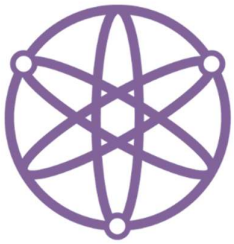


WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Dariusz Strojny

Algorytmy i Struktury Danych

Projekt zaliczeniowy nr 2

Rzeszów, 2022

1 Spis treści

2 Spis ilustracji

3 Temat

Porównaj algorytmy sortowania gnoma oraz sortowania kubełkowego.

4 Analiza, projektowanie

4.1 Zasada działania programu

Program ten ma za zadanie wygenerowanie ciągów wypełnionych losowymi wartościami a następnie posortowanie ich przy pomocy algorytmów sortowania gnoma i sortowania kubełkowego. Program zapisuje wyniki do plików tekstowych w celu dalszej analizy.

4.2 Struktury danych

Dane przechowywane są w tabeli jednowymiarowej składającej się z n elementów. Zakres danych, które są losowane do tabeli n jest dobierany przez program w ten sposób, aby wykonać trzy serie pomiarów, w której każda ma inną proporcję ilości danych do zakresu, z którego są one losowane. Proporcje te to 1:1000, 1:1 oraz 1000:1. Użycie takich proporcji pozwala przebadać zachowanie przebiegu algorytmu sortowania kubełkowego, którego wydajność zależy od zakresu danych w sortowanej strukturze. Wylosowane dane zawsze będą liczbami nieujemnymi, całkowitymi.

Użycie ograniczonej n-elementowej tablicy pozwoli nam na zwiększenie wydajności działania naszego programu oraz ograniczy możliwość popełnienia błędów mogących pojawić się podczas pracy na tych danych.

4.3 Metodyka

4.3.1 Definicje

• PRO_FILE_VALUE_DELIMITER ' '
Domyślny znak oddzielający wartości w plikach tekstowych

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

4.3.2 Funkcje pomocnicze użyte w programie

• 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

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

Wypisuje w konsoli wymiary tablicy.

Parametry

arr - Opisywana tablica

• 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::**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

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

Testuje poprawność wykonania algorytmów sortowania.

W przypadku błędu w sortowaniu wyrzuca błąd typu std::string().

Parametry

arr – Potencjalnie posortowany ciąg

4.3.3 Główne funkcje algorytmów

• std::vector<int>
gnome_sort(std::vector<int> array)

Funkcja implementująca sortowanie gnoma z treści zadania.

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

Parametry

array – tablica wejściowa do posortowania

Zwraca

Posortowana tablica, która została przekazana jako argument

• std::vector<int>
bucket_sort(std::vector<int> array, std::pair<int, int> range)

Funkcja implementująca sortowanie kubełkowe z treści zadania.

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

Parametry

array – tablica wejściowa do posortowania

range – para wartości określających zakres danych, jaki może znaleźć się w sortowanej tabeli

Zwraca

Posortowana tablica, która została przekazana jako argument

5 Opis działania algorytmu sortowania gnoma

Algorytm sortowania metodą gnoma polega na iteracji wewnątrz sortowanej tabeli, zmieniając kierunki w zależności od stosunku rozmiarów jej porównywanych sąsiednich elementów.

W podstawowej wersji tego algorytmu sortowanie rozpoczyna się od drugiego elementu na liście. Jeżeli aktualnie rozpatrujemy pierwszy element z listy, albo element po lewej od rozpatrywanego elementu jest od niego nie większy, to należy rozpatrzyć te same warunki dla elementu po jego prawej stronie. W przeciwnym wypadku należy zamienić aktualny element z poprzednim i kontynuować sortowanie dla elementu po lewej.

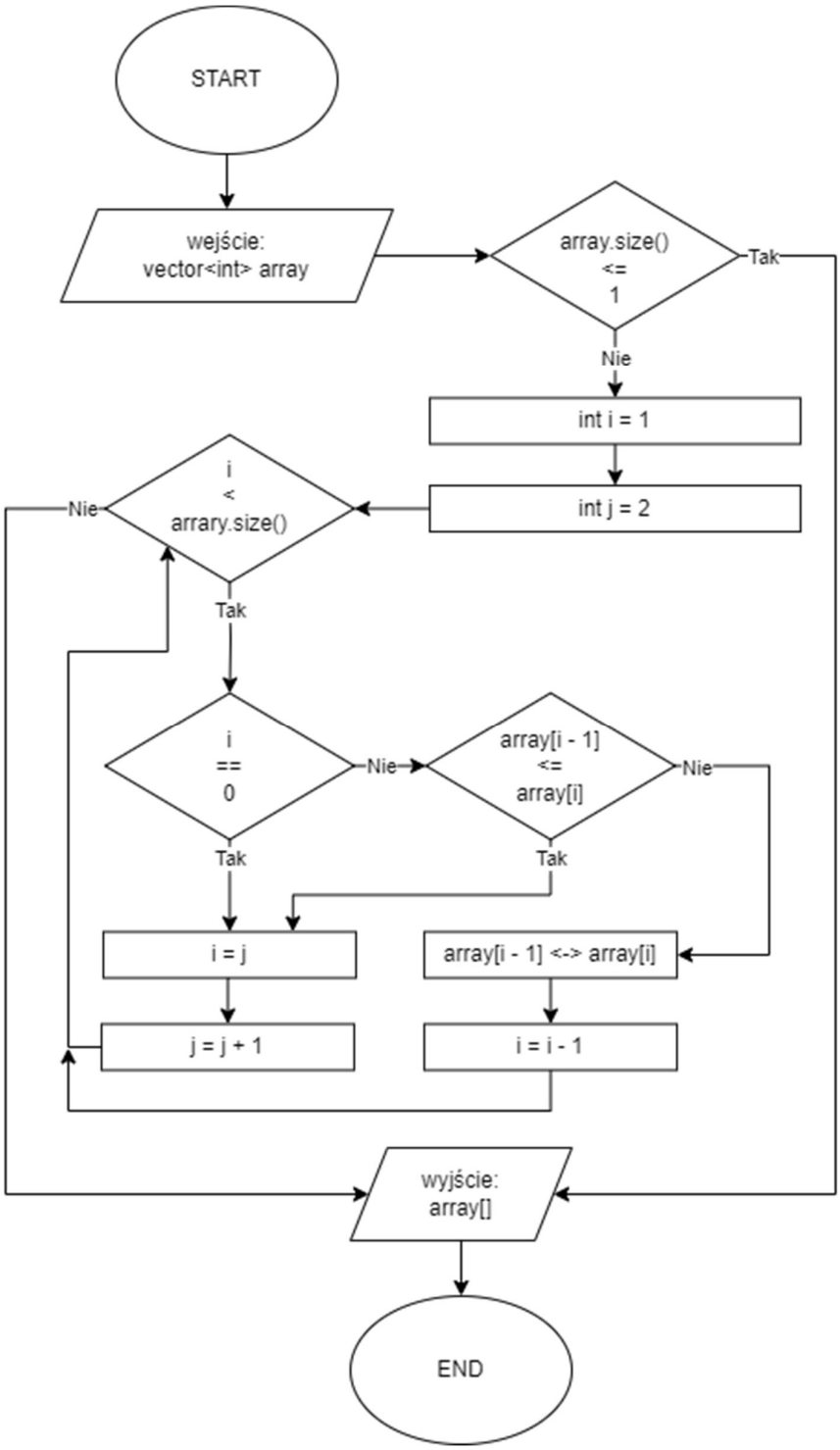
Wersja usprawniona algorytmu zapamiętuje, które elementy w tablicy zostały już posortowane poprzez wprowadzenie drugiego iteratora wskazującego na początku na pierwszy element. W momencie, gdy spełnione są warunki do rozpatrywania elementu po prawej algorytm rozpatruje zamiast tego element, na który wskazuje drugi iterator a następnie go inkrementuje. Pozwala to przeskoczyć od razu do rozpatrywania najdalszego posortowanego elementu gdy element z prawej strony został już posortowany w lewej części sortowanego ciągu.

W swoim przykładzie przedstawiłem usprawnioną wersję algorytmu. Dzięki właściwościom języka C++ powodującym, że wartości przekazane jako argumenty są domyślnie kopią oryginału a przypisanie tablicy typu std::vector do innego obiektu tego typu wykonuje jego głęboką kopię, w mojej implementacji algorytmu nie definiujemy tabeli przechowującej wynik sortowania. Zamiast tego pracujemy bezpośrednio na tablicy wejściowej, która nie jest oryginałem więc nie musimy się martwić o zmianę wartości w ciągu przekazywanym jako argument z innych miejsc w kodzie.

5.1 Pseudokod

```
wejście:

array             - tablica zawierająca dane do posortowania

dane:

i                 - iterator przechowujący aktualną pozycję w tablicy
j                 - iterator przechowujący następną nieposortowaną
                  pozycję

algorytm:

jeżeli rozmiar tablicy array jest <= od 1:
      zwróć tablicę array

i <- 1
j <- 2
dopóki i < rozmiar tablicy array:
      jeżeli i == 0 lub array[i - 1] <= array[i]:
            i <- j
            j <- j + 1
      w przeciwnym przypadku:
            array[i - 1] <-> array[i]
            i <- i - 1
zwróć tablicę array
```

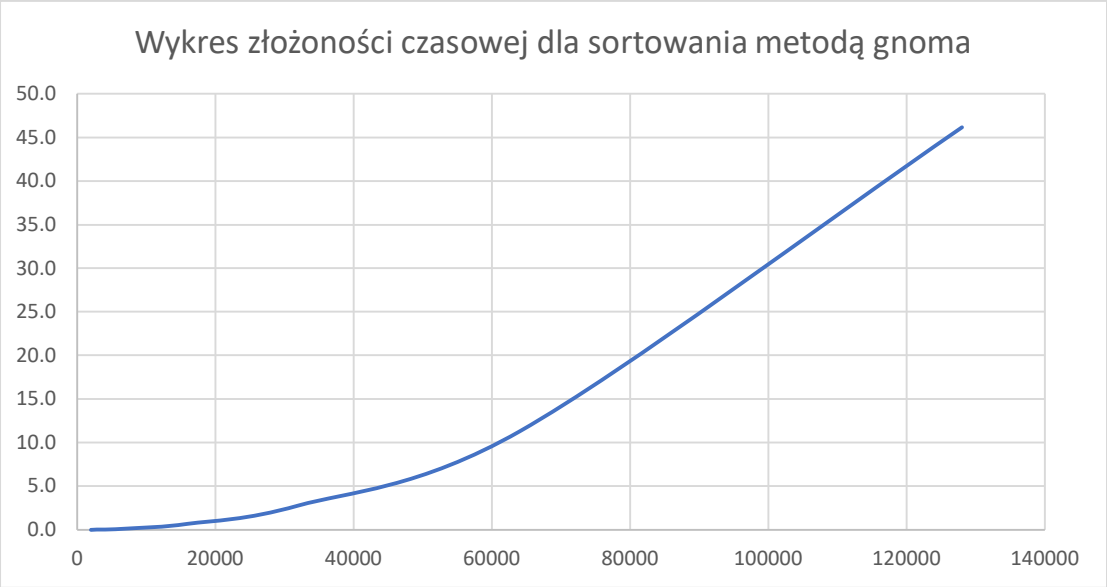
5.2 Schemat blokowy



Rys 1 – schemat blokowy algorytmu sortowania gnoma.

5.3 Złożoność obliczeniowa

Algorytm wyróżnia się prostotą, nie zawiera zagnieżdżonych pętli. Jego złożoność obliczeniowa to O(n^2) w średnim przypadku, jednak zbliża się do O(n), jeśli zbiór wejściowy jest prawie posortowany lub jego elementy znajdują się niedaleko miejsc, na których znalazły by się po sortowaniu.



Rys 2 – Wykres złożoności czasowej algorytmu sortowania gnoma.

6 Opis działania algorytmu sortowania kubełkowego

Algorytm sortowania kubełkowego jest algorytmem wykorzystywanym najczęściej, gdy sortowany zbiór posiada dużą liczbę elementów o małym zbiorze wartości. Polega on na utworzeniu dodatkowej tablicy o rozmiarze będącym ilością liczb całkowitych w zbiorze wartości podanym jako argument a następnie iterowaniu przez kolejne elementy ciągu wejściowego, inkrementując wartości dodatkowej tablicy pod indeksem odpowiadającym wartości tego elementu. Tak utworzoną tablicę możemy wykorzystać do utworzenia ciągu będącą posortowaną tablicą wejściową. Aby to zrobić, należy iterując przez każdy element tablicy dodatkowej spisać do nowej tablicy tyle elementów równych aktualnemu indeksowi ile wynosi wartość elementu, na które ten indeks wskazuje.

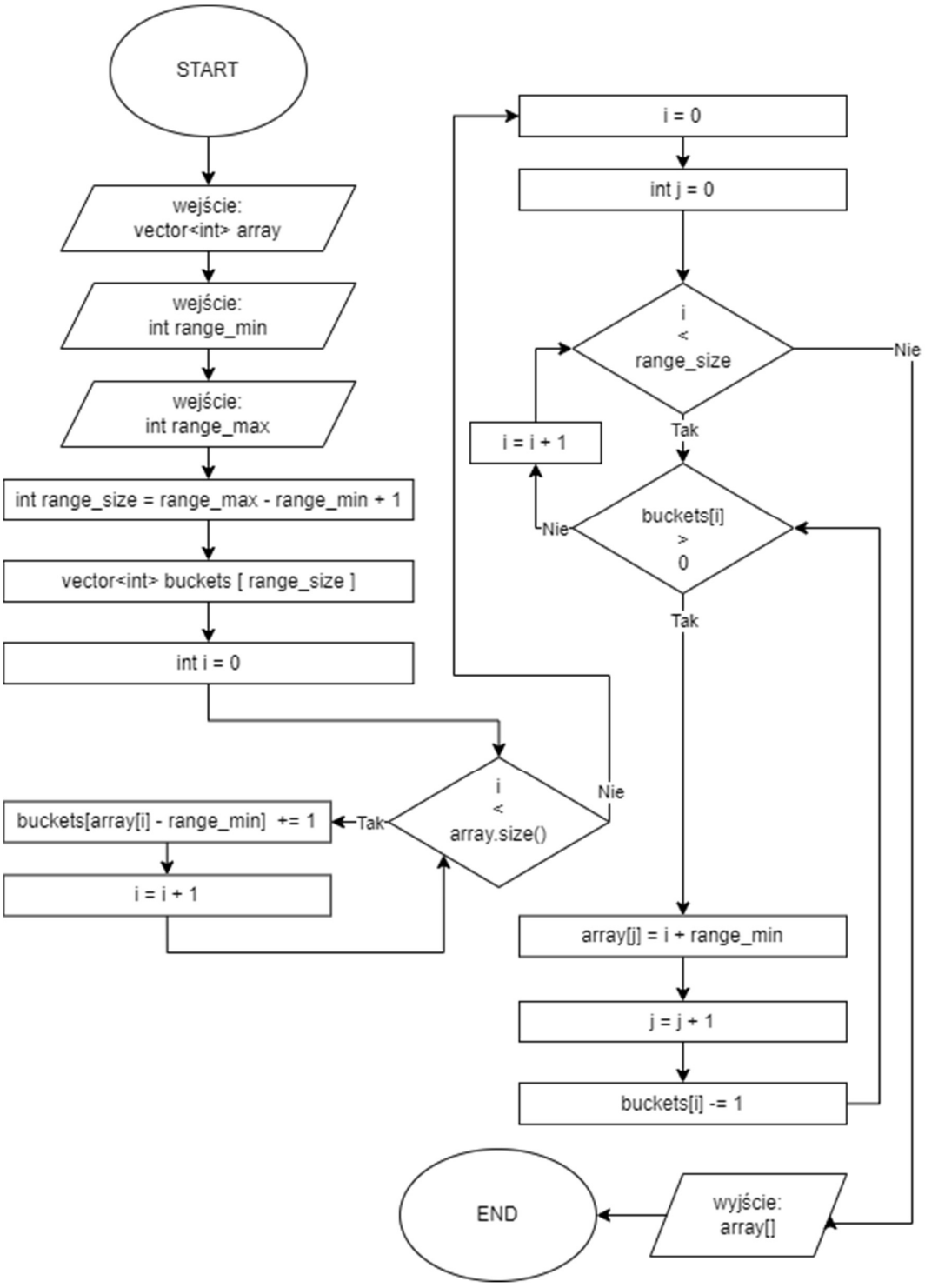
W implementacji algorytmu, którą napisałem, nie tworzę tabeli na wartości zwracane. Tak samo jak w przypadku algorytmu sortowania gnoma wykorzystuję tablicę wejściową jako obiekt, w którym przechowuję dane wyjściowe aż do ich zwrócenia.

Sortowanie kubełkowe jest jednym z najszybszych algorytmów sortowania, jednak ma on swoje ograniczenia. Jeżeli zakres wartości podany jako argument jest bardzo duży to tablica do zliczania elementów zajmuje odpowiednio dużo pamięci, co może być problemem na przykład w systemach wbudowanych, gdzie odpowiednie zarządzanie pamięcią jest bardzo istotne. Dużym ograniczeniem jest też wymaganie posiadania wiedzy na temat dokładnego rozstępu zbioru (różnicę między największą i najmniejszą wartością do posortowania).

6.1 Pseudokod

```
wejście:

array             - tablica zawierająca dane do posortowania
range_min         - najmniejsza możliwa wartość w tablicy
range_max         - największa możliwa wartość w tablicy

dane:

range_size        - zmienna przechowująca rozmiar zakresu danych
buckets           - tablica przechowująca liczniki sortowanych wartości
i                 - iterator przechowujący aktualną pozycję w tablicy
j                 - iterator wskazujący następne nienadpisane pole w
                  tabeli wejściowej

algorytm:

jeżeli rozmiar tablicy array jest <= od 1:
      zwróć tablicę array

range_size <- range_max - range_min + 1

buckets <- tablica o rozmiarze range_size

i <- 0
dopóki i < rozmiar tablicy array:
      buckets[array[i] - range_min] <- buckets[array[i] - range_min] +
1
      i <- i + 1

i <- 0
j <- 0

dopóki i < range_size:
      dopóki buckets[i] > 0:
            array[j] <- i + range_min
            j <- j + 1
            buckets[i] <- buckets[i] - 1
      i <- i + 1

zwróć tablicę array
```

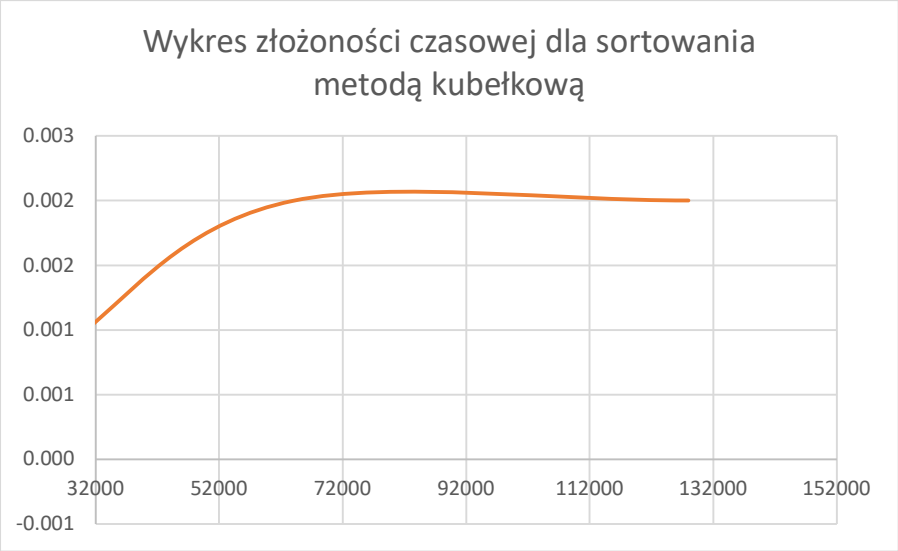
6.2 Schemat blokowy



Rys 3 – Schemat blokowy algorytmu sortowania kubełkowego.

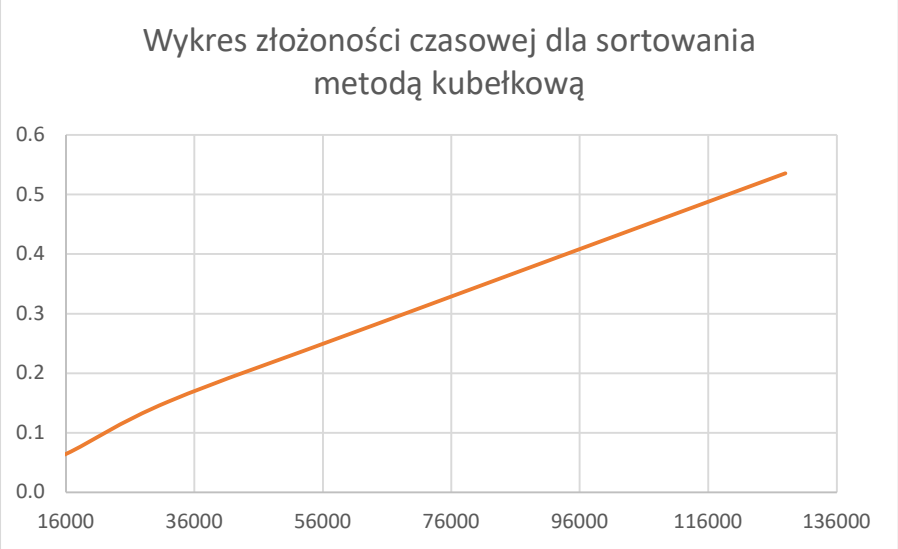
6.3 Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu kubełkowego jest zależna od rodzaju danych, na których operuje. Jeżeli spełniony zostanie warunek równomiernego rozłożenia danych to złożoność tego algorytmu jest liniowa i wynosi O(n). W najgorszym przypadku złożoność tego sortowania wynosi O(n^2).



Rys 4 – Wykres algorytmu sortowania kubełkowego dla złożoności O(n).

Powyższy wykres przedstawia złożoność czasową algorytmu sortowania kubełkowego w przypadku podania równomiernie rozłożonych danych z bardzo małego zakresu. Dla tego przykładu wygenerowano tablicę, w której każda liczba powtarza się około 1000 razy.



Rys 5 – Wykres algorytmu sortowania kubełkowego dla złożoności O(n^2).

powyższy wykres przedstawia złożoność czasową algorytmu sortowania kubełkowego w przypadku podania danych z bardzo małego dużego zakresu. Dla tego przykładu wygenerowana została tablica, w której dane oddalone są od siebie średnio o 1000 wartości.

7 Porównanie algorytmów

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

Wykonanie algorytmów sortowania dla przedziałów danych z częstymi powtórzeniami elementów zwróciło wyniki po następujących czasach pracy:

Ilość danych	sortowanie gnoma	sortowanie kubełkowe
2000	0.007003	0.000000
4000	0.035855	0.000000
8000	0.160199	0.000000
16000	0.673587	0.000000
32000	2.784370	0.001036
64000	11.297500	0.001508
128000	46.141500	0.001999

Wykonanie tych samych algorytmów dla rzadkiego rozłożenia elementów w tablicach, dla sortowania gnoma zajęło bardzo podobną ilość czasu, natomiast w przypadku sortowania kubełkowego, zgonie z wcześniejszymi założeniami, czas sortowania znacznie wzrósł:

ilość danych	sortowanie gnoma	sortowanie kubełkowe
2000	0.011002	0.008469
4000	0.048022	0.017056
8000	0.181780	0.033567
16000	0.722450	0.064446
32000	2.906400	0.152528
64000	11.528600	0.280779
128000	46.437900	0.535190

8 Kod programu

Wybrane funkcje kodu programu:

```
int main()
{
    // inicjalizacja funkcji pomocniczych
    pro::init();

    // stworzenie wybranego zakresu początkowego dla wszystkich testów
    std::pair<int, int> range = { 0, 1 };

    // wykonanie trzech zestawów testów
    for (int i = 1; i <= 3; i++)
    {

        // przechwytywanie wszystkich błędów typu std::string
        try
        {
            // wpisanie wyników testów do tabeli
            auto wyniki = testy(8, 1000, range, 2);

            // zapisanie tabeli do pliku odpowiadającego zestawowi testów
            std::string filename = std::string("test/Testy ") + std::to_string(i) +
".txt";
            pro::zapisz_ciag_2d_do_pliku(filename.c_str(), wyniki);
            std::cout << "Wyniki zapisane w pliku " << filename << "\n";
        }
        catch (std::string& e)
        {
            // wypisanie przechwyconego błędu
            std::cout << "Error: " << e << "\n";
        }

        range.first *= 500;
        range.second *= 500;
    }



    return 0;
}
```

```
funkcja wywołująca testy czasów na obu algorytmach
std::vector<std::vector<double>> testy(int ilosc_testow, int start_len,
std::pair<int, int> range_, int multiplier)
{
    // inicjalizacja struktur czasu
    std::chrono::high_resolution_clock::time_point start, stop;
    std::chrono::duration<double> timediff;

    // definicja tabeli wyników czasów trwania i obliczeń
    std::vector<std::vector<double>> wyniki;
    std::vector<int> wynik_obliczen;

    int arr_len;
```

```
    // dla podanej ilości testów
    for (int nr_testu = 0; nr_testu < ilosc_testow; nr_testu++)
    {
        // obliczanie mnożnika dla aktualnego numeru testu
        int mp_pow = std::pow(multiplier, nr_testu);
        // obliczenie zakresu losowanych liczb
        std::pair<int, int> range ={range_.first * mp_pow, range_.second * mp_pow};
        // obliczanie długości generowanego ciągu
        arr_len = start_len * mp_pow;

        // definicja tablicy przechowującej wyniki pojedyńczego testu
        std::vector<double> zebrane_dane;

        // wpisanie ilości danych do wyników
        zebrane_dane.push_back(arr_len);

        std::cout << "Test nr: " << nr_testu + 1 << "\n";
        std::cout << "Generowanie tablicy o rozmiarze " << arr_len << " i zakresie
danych " << range.first << " do " << range.second << "\n";

        // generowanie ciągu o zadanych parametrach
        auto data = pro::generuj_losowy_ciag(range.first, range.second, arr_len);

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

        // wykonanie testu dla pierwszego algorytmu z pomiarem czasu
        std::cout << "Start A:\n";
        start = std::chrono::high_resolution_clock::now();
        wynik_obliczen = gnome_sort(data);
        stop = std::chrono::high_resolution_clock::now();
        // sprawdzenie poprawności wyników
        test_sort(wynik_obliczen);

        // obliczenie czasu trwania algorytmu
        timediff = stop - start;

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

        // wykonanie testu dla drugiego algorytmu z pomiarem czasu
        std::cout << "Start B:\n";
        start = std::chrono::high_resolution_clock::now();
        wynik_obliczen = bucket_sort(data, range);
        stop = std::chrono::high_resolution_clock::now();
        // sprawdzenie poprawności wyników
        test_sort(wynik_obliczen);

        // obliczenie czasu trwania algorytmu
        timediff = stop - start;

        // wpisanie czasu do wyników
        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";

        // wpisanie wyników do zwracanej tabeli
        wyniki.push_back(zebrane_dane);

    }

    return wyniki;
}
```

```cpp
// algorytm sortowania gnoma
std::vector<int> gnome_sort(std::vector<int> array)
{
    if (array.size() <= 1) return array;

    // iterator przechowujący aktualną pozycję w tablicy
    size_t i = 1;
    // iterator przechowujący następną nieposortowaną pozycję
    size_t j = 2;

    // dopóki iterator "i" ma mniejszą wartość niż jest liczb w tablicy
    while (i < array.size())
    {
        // jeżeli i jest równe 0 lub
        // liczba na lewo od liczby wskazywanej przez iterator jest niewiększa niż
liczba, którą wskazuje ten iterator
        if (i == 0 || array[i - 1] <= array[i])
        {
            // przypisz do iteratora "i" element następnej
            i = j;
            // zwiększ wartość następnej pozycji do sprawdzenia
            j++;
        }
        // jeżeli liczba na lewo od sprawdzanej jest od niej mniejsza
        else
        {
            // zamień te dwie liczby miejscami
            std::swap(array[i - 1], array[i]);;
            // zmniejsz wartość iteratora
            i--;
        }

    }

    // zwróć wartość posortowanego ciągu
    return array;
}
```

```
// algorytm sortowania kubełkowego
std::vector<int> bucket_sort(std::vector<int> array, std::pair<int, int> range)
{
    // zwrócenie ciągu jako posortowany, jeżeli jego rozmiar jest niewiększy niż 1
    if (array.size() <= 1) return array;
    // wyrzucenie błędu, jeżeli zakres danych jest niepoprawny
    if (range.second - range.first < 0) throw "Podany zakres danych jest
niepoprawny!\n";

    // obliczenie rozmiaru tablicy liczników
    int range_size = range.second - range.first + 1;

    // definicja tablicy liczników
    std::vector<int> buckets;
    // zanicjowanie tablicy liczników wartościami zerowymi
    buckets.resize(range_size);

    // dla każdego elementu w tablicy
    for (const int& el : array)
        // zwiększenie wartość licznika o indeksie tego elementu
        buckets[el - range.first]++;

    // definicja iteratora "i" przechodzącego przez tablice liczników
    int i = 0;

    // definicja iteratora "j" wskazującego następne nienadpisane pole w tabeli
wejściowej
    int j = 0;

    // dla każdego i w zakresie tablicy liczników
    while (i < range_size)
    {
        // dopóki licznik wskazywany przez indeks "i" jest większy od zera
        while (buckets[i] > 0)
        {
            // wpisanie do tablicy wejściowej na pierwszym nienadpisanym indeksie
sumy wartości iteratora "i" oraz początku zakresu danych
            array[j] = i + range.first;
            // inkrementacja wskaźnika na nienadpisaną pozycję
            ++j;
            // dekrementacja licznika
            --buckets[i];
        }
        i++;
    }

    // zwracanie tablicy wejściowej nadpisanej posortowanymi wartościami
    return array;
}
```

9 Wnioski

W projekcie udało się zaprezentować poprawne działanie algorytmów sortujących, zbadać ich złożoność obliczeniową w zależności od różnych konfiguracji danych wprowadzanych jako argumenty oraz potwierdzić ją obliczeniami i wykresami.

Algorytm sortowania gnoma jest algorytmem sortującym w miejscu, ponieważ nie wymaga żadnej dodatkowej przestrzeni na dane, natomiast sortowanie kubełkowe nie jest takim algorytmem ponieważ potrzebuje ono zapisać liczniki do dodatkowej tabeli.

Pomimo, że algorytm sortowania gnoma jest wolniejszy od sortowania kubełkowego, mógł by się on sprawdzić lepiej w przypadkach, gdy nie jest znany zbiór wartości tabeli do sortowania, rozbieżność danych jest bardzo duża lub zakres sortowanych wartości zajął by za dużo pamięci.