AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Dokumentacja algorytmu sortowania MergeSort z zastosowaniem GitHub

Autor:
Blavitskyi Mykola

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

1.1. Cel projektu

Celem projektu jest zaprojektowanie, implementacja i przetestowanie algorytmu sortowania przez scalanie (*MergeSort*) w języku C++. Projekt ma na celu:

- **Implementację algorytmu sortowania przez scalanie:**
  - Podział tablicy na mniejsze części,
  - Rekurencyjne sortowanie podtablic,
  - Scalanie posortowanych części w jedną całość.
- **Przetestowanie algorytmu za pomocą testów jednostkowych:**
  - Sprawdzenie poprawności dla różnych przypadków danych (tablice rosnące, losowe, mieszane itp.),
  - Obsługa specjalnych przypadków (tablica pusta, jednoelementowa).
- **Zastosowanie biblioteki Google Test:**
  - Nauka konfiguracji środowiska testowego,
  - Pisanie testów jednostkowych w standardzie przemysłowym.
- **Strukturyzacja projektu:**
  - Oddzielenie kodu głównego od implementacji algorytmu,
  - Stworzenie modułów z nagłówkami i implementacją (`MergeSort.h`, `MergeSort.cpp`, `main.cpp`, `tests.cpp`).
- **Tworzenie dokumentacji:**
  - Generowanie dokumentacji technicznej za pomocą Doxygen,
  - Opis projektu w języku LaTeX.

Projekt realizuje założenia edukacyjne, takie jak nauka pracy z językiem C++, testowanie kodu oraz tworzenie dokumentacji w profesjonalny sposób.

2. Analiza problemu

2.1. Zastosowanie algorytmu sortowania przez scalanie

Algorytm sortowania przez scalanie (*MergeSort*) znajduje szerokie zastosowanie w różnych dziedzinach informatyki i technologii. Przykładowe zastosowania obejmują:

- Sortowanie dużych zbiorów danych w systemach bazodanowych,
- Porządkowanie danych w aplikacjach analitycznych i naukowych,
- Przygotowywanie danych do dalszych algorytmów, takich jak wyszukiwanie binarne,
- Optymalizację w algorytmach grafowych (np. algorytm Kruskala),
- Zastosowanie w środowiskach rozproszonych i zrównoleglonych, gdzie jego struktura jest szczególnie wydajna.

2.2. Opis działania algorytmu

Algorytm *MergeSort* działa zgodnie z zasadą **dziel i zwyciężaj**:

1. Tablica wejściowa jest rekurencyjnie dzielona na dwie mniejsze części aż do momentu, gdy każda część zawiera tylko jeden element.
2. Każda z mniejszych części jest porządkowana, a następnie scalana z innymi posortowanymi częściami.
3. Proces scalania polega na porównywaniu elementów z dwóch tablic i zapisywaniu ich w kolejności rosnącej w nowej tablicy wynikowej.

Dzięki tej metodzie algorytm osiąga złożoność czasową $\mathcal{O}(n \log n)$, co czyni go jednym z najbardziej wydajnych algorytmów sortowania.

2.3. Przykład działania algorytmu

Rozważmy tablicę wejściową:

$$\text{Tablica wejściowa: } [38, 27, 43, 3, 9, 82, 10]$$

Kroki działania algorytmu:

1. **Podział tablicy:**

$$[38, 27, 43, 3, 9, 82, 10] \rightarrow [38, 27, 43] \text{ i } [3, 9, 82, 10]$$

2. **Rekurencyjne dzielenie:**

$$[38, 27, 43] \rightarrow [38, 27] \text{ i } [43], \quad [38, 27] \rightarrow [38] \text{ i } [27]$$

3. **Scalanie:**

$$[38] \text{ i } [27] \rightarrow [27, 38], \quad [27, 38] \text{ i } [43] \rightarrow [27, 38, 43]$$

4. Podobny proces wykonywany jest dla prawej części tablicy, a następnie dwie części są scalane:

$$[27, 38, 43] \text{ i } [3, 9, 10, 82] \rightarrow [3, 9, 10, 27, 38, 43, 82]$$

Tablica wynikowa: $[3, 9, 10, 27, 38, 43, 82]$

2.4. Wykorzystanie narzędzi

W ramach projektu wykorzystano następujące narzędzia:

- **Google Test:** Biblioteka do testów jednostkowych w języku C++. Pozwala na szybkie i efektywne sprawdzanie poprawności działania kodu.
- **Doxygen:** Narzędzie do generowania dokumentacji technicznej na podstawie komentarzy w kodzie.
- **Git:** System kontroli wersji umożliwiający zarządzanie zmianami w kodzie, współpracę i przechowywanie projektu na platformie GitHub.
- **LaTeX:** Narzędzie do tworzenia profesjonalnych dokumentów tekstowych.

2.5. Implementacja w programie C++

Implementacja algorytmu sortowania przez scalanie została zrealizowana w języku C++. Program składa się z kilku modułów, które współpracują w celu zademonstrowania działania algorytmu:

- **MergeSort.h:** Plik nagłówkowy zawierający deklarację klasy `MergeSort`. Klasa definiuje metody takie jak `sort()` oraz funkcje pomocnicze, które realizują algorytm sortowania przez scalanie. Listing 1[1]

[1]Implementacja klasy MergeSort do sortowania przez scalanie[1].

```cpp
#pragma once
#include <vector>
#include <stdexcept>

class MergeSort {
public:
    static void sort(std::vector<int>& arr);

private:
    static void mergeSort(std::vector<int>& arr, int left, int right);
    static void merge(std::vector<int>& arr, int left, int middle, int right);
};

```

Listing 1. Fragment kodu - Implementacja klasy MergeSort do sortowania przez scalanie

- **MergeSort.cpp:** Plik źródłowy, który zawiera implementację metod z klasy `MergeSort`. Poniżej przykład funkcji `merge`, która scala dwie posortowane części tablicy. Listing 2[2]

```cpp
void MergeSort::merge(std::vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    std::vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; ++i) L[i] = arr[left + i];
    for (int i = 0; i < n2; ++i) R[i] = arr[middle + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        }
        else {
            arr[k++] = R[j++];
        }
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

[2]Funkcja merge w MergeSort[1].

```
21
```

Listing 2. Fragment kodu - Funkcja merge w MergeSort

- **main.cpp:** Główny punkt programu, w którym demonstrowane jest działanie algorytmu. Program najpierw wyświetla tablicę wejściową, następnie sortuje ją za pomocą klasy `MergeSort`, a na końcu wyświetla posortowaną tablicę. Listing 3[3]

```
1  #include <iostream>
2  #include "MergeSort.h"
3
4  int main() {
5      std::vector<int> arr = {38, 27, 43, 3, 9, 82, 10};
6
7      std::cout << "Tablica przed sortowaniem: ";
8      for (const auto& num : arr) std::cout << num << " ";
9      std::cout << std::endl;
10
11     MergeSort::sort(arr);
12
13     std::cout << "Tablica po sortowaniu: ";
14     for (const auto& num : arr) std::cout << num << " ";
15     std::cout << std::endl;
16
17     return 0;
18 }
19
```

Listing 3. Fragment kodu - Przykład użycia algorytmu MergeSort

- **test.cpp:** Plik zawierający testy jednostkowe napisane z wykorzystaniem biblioteki Google Test. Testy weryfikują poprawność działania algorytmu w różnych przypadkach. Na przykład, test sprawdzający poprawność sortowania tablicy z liczbami ujemnymi wygląda następująco. Listing 4[4]

```
1  TEST(MergeSortTest, NegativeNumbersArray) {
2      std::vector<int> arr = { -3, -1, -4, -2, -5 };
3      MergeSort::sort(arr);
4      EXPECT_EQ(arr, (std::vector<int>{-5, -4, -3, -2, -1}));
5  }
6
```

Listing 4. Fragment kodu - Test dla tablicy z liczbami ujemnymi

[3]Przykład użycia algorytmu MergeSort[1].
[4]Test dla tablicy z liczbami ujemnymi[1].

3. Projektowanie

W tym rozdziale przedstawiamy proces projektowania algorytmu sortowania przez scalanie, który został zaimplementowany w programie C++. Algorytm ten działa na tablicach liczb całkowitych, sortując je w porządku rosnącym. Główne elementy projektu to klasa `MergeSort`, która zawiera metody do sortowania tablicy, oraz metoda pomocnicza do scalania dwóch części tablicy.

3.1. Struktura klasy `MergeSort`

Klasa `MergeSort` jest głównym elementem programu, odpowiedzialnym za wykonanie algorytmu sortowania przez scalanie. Klasa ta zawiera:

- **Metoda `sort()`:** Jest to metoda publiczna, która jest wykorzystywana do wywołania algorytmu sortowania. W metodzie tej sprawdzamy, czy tablica nie jest pusta, a następnie wywołujemy rekurencyjnie funkcję `mergeSort()`.
- **Metoda `mergeSort()`:** Jest to funkcja rekurencyjna, która dzieli tablicę na mniejsze części i rekurencyjnie je sortuje. Po podzieleniu tablicy na dwie części, wywoływana jest funkcja `merge()` do połączenia tych części w posortowaną tablicę.
- **Metoda `merge()`:** Funkcja scalająca dwie posortowane części tablicy. Na podstawie porównań elementów z obu części tablicy, wynikowy zbiór jest wstawiany do tablicy wejściowej w odpowiedniej kolejności.

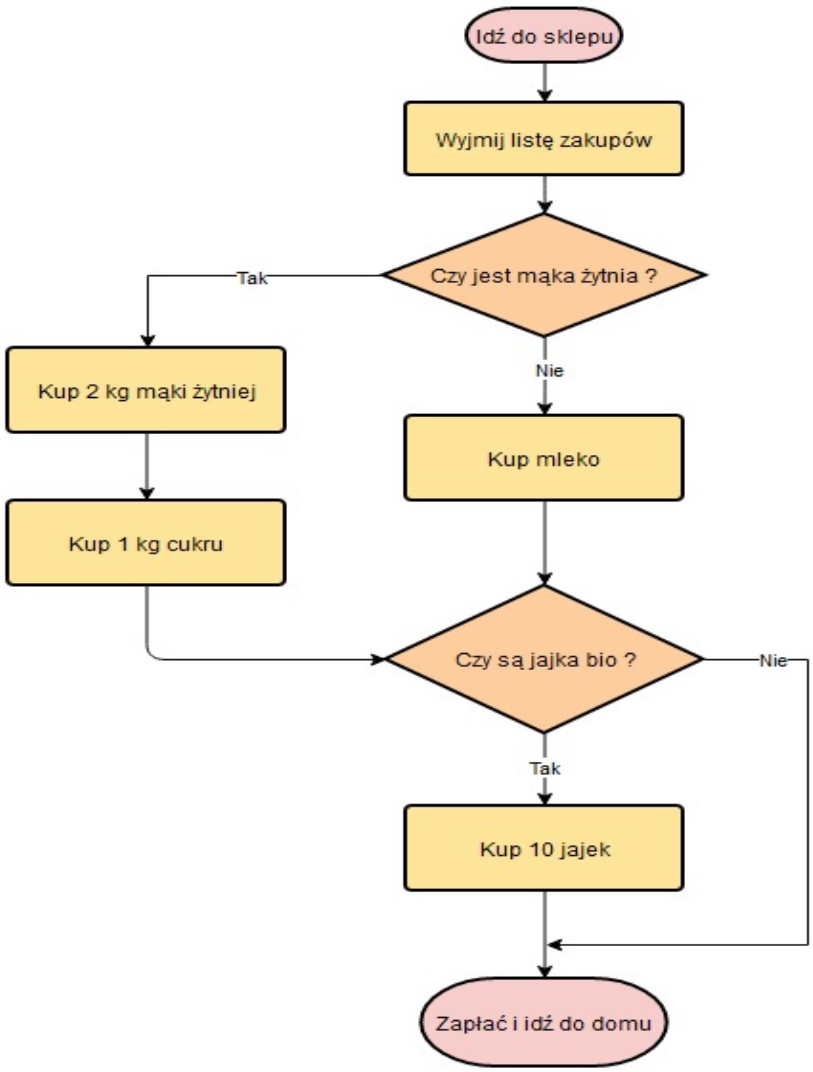
3.2. Przebieg działania algorytmu

Algorytm sortowania przez scalanie działa na zasadzie dziel i zwyciężaj. Oto jak przebiega jego działanie:

1. Jeśli tablica ma więcej niż jeden element, dzielimy ją na dwie części. Dzielimy tablicę aż do momentu, kiedy każda część będzie miała tylko jeden element.
2. Każdy z tych podzbiorów jest już posortowany, ponieważ zawiera tylko jeden element.
3. Następnie zaczynamy scalać poszczególne części, porównując je i łącząc w sposób, który zapewnia posortowanie.
4. Proces scalania powtarza się, aż cała tablica będzie posortowana.

3.3. Diagram blokowy

Aby lepiej zobrazować przebieg algorytmu, można stworzyć diagram blokowy, który pokazuje sposób podziału tablicy oraz proces scalania. Diagram przedstawia kolejne kroki algorytmu, począwszy od dzielenia tablicy na mniejsze części, aż po scalanie poszczególnych elementów w całość. Pokazane na rysunku 3.1[5]



Rys. 3.1. Diagram blokowy przedstawiający algorytm sortowania przez scalanie.

[5]Diagram blokowy przedstawiający algorytm sortowania przez scalanie.[1].

Składniki diagramu:

1. **Podział tablicy na mniejsze części**
   Diagram przedstawia proces rekurencyjnego dzielenia tablicy na pół, aż do osiągnięcia pojedynczych elementów, co jest istotne dla algorytmu.

2. **Scalanie elementów**
   Po podziale następuje proces scalania mniejszych części. Jest to kluczowa część algorytmu, w której elementy są łączone w uporządkowane podtablice, aż do utworzenia pełnej posortowanej tablicy.

3. **Warunki stopu i rekurencja**
   Bloki decyzyjne wskazują warunek stopu (np. „Czy tablica ma więcej niż jeden element?"). Jeśli nie, proces dzielenia się kończy i następuje scalanie.

4. **Przepływ danych i strzałki**
   Strzałki wskazują kolejność działań – od dzielenia tablicy na etapie początkowym, aż po scalanie na końcu.

Przydatność diagramu

Taki diagram pozwala łatwiej zrozumieć działanie algorytmu, co jest szczególnie przydatne w nauce podstaw algorytmów oraz przy implementacji kodu. Pomaga również w debugowaniu i analizie działania programu.

3.4. Projektowanie testów

Aby upewnić się, że zaimplementowany algorytm działa poprawnie, zaplanowano szereg testów jednostkowych. Testy te sprawdzają różne przypadki, w tym:

- Sortowanie już posortowanej tablicy,
- Sortowanie tablicy posortowanej w odwrotnej kolejności,
- Sortowanie tablicy z losowymi liczbami,
- Sortowanie tablicy zawierającej liczby ujemne,
- Sortowanie tablicy zawierającej zarówno liczby ujemne, jak i dodatnie,
- Obsługę pustych tablic,

- Obsługę tablicy z jednym elementem,
- Sortowanie tablicy z duplikatami.

Każdy z tych testów jest realizowany przy użyciu biblioteki Google Test, co umożliwia automatyczne sprawdzenie poprawności działania algorytmu w różnych sytuacjach.

3.5. Git

Git to system kontroli wersji, który umożliwia zarządzanie historią zmian w kodzie źródłowym. Aby efektywnie korzystać z Git, wykonaj następujące kroki:

- **Inicjalizacja repozytorium:** Użyj komendy `git init` w katalogu projektu, aby utworzyć nowe repozytorium.
- **Dodawanie plików:** Użyj `git add .`, aby dodać wszystkie pliki do repozytorium.
- **Tworzenie commitów:** Użyj `git commit -m "Opis zmian"`, aby zapisać zmiany z odpowiednim komentarzem.
- **Tworzenie gałęzi:** Użyj `git branch nazwa_gałęzi`, aby utworzyć nową gałąź.
- **Przełączanie gałęzi:** Użyj `git checkout nazwa_gałęzi`, aby przełączyć się na inną gałąź.
- **Wysyłanie zmian na GitHub:** Użyj `git push origin nazwa_gałęzi`, aby wysłać zmiany na zdalne repozytorium.

3.6. Doxygen

Doxygen jest narzędziem do generowania dokumentacji z kodu źródłowego. Aby z niego skorzystać:

- **Instalacja:** Zainstaluj Doxygen na swoim systemie.
- **Konfiguracja:** Utwórz plik konfiguracyjny `Doxyfile` przy użyciu polecenia `doxygen -g`.
- **Dodawanie komentarzy:** Używaj specjalnych komentarzy w kodzie źródłowym, aby opisać funkcje, klasy i zmienne. Przykład:

```
/**
 * @brief Funkcja dodaje dwie liczby.
 * @param a Pierwsza liczba.
 * @param b Druga liczba.
 * @return Suma a i b.
 */
int add(int a, int b) {
    return a + b;
}
```

- **Generowanie dokumentacji:** Użyj polecenia `doxygen Doxyfile`, aby wygenerować dokumentację w formacie HTML lub PDF.

3.7. Overleaf

Overleaf to platforma do edytowania dokumentów w LaTeX. Aby z niej skorzystać:

- **Rejestracja:** Załóż konto na stronie Overleaf.
- **Tworzenie projektu:** Utwórz nowy projekt i wybierz szablon dokumentu.
- **Edycja:** Edytuj dokument w edytorze tekstu. Zmiany są automatycznie zapisywane.
- **Współpraca:** Zaproś innych użytkowników do współpracy, udostępniając link do projektu.
- **Eksport:** Eksportuj dokument do formatu PDF lub innego formatu, korzystając z opcji eksportu.

4. Implementacja

4.1. Wprowadzenie

Algorytm sortowania przez scalanie (Merge Sort) jest jednym z klasycznych algorytmów sortowania, który wykorzystuje podejście dziel i zwyciężaj. Jego główną zaletą jest stabilność i gwarantowana złożoność czasowa *O(n log n)* w najgorszym przypadku, co czyni go jednym z efektywniejszych algorytmów sortowania dla dużych zbiorów danych.

W implementacji algorytmu MergeSort w C++ posługujemy się klasą `MergeSort`, która zawiera dwie funkcje pomocnicze:

- `sort()` – funkcja publiczna, która jest wywoływana w celu posortowania tablicy.
- `mergeSort()` – funkcja rekurencyjna, która dzieli tablicę na mniejsze części.
- `merge()` – funkcja, która scala dwie posortowane części tablicy w jedną, zachowując porządek.

Celem implementacji było zapewnienie efektywności algorytmu oraz jego użyteczności w różnych przypadkach testowych. Zostały przygotowane testy jednostkowe, które sprawdzają działanie algorytmu na różnych typach danych, takich jak: tablice już posortowane, odwrotnie posortowane, tablice z liczbami ujemnymi, tablice z duplikatami i tablice puste.

4.2. Implementacja algorytmu MergeSort

Algorytm sortowania przez scalanie w C++ jest zrealizowany w trzech głównych plikach:

- `MergeSort.h` – plik nagłówkowy zawierający deklarację klasy `MergeSort` i jej metod.
- `MergeSort.cpp` – plik źródłowy zawierający implementację funkcji sortujących.
- `main.cpp` – plik, w którym testujemy algorytm na przykładowych danych.

W poniższym przykładzie kodu przedstawiamy implementację klasy `MergeSort`, która implementuje algorytm sortowania przez scalanie. Listing 5[6]

[6]Deklaracja klasy MergeSort[1].

```cpp
#include <vector>
#include <stdexcept>

// Klasa implementuj ca algorytm sortowania przez scalanie
class MergeSort {
public:
    static void sort(std::vector<int>& arr);

private:
    static void mergeSort(std::vector<int>& arr, int left, int right);
    static void merge(std::vector<int>& arr, int left, int middle, int right);
};
```

Listing 5. Fragment kodu - Deklaracja klasy MergeSort

`MergeSort::sort()` jest funkcją publiczną, która sprawdza, czy tablica nie jest pusta, a następnie wywołuje funkcję rekurencyjną `mergeSort`, która dzieli tablicę na mniejsze podtablice. Funkcja `merge()` scala dwie posortowane części tablicy.

Przykład funkcji `mergeSort()`. Listing 6[7]

```cpp
void MergeSort::mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;
        mergeSort(arr, left, middle);   // Sortuje lew  cz
        mergeSort(arr, middle + 1, right);  // Sortuje praw cz
        merge(arr, left, middle, right);  // Scal dwie posortowane cz  ci
    }
}
```

Listing 6. Fragment kodu - Funkcja rekurencyjna mergeSort

Funkcja `merge()` scala dwie posortowane tablice w jedną, zapewniając ich prawidłowy porządek. Listing 7[8]

```cpp
void MergeSort::merge(std::vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

```

[7]Funkcja rekurencyjna mergeSort[1].
[8]Funkcja scalająca merge[1].

```
5       std::vector<int> L(n1), R(n2);
6       for (int i = 0; i < n1; ++i) L[i] = arr[left + i];
7       for (int i = 0; i < n2; ++i) R[i] = arr[middle + 1 + i];
8
9       int i = 0, j = 0, k = left;
10      while (i < n1 && j < n2) {
11          if (L[i] <= R[j]) {
12              arr[k++] = L[i++];
13          }
14          else {
15              arr[k++] = R[j++];
16          }
17      }
18      while (i < n1) arr[k++] = L[i++];
19      while (j < n2) arr[k++] = R[j++];
20  }
```

Listing 7. Fragment kodu - Funkcja scalająca merge

W ten sposób algorytm dzieli tablicę na mniejsze części, aż do momentu, gdy każda część zawiera tylko jeden element, a następnie scala je w większe, posortowane części, tworząc pełną, posortowaną tablicę.

4.3. Testowanie algorytmu

Aby upewnić się, że implementacja algorytmu działa prawidłowo, przeprowadziliśmy szereg testów jednostkowych. Testy obejmowały różne przypadki:

- Tablica już posortowana.
- Tablica posortowana w odwrotnej kolejności.
- Losowa tablica liczb całkowitych.
- Tablica zawierająca liczby ujemne.
- Tablica z liczbami ujemnymi i dodatnimi.
- Tablica z duplikatami liczb.
- Pusta tablica.
- Tablica z jednym elementem.

Testy zostały zaimplementowane przy pomocy frameworka Google Test, co pozwoliło na automatyczne sprawdzenie poprawności działania algorytmu. Listing 8[9]

```
#include "pch.h"
#include "MergeSort.h"
#include <vector>

// Test dla posortowanej rosn co tablicy
TEST(MergeSortTest, AlreadySortedArray) {
    std::vector<int> arr = { 1, 2, 3, 4, 5 };
    MergeSort::sort(arr);
    EXPECT_EQ(arr, (std::vector<int>{1, 2, 3, 4, 5}));
}

// Test dla odwrotnie posortowanej tablicy
TEST(MergeSortTest, ReverseSortedArray) {
    std::vector<int> arr = { 5, 4, 3, 2, 1 };
    MergeSort::sort(arr);
    EXPECT_EQ(arr, (std::vector<int>{1, 2, 3, 4, 5}));
}
```

Listing 8. Fragment kodu - Testy jednostkowe algorytmu MergeSort

Powyższe testy sprawdzają, czy algorytm poprawnie sortuje różne typy tablic, w tym tablice posortowane rosnąco, odwrotnie, tablice z duplikatami i tablice z liczbami ujemnymi.

[9]Testy jednostkowe algorytmu MergeSort[1].

5. Wnioski

5.1. Wnioski Końcowe

Algorytm sortowania przez scalanie (Merge Sort) jest efektywnym rozwiązaniem do sortowania dużych zbiorów danych. Jego główną zaletą jest gwarantowana złożoność czasowa *O(n log n)* w przypadku wszystkich rodzajów danych, co sprawia, że jest on stabilny i sprawdzony w wielu zastosowaniach, szczególnie gdy dane są bardzo rozproszone lub gdy musimy pracować z dużymi zbiorami danych.

Podczas implementacji algorytmu MergeSort w C++ udało się osiągnąć poprawność działania algorytmu, co potwierdzają wyniki testów jednostkowych. Testy te obejmowały różne przypadki, takie jak:

- Tablice już posortowane,
- Tablice posortowane w odwrotnej kolejności,
- Tablice z duplikatami,
- Tablice zawierające liczby ujemne,
- Tablice puste,
- Tablice z jednym elementem.

Testy przeprowadzono przy pomocy frameworka Google Test, który pozwolił na automatyczne sprawdzenie poprawności algorytmu w różnych scenariuszach.

Implementacja algorytmu MergeSort w języku C++ z wykorzystaniem klas i funkcji statycznych jest elastyczna, czytelna oraz efektywna. Jednym z wyzwań, które zostały napotkane podczas realizacji projektu, była optymalizacja zarządzania pamięcią, zwłaszcza przy pracy z dużymi tablicami, ale algorytm MergeSort, mimo że wymaga dodatkowej przestrzeni, nadal wykazuje się bardzo dobrą wydajnością w porównaniu do innych algorytmów sortowania o podobnej złożoności czasowej.

W przyszłości można rozważyć zaimplementowanie bardziej zaawansowanych metod, takich jak adaptacyjne algorytmy sortowania, które mogą poprawić wydajność w zależności od struktury danych wejściowych. Możliwe jest także dalsze testowanie algorytmu w bardziej złożonych scenariuszach, jak np. przy dużych tablicach lub w połączeniu z równoległym przetwarzaniem.

Podsumowując, MergeSort jest solidnym rozwiązaniem w dziedzinie algorytmów sortujących i może być stosowany zarówno w zadaniach o mniejszych wymaganiach czasowych, jak i w aplikacjach wymagających wysokiej wydajności.

Bibliografia

[1] Blavitskyi Mykola. *Student*. Broshniv - Osada, 2005.

Spis rysunków

Spis tabel

Spis listingów