AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Implementacja algorytmu Sortowania przez Scalanie z testami Google Test

Autor:
Dawid Michura

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Celem niniejszego projektu jest implementacja efektywnego algorytmu sortowania przez scalanie (Merge Sort)[1] zrealizowana w języku C++. Projekt kładzie szczególny nacisk na uniwersalność kodu poprzez zastosowanie szablonów (templates) [2], co pozwala na sortowanie danych różnych typów (np. `int`, `double`). Kluczowym elementem projektu jest weryfikacja poprawności algorytmu za pomocą zaawansowanych testów jednostkowych z wykorzystaniem frameworka Google Test[3], zarządzanych przez system budowania CMake.

Projekt ma umożliwić:

- zrozumienie rekurencyjnej natury algorytmu "Dziel i Zwyciężaj"[4],
- praktyczne zastosowanie programowania uogólnionego (szablony C++),
- konfigurację projektu C++ przy użyciu nowoczesnego systemu CMake,
- zdobycie doświadczenia w pisaniu testów jednostkowych pokrywających przypadki brzegowe.

Projekt kończy się implementacją działającego programu, przeprowadzenia testów oraz przygotowaniem szczegółowej dokumentacji technicznej.

1.1. Cel i zakres pracy

Celem pracy jest opracowanie klasy `MergeSorter`, która umożliwia:

1. sortowanie wektorów liczb całkowitych i zmiennoprzecinkowych,
2. obsługę skrajnych przypadków (puste tablice, duplikaty, odwrócona kolejność),
3. automatyczną weryfikację poprawności działania poprzez zestaw 13 testów jednostkowych,
4. łatwą kompilację na systemach Linux i Windows.

Ponadto projekt przewiduje modularyzację struktury programu poprzez separację plików nagłówkowych i źródłowych, a warunkiem zatwierdzenia kodu jest pomyślne przejście testów jednostkowych.

[1] Szczegółowy opis algorytmu Merge Sort [1].
[2] Więcej o templates w języku C++ [2].
[3] Dokumentacja GoogleTest [3].
[4] Więcej o tym algortymie [4]

1.2. Uzasadnienie wyboru problemu

Algorytmy sortowania są fundamentalnym zagadnieniem informatyki. Merge Sort, ze swoją złożonością czasową $O(n \log n)$, jest doskonałym przykładem algorytmu stabilnego i przewidywalnego. Implementacja tego algorytmu w połączeniu z Google Test pozwala na:

- naukę pisania testowalnego kodu (Clean Code),
- zrozumienie cyklu życia oprogramowania: Implementacja → Test → Refaktoryzacja,
- poznanie narzędzi używanych w komercyjnych projektach (CMake, CI/CD pipelines).

2. Analiza problemu

W niniejszym rozdziale przedstawiono teoretyczne podstawy algorytmu Sortowania przez Scalanie.

2.1. Zastosowanie algorytmu

Merge Sort to algorytm typu "Dziel i Rządź"(Divide and Conquer). Jego działanie opiera się na trzech krokach:

1. **Podział (Divide):** Tablica wejściowa dzielona jest na dwie równe (lub prawie równe) połowy.

2. **Zwyciężanie (Conquer):** Obie połowy są sortowane rekurencyjnie. Rekurencja kończy się, gdy podtablica ma rozmiar 0 lub 1 (jest wtedy uznana za posortowaną).

3. **Scalanie (Merge):** Dwie posortowane podtablice są łączone w jedną, zachowując porządek rosnący.

2.2. Złożoność obliczeniowa

Merge Sort gwarantuje złożoność czasową rzędu $O(n \log n)$ w każdym przypadku (optymistycznym, średnim i pesymistycznym).

- **Czas:** $T(n) = 2T(n/2) + O(n)$. Dzielenie tablicy generuje drzewo wywołań o wysokości $\log n$, a na każdym poziomie wykonujemy scalanie o koszcie liniowym n.

- **Pamięć:** Algorytm wymaga dodatkowej pamięci $O(n)$ na tymczasowe tablice pomocnicze wykorzystywane podczas scalania.

2.3. Opis działania algorytmu

Logika działania algorytmu Sortowania przez Scalanie (Merge Sort) opiera się na rekurencyjnym podziale zbioru danych, a następnie scalaniu posortowanych fragmentów. Algorytm nie operuje na strukturze danych w miejscu (in-place), lecz wykorzystuje pomocniczy obszar pamięci do konstruowania posortowanego wyniku. Proces ten można podzielić na trzy logiczne etapy:

1. **Etap Podziału (Divide):** Na tym etapie algorytm wyznacza punkt środkowy aktualnie rozpatrywanego zbioru danych. Ciąg wejściowy zostaje logicznie podzielony na dwie równe (lub różniące się o jeden element) podtablice: lewą i prawą. Operacja ta nie wiąże się jeszcze z porównywaniem wartości kluczy sortowania, a jedynie z manipulacją indeksami.

2. **Etap Rekurencji (Conquer):** Dla obu powstałych podtablic algorytm wywoływany jest ponownie. Proces podziału postępuje kaskadowo w dół drzewa rekurencji, aż do osiągnięcia przypadku trywialnego, w którym podtablica zawiera tylko jeden element. Zbiór jednoelementowy z definicji uznaje się za posortowany, co stanowi warunek stopu rekurencji.

3. **Etap Scalania (Merge):** Jest to kluczowa faza, w której następuje właściwe sortowanie. Dwie posortowane podtablice (zwrócone przez wywołania rekurencyjne) są łączone w jeden uporządkowany ciąg. Algorytm wykorzystuje dwa wskaźniki, które śledzą aktualne pozycje w obu podtablicach. W pętli porównywane są elementy wskazywane przez wskaźniki:

   - Element mniejszy (lub równy, dla zachowania stabilności) jest przenoszony do tablicy wynikowej.
   - Wskaźnik odpowiedniej podtablicy jest przesuwany o jedną pozycję.

   Proces trwa do momentu wyczerpania elementów w jednej z podtablic. Elementy pozostałe w drugiej podtablicy są wówczas przepisywane na koniec ciągu wynikowego.

Schemat ten gwarantuje, że praca wykonana podczas scalania na niższych poziomach rekurencji jest wykorzystywana na poziomach wyższych, aż do uzyskania całkowicie posortowanego zbioru danych na poziomie korzenia wywołań.

2.4. Ręczna analiza przykładu

Aby precyzyjnie zobrazować działanie algorytmu, przeanalizujmy ręcznie proces sortowania dla zestawu danych o parzystej liczbie elementów: $\mathbf{A} = [\mathbf{50}, \mathbf{30}, \mathbf{70}, \mathbf{20}, \mathbf{40}, \mathbf{60}]$.

2.4.1. Etap 1: Podział (Divide)

Algorytm dzieli tablicę (zakres indeksów 0-5, środek = 2):

1. Tablica $\mathbf{A}$ dzieli się na część lewą $\mathbf{L} = [\mathbf{50}, \mathbf{30}, \mathbf{70}]$ oraz prawą $\mathbf{R} = [\mathbf{20}, \mathbf{40}, \mathbf{60}]$.

2. Lewa część **L** dzieli się dalej na [**50**, **30**] oraz [**70**].
3. [**50**, **30**] dzieli się ostatecznie na elementy atomowe [**50**] i [**30**].
4. Prawa część **R** dzieli się analogicznie na [**20**, **40**] oraz [**60**].
5. [**20**, **40**] dzieli się ostatecznie na [**20**] i [**40**].

Stan po całkowitym podziale na elementy jednoelementowe (uznawane za posortowane) to: [**50**], [**30**], [**70**], [**20**], [**40**], [**60**]. Sytuację tę obrazuje schematycznie Rysunek 2.1.

2.4.2. Etap 2: Scalanie wstępne (Merge)

Następuje proces scalania mniejszych podtablic w uporządkowane sekwencje (powrót z rekurencji):

Lewa gałąź:

1. Scalanie [**50**] i [**30**]: Porównujemy **50** > **30**. Zamiana. Wynik: [**30**, **50**].
2. Dołączenie [**70**]: Scalamy [**30**, **50**] z [**70**]. Element 70 jest największy, trafia na koniec. Lewa połówka gotowa: $\mathbf{L_{sorted}} = [\mathbf{30}, \mathbf{50}, \mathbf{70}]$.

Wynik pośredni (lewa oraz prawa posortowana gałąź gotowe do finalnego złączenia) przedstawia Rysunek 2.2.

Prawa gałąź:

1. Scalanie [**20**] i [**40**]: Porównujemy **20** < **40**. Bez zmian. Wynik: [**20**, **40**].
2. Dołączenie [**60**]: Scalamy [**20**, **40**] z [**60**]. Element 60 trafia na koniec. Prawa połówka gotowa: $\mathbf{R_{sorted}} = [\mathbf{20}, \mathbf{40}, \mathbf{60}]$.

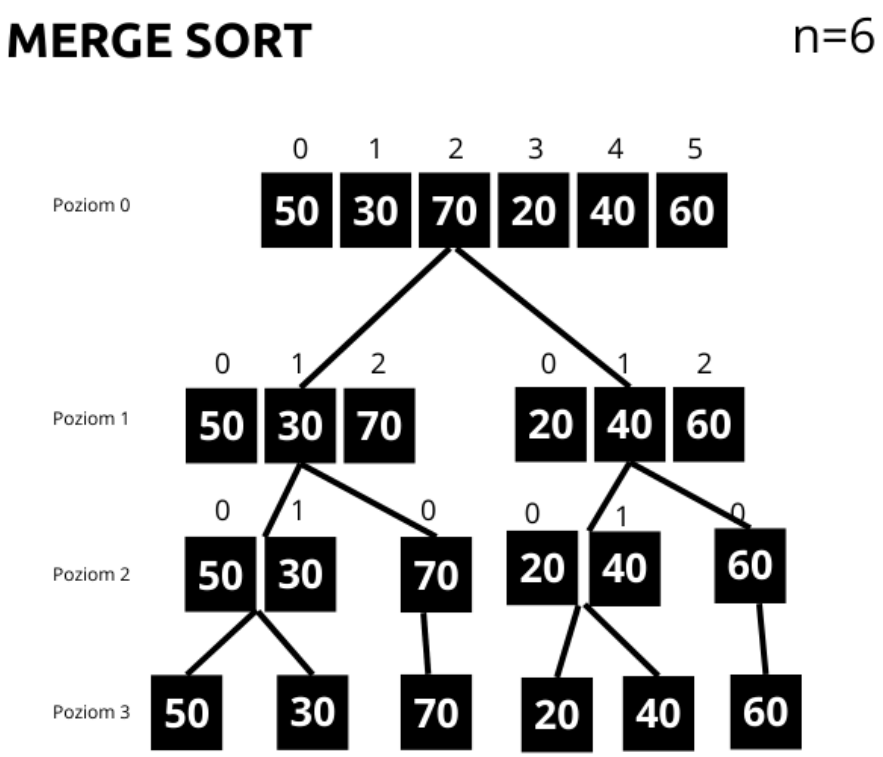
2.4.3. Etap 3: Scalanie końcowe

Łączymy dwie posortowane tablice: lewą $\mathbf{L} = [\mathbf{30}, \mathbf{50}, \mathbf{70}]$ oraz prawą $\mathbf{R} = [\mathbf{20}, \mathbf{40}, \mathbf{60}]$:

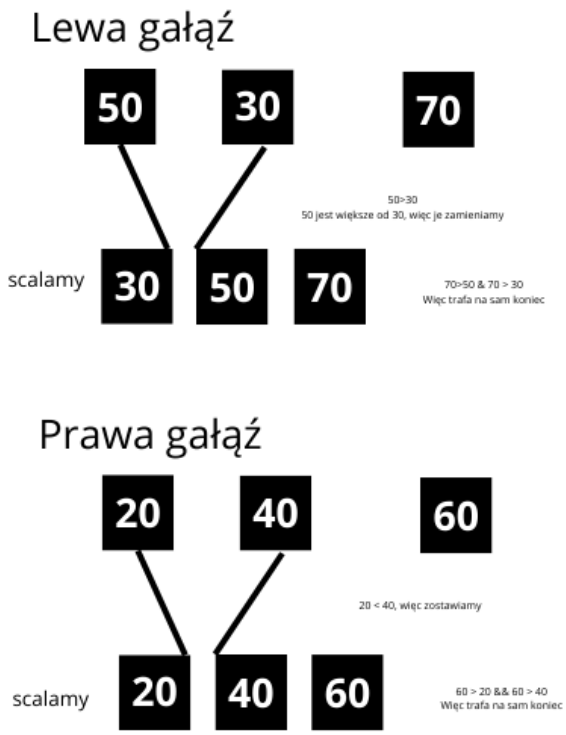
- Porównaj **30** (L) i **20** (R). **20** < **30**. Wstaw 20. Wynik: [**20**].
- Porównaj **30** (L) i **40** (R). **30** < **40**. Wstaw 30. Wynik: [**20**, **30**].
- Porównaj **50** (L) i **40** (R). **40** < **50**. Wstaw 40. Wynik: [**20**, **30**, **40**].
- Porównaj **50** (L) i **60** (R). **50** < **60**. Wstaw 50. Wynik: [**20**, **30**, **40**, **50**].

- Porównaj **70** (L) i **60** (R). **60** < **70**. Wstaw 60. Wynik: [**20**, **30**, **40**, **50**, **60**].
- Prawa tablica **R** wyczerpana. Kopiujemy pozostały element z **L** (70).
- Dopisujemy resztę: 70.

Wynik końcowy: [**20**, **30**, **40**, **50**, **60**, **70**].



Rys. 2.1. Etap 1



Rys. 2.2. Wynik scalania podtablic (dwie posortowane połówki)

3. Projektowanie

Niniejszy rozdział poświęcono szczegółowej specyfikacji technicznej projektowanego rozwiązania. Omówiono dobrane środowisko wytwórcze, konfigurację narzędzi kompilacji, architekturę algorytmu oraz kluczowe decyzje implementacyjne.

3.1. Środowisko wytwórcze i stos technologiczny

Fundamentem realizacji projektu był dobór nowoczesnego i wydajnego stosu technologicznego, zapewniającego przenośność kodu oraz możliwość jego rygorystycznego testowania. Proces deweloperski oparto na następujących komponentach:

- **Język implementacji:** C++ (Standard C++17). Decyzja ta podyktowana była potrzebą wykorzystania zaawansowanych mechanizmów programowania uogólnionego (szablony) oraz narzędzi do bezpiecznego zarządzania zasobami pamięci.

- **Toolchain kompilacji:** Zestaw GCC/g++ (GNU Compiler Collection). Wybrano go ze względu na powszechność zastosowań w przemyśle i ścisłą zgodność z normami ISO C++.

- **Zarządzanie kodem:** System kontroli wersji Git wspierany przez platformę GitHub. Umożliwiło to pracę w modelu rozproszonym, zarządzanie gałęziami funkcjonalnymi oraz bezpieczną archiwizację historii zmian.

- **Weryfikacja jakości:** Framework Google Test (GTest), który posłużył do zautomatyzowania procesu testów jednostkowych.

Dodatkowo, do celów dokumentacyjnych wykorzystano system składu tekstu LaTeX oraz narzędzie Doxygen, służące do automatycznej generacji dokumentacji API na podstawie komentarzy w kodzie źródłowym.

3.2. Rygory kompilacji i konsolidacji

W celu zapewnienia najwyższej jakości kodu wynikowego oraz wczesnego wykrywania potencjalnych błędów, przyjęto rygorystyczną politykę kompilacji. Proces budowania aplikacji wykorzystuje zestaw flag wymuszających standardy i statyczną analizę kodu:

- `-std=c++17` – jawne wskazanie standardu języka, co umożliwia korzystanie z jego najnowszych funkcjonalności.

- `-Wall -Wextra` – aktywacja szerokiego spektrum ostrzeżeń kompilatora. Pozwala to na identyfikację problematycznych konstrukcji (np. nieużywanych zmiennych, niejawnych konwersji) już na etapie budowania.

- `-Iinclude` – definicja ścieżki do plików nagłówkowych, promująca czystą strukturę projektu poprzez separację interfejsów (`.h`) od implementacji (`.cpp`).

Kluczowym aspektem budowania modułu testowego jest etap konsolidacji (linkowania) z zewnętrznymi bibliotekami. Wymaga to przekazania linkerowi flag `-lgtest` oraz `-lgtest_main`, które dołączają niezbędne binaria frameworka Google Test. Dodatkowo, flaga `-pthread` jest konieczna do zapewnienia obsługi mechanizmów wielowątkowości, z których korzysta biblioteka testowa.

3.3. Architektura algorytmu i wizualizacja działania

Zaprojektowany system charakteryzuje się architekturą modułową, w której logika obliczeniowa (core) jest odseparowana od warstwy prezentacji i testów. Centralnym punktem implementacji jest szablon klasy `MergeSorter`. Poniżej przedstawiono graficzną analizę przepływu sterowania zaimplementowanego algorytmu sortowania przez scalanie.

Proces rozpoczyna się od fazy podziału (Divide). Algorytm wyznacza punkt centralny zbioru danych i dokonuje logicznej dywersyfikacji tablicy wejściowej na dwie podtablice o zbliżonym rozmiarze.

Następnie, zgodnie z paradygmatem rekurencji, operacja ta jest kaskadowo powtarzana dla każdej z powstałych części. Dekompozycja postępuje aż do momentu uzyskania zbiorów atomowych (jednoelementowych), które z definicji są posortowane.

Po osiągnięciu dna rekurencji następuje faza "Rządź"(Conquer). Sąsiadujące elementy atomowe są porównywane i scalane w większe, uporządkowane struktury. Proces łączenia komórek w posortowane pary i większe bloki

Ostatnim krokiem jest finalna konsolidacja dwóch głównych, posortowanych podtablic w jeden wynikowy wektor. Na tym etapie algorytm wykorzystuje pomocnicze bufory pamięci do zapewnienia stabilności i poprawności ułożenia wszystkich elementów, co kończy proces sortowania

3.4. Metodyka weryfikacji z użyciem Google Test

Zapewnienie poprawności implementacji algorytmu zrealizowano poprzez wdrożenie zaawansowanego frameworka Google Test. Pozwoliło to na zautomatyzowanie procesu kontroli jakości. Metodyka pracy z tym narzędziem w projekcie opiera się na trzech filarach:

1. **Definicja przypadków testowych:** Scenariusze testowe tworzone są przy użyciu makra `TEST(NazwaZestawu, NazwaTestu)`. Pozwala to na logiczne grupowanie testów i ich automatyczną rejestrację w runnerze.

2. **Mechanizm asercji:** Weryfikacja wyników odbywa się za pomocą makr asercji, takich jak `ASSERT_EQ(oczekiwany, aktualny)` (twarde sprawdzenie równości) czy `EXPECT_NO_THROW(...)` (weryfikacja braku wyjątków). Niespełnienie warunku asercji skutkuje natychmiastowym raportem błędu.

3. **Automatyzacja uruchamiania:** Główna funkcja programu testowego inicjalizuje środowisko (`::testing::InitGoogleTest`) i uruchamia wszystkie zarejestrowane scenariusze jednym poleceniem `RUN_ALL_TESTS()`.

Przyjęte podejście umożliwia błyskawiczną weryfikację regresji dla 13 zdefiniowanych przypadków brzegowych po każdej modyfikacji kodu źródłowego.

3.5. Kluczowe decyzje projektowe i implementacyjne

Na etapie analizy technicznej podjęto szereg decyzji mających kluczowy wpływ na kształt i właściwości ostatecznego rozwiązania:

- **Abstrakcja pamięci (std::vector):** Zdecydowano się na użycie kontenera `std::vector` zamiast surowych tablic dynamicznych. Zapewnia to wyższy poziom bezpieczeństwa pamięci (RAII) oraz elastyczność w zarządzaniu rozmiarem danych.
- **Programowanie uogólnione (Templates):** Implementację algorytmu zrealizowano w formie szablonu w pliku nagłówkowym (`.hpp`). Umożliwia to kompilatorowi generowanie dedykowanego kodu dla dowolnego typu danych, który posiada zdefiniowany operator porównania, zwiększając reużywalność kodu.
- **Bezpieczeństwo arytmetyczne:** Przy obliczaniu indeksu środkowego zastosowano bezpieczną formułę `left + (right - left) / 2`. Chroni ona przed potencjalnym przepełnieniem typu całkowitego (integer overflow), które mogłoby wystąpić przy sumowaniu dużych indeksów w klasycznym podejściu `(left + right) / 2`.

4. Implementacja

W niniejszym rozdziale przedstawiono szczegóły implementacyjne wytworzonego oprogramowania. Opisano strukturę kodu źródłowego, kluczowe fragmenty logiki algorytmicznej, implementację mechanizmów testowych oraz konfigurację procesu automatycznego budowania.

4.1. Struktura kodu źródłowego

Kod projektu został zorganizowany w sposób zapewniający separację logiki biznesowej (algorytmu) od warstwy prezentacji oraz warstwy testowej. Główna implementacja algorytmu Sortowania przez Scalanie została zamknięta w pliku nagłówkowym `src/MergeSorter.h`. Decyzja o umieszczeniu definicji metod bezpośrednio w pliku nagłówkowym wynika ze specyfiki kompilacji szablonów w języku C++. Kompilator musi mieć dostęp do pełnej definicji szablonu w każdym pliku tłumaczenia, który go konkretyzuje.

4.2. Implementacja klasy generycznej MergeSorter

Rdzeniem systemu jest szablon klasy `MergeSorter`. Zastosowanie słowa kluczowego `template <typename T>` umożliwia parametryzację typu danych. Klasa udostępnia publiczny interfejs (metodę `sort`), który ukrywa przed użytkownikiem szczegóły rekurencyjnego wywoływania funkcji wewnętrznych (Listing 1).

```
template <typename T>
class MergeSorter {
    private:
        // Metoda scalajaca dwa podzakresy
        void merge(vector<T>& array, int left, int mid, int right);

        // Metoda rekurencyjna dzielaca tablice
        void mergeSortRecursive(vector<T>& array, int left, int
    right);

    public:
        // Publiczny interfejs
        void sort(vector<T>& array) {
            if (array.size() <= 1) {
                return; // Warunek brzegowy: tablica pusta lub 1-
    elementowa
            }
            mergeSortRecursive(array, 0, array.size() - 1);
```

```
        }
};
```

Listing 1. Definicja szablonu klasy MergeSorter (‘src/MergeSorter.h‘).

4.2.1. Bezpieczna dekompozycja tablicy

W metodzie `mergeSortRecursive` zaimplementowano mechanizm podziału tablicy. Kluczowym detalem implementacyjnym jest sposób obliczania indeksu środkowego (`mid`). Zamiast klasycznej średniej arytmetycznej, zastosowano przesunięcie względem lewej granicy (Listing 2). Takie podejście eliminuje ryzyko wystąpienia błędu przepełnienia zmiennej całkowitej (*integer overflow*) w przypadku sortowania tablic o rozmiarach zbliżonych do maksymalnego zakresu typu `int`.

```
void mergeSortRecursive(vector<T>& array, int left, int right) {
    if (left < right) {
        // Bezpieczne obliczanie srodka (chroni przed overflow)
        int mid = left + (right - left) / 2;

        mergeSortRecursive(array, left, mid);
        mergeSortRecursive(array, mid + 1, right);

        merge(array, left, mid, right);
    }
}
```

Listing 2. Implementacja bezpiecznego obliczania środka zakresu.

4.2.2. Logika scalania (Merge)

Metoda `merge` realizuje właściwe sortowanie. W pierwszym kroku alokuje ona dwa tymczasowe wektory pomocnicze L i R, do których kopiowane są dane z odpowiednich połówek oryginalnej tablicy. Następnie w pętli `while` następuje porównywanie elementów i wpisywanie mniejszego (lub równego) z powrotem do tablicy głównej. Złożoność pamięciowa tej operacji wynosi $O(n)$ ze względu na konieczność utworzenia kopii danych.

4.3. Warstwa demonstracyjna

Plik `src/main.cpp` pełni rolę aplikacji klienckiej, demonstrującej poprawność działania biblioteki. Wykorzystano w nim konkretyzację szablonu dla dwóch typów

danych: liczb całkowitych (`int`) oraz zmiennoprzecinkowych (`double`). Pozwala to na empiryczne potwierdzenie uniwersalności napisanego kodu (Listing 3).

```
// Instancja dla typu double
vector<double> doubleTab = { 3.14, -1.1, 0.0, 2.5, 3.14 };
MergeSorter<double> doubleSorter;

cout << "NO SORT | ";
printArray(doubleTab);

doubleSorter.sort(doubleTab);

cout << "SORTED  | ";
printArray(doubleTab);
```

Listing 3. Fragment funkcji main demonstrujący sortowanie typów zmiennoprzecinkowych.

4.4. Implementacja testów automatycznych

Weryfikacja poprawności algorytmu została zrealizowana w pliku `tests/test_mergesort.cpp` przy użyciu makr biblioteki Google Test. Zaimplementowano 13 scenariuszy testowych, pokrywających różne klasy równoważności danych wejściowych.

Przykładem testu weryfikującego stabilność i poprawność sortowania danych losowych jest test `RandomArray` (Listing 4), który wykorzystuje algorytm weryfikacyjny `std::is_sorted` z biblioteki standardowej. Z kolei testy przypadków brzegowych, takich jak pusta tablica, sprawdzają odporność kodu na błędy wykonania (Listing 5).

```
TEST(MergeSortTest, RandomArray) {
    vector<int> arr = {10, 2, 8, 1, 5, 9};
    MergeSorter<int> sorter;
    sorter.sort(arr);
    // Asercja sprawdzajaca, czy tablica jest posortowana rosnaco
    EXPECT_TRUE(is_sorted(arr.begin(), arr.end()));
}
```

Listing 4. Implementacja testu dla danych losowych.

```
TEST(MergeSortTest, EmptyArray) {
    vector<int> arr = {};
    MergeSorter<int> sorter;
    // Oczekujemy, ze metoda nie rzuci wyjatkiem
    EXPECT_NO_THROW(sorter.sort(arr));
```

```
6     EXPECT_TRUE(arr.empty());
7 }
```

Listing 5. Test odporności na puste dane wejściowe.

4.5. Konfiguracja systemu budowania CMake

Proces kompilacji i linkowania został w pełni zautomatyzowany za pomocą skryptu `CMakeLists.txt`. Kluczowym elementem konfiguracji jest użycie modułu `FetchContent`, który dynamicznie pobiera kod źródłowy Google Test z repozytorium GitHub w momencie generowania plików budowania (Listing 6).

```
1 include(FetchContent)
2 FetchContent_Declare(
3   googletest
4   URL https://github.com/google/googletest/archive/refs/heads/main.
     zip
5 )
6 set(INSTALL_GTEST OFF)
7 FetchContent_MakeAvailable(googletest)
8
9 enable_testing()
10
11 add_executable(runTests tests/test_mergesort.cpp)
12 target_link_libraries(runTests gtest_main)
13
14 include(GoogleTest)
15 gtest_discover_tests(runTests)
```

Listing 6. Konfiguracja automatycznego pobierania Google Test w CMake.

Dzięki takiej konfiguracji projekt jest samowystarczalny – nie wymaga od użytkownika ręcznej instalacji bibliotek w systemie operacyjnym, co znacząco ułatwia przenoszenie kodu między różnymi środowiskami deweloperskimi.

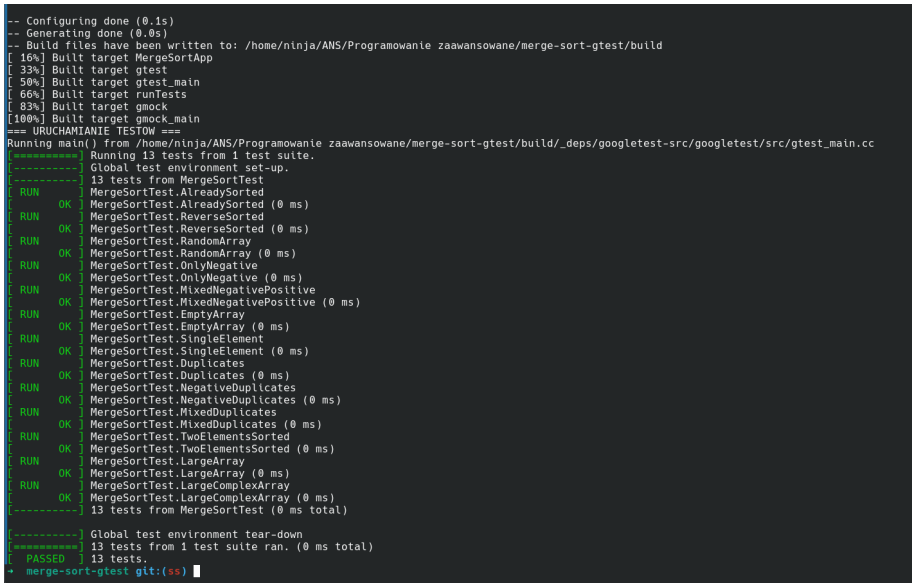
4.6. Analiza wyników

Proces weryfikacji przygotowanego oprogramowania zakończył się całkowitym powodzeniem. Uruchomienie pełnego zestawu testów jednostkowych potwierdziło poprawne działanie algorytmu we wszystkich zdefiniowanych przypadkach granicznych. Zestawienie najważniejszych rezultatów przedstawiono w Tabeli 4.1.

Tab. 4.1. Zestawienie wyników testów jednostkowych

Lp.	Scenariusz testowy	Status
1	Sortowanie tablicy uporządkowanej	PASSED
2	Sortowanie tablicy malejącej	PASSED
3	Sortowanie liczb losowych	PASSED
4	Obsługa wartości ujemnych	PASSED
5	Zbiór mieszany (wartości dodatnie i ujemne)	PASSED
6	Pusta tablica	PASSED
7	Tablica jednoelementowa	PASSED
8–10	Obsługa duplikatów	PASSED
11	Tablica dwuelementowa	PASSED
12–13	Duże zbiory danych (>100 elementów)	PASSED

Wynik działania testów w środowisku konsolowym zaprezentowano na Rysunku4.1. Końcowy komunikat w kolorze zielonym potwierdza, że wszystkie 13 testów zakończyło się pozytywnie, co jednoznacznie wskazuje na poprawność implementacji.



Rys. 4.1. Wyniki testów

5. Podsumowanie i wnioski

Realizacja projektu polegającego na implementacji generycznego algorytmu Merge Sort w języku C++ stanowiła kompleksowe ćwiczenie, które pozwoliło na zintegrowanie wiedzy teoretycznej z zakresu algorytmiki z praktycznymi wymogami nowoczesnej inżynierii oprogramowania. Proces ten wykroczył daleko poza ramy prostego kodowania, stając się lekcją projektowania architektury, zarządzania jakością oraz automatyzacji procesów wytwórczych.

5.1. Analiza aspektów programistycznych

Z perspektywy czysto implementacyjnej, największym wyzwaniem i zarazem wartością edukacyjną było praktyczne zastosowanie paradygmatu programowania uogólnionego. Przekształcenie standardowej funkcji sortującej tablice liczb całkowitych w szablon klasy `MergeSorter<T>` wymusiło zmianę sposobu myślenia o kodzie. Konieczność operowania na abstrakcjach zamiast na konkretnych typach danych oraz specyficzne wymagania kompilatora C++ (implementacja szablonów w plikach nagłówkowych) pozwoliły mi lepiej zrozumieć mechanizmy działające "pod maską"biblioteki standardowej STL[5]. Dzięki temu stworzone rozwiązanie jest elastyczne i gotowe do użycia z dowolnym typem danych wspierającym operator porównania.

Istotnym wnioskiem technicznym jest również rola bezpiecznej arytmetyki w systemach informatycznych. Zastosowanie zmodyfikowanej formuły obliczania środka przedziału ($mid = left + (right - left)/2$) zamiast klasycznej średniej arytmetycznej, zabezpieczyło aplikację przed błędem przepełnienia typu całkowitego (*integer overflow*). Jest to detel implementacyjny, który odróżnia kod amatorski od kodu o jakości produkcyjnej.

5.2. Wnioski z procesu testowania (Google Test)

Fundamentalną zmianę w kulturze pracy wprowadziło wdrożenie frameworka Google Test. Odejście od manualnej weryfikacji wyników ("print debugging") na rzecz zestawu 13 automatycznych scenariuszy testowych diametralnie zmieniło poczucie bezpieczeństwa podczas wprowadzania zmian. Testy jednostkowe pełniły w projekcie podwójną rolę:

1. **Weryfikacyjną:** Potwierdzały poprawność matematyczną algorytmu dla przy-

[5]Kontenery STL [5]

padków brzegowych (tablice puste, jednoelementowe, duplikaty).

2. **Regresyjną:** Pozwalały na natychmiastowe wykrycie błędów powstałych w wyniku refaktoryzacji kodu.

Doświadczenie to utwierdziło mnie w przekonaniu, że w profesjonalnym projekcie informatycznym kod niepokryty testami należy traktować jako dług technologiczny.

5.3. Organizacja środowiska i narzędzia

Nieodłącznym elementem sukcesu projektu była odpowiednia konfiguracja środowiska wytwórczego. Zastosowanie systemu CMake. Automatyczne pobieranie zależności (Google Test) sprawiło, że projekt jest w pełni przenośny i gotowy do kompilacji na dowolnym systemie operacyjnym (Linux/Windows) bez konieczności ręcznej instalacji bibliotek.

Równie kluczowe było wykorzystanie systemu kontroli wersji Git oraz strategii pracy na gałęziach (*feature branching*). Separacja prac nad logiką algorytmu, testami oraz dokumentacją zapewniła przejrzystość historii zmian i pozwoliła na symulację profesjonalnego workflow, w którym każda funkcjonalność jest izolowana aż do momentu jej stabilizacji. Repozytorium stało się dzięki temu czytelnym dziennikiem rozwoju produktu, a nie tylko magazynem plików.

5.4. Znaczenie dokumentacji technicznej

Dopełnieniem projektu było wygenerowanie dokumentacji technicznej przy użyciu narzędzia Doxygen. Proces ten uświadomił mi, że czytelność kodu dla człowieka jest równie istotna, jak jego poprawność dla kompilatora. Utrzymywanie spójnych komentarzy zgodnych ze standardem Javadoc wymusiło dbałość o precyzyjne nazewnictwo metod i zmiennych, co bezpośrednio przełożyło się na jakość kodu źródłowego (*self-documenting code*).

5.5. Podsumowanie końcowe

Podsumowując, projekt implementacji algorytmu Merge Sort był wielowymiarową lekcją inżynierii oprogramowania. Udowodnił, że stworzenie wysokiej jakości aplikacji to wypadkowa trzech elementów: wydajnego algorytmu, rygorystycznej weryfikacji automatycznej oraz przemyślanej architektury wspieranej przez nowoczesne narzędzia budowania. Projekt ten pozwolił mi wyjść poza rolę programisty skupio-

nego wyłącznie na składni języka i spojrzeć na proces tworzenia oprogramowania w sposób holistyczny, typowy dla inżyniera systemów.

Bibliografia

[1] *Merge Sort - Wikipedia*. URL: https://en.wikipedia.org/wiki/Merge_sort (term. wiz. 29. 11. 2025).

[2] *Więcej o templates w C++*. URL: https://www.geeksforgeeks.org/cpp/templates-cpp/ (term. wiz. 29. 11. 2025).

[3] *GoogleTest User's Guide*. URL: https://google.github.io/googletest/ (term. wiz. 29. 11. 2025).

[4] *Więcej o algortymie "Dziel i Zwyciężaj"*. URL: https://pl.wikipedia.org/wiki/Dziel_i_zwyci%C4%99%C5%BCaj (term. wiz. 29. 11. 2025).

[5] *Kontenery STL*. URL: https://www.geeksforgeeks.org/cpp/the-c-standard-template-library-stl/ (term. wiz. 29. 11. 2025).

Spis rysunków

Spis tabel

Spis listingów