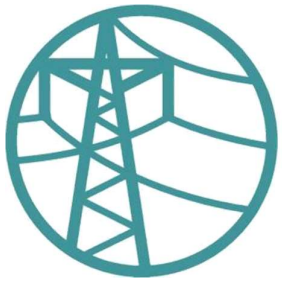


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Dariusz Strojny

Algorytmy i Struktury Danych

Projekt zaliczeniowy nr 1

Rzeszów, 2022

1 Spis treści

2 Temat

Zadanie 10.

Sprawdź, które elementy tablicy dwuwymiarowej występują w każdym wierszu tej tablicy.

2.1 Przykład:

Wejście:

[2,4,3,8,7]
[4,7,1,3,6]
[3,5,2,1,3]
[4,5,0,2,3]

Wyjście: 3

3 Analiza, projektowanie

3.1 Zasada działania programu

Program ten ma za zadanie znalezienie liczb powtarzających się w każdym wierszu znajdującym się w tabeli dwuwymiarowej. Dane muszą zostać wylosowane do tabeli w związku z czym musimy wprowadzić dane odpowiedzialne za rozmiar naszej tablicy. Dane te zostają wylosowane z przedziału podanego w kodzie programu.

3.2 Struktury danych

Dane przechowywane są w tabeli dwuwymiarowej składającej się z n * m elementów, gdzie n to ilość wierszy a m to ilość kolumn. Zakres danych, które będą losowane do naszej tabeli, m i n są liczbami dodatnimi, całkowitymi, niemniejszymi od zera. Użycie ograniczonej n*m-elementowej tablicy pozwoli nam na zwiększenie wydajności działania naszego programu oraz ograniczy możliwość popełnienia błędów mogących pojawić się podczas pracy na tych danych.

3.3 Metodyka

3.3.1 Definicje:

• PRO_FILE_VALUE_DELIMITER ' '
Domyślny znak oddzielający wartości w plikach tekstowych

• PRO_FILE_ARRAY_DELIMITER '\n'
Domyślny znak oddzielający wiersze w plikach tekstowych

3.3.2 Funkcje:

• void pro::**init** ()

Inicjalizuje bibliotekę pomocniczą.

• int pro::**losowa_liczba** (int min, int max)

Generuje losową liczbę z przedziału [min, max].

Parametry

min Minimalna wartość liczby

max Maksymalna wartość liczby

Zwraca

wygenerowana liczba

• std::vector< int >
pro::**generuj_losowy_ciag** (int min, int max, int width)

Generuje losowy ciąg o podanej długości z wartościami z podanego przedziału.

Parametry

min - Minimalna wartość elementu w ciągu

max - Maksymalna wartość elementu w ciągu

width - Ilość elementów w ciągu

Zwraca

wygenerowany ciąg

• std::vector< std::vector< int > >

pro::**generuj_losowy_ciag_2d** (int min, int max, int width, int height)

Generuje losowy dwuwymiarowy ciąg o podanych wymiarach z wartościami z podanego przedziału.

Parametry

min - Minimalna wartość elementu w ciągu

max - Maksymalna wartość elementu w ciągu

width - Ilość kolumn w ciągu

height - Ilość wierszy ciągu

Zwraca

wygenerowany ciąg

• std::vector< int >

pro::**generuj_ciag_z_zakresu** (int start, int end, int step=1)

Zwraca ciąg z zakresu start do end z krokiem step.

np. f(2, 6, 2) -> [2, 4, 6]

Parametry

start - Początkowa wartość iteratora

end - Maksymalna wartość iteratora (włącznie)

step - Krok o jaki zwiększany jest iterator

• std::pair< std::vector< int >::iterator, std::vector< int >::iterator >
pro::**quicksort_iterator_three_way_partition** (std::vector< int >::iterator start, std::vector< int >::iterator end)

Funkcja pomocnicza sortowania quicksort wykorzystująca usprawnienie dla ciągów z często powtarzającymi się wartościami.

Parametry

begin – Iterator wskazujący na początek zakresu do posortowania

end – Iterator wskazujący na koniec zakresu do posortowania

Zwraca

Para iteratorów wskazujących odpowiednio na koniec i początek przedziałów oddzielonych ciągiem złożonym z wartości równych wybranej wartości pivot.

• void pro::**quicksort_three_way_iterator** (std::vector< int >::iterator begin, std::vector< int >::iterator end)

Sortowanie metodą quicksort wykorzystujące usprawnienie dla ciągów z często powtarzającymi się wartościami na podanym przedziale.

Parametry

begin - Iterator wskazujący na początek przedziału

end - Iterator wskazujący na koniec przedziału

• std::vector< int >::iterator
pro::**linear_search_iterator** (std::vector< int > &arr, int val)

Przeprowadza wyszukiwanie liniowe w wartościach tablicy.

Parametry

arr - Tablica, na której wykonywane jest wyszukiwanie

val - Wartość szukana w tablicy

Zwraca

Iterator wskazujący na znaleziony element lub na koniec przedziału

• std::vector< int >::iterator
pro::**set_intersection** (const std::vector< int > &arr1,
const std::vector< int > &arr2, std::vector< int >::iterator res)

Wyszukuje wspólne elementy dwóch tablic.

Funkcja wykonuje wyszukiwanie wspólnych elementów poprzez skrzyżowanie ze sobą dwóch tablic. Tablice wejściowe muszą być posortowane rosnąco.

Parametry

arr1 - Pierwsza tablica

arr2 - Druga tablica

res – Referencja na iterator wskazujący na pierwszy element tablicy o rozmiarze przynajmniej min(rozmiar arr1, rozmiar arr2)

Zwraca

Iterator wskazujący na element za ostatnim wpisanym elementem

• void pro::**opisz_ciag** (const std::vector< int > &arr)

Wypisuje w konsoli wymiary tablicy.

Parametry

arr - Opisywana tablica

• void pro::**opisz_ciag** (const std::vector< std::vector< int > > &arr)

Wypisuje w konsoli wymiary tablicy dwuwymiarowej.

Parametry

arr - Opisywana tablica

• std::vector< int >
pro::**odczytaj_ciag_z_pliku** (const char *nazwa_pliku,
char delimiter = PRO_FILE_VALUE_DELIMITER)

Odczytuje ciąg z pliku wejściowego.

Parametry

nazwa_pliku - Ścieżka do pliku

delimiter - Znak oddzielający wartości w pliku

Zwraca

Ciąg odczytany z pliku

• std::vector< std::vector< int > >
pro::**odczytaj_ciag_2d_z_pliku** (const char ∗nazwa_pliku,
char delimiter_val = PRO_FILE_VALUE_DELIMITER,
char delimiter_array = PRO_FILE_ARRAY_DELIMITER)

Odczytuje dwuwymiarową tablicę z pliku wejściowego.

Parametry

nazwa_pliku - Ścieżka do pliku

delimiter_val - Znak oddzielający wartości wiersza w pliku

delimiter_array - Znak oddzielający wiersze w pliku

Zwraca

Dwuwymiarowa tablica odczytana z pliku

• std::pair< std::vector< std::vector< int > >::const_iterator,
std::vector< std::vector< int > >::const_iterator >
pro::**thread_bounds** (const std::vector< std::vector< int > > &data, int
thread_count, int thread_id)

Oblicza zakres danych, na których mają być wykonane operacje dla podanego wątku.

Parametry

data - Dane do podzielenia

thread_count - Łączna ilość wątków

thread_id - Numer wątku, dla którego obliczany jest zakres

Zwraca

Para iteratorów wskazujących na początek i koniec wyznaczonego zakresu danych

• template<class T>
void pro::**wypisz_ciag** (const std::vector< T > &arr, unsigned spacing=0)

Wypisuje zawartość tablicy na ekranie.

Parametry Szablonu

T - Rodzaj danych przechowywanych w tablicy

Parametry

arr - Tablica do wyświetlenia

spacing - Dopełnienie każdej komórki danych znakami białymi do podanej ilości znaków

• template<class T >
void pro::**wypisz_ciag** (const std::vector< std::vector< T > > &data,
unsigned spacing=0)

Wypisuje zawartość tablicy dwuwymiarowej na ekranie.

Parametry Szablonu

T - Rodzaj danych przechowywanych w tablicy

Parametry

data - Tablica do wyświetlenia

spacing - Dopełnienie każdej komórki danych znakami białymi do podanej ilości znaków

• template<class T >
void pro::**zapisz_ciag_do_pliku** (const char ∗nazwa_pliku,
const std::vector< T > &data, char delimiter = PRO_FILE_VALUE_DELIMITER)

Zapisuje ciąg do pliku wyjściowego.

Parametry Szablonu

T - Rodzaj danych przechowywanych w ciągu

Parametry

nazwa_pliku - Ścieżka do pliku

data - Ciąg do zapisania

delimiter - Znak oddzielający wartości w pliku

• template<class T >
void pro::**zapisz_ciag_2d_do_pliku** (const char ∗nazwa_pliku,
const std::vector< std::vector< T > > &data,
char delimiter_val=PRO_FILE_VALUE_DELIMITER,
char delimiter_array=PRO_FILE_ARRAY_DELIMITER)

Zapisuje tablicę dwuwymiarową do pliku wyjściowego

Parametry Szablonu

T - Rodzaj danych przechowywanych w tablicy

Parametry

nazwa_pliku - Ścieżka do pliku

data - Tablica do zapisania

delimiter_val - Znak oddzielający wartości wiersza w pliku

delimiter_array - Znak oddzielający wiersze w pliku

3.3.3 Główne funkcje algorytmów

• std::vector<int>

znajdz_powtorzenia_a(
const std::vector<std::vector<int>>::const_iterator& data_first,
const std::vector<std::vector<int>>::const_iterator& data_last);

Funkcja implementująca realizację naiwną algorytmu z treści zadania

Funkcja ta nie modyfikuje danych wejściowych przez co mogą one bezpiecznie zostać użyte po jej wywołaniu

Parametry

data_first – Iterator wskazujący na początek zakresu zawierającego wiersze wejściowe

data_last – Iterator wskazujący na koniec zakresu zawierającego wiersze wejściowe

Zwraca

Tablica wartości występujących w każdym z wprowadzonych wierszy

• std::vector<int>

znajdz_powtorzenia_b(
const std::vector<std::vector<int>>::const_iterator& data_first,
const std::vector<std::vector<int>>::const_iterator& data_last);

Funkcja implementująca realizację sprytniejszą algorytmu z treści zadania

Funkcja ta nie modyfikuje danych wejściowych przez co mogą one bezpiecznie zostać użyte po jej wywołaniu

Parametry

data_first – Iterator wskazujący na początek zakresu zawierającego wiersze wejściowe

data_last – Iterator wskazujący na koniec zakresu zawierającego wiersze wejściowe

Zwraca

Tablica wartości występujących w każdym z wprowadzonych wierszy

4 Opis działania algorytmu naiwnego

Jako algorytm naiwny w podanym zadaniu wykorzystane zostało skopiowanie elementów pierwszego wiersza do nowej tablicy powtórzeń a następnie w trakcie iteracji przez wszystkie pozostałe wiersze zmniejszanie jej zawartości o elementy nie występujące w danym wierszu do momentu uzyskania pustej tablicy powtórzeń lub zakończenia iteracji.

W celu uniknięcia wielokrotnego przepisywania elementów z tablicy powtórzeń, każdy znaleziony element zastąpiony zostaje pustym znacznikiem, którego wartość jest mniejsza niż najmniejszy element w pierwszym wierszu a każdy element występujący w iterowanej tablicy, który jest równy wartością znacznikowi, jest pomijany.

Wynikiem działania tego algorytmu jest tablica zawierająca wszystkie elementy powtarzające się w każdym wierszu.

4.1 Pseudokod

4.1.1 Pseudokod algorytmu wyszukiwania liniowego

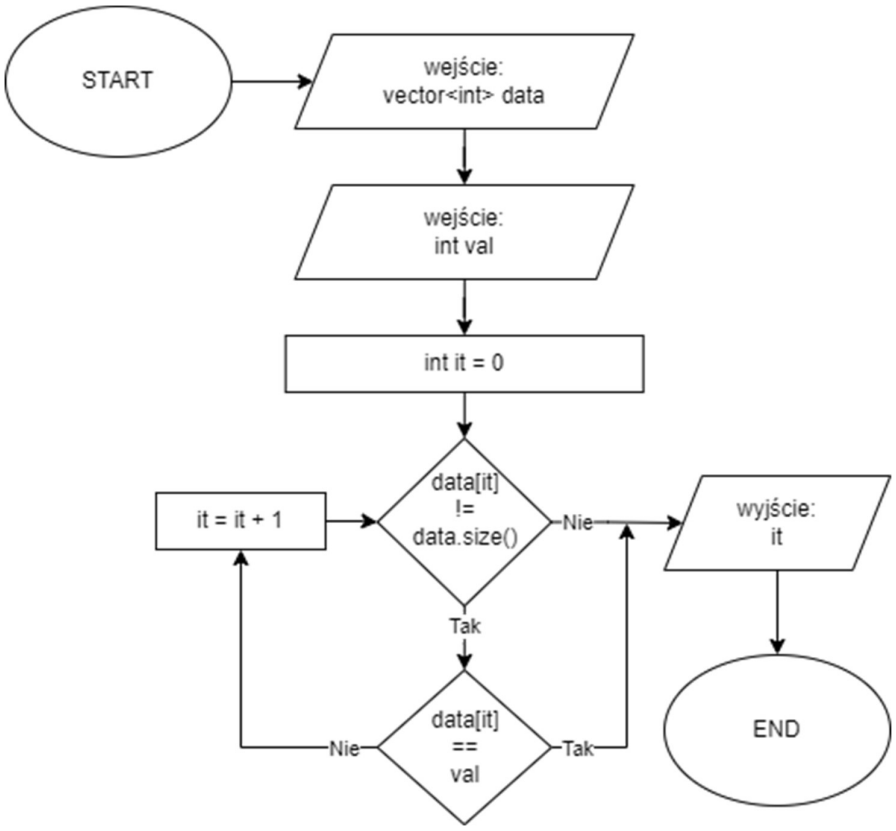
```
wejście:
data              - tablica danych wejściowych
val               - wartość szukana

dane:
it                - iterator przechodzący przez wszystkie elementy
                  tablicy data


algorytm:

it <- 0
dopóki data[it] != rozmiar tablicy data:
      jeżeli data[it] == val:
            zwróć wartość iteratora it
      it <- it + 1
zwróć wartość iteratora it
```

4.1.2 Pseudokod realizacji naiwnej algorytmu

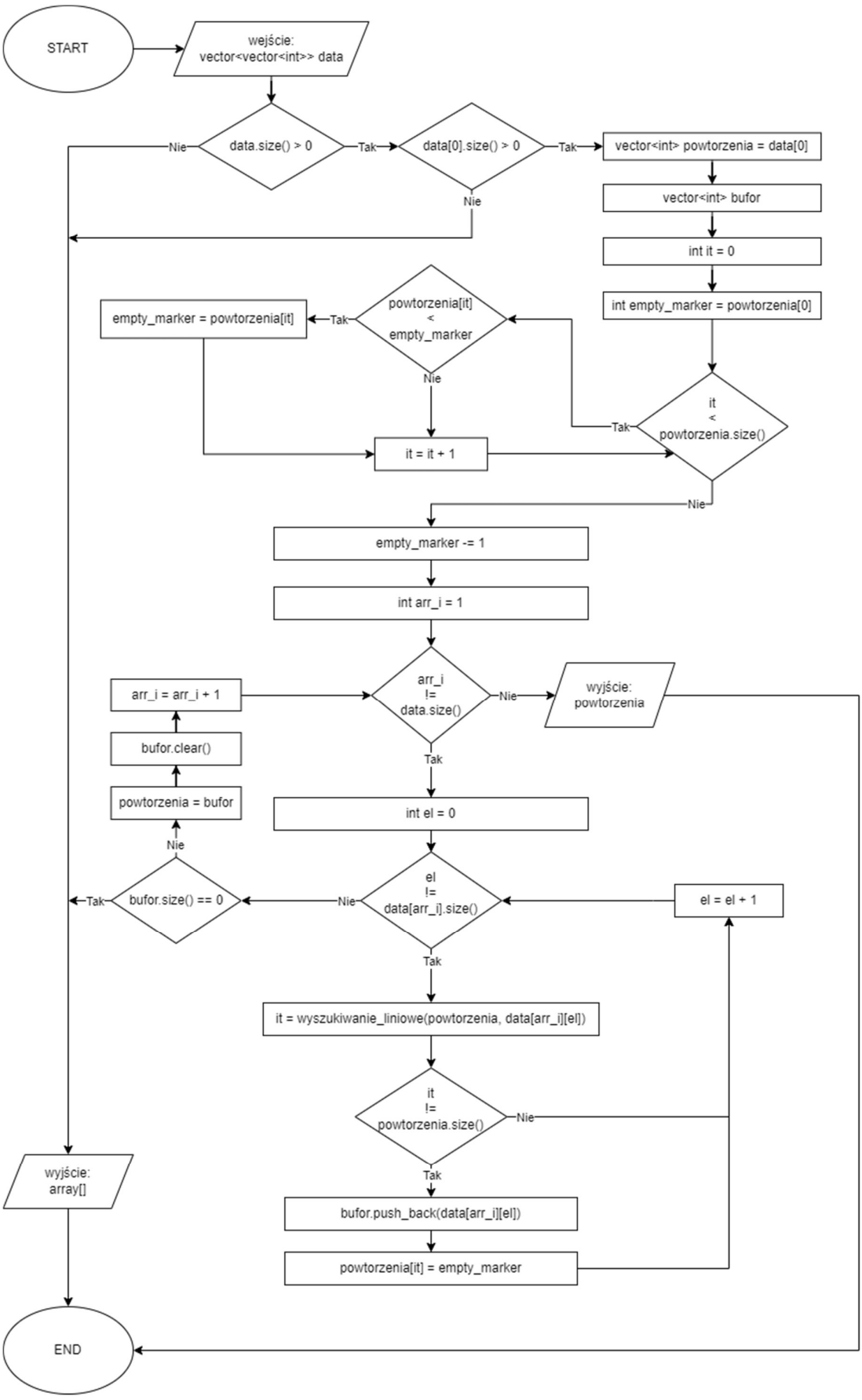
```
wejście:

data              - tablica dwuwymiarowa

dane:

powtorzenia       - tablica przechowująca powtorzenia z wszystkich
                  iteracji
bufor             - tablica przechowująca powtórzenia z aktualnej
                  iteracji
empty_marker      - liczba oznaczająca wartość usuniętą z tablicy
arr_i             - iterator przechodzący przez podciągi tablicy
                  data
el                - iterator przechodzący przez elementy podciągu
                  pod indeksem data
it                - iterator pomocniczy

algorytm:

jeżeli rozmiar data == 0 albo rozmiar data[0] == 0:
      zwróć pustą tablicę

powtorzenia <- data[0]
it <- 0
empty_marker <- powtorzenia[0]

dopóki it < od rozmiaru tablicy powtorzenia:
      jeżeli powtorzenia[it] < empty_marker:
            empty_marker <- powtorzenia[it]
      it <- it + 1
empty_marker <- empty_marker - 1

arr_i <- 1
dopóki arr_i != długość tablicy data:
      el <- 0
      dopóki el != długość tablicy data[arr_i]:
            it <- wyszukiwanie_liniowe(powtorzenia,
data[arr_i][el])
            jeżeli it != długość tablicy powtorzenia:
                  bufor <- bufor, data[arr_i][el]
                  powtorzenia[it] = empty_marker
            el <- el + 1
      jeżeli długość tablicy bufor == 0:
            zwróć pustą tablicę
      powtorzenia <- bufor
      bufor <- []
      arr_i <- arr_i + 1
zwróć tablicę powtorzenia
```

4.2 Schemat blokowy

4.2.1 Schemat blokowy algorytmu wyszukiwania liniowego



4.2.2 Schemat blokowy realizacji naiwnej algorytmu



4.3 Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu szukania liniowego wynosi O(n)

Algorytm posiada najlepszą złożoność obliczeniową O(1) dla zbioru pustego lub w przypadku poprawnych danych wejściowych O(n^2) dla tabeli, w której dane w pierwszym wierszu różnią się całkowicie od danych w drugim wierszu.

Złożoność obliczeniowa algorytmu w najgorszym przypadku (Gdy w każdym wierszu występują dokładnie te same wartości) jest równa O(n^3), ponieważ wykonywana jest iteracja przez ilość wierszy, ilość kolumn i ilość danych w tabeli porównań.

5 Opis działania algorytmu sprytniejszego

Algorytm sprytniejszy powstał w oparciu o wykorzystanie połączenia sortowania Quicksort z usprawnieniem dla często powtarzających się wartości (DNF problem) oraz algorytmu szukania części wspólnej dwóch tablic poprzez ich intersekcję.

Wykonanie algorytmu polega na posortowaniu pierwszego wiersza metodą Quicksort i zapisaniu go jako tablicy powtórzeń a następnie iterując przez każdy kolejny wiersz przypisywaniu do niej wyniku działania algorytmu intersekcji z nią samą i aktualnym wierszem, po posortowaniu go tą samą metodą, do momentu uzyskania pustej tablicy powtórzeń lub zakończenia iteracji.

Wynikiem działania algorytmu jest tablica zawierająca wszystkie elementy powtarzające się w każdym wierszu.

Algorytm Quicksort polega na sortowaniu tablicy przez wybranie pewnej wartości (tzw. pivot) znajdującej się gdziekolwiek na przedziale danych wejściowych, przeniesienie elementów mniejszych od niej na jej lewą stronę a większych na prawą po czym wykonanie tych samych operacji na powstałych w ten sposób nieposortowanych przedziałach, rozdzielonych tą wartością.

Problem DNF (z ang. Dutch National Flag) polega na tym, że dzieląc w ten sposób zbiór często powtarzających się wartości, dla każdej z nich operacja przenoszenia wykonuje się ze złożonością czasową O(n^2), co spowodowane jest porównywaniem elementów o tej samej wartości. Rozwiązaniem problemu DNF jest alternatywny sposób dzielenia przedziałów w tablicy. Zamiast dzielić elementy na mniejsze i większe od wartości pivot, dzielimy je na mniejsze, równe lub większe, poprzez zamianę porównywanego elementu z pierwszym lub ostatnim elementem aktualnego zakresu i zmniejszenie rozmiaru zakresu o jeden z prawej lub lewej, w zależności od tego, gdzie daliśmy porównywany element. Ciągi, które są sortowane w następnej iteracji po wykonaniu takich działań znajdują się w przedziale od początku tablicy do pierwszego elementu równego wartością elementowi pivot oraz od pierwszego elementu większego od wartości pivot do końca tablicy.

Algorytm intersekcji służy do znajdywania części wspólnej dwóch posortowanych ciągów. Działanie tego algorytmu polega na ustawieniu dwóch iteratorów na wartościach początkowych tablic i dopóki wartości, które te iteratory wskazują są różne, zmniejszanie tego z nich, który wskazuje mniejszy element. Jeżeli obie wartości są takie same, wartość wskazywana przez którykolwiek z nich jest zapisywana do tablicy wyników a oba iteratory

są inkrementowane. Algorytm wykonuje się, dopóki którykolwiek z iteratorów nie dotrze do końca swojej tablicy.

5.1 Pseudokod

5.1.1 Pseudokod algorytmu sortowania quicksort

Funkcja odpowiedzialna za wybieranie fragmentów dzielonych do sortowania:

```
*** Algorytm 1 - quicksort

wejście:

data              - wskaźnik na tablicę danych do posortowania

dane:

a                 - iterator wskazujący na początek posortowanego
                  zakresu
b                 - iterator wskazujący na pierwszy element za
                  posortowanym zakresem

algorytm:

jeżeli rozmiar tablicy data <= 1:
      zakończ działanie algorytmu

jeżeli rozmiar tablicy data == 2:
      jeżeli data[0] > data[1]:
            data[0] <-> data[1]
      zakończ działanie algorytmu

a, b <- quicksort_partition(data)

quicksort(podciąg tablicy data w zakresie od 0 do a)
quicksort(podciąg tablicy data w zakresie od b do rozmiaru tablicy
data)

zakończ działanie algorytmu
```

Funkcja odpowiedzialna, za posortowanie wartości na podanym przedziale względem wartości pivot:

```
*** Algorytm 2 - quicksort_partition

wejście:
data              - wskaźnik na fragment tablicy do posortowania

dane:
mid               - iterator wskazujący na porównywany element
start             - iterator wskazujący na początek nieposortowanego
podciągu
end               - iterator wskazujący na koniec nieposortowanego
podciągu
pivot             - wartość, z którą porównywane są elementy do
sortowania

algorytm:

mid <- 0
start <- 0
end <- długość tablicy data
pivot <- data[end - 1]

dopóki mid != end:
      jeżeli data[mid] < pivot:
            data[start] <-> data[mid]
            start <- start + 1
            mid <- mid + 1
      w przeciwnym wypadku, jeżeli data[mid] > pivot:
            data[mid] <-> data[end - 1]
            end <- end - 1
      w przeciwnym wypadku:
            mid <- mid + 1

zakończ działanie algorytmu
```

5.1.2 Pseudokod algorytmu intersekcji

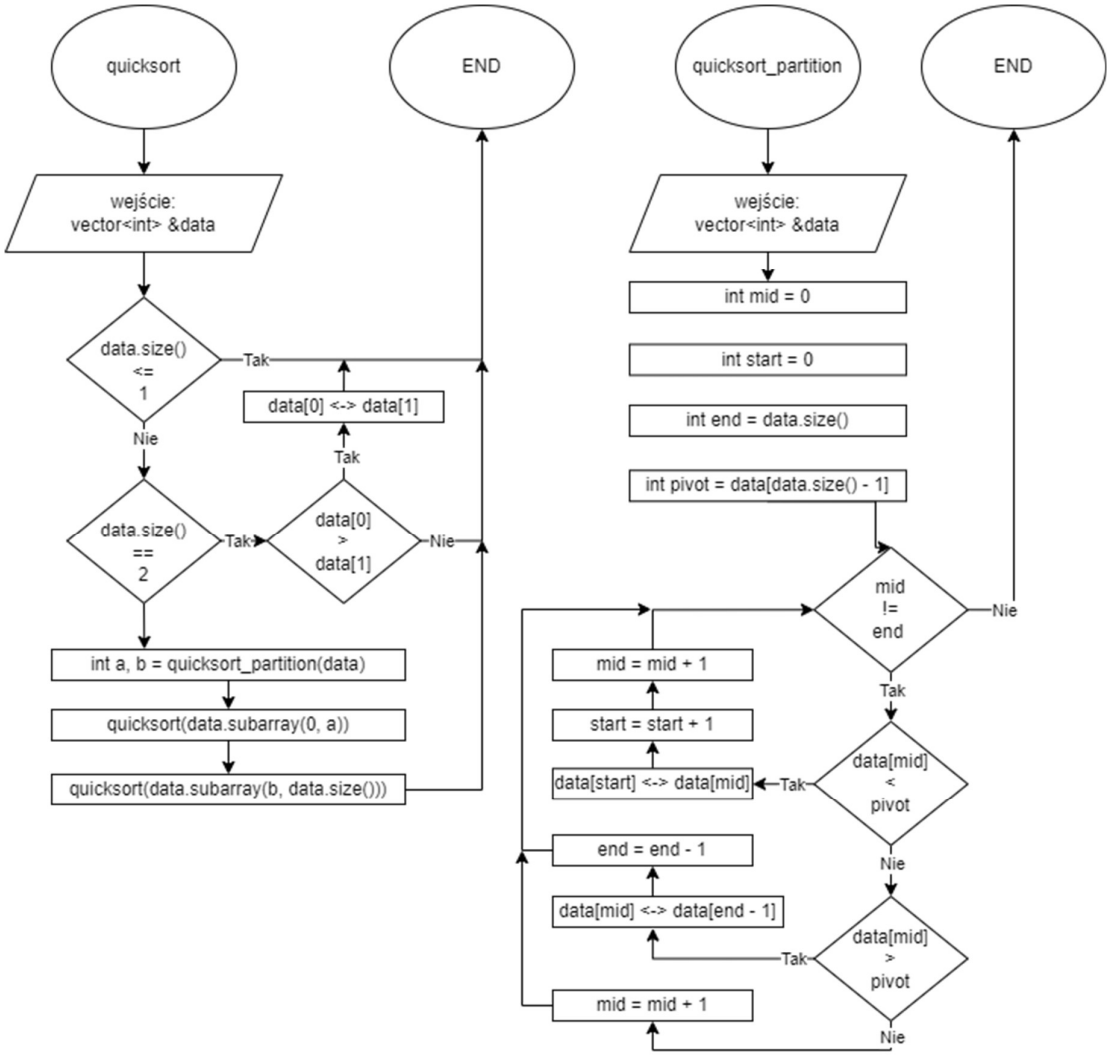
```
wejście:

arr1              - pierwsza tablica wejściowa
arr2              - druga tablica wejściowa
res               - wskaźnik na pustą tablicę o rozmiarze przynajmniej
                  min(długość arr1, długość arr2),
                        w której znajdują się zwracane wartości

dane:

res_i       - licznik elementów w zwracanej tablicy
first1      - iterator przechodzący przez tablicę arr1
last1             - zmienna przechowująca długość tablicy arr1
first2      - iterator przechodzący przez tabicę arr2
last2             - zmienna przechowująca długosć tablicy arr2

algorytm:

res_i <- 0
first1 <- 0
last1 <- długość tablicy arr1
first2 <- 0
last2 <- długość tablicy arr2

dopóki first1 != last1:
      jeżeli arr1[first1] < arr2[first2]:
            first1 <- first1 + 1
      w przeciwnym wypadku, jeżeli arr2[first2] < arr1[first1]:
            first2 <- first2 + 1
      w przeciwnym wypadku:
            res[res_i] <- arr1[first1]
            res_i <- res_i + 1
            first1 <- first1 + 1
            first2 <- first2 + 1
zwróć wartość licznika res_i
```

5.1.3 Pseudokod realizacji sprytniejszej algorytmu

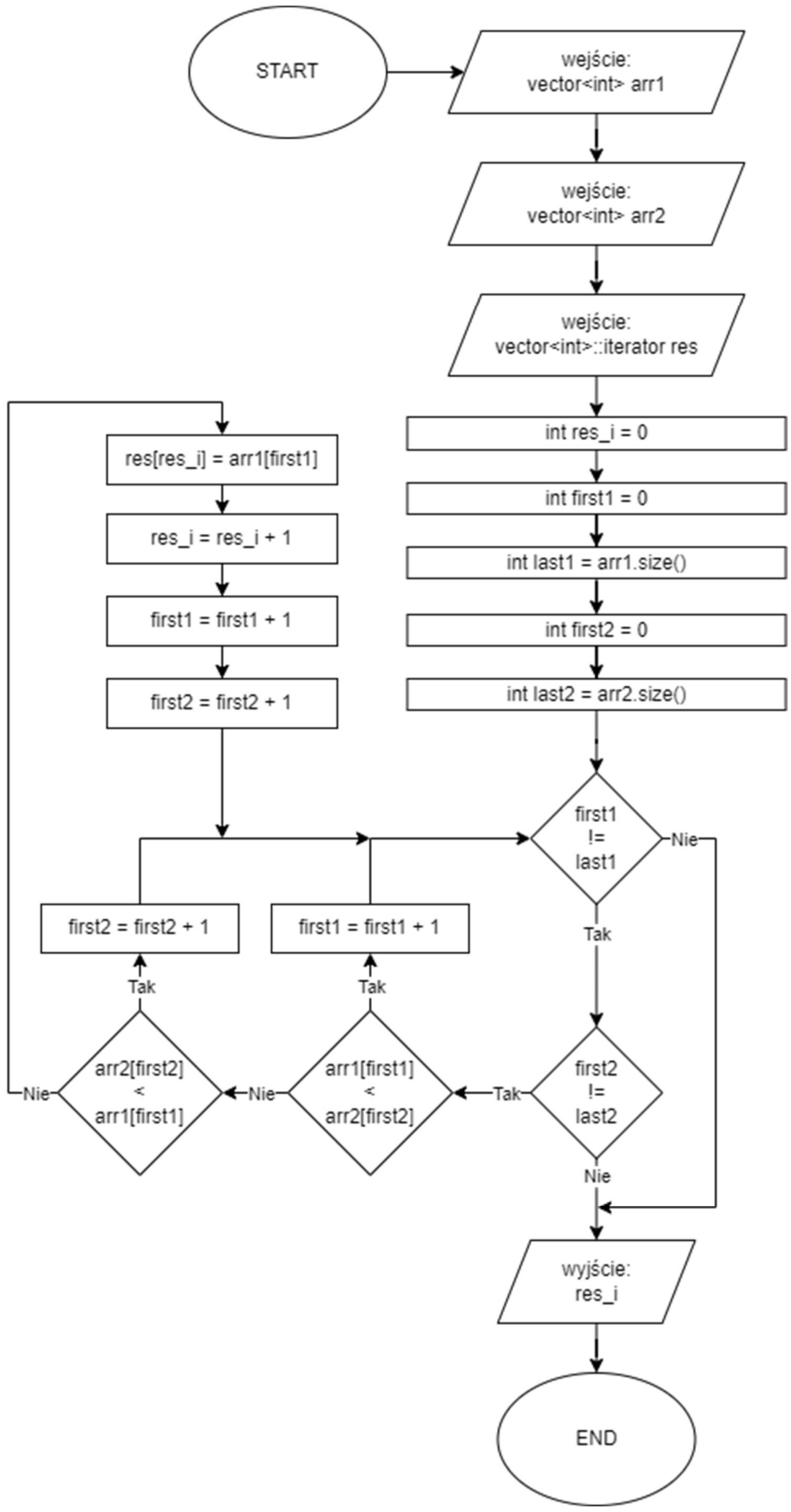
```
wejście:
data              - tablica dwuwymiarowa

dane:

powtorzenia       - tablica przechowująca powtorzenia z wszystkich
                  iteracji
bufor             - tablica przechowująca powtórzenia z aktualnej
                  iteracji
arr_i             - iterator przechodzący przez podciągi tablicy data
it                - iterator pomocniczy

algorytm:

jeżeli rozmiar data == 0 albo rozmiar data[0] == 0:
      zwróć pustą tablicę

powtorzenia <- data[0]
quicksort(powtorzenia)

arr_i <- 1
dopóki arr_i != długość tablicy data:
      el <- 0
      quicksort(data[arr_i])
      it = intersection(powtorzenia, data[arr_i], &bufor)
      if rozmiar tablicy bufor == 0:
            zwróć pustą tablicę
      powtórzenia <- bufor
      arr_i <- arr_i + 1
zwróć tablicę powtórzenia
```

5.2 Schemat blokowy

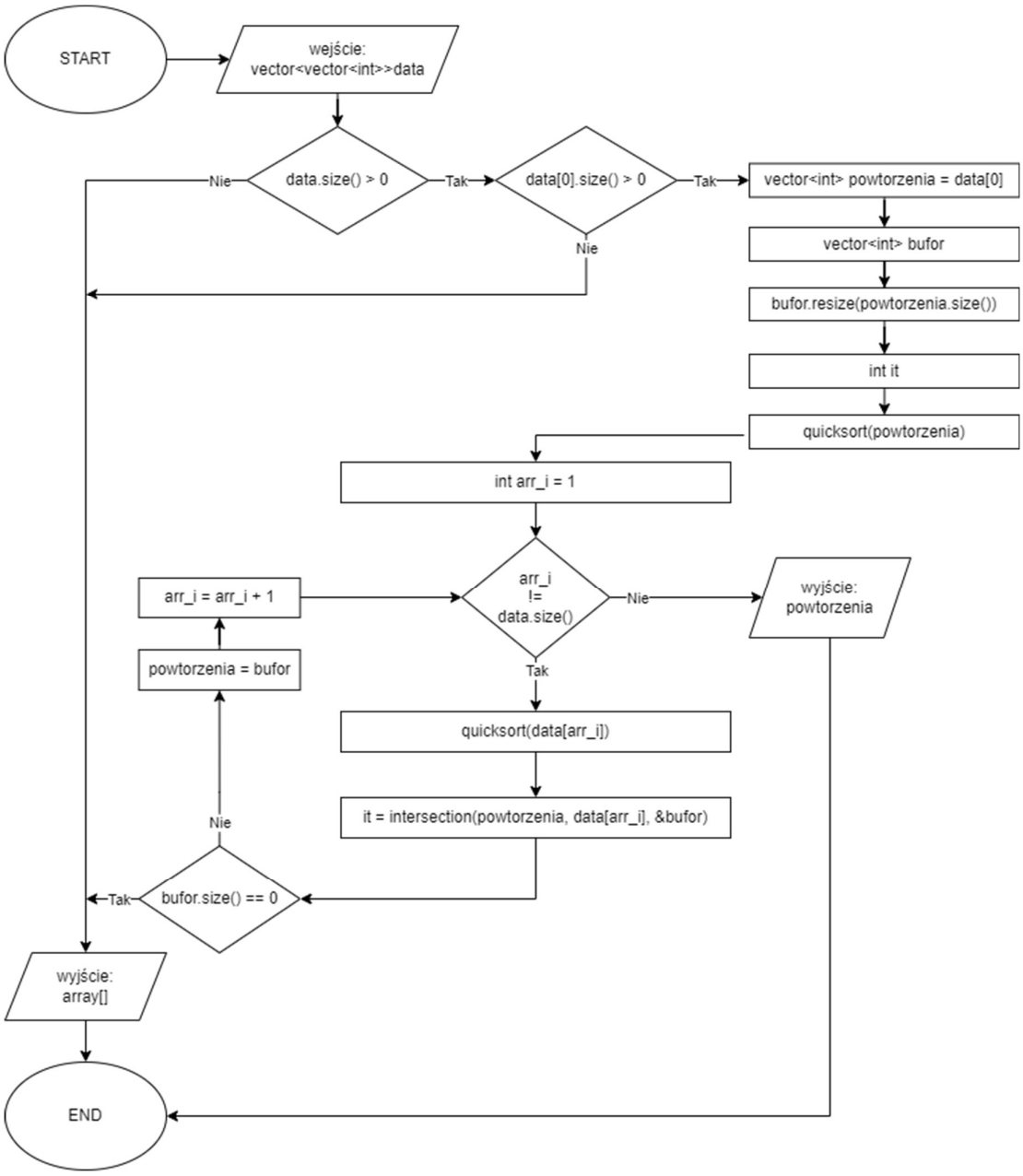
5.2.1 Schemat blokowy algorytmu sortowania quicksort



5.2.2 Schemat blokowy algorytmu intersekcji



5.2.3 Schemat blokowy realizacji sprytniejszej algorytmu



5.3 Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu sortowania Quicksort z uwzględnieniem poprawki dla często powtarzających się wartości wynosi O($n \cdot \log(n)$).

Algorytm intersekcji przechodzi jednocześnie przez obie tablice, na których operuje i robi to tylko raz z czego wynika, że jego złożoność obliczeniowa to O(n).

Dla każdej iteracji algorytmu wykonujemy operację sortowania i porównania przy pomocy wyżej wymienionych funkcji, co daje nam $n \cdot \log(n) \cdot n$ operacji, z czego możemy wywnioskować, że złożoność obliczeniowa całego algorytmu jest równa O($n^2 \cdot \log(n)$).

6 Porównanie algorytmów

Po wykonaniu testów na obu algorytmach dla różnych rozmiarów danych otrzymałem takie wyniki:

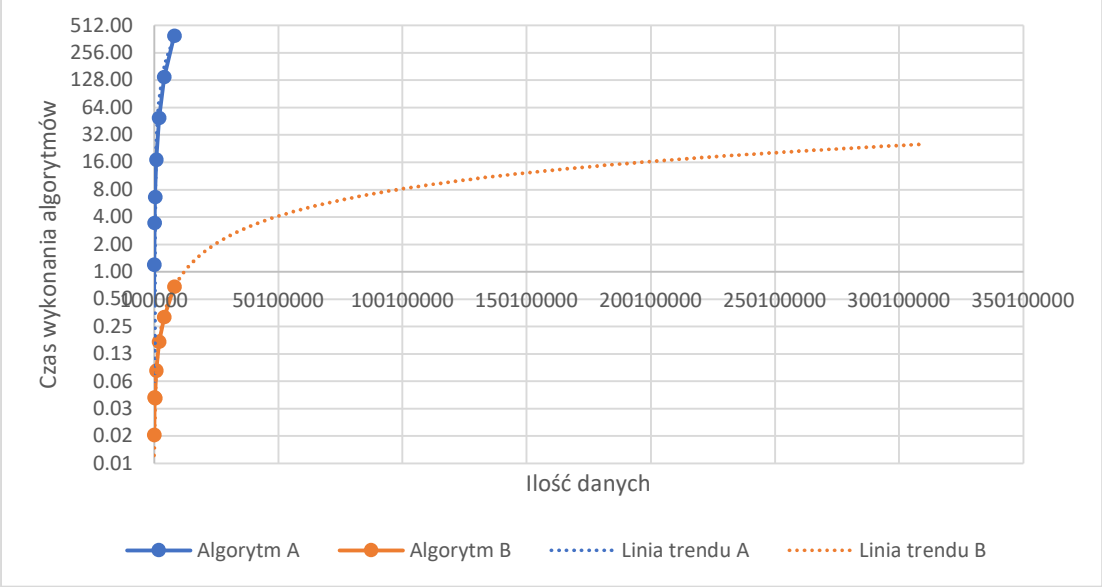
Ilość danych	Algorytm A (s)	Algorytm B (s)
64000	0.44	0.02
127803	1.19	0.02
256000	3.45	0.04
511438	6.63	0.04
1024000	16.98	0.08
2049820	49.15	0.17
4096000	137.97	0.32
8191160	392.55	0.69

Ilość danych liczona jest jako iloczyn ilości wierszy i kolumn tablicy wejściowej, której wysokość i szerokość dla każdego następnego testu obliczany jest ze wzoru:

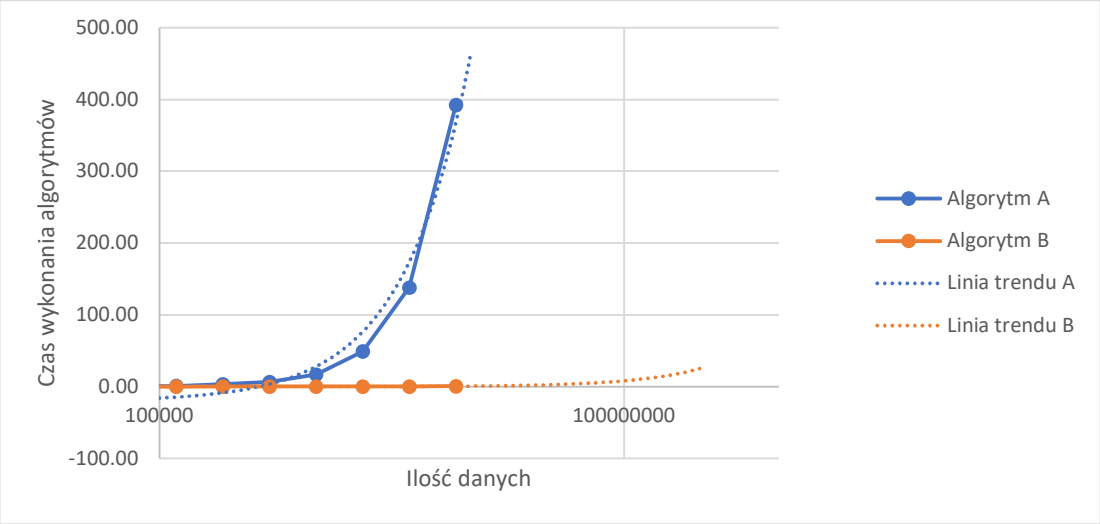
$$x[i] = \lfloor x[0] \cdot sqrt(2)^i \rfloor$$

Zastosowanie takiego wzoru powoduje, że rozmiar każdej następnej tablicy jest większy o 2 od poprzeniego.

Jak widać na załączonym wykresie przedstawiającym stosunek logarytmów czasów wykoniania algorytmów do ilości danych, Algorytm sprytniejszy (B) dąży do wykresu przypominającego logarytm, natomiast złożoność czasowa algorytmu naiwnego (A) bardzo szybko dociera do ogromnych czasów liczenia.



Podobne wnioski możemy wyciągnąć, gdy przedstawimy obydwa algorytmu na wykresie, gdzie oś X będzie osią logarytmiczną.



7 Kod programu

Wybrane funkcje kodu programu:

```
int main()
{
    pro::init();

    // 11 release
    // 8 debug
    auto wyniki = testy(8, 800, 80, 2, 1);

    try
    {
        std::string filename = "test/Testy.txt";
        pro::zapisz_ciag_2d_do_pliku(filename.c_str(), wyniki);
        std::cout << "Wyniki zapisane w pliku " << filename << "\n";
    }
    catch (std::string& e)
    {
        std::cout << "Error: " << e << "\n";
    }

    return 0;
}
```

```
std::vector<std::vector<double>> testy(int ilosc_testow, int start_w, int start_h,
int multiplier, int ilosc_watkow)
// funkcja wywołująca testy czasów na obu algorytmach
{
    std::chrono::high_resolution_clock::time_point start, stop;
    std::chrono::duration<double> timediff;

    std::vector<std::vector<double>> wyniki;

    std::vector<int> wynik_obliczen;

    int width, height;

    for (int nr_testu = 0; nr_testu < ilosc_testow; nr_testu++)
    {

        width = std::round((double)start_w * std::pow(std::sqrt(multiplier),
nr_testu));
        height = std::round((double)start_h * std::pow(std::sqrt(multiplier),
nr_testu));

        std::vector<double> zebrane_dane;

        zebrane_dane.push_back(width*height);

        std::cout << "Test nr: " << nr_testu + 1 << "\n";
        std::cout << "Generowanie tablicy o wymiarach " << width << " na " <<
height << "\n";

        auto data = pro::generuj_losowy_ciag_2d(0, 9, width, height);

        std::cout << "Tablica wejsciowa: ";
        pro::opisz_ciag(data);

        std::cout << "Start A [" << ilosc_watkow << " thread(s)]:\n";
        start = std::chrono::high_resolution_clock::now();
        wynik_obliczen = znajdz_powtorzenia_multithread(data, ilosc_watkow,
&znajdz_powtorzenia_a);
        stop = std::chrono::high_resolution_clock::now();

        timediff = stop - start;

        zebrane_dane.push_back(timediff.count());
        std::cout << "Czas wykonania algorytmu A: " << timediff.count() << "\n";
        std::cout << "Ilosc wynikow: " << wynik_obliczen.size() << "\n";

        std::cout << "Start B [" << ilosc_watkow << " thread(s)]:\n";
        start = std::chrono::high_resolution_clock::now();
        wynik_obliczen = znajdz_powtorzenia_multithread(data, ilosc_watkow,
&znajdz_powtorzenia_b);
        stop = std::chrono::high_resolution_clock::now();

        timediff = stop - start;

        zebrane_dane.push_back(timediff.count());
        std::cout << "Czas wykonania algorytmu B: " << timediff.count() << "\n";
        std::cout << "Ilosc wynikow: " << wynik_obliczen.size() << "\n";

        std::cout << "<int>";
        pro::opisz_ciag(wynik_obliczen);
        std::cout << "\n";


        wyniki.push_back(zebrane_dane);

    }

    return wyniki;
}
```

```
// algorytm - realizacja naiwna
std::vector<int> znajdz_powtorzenia_a(const
std::vector<std::vector<int>>::const_iterator& data_first, const
std::vector<std::vector<int>>::const_iterator& data_last)
{
    // Zwrócenie pustej tablicy jeżeli ilość elementów w podanym zakresie jest
niedodatnia lub pierwsza tablica z podanego zakresu jest pusta
    if (std::distance(data_first, data_last) <= 0 || !data_first->size()) return
std::vector<int>();
    // wykonanie kopii podanego zakresu do nowej tablicy dwuwymiarowej
    auto data = std::vector<std::vector<int>>(data_first, data_last);

    // deklaracja tablicy do wyszukiwania powtórzeń i skopiowanie do niej
zawartości pierwszego ciągu
    std::vector<int> powtorzenia(data[0]);
    // deklaracja tablicy, do której wpiswane będą wyniki porównań w trakcie jednej
iteracji
    std::vector<int> bufor;

    // deklaracja iteratora przechowującego wynik wyszukiwania liniowego w tablicy
powtórzeń
    std::vector<int>::iterator it;

    // deklaracja wartości, którą oznaczane będą wartości już znalezione w tablicy
z początkową wartością pierwszego elementu tablicy powtórzeń
    int empty_marker = powtorzenia[0];
    // przypisanie do pustego znacznika najmniejszej wartości z tablicy powtórzeń
    for (it = powtorzenia.begin(); it != powtorzenia.end(); it++)
        if (*it < empty_marker) empty_marker = *it;

    // w przypadku gdy najmniejszy element w tablicy jest minimalną wartością
możliwą do zapisania w zmiennej typu int
    if (empty_marker == INT_MIN)
        // zmniejszanie wartości pustego markera dopóki element z taką samą
wartością znajduje się w tablicy powtórzeń
        while (pro::linear_search_iterator(powtorzenia, empty_marker) !=
powtorzenia.end()) empty_marker--;
    // w przeciwnym wypadku zmniejszenie wartości pustego znacznika
    else empty_marker--;

    // dla każdej poza pierwszą podtablicy w tablicy wejściowej
    for (auto arr_i = data.begin() + 1; arr_i != data.end(); arr_i++)
    {
        // dla każdego elementu w podtablicy
        for (auto el = arr_i->begin(); el != arr_i->end(); el++)
        {
            // jeżeli szukany element znajduje się w tablicy powtórzeń
            if (*el != empty_marker && (it =
pro::linear_search_iterator(powtorzenia, *el)) != powtorzenia.end())
            {
                // wpisanie znalezionego elementu do tablicy bufor
                bufor.push_back(*it);
                // zamiana wartości znalezionego elementu na pusty znacznik
                *it = empty_marker;
            }

        }
        // jeżeli rozmiar tablicy bufor jest równy 0 zwraca pustą tablicę (brak
powtórzeń)
        if (bufor.size() == 0) return std::vector<int>();
        // przypisanie zawartości tablicy bufor to tablicy powtórzeń
        powtorzenia = bufor;
        // wyczyszczenie zawartości tablicy bufor
        bufor.clear();
    }

    // zwrócenie tablicy powtórzeń
    return powtorzenia;
}
```

```
// algorytm - realizacja sprytniejsza
std::vector<int> znajdz_powtorzenia_b(const
std::vector<std::vector<int>>::const_iterator& data_first, const
std::vector<std::vector<int>>::const_iterator& data_last)
{
    //if (!data_origin.size() || !data_origin[0].size()) return std::vector<int>();
    if (std::distance(data_first, data_last) <= 0 || !data_first->size()) return
std::vector<int>();

    // wykonanie kopii podanego zakresu do nowej tablicy dwuwymiarowej
    auto data = std::vector<std::vector<int>>(data_first, data_last);

    // deklaracja tablicy do wyszukiwania powtórzeń i skopiowanie do niej
zawartości pierwszego ciągu
    std::vector<int> powtorzenia(data[0]);

    // deklaracja tablicy, do której wpiswane będą wyniki porównań w trakcie jednej
iteracji
    std::vector<int> bufor;
    // ustawienie rozmiaru tablicy bufor na równy ilości elementów w tablicy
powtórzenia
    bufor.resize(powtorzenia.size());

    // deklaracja iteratora przechowującego wynik skrzyżowania wartości dwóch
tablic
    std::vector<int>::iterator it;

    // posortowanie tablicy powtórzeń przy użyciu algorytmu quicksort z
usprawnieniem dla często powtarzających się elementów
    // (Dutch national flag problem)
    pro::quicksort_three_way_iterator(powtorzenia.begin(), powtorzenia.end());

    // dla każdej poza pierwszą podtablicy w tablicy wejściowej
    for (auto arr_i = data.begin() + 1; arr_i != data.end(); arr_i++)
    {
        // posortowanie tablicy przy użyciu algorytmu quicksort z usprawnieniem dla
często powtarzających się elementów
        // (Dutch national flag problem)
        pro::quicksort_three_way_iterator(arr_i->begin(), arr_i->end());

        // skrzyżowanie wartości tablicy powtórzeń i podtablicy wejściowej,
wpisanie wyników do tablicy bufor i iteratora końcowego do it
        it = pro::set_intersection(powtorzenia, *arr_i, bufor.begin());

        // jeżeli funkcja krzyżująca zwróciła 0 elementów, zwróć pustą tablicę
        if(bufor.begin() == it) return std::vector<int>();

        // zmiana rozmiaru tablicy bufor na ilość elementów zwróconych z funkcji
krzyżującej
        bufor.resize(std::distance(bufor.begin(), it));

        // przypisanie zawartości tablicy bufor to tablicy powtórzeń
        powtorzenia = bufor;
    }

    // zwrócenie tablicy powtórzeń
    return powtorzenia;
}
```

```
std::vector<int>::iterator pro::linear_search_iterator(std::vector<int>& arr, int
val)
{
	// stworzenie iteratora początku z tablicy wejściowej
	auto it = arr.begin();
	// dopóki iterator nie dotarł do końca tablicy
	while (it != arr.end())
	{
		// jeżeli wartość wskazywana przez iterator jest równa wartości
szukanej zwróć ten iterator
		if (*it == val) return it;
		// zwiększ wartość iteratora
		it++;
	}
	// zwróć iterator równy końcowi tablicy (brak wyników)
	return it;
}
```

```
std::vector<int>::iterator pro::set_intersection(const std::vector<int>& arr1,
const std::vector<int>& arr2, std::vector<int>::iterator res)
{
	// swtorzenie iteratorów z danych wejściowych
	std::vector<int>::const_iterator first1 = arr1.begin(), last1 = arr1.end(),
first2 = arr2.begin(), last2 = arr2.end();

	// dopóki iterator żadnej z tablicy nie dotarł do jej końca
	while (first1 != last1 && first2 != last2)
	{
		// jeżeli wartość wskazywana przez pierwszy iterator jest mniejsza
niż wartość wskazywana przez drugi iterator
		if (*first1 < *first2)
		{
			// zwiększ pierwszy iterator
			++first1;
		}
		// w przeciwnym wypadku, jeżeli wartość wskazywana przez drugi
iterator jest mniejsza od wartości wskazywanej przez pierwszy iterator
		else if (*first2 < *first1)
		{
			// zwiększ drugi operator
			++first2;
		}
		// w przeciwnych wypadkach
		else
		{
			// w tym momencie wiemy, że pierwszy i drugi iterator wskazują
na taką samą wartość
			// wpisz wartość wskazywaną przez pierwszy iterator do pamięci
wskazywanej przez iterator tablicy z wynikami
			*res = *first1;

			// zwiększ iterator tablicy z wynikami oraz pierwszej i drugiej
tablicy
			++res;
			++first1;
			++first2;
		}
	}
	// zwróć iterator tablicy z wynikami
	return res;
}
```

```
void pro::quicksort_three_way_iterator(std::vector<int>::iterator begin,
std::vector<int>::iterator end)
{
	// jeżeli ciąg jest długości niewiększej od 1 to jest już posortowany
	if (std::distance(begin, end) <= 1)
	{
		return;
	}

	// jeżeli ciąg ma 2 elementy
	if (std::distance(begin, end) == 2)
	{
		// jeżeli pierwszy element jest większy od drugiego
		if (*begin > *(begin + 1))
		{
			// zamień je miejscami
			std::swap(*begin, *(begin + 1));
		}
		return;
	}

	// oblicz zakresy następnych obszarów do posortowania
	auto pivot = pro::quicksort_iterator_three_way_partition(begin, end);

	// posortuj lewą część tablicy
	pro::quicksort_three_way_iterator(begin, pivot.first);

	// posortuj prawą część tablicy
	pro::quicksort_three_way_iterator(pivot.second, end);
}
```

8 Wnioski

Program poprawnie rozwiązuje problem zadeklarowany na początku pracy dwoma różnymi algorytmami. Czasy trwania programu są zależne od rozmiaru tablicy podanej mu na wejściu i wzrastają zgodnie z oczekiwaniami dla obliczonej złożoności czasowej. Algorytm sprytniejszy wykorzystując operację sortowania przyśpieszył bardzo mocno czas pracy programu dla dużych tablic.