POLITECHNIKA KRAKOWSKA im. T. Kościuszki
Wydział Inżynierii Elektrycznej i Komputerowej

Kierunek studiów: IWIK

STUDIA STACJONARNE

Metody Programowania – Projekt

Sortowanie: bąbelkowe, przez scalanie, szybkie, kopcowanie

Wiktoria Kurpyta
Filip Majewski
Leon Wałach

1.Wstęp teoretyczny

Sortowanie jest fundamentalnym problemem w informatyce i matematyce, który polega na uporządkowaniu elementów zbioru w określonej kolejności, zazwyczaj rosnącej lub malejącej. Efektywne sortowanie jest kluczowe dla optymalizacji wielu algorytmów, takich jak np. wyszukiwanie binarne. Jest również niezbędne w wielu aplikacjach, od baz danych po systemy operacyjne. Poprawia także czytelność danych i ułatwia ich analizę.

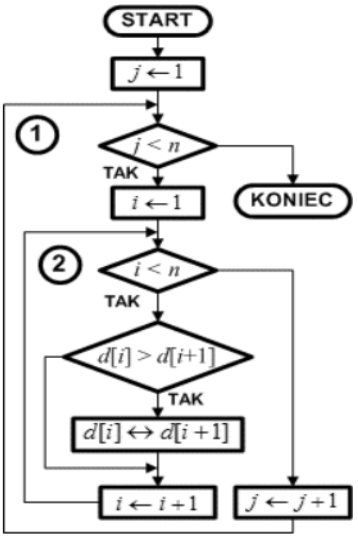
W praktyce istnieje wiele algorytmów sortowania, z których każdy ma swoje unikalne wady i zalety. Wybór odpowiedniego algorytmu zależy od specyfiki problemu, wielkości zbioru danych, dostępnych zasobów pamięci oraz wymagań dotyczących stabilności i złożoności czasowej. W niniejszym sprawozdaniu omówimy i zaimplementujemy cztery popularne algorytmy sortowania: **sortowanie bąbelkowe**, **sortowanie przez scalanie**, **sortowanie szybkie** oraz **sortowanie przez kopcowanie**.

- ## Sortowanie bąbelkowe

  Nazwa pochodzi stąd, że podczas sortowania dane, podobnie jak bąbelki w napoju gazowanym, przesuwają się na prawą stronę tablicy, stopniowo ustawiając się w odpowiedniej kolejności. Jest to prosty algorytm, który działa poprzez wielokrotne **porównywanie sąsiadujących elementów**. Jeśli dwa sąsiadujące elementy są w niewłaściwej kolejności (np. nie są posortowane rosnąco), są one zamieniane miejscami. Ten proces powtarza się aż do momentu, gdy cała tablica zostanie posortowana.

  Złożoność obliczeniowa tego sposobu sortowania wynosi $O(n^2)$.

  Schemat blokowy:



gdzie:

n - liczba elementów w sortowanym zbiorze, $n \in N$

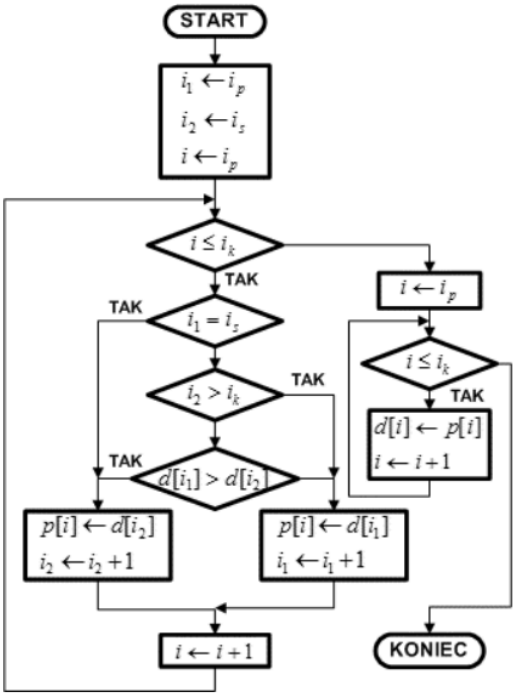
$d[]$ *- zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do n.*

i, j - zmienne sterujące pętli, $i,j \in N$

• Sortowanie przez scalanie

Sortowanie przez scalanie to rodzaj sortowania, który został wynaleziony w 1945 roku przez Johna von Neumanna. Wykorzystuje on technikę **dziel i zwyciężaj**. Algorytm działa poprzez podzielenie tablicy na dwie równe części, następnie **rekurencyjne sortowanie** każdej z tych części, a na koniec scalenie dwóch posortowanych części w jedną posortowaną tablicę.
Złożoność obliczeniowa wynosi w tym przypadku $O(n \cdot logn)$ co czyni algorytm wydajniejszym niż przy sortowaniu bąbelkowym.

Schemat blokowy:



gdzie:

$d[]$ *- scalany zbiór*

$p[]$ *- zbiór pomocniczy, który zawiera tyle samo elementów, co zbiór* $d[]$

i_p *- indeks pierwszego elementu w młodszym podzbiorze,* $i_p \in N$

i_s *- indeks pierwszego elementu w starszym podzbiorze,* $i_s \in N$

i_k *- indeks ostatniego elementu w starszym podzbiorze,* $i_k \in N$

i_1 *- indeks elementów w młodszej połówce zbioru* $d[]$, $i_1 \in N$

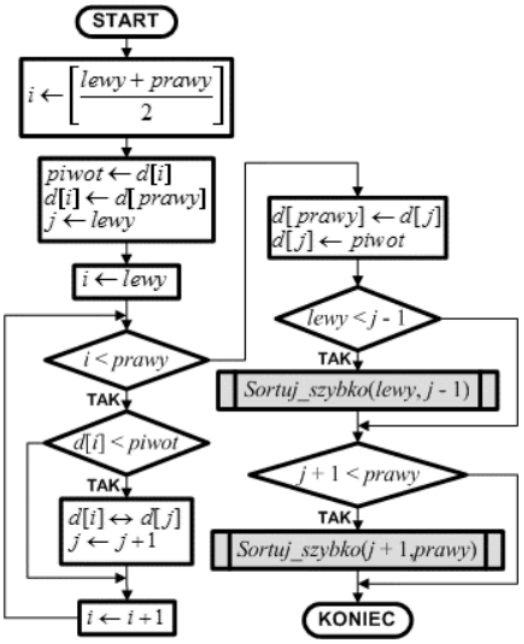
i_2 *- indeks elementów w starszej połówce zbioru* $d[]$, $i_2 \in N$

i *- indeks elementów w zbiorze pomocniczym* $p[]$, $i \in N$

- ## Sortowanie szybkie

Sortowanie szybkie to algorytm, wynaleziony przez Tony'ego Hoare'a, który również wykorzystuje technikę **dziel i zwyciężaj**. Działa poprzez wybór elementu zwanego pivotem (jako punkt odniesienia np. pierwszy element, ostatni element, środkowy element lub losowy) , a następnie uporządkowanie tablicy w taki sposób, że elementy mniejsze od pivota znajdują się przed nim, a większe za nim. Proces ten jest powtarzany **rekurencyjnie** dla podtablic po lewej i prawej stronie pivota.
Sortowanie szybkie jest często preferowanym algorytmem sortującym ze względu na swoją szybkość w typowych przypadkach, gdzie jego **złożoność obliczeniowa** wynosi $O(n\ logn)$. Jednakże, istnieje ryzyko, że w niekorzystnych sytuacjach, takich jak nieoptymalny wybór pivota lub prawie posortowane dane wejściowe, jego złożoność może wzrosnąć do $O(n^2)$. Dodatkowo, ze względu na swoją rekurencyjną naturę, istnieje ryzyko przepełnienia stosu, co może prowadzić do awarii programu lub systemu. Zalecane jest więc przeprowadzić analizę danych wejściowych pod kątem potencjalnych zagrożeń.

Schemat blokowy:



gdzie:

$d[]$ - *Zbiór zawierający elementy do posortowania. Zakres indeksów elementów jest dowolny.*

lewy - indeks pierwszego elementu w zbiorze, lewy $\in C$

prawy - indeks ostatniego elementu w zbiorze, prawy $\in C$

piwot - element podziałowy

i, j - indeksy, i, j $\in C$

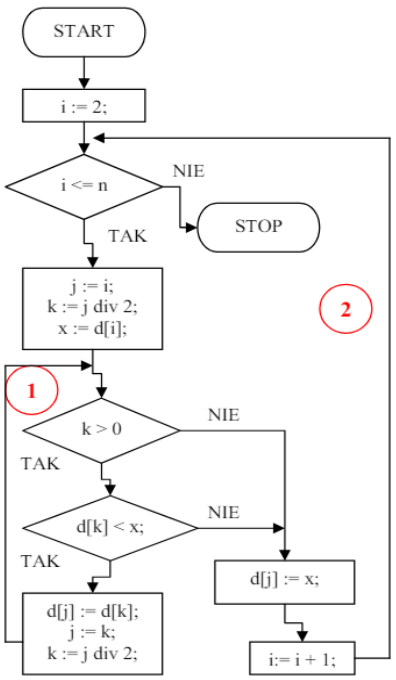
- ## Sortowanie przez kopcowanie

Sortowanie przez kopcowanie jest algorytmem, który przekształca dane wejściowe w strukturę kopca (*heap*). Jest to kompletne drzewo binarne, czyli hierarchiczna struktura danych, w której elementy, zwane węzłami lub wierzchołkami (ang. *nodes*), są zorganizowane w sposób umożliwiający każdemu węzłowi posiadanie maksymalnie dwóch bezpośrednich następców. W tradycyjnej strukturze liniowej każdy element może mieć tylko jednego następcę. W drzewie binarnym natomiast każdy węzeł może mieć dwóch następców, co daje nazwę "***binarny***" - oznaczającą dwa elementy. Ci następcy są nazywani potomkami, dziećmi lub węzłami potomnymi (ang. child nodes).
Aby nazwać drzewo kopcem musi spełniać **warunek kopca**:

- w kopcu maksymalnym każdy rodzic jest większy lub równy swoim dzieciom,
- w kopcu minimalnym każdy rodzic jest mniejszy lub równy swoim dzieciom.

Algorytm działa w dwóch głównych etapach: najpierw przekształca tablicę wejściową w kopiec maksymalny, a następnie systematycznie usuwa największy element z kopca i umieszcza go na końcu tablicy. Nazywamy to **rozbiorem kopca**. Proces ten jest powtarzany, aż cała tablica zostanie posortowana.
Złożoność obliczeniowa tego algorytmu wynosi $O(n \log n)$ zarówno w najlepszym, jak i najgorszym przypadku, co sprawia, że jest niezawodny bez względu na początkowy układ danych.

Schemat blokowy przedstawia kolejno algorytm tworzenia i rozbioru kopca:



gdzie:

d [ ] - Zbiór zawierający elementy do wstawienia do kopca. Numeracja elementów rozpoczyna się od 1

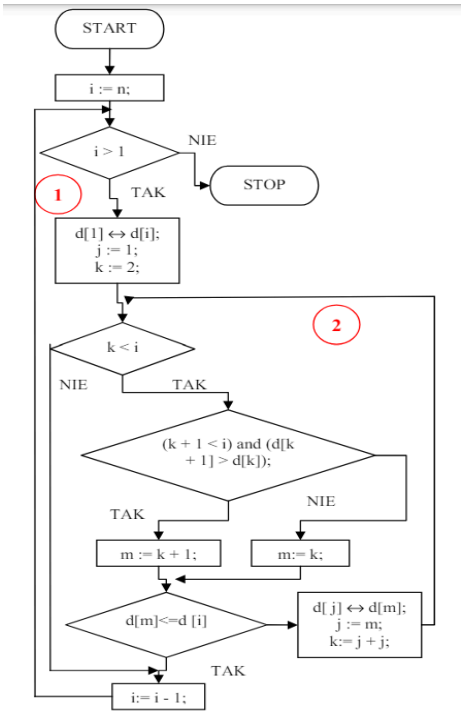
n - Ilość elementów w zbiorze, $n \in N$

i - zmienna licznikowa pętli umieszczającej kolejne elementy zbioru w kopcu, $i \in N$,

i ∈{2,3,...,n}

j,k - indeksy elementów leżących na ścieżce od wstawianego elementu do korzenia, j,k ∈ C

x - zmienna pomocnicza przechowująca tymczasowo element wstawiany do kopca



gdzie:

d [ ] - Zbiór zawierający poprawną strukturę kopca. Numeracja elementów rozpoczyna się od 1

n - Ilość elementów w zbiorze, n ∈N

i - indeksy elementów leżących na ścieżce w dół od korzenia, i ∈ N, i ∈{n, n-1 , ...2}

j,k - indeksy elementów leżących na ścieżce w dół od korzenia, j,k ∈ N

m - indeks większego z dwóch elementów potomnych, m ∈ N,

2.Implementacja

W projekcie wykorzystano różne biblioteki, aby zapewnić jego pełną funkcjonalność: ***iostream*** do obsługi wejścia i wyjścia, ***fstream*** do pracy z plikami, ***vector*** do używania dynamicznych tablic, ***string*** do manipulacji ciągami znaków, ***sstream*** do konwersji między typami danych, ***cstdlib*** i ***ctime*** do operacji związanych z czasem, oraz ***chrono*** do mierzenia czasu wykonania poszczególnych operacji. W celu lepszego zrozumienia i analizy tych algorytmów, dołączone zostały odpowiednie pliki nagłówkowe, które zawierają definicje funkcji oraz ich szczegółowe opisy

Pliki_obsluga.cpp

Klasa "***Pliki_obsluga***" jest odpowiedzialna za obsługę plików tekstowych, umożliwiając odczytanie danych z pliku i zapisanie wyników do innego pliku.

Pola prywatne:

- "***nazwa***" - string przechowujący nazwę pliku.
- "***dane***" - wektor liczb całkowitych przechowujący dane odczytane z pliku.

Deklaracja przyjacielskiej klasy:

- "***Algorytmy***" - pozwala klasie "Algorytmy" na dostęp do prywatnych członków klasy "*Pliki_obsluga*".

Metody publiczne:

- "***read_file()***" - odczytuje dane z pliku o nazwie "*nazwa*". Jeśli plik nie może zostać otwarty, użytkownik jest proszony o podanie poprawnej nazwy pliku. Dane są przechowywane w wektorze "*dane*".

```
void Pliki_obsluga::read_file() {
    do {
        std::ifstream plik(nazwa);
        if (!plik.is_open()) {
            std::cout << "Nie mozna otworzyc pliku" << std::endl;
            std::cout << "Wprowadz poprawna nazwe pliku: ";
            std::cin >> nazwa;
            nazwa += ".txt";
        }
        else {
            std::string line;
            while (std::getline(plik, line)) {
                std::stringstream ss(line);
                int number;
                while (ss >> number) {
                    dane.push_back(number);
                }
            }
            plik.close();
        }
    } while (dane.empty());
}
```

- "***zapisz_do_pliku***" - zapisuje posortowane dane do pliku "*wynik.txt*". Jeśli plik nie może zostać otwarty, wyświetla odpowiedni komunikat.

```
void Pliki_obsluga::zapisz_do_pliku(const std::vector<int>& sorted_data) {
    std::ofstream wynik("wynik.txt");
    if (wynik.is_open()) {
        for (int num : sorted_data) {
            wynik << num << " ";
        }
        wynik.close();
        std::cout << "Dane zostaly zapisane do pliku wynik.txt" << std::endl;
    }
    else {
        std::cout << "Nie mozna otworzyc pliku wynik.txt" << std::endl;
    }
}
```

- "***wyswietl_dane()***" - wyświetla dane przechowywane w wektorze "*dane*" na standardowym wyjściu.

```
void Pliki_obsluga::wyswietl_dane() {
    std::cout << "Liczby: " << std::endl;
    for (int num : dane) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
```

- "***get_data()***" - zwraca kopię danych przechowywanych w wektorze "*dane*", umożliwiając dostęp do nich innym klasom bez możliwości ich modyfikacji.
- "***set_data***" - ustawia nową wartość dla wektora "*dane*", co jest użyteczne po posortowaniu danych.

```
std::vector<int> Pliki_obsluga::get_data() const {
    return dane; // Zwracanie danych
}

void Pliki_obsluga::set_data(const std::vector<int>& new_data) {
    dane = new_data;
}
```

Klasa ta jest kluczowym elementem projektu, umożliwiającym bezpieczne i efektywne zarządzanie danymi wejściowymi i wyjściowymi, co stanowi podstawę dla dalszych operacji sortowania.

Heap_sort.cpp

Następnie zostały zaimplementowane oddzielne klasy dla każdego rodzaju sortowania.

- Klasa "***Heap_sort***"

  Klasa została utworzona do **sortowania przez kopcowanie**. Metoda "***wlasnosc_kopca***" służy do naprawiania właściwości kopca dla danego węzła. Natomiast "***kopiec_sort***" sortuje wektor liczb całkowitych poprzez najpierw zbudowanie kopca, a następnie iteracyjne usuwanie największego elementu z korzenia i przywracanie właściwości kopca.

```
void Heap_sort::wlasnosc_kopca(std::vector<int>& dane, int n, int i) {
    int korzen = i; // najwiekszy element jako korzen
    int left = 2 * i + 1; // lewe dziecko
    int right = 2 * i + 2; // prawe dziecko

    // czy lewe dziecko istnieje i czy jest wieksze od korzenia
    if (left < n && dane[left] > dane[korzen])
        korzen = left;

    // czy prawe dziecko istnieje i czy jest wieksze od korzenia lub lewego dziecka
    if (right < n && dane[right] > dane[korzen])
        korzen = right;

    // gdy najwiekszy element nie jest korzeniem
    if (korzen != i) {
        swap(dane[i], dane[korzen]);

        // rekurencyjnie przywracanie wlasnosci kopca
        wlasnosc_kopca(dane, n, korzen);
```

```cpp
void Heap_sort::kopiec_sort(std::vector<int>& dane) {
        auto start = chrono::steady_clock::now();

        int n = dane.size();

        // budowanie kopca
        for (int i = n / 2 - 1; i >= 0; i--)
            wlasnosc_kopca(dane, n, i);

        for (int i = n - 1; i > 0; i--) {
            // zamiana korzenia z ostatnim elementem
            swap(dane[0], dane[i]);

            // zmniejszenie kopca i przywracanie wlasnosci kopca
            wlasnosc_kopca(dane, i, 0);
        }
        auto end = chrono::steady_clock::now();
        auto diff = end - start;
        cout << "Czas sortowania: " << chrono::duration <double, milli>(diff).count() << " ms" << endl;

}
```

Merge_sort.cpp

- Klasa "***Merge_sort***"

  Klasa została zaprojektowana do realizacji **sortowania przez scalanie**. Metoda "***merge***" służy do łączenia dwóch podwektorów w jeden. Funkcja "***mergesort***" wywołuje się rekurencyjnie, dzieląc wektor na coraz mniejsze części, aż do momentu, gdy pojedyncze elementy zostaną posortowane.

```cpp
void Merge_sort::merge(std::vector<int>& A, int left, int mid, int right) {

    int x = mid - left + 1;
    int y = right - mid;

    // tymczasowe wektory temp
    vector<int> temp_L(x);
    vector<int> temp_R(y);

    // Kopiowanie danych do wektor◆w tymczasowych L[] i R[]
    for (int i = 0; i < x; i++)
        temp_L[i] = A[left + i];
    for (int j = 0; j < y; j++)
        temp_R[j] = A[mid + 1 + j];

    // scalanie wektor◆w temp
    int i = 0; // poczatakowy indeks pierwszego wektora temp
    int j = 0; // poczatakowy indeks drugiego wektora temp
    int k = left; // pierwszy indeks wektora do kt◆rego b◆dziemy ◆aczy◆
```

```cpp
while (i < x && j < y) {
    if (temp_L[i] <= temp_R[j]) {
        A[k] = temp_L[i];
        i++;
    }
    else {
        A[k] = temp_R[j];
        j++;
    }
    k++;
}
// wrzucamy pozosta◆e elementy kt◆re nie zosta◆y wrzucone
while (i < x) {
    A[k] = temp_L[i];
    i++;
    k++;
}
while (j < y) {
    A[k] = temp_R[j];
    j++;
    k++;
}
```

```
void Merge_sort::mergesort(std::vector<int>& A, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergesort(A, left, mid);
        mergesort(A, mid + 1, right);
        merge(A, left, mid, right);
    }
}
```

Bubble_sort.cpp

- Klasa "***Bubble_sort***"

  Odpowiada ona za sortowanie danych za pomocą algorytmu **sortowania bąbelkowego**. Metoda "***bubblesort***" przyjmuje referencję do wektora liczb całkowitych i sortuje go w miejscu. Dodatkowo, mierzony jest czas wykonania sortowania, który jest wyświetlany na standardowym wyjściu. Prywatna metoda "***swap***" służy do zamiany miejscami dwóch elementów w wektorze.

```
void Bubble_sort::bubblesort(std::vector<int>& arr) {
    auto start = chrono::steady_clock::now();
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
    auto end = chrono::steady_clock::now();
    auto diff = end - start;
    cout << "Czas sortowania: " << chrono::duration <double, milli>(diff).count() << " ms" << endl;
}

void Bubble_sort::swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Quick_sort.cpp

- Klasa "***Quick_sort***"

  Klasa implementuje algorytm **sortowania szybkiego** w swojej podstawowej wersji. Metoda "***quicksort***" przyjmuje referencję do wektora liczb całkowitych oraz indeksy początkowy (***low***) i końcowy (***high***) wektora. Funkcja "***partition***" dzieli wektor na dwie części względem elementu osiowego ***(pivot)***, zwracając indeks pivotu. Metoda "***swap***" służy do zamiany miejscami dwóch elementów w wektorze.

```
void Quick_sort::quicksort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
```

```cpp
int Quick_sort::partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void Quick_sort::swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Klasa "***Algorytmy***" zarządza wyborem i zastosowaniem różnych algorytmów sortowania na zestawie danych. Jej konstruktor przyjmuje nazwę pliku jako parametr i inicjuje obiekty odpowiedzialne za obsługę plików oraz różne algorytmy sortowania. Metoda "***menu()***" umożliwia użytkownikowi wybór konkretnego algorytmu, a następnie stosuje go do danych, wyświetlając czas sortowania oraz zapisując wyniki do pliku.

```cpp
class Algorytmy {
private:
    //tworzymy obiekty klas
    Pliki_obsluga plik;
    Bubble_sort bubble;
    Quick_sort quick;
    Merge_sort scalanie;
    Heap_sort kopiec;
public:
    //konstruktor
    Algorytmy(const string& file_name) : plik(file_name) {
        //odrazu odowlujemy sie zeby nam przeczytalo plik i wyswietlio pomocniczo ciag
        plik.read_file();
        plik.wyswietl_dane();
        // zabezpieczyc plik ze jak uzytkownik wprowadzi zla nazwe pliku to wtedy kaze mu wprowadzic nazwe jeszcze raz
        menu();
```

```cpp
// metoda odpowiadajaca za menu w ktorym uzytkownik  moze wybierac algorytm do uzycia
void menu() {
    int wybor;
    bool test = false;
    cout << "Wybierz algorytm ktory chcesz uzyc do sortowania swoich liczb poprzez wprowadznie z klawiatury liczby ktora stoi obok algorytmu" << endl;
    cout << "1. Sortowanie babelkowe" << endl;
    cout << "2. Sortowanie Szybkie" << endl;
    cout << "3. Sortowanie przez Scalanie" << endl;
    cout << "4. Sortowanie przez Kopcowanie" << endl;

    vector<int> dane = plik.get_data();
```

```cpp
vector<int> dane = plik.get_data();
//petla while zabezpieczajaca zeby uzytkownik nie wybral liczby innej niz z zakresu 1 - 4
do {
    cin >> wybor;
    //switch odpowiadajcy za wybor algorytmu przez uzytkownika
    switch (wybor) {
    case 1:
        bubble.bubblesort(dane);
        plik.set_data(dane);
        plik.wyswietl_dane();
        plik.zapisz_do_pliku(dane);
        test = true;
        return;

    case 2:
        quick.quicksort_czas(dane); //, 0, dane.size() - 1);
        plik.set_data(dane);
        plik.wyswietl_dane();
        plik.zapisz_do_pliku(dane);
        test = true;
        return;
```

```
    case 3:
        scalanie.merge_czas(dane); // , 0, ((dane.size()) / 2) - 1, dane.size() - 1);
        plik.set_data(dane);
        plik.wyswietl_dane();
        plik.zapisz_do_pliku(dane);
        test = true;
        return;

    case 4:
        kopiec.kopiec_sort(dane);
        plik.set_data(dane);
        plik.wyswietl_dane();
        plik.zapisz_do_pliku(dane);
        test = true;
        return;

    default:
        cout << "Wybrana liczba nie odpowiada zadnemu algorytmowi wybierz liczbe z zakresu 1 - 4" << endl;
        test = false;
    }
} while (!test);
```

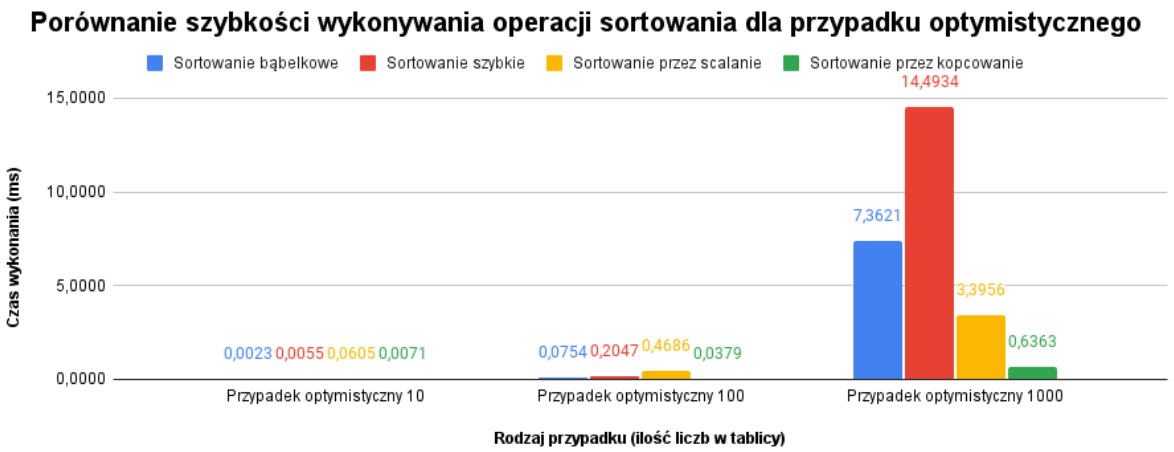
```
int main()
{
    string file_name;

    cout << "Wprowadz nazwe pliku z ciagiem liczb, ktory chcesz posortowac." << endl;
    cin >> file_name;

    // unikamy w ten sposob niepotrzebnego wprowdzania .txt po nazwie pliku
    file_name += ".txt";
    Algorytmy sortowanie(file_name);

    return 0;
}
```

3.Testy liczbowe

Test liczbowy przeprowadzony dla przypadku optymistycznego dla tablic o trzech różnych długościach.

Przypadek optymistyczny w kontekście algorytmów sortowania odnosi się do sytuacji, w której dane wejściowe są już uporządkowane lub prawie uporządkowane. W tym przypadku użyliśmy tablic uporządkowanych kolejno od 1 - 10, 1 - 100 oraz od 1 - 1000.

Rodzaj algorytmu	Tablica długości 10	Tablica długości 100	Tablica długości 1000
Sortowanie bąbelkowe	0,0023 ms	0,0754 ms	7,3621 ms
Sortowanie szybkie	0,0055 ms	0,2047 ms	14,4934 ms
Sortowanie przez scalanie	0,0605 ms	0,4686 ms	3,3956 ms
Sortowanie przez kopcowanie	0,0071 ms	0,0379 ms	0,6363 ms



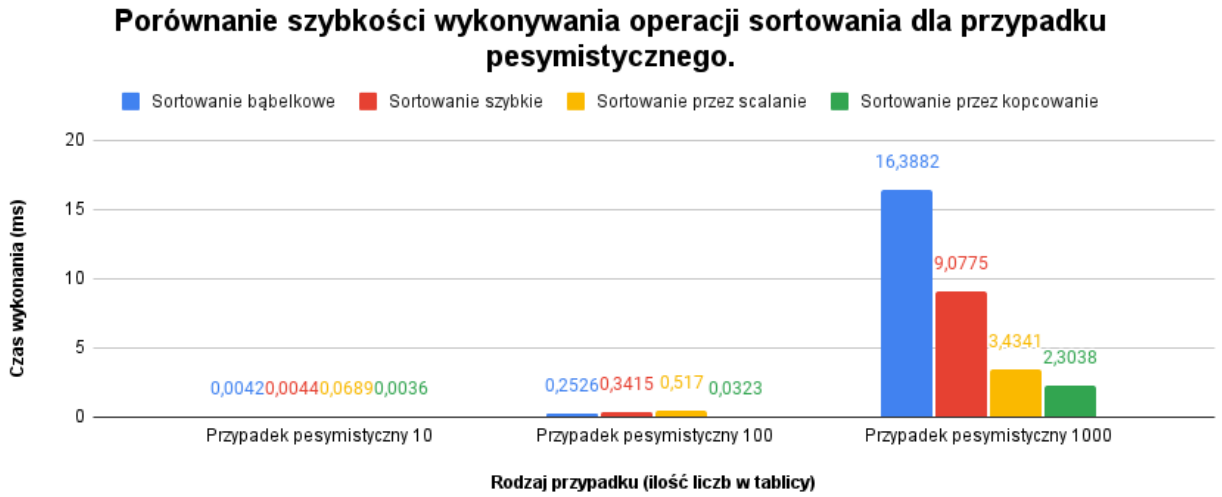
Oto wnioski jakie można wyciągnąć z przedstawionych powyżej danych:

- Sortowanie bąbelkowe w przypadku optymistycznym działa bardzo dobrze dla małych rozmiarów danych, ale czas wykonywania rośnie gwałtownie w miarę zwiększania się liczby elementów. Wynika to z tego, że choć dla prawie posortowanych danych działa szybko, to jego złożoność czasowa wciąż jest $O(n^2)$.
- W przypadku optymistycznym dla małych i średnich zbiorów danych sortowanie szybkie okazuje się mniej wydajne, mimo swojej optymalnej złożoności czasowej $O(n\ log\ n)$. Jego czas wykonania dla 1000 elementów jest najdłuższy spośród wszystkich algorytmów, co może sugerować, że dla prawie posortowanych danych inne algorytmy są bardziej efektywne.
- Sortowanie przez scalanie wydaje się być mniej wydajne dla małych danych w porównaniu do innych algorytmów, ale jego czas wzrasta wolniej w miarę wzrostu liczby elementów. Ma stabilną złożoność czasową $O(n\ log\ n)$.
- Sortowanie przez kopcowanie jest bardzo wydajne, zwłaszcza dla większych danych. Jego złożoność czasowa $O(n\ log\ n)$ sprawia, że czas wykonania faktycznie rośnie relatywnie wolno.

Test liczbowy przeprowadzony dla przypadku pesymistycznego dla tablic o trzech różnych długościach.

Przypadek pesymistyczny odnosi się do sytuacji, w której dane wejściowe są w najgorszym możliwym układzie dla danego algorytmu. W tym kontekście użyliśmy tablic posortowanych w odwrotnej kolejności, również dla przypadków 10, 100, 1000. Taki układ danych maksymalizuje liczbę operacji, które algorytm musi wykonać, co prowadzi do najdłuższego czasu wykonania.

Rodzaj algorytmu	Tablica długości 10	Tablica długości 100	Tablica długości 1000
Sortowanie bąbelkowe	0,0042 ms	0,2526 ms	16,3882 ms
Sortowanie szybkie	0,0044 ms	0,3415 ms	9,0775 ms
Sortowanie przez scalanie	0,0689 ms	0,517 ms	3,4341 ms
Sortowanie przez kopcowanie	0,0036 ms	0,0323 ms	2,3038 ms



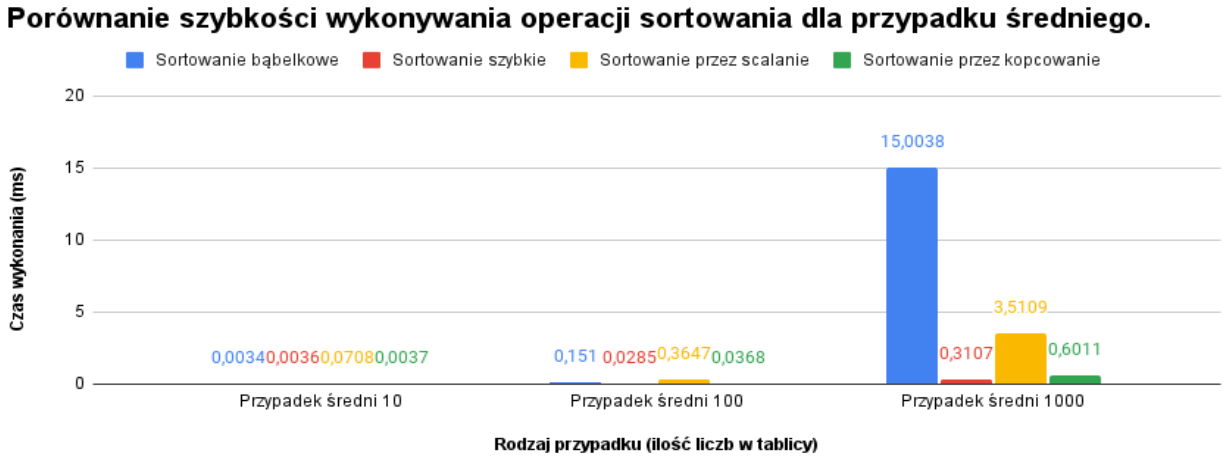
Wnioski z przedstawionych danych:

- Sortowanie bąbelkowe wykazuje dramatyczny wzrost czasu wykonania. Teoretyczna złożoność czasowa: $O(n^2)$. Faktyczne dane potwierdzają, że algorytm ma długi czas wykonania, szczególnie dla większych długości tablic, co jest zgodne z jego kwadratową złożonością czasową. Czyni go niepraktycznym dla większych zbiorów danych.
- Chociaż sortowanie szybkie radzi sobie lepiej niż bąbelkowe, jego wydajność w tym przypadku nadal spada, szczególnie dla większych danych. Teoretyczna średnia złożoność czasowa: $O(n \log n)$, ale w najgorszym przypadku może osiągać $O(n^2)$. Dla tablicy o długości 1000, czas wykonania wynosi 9,0775 ms, co jest zgodne z teoretyczną złożonością algorytmu szybkiego sortowania.
- Sortowanie przez scalanie ma stabilny czas wykonania, który rośnie stosunkowo wolno. Teoretyczna złożoność czasowa:$O(n \log n)$, co sugeruje, że algorytm powinien być efektywny dla różnych przypadków, rzeczywiste czasy wykonania potwierdzają jego stabilność i skuteczność.
- Sortowanie przez kopcowanie wykazuje najlepsze czasy wykonania w przypadku pesymistycznym, co czyni go najefektywniejszym algorytmem spośród analizowanych. Przewidywana złożoność czasowa wynosi $O(n \log n)$, a dane czasy wykonania potwierdzają jego skuteczność nawet w przypadku pesymistycznym.

Test liczbowy przeprowadzony dla przypadku średniego dla tablic o trzech różnych długościach.

W przypadku średnim dane wejściowe są w losowej kolejności lub w bardziej zróżnicowanym układzie niż w przypadku pesymistycznym, ale nie są również w optymalnym układzie dla danych algorytmów sortowania. Zastosowano tablice o długościach 10, 100 oraz 1000. W przypadkach 10 oraz 100 liczby sortowane są kolejno od 1, natomiast w przypadku 1000 występują też zdublowane dane w tablicy.

Rodzaj algorytmu	Tablica długości 10	Tablica długości 100	Tablica długości 1000
Sortowanie bąbelkowe	0,0034 ms	0,151 ms	15,0038 ms
Sortowanie szybkie	0,0036 ms	0,0285 ms	0,3107 ms
Sortowanie przez scalanie	0,0708 ms	0,3647 ms	3,5109 ms
Sortowanie przez kopcowanie	0,0037 ms	0,0368 ms	0,6011 ms



Wnioski:

- Spodziewana złożoność czasowa: $O(n^2)$. Pomimo niewielkiego czasu wykonania dla krótkich tablic, wydajność sortowania bąbelkowego drastycznie maleje wraz ze wzrostem rozmiaru danych. Dla tablicy o długości 1000 czas wykonania jest znacząco wyższy, co czyni ten algorytm niepraktycznym dla większych zbiorów danych.
- Sortowanie szybkie wykazuje znacznie lepszą wydajność niż sortowanie bąbelkowe dla wszystkich rozmiarów tablic. Średnia spodziewana złożoność czasowa: $O(n\ log\ n)$, ale w najgorszym przypadku może osiągać $O(n^2)$. Czas wykonania pozostaje niski, nawet dla większych danych, co czyni go dobrym wyborem dla przypadku średniego. Warto zauważyć, że faktyczne czasy wykonania są bliższe złożoności $O(n\ log\ n)$, co wskazuje na poprawność teoretycznej złożoności czasowej algorytmu szybkiego sortowania.
- Mimo nieco dłuższego czasu wykonania dla mniejszych tablic, algorytm sortowania przez scalanie staje się bardziej wydajny w miarę wzrostu rozmiaru danych. Spodziewana złożoność czasowa: $O(n\ log\ n)$. Dla tablic o długości 1000, osiąga akceptowalny czas wykonania.
- Sortowanie przez kopcowanie charakteryzuje się stabilnie niskim czasem wykonania dla wszystkich rozmiarów tablic. Spodziewana złożoność czasowa: $O(n\ log\ n)$. Jest to jedna z najlepszych opcji dla przypadku średniego, szczególnie dla większych zbiorów danych.

Dla małych zbiorów danych, jak w przypadku tablicy o długości 10, wszystkie algorytmy wykazują dobre wyniki, ale różnice zaczynają się uwidaczniać wraz z wzrostem rozmiaru danych.

4.Podsumowanie

W ramach przeprowadzonych testów porównawczych analizowano czas wykonywania operacji sortowania dla czterech algorytmów: sortowania bąbelkowego, sortowania szybkiego, sortowania przez scalanie oraz sortowania przez kopcowanie. Testy obejmowały trzy różne rozmiary tablic (10, 100, 1000 elementów) dla przypadków optymistycznych, pesymistycznych i średnich. Uzyskane wyniki pozwoliły na porównanie faktycznych czasów wykonania z teoretyczną złożonością czasową każdego z algorytmów.

1. **Sortowanie bąbelkowe**:
    - Złożoność czasowa: $O(n^2)$.
    - Wyniki testów pokazały, że sortowanie bąbelkowe ma najgorszą wydajność, szczególnie dla większych tablic, co jest zgodne z jego kwadratową złożonością czasową. Czas wykonania wzrasta drastycznie w przypadku tablic o długości 1000 elementów, osiągając nawet 16,3882 ms w przypadku pesymistycznym.
2. **Sortowanie szybkie**:
    - Złożoność czasowa: $O(n\ log\ n)$ średnio, $O(n^2)$ w najgorszym przypadku.
    - Algorytm szybki wykazał lepszą wydajność niż sortowanie bąbelkowe, jednak dla dużych tablic jego czas wykonania również wzrasta, szczególnie w przypadkach pesymistycznych. Dla tablicy o długości 1000 elementów czas wykonania wynosił 9,0775 ms, co potwierdza jego zmienną wydajność w porównaniu z przypadkiem średnim.
3. **Sortowanie przez scalanie**:
    - Złożoność czasowa: $O(n\ log\ n)$.
    - Testy potwierdziły, że sortowanie przez scalanie jest stabilne i efektywne dla różnych rozmiarów danych, zarówno w przypadkach optymistycznych, jak i pesymistycznych. Czas wykonania dla tablicy o długości 1000 elementów wynosił 3,4341 ms, co jest zgodne z teoretyczną wydajnością.
4. **Sortowanie przez kopcowanie**:
    - Złożoność czasowa: $O(n\ log\ n)$.
    - Algorytm przez kopcowanie wykazał najlepszą wydajność w większości testów, z najkrótszym czasem wykonania dla dużych tablic. Dla tablicy o długości 1000 elementów w przypadku pesymistycznym czas wykonania wynosił 2,3038 ms, co potwierdza jego efektywność i zgodność z przewidywaną złożonością czasową.

Podsumowując, przeprowadzone testy potwierdziły teoretyczne przewidywania dotyczące złożoności czasowej analizowanych algorytmów sortowania, co pozwala na ich świadomy wybór w zależności od wielkości i charakteru danych.

Źródła:

- https://eduinf.waw.pl/inf/alg/003_sort/0004.php
- https://eduinf.waw.pl/inf/alg/003_sort/0013.php
- https://eduinf.waw.pl/inf/alg/003_sort/0018.php
- https://informatyka.2ap.pl/ftp/3d/algorytmy/podr%C4%99cznik_algorytmy.pdf