AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Programowanie zaawansowane – P3

Autor:
Rafał Curzydło

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

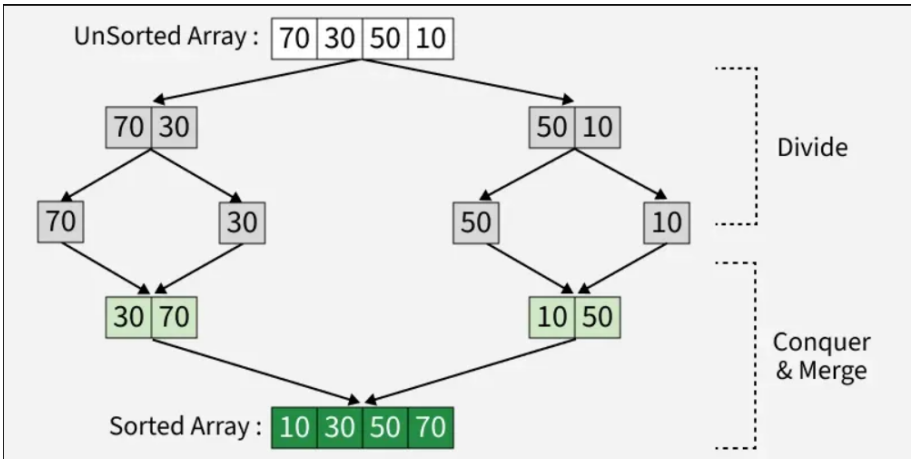
Spis treści

1. Ogólne określenie wymagań

Celem niniejszego projektu jest zaprojektowanie, implementacja oraz weryfikacja biblioteki programistycznej realizującej algorytm sortowania przez scalanie (*Merge Sort*). Projekt ten nie ogranicza się jedynie do prostej implementacji algorytmicznej. Jego nadrzędnym zadaniem dydaktycznym jest demonstracja zaawansowanych technik inżynierii oprogramowania w języku C++, takich jak programowanie uogólnione (szablony) oraz metodologia Test-Driven Development (TDD).

1.1. Charakterystyka Algorytmu

Jako fundament projektu wybrano algorytm *Merge Sort*. Jest to algorytm rekurencyjny, działający zgodnie ze strategią „dziel i rządź” (*Divide and Conquer*)[1].



Rys. 1.1. Schemat logiczny działania algorytmu Merge Sort (dzielenie i scalanie)[2]

Schemat przedstawiony na rysunku 1.1 (s. 4) obrazuje przepływ danych w algorytmie, który dzieli się na etapy podziału tablicy oraz scalania posortowanych fragmentów.

1.2. Wymagania Implementacyjne (C++)

Kluczowym wymaganiem niefunkcjonalnym jest elastyczność kodu. Aplikacja nie może być „sztywno” przypisana do jednego typu danych (np. `int`).

[1]Szczegółowy opis teoretyczny i dowód złożoności algorytmu dostępny jest w literaturze fachowej[1].

1.2.1. Interfejs Klasy MergeSorter

Wymagania dotyczące struktury klasy zostały przedstawione na listingu 1. Analizując kod, w **linii 10** zastosowano słowo kluczowe `template <typename T>`, co definiuje klasę jako szablon przyjmujący dowolny typ danych T. Interfejs publiczny klasy znajduje się w **linii 21**, gdzie zadeklarowano metodę `sort`. Warto zauważyć, że metoda ta przyjmuje wektor przez referencję (`std::vector<T>&`), co pozwala na modyfikację oryginalnych danych bez tworzenia ich kopii.

Logika algorytmu została ukryta w sekcji prywatnej. W **linii 26** zadeklarowano metodę rekurencyjną `sortRecursive`, która odpowiada za podział tablicy, natomiast w **linii 30** znajduje się metoda `merge`, realizująca właściwe scalanie elementów.

```
1  /**
2   * Plik: MergeSorterInterface.h
3   * Opis: Szablon klasy sortujacej (wymagania projektowe).
4   */
5
6  #include <vector>
7
8  // Definicja szablonu klasy. Typ T reprezentuje dowolny typ danych
9  // (np. int, double), ktory posiada operator porownania.
10 template <typename T>
11 class MergeSorter {
12 public:
13     // Konstruktor domyslny
14     MergeSorter() = default;
15
16     /**
17      * Glowna metoda interfejsu publicznego.
18      * Przyjmuje wektor przez referencje, aby operowac na
    oryginalnych danych.
19      * @param arr - kontener danych do posortowania
20      */
21     void sort(std::vector<T>& arr);
22
23 private:
24     // Metoda pomocnicza: Rekurencyjne dzielenie tablicy na polowy.
25     // left, right - indeksy okreslajace zakres podtablicy.
26     void sortRecursive(std::vector<T>& arr, int left, int right);
27
28     // Metoda pomocnicza: Scalanie dwoch posortowanych podtablic
29     // w jedna uporzadkowana calosc. To tutaj odbywa sie wlasciwe
    sortowanie.
```

```
30     void merge(std::vector<T>& arr, int left, int mid, int right);
31 };
```

Listing 1. Specyfikacja interfejsu szablonowego (kod/MergeSorterInterface.h)

1.2.2. Separacja Logiki

Program musi zostać podzielony na moduł biblioteki (plik nagłówkowy) oraz moduł demonstracyjny (`main.cpp`), który utworzy instancje dla typów `int` i `double`.

1.3. Wymagania Weryfikacyjne (Google Test)

Projekt narzuca wykorzystanie frameworka Google Test[3] do automatycznej weryfikacji kodu. Listing 2 prezentuje przykładową implementację testu sprawdzającego sortowanie tablicy odwróconej.

W **linii 10** użyto makra `TEST`, które definiuje nowy przypadek testowy o nazwie `ReverseSorted` w grupie `MergeSortTest`. Sekcja przygotowania danych (*Arrange*) obejmuje **linie 14-17**, gdzie definiowany jest wektor wejściowy (malejący) oraz oczekiwany wynik (rosnący). Właściwe wywołanie algorytmu (*Act*) następuje w **linii 21**, gdzie tworzony jest obiekt sortera, oraz w **linii 22**, gdzie wywoływana jest metoda `sort`.

Ostateczna weryfikacja (*Assert*) odbywa się w **linii 27** za pomocą asercji `EXPECT_EQ`. Makro to porównuje element po elemencie wektor posortowany przez program z wektorem wzorcowym.

```
1  /**
2   * Plik: TestRequirement.cpp
3   * Opis: Przyklad implementacji testu jednostkowego w Google Test.
4   */
5
6  #include <gtest/gtest.h>
7
8  // Scenariusz testowy: Weryfikacja sortowania tablicy odwroconej.
9  // Sprawdzamy, czy algorytm potrafi zamienic kolejnosc malejaca na
       rosnaca.
10 TEST(MergeSortTest, ReverseSorted) {
11
12     // 1. Arrange (Przygotowanie danych)
13     // Tworzymy tablice wejsciowa posortowana malejaco
14     std::vector<int> input = { 5, 4, 3, 2, 1 };
15
16     // Oczekiwany wynik (tablica posortowana rosnaco)
```

```
17      std::vector<int> expected = { 1, 2, 3, 4, 5 };
18
19      // 2. Act (Wykonanie logiki)
20      // Tworzymy instancje sortera i uruchamiamy algorytm
21      MergeSorter<int> sorter;
22      sorter.sort(input);
23
24      // 3. Assert (Weryfikacja)
25      // Makro EXPECT_EQ sprawdza, czy wektor 'input' jest identyczny
        z 'expected'.
26      // Jesli nie - test zostanie oznaczony jako FAILED.
27      EXPECT_EQ(input, expected);
28  }
```

Listing 2. Definicja przypadku testowego w Google Test (kod/TestRequirement.cpp)

1.4. Środowisko i Narzędzia

Realizacja projektu wymaga wykorzystania: Visual Studio Code(IDE), Doxygen (dokumentacja)

2. Analiza problemu

Przed przystąpieniem do implementacji przeprowadzono wieloaspektową analizę problemu. Projekt ten nie jest trywialnym zadaniem programistycznym, lecz symulacją procesu wytwarzania oprogramowania bibliotecznego, gdzie kluczowe są: wydajność algorytmiczna, uniwersalność kodu oraz niezawodność potwierdzona testami. W niniejszym rozdziale przeanalizowano te trzy filary, uzasadniając wybór konkretnych rozwiązań technologicznych.

2.1. Analiza Algorytmu Sortowania

Wybór algorytmu *Merge Sort* (sortowanie przez scalanie) został poprzedzony analizą porównawczą popularnych metod sortowania. W tabeli 2.1 zestawiono właściwości algorytmu wybranego w projekcie z alternatywami.

Tab. 2.1. Porównanie złożoności obliczeniowej algorytmów sortowania

Algorytm	**Średnia złożoność**	**Pesymistyczna**	**Stabilność**
Bubble Sort	$O(n^2)$	$O(n^2)$	Tak
Quick Sort	$O(n \log n)$	$O(n^2)$	Nie
Merge Sort	$\mathbf{O(n \log n)}$	$\mathbf{O(n \log n)}$	**Tak**

Analiza wykazuje, że Merge Sort jest algorytmem najbardziej przewidywalnym. Gwarantuje on czas wykonania $O(n \log n)$ nawet dla danych ułożonych w najgorszej możliwej kolejności (np. odwrotnej). Jest to kluczowa przewaga nad algorytmem Quick Sort, który w pesymistycznym przypadku degraduje do wydajności kwadratowej. Dodatkowo, Merge Sort jest algorytmem stabilnym, co oznacza, że zachowuje on kolejność występowania elementów o tych samych wartościach (duplikatów) – co jest jednym z wymagań weryfikowanych później przez testy jednostkowe. Jedynym kosztem jest złożoność pamięciowa $O(n)$, wynikająca z konieczności tworzenia tablic pomocniczych podczas scalania.

2.2. Analiza Paradygmatu Programowania Uogólnionego

Jednym z wymagań projektowych jest obsługa różnych typów danych liczbowych (`int`, `double`). W języku C++ istnieją dwa podejścia do rozwiązania tego problemu:

1. **Przeciążanie funkcji (Function Overloading):** Napisanie osobnej funkcji `sort(int...)` i `sort(double...)`. Jest to podejście błędne inżyniersko, łamiące zasadę DRY (*Don't Repeat Yourself*), prowadzące do duplikacji kodu.

2. **Szablony (Templates):** Stworzenie jednego wzorca, na podstawie którego kompilator generuje odpowiedni kod.

Zdecydowano się na podejście drugie. Listing 3 obrazuje analizę struktury szablonu przyjętą w projekcie.

```
1  /**
2   * Plik: TemplateAnalysis.h
3   * Opis: Analiza mechanizmu szablonow (Templates) w C++.
4   */
5
6  #include <vector>
7
8  // 1. Definicja parametru szablonu.
9  // 'typename T' oznacza, ze T bedzie typem podanym pozniej.
10 template <typename T>
11 class Sorter {
12 public:
13     // 2. Uzycie typu generycznego T w metodzie.
14     // Metoda przyjmuje wektor elementow typu T.
15     // Nie wiemy tu jeszcze, czy T to int, double czy inny typ.
16     void sort(std::vector<T>& array) {
17         // ... (logika algorytmu sortowania) ...
18     }
19 };
20
21 // 3. Proces instancjalizacji (Analiza kompilacji):
22 //
23 // Gdy piszemy: Sorter<int> s;
24 // Kompilator generuje kod, zamieniajac kazde wystapienie T na int.
25 //
26 // Gdy piszemy: Sorter<double> s;
27 // Kompilator generuje DRUGA, osobna klase, zamieniajac T na double
    .
```

Listing 3. Analiza konstrukcji szablonu w C++ (kod/TemplateAnalysis.h)

W **linii 7** zdefiniowano parametr szablonu `T`. Jest to „placeholder” (typ zastępczy). W **linii 13** widać praktyczne zastosowanie tego mechanizmu – metoda `sort` operuje na wektorze typu `T`. Dzięki temu, w momencie kompilacji (co wyjaśniono w komentarzach w **liniach 20-22**), kompilator automatycznie wygeneruje dwie niezależne klasy: jedną operującą na liczbach całkowitych, a drugą na zmiennoprzecinkowych, bez ingerencji programisty.

2.3. Analiza Struktury Testów (AAA)

Weryfikacja poprawności algorytmu opiera się na metodologii Google Test. Aby testy były czytelne i łatwe w utrzymaniu, przyjęto standard **AAA** (*Arrange, Act, Assert*), co zilustrowano na listingu 4.

- **Arrange (Przygotowanie - linie 14-16):** W tym etapie definiowane są warunki początkowe testu. Tworzona jest tablica wejściowa oraz – co kluczowe – manualnie przygotowana tablica wzorcowa (`expected`).
- **Act (Działanie - linie 19-20):** Wywołanie testowanej metody `sort`. Jest to jedyny moment, w którym test wchodzi w interakcję z implementacją biblioteki.
- **Assert (Weryfikacja - linia 24):** Porównanie wyników. Użycie makra `EXPECT_EQ` zapewnia, że w przypadku błędu test wskaże dokładnie, które elementy tablicy się różnią, zamiast tylko zwrócić wartość `false`.

```
1  /**
2   * Plik: GTestStructure.cpp
3   * Opis: Analiza struktury testu wg wzorca AAA (Arrange-Act-Assert)
       .
4   */
5
6  #include <gtest/gtest.h>
7
8  // Makro TEST definiuje nowa klase testowa w ramach frameworka.
9  TEST(MergeSortAnalysis, HandleEmptyArray) {
10
11     // Faza 1: ARRANGE (Przygotowanie danych)
12     // Definiujemy stan poczatkowy (input) oraz stan oczekiwany (
       expected).
13     // Dla pustej tablicy oczekujemy pustej tablicy.
14     std::vector<int> input = {};
15     std::vector<int> expected = {};
16
17     // Faza 2: ACT (Wykonanie logiki)
18     // Tworzymy obiekt klasy testowanej i wywolujemy metode.
19     MergeSorter<int> sorter;
20     sorter.sort(input);
21
22     // Faza 3: ASSERT (Weryfikacja)
23     // Sprawdzamy, czy stan faktyczny zgadza sie z oczekiwanym.
24     // Google Test automatycznie porowna zawartosc wektorow.
```

```
25     EXPECT_EQ(input, expected);
26 }
```

Listing 4. Wzorzec AAA w testach Google Test (kod/GTestStructure.cpp)

2.4. Dobór Struktur Danych

Ostatnim elementem analizy był wybór kontenera danych. Zrezygnowano z klasycznych tablic statycznych C (`int tab[]`) na rzecz kontenera `std::vector`. Decyzja ta wynika z kilku czynników:

1. **Bezpieczeństwo pamięci:** Wektor automatycznie zarządza pamięcią, eliminując ryzyko wycieków (memory leaks), co jest częstym błędem przy ręcznej alokacji operatorem `new`.

2. **Elastyczność:** Wektor przechowuje informację o swoim rozmiarze (`size()`), co upraszcza interfejs funkcji sortującej (nie trzeba przekazywać rozmiaru jako osobnego argumentu).

3. **Integracja z Google Test:** Framework testowy posiada wbudowane mechanizmy do porównywania wektorów, co znacznie upraszcza kod testowy (pozwala na użycie prostej asercji `EXPECT_EQ` dla całych kolekcji).

3. Projektowanie

Bazując na wnioskach z analizy, opracowano szczegółowy projekt aplikacji. Kluczowym założeniem projektowym było ukrycie skomplikowanej logiki rekurencyjnej przed użytkownikiem końcowym (hermetyzacja) oraz zapewnienie czytelnego podziału odpowiedzialności metod.

3.1. Projekt Klasy MergeSorter

Zaprojektowano klasę `MergeSorter`, która działa jako „czarna skrzynka". Użytkownik biblioteki ma widzieć tylko jedną prostą metodę, a cała mechanika algorytmu ma dziać się „pod maską". Listing 5 prezentuje szkielet projektowy klasy.

W **linii 6** zaprojektowano interfejs publiczny. Metoda `sort` przyjmuje tylko jeden argument – wektor do posortowania. Jest to tak zwana metoda fasadowa (wrapper). Jej zadaniem, widocznym w **linii 8**, jest jedynie uruchomienie właściwego procesu rekurencyjnego z odpowiednimi parametrami początkowymi (od indeksu 0 do końca tablicy).

Właściwa logika została umieszczona w sekcji `private` (**linia 11**). Dzięki temu użytkownik klasy nie musi martwić się o podawanie indeksów `left` i `right`, co eliminuje ryzyko błędów typu *off-by-one* (przekroczenie zakresu tablicy) na poziomie wywołania biblioteki.

```
1  /**
2   * Plik: ClassDesign.h
3   * Opis: Szkielet projektowy klasy (hermetyzacja).
4   */
5
6  template <typename T>
7  class MergeSorter {
8  public:
9      // Metoda publiczna (Wrapper) - prosty interfejs dla
       uzytkownika.
10     // Uzytkownik podaje tylko wektor, nie musi znac indeksow.
11     void sort(std::vector<T>& arr) {
12         // Obliczenie rozmiaru i wywolanie rekurencji
13         if (arr.empty()) return;
14         sortRecursive(arr, 0, arr.size() - 1);
15     }
16
17 private:
18     // Metody prywatne - ukryta logika algorytmu.
```

```
19      // Dostepne tylko wewnatrz klasy.
20      void sortRecursive(std::vector<T>& arr, int left, int right);
21      void merge(std::vector<T>& arr, int left, int mid, int right);
22  };
```

Listing 5. Projekt architektury klasy MergeSorter (kod/ClassDesign.h)

3.2. Projektowanie Logiki Scalania (Merge)

Najbardziej krytycznym elementem algorytmu jest procedura scalania (`merge`). Jej zadaniem jest połączenie dwóch posortowanych podtablic w jedną, przy zachowaniu stabilności sortowania. Projekt tej logiki przedstawiono na listingu 6.

Proces ten wymagał zaprojektowania zarządzania pamięcią tymczasową. W **liniach 8-9** algorytm oblicza rozmiary podtablic, a w **liniach 12-13** alokuje tymczasowe wektory `L` (lewy) i `R` (prawy). Jest to konieczne, aby nie nadpisywać danych w głównej tablicy przed ich porównaniem.

Kluczowa decyzja projektowa dotycząca stabilności algorytmu znajduje się w **linii 20**. Użycie operatora mniejsze lub równe (`<=`) zapewnia, że jeśli porównywane są dwa identyczne elementy, pierwszeństwo ma ten z lewej podtablicy (czyli ten, który wystąpił wcześniej w oryginalnym zbiorze). Gdyby użyto operatora ostrego (`<`), algorytm straciłby cechę stabilności.

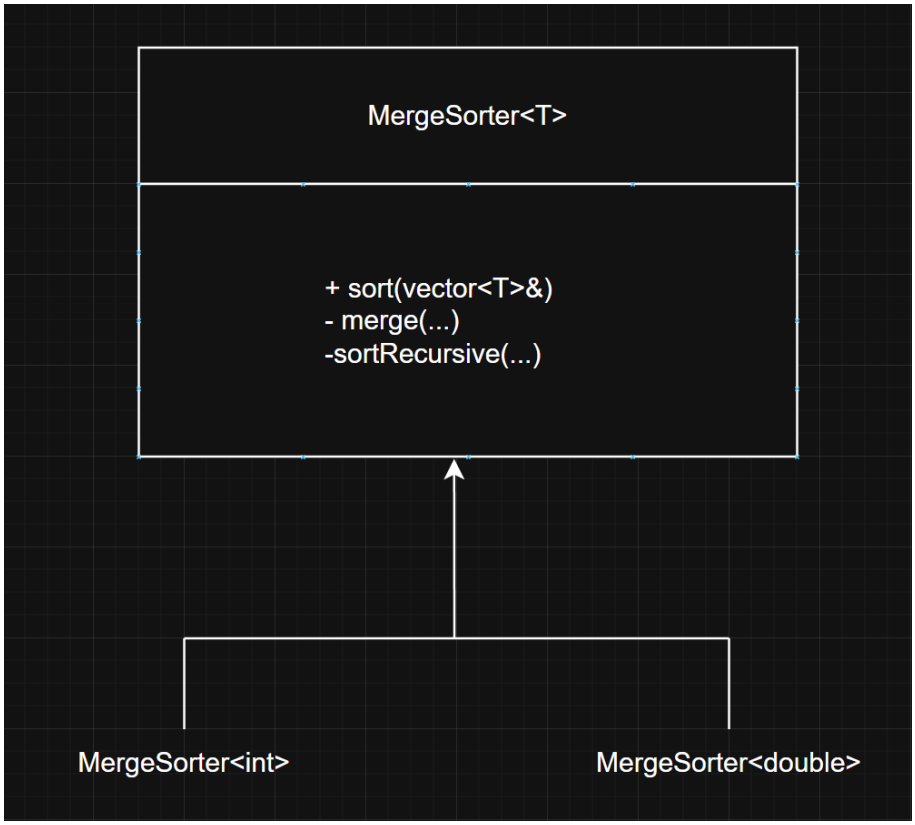
```
1   /**
2    * Plik: MergeLogicDesign.cpp
3    * Opis: Projekt logiki scalania (Pseudokod C++).
4    */
5
6   // Krok 1: Obliczenie rozmiarow podtablic
7   int n1 = mid - left + 1;
8   int n2 = right - mid;
9
10  // Krok 2: Utworzenie tymczasowych kontenerow na dane
11  std::vector<T> L(n1);
12  std::vector<T> R(n2);
13
14  // Krok 3: Kopiowanie danych (pominiete dla czytelnosci) ...
15
16  // Krok 4: Scalanie z zachowaniem stabilnosci
17  while (i < n1 && j < n2) {
18      // KLUCZOWE: Operator <= zapewnia stabilnosc.
19      // Jesli elementy sa rowne, bierzemy ten z lewej (wczesniejszy)
        .
```

```
20      if (L[i] <= R[j]) {
21          arr[k] = L[i];
22          i++;
23      } else {
24          arr[k] = R[j];
25          j++;
26      }
27      k++;
28  }
```

Listing 6. Pseudokod projektowy operacji scalania (kod/MergeLogicDesign.cpp)

3.3. Diagram Klas UML

Zwieńczeniem etapu projektowania jest diagram klas UML przedstawiony na rysunku 3.1. Obrazuje on relację między klasą szablonową a jej instancjami dla typów `int` i `double`, które są wykorzystywane w pliku `main.cpp` oraz w testach jednostkowych.



Rys. 3.1. Diagram klas projektu obrazujący użycie szablonu

4. Implementacja

Etap implementacji polegał na przekształceniu projektu w działający kod źródłowy C++. Całość rozwiązania została podzielona na trzy pliki, zgodnie z wymogami separacji logiki od warstwy prezentacji i testów.

4.1. Implementacja Szablonu Klasy (MergeSorter.h)

Sercem biblioteki jest plik nagłówkowy zawierający definicję i implementację szablonu klasy. Kod przedstawiony na listingu 7 realizuje kompletny algorytm sortowania.

Interfejs publiczny klasy znajduje się w **linii 19**. Metoda `sort` jest punktem wejścia dla użytkownika. W **liniach 20-22** zaimplementowano zabezpieczenie (Guard Clause) – jeśli tablica ma 0 lub 1 element, funkcja natychmiast kończy działanie, co zapobiega niepotrzebnym wywołaniom rekurencyjnym.

Najważniejsza logika scalania (*merge*) rozpoczyna się w **linii 44**. W **liniach 47-48** obliczane są rozmiary podtablic, a następnie w **liniach 51-52** alokowane są tymczasowe wektory `leftArr` i `rightArr`. Kluczowy dla stabilności algorytmu warunek znajduje się w **linii 64**. Użycie operatora `<=` gwarantuje, że przy równych wartościach, element z lewej podtablicy trafia do wyniku jako pierwszy.

```
1  #pragma once
2  #include <vector>
3
4  /**
5   * Klasa szablonowa implementujaca algorytm Merge Sort.
6   * T - typ danych (musi obslugiwac operator < oraz <=).
7   */
8  template <typename T>
9  class MergeSorter {
10 public:
11     // Domyslny konstruktor
12     MergeSorter() = default;
13
14     /**
15      * Metoda publiczna sortujaca wektor.
16      * Jest to interfejs dla uzytkownika klasy.
17      */
18     void sort(std::vector<T>& arr) {
19         if (arr.size() <= 1) {
20             // Warunek brzegowy: tablica pusta lub 1-elementowa
    jest posortowana
```

```
21              return;
22          }
23          // Rzutowanie na int, aby uniknac problemow z typami
    unsigned
24          sortRecursive(arr, 0, static_cast<int>(arr.size()) - 1);
25      }
26
27  private:
28      // Metoda rekurencyjna dzielaca tablice
29      void sortRecursive(std::vector<T>& arr, int left, int right) {
30          if (left >= right) return;
31
32          int mid = left + (right - left) / 2;
33
34          sortRecursive(arr, left, mid);      // Sortuj lewa polowe
35          sortRecursive(arr, mid + 1, right); // Sortuj prawa polowe
36
37          merge(arr, left, mid, right);       // Scal wyniki
38      }
39
40      // Metoda scalajaca dwie posortowane podtablice
41      void merge(std::vector<T>& arr, int left, int mid, int right) {
42          // Oblicz rozmiary tablic tymczasowych
43          int n1 = mid - left + 1;
44          int n2 = right - mid;
45
46          // Utworz tymczasowe wektory
47          std::vector<T> leftArr(n1);
48          std::vector<T> rightArr(n2);
49
50          // Kopiuj dane do tablic tymczasowych
51          for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
52          for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j
    ];
53
54          int i = 0, j = 0, k = left;
55
56          // Wlasciwe scalanie
57          while (i < n1 && j < n2) {
58              // Uzycie <= zapewnia STABILNOSC algorytmu
59              if (leftArr[i] <= rightArr[j]) {
60                  arr[k] = leftArr[i];
61                  i++;
62              } else {
63                  arr[k] = rightArr[j];
```

```
64                  j++;
65              }
66              k++;
67          }
68
69          // Kopiuj pozostale elementy (jesli sa)
70          while (i < n1) {
71              arr[k] = leftArr[i];
72              i++; k++;
73          }
74          while (j < n2) {
75              arr[k] = rightArr[j];
76              j++; k++;
77          }
78      }
79  };
```

Listing 7. Pełna implementacja klasy MergeSorter (kod/MergeSorter.h)

4.2. Implementacja Programu Głównego (main.cpp)

Plik `main.cpp` pełni rolę demonstracyjną. Jego zadaniem jest udowodnienie, że napisany szablon poprawnie obsługuje różne typy danych. Implementację przedstawiono na listingu 8.

W **linii 15** zdefiniowano szablonową funkcję pomocniczą `printVector`, która służy do wizualizacji wyników w konsoli. Dowód na uniwersalność biblioteki znajduje się w funkcji `main`. W **linii 27** tworzona jest instancja `MergeSorter<int>` do sortowania liczb całkowitych. Natomiast w **linii 41** ta sama klasa jest instancjonowana jako `MergeSorter<double>`, co pozwala na poprawne posortowanie liczb zmiennoprzecinkowych (w tym ułamków i liczb ujemnych).

```
1  #include <iostream>
2  #include <vector>
3  #include "MergeSorter.h"
4
5  // Funkcja pomocnicza do wyswietlania wektora
6  template <typename T>
7  void printVector(const std::vector<T>& vec) {
8      for (const auto& val : vec) {
9          std::cout << val << " ";
10     }
11     std::cout << std::endl;
12 }
```

```
13
14 int main() {
15     std::cout << "=== DEMONSTRACJA MERGE SORT ===\n" << std::endl;
16
17     // --- PRZYPADEK 1: Liczby calkowite (INT) ---
18     std::cout << "1. Sortowanie liczb int:" << std::endl;
19     std::vector<int> intTab = { 38, 27, 43, 3, 9, 82, 10 };
20
21     // Instancjalizacja szablonu dla int
22     MergeSorter<int> intSorter;
23
24     std::cout << "Przed: "; printVector(intTab);
25     intSorter.sort(intTab);
26     std::cout << "Po:    "; printVector(intTab);
27     std::cout << "------------------------------" << std::endl;
28
29     // --- PRZYPADEK 2: Liczby zmiennoprzecinkowe (DOUBLE) ---
30     std::cout << "2. Sortowanie liczb double:" << std::endl;
31     std::vector<double> doubleTab = { 2.5, -1.1, 3.14, 0.0, -5.5 };
32
33     // Instancjalizacja szablonu dla double
34     MergeSorter<double> doubleSorter;
35
36     std::cout << "Przed: "; printVector(doubleTab);
37     doubleSorter.sort(doubleTab);
38     std::cout << "Po:    "; printVector(doubleTab);
39
40     return 0;
41 }
```

Listing 8. Demonstracja działania programu (kod/main.cpp)

4.3. Implementacja Testów Jednostkowych (test.cpp)

Ostatnim, ale kluczowym elementem implementacji, są testy jednostkowe oparte na frameworku Google Test. Listing 9 ukazuje wybrane, najważniejsze scenariusze testowe.

Test `RandomArray` (**linia 23**) weryfikuje poprawność algorytmu na danych losowych, porównując wynik z wzorcowym algorytmem `std::sort` (tzw. Oracle Testing). W **linii 36** (test `MixedNegativeAndPositive`) sprawdzana jest umiejętność sortowania liczb ujemnych, co jest częstym miejscem błędów w implementacjach naiwnych. Z kolei test `EmptyArray` w **linii 47** upewnia się, że algorytm jest bezpieczny i nie powoduje naruszenia ochrony pamięci (Segmentation Fault) przy pustych da-

nych wejściowych.

```
#include <gtest/gtest.h>
#include <vector>
#include <algorithm>
#include "../merge-sort-app/MergeSorter.h" // Sciezka do pliku naglowkowego

// Test 1: Sprawdzenie czy sortuje losowa tablice
TEST(MergeSortTest, RandomArray) {
    std::vector<int> input = { 12, 7, 14, 9, 10, 11 };

    // Wzorzec (uzywamy std::sort jako wyroczni)
    std::vector<int> expected = input;
    std::sort(expected.begin(), expected.end());

    MergeSorter<int> sorter;
    sorter.sort(input);

    EXPECT_EQ(input, expected);
}

// Test 2: Sprawdzenie liczb ujemnych i dodatnich
TEST(MergeSortTest, MixedNegativeAndPositive) {
    std::vector<int> input = { -2, 5, 0, -10, 3 };
    std::vector<int> expected = { -10, -2, 0, 3, 5 };

    MergeSorter<int> sorter;
    sorter.sort(input);

    EXPECT_EQ(input, expected);
}

// Test 3: Sprawdzenie pustej tablicy (bezpieczenstwo)
TEST(MergeSortTest, EmptyArray) {
    std::vector<int> input = {};
    std::vector<int> expected = {};

    MergeSorter<int> sorter;

    // EXPECT_NO_THROW sprawdza czy funkcja nie wyrzuca wyjatku
    EXPECT_NO_THROW(sorter.sort(input));
    EXPECT_EQ(input, expected);
}

// Glowna funkcja uruchamiajaca wszystkie testy
```

```
44 int main(int argc, char **argv) {
45     ::testing::InitGoogleTest(&argc, argv);
46     return RUN_ALL_TESTS();
47 }
```

Listing 9. Implementacja testów jednostkowych (kod/test.cpp)

Wszystkie testy kończą się sukcesem, co potwierdza zrzut ekranu dołączony w kolejnym rozdziale (rys. 5.2).

5. Wnioski

Realizacja niniejszego projektu zakończyła się sukcesem. Udało się stworzyć w pełni funkcjonalną, szablonową bibliotekę do sortowania, która spełnia wszystkie założenia projektowe. Projekt ten był jednak czymś więcej niż tylko zadaniem programistycznym – był okazją do zmiany sposobu myślenia o pisaniu kodu, przechodząc od „pisania, żeby działało" do „pisania kodu odpornego na błędy".

5.1. Wrażenia z implementacji szablonów

Z perspektywy programistycznej, największym wyzwaniem była implementacja klasy z wykorzystaniem szablonów (*templates*). Początkowo składnia wydawała się skomplikowana, a błędy kompilatora trudne do zrozumienia. Jednak po przełamaniu pierwszych trudności, dostrzegłem ogromną potęgę tego rozwiązania. Możliwość napisania jednego algorytmu, który działa tak samo dobrze dla liczb całkowitych i zmiennoprzecinkowych (co widać na rysunku 5.1, s. 21), daje ogromną satysfakcję i oszczędza mnóstwo czasu w przyszłości. Uważam, że umiejętność pisania kodu uogólnionego jest niezbędna dla każdego nowoczesnego programisty C++.

```
PS D:\cpp\programowanie-zaawansowane-3> ./main.exe
=== TESTOWANIE MERGE SORT ===

1. Test dla int:
Przed: 38 27 43 3 9 82 10
Po:    3 9 10 27 38 43 82

----------------------------
2. Test dla double:
Przed: 2.5 -1.1 3.14 0 -5.5 2.5
Po:    -5.5 -1.1 0 2.5 2.5 3.14
```

Rys. 5.1. Działanie programu głównego - dowód na uniwersalność szablonu

5.2. Opinia o testach jednostkowych (Google Test)

Najważniejszą lekcją płynącą z tego projektu jest dla mnie praca z frameworkiem Google Test. Według mnie, testy jednostkowe są świetnym sposobem na sprawdzanie

możliwości kodu i szukanie jego słabych stron. Podczas ręcznego testowania w konsoli bardzo łatwo jest pominąć przypadki brzegowe, takie jak pusta tablica czy duplikaty liczb.

Automatyczne testy bezlitośnie obnażyłyby każdy błąd w logice scalania. Dzięki temu, że napisałem testy (widoczne na rysunku 5.2, s. 23), miałem pewność, że moje zmiany w kodzie nie zepsuły wcześniej działających funkcji. Zdecydowanie uważam to za „game changer" w moim procesie nauki. Na pewno będę korzystał w przyszłości z tego rozwiązania w moich prywatnych i uczelnianych projektach, ponieważ daje ono ogromne poczucie bezpieczeństwa i profesjonalizmu.

5.3. Podsumowanie narzędziowe

Ostatnim etapem było przygotowanie dokumentacji. Wykorzystanie generatora Doxygen (rys. 5.3, s. 24) pokazało mi, jak ważne jest pisanie czytelnych komentarzy w kodzie. Również współpraca z asystentem GitHub Copilot była ciekawym doświadczeniem – narzędzie to świetnie radziło sobie z generowaniem powtarzalnego kodu testów, co pozwoliło mi skupić się na trudniejszej logice algorytmu.

Podsumowując, projekt ten pozwolił mi połączyć wiedzę teoretyczną o algorytmach z praktycznymi narzędziami, które są standardem w branży IT.

```
PS D:\cpp\programowanie-zaawansowane-3> .\testy.exe
[==========] 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.OnlyNegativeNumbers
[       OK ] MergeSortTest.OnlyNegativeNumbers (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.Duplicates
[       OK ] MergeSortTest.Duplicates (0 ms)
[ RUN      ] MergeSortTest.NegativeDuplicates
[       OK ] MergeSortTest.NegativeDuplicates (0 ms)
[ RUN      ] MergeSortTest.MixedWithDuplicates
[       OK ] MergeSortTest.MixedWithDuplicates (0 ms)
[ RUN      ] MergeSortTest.TwoElementsSorted
[       OK ] MergeSortTest.TwoElementsSorted (0 ms)
[ RUN      ] MergeSortTest.LargeArray
[       OK ] MergeSortTest.LargeArray (0 ms)
[ RUN      ] MergeSortTest.LargeArrayMixedComplex
[       OK ] MergeSortTest.LargeArrayMixedComplex (0 ms)
[----------] 13 tests from MergeSortTest (20 ms total)

[----------] Global test environment tear-down
[==========] 13 tests from 1 test suite ran. (24 ms total)
[  PASSED  ] 13 tests.
0
```

Rys. 5.2. Zaliczone testy jednostkowe w konsoli (13/13)

Bibliografia

[1] *Sortowanie przez scalanie (Merge Sort)*. URL: https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie (term. wiz. 27. 11. 2025).

[2] *Merge Sort – Data Structure and Algorithms Tutorials*. URL: https://www.geeksforgeeks.org/dsa/merge-sort/ (term. wiz. 27. 11. 2025).

Rys. 5.3. Fragment wygenerowanej dokumentacji HTML

[3] *GoogleTest User's Guide*. URL: https://google.github.io/googletest/ (term. wiz. 27. 11. 2025).

Spis rysunków

Spis tabel

Spis listingów