AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Algorytm sortowania przez scalanie (Merge Sort)

Autor:

Michał Marecik

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest implementacja oraz przetestowanie algorytmu sortowania przez scalanie (*Merge Sort*) w języku C++. Kluczowym aspektem zadania jest wykorzystanie paradygmatu programowania uogólnionego (szablonów), co pozwoli na sortowanie różnych typów danych liczbowych. Projekt kładzie duży nacisk na weryfikację poprawności kodu poprzez testy jednostkowe oraz na wykorzystanie nowoczesnych narzędzi inżynierskich (Git, Doxygen).

1.1. Cel i zakres projektu

Głównym zadaniem jest stworzenie klasy szablonowej `MergeSort`, która będzie udostępniać statyczną metodę sortującą. Aplikacja musi zostać podzielona na dwa moduły w ramach jednego rozwiązania w Visual Studio:

1. **Aplikacja główna:** Zawierająca funkcję `main` oraz demonstrację działania na tablicach typów `int` i `double`.
2. **Moduł testowy:** Zawierający zestaw testów jednostkowych opartych na frameworku Google Test.

1.2. Specyfikacja wymagań funkcjonalnych (Testy)

Zgodnie z poleceniem, algorytm musi zostać poddany rygorystycznym testom. Zaimplementowane testy jednostkowe muszą weryfikować następujące scenariusze:

- **Sortowanie podstawowe:** Poprawne sortowanie losowej tablicy liczb.
- **Optymalizacja:** Brak zmian w tablicy, gdy jest ona już posortowana rosnąco.
- **Odwrotna kolejność:** Poprawne sortowanie tablicy posortowanej malejąco.
- **Wartości ujemne:** Obsługa tablic z samymi liczbami ujemnymi oraz mieszanych (ujemne i dodatnie).
- **Duplikaty:** Poprawne sortowanie tablic zawierających powtarzające się wartości (dla liczb dodatnich, ujemnych i mieszanych).
- **Przypadki brzegowe:**
  - Obsługa pustej tablicy (brak wyjątków).
  - Obsługa tablicy jednoelementowej (brak zmian).

  - Obsługa tablicy dwuelementowej.

- **Wydajność/Skala:** Poprawne sortowanie dużych tablic (ponad 100 elementów), również w wariantach z liczbami ujemnymi i duplikatami.

1.3. Wymagania pozafunkcjonalne i narzędziowe

Projekt musi spełniać standardy inżynierskie w zakresie wytwarzania oprogramowania:

- **Środowisko:** Visual Studio (C++).
- **Szablony:** Kod musi być uniwersalny (obsługa `int`, `double`, `float`, `long`).
- **Kontrola wersji:** Kod musi być przechowywany w zdalnym repozytorium GitHub.
- **Dokumentacja:**
  - Dokumentacja techniczna wygenerowana automatycznie (Doxygen) do formatu PDF.
  - Dokumentacja projektowa (sprawozdanie) w systemie LaTeX.

2. Analiza problemu

Celem projektu jest implementacja algorytmu sortowania przez scalanie (*Merge Sort*) w języku C++. Jest to jeden z podstawowych algorytmów sortowania, działający w oparciu o metodę "dziel i zwyciężaj"(*divide and conquer*).

Algorytm ten charakteryzuje się złożonością czasową rzędu $O(n \log n)$, co czyni go znacznie wydajniejszym od prostych metod sortowania (takich jak sortowanie bąbelkowe) dla dużych zbiorów danych. Kluczowym aspektem projektu jest zastosowanie **szablonów** (*templates*), co pozwala na uniezależnienie implementacji od konkretnego typu danych liczbowych.

2.1. Zastosowanie algorytmu

Sortowanie przez scalanie jest szeroko wykorzystywane w informatyce ze względu na swoją stabilność oraz gwarantowaną złożoność obliczeniową. Przykładowe zastosowania to:

- **Sortowanie dużych zbiorów danych:** Dzięki wydajności $O(n \log n)$ algorytm świetnie radzi sobie z tablicami zawierającymi miliony elementów.
- **Sortowanie list jednokierunkowych:** W przeciwieństwie do Quick Sort, Merge Sort nie wymaga swobodnego dostępu do pamięci (random access), co czyni go idealnym dla struktur listowych.
- **Sortowanie zewnętrzne:** Jest wykorzystywany w systemach bazodanowych do sortowania danych, które nie mieszczą się w pamięci RAM (dane są dzielone na fragmenty, sortowane i scalane z dysku).
- **Zastosowania ogólne:** Biblioteki standardowe w wielu językach programowania używają wariantów Merge Sort jako domyślnego algorytmu sortowania stabilnego.

2.2. Opis działania programu

Zaimplementowane rozwiązanie składa się z dwóch głównych modułów w środowisku Visual Studio:

1. **Moduł Algorytmu (Aplikacja):**
    - Klasa `MergeSort` – statyczna klasa szablonowa zawierająca metody `sort` (publiczna) oraz `mergeSort` i `merge` (prywatne, pomocnicze).

- Funkcja `main` – demonstruje działanie algorytmu na wektorach typów `int` oraz `double`, wypisując wyniki na standardowe wyjście.

2. **Moduł Testowy (Google Test):**

   - Zawiera zestaw 13 testów jednostkowych weryfikujących poprawność algorytmu w skrajnych przypadkach (np. pusta tablica, liczby ujemne, duplikaty).

2.3. Przykład działania algorytmu

Aby zilustrować działanie metody "dziel i zwyciężaj", rozważmy sortowanie tablicy: `[38, 27, 43, 3]`.

1. **Podział:** Tablica jest dzielona na dwie połowy: `[38, 27]` oraz `[43, 3]`.
2. **Rekurencja:** Podział trwa aż do uzyskania tablic jednoelementowych: `[38]`, `[27]`, `[43]`, `[3]`.
3. **Scalanie (Merge):**
   - Scalenie `[38]` i `[27]` daje posortowane `[27, 38]`.
   - Scalenie `[43]` i `[3]` daje posortowane `[3, 43]`.
4. **Scalanie końcowe:** Scalenie `[27, 38]` i `[3, 43]` daje wynik końcowy: `[3, 27, 38, 43]`.

2.4. Opis wykorzystanych narzędzi

Projekt został zrealizowany w języku `C++` (standard C++14) w środowisku **Visual Studio 2022**. Do weryfikacji poprawności kodu wykorzystano framework **Google Test**, który jest standardem w testowaniu oprogramowania C++.

Do zarządzania wersjami użyto systemu **Git**, a repozytorium zdalne umieszczono w serwisie **GitHub**. Pozwoliło to na śledzenie historii zmian i symulację pracy zespołowej.

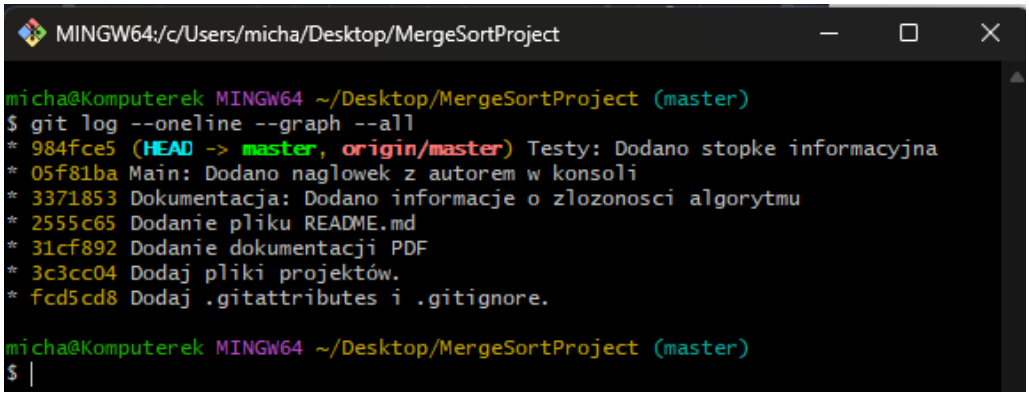
Dokumentacja techniczna została wygenerowana automatycznie przy użyciu narzędzia **Doxygen**, co pozwoliło na uzyskanie profesjonalnego opisu klas i metod w formacie PDF.

2.5. Sposób wykorzystania narzędzi (Git i GitHub)

Podczas pracy nad projektem system kontroli wersji był kluczowy dla zachowania spójności kodu. Poniżej przedstawiono realizację wymaganych scenariuszy użycia Gita.

2.5.1. Historia commitów

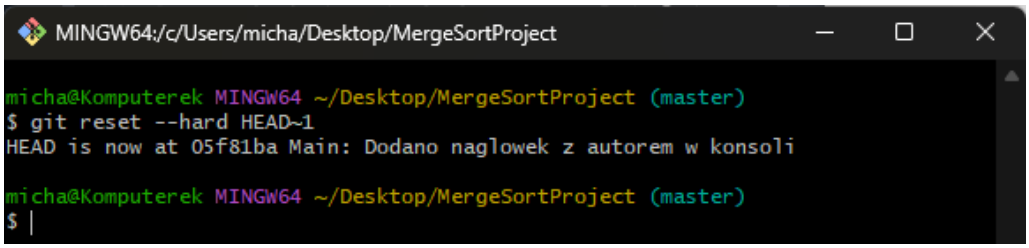
Projekt był rozwijany etapami. Po każdej istotnej zmianie (dodanie algorytmu, dodanie testów, poprawki w dokumentacji) wykonywano *commit*. Rysunek 2.1 przedstawia historię zmian projektu.



Rys. 2.1. Historia commitów repozytorium (Git Bash).

2.5.2. Cofanie zmian (git reset)

W celu przetestowania przywracania starszej wersji kodu, użyto polecenia `git reset`. Pozwala to na wycofanie się do wybranego punktu w historii, usuwając późniejsze zmiany (w trybie hard). Rysunek 2.2 przedstawia efekt tej operacji.

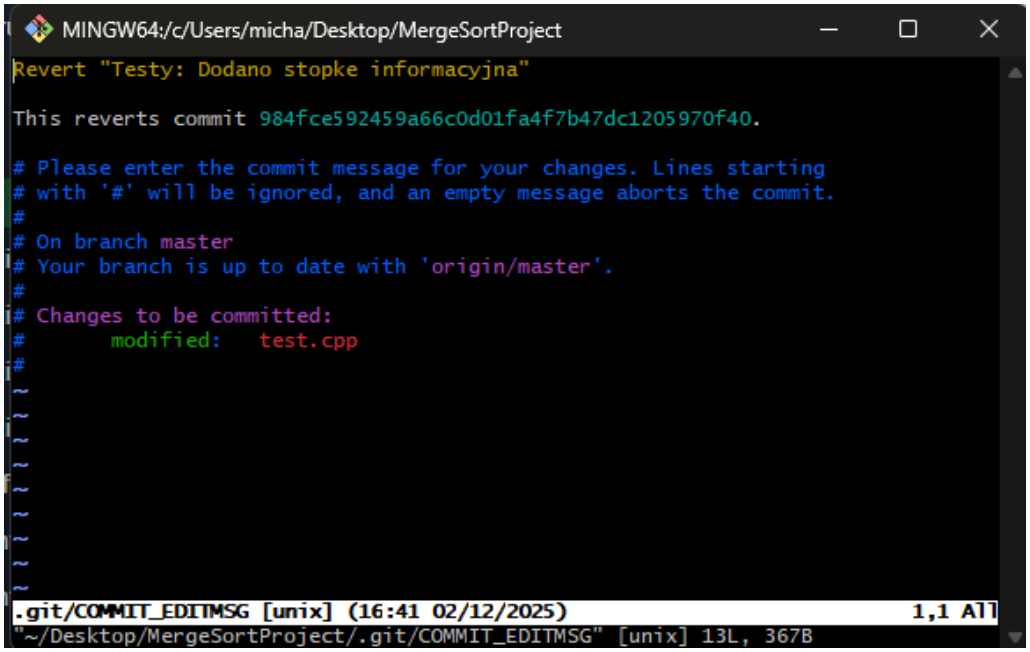


Rys. 2.2. Efekt operacji `git reset` w konsoli.

2.5.3. Usuwanie błędnego commita (git revert)

Aby bezpiecznie wycofać zmiany bez naruszania historii (np. w zespole), zastosowano polecenie `git revert`. Tworzy ono nowy commit, który jest odwrotnością commita

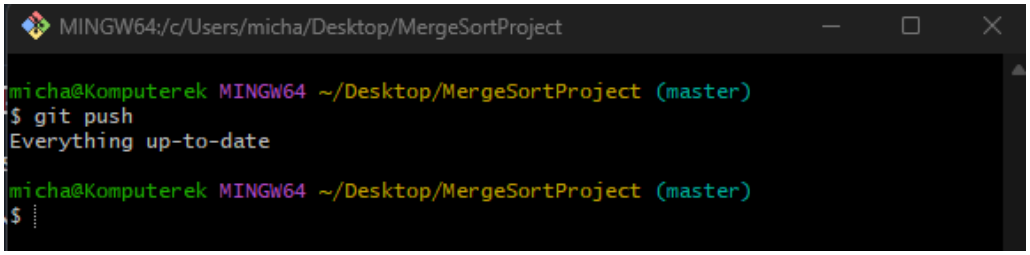
błędnego. Rysunek 2.3 przedstawia przebieg operacji.



Rys. 2.3. Przykład cofnięcia zmian za pomocą `git revert`.

2.5.4. Praca na dwóch lokalizacjach (git pull/push)

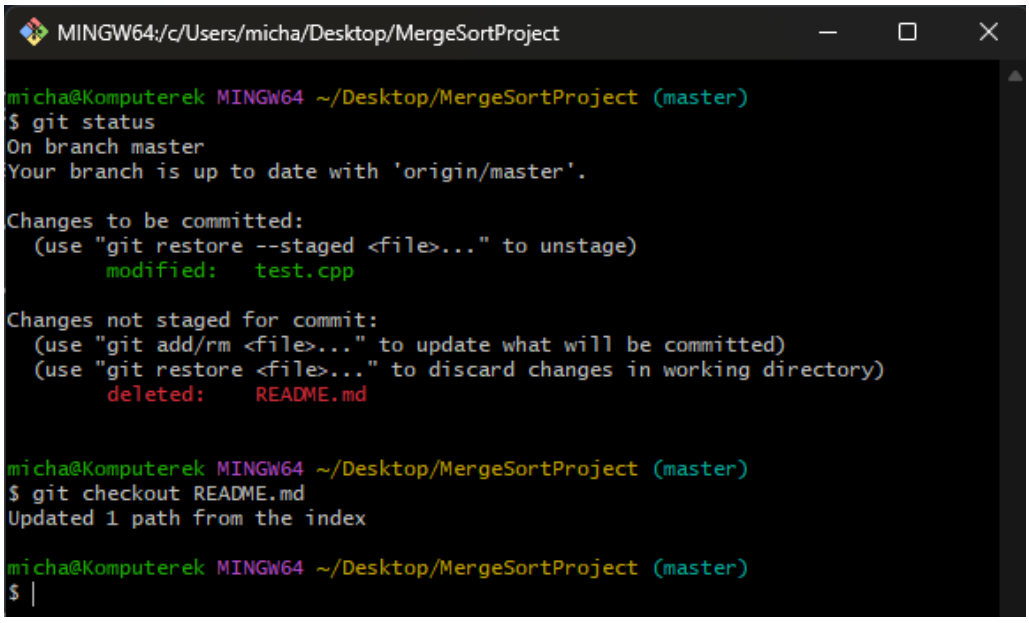
Projekt był synchronizowany ze zdalnym repozytorium GitHub. Zmiany wprowadzone lokalnie były wysyłane poleceniem `git push`, a ewentualne zmiany zdalne pobierane za pomocą `git pull`. Rysunek 2.4 ilustruje proces synchronizacji.



Rys. 2.4. Synchronizacja zmian ze zdalnym repozytorium GitHub.

2.5.5. Odzyskiwanie usuniętego pliku (git checkout)

Zasymulowano awarię polegającą na usunięciu pliku `README.md`. Dzięki systemowi Git, plik został natychmiast przywrócony ze stanu ostatniego commita przy użyciu polecenia `git checkout`. Rysunek 2.5 przedstawia przebieg tej operacji.



Rys. 2.5. Odzyskiwanie usuniętego pliku z repozytorium.

3. Projektowanie

Etap projektowania obejmował zaplanowanie architektury aplikacji, podział na moduły oraz analizę teoretyczną algorytmu. Projekt został zrealizowany zgodnie z paradygmatem programowania obiektowego w języku C++ z wykorzystaniem środowiska Visual Studio.

3.1. Struktura rozwiązania

Rozwiązanie w Visual Studio (*Solution*) zostało podzielone na dwa odrębne projekty, co zapewnia separację logiki biznesowej od testów i zwiększa czytelność kodu:

1. **MergeSortApp** (Projekt główny):
   - `MergeSort.h` – Plik nagłówkowy zawierający definicję szablonowej klasy `MergeSort` oraz pełną implementację algorytmu (ze względu na specyfikę szablonów w C++).
   - `MergeSortApp.cpp` – Plik źródłowy zawierający funkcję `main`, odpowiedzialny za interakcję z użytkownikiem, inicjalizację danych testowych i prezentację wyników w konsoli.
2. **MergeSortProject** (Projekt testowy):
   - `test.cpp` – Plik zawierający implementację 13 testów jednostkowych przy użyciu biblioteki Google Test.

- `packages.config` – Plik konfiguracyjny menedżera pakietów NuGet, zarządzający zależnościami biblioteki `gtest`.

3.2. Projekt klasy MergeSort

Algorytm został zenkapsulowany w klasie statycznej, co jest zgodne z zasadą *Utility Class* (klasa narzędziowa). Dzięki zastosowaniu szablonów (*templates*), klasa jest uniwersalna i może sortować dowolne typy danych, które posiadają zdefiniowany operator porównania (`<`).

Diagram logiczny klasy przedstawia się następująco:

- **Szablon:** `template <typename T>`
- **Metoda publiczna (Interfejs):**
  - `static void sort(std::vector<T>& arr)` – Przyjmuje wektor przez referencję, co zapobiega zbędnemu kopiowaniu danych i zwiększa wydajność. Sprawdza warunki brzegowe (np. pusta tablica) i uruchamia rekurencję.
- **Metody prywatne (Logika wewnętrzna):**
  - `mergeSort(arr, left, right)` – Odpowiada za rekurencyjny podział tablicy na połowy, aż do uzyskania tablic jednoelementowych.
  - `merge(arr, left, mid, right)` – Kluczowa metoda algorytmu. Odpowiada za scalanie dwóch posortowanych podtablic w jedną, wykorzystując pomocniczy wektor tymczasowy.

3.3. Analiza złożoności algorytmu

W fazie projektowania przeanalizowano wydajność algorytmu. Merge Sort jest algorytmem stabilnym, co oznacza, że nie zmienia kolejności elementów o tych samych wartościach klucza.

3.3.1. Złożoność czasowa

Algorytm działa w oparciu o rekurencję:

1. Podział tablicy na połowy zajmuje czas stały $O(1)$.
2. Rekurencyjne sortowanie dwóch połówek o rozmiarze $n/2$ zajmuje $2T(n/2)$.

3. Scalanie (merge) tablicy o rozmiarze n zajmuje czas liniowy $O(n)$.

Całkowita złożoność czasowa dla wszystkich przypadków (optymistycznego, średniego i pesymistycznego) wynosi:

$$T(n) = O(n \log n) \tag{3.1}$$

Jest to znacząca przewaga nad prostymi algorytmami takimi jak Bubble Sort czy Insertion Sort, których złożoność wynosi $O(n^2)$.

3.3.2. Złożoność pamięciowa

Ze względu na konieczność użycia dodatkowej tablicy pomocniczej w procesie scalania (`std::vector<T> temp`), algorytm posiada złożoność pamięciową:

$$S(n) = O(n) \tag{3.2}$$

W projekcie wykorzystano kontenery `std::vector`, które zarządzają pamięcią na stercie (*heap*), co pozwala na sortowanie dużych zbiorów danych bez ryzyka przepełnienia stosu.

3.4. Pseudokod algorytmu

Logiczną strukturę zaimplementowanego algorytmu (niezależną od języka programowania) zaprojektowano w następujący sposób:

```
Funkcja MergeSort(Tablica A, lewy, prawy):
    Jezeli lewy >= prawy:
        Wroc (Koniec rekurencji)

    srodek = lewy + (prawy - lewy) / 2

    MergeSort(A, lewy, srodek)
    MergeSort(A, srodek + 1, prawy)

    Merge(A, lewy, srodek, prawy)

Funkcja Merge(Tablica A, lewy, srodek, prawy):
    Utworz tablice pomocnicza Temp
    i = lewy, j = srodek + 1, k = 0

    Dopoki i <= srodek ORAZ j <= prawy:
        Jezeli A[i] <= A[j]:
            Temp[k++] = A[i++]
```

```
19          W przeciwnym razie:
20              Temp[k++] = A[j++]
21
22      Przepisz pozostale elementy z lewej lub prawej czesci
23      Skopiuj Temp z powrotem do A
```

Listing 1. Pseudokod algorytmu Merge Sort

4. Implementacja

Etap implementacji polegał na utworzeniu klasy szablonowej `MergeSort`, która realizuje algorytm sortowania przez scalanie, oraz przygotowaniu środowiska testowego. Cały projekt został zrealizowany w języku C++ (standard C++14) z użyciem środowiska Visual Studio.

4.1. Struktura projektu

Projekt w Visual Studio składa się z następujących kluczowych plików:

- `MergeSort.h` – Definicja i implementacja szablonowej klasy sortującej.
- `MergeSortApp.cpp` – Główny plik aplikacji demonstracyjnej (funkcja `main`).
- `test.cpp` – Plik zawierający testy jednostkowe Google Test.

4.2. Implementacja klasy `MergeSort`

Klasa `MergeSort` jest klasą statyczną, co oznacza, że nie wymaga tworzenia instancji obiektu, aby wykonać sortowanie. Wykorzystuje szablony (`templates`), aby obsługiwać dowolne typy liczbowe. Jej interfejs przedstawiono na listingu 2.

```
template<typename T>
class MergeSort {
public:
    // Metoda publiczna inicjalizujaca sortowanie
    static void sort(std::vector<T>& arr) {
        if (arr.size() <= 1) return;
        mergeSort(arr, 0, static_cast<int>(arr.size()) - 1);
    }

private:
    // Metoda rekurencyjna dzielaca tablice
    static void mergeSort(std::vector<T>& arr, int left, int right)
    {
        if (left >= right) return;
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }

```

```
20      // Metoda scalajaca posortowane podzbiory
21      static void merge(std::vector<T>& arr, int left, int mid, int
        right);
22  };
```

Listing 2. Interfejs klasy MergeSort (MergeSort.h)

Kluczowym elementem jest metoda `merge`, która scala dwa posortowane podzbiory w jeden większy, zachowując kolejność rosnącą. Dzięki użyciu `std::vector`, algorytm jest bezpieczny i łatwy w użyciu.

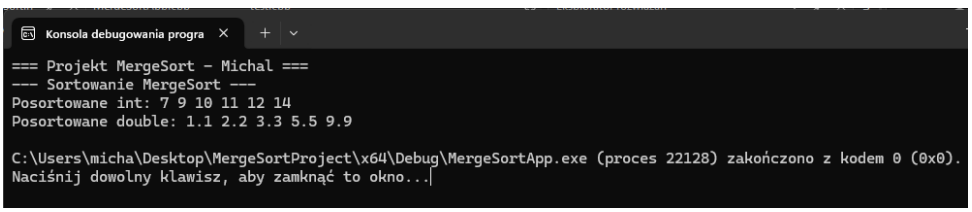
4.3. Aplikacja demonstracyjna (Main)

W pliku `MergeSortApp.cpp` zaimplementowano funkcję `main`, która tworzy dwie instancje tablic (dla typów `int` oraz `double`), wypełnia je danymi, a następnie sortuje i wyświetla wynik. Listing 3 prezentuje fragment tego pliku.

```
1  int main() {
2      std::cout << "=== Projekt MergeSort - Michal ===" << std::endl;
3
4      // Instancja 1: Liczby calkowite
5      std::vector<int> ints = { 12, 7, 14, 9, 10, 11 };
6      MergeSort<int>::sort(ints);
7
8      // Instancja 2: Liczby zmiennoprzecinkowe
9      std::vector<double> doubles = { 1.1, 9.9, 5.5, 2.2 };
10     MergeSort<double>::sort(doubles);
11
12     // Wyswietlanie wynikow (kod pominiety dla czytelnosci)
13     return 0;
14 }
```

Listing 3. Fragment pliku MergeSortApp.cpp

Wynik działania programu skompilowanego w trybie *Release* przedstawiono na rysunku 4.1.



Rys. 4.1. Wynik działania programu – zrzut z konsoli.

4.4. Testowanie i wyniki (Google Test)

Najważniejszą częścią weryfikacji były testy jednostkowe. Zaimplementowano 13 scenariuszy testowych w pliku `test.cpp`, pokrywających wymagania funkcjonalne (m.in. liczby ujemne, duplikaty, puste tablice). Fragment testów przedstawiono na listingu 4.

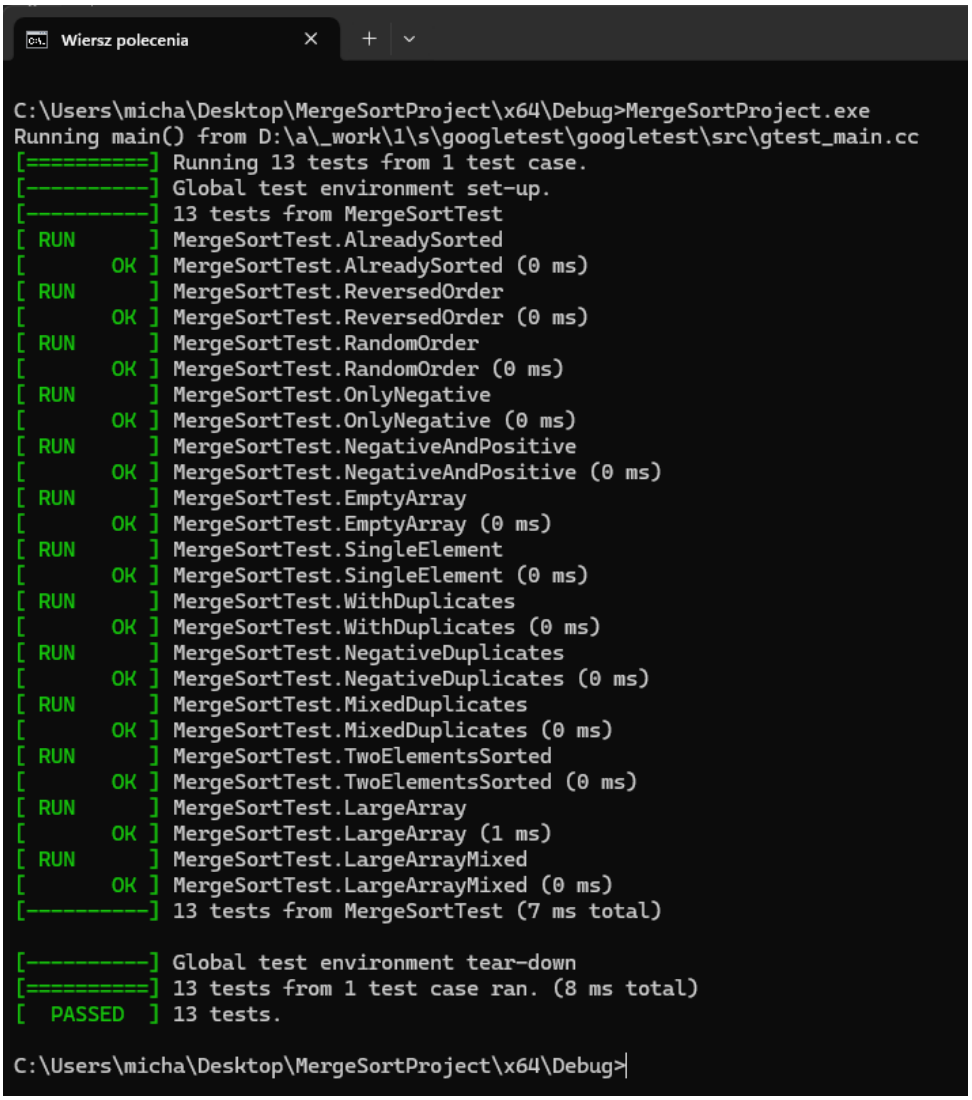
```
TEST(MergeSortTest, AlreadySorted) {
    std::vector<int> arr = { 1, 2, 3, 4, 5 };
    MergeSort<int>::sort(arr);
    EXPECT_EQ(arr, (std::vector<int>{1, 2, 3, 4, 5}));
}

TEST(MergeSortTest, RandomOrder) {
    std::vector<int> arr = { 10, 2, 8, 1, 5 };
    MergeSort<int>::sort(arr);
    EXPECT_EQ(arr, (std::vector<int>{1, 2, 5, 8, 10}));
}

TEST(MergeSortTest, OnlyNegative) {
    std::vector<int> arr = { -5, -1, -10, -3 };
    MergeSort<int>::sort(arr);
    EXPECT_EQ(arr, (std::vector<int>{-10, -5, -3, -1}));
}
```

Listing 4. Przykładowe testy jednostkowe (test.cpp)

Wszystkie testy zakończyły się wynikiem pozytywnym, co potwierdza poprawność implementacji algorytmu. Rezultat wykonania testów w oknie Eksploratora testów Visual Studio (lub konsoli) przedstawiono na rysunku 4.2.



Rys. 4.2. Wynik uruchomienia testów w Eksploratorze testów Visual Studio.

Zbiorcze zestawienie zweryfikowanych przypadków testowych przedstawiono w tabeli 4.1.

Tab. 4.1. Zestawienie wykonanych testów jednostkowych

Nazwa testu	**Opis przypadku**	**Wynik**
AlreadySorted	Tablica już posortowana	OK
ReversedOrder	Tablica odwrócona	OK
RandomOrder	Tablica losowa	OK
OnlyNegative	Tylko liczby ujemne	OK
NegativeAndPositive	Liczby mieszane	OK
EmptyArray	Pusta tablica	OK
SingleElement	Jeden element	OK
WithDuplicates	Duplikaty	OK
NegativeDuplicates	Ujemne z duplikatami	OK
MixedDuplicates	Mieszane z duplikatami	OK
TwoElementsSorted	Dwa elementy rosnąco	OK
LargeArray	Duża tablica (¿100 el.)	OK
LargeArrayMixed	Duża tablica mieszana	OK

5. Wnioski

Realizacja projektu pozwoliła na praktyczne zrozumienie działania algorytmu sortowania przez scalanie (*Merge Sort*), rekurencji oraz mechanizmu **szablonów (*templates*)** w języku `C++`. Dzięki zastosowaniu programowania uogólnionego, udało się stworzyć elastyczne rozwiązanie, które poprawnie obsługuje różne typy danych liczbowych (np. `int`, `double`) bez konieczności duplikowania kodu.

W trakcie realizacji projektu zwrócono szczególną uwagę na:

- implementację algorytmu o złożoności obliczeniowej $O(n \log n)$ zgodnie z paradygmatem "dziel i zwyciężaj",
- weryfikację poprawności oprogramowania za pomocą profesjonalnego frameworka **Google Test**,
- obsługę przypadków brzegowych, takich jak puste tablice, tablice jednoelementowe czy zbiory zawierające duplikaty i liczby ujemne,
- separację logiki aplikacji od kodu testowego poprzez podział na dwa projekty w solucji Visual Studio.

Dodatkowo, zastosowanie narzędzia `Git` oraz platformy `GitHub` umożliwiło kontrolowanie wersji projektu, bezpieczne wprowadzanie zmian oraz symulację pracy w środowisku rozproszonym. Z kolei narzędzie `Doxygen` pozwoliło na automatyczne wygenerowanie profesjonalnej dokumentacji technicznej w formacie LaTeX, co znacząco usprawniło proces tworzenia raportu.

Przeprowadzone testy jednostkowe (łącznie 13 scenariuszy) zakończyły się pełnym sukcesem, co potwierdza niezawodność i stabilność zaimplementowanego algorytmu. Program jest gotowy do dalszego rozwoju lub integracji z większymi systemami wymagającymi efektywnego sortowania danych.

Bibliografia

[1] GeeksforGeeks. *Merge Sort Algorithm*. URL: https://www.geeksforgeeks.org/merge-sort/ (term. wiz. 30. 11. 2025).

[2] GeeksforGeeks. *Templates in C++ with Examples*. URL: https://www.geeksforgeeks.org/templates-cpp/ (term. wiz. 30. 11. 2025).

[3] Google. *GoogleTest Primer*. URL: https://google.github.io/googletest/primer.html (term. wiz. 30. 11. 2025).

[4] Dimitri van Heesch. *Doxygen Manual*. URL: https://www.doxygen.nl/manual/index.html (term. wiz. 30. 11. 2025).

Spis rysunków

Spis tabel

Spis listingów