AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Implementacja algorytmu sortowania przez scalanie (Merge Sort) z wykorzystaniem szablonów i testów jednostkowych

Autor:
Sebastian Tatara

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań **4**

1.1. Wymagania funkcjonalne 4
1.2. Specyfikacja testów jednostkowych 4
1.3. Narzędzia i środowisko 5

2. Analiza problemu **6**

2.1. Zastosowanie i charakterystyka algorytmu 6
2.2. Zasada działania 6
2.3. Przykład obliczeniowy (Symulacja ręczna) 7
2.4. Opis wykorzystanych narzędzi 8
2.4.1. Google Test (GTest) 8
2.4.2. System kontroli wersji Git 8

3. Projektowanie **9**

3.1. Środowisko i narzędzia 9
3.2. Konfiguracja kompilatora i bibliotek 9
3.3. Projekt architektury (Diagram Klas) 10
3.4. Sposób wykorzystania narzędzi 10
3.4.1. System kontroli wersji Git 10
3.4.2. Automatyzacja uruchamiania 11

4. Implementacja **12**

4.1. Kluczowy algorytm (Scalanie) 12
4.2. Implementacja testów 13
4.3. Wykorzystanie szablonów w praktyce 13
4.4. Wyniki działania 14
4.4.1. Weryfikacja poprawności 14
4.4.2. Demonstracja działania 15

5. Wnioski **17**

Literatura **18**

Spis rysunków **19**

Spis tabel **20**

Spis listingów **21**

1. Ogólne określenie wymagań

Niniejszy dokument stanowi dokumentację projektową aplikacji realizującej algorytm sortowania przez scalanie (*Merge Sort*). Głównym celem projektu jest stworzenie elastycznej implementacji algorytmu w języku C++ przy użyciu szablonów klas (*templates*), co pozwoli na sortowanie danych różnych typów (m.in. `int`, `double`).

Projekt ma na celu nie tylko implementację samego algorytmu, ale również weryfikację jego poprawności poprzez zestaw automatycznych testów jednostkowych stworzonych z wykorzystaniem biblioteki Google Test.

1.1. Wymagania funkcjonalne

Aplikacja musi spełniać szereg wymagań funkcjonalnych i niefunkcjonalnych określonych w specyfikacji zadania. Do najważniejszych z nich należą:

1. **Implementacja obiektowa:** Algorytm musi być zaimplementowany wewnątrz klasy.
2. **Generyczność:** Klasa musi wykorzystywać szablony (`template`), aby obsługiwać różne typy liczbowe.
3. **Program demonstracyjny:** Należy stworzyć plik `main.cpp`, w którym zostaną utworzone i posortowane dwie instancje tablic (dla liczb całkowitych i zmiennoprzecinkowych).
4. **Środowisko testowe:** Wykorzystanie frameworka Google Test do weryfikacji poprawności kodu.

1.2. Specyfikacja testów jednostkowych

Zgodnie z wymaganiami, algorytm musi zostać poddany rygorystycznym testom. Zaplanowano wykonanie 13 scenariuszy testowych sprawdzających czy algorytm:

- Zachowuje tablicę niezmienioną, gdy jest już posortowana.
- Poprawnie sortuje tablicę posortowaną odwrotnie (malejąco).
- Radzi sobie z losowym układem liczb.
- Poprawnie sortuje tablice zawierające wyłącznie liczby ujemne.
- Obsługuje tablice z liczbami mieszanymi (ujemne i dodatnie).

- Jest odporny na podanie pustej tablicy (nie rzuca wyjątkami).
- Prawidłowo obsługuje tablice jednoelementowe.
- Poprawnie sortuje tablice zawierające duplikaty (liczb dodatnich oraz ujemnych).
- Sortuje małe tablice (2 elementy) oraz bardzo duże zbiory danych (powyżej 100 elementów).

1.3. Narzędzia i środowisko

Projekt jest realizowany przy użyciu nowoczesnych narzędzi programistycznych:

- **Język:** C++ (Standard C++14).
- **System budowania:** CMake – do automatyzacji procesu kompilacji.
- **IDE:** Visual Studio Code / Visual Studio.
- **Kontrola wersji:** Git oraz GitHub.
- **Dokumentacja:** System LaTeX do składu tekstu oraz Doxygen do dokumentacji kodu.

2. Analiza problemu

W niniejszym rozdziale przedstawiono teoretyczne podstawy algorytmu sortowania przez scalanie (*Merge Sort*), analizę jego złożoności oraz opis narzędzi wykorzystanych w projekcie.

2.1. Zastosowanie i charakterystyka algorytmu

Sortowanie przez scalanie to jeden z podstawowych algorytmów sortowania, opisany szeroko w literaturze fachowej [1]. Należy do rodziny algorytmów „dziel i zwyciężaj" (*divide and conquer*).

Algorytm ten znajduje zastosowanie wszędzie tam, gdzie kluczowa jest:

- **Stabilność sortowania:** Kolejność elementów o tych samych kluczach nie ulega zmianie. Jest to istotne np. przy wielokrotnym sortowaniu baz danych według różnych kryteriów.
- **Gwarantowana złożoność:** W przeciwieństwie do QuickSort, Merge Sort ma pesymistyczną złożoność $O(n \log n)$, co czyni go przewidywalnym [1].
- **Sortowanie list połączonych:** Merge Sort działa bardzo efektywnie na listach, nie wymagając dodatkowej pamięci (w przeciwieństwie do tablic).
- **Przetwarzanie równoległe:** Ze względu na niezależność sortowania podzbiorów, algorytm łatwo zrównoleglić.

2.2. Zasada działania

Działanie programu opiera się na rekurencji. Proces sortowania tablicy A można opisać w trzech krokach [1]:

1. **Podział (Divide):** Jeśli tablica ma więcej niż jeden element, dzielimy ją na dwie połowy: lewą L i prawą R.
2. **Zwyciężanie (Conquer):** Rekurencyjnie wywołujemy sortowanie dla obu połów (L i R).
3. **Połączenie (Combine):** Scalamy dwie posortowane już podtablice w jedną tablicę wynikową, wybierając kolejno mniejsze elementy z L lub R.

2.3. Przykład obliczeniowy (Symulacja ręczna)

Aby zobrazować działanie algorytmu, prześledźmy proces sortowania dla przykładowego zbioru danych:

$$A = [38, 27, 43, 3, 9, 82, 10]$$

Krok 1: Podział rekurencyjny Tablica jest dzielona na mniejsze fragmenty aż do uzyskania tablic jednoelementowych:

1. $[38, 27, 43, 3, 9, 82, 10] \xrightarrow{\text{podział}} [38, 27, 43, 3]$ oraz $[9, 82, 10]$
2. $[38, 27, 43, 3] \xrightarrow{\text{podział}} [38, 27]$ oraz $[43, 3]$
3. $[38, 27] \xrightarrow{\text{podział}} [38]$ oraz $[27]$

Krok 2: Scalanie (Merge) Teraz następuje proces łączenia i porządkowania:

1. Scalamy $[38]$ i $[27]$. Porównujemy: $27 < 38$.

Wynik: $[27, 38]$

2. Scalamy $[43]$ i $[3]$. Porównujemy: $3 < 43$.

Wynik: $[3, 43]$

3. Scalamy $[27, 38]$ i $[3, 43]$.

- Porównujemy 27 i 3 $\rightarrow$ wybieramy 3.
- Porównujemy 27 i 43 $\rightarrow$ wybieramy 27.
- Porównujemy 38 i 43 $\rightarrow$ wybieramy 38.
- Zostało 43 $\rightarrow$ wybieramy 43.

Wynik pośredni: $[3, 27, 38, 43]$

Analogicznie postępujemy dla drugiej połowy $[9, 82, 10]$, uzyskując $[9, 10, 82]$.

Krok 3: Scalanie końcowe Na koniec scalamy dwie duże połówki: $[3, 27, 38, 43]$ oraz $[9, 10, 82]$.

- $3 < 9 \rightarrow [3]$
- $9 < 27 \rightarrow [3, 9]$
- $10 < 27 \rightarrow [3, 9, 10]$

- $27 < 82 \rightarrow [3, 9, 10, 27]$
- ... i tak dalej.

Ostateczny wynik:

$$[3, 9, 10, 27, 38, 43, 82]$$

2.4. Opis wykorzystanych narzędzi

Zgodnie z wymaganiami projektowymi, w pracy wykorzystano specjalistyczne narzędzia wspierające proces wytwarzania oprogramowania.

2.4.1. Google Test (GTest)

Google Test to biblioteka dla języka C++ służąca do tworzenia i uruchamiania testów jednostkowych [2]. W projekcie wykorzystano ją do automatycznej weryfikacji poprawności algorytmu. GTest udostępnia makra asercji, takie jak `EXPECT_EQ` (oczekuj równości) czy `EXPECT_TRUE` (oczekuj prawdy), które pozwalają sprawdzić, czy wynik zwracany przez funkcję sortującą jest zgodny z oczekiwaniami (np. czy tablica jest posortowana rosnąco).

2.4.2. System kontroli wersji Git

Git to rozproszony system kontroli wersji, który pozwala śledzić zmiany w kodzie źródłowym. W projekcie użyto go do:

- Rejestrowania postępów prac (commity).
- Zarządzania wersjami plików.
- Synchronizacji kodu z serwisem GitHub [3], co umożliwia zdalny dostęp do repozytorium.

3. Projektowanie

Faza projektowania obejmowała dobór odpowiednich narzędzi programistycznych, konfigurację środowiska oraz zaprojektowanie struktury klas realizujących algorytm.

3.1. Środowisko i narzędzia

Do realizacji projektu wybrano zestaw narzędzi zapewniający przenośność kodu oraz automatyzację procesu testowania. Wykorzystano:

- **Język programowania:** C++ w standardzie C++14, zgodnie z zaleceniami twórcy języka [4].
- **Kompilator:** GCC (MinGW-w64) – popularny kompilator dla systemów Windows.
- **System budowania:** CMake – narzędzie do automatyzacji procesu kompilacji [5].
- **Biblioteki:** Google Test (GTest) – framework do testów jednostkowych [2].
- **Kontrola wersji:** Git – do śledzenia historii zmian [3].

3.2. Konfiguracja kompilatora i bibliotek

Proces kompilacji jest sterowany przez plik `CMakeLists.txt`. Kluczowym elementem konfiguracji jest automatyczne pobieranie i dołączanie biblioteki Google Test przy użyciu modułu `FetchContent`, co jest nowoczesną metodą zarządzania zależnościami w CMake [5].

Szczegółową konfigurację CMake, w tym wymuszenie standardu C++14 oraz linkowanie bibliotek, przedstawia Listing 1.

```
cmake_minimum_required(VERSION 3.14)
project(MergeSortProject)

set(CMAKE_CXX_STANDARD 14)

# Pobieranie biblioteki Google Test
include(FetchContent)
FetchContent_Declare(
  googletest
  URL https://github.com/google/googletest/archive/refs/heads/main.
   zip
```

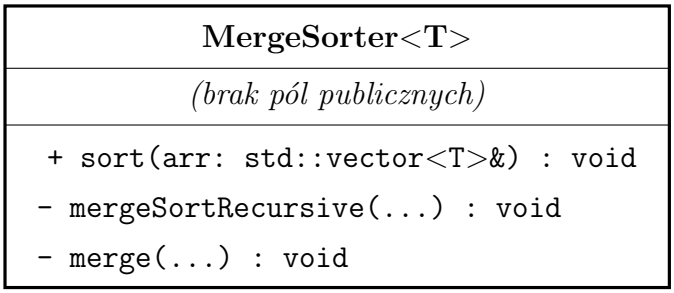
```
)
FetchContent_MakeAvailable(googletest)

# Definicja pliku wykonywalnego testow
add_executable(MergeSortTests test.cpp)
target_link_libraries(MergeSortTests gtest gtest_main)
```

Listing 1. Konfiguracja CMake (CMakeLists.txt)

3.3. Projekt architektury (Diagram Klas)

Głównym elementem systemu jest szablonowa klasa `MergeSorter`. Zaprojektowano ją zgodnie z paradygmatem programowania obiektowego, ukrywając szczegóły implementacyjne (metody pomocnicze) w sekcji prywatnej.

Schemat budowy klasy oraz jej metody przedstawiono na Rysunku 3.1.



Rys. 3.1. Diagram klasy MergeSorter

Publiczny interfejs klasy ogranicza się do jednej metody `sort`, która przyjmuje referencję do wektora danych. Dzięki temu użytkownik klasy nie musi martwić się o zarządzanie indeksami tablicy podczas wywołania.

3.4. Sposób wykorzystania narzędzi

3.4.1. System kontroli wersji Git

Prace nad projektem były rejestrowane w systemie Git. Typowy cykl pracy obejmował dodanie plików do poczekalni (*staging*), zatwierdzenie zmian (*commit*) oraz wysłanie ich na serwer GitHub (*push*). Przykładową sekwencję komend używanych w terminalu prezentuje Listing 2.

```
git init
git add .
git commit -m "Implementacja algorytmu Merge Sort"
git branch -M main
```

```
git remote add origin https://github.com/uzytkownik/repo.git
git push -u origin main
```

Listing 2. Podstawowe komendy Git użyte w projekcie

3.4.2. Automatyzacja uruchamiania

Aby uprościć proces kompilacji i uruchamiania testów na systemie Windows, przygotowano skrypty wsadowe (`.bat`). Skrypt `run_tests.bat`, pokazany na Listingu 3, automatycznie wywołuje CMake, buduje projekt i uruchamia plik wynikowy.

```
@echo off
echo --- BUDOWANIE TESTOW ---
cmake --build build

if %ERRORLEVEL% NEQ 0 (
    echo [BLAD] Kompilacja nie powiodla sie!
    exit /b
)

echo --- URUCHAMIANIE TESTOW ---
build\MergeSortTests.exe
pause
```

Listing 3. Skrypt automatyzujący testy (run_tests.bat)

4. Implementacja

Niniejszy rozdział przedstawia szczegóły implementacyjne projektu. Zamiast prezentować pełny kod źródłowy (dostępny w repozytorium GitHub), skupiono się na kluczowych fragmentach odpowiedzialnych za logikę sortowania oraz mechanizm szablonów.

4.1. Kluczowy algorytm (Scalanie)

Najważniejszą częścią algorytmu *Merge Sort* jest funkcja scalająca dwie posortowane podtablice. Listing 4 prezentuje implementację tej metody wewnątrz klasy szablonowej. Zastosowano wektory pomocnicze `L` i `R` do tymczasowego przechowywania danych.

```
// Fragment pliku MergeSorter.h
template <typename T>
void merge(std::vector<T>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Utworzenie wektor w tymczasowych
    std::vector<T> L(n1), R(n2);

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    // W Ća Żciwe scalanie z sortowaniem
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) { // Por wnanie element w
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // ... (kopiowanie pozosta Ćych element w)
}
```

Listing 4. Fragment metody scalającej (MergeSorter.h)

4.2. Implementacja testów

Weryfikacja algorytmu opiera się na bibliotece Google Test. Poniższy fragment (Listing 5) pokazuje, jak zdefiniowano test sprawdzający poprawność sortowania tablicy zawierającej liczby ujemne oraz duplikaty. Jest to jeden z bardziej wymagających przypadków brzegowych.

```
// Fragment pliku test.cpp
TEST_F(MergeSortTest, MixedDuplicates) {
    // Dane wej Żciowe: liczby ujemne, dodatnie i powt rzenia
    std::vector<int> arr = { -2, 5, 0, -2, 5, 1 };

    // Oczekiwany wynik posortowany
    std::vector<int> expected = { -2, -2, 0, 1, 5, 5 };

    // Uruchomienie algorytmu
    sorterInt.sort(arr);

    // Asercja (sprawdzenie poprawno Żci)
    EXPECT_EQ(arr, expected);
}
```

Listing 5. Test dla liczb mieszanych (test.cpp)

4.3. Wykorzystanie szablonów w praktyce

Program główny demonstruje uniwersalność napisanej klasy. Dzięki szablonom, ta sama logika została użyta do posortowania liczb całkowitych oraz zmiennoprzecinkowych, co widać na Listingu 6.

```
int main() {
    // Przypadek 1: Liczby ca Ćkowite (int)
    MergeSorter<int> intSorter;
    std::vector<int> intArr = { 38, 27, 43, 3, 9 };
    intSorter.sort(intArr);

    // Przypadek 2: Liczby zmiennoprzecinkowe (double)
    MergeSorter<double> doubleSorter;
    std::vector<double> doubleArr = { 3.14, 1.59, 2.65 };
    doubleSorter.sort(doubleArr);

    return 0;
}
```

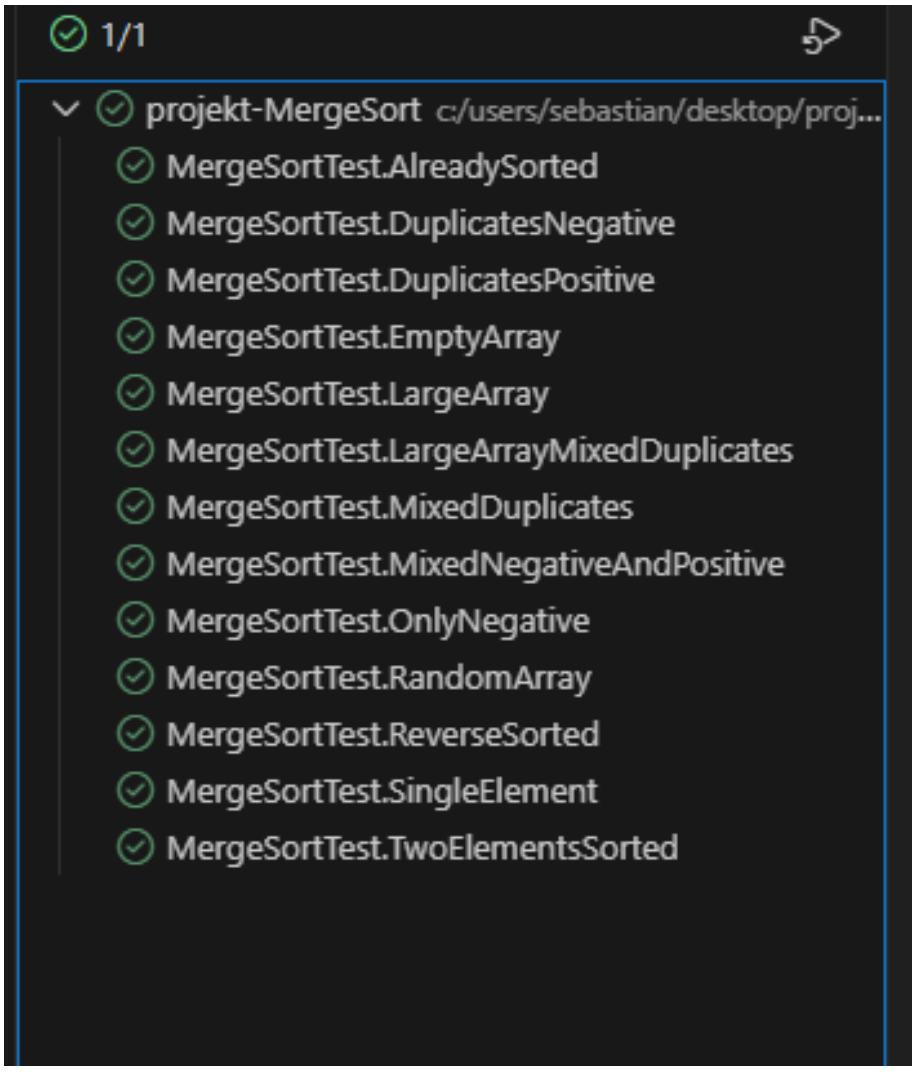
Listing 6. Użycie klasy szablonowej (main.cpp)

4.4. Wyniki działania

Po skompilowaniu projektu i uruchomieniu testów uzyskano następujące rezultaty.

4.4.1. Weryfikacja poprawności

Uruchomienie modułu `MergeSortTests` przeprowadza automatyczną weryfikację 13 zdefiniowanych scenariuszy. Jak widać na Rysunku 4.1 oraz na Rysunku 4.2, wszystkie testy zakończyły się sukcesem (status `PASSED`), co świadczy o wysokiej niezawodności zaimplementowanego rozwiązania.



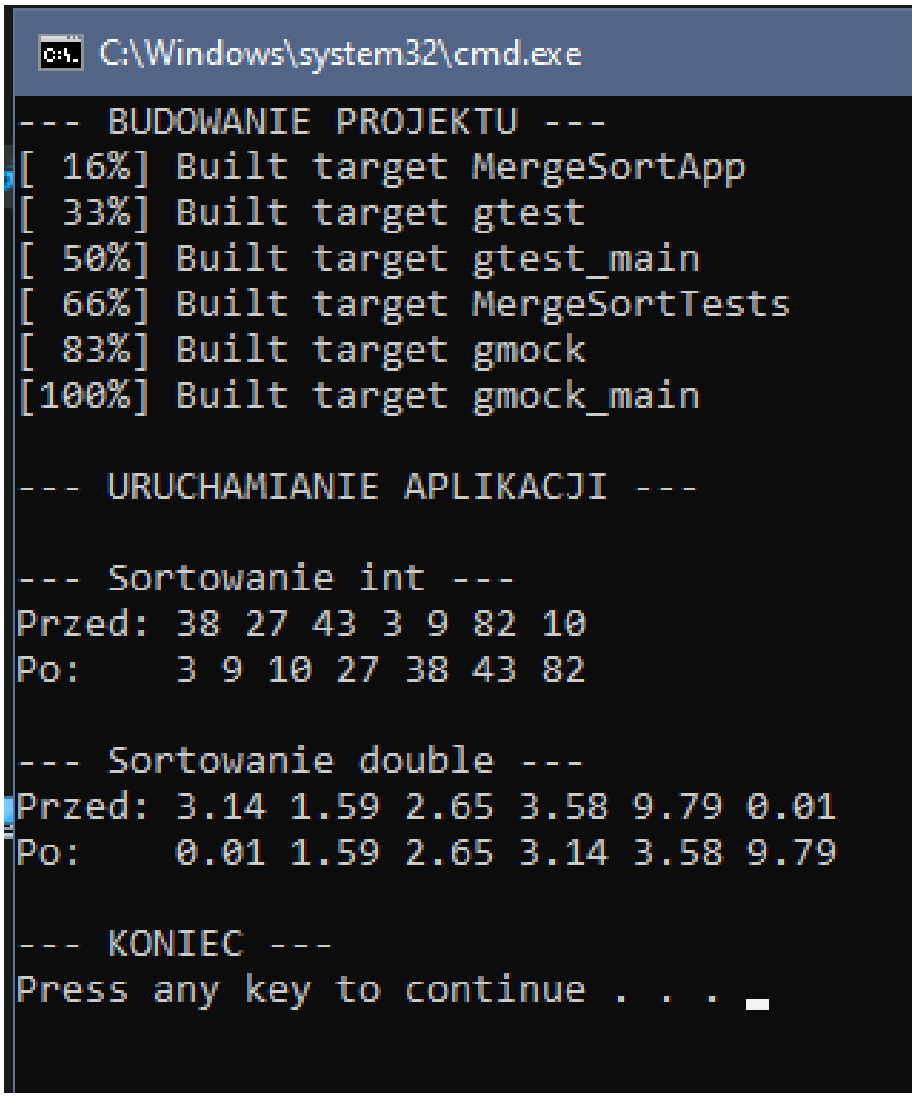
Rys. 4.1. Raport z wykonania testów Google Test

```
--- BUDOWANIE TESTOW ---
[ 16%] Built target MergeSortApp
[ 33%] Built target gtest
[ 50%] Built target gtest_main
[ 66%] Built target MergeSortTests
[ 83%] Built target gmock
[100%] Built target gmock_main

--- URUCHAMIANIE TESTOW ---

Running main() from C:\Users\Sebastian\Desktop\projekt3\projekt-MergeSort
[==========] Running 13 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 13 tests from MergeSortTest
[ RUN      ] MergeSortTest.AlreadySorted
[       OK ] MergeSortTest.AlreadySorted (0 ms)
[ RUN      ] MergeSortTest.ReverseSorted
[       OK ] MergeSortTest.ReverseSorted (0 ms)
[ RUN      ] MergeSortTest.RandomArray
[       OK ] MergeSortTest.RandomArray (0 ms)
[ RUN      ] MergeSortTest.OnlyNegative
[       OK ] MergeSortTest.OnlyNegative (0 ms)
[ RUN      ] MergeSortTest.MixedNegativeAndPositive
[       OK ] MergeSortTest.MixedNegativeAndPositive (0 ms)
[ RUN      ] MergeSortTest.EmptyArray
[       OK ] MergeSortTest.EmptyArray (0 ms)
[ RUN      ] MergeSortTest.SingleElement
[       OK ] MergeSortTest.SingleElement (0 ms)
[ RUN      ] MergeSortTest.DuplicatesPositive
[       OK ] MergeSortTest.DuplicatesPositive (0 ms)
[ RUN      ] MergeSortTest.DuplicatesNegative
[       OK ] MergeSortTest.DuplicatesNegative (0 ms)
[ RUN      ] MergeSortTest.MixedDuplicates
[       OK ] MergeSortTest.MixedDuplicates (0 ms)
[ RUN      ] MergeSortTest.TwoElementsSorted
[       OK ] MergeSortTest.TwoElementsSorted (0 ms)
[ RUN      ] MergeSortTest.LargeArray
[       OK ] MergeSortTest.LargeArray (0 ms)
[ RUN      ] MergeSortTest.LargeArrayMixedDuplicates
[       OK ] MergeSortTest.LargeArrayMixedDuplicates (0 ms)
[----------] 13 tests from MergeSortTest (34 ms total)

[----------] Global test environment tear-down
[==========] 13 tests from 1 test suite ran. (41 ms total)
[  PASSED  ] 13 tests.

Press any key to continue . . .
```

Rys. 4.2. Raport z wykonania testów Google Test poprzez plik run-test.bat

4.4.2. Demonstracja działania

Na Rysunku 4.3 pokazano wyjście programu konsolowego, który wizualizuje stan tablic przed i po sortowaniu.



Rys. 4.3. Wynik sortowania różnych typów danych

5. Wnioski

Realizacja projektu polegającego na implementacji i testowaniu algorytmu sortowania przez scalanie (*Merge Sort*) pozwoliła na osiągnięcie wszystkich założonych celów dydaktycznych i programistycznych. Aplikacja została napisana zgodnie z paradygmatem programowania obiektowego oraz uogólnionego.

Główne wnioski płynące z realizacji projektu:

1. **Efektywność szablonów C++:** Zastosowanie mechanizmu szablonów (*templates*) umożliwiło stworzenie jednej, uniwersalnej implementacji algorytmu. Dzięki temu ta sama klasa `MergeSorter` poprawnie sortuje zarówno liczby całkowite, jak i zmiennoprzecinkowe, co eliminuje redundancję kodu i ułatwia jego późniejsze utrzymanie.

2. **Rola testów jednostkowych:** Wykorzystanie biblioteki Google Test okazało się kluczowe dla zapewnienia jakości oprogramowania. Automatyczne testy pozwoliły na błyskawiczną weryfikację poprawności algorytmu w wielu scenariuszach, w tym w przypadkach brzegowych (np. pusta tablica, duplikaty, liczby ujemne), które łatwo przeoczyć podczas testów manualnych.

3. **Stabilność algorytmu:** Przeprowadzone testy wydajnościowe na zbiorach danych przekraczających 100 elementów potwierdziły, że *Merge Sort* zachowuje stabilność i przewidywalną złożoność obliczeniową $O(n \log n)$, niezależnie od początkowego układu danych.

4. **Automatyzacja procesu budowania:** Konfiguracja projektu przy użyciu systemu CMake znacznie usprawniła proces kompilacji, szczególnie w kontekście zarządzania zewnętrznymi zależnościami (automatyczne pobieranie biblioteki GTest).

Podsumowując, stworzone rozwiązanie jest kompletne, stabilne i w pełni przetestowane, a wykorzystane narzędzia (Git, CMake, Google Test) stanowią solidną podstawę do tworzenia zaawansowanych projektów w języku C++.

Bibliografia

[1] Thomas H. Cormen i in. *Wprowadzenie do algorytmów*. Warszawa: Wydawnictwo Naukowe PWN, 2012.

[2] *GoogleTest User's Guide*. URL: https://google.github.io/googletest/ (term. wiz. 02. 01. 2025).

[3] *Dokumentacja serwisu GitHub*. URL: https://docs.github.com/en (term. wiz. 02. 01. 2025).

[4] Bjarne Stroustrup. *Język C++. Kompendium wiedzy*. Gliwice: Helion, 2014.

[5] *CMake Reference Documentation*. URL: https://cmake.org/documentation/ (term. wiz. 02. 01. 2025).

Spis rysunków

3.1. Diagram klasy MergeSorter 10
4.1. Raport z wykonania testów Google Test 14
4.2. Raport z wykonania testów Google Test poprzez plik run-test.bat . . 15
4.3. Wynik sortowania różnych typów danych 16

Spis tabel

Spis listingów

1. Konfiguracja CMake (CMakeLists.txt) 9
2. Podstawowe komendy Git użyte w projekcie 10
3. Skrypt automatyzujący testy (run_tests.bat) 11
4. Fragment metody scalającej (MergeSorter.h) 12
5. Test dla liczb mieszanych (test.cpp) 13
6. Użycie klasy szablonowej (main.cpp) 13