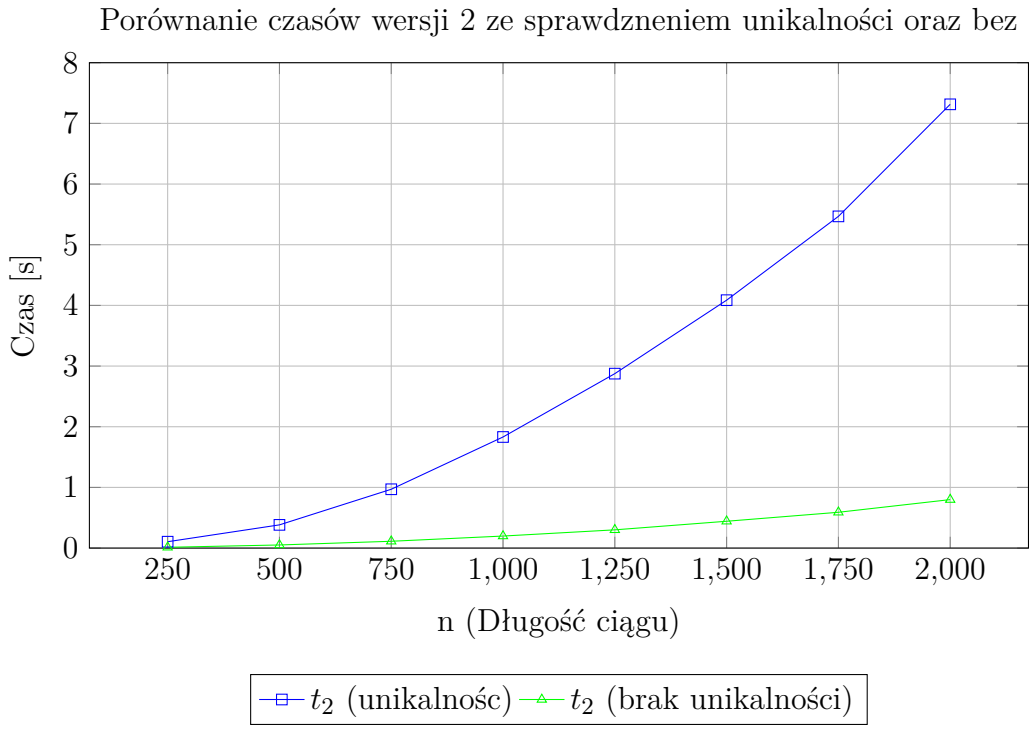
Porównanie czasów wykonania wersji 1 i 2 bez sprawdzania unikalności



Powyższe wykresy dobrze obrazują jak bardzo zmienia się czas wykonania naszych algorytmów, jeżeli będziemy sprawdzać unikalność znalezionych kombinacji. Jednocześnie możemy zaobserwować, że wersja 2 algorytmu jest lepiej zoptymalizowana, dla takiej samej dlugości ciągu potrzebuje znacznie mnniej czasu na wykonanie niż wersja 1.

Porównanie czasów wersji 1 ze sprawdzneniem unikalności oraz bez





5. Podsumowanie

Zadanie polegało na znalezieniu liczby trójelementowych kombinacji liczb z zadanego ciągu, których suma jest równa określonej wartości . W ramach projektu zaimplementowano dwie wersje algorytmu: pierwsza, wykorzystująca podejście brute force, opiera się na trzech zagnieżdżonych pętlach, co skutkuje wysoką złożonością obliczeniową (przy sprawdzaniu unikalności i bez tego sprawdzania). Druga wersja została zoptymalizowana poprzez redukcję jednej pętli i zastosowanie pomocniczej tablicy do śledzenia wcześniej widzianych liczb. Dzięki temu złożoność obliczeniowa została obniżona do przy sprawdzaniu unikalności i bez niej. Testy wydajności potwierdziły, że druga wersja algorytmu działa znacznie szybciej, szczególnie dla większych zbiorów danych. Dodatkowo analiza wykazała, że sprawdzanie unikalności znacząco wydłuża czas działania obu wersji, ale jednocześnie gwarantuje dokładność wyników. Optymalizacja algorytmu pozwoliła osiągnąć lepszą wydajność, co czyni drugą wersję bardziej odpowiednią dla dużych danych. Implementacja spełnia założenia projektu, oferując zarówno poprawność, jak i wydajność działania.

A. Kod programu

Kod programu w języku C++

Poniżej znajduje się pełny kod programu:

```cpp
#include <iostream>
#include <iostream>
#include <vector>
#include <unordered_set>
#include <fstream>
#include <sstream>
#include <chrono>
#include <cstdlib>
#include <iomanip>
using namespace std;

string plik_wejsciowy = "dane.txt"; //plik wejsciowy
string plik_wyjsciowy = "wyniki.txt"; //plik wyjsciowy

//funkcja wyswietlajaca wyniki w konsoli oraz zapisujaca je do pliku wyjsciowego
void wyswietl_wyniki_i_zapisz(vector<vector<int>>wyniki, int l_trojek, int wersja,
↪  int n) {
    static bool czy_pliku_czyszczono = false; // Flaga do jednorazowego
    ↪  czyszczenia pliku, tylko raz po kompilacji
    if (!czy_pliku_czyszczono) {
        ofstream plik_czyszczenie(plik_wyjsciowy, ios::trunc);
        plik_czyszczenie.close();
        czy_pliku_czyszczono = true; // Plik został wyczyszczony
    }

    ofstream plik(plik_wyjsciowy, ios::app);
        if(plik.is_open()) {
        cout<< "wersja "<<wersja<<": Liczba kombinacji wynosi " << l_trojek <<":
        ↪  "; //wyswietlamy wyniki w konsoli
        plik << "wersja "<<wersja<<": Liczba kombinacji wynosi " << l_trojek <<":
        ↪  "; //zapisujemy wyniki do pliku
        for(int i=0; i<wyniki.size(); i++) {
            cout <<"["<< wyniki[i][0] <<" "<< wyniki[i][1]<< " " << wyniki[i][2]
            ↪  << "] ";
```

```
30              plik << "["<< wyniki[i][0] <<" "<< wyniki[i][1]<< " " << wyniki[i][2]
                ↪  << "] ";
31          }
32          }
33          plik << "\n";
34          cout << endl;
35          plik.close();
36          if(n<3) {
37              cout << "wersja "<<wersja<<": Liczba kombinacji wynosi 0." << endl;
38              ofstream plik(plik_wyjsciowy, ios::app);
39              if (plik.is_open()) {
40                  plik<< "wersja "<<wersja<<": Liczba kombinacji wynosi 0." << endl;
41                  plik.close();
42              }
43          }
44  }
45
46  //pierwsza wersja algorytmu
47  void wersja_1(vector<int>tab, int M, bool sprawdz_unikalnosc) {
48      vector<vector<int>> wyniki; //wektor przechowywujacy znalezione trojki
49      int n = tab.size(); // dlugosc tablicy poczatkowej
50      int l_trojek = 0; // zmienna do liczenia wystpien odpowiednich trojek
51      int wersja = 1;
52
53      if (n >=3) {
54          for(int i=0; i<n-2; i++) {
55              for(int j=i+1; j<n-1; j++) {
56                  for(int k=j+1; k<n; k++) { // 3 zagniedzone petle do sprawdzania
                    ↪  wszystkich kombinacji
57                      if(tab[i] + tab[j] + tab[k] == M) { // sprawdzamy czy suma
                        ↪  danej kombinacji wynosi M
58                          vector<int> trojka={tab[i],tab[j], tab[k]};
59
60                          if(sprawdz_unikalnosc) {
61                          bool unikalna = true;
62                          for(int l=0; l<wyniki.size(); l++) { // sprawdzamy czy
                            ↪  aktualna trojka
63                              if(wyniki[l] == trojka) {          //  znajduje sie w
                                ↪  wynikach
```

```
                                unikalna = false;
                                break;
                            }
                        }

                        if(unikalna) {
                            wyniki.push_back(trojka); //dodajemy jesli trojka jest
                            ↪ unikalna
                            l_trojek++; //
                        }
                    } else {
                        wyniki.push_back(trojka);
                        l_trojek++;
                    }
                    }
                }
            }
        }
    wyswietl_wyniki_i_zapisz(wyniki, l_trojek, wersja, n); // wyswietlamy wyniki i
    ↪ zapisujemy do pliku
    }
}

//druga wersja algorytmu
void wersja_2(vector<int>tab, int M, bool sprawdz_unikalnosc) {
    vector<vector<int>> wyniki; //wektor przechowywujacy znalezione trojki
    int n = tab.size(); // dlugosc tablicy poczatkowej
    int l_trojek = 0; // zmienna do liczenia wystpien odpowiednich trojek
    int wersja = 2;

    if (n >=3) {
        for(int i=0; i<n-1; i++) {
            unordered_set<int> widziane; //zbior do przechowywania napotkanych juz
            ↪ liczb

            for(int j=i+1; j<n; j++) {
                int x = M - tab[i] - tab[j];

                if(widziane.find(x) != widziane.end()) { //sprawdzamy, czy x
                ↪ znajduje sie w widziane
```

```
100                     vector<int> trojka = {tab[i], x, tab[j]}; //tworzymy trojke
101
102                     if(sprawdz_unikalnosc) {
103
104                     bool unikalna = true;
105                     for(int l=0; l<wyniki.size(); l++) { // sprawdzamy czy
                        ↪ aktualna trojka
106                         if(wyniki[l] == trojka) {          //  znajduje sie w
                            ↪ wynikach
107                             unikalna = false;
108                             break;
109                         }
110                     }
111
112                     if(unikalna) {
113                         wyniki.push_back(trojka); //dodajemy jesli trojka jest
                            ↪ unikalna
114                         l_trojek++; //zwiekszamy licznik trojek
115                     }
116                     } else {
117                         wyniki.push_back(trojka);
118                         l_trojek++;
119                     }
120                 }
121                 widziane.insert(tab[j]); //dodajemy napotkana liczbe do zbioru
                    ↪ widziane
122             }
123         }
124     wyswietl_wyniki_i_zapisz(wyniki, l_trojek, wersja,n); // wyswietlamy wyniki i
        ↪ zapisujemy do pliku
125     }
126 }
127
128 //funkcja wczytujaca dane z pliku
129 void wczytaj_z_pliku(string plik_wejsciowy) {
130     ifstream wejscie(plik_wejsciowy);
131
132     if (!wejscie.is_open()) {
133         cout << "Nie mozna otworzyc pliku wejsciowego!" << endl;
```

```
134         return;
135     }
136 
137     vector<int> tab; // Tablica do przechowywania liczb
138     int M;           // Wartosc
139     int zestaw_numer = 1;
140 
141     while (true) {
142         tab.clear();
143 
144         // Wczytaj tablice liczb
145         string linia;
146         if (!getline(wejscie, linia)) break;
147         stringstream ss(linia);
148         int liczba;
149         while (ss >> liczba) {
150             tab.push_back(liczba);
151         }
152 
153         // Wczytaj wartosc M
154         if (!getline(wejscie, linia)) break;
155         M = stoi(linia);
156 
157         //wywolujemy funkcje dla danych z pliku
158         bool sprawdz_unikalnosc=false;
159         wersja_1(tab, M, sprawdz_unikalnosc);
160         wersja_2(tab, M, sprawdz_unikalnosc);
161 
162         zestaw_numer++;
163     }
164     wejscie.close();
165 
166 }
167 
168 //Dane do testowania
169 vector<int>tab1={-5,6,1,2,5,1,2,1,2,4};
170 vector<int>tab2={};
171 vector<int>tab3={1,2,3,1,2,3,1,2,3};
172 vector<int>tab4={2,2,2,2,2,2,2,2};
```

```
173
174 //Szukana suma
175 int M=6;
176
177 //Funkcja testujaca wersje 1 algorytmu
178 void testy_wersji_1() {
179     bool sprawdz_unikalnosc=true;
180     wersja_1(tab1, M, sprawdz_unikalnosc);
181     wersja_1(tab2, M, sprawdz_unikalnosc);
182     wersja_1(tab3, M, sprawdz_unikalnosc);
183     wersja_1(tab4, M, sprawdz_unikalnosc);
184 }
185
186 //Funkcja testujaca wersje 2 algorytmu
187 void testy_wersji_2() {
188     bool sprawdz_unikalnosc=true;
189     wersja_2(tab1, M, sprawdz_unikalnosc);
190     wersja_2(tab2, M, sprawdz_unikalnosc);
191     wersja_2(tab3, M, sprawdz_unikalnosc);
192     wersja_2(tab4, M, sprawdz_unikalnosc);
193 }
194
195 vector<int> generuj_tablice(int n, int min_val, int max_val) {
196     vector<int> tab;
197     for (int i = 0; i < n; ++i) {
198         tab.push_back(min_val + rand() % (max_val - min_val + 1));
199     }
200     return tab;
201 }
202
203 // Funkcja generująca tabelkę wyników
204 void generuj_tabelke() {
205     cout << setw(4) << "L.p." << setw(10) << "n" << setw(15) << "t1 [s]" <<
    ↪  setw(15) << "t2 [s]" << endl;
206     cout << string(44, '-') << endl;
207     vector<int> rozmiary_n = {250, 500, 750, 1000, 1250, 1500, 1750, 2000};
208     int min_val = -10, max_val = 10, M = 6;
209     for (size_t i = 0; i < rozmiary_n.size(); ++i) {
210         int n = rozmiary_n[i];
```

```
211         vector<int> tab = generuj_tablice(n, min_val, max_val);
212 
213         bool sprawdz_unikalnosc=false;
214 
215         // Mierzenie czasu dla wersja_1
216         auto start1 = chrono::high_resolution_clock::now();
217         wersja_1(tab, M, sprawdz_unikalnosc); // Zakładamy, że wersja_1 jest
            ↪  poprawnie zaimplementowana
218         auto end1 = chrono::high_resolution_clock::now();
219         chrono::duration<double> czas1 = end1 - start1;
220 
221         // Mierzenie czasu dla wersja_2
222         auto start2 = chrono::high_resolution_clock::now();
223         wersja_2(tab, M, sprawdz_unikalnosc); // Zakładamy, że wersja_2 jest
            ↪  poprawnie zaimplementowana
224         auto end2 = chrono::high_resolution_clock::now();
225         chrono::duration<double> czas2 = end2 - start2;
226 
227         // Wypisanie wyników
228         cout << setw(4) << i + 1
229              << setw(10) << n
230              << setw(15) << fixed << setprecision(6) << czas1.count()
231              << setw(15) << fixed << setprecision(6) << czas2.count() << endl;
232     }
233 }
234 
235 int main()
236 {
237     testy_wersji_1();
238     testy_wersji_2();
239     wczytaj_z_pliku(plik_wejsciowy); //wczytanie danych z pliku
240     //generuj_tabelke();
241 
242     return 0;
243 }
244 }
```