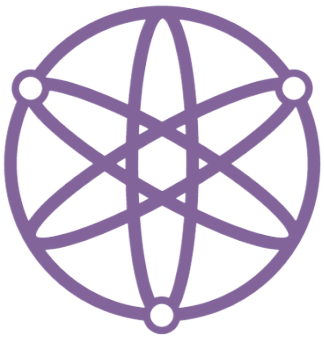


Algorytmy i struktury danych

zadanie projektowe

Jakub Świątek – 179987 – L8

2025-01-27

Spis treści

1 Treść zadania

Dla zadanego zbioru liczb wypisz jego najmniej liczny podzbiór, którego suma jest większa niż suma pozostałych elementów tego zbioru.
Przykład:
Wejście: [2,4,1,3,9,0,3]
Wyjście: [4,9],[3,9]

2 Pierwsze podejście (brute force)

2.1 Analiza problemu

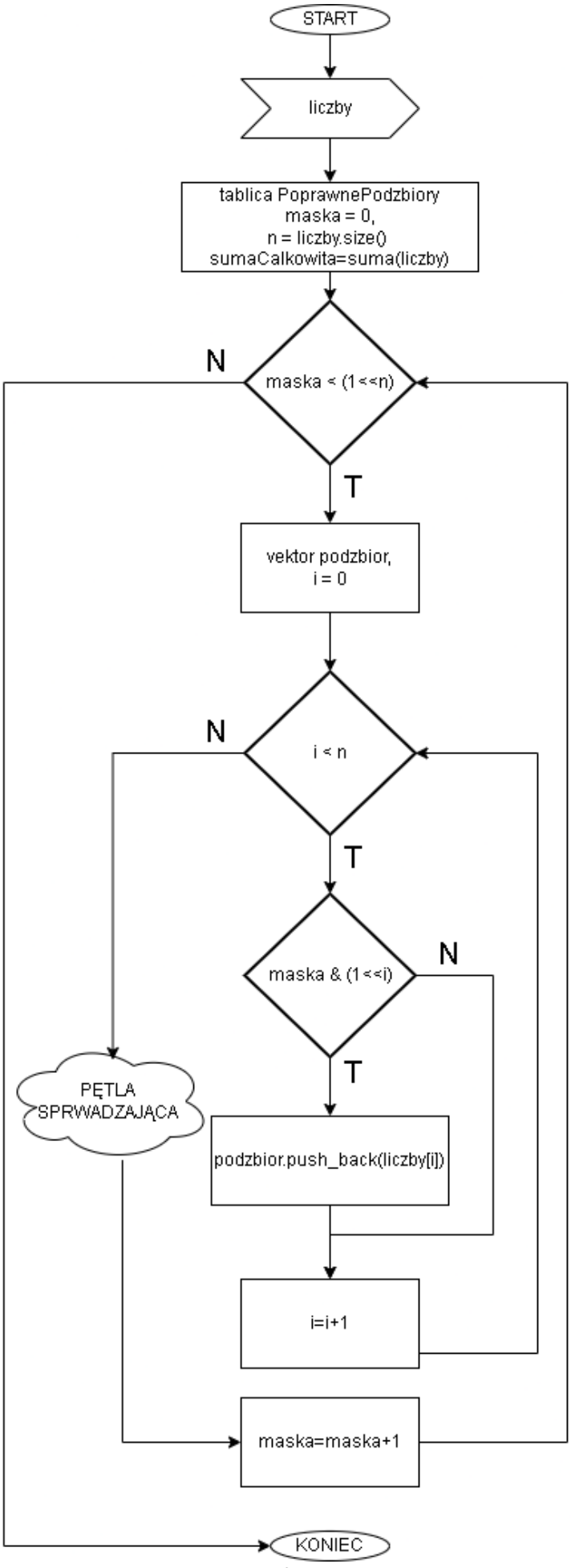
Pierwsze podejście będzie cechować się umyślnym brakiem "finezji". Zgodnie z poleceniem, docelowym wyjściem mają być najmniejsze podzbiory. Program ten w takim razie będzie można podzielić na dwie części.

- Pierwsza część: Funkcja generująca podzbiory do analizy
- Druga część: Funkcja sprawdzająca czy podzbiór spełnia warunek polecenia

Zgodnie z "siłową"filozofią metody bruteforce problem rozwiążemy w sposób następujący.
W pierwszej kolejności na podstawie otrzymanego zbioru, generujemy wszystkie możliwe podzbiory (o wszystkich możliwych rozmiarach oraz sumach). Po czym każdy z podzbiorów przechodzi weryfikację przez funkcję i poprawne z nich zapisuje w zmiennej. Następnie wśród poprawnych podzbiorów szukamy najmniejszego z nich, zachowujemy jego rozmiar w zmiennej, a na sam koniec wypisujemy wszystkie podzbiory, których rozmiar jest równy zachowanemu rozmiarowi.
Ułatwieniem wartym dodania jest także funkcja pomocnicza suma(), która będzie zwracać sumę danego zbioru.

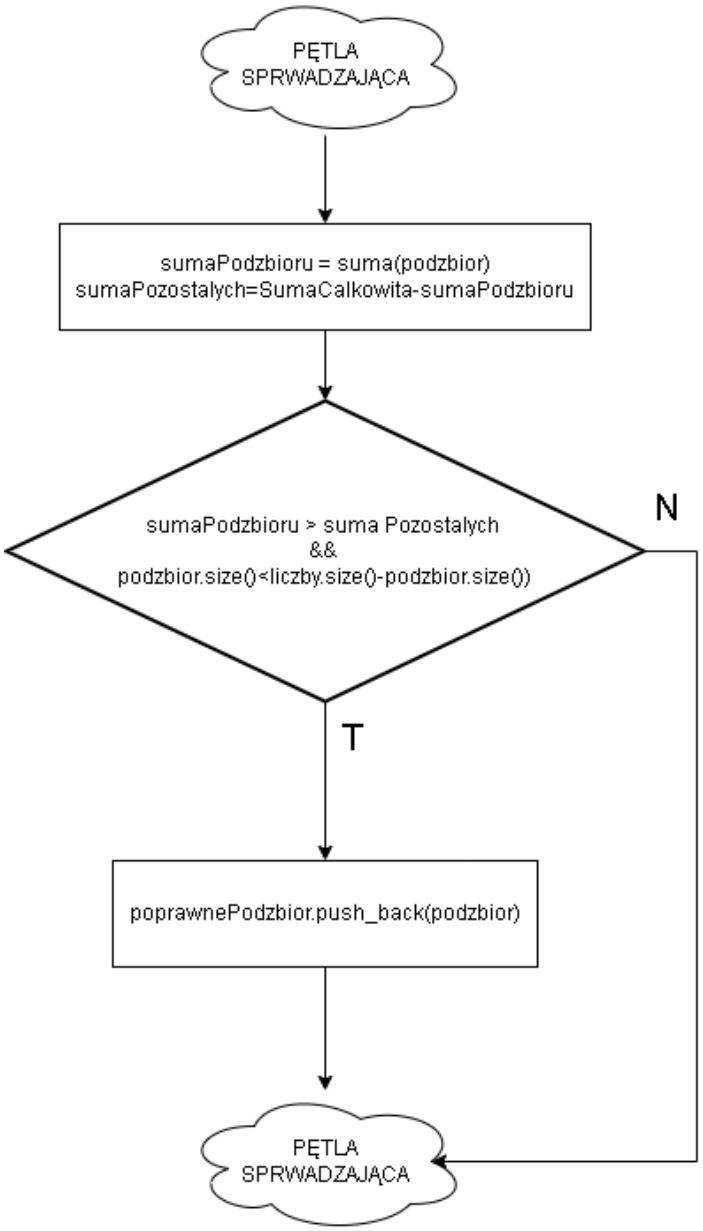
2.2 Schemat blokowy algorytmu

Aby zachować przejrzystość oraz w celu łatwiejszej identyfikacji, schemat blokowy został podzielony na dwie ilustracje.



Rysunek 1: Schemat blokowy algorytmu bruteforce-część 1

Schemat przedstawiający część pętli sprawdzającej, celu zachowania przejrzystości został oznaczony symbolicznie "chmurką"i przedstawiony na następującym schemacie:



Rysunek 2: Schemat blokowy algorytmu bruteforce-część 2

2.3 Komentarz względem schematu

W schematach użyto sformułowań używanych w bibliotekach C++ oraz funkcji, które zostaną zaimplementowane w kodzie, ale nie zostały zawarte w schemacie. Rozwiązanie takie służy zapobiegnięciu nieporozumień oraz zmniejszeniu skomplikowania schematu blokowego. Oznaczają one:

- zmienna.push_back() - dodaje nowy element na końcu wektora
- suma() - zwraca sumę elementów zawartych w wektorze
- zmienna.size() - zwraca liczbę równą rozmiarowi podanej zmiennej

Przesunięcie bitowe

W celu wygenerowania wszystkich możliwych podzbiorów zastosowano przesunięcie bitowe.

(1 « n) oznacza przesunięcie 1 o n bitów np. 1 « 2 oznacza binarnie 100 (dziesiętnie 2^2)

Jest to najważniejsze dla warunku: maska & (1 « i). Wyrażenie sprawdza, czy i-ty bit w liczbie maska jest równy 1. Jeśli tak, dodaje tą liczbę do podzbioru.

Przykład: liczby={1,2,3} n=3, w zależności od maska

Dla maska=5 (binarnie 101):

- Bit pierwszy i trzeci -> i=0 oraz i=2 -> liczby[0] oraz liczby[2] trafia do podzbioru

Przez co otrzymujemy podzbiór złożony z {1,3}

Dla maska=2 (binarnie 010):

- Bit drugi -> i=1 -> liczby[1] trafia do podzbioru

Przez co otrzymujemy podzbiór złożony z {2}

W ten sposób przechodząc przez 001,010,011,... otrzymamy wszystkie możliwe podzbiory utworzone z podanego zbioru. Wszystkie możliwości są zależne od rozmiaru zbioru (rozmiar jest przechowywany w n). Przez takie rozwiązanie złożoność obliczeniowa będzie ściśle związana właśnie z tym procesem.

2.4 Algorytm bruteforce w pseudokodzie

```
FUNCKJA ZnajdzPodzbiory(liczby):
    TABLICA PoprawnePodzbiory
    maska=0
    n=liczby.rozmiar() // ilość liczb w zbiorze liczby
    sumaCalkowita=suma(liczby) //suma wszystkich liczb w zbiorze liczby

    DOPÓKI maska < (1«n): //Wykonuje pętlę dla każdego możliwego podzbioru
      wektor podzbior

      DOPÓKI i<n:
        JEŻELI maska AND (1«i): //koniunkcja bitowa AND (inaczej mnożenie)
          podzbior.dodajNaKoniec(liczby[i])
      i=i+1

      sumaPodzbioru=suma(podzbior)
      sumaPozostalych=sumaCalkowita - suma Podzbioru
        JEŻELI sumaPodzbioru > sumaPozostalych) I podzbior.rozmiar() < n-podzbior.rozmiar())
          PoprawnePodzbiory.dodajNaKoniec(podzbior)

    maska=maska+1

    wypisz(PoprawnePodzbiory) //wypisujemy poprawne wyniki
```

2.5 Ołówkowe sprawdzenie algorytmu

Oto przykładowe fragmenty działania algorytmu:
n=3 liczby = {1,2,4}

m	m < (1«n)	i	i<n	m&(1«i)	podz	sPoz	sPodz	pPod
0	0 < 8	0	0<3	0 AND 1	-	7	-	-
0	0 < 8	1	1<3	0 AND 10	-	7	-	-
0	0 < 8	2	2<3	0 AND 100	-	7	-	-
1	1 < 8	0	0<3	1 AND 1	{4}	7	-	-
1	1 < 8	1	1<3	1 AND 10	{4}	7	-	-
1	1 < 8	2	2<3	1 AND 100	{4}	3	4	push({4})
2	2 < 8	0	0<3	10 AND 1	-	-	-	-
2	2 < 8	1	1<3	10 AND 10	{2}	-	-	-
2	2 < 8	2	2<3	10 AND 100	{2}	2	5	-
3	3 < 8	0	0<3	11 AND 1	{4}	-	-	-
3	3 < 8	1	1<3	11 AND 10	{2,4}	-	-	-
3	3 < 8	2	2<3	11 AND 100	{2,4}	1	6	-

Tabela 1: Tabela wartości zmiennych

W ostatniej linijce podzbiór nie przechodzi przez podzbior.rozmiar() < n-podzbior.rozmiar())

n=4 liczby = {2,1,1,5}

m	m < (1«n)	i	i<n	m&(1«i)	podz	sPoz	sPodz	pPod
0	0 < 16	0	0<4	0 AND 1	-	7	-	-
0	0 < 16	1	1<4	0 AND 10	-	7	-	-
0	0 < 16	2	2<4	0 AND 100	-	7	-	-
0	0 < 16	3	3<4	0 AND 1000	-	7	-	-
1	1 < 16	0	0<4	1 AND 1	{5}	7	-	-
1	1 < 16	1	1<4	1 AND 10	{5}	7	-	-
1	1 < 16	2	2<4	1 AND 100	{5}	7	-	-
1	1 < 16	3	3<4	1 AND 1000	{5}	4	5	push({5})

Tabela 2: Druga abela wartości zmiennych

2.6 Oszacowanie złożoności obliczeniowej

Analizując algorytm docieramy do dwóch kluczowych momentów.
Obliczenia ilości wszystkich możliwych podzbiorów oraz ustalenia rozmiaru zbioru. Przesunięcie bitowe to dodanie n zer przed jedynką w systemie binarnym. Inaczej przedstawić ten zabieg można jako 2^n.

- iteracja po każdym możliwym podzbiorze O(2^n)
- budowanie podzbioru zajmuje O(n)
- sumowanie elementów podzbioru O(n)

Szacując złożoność obliczeniową otrzymujemy

$$O(2^n \cdot n \cdot n)$$

Liczba obliczeń będzie drastycznie rosnąć wraz z rozmiarem zbioru.

- dla n=1: O($2^1 \cdot 1 \cdot 1$) = 2
- dla n=10: O($2^{10} \cdot 10 \cdot 10$) = 102400
- dla n=20: O($2^{20} \cdot 20 \cdot 20$) = 419430400

2.7 Implementacja algorytmu w C++

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

// Funkcja pomocnicza do obliczenia sumy elementow wektora
int suma(const vector<int>& wektor) { //przyjmuje referencje
//do wektora wektor, zamiast kopiowac dane
//const aby nie zmienic wartosci
    int sumaCalkowita = 0;
    for (int liczba : wektor) { //petla range for, liczba
    //w kolejnych iteracjach przyjmuje kolejne wartosci
    //w wektorze wektor
        sumaCalkowita += liczba;
    }
    return sumaCalkowita;
}

// Funkcja generujaca wszystkie podzbiory i sprawdzajaca warunek
void znajdzPodzbiory(vector<int>& liczby) {
    int n = liczby.size();
    int sumaCalkowita = suma(liczby);
    vector<vector<int>> poprawnePodzbiory; //wektor wektorow
    //(w pewnym sensie tablica liczb) przechowujacy podzbiory
    //spelniajace warunki

    // Iterujemy po wszystkich podzbiorach (2^n mozliwosci)
    for (int maska = 0; maska < (1 << n); ++maska) {
        vector<int> podzbior;
        for (int i = 0; i < n; ++i) {
            if (maska & (1 << i)) {
                podzbior.push_back(liczby[i]);
            }
        }
        // Sprawdzamy czy suma podzbioru spelnia warunek
        int sumaPodzbioru = suma(podzbior);
        int sumaPozostalych = sumaCalkowita - sumaPodzbioru;
        if ((sumaPodzbioru > sumaPozostalych)&&
        (podzbior.size()<liczby.size()-podzbior.size())) {
            poprawnePodzbiory.push_back(podzbior);
        }
    }



    // Wypisujemy wyniki
```

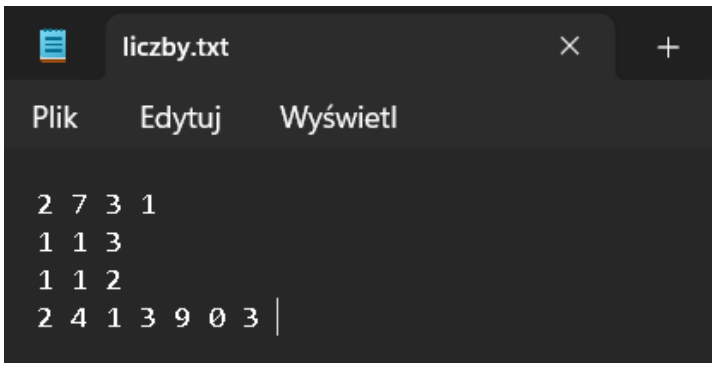
```
    if (poprawnePodzbiory.empty()) {
        cout << "Brak podzbiorow ";
        cout << "spelniajacych zadane kryteria." << endl;
    }
    else {
        int minimalnyRozmiar=poprawnePodzbiory[0].size();
        for(int i=0;i<poprawnePodzbiory.size();i++){
        if(poprawnePodzbiory[i].size()<minimalnyRozmiar){
        minimalnyRozmiar=poprawnePodzbiory[i].size();}}
        for (const auto& podzbior : poprawnePodzbiory) {
            if (podzbior.size() == minimalnyRozmiar){
                cout << "[";
                for (size_t i = 0; i < podzbior.size(); ++i) {
                    cout << podzbior[i];
                    if (i < podzbior.size() - 1)cout << ", ";
                }
            cout << "]\n";
            }
        }
    }
}

int main() {
    vector<int> liczby = {2,1,1,4,7,10,7}; // Przykladowe dane
    znajdzPodzbiory(liczby); //wywolanie funckji

    return 0;
}
```

W celu zmieszczenia całej zawartości w określonym formacie niektóre linijki kodu oraz zdania zostały celowo skrócone lub przerzucone do kolejnej linii. Lepsza wersja dostępna na githubie.

2.8 Przykładowe wyniki i ich analiza

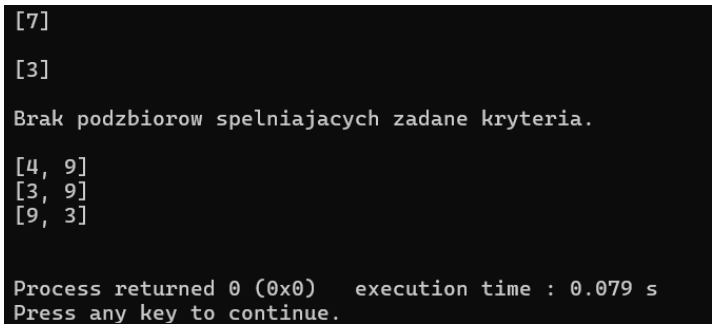
Do utworzenia poniższych wyników użyto innej wersji programu niż podanej wyżej. Wyniki wygenerował BRUTEFORCEzPliku.cpp dostępny na GitHubie.

Przygotowujemy plik z przykładowymi danymi i zapisujemy go pod nazwą liczby.txt



Rysunek 3: Przykładowe dane w pliku .txt

Program zwraca dane:



Rysunek 4: Dane wyjściowe dla przykładowych danych

Zgodnie z przykładowymi danymi z polecenia. Jeśli podzbiór posiada większy lub równy rozmiar niż zbiór pozostałych elementów nie spełnia on wymagań.
Dzięki przeprowadzonym testom dowiadujemy się o słabości naszego programu.

Jeśli w wprowadzonym podzbiorze znajduje się wielokrotnie ta sama liczba dochodzi do powtórzenia danych. [3,9] i [9,3] to ten sam pozdbiór ale teoretycznie złożony z "innych trójek"(są w zbiorze na różnych pozycjach).

Zostanie to wzięte pod uwagę przy tworzeniu "sprytnego"kodu. W dalszej części sprawozdania znajduje się rozwiązanie tego problemu.

3 Drugie podejście

Nie ograniczając się podejściem siłowym, drugie podejście wykorzysta wiedzę o słabościach pierwszego programu aby zredukować złożność obliczeniową w stosunku do poprzednika.

3.1 Ponowna analiza problemu

Największą wadą pierwszego programu jest ilośc analizowanych podzbiorów. Powoduje to dużą ilość niepotrzebnych operacji.

Poprawki wymyślone na podstawie poprzedniego programu:

1. Redukcja generowanych podzbiorów
2. Skromna zmiana warunku porównywania podzbioru
3. Usunięcie powtórek

Jak zostaną wprowadzone poprawki:

1. W praktyce analizować musimy tylko i wyłącznie te podzbiory, które mają rozmiar najmniejszego z nich, który spełnia warunek. Wyszukując taki podzbiór, możemy wykorzystać wiedzę o jego rozmiarze do wygenerowania odpowiednich podzbiorów.

   Czym w praktyce jest najmniejszy podzbiór?

   Zbiorem zawierającym kolejno: największy element, drugi największy element, trzeci największy element... itd.
   Dlatego lepsza wersja na samym początku wyszuka takowy podzbiór a potem wykorzysta jego rozmiar do zmniejszenia ilości analizowanych podzbiorów.

2. Poprzednio warunek wyglądał tak: suma(podzbioru)>suma(pozostałych elementów zbioru)
   Takie rozwiązanie zmuszało komputer do wywołania funckji sumującej aż dwa razy na każdy podzbiór. Tak naprawdę nie jest to konieczne.
   Przykład:

   $$nic > 7, 3, 2, 1/0 > 13$$

   $$7 > 3, 2, 1/7 > 6$$

   W ten sposób odejmujemy 7 z prawej strony i dodajemy z lewej. Jest to równoznaczne z:

   $$nic > 7, 3, 2, 1/0 > 13$$

   $$14 > 7, 3, 2, 1/14 > 13| - 7 => 7 > 6$$

   Dzięki czemu nasz warunek może wyglądać następująco:

   $$suma(podzbioru) \cdot 2 > sumaZbioru$$

   Gdzie sumę zbioru policzymy o wiele wcześniej i nie będziemy powtarzać tej operacji.

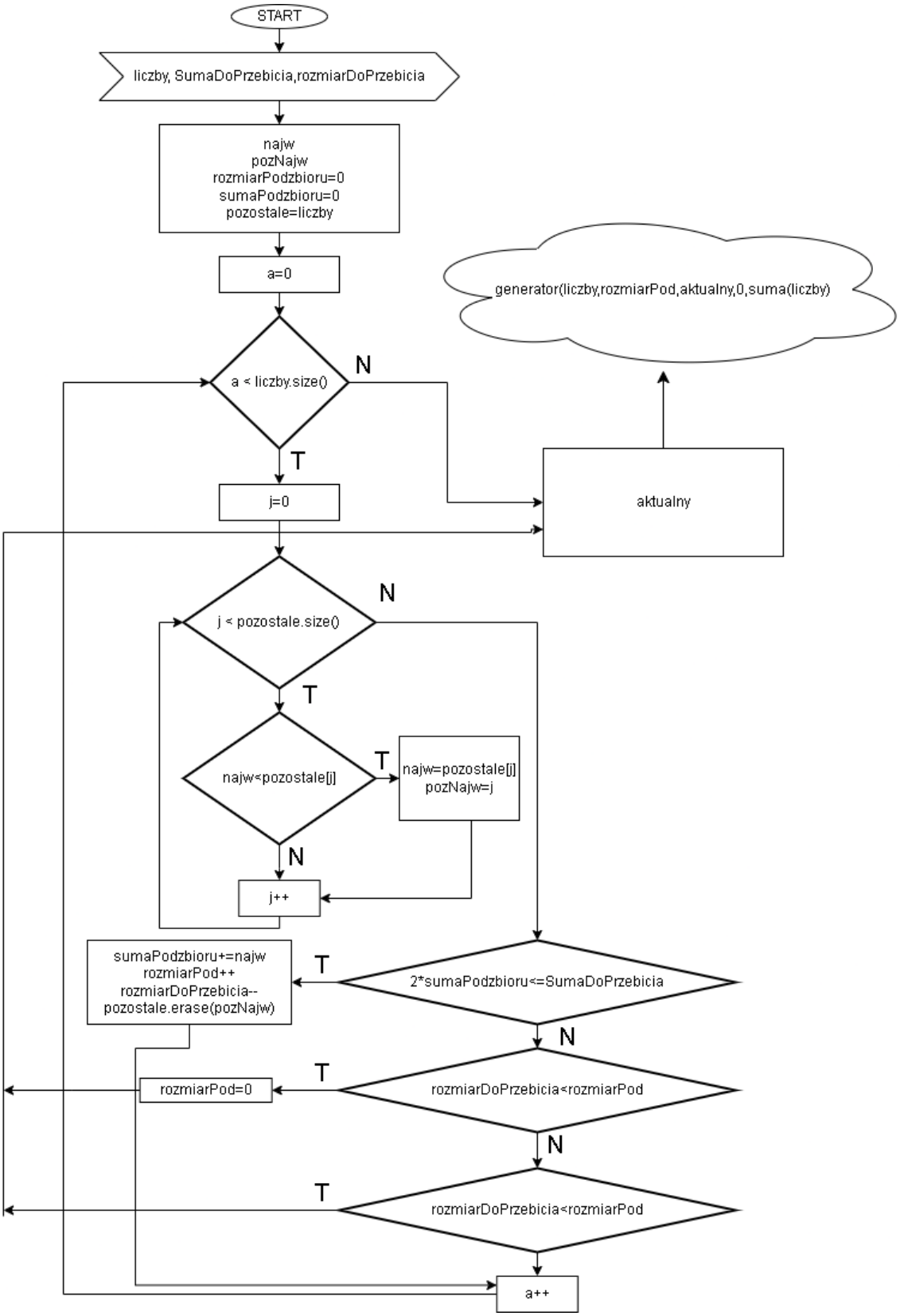
3. Powtóri pojawiają się z powodu wystąpienia tej samej liczby kilka razy ale w różnych miejscach. Podzbiory {3,9} i {9,3} składają się z tych samych liczb ale dla komputera to dwa oddzielne podzbiory i przede wszystkim dwie różne cyfry 3. Aby pozbyć się takich powtórzeń:

   - Te same liczby zostaną ustawione obok siebie
   - Powtórki zostaną usunięte za pomocą zmiennej set

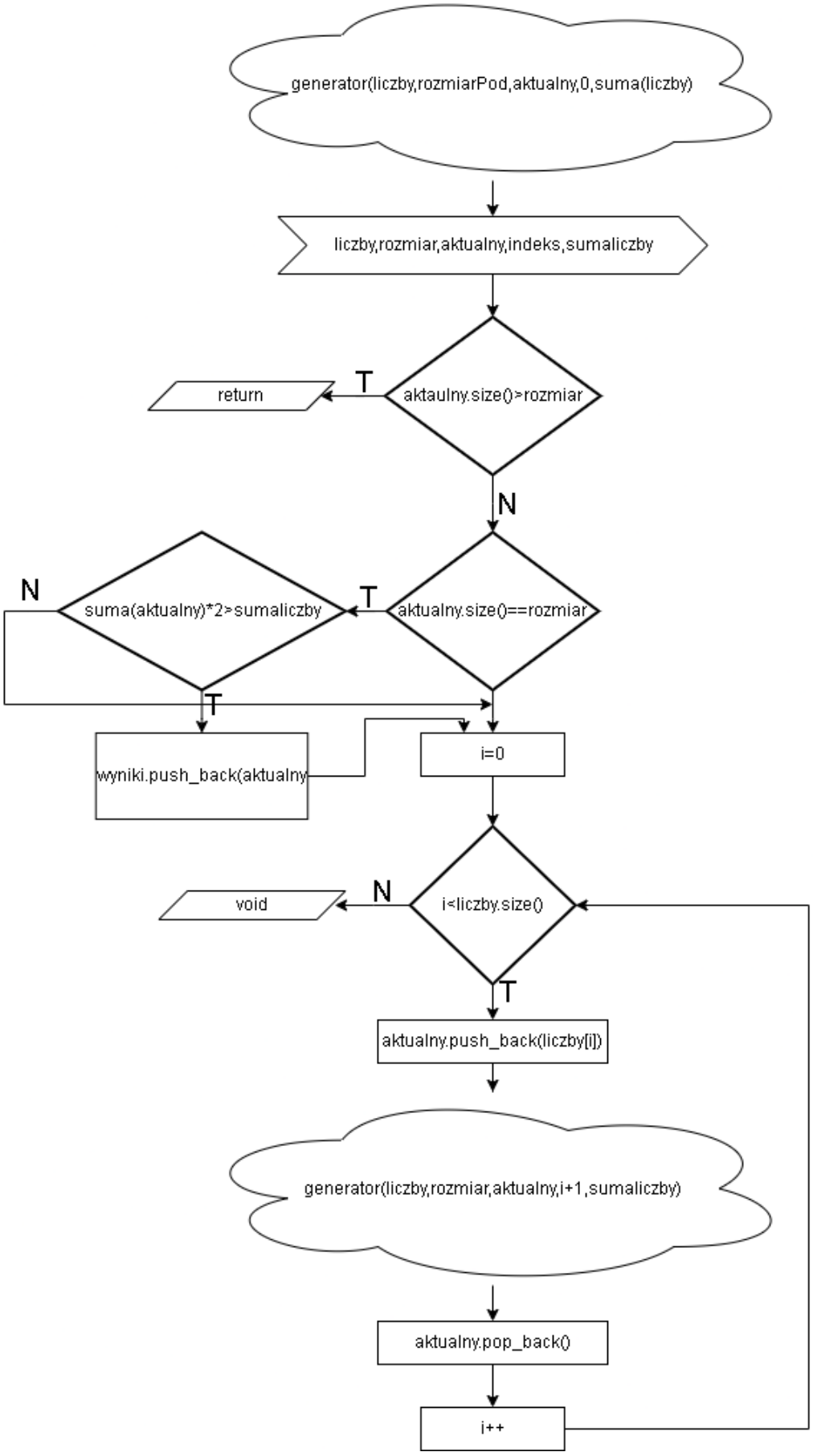
   Dzięki zmiennej set rozwiążemy problem powtórek. Kontener "set" przeznaczony jest do przechowywania **unikalnych** elementów w określonej kolejności. Wartości elementów w kontenerze set nie mogą się powtarzać dzięki czemu automatycznie usunie powtórki.

3.2 Schemat blokowy algorytmu

Aby zachować przejrzystość oraz w celu łatwiejszej identyfikacji, schemat blokowy został podzielony na dwie ilustracje.



Rysunek 5: Schemat blokowy lepszego algorytmu-część 1



Rysunek 6: Schemat blokowy lepszego algorytmu-część 1

3.3 Komentarz do schematu blokowego

Druga część schematu blokowego przedstawia funkcję rekurencyjną. W środku głównej funkcji zawarto "generator", jako oddzielną funkcję wywołującą samą siebie. Działanie tej funkcji zostanie przedstawione w dalszej części dokumentu. Ponownie jak poprzednio ze względu na konieczność użycia wektorów, niektóre polecenia zapożyczają nazwę z biblioteki vector.

W schemacie pojawiają się także dwa niestandardowe bloki "void"oraz "return"(bez zwracanej wartości). Pojawiają się one ze względu, że sama funkcja nie ma na celu zwrócenia konkretnej wartości. Dlatego, w implementacji zostanie użyta funkcja void. Mając to na względzie już teraz wprowadzono takowe nazwenictwo i właściwości funkcji z nim związane.

3.4 Pseudokod dla lepszej wersji algorytmu

```
funkcja generator(liczby,rozmiar,aktualny,indeks,sumaliczby) {
jeśli(aktualny.rozmiar()>rozmiar)return
jeśli(aktualny.rozmiar()==rozmiar)
    jeśli(suma(aktualny)*2>sumaliczby){
       wyniki.push_back(aktualny)
}
}
i=indeks
dopóki(i<liczby.rozmiar())
     aktualny.push_back(liczby[i])
     generator(liczby,rozmiar,aktualny,i+1,sumaliczby)
     aktualny.pop_back()
}
}
funkcja ZnajdzPodzbiory(liczby, sumaDoPrzebicia, rozmiarDoPrzebicia) {
najw=0,rozmiarPod=0,sumaPodzbioru=0,pozNajw=0
pozostale=liczby //wektory
a=0
     dopóki(a<liczby.rozmiar()){
        najw=0
        j=0
           dopóki(j<pozostale.rozmiar()){
              jeśli(najw<pozostale[j]){najw=pozostale[j],pozNajw=j}
              j++
}
        if(2*sumaPodzbioru<=SumaDoPrzebicia){
           sumaPodzbioru+=najw
           rozmiarPod++
           rozmiarDoPrzebicia- -
           pozostale.usun(pozNajw)
}
        if(rozmiarDoPrzebicia<rozmiarPod){
           rozmiarPod=0
           break
}
        if(2*sumaPodzbioru>SumaDoPrzebicia){
           break
}
     a++
}
aktualny
generator(liczby,rozmiarPod,aktualny,0,suma)
}
```

3.5 Ołówkowe sprawdzenie poprawności algorytmu

W tym segmencie wyjaśniona zostanie część bez funkcji generator. Schemat rekurencji zostanie przedstawiony w inny sposób.

Przykład dla [1,5,2,6,1]

a < l.size()	j < poz.size()	poz[j]	najw	sumaPod	sumaDoPrzeb	rozDoPrzeb	Akcja
0	0	1	0	0	0>14	5<0	-
0	1	5	0	0	0>14	5<0	-
0	2	6	6	6	12>14	4<1	-
1	0	1	0	6	12>14	4<1	-
1	2	5	5	11	22>14	3<2	push{6,5}

Tabela 3: Tabela wartości zmiennych

Po znalezeniu takiego podzbioru wywołujemy funkcję generator.

Przykład dla [1,1,2]

a < l.size()	j < poz.size()	poz[j]	najw	sumaPod	sumaDoPrzeb	rozDoPrzeb	Akcja
0	0	1	0	0	0>5	3<0	-
0	1	1	0	0	0>5	3<0	-
0	2	2	2	2	4>5	2<1	-
1	0	1	1	3	6>5	1<2	break

Tabela 4: Tabela wartości zmiennych

Najmniejszy podzbiór, który spełnia warunek jest za duży, przez to funkcja zwraca najmniejszy rozmiar 0. W ten sposób wygenerowany nie zostanie żaden podzbiór. Brak odpowiedzi

Omówienie rekurencji

Rekurencja jest na tyle złożonym procesem, że ciężko byłoby to przedstawić jednocześnie wraz z zmiennymi w tabeli. Oto przedstawienie działania funkcji dla przykładowych danych.

Przykład: {3,1,3,2,1} rozmiar=2

1. Pierwsza iteracja
   Dodaj liczby[0] do aktualny => 3
   Wywołanie rekurencji
   - Rekurencja, i=1
     Dodaj liczby[1] do aktualny => 3,1
     Warunek na rozmiar spełniony, warunek na sume nie
     Usuwamy ostatni element
   - Rekurencja, i=2
     Dodaj liczby[2] do aktualny => 3,3
     Warunek na rozmiar spełniony, warunek na sume też, dodajemy 3,3 do wyniki

- Rekurencja, i=3
  Dodaj liczby[3] do aktualny => 3,2
  Warunek na rozmiar spełniony, warunek na sume nie

- Rekurencja, i=4
  Dodaj liczby[2] do aktualny => 3,1
  Warunek na rozmiar spełniony, warunek na sume nie

2. Druga iteracja
   Dodaj liczby[1] do aktualny => 1
   Wywołanie rekurencji

   - Rekurencja 2, i=2
     Dodaj liczby[2] do aktualny => 1,3
     Warunek na rozmiar spełniony, warunek na sume nie
     Usuwamy ostatni element

   - Rekurencja 2, i=3
     Dodaj liczby[3] do aktualny => 1,2
     Warunek na rozmiar spełniony, warunek na sume nie
     Usuwamy ostatni element

   - Rekurencja 2, i=4
     Dodaj liczby[4] do aktualny => 1,1
     Warunek na rozmiar spełniony, warunek na sume nie
     Usuwamy ostatni element

3. Trzecie iteracja,i=3

   - Itd.

Dzięki zastosowaniu takiego rozwiązania, nie otrzymujemy także niektórych powtórzeń. Np Otrzymujemy {liczby[0],liczby[1]} ale {liczby[1],liczby[0]} już nie.

3.6 Oszacowanie złożoności obliczeniowej

Ponownie w kodzie są dwa kluczowe momenty: funckja generator oraz znalezienie najmneijszego rozmiaru. Ich złożoność obliczeniowa to:

- Rekurencja w najgorszym wypadku generuje wszystkie możliwe podzbiory o rozmiarze (n/2)-1 (zaokrąglając w górę)

$$O(\frac{n!}{(\frac{n}{2}-1)! \cdot (n-(\frac{n}{2}-1))!})$$

- Znalezienie najmniejszego podzbioru spełniającego warunek(pamiętając, że rozmiar podzbioru nie może być większy niż rozmiar pozostałych elementów)

$$O(\frac{n^2}{4})$$

- Sumowanie elementów podzbioru

$$O(n)$$

Przybliżając klasa złożoności obliczeniowej wynosi:

$$O(n^2 + 2^n \cdot n) \approx O(2^n \cdot n)$$

Okazuje się, że algorytm ma klasę złożoności bliską poprzedniemu. Jest to jedynie pozór ponieważ sam kod działa o wiele szybciej. Mimo podobnej klasy złożoność obliczeniowa lepszego kodu wygląda mniej więcej:

$$O(n \cdot \frac{2^n}{n} + \frac{n^2}{4})$$

Warto dodać, że oba działania zostały zaokrąglone w górę.

Klasa klasie nie równa

Przyjrzyjmy się konkretnym przypadkom

Wszystkie działania mają JEDYNIE charakter poglądowy. Nie mają na celu nieść konkretnej wiedzy na temat złożoności obliczeniowej programów. Segment ten został zawarty dla przedstawienia różnic między wersjami algorytmów. Ogromne zaokrąglenia i przeprowadzone działania pokazują jedynie "szkic"rozbieżności między kodami.

- dla n=20:
  Brute-force: O($2^{20} \cdot 20 \cdot 20$) = 4194304008
  Optymalny: O(?) = ?

  W rzeczywistości ilość operacji w wersji optymalnej jest ściśle związana z wynikiem. Co znacząco wpływa na czas wymagany na rozwiązanie problemu. Przyjrzyjmy się temu samemu przykładowi ale zakładając, że znamy rozmiar szukanego podzbioru.

- dla n=20 z brakiem odpowiedzi:
  Brute-force: O($2^{20} \cdot 20 \cdot 20$) = 4194304008
  Optymalny: $O(1 + 20^2) \approx 400$

  Ponieważ ostatni analizowany podzbiór będzie miał rozmiar 9. Aby znaleźć każdy jego element przeszukamy pozostałe o rozmiarze 20,19,18,... co zaokrąglamy do 20. Funkcja generator wywoła się tylko dla 0, czyli jeden raz.Wszystko z powodu warunków, których wersja bruteforce nie posiada. W rzeczywistości ilość operacji byłaby mniejsza niż 400 (tutaj wzięto pod uwagę tylko warunek w rekurencji).

- dla n=20 odpowiedź wymagająca najwięcej obliczeń
  Brute-force: O($2^{20} \cdot 20 \cdot 20$) = 4194304008
  Optymalny: O($20^2 + 2^{20} \cdot 20$) $\approx$ 20971920

  Wyniki byłyby takowe w przypadku podzbioru, który ma rozmiar 9 oraz spełnia warunek, przez co konieczna jest analiza wszystkich możliwych podzbiorów dla rozmiaru 9. Warto zauważyć że liczba operacji dla optymalnego byłaby o wiele mniejsza mimo że w tym przypadku nie wzięto pod uwagę, żadnego z warunków, który skróciłby ilość wykonanych operacji.

- dla n=20 rozmiar najmniejszego podzbioru = 1
  Brute-force: O($2^{20} \cdot 20 \cdot 20$) = 4194304008
  Optymalny: O($20 + 1 \cdot 1$)$\approx$ 21

  Celem byłoby wyszukanie największego elementu i sprawdzenie jedynie tego elementu.

3.7 Implementacja kodu w C++

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;
vector<vector<int>> wyniki; //globalna zmienna
przechowywujaca wyniki

vector<int> powtorki(vector<int>liczby){ /*funckja zmieniajaca
pozycje tych samych liczb (przydatne do set)*/
for(int i=0;i<liczby.size()-1;i++){
    for(int j=liczby.size()-1;j>i;j--){
        if(liczby[i]==liczby[j])swap(liczby[i+1],liczby[j]);
    }
}
return liczby;
}

int suma(const vector<int> wektor) {
    int sumaCalkowita = 0;
    for (int liczba : wektor) { /*petla range for,
    liczba w kolejnych iteracjach przyjmuje kolejne
    wartosci w wektorze wektor*/
        sumaCalkowita += liczba;
    }
    return sumaCalkowita;
}

void wypisz(vector<int>podzbior)
{
    cout<<"[";
 for (int i = 0; i < podzbior.size(); i++) {
                cout << podzbior[i];
                if (i < podzbior.size() - 1) cout << ", ";
            }
    cout<<"]"<<endl;
}

void generator(const vector<int> liczby,int rozmiar,
vector <int> aktualny, int indeks,int sumaliczby)
{
    if(aktualny.size()>rozmiar)return; //warunek przerwania
    if(aktualny.size()==rozmiar){ /*analizujemy jedynie
    podzbiory o danym rozmiarze*/
        if(suma(aktualny)*2>sumaliczby){
        wyniki.push_back(aktualny);
        }
```

```
    }
    for(int i=indeks;i<liczby.size();++i){ /*indeks przesuwa
    rekurencje na dalsze elementy zbioru liczby*/
        aktualny.push_back(liczby[i]); //dodajemy liczbe
        generator(liczby,rozmiar,aktualny,i+1,sumaliczby);
        //rekurencyjnie tworzy drzewo kombinacji
        aktualny.pop_back(); //usuwa najmlodszy element
    }
}

void RozmiarNajmniejszyDominujacyPodzbior(
const vector<int>liczby, int SumaDoPrzebicia,
int rozmiarDoPrzebicia) {
int najw=0;
int rozmiarPod=0;
int sumaPodzbioru=0;
int pozNajw=0;
vector <int>pozostale=liczby;
for(int a=0;a<liczby.size();a++){
    najw=0;
    for (int j = 0; j < pozostale.size(); j++) {
        if(najw<pozostale[j]){najw=pozostale[j];pozNajw=j;}
    }
    if(2*sumaPodzbioru<=SumaDoPrzebicia){
        sumaPodzbioru+=najw;rozmiarPod++;rozmiarDoPrzebicia--;
        pozostale.erase(pozostale.begin()+pozNajw);
    }
    if(rozmiarDoPrzebicia<rozmiarPod){
        cout<<"Brak";rozmiarPod=0;break;
    }
    if(2*sumaPodzbioru>SumaDoPrzebicia){
        break;
    }
}
cout<<rozmiarPod<<"<--"<<endl;
//!ZNALEZIONO ROZMIAR NAJMNIESZEGO PODZIORU
/*teraz generujemy mozliwe podzbiory, a te
ktore spelniaja warunek dodajemy do wyniki*/
vector<int> aktualny;
generator(liczby,rozmiarPod,aktualny,0,suma(liczby));
//segment usuwania powtorek
set<vector<int>> unikalneWyniki(wyniki.begin(), wyniki.end());
/*wywolanie set przyjmuje jedynie unikalne rekordy
przez co znikaja powtorzenia*/
wyniki.assign(unikalneWyniki.begin(), unikalneWyniki.end());
//przywroceie wyniki do postaci wektora
for(int i=0;i<wyniki.size();i++)wypisz(wyniki[i]);
//wypisanie wynikow
}
```

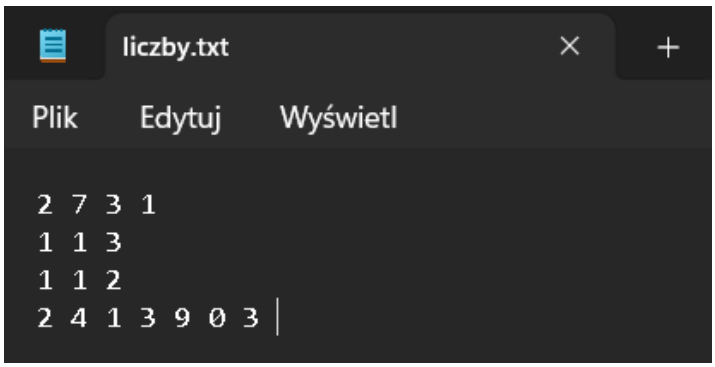
```
int main() {
    vector<int> wejscie = {9, 76, 3, 3, 100, 63, 9, 7,
    3, 63, 3, 1, 76, 5, 3, 2, 23, 4, 4, 6, 8, 5, 55, 33, 3, 6};
    cout<<suma(wejscie)<<endl; //suma elementow zbioru
    cout<<"Rozmiar najmniejszego ";
    wypisz(wejscie);
    cout<<endl;
    RozmiarNajmniejszyDominujacyPodzbior(powtorki(wejscie),
    suma(wejscie),wejscie.size());
    return 0;
}
```

W celu zmieszczenia całej zawartości w określonym formacie niektóre linijki kodu oraz zdania zostały celowo skrócone lub przerzucone do kolejnej linii. Lepsza wersja dostępna na githubie (OPTYMALNY.cpp).

3.8 Przykładowe wyniki i ich analiza

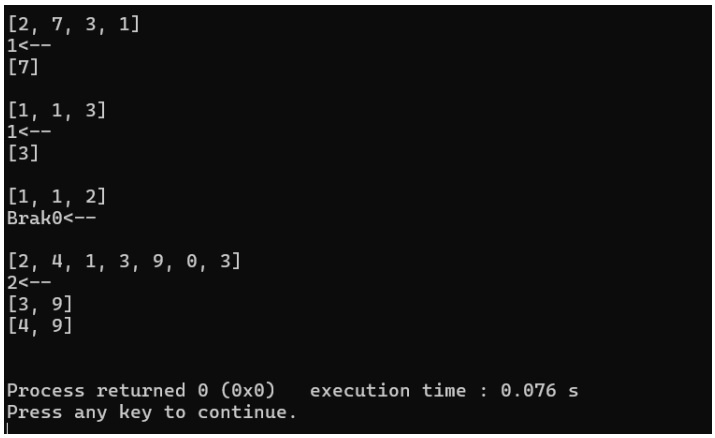
Do utworzenia poniższych wyników użyto innej wersji programu niż podanej wyżej. Wyniki wygenerował OPTYMLANYzPliku.cpp dostępny na GitHubie.

Przygotowujemy plik z przykładowymi danymi i zapisujemy go pod nazwą liczby.txt



Rysunek 7: Przykładowe dane w pliku .txt

Program zwraca dane:



Rysunek 8: Dane wyjściowe OPTYMALNY dla przykładowych danych

Program wypisuje także jaki rozmiar podzbiorów analizujemy. Dzięki nowym rozwiązaniom w wynikach nie otrzymujemy powtórek(rozwiązania zostały zaimplementowane także do bruteforce na GitHubie).Gdy już posiadamy oba kody możemy przejść do eksperymentalnego sprawdzenia ich wydajności.

4 Eksperymentalne porównanie kodów

W tym punkcie zawarte są:

- porównanie czasów wymagany do rozwiązania problemu
- test dla specyficznych danych
- graficzne przedstawienie wymaganego czasu obliczeń

4.1 Porównanie czasów

Po kilku testach utworzony został zbiór danych, który dobrze wizualizuje różnice w czasach obu algorytmów. Dane to:

- 10: 52 5 1 3 3 0 46 0 50 3
- 20: 9 1 6 3 2 96 8 3 0 1 2 101 1 4 8 2 7 3 2 1
- 23: 120 9 1 101 7 6 8 2 3 0 2 5 2 3 96 4 1 2 3 8 1 1 1
- 25: 9 3 1 3 7 8 2 0 8 120 1 2 2 1 3 96 8 7 5 4 6 101 1 2
- 26: 6 5 9 62 91 100 3 77 1 4 4 6 1 62 2 2 9 76 6 3 8 6 1 0 2 100
- 27: 6 5 9 62 91 100 3 77 1 4 4 6 1 62 2 2 9 76 6 3 8 6 1 0 2 100 7

Wynik dla BruteForce:

```
[52, 5, 1, 3, 3, 0, 46, 0, 50, 3]
[46, 50]
[52, 46]
[52, 50]
Zmierzony czas: 0.0000000000 seconds.

[9, 1, 6, 3, 2, 96, 8, 3, 0, 1, 2, 101, 1, 4, 8, 2, 7, 3, 2, 1]
[96, 101]
Zmierzony czas: 1.9016500000 seconds.

[120, 9, 1, 101, 7, 6, 8, 2, 3, 0, 2, 5, 2, 3, 96, 4, 1, 2, 3, 8, 1, 1, 1]
[96, 101]
[120, 96]
[120, 101]
Zmierzony czas: 18.5509350000 seconds.

[1, 9, 3, 1, 3, 7, 8, 2, 0, 8, 120, 1, 2, 2, 1, 3, 96, 8, 7, 5, 4, 6, 101, 1, 2]
[96, 120]
[120, 101]
Zmierzony czas: 77.1061860000 seconds.

[6, 5, 9, 62, 91, 100, 3, 77, 1, 4, 4, 6, 1, 62, 2, 2, 9, 76, 6, 3, 8, 6, 1, 0, 2, 100]
[91, 100, 62, 76]
[91, 100, 77, 62]
[91, 100, 77, 76]
[91, 100, 100, 62]
[91, 100, 100, 76]
[91, 100, 100, 77]
[100, 100, 62, 62]
[100, 100, 62, 76]
[100, 100, 77, 62]
[100, 100, 77, 76]
Zmierzony czas: 144.8868770000 seconds.

[6, 5, 9, 62, 91, 100, 3, 77, 1, 4, 4, 6, 1, 62, 2, 2, 9, 76, 6, 3, 8, 6, 1, 0, 2, 100, 7]
[91, 100, 62, 76]
[91, 100, 77, 62]
[91, 100, 77, 76]
[91, 100, 100, 62]
[91, 100, 100, 76]
[91, 100, 100, 77]
[100, 100, 62, 76]
[100, 100, 77, 62]
[100, 100, 77, 76]
Zmierzony czas: 316.5183210000 seconds.
```

Rysunek 9: Mierzenie czasu algorytmu Bruteforce

Wyniki dla Optymalnego:

```
[52, 5, 1, 3, 3, 0, 46, 0, 50, 3]
2<--
[46, 50]
[52, 46]
[52, 50]
Zmierzony czas: 0.0000000000 seconds.

[9, 1, 6, 3, 2, 96, 8, 3, 0, 1, 2, 101, 1, 4, 8, 2, 7, 3, 2, 1]
2<--
[96, 101]
Zmierzony czas: 0.0000000000 seconds.

[120, 9, 1, 101, 7, 6, 8, 2, 3, 0, 2, 5, 2, 3, 96, 4, 1, 2, 3, 8, 1, 1, 1]
2<--
[96, 101]
[120, 96]
[120, 101]
Zmierzony czas: 0.0000000000 seconds.

[1, 9, 3, 1, 3, 7, 8, 2, 0, 8, 120, 1, 2, 2, 1, 3, 96, 8, 7, 5, 4, 6, 101, 1, 2]
2<--
[96, 120]
[120, 101]
Zmierzony czas: 0.0000000000 seconds.

[6, 5, 9, 62, 91, 100, 3, 77, 1, 4, 4, 6, 1, 62, 2, 2, 9, 76, 6, 3, 8, 6, 1, 0, 2, 100]
4<--
[91, 100, 62, 76]
[91, 100, 77, 62]
[91, 100, 77, 76]
[91, 100, 100, 62]
[91, 100, 100, 76]
[91, 100, 100, 77]
[100, 100, 62, 62]
[100, 100, 62, 76]
[100, 100, 77, 62]
[100, 100, 77, 76]
Zmierzony czas: 0.0629360000 seconds.

[6, 5, 9, 62, 91, 100, 3, 77, 1, 4, 4, 6, 1, 62, 2, 2, 9, 76, 6, 3, 8, 6, 1, 0, 2, 100, 7]
4<--
[91, 100, 62, 76]
[91, 100, 77, 62]
[91, 100, 77, 76]
[91, 100, 100, 62]
[91, 100, 100, 76]
[91, 100, 100, 77]
[100, 100, 62, 76]
[100, 100, 77, 62]
[100, 100, 77, 76]
Zmierzony czas: 0.0787870000 seconds.
```

Rysunek 10: Mierzenie czasu algorytmu Optymalny

Wnioski

Wyświetlany czas pokazuje 10 zer po przecinku. Jeśli czas pokazuje same zera oznacza to, że do rozwiązania problemu dla danego zbioru danych, był mniejszy niż nanosekunda.

Problemy z algorytmem Bruteforce widać na pierwszy rzut oka. Wraz z wzrostem rozmiaru zbioru wzrasta czas wymagany na otrzymanie wyniku. Dla rozmiaru 27 jest to już ponad 5 minut.

Coś w co ciężko uwierzyć to czas jaki algorytm optymalny osiąga, dla tych samych danych. Właśnie tutaj widać jego przewagę nad poprzednikiem.

Algorytm Bruteforce jest ściśle związany z rozmiarem zbioru. Natomiast algorytm optymalny opiera swój czas o rozmiar podzbioru, który spełnia warunek problemu. Różnica czasów między zbiorem 27 i 26 elementowym dla Bruteforce to aż około 172 sekundy. Optymalny dla tych samych zbiorów podaje odpowiedź w zbliżonym czasie (jest tak ponieważ te zbiory posiadają taki sam rozmiar najmniejszego podzbioru spełniającego warunek).

4.2 Specjalne przypadki

Algorytm optymalny spisuje się dobrze dla zbiorów, których oczekiwany wynik to mały podzbiór. Wraz z wzrostem rozmiaru podzbioru wyniku, czas jego działania rośnie. Oto zestaw danych obrazujący wrażliwości kodu optymalnego:

- 16 zer i 14 dziewiątek: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9 9 9 9 9 9 9 9 9 9
- 18 zer i 16 dziewiątek: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9 9 9 9 9 9 9 9 9 9 9 9

Wynik dla powyższych danych:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
8<--
[9, 9, 9, 9, 9, 9, 9, 9]
Zmierzony czas: 17.1855350000 seconds.

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
9<--
[9, 9, 9, 9, 9, 9, 9, 9, 9]
Zmierzony czas: 162.0231910000 seconds.
```

Rysunek 11: Wyjątkowe przypadki algorytmu optymalnego

Mimo że teoretycznie rozmiar najmniejszego z podzbiorów zwiększył się tylko o 1, to zmierzony czas wzrósł około 10-krotnie. Czy ma to "poparcie w liczbach"?

Pierwszy zestaw to 30-elementowy zbiór, z czego wybieramy kombinacje 8-elementowe daje to:

$$C(30, 8) = \frac{30!}{8!(30-8)!} = 5852925$$

Drugi zestaw to 34-elementowy zbiór, z czego wybieramy kombinacje 9-elementowe daje to:

$$C(34, 9) = \frac{34!}{9!(34-9)!} = 52451256$$

Rzeczywiście w dużym przybliżeniu drugi wynik jest 10-razy większy od poprzedniego.

4.3 Wykres dla algorytmu bruteforce

Z racji ścisłego związania algorytmu bruteforce z rozmiarem zbioru, czas wymagany aby rozwiązać problem można przedstawić następująco:

Rysunek 12: Wykres stosunku rozmiaru zbioru do czasu działania algorytmów

Algoryty dobrze radzą sobie z małymi zbiorami. Niestety dla Bruteforce, gdy rozmiar zbioru przekracza liczbę 30, często program uznaje, że czas na wykonanie algorytmu mija i wyrzuca błąd(na wykresie zawarto wyniki algorytmu bruteforce, które udało się rzetelnie zmierzyć) Patrząc na dane dla n z przedziału <11;34> można wyciągnąć pewne wnioski:

- Czas wymagany do rozwiązania problemu dla algorytmu optymalnego zaczyna wzrastać później
- Skos niebieskiego wykresu jest mniejszy niż pomarańczowego
- Oba wykresu wzrastają wykładniczo

5 Podsumowanie

Zgodnie z nazwą algorytm optymalny cechuje się niższymi czasami wymaganymi, aby rozwiązać problem dla danego zbioru danych niż jego poprzednik. Ważną różnicą między tymi algorytmami są rozmiary n na jakich się opierają. Dla bruteforce zawsze będzie to n-rozmiar zbioru, a dla optymalnego będzie to zależne od rozmiaru najmniejszego zbioru spełniającego warunki (maksymalnie $\frac{n}{2} - 1$).

Link do GitHub'a związanego z dokumentem : `https://github.com/qertop/Projekt.git`