AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

PROGRAMOWANIE ZAAWANSOWANE

Implementacja działania sortowania przez scalanie i zaznajomienie się z podstawowym działaniem testów

Autor:

Rafał Grzegorzek

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2023

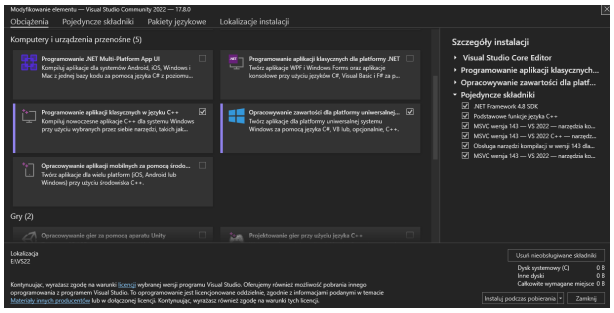
Spis treści

1. Ogólne określenie wymagań

Zadanie przewiduje napisanie algorytmu sortowania przez scalanie w języku C++. Następnie główną częścią zadania jest wykonanie odpowiednich testów i zaznajomienie się z ich funkcjami i sposobem działania. Postępy są zapisywane za pomocą technologii kontroli wersji GiT.

1.1. Przygotowania

Na samym początku zaleca się pobranie Visual Studio 2022, który znacznie ułatwi pracę na projektem. Kolejno wybieramy VS installer wybieramy funkcję, która umożliwi pisanie aplikacji konsolowych w języku C++ (rys.1.1).



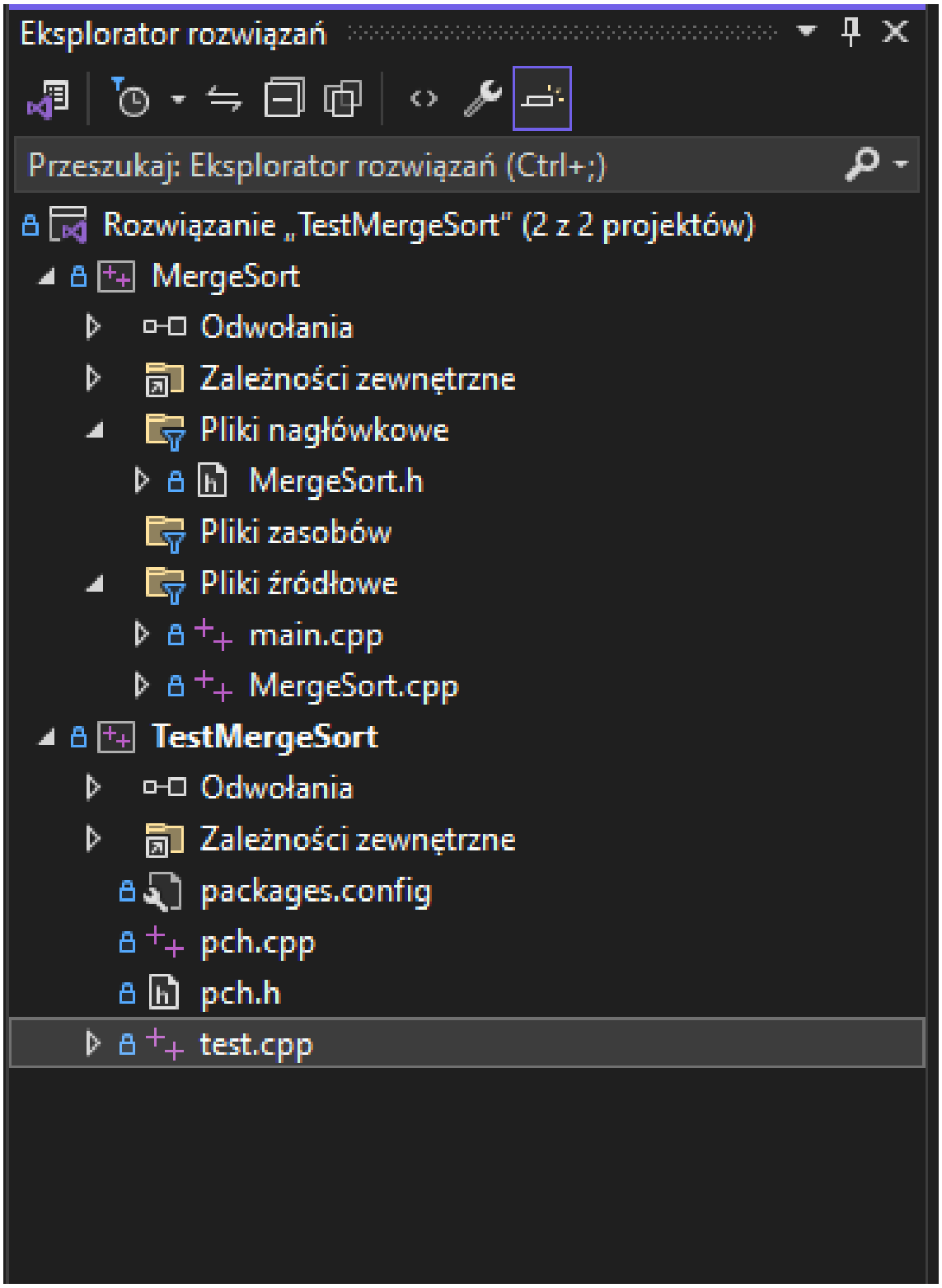
Rys. 1.1. Instalacja VS22

1.2. Google test

Visual studio 22 powinno zawierać gotowy, darmowy szablon o nazwie Google-Test. Tworząc nowe rozwiązanie wybieramy właśnie wspomnianą opcję.

Następnie możliwe jest zapoznanie się z samym programem testowym i sprawdzenie przykładowego działania. W dalszej kolejności można przystąpić do kolejnych kroków:

1. Połącz się ze swoim kontem Git. Może okazać się niezbędny w przypadku wystąpienia błędów.
2. Przygotuj potrzebny algorytm i rozwiązanie.
3. Dodaj kolejny projekt do już używanego rozwiązania i powiąż z plikiem testowym.
4. Wprowadź potrzebne fragmenty kody odpowiedzialne za testy.



Rys. 1.2. Wydląd plików w VS22

Z gotowym środowiskiem i przygotowanym plikiem testowym, możemy przejść do planowania kodu i rozwiązania problemu.

2. Przedstawienie problemu

Problem przedstawia się następująco:

Napisz program sortowanie przez scalanie (MergeSort). Algorytm musi być zaimplementowana w klasie. Funkcja main ma być zaimplementowana w osobnym pliku. Następnie za pomocą pakietu cpputest należy wykonać testy jednostkowe algorytmu. Należy pobrać z GitHuba framework do testów. Link do repozytorium to: https://github.com/cpputest/cpputest (wykonać fork). Testy muszą sprawdzić co najmniej następujące przypadki:

• Test sprawdza, czy algorytm zachowuje tablicę niezmienioną, gdy ona jest już posortowana rosnąco,

• Test sprawdza, czy algorytm potrafi posortować tablicę, która jest posortowana w odwrotnej kolejności,

• Test sprawdza, czy algorytm poprawnie sortuje losową tablicę liczb,

• Test sprawdza, czy algorytm poprawnie sortuje tablice tylko z liczbami ujemnymi,

• Test sprawdza, czy algorytm poprawnie sortuje tablice z liczbami ujemnymi i liczbami dodatnimi,

• Test sprawdza, czy algorytm obsługuje pustą tablicę bez rzucania wyjątkiem,

• Test sprawdza, czy algorytm nie zmienia tablicy, która zawiera tylko jeden element,

• Test sprawdza, czy algorytm poprawnie sortuje tablicę z duplikatami liczb,

• Test sprawdza, czy algorytm poprawnie sortuje tablice ujemną z duplikatami,

• Test sprawdza, czy algorytm poprawnie sortuje tablice z liczbami ujemnymi, dodatnimi oraz duplikatami,

• Test sprawdza, czy algorytm poprawnie sortuje tablicę zawierającą tylko dwa elementy w kolejności rosnącej,

• Test sprawdza, czy algorytm poprawnie sortuje dużą tablicę zawierającą ponad 100 elementów,

• Test sprawdza, czy algorytm poprawnie sortuje dużą tablicę zawierającą ponad 100 elementów z liczbami ujemnymi, dodatnimi oraz duplikatami,

Celem projektu jest zaznajomienie studenta z podstawowym rodzajem testów jakim są testy jednostkowe. Drugą ważną cechą tego projektu jest pobranie repozytorium innej osoby (pliki muszą być na licencji darmowej). Po pobraniu plików (fork) należy zapisać projekt na swoim koncie GitHuba, a następnie komitować projekt wraz ze swoimi zmianami.

3. Rozwiązanie i działanie

W tym rozdziale przedtsawione zostaną części kodu i ich działanie..

3.1. main.cpp

```
1
2  #include "MergeSort.h"
3  #include <iostream>
4  #include <vector>
5
6  int main() {
7    // Wczytaj liczby z klawiatury
8    std::cout << "Podaj liczby do posortowania (wpisz 0, aby przerwac
       ): ";
9    std::vector<int> lista_do_posortowania;
10   int liczba;
11
12   while (std::cin >> liczba && liczba != 0) {
13     lista_do_posortowania.push_back(liczba);
14   }
15
16   std::cout << "Lista przed posortowaniem:";
17   for (int liczba : lista_do_posortowania) {
18     std::cout << " " << liczba;
19   }
20   std::cout << std::endl;
21
22   MergeSort sorter;
23   sorter.sortuj(lista_do_posortowania, 0, lista_do_posortowania.
       size() - 1);
24
25   std::cout << "Lista po posortowaniu:";
26   for (int liczba : lista_do_posortowania) {
27     std::cout << " " << liczba;
28   }
29   std::cout << std::endl;
30
31   return 0;
32 }
```

```
33
```

Listing 1. Plik main.cpp

Ten kod (Listing 1) to program w języku C++, który implementuje interaktywny interfejs dla sortowania przez scalanie - Merge Sort. Działanie przedstawia się następująco:

1. Włączenie pliku nagłówkowego MergeSort.h.
2. Włączenie nagłówków iostream oraz vector.
3. Główna funkcja programu:
   (a) Wyświetlanie informacji dla użytkownika.
   (b) Inicjalizacja wektora i wczytywanie liczb z klawiatury.
   (c) Wyświetlanie listy przed posortowaniem.
   (d) Inicjalizacja obiektu ‘MergeSort‘.
   (e) Wywołanie funkcji sortującej na wektorze.
   (f) Wyświetlanie listy po posortowaniu.

Plik main służy inicjacji działania programu. W nim wywoływane są zdefiniowane w innych plikach funckje.

3.2. MergeSort.cpp

```
1  #include "MergeSort.h"
2
3  void MergeSort::sortuj(std::vector<int>& lista, int lewy, int
    prawy) {
4    if (lewy < prawy) {
5      int srodek = lewy + (prawy - lewy) / 2;
6      sortuj(lista, lewy, srodek);
7      sortuj(lista, srodek + 1, prawy);
8      scalaj(lista, lewy, srodek, prawy);
9    }
10 }
11
12 void MergeSort::scalaj(std::vector<int>& lista, int lewy, int
    srodek, int prawy) {
13   int n1 = srodek - lewy + 1;
```

```
14    int n2 = prawy - srodek;
15
16    std::vector<int> lewa_czesc(lista.begin() + lewy, lista.begin()
      + lewy + n1);
17    std::vector<int> prawa_czesc(lista.begin() + srodek + 1, lista.
    begin() + srodek + 1 + n2);
18
19    int i = 0, j = 0, k = lewy;
20
21    while (i < n1 && j < n2) {
22      if (lewa_czesc[i] <= prawa_czesc[j]) {
23        lista[k] = lewa_czesc[i];
24        i++;
25      }
26      else {
27        lista[k] = prawa_czesc[j];
28        j++;
29      }
30      k++;
31    }
32
33    while (i < n1) {
34      lista[k] = lewa_czesc[i];
35      i++;
36      k++;
37    }
38
39    while (j < n2) {
40      lista[k] = prawa_czesc[j];
41      j++;
42      k++;
43    }
44  }
45
```

Listing 2. Plik MergeSort.cpp

Informacje na temat listing 2.

1. Linie 3-9: - Deklaracja funkcji sortuj w klasie MergeSort.
2. Linie 11-13: - Sprawdzenie warunku: czy lewy indeks jest mniejszy od prawego indeksu.
3. Linie 14-16: - Obliczenie środka (indeks srodka) jako średnia między lewym i prawym indeksem.

4. Linie 17-19: - Rekurencyjne wywołanie funkcji sortuj dla lewej części wektora.

5. Linie 20-22: - Rekurencyjne wywołanie funkcji sortuj dla prawej części wektora.

6. Linie 23-25: - Wywołanie funkcji scalaj do połączenia (scalania) posortowanych części.

3.3. MergeSort.h

```
1  #ifndef MERGESORT_H
2  #define MERGESORT_H
3
4  #include <vector>
5
6  class MergeSort {
7    public:
8    void sortuj(std::vector<int>& lista, int lewy, int prawy);
9
10   private:
11   void scalaj(std::vector<int>& lista, int lewy, int srodek, int
     prawy);
12 };
13
14 #endif // MERGESORT_H
15
16
17
18
```

Listing 3. Plik MergeSort.h

Listing 3:

1. Linia 1: Sprawdza, czy identyfikator nie został wcześniej zdefiniowany (zapobiega wielokrotnemu dołączaniu pliku nagłówkowego).

2. Linia 2: Definiuje identyfikator, co unika konfliktów przy wielokrotnym dołączaniu pliku nagłówkowego.

3. Linia 4: Dołącza plik nagłówkowy vector, który umożliwia korzystanie z wektorów liczb całkowitych w programie.

4. Linie 6-8: Deklaruje klasę MergeSort, która zawiera metody sortuj i scalaj.

5. Linie 9-11: Oznacza sekcję publiczną klasy, gdzie znajduje się metoda sortuj, akceptująca wektor liczb całkowitych oraz indeksy lewego i prawego zakresu do posortowania.

6. Linie 13-15: Oznacza sekcję prywatną klasy, gdzie znajduje się metoda scalaj, akceptująca wektor liczb całkowitych oraz indeksy lewego, środkowego i prawego zakresu do scalenia.

3.4. test.cpp

Kolejne listingi (4-16) przedstawiają poszczególne działanie fragmentów w pliku test.cpp, odpowiedzialnego za testowanie pozostałych.

```
1  TEST(MergeSortTest, LeavesSortedArrayUnchanged) {
2
3    std::vector<int> input = { 1, 2, 3, 4, 5, 6 };
4    std::vector<int> expected_output = { 1, 2, 3, 4, 5, 6 };
5
6
7    MergeSort sorter;
8    sorter.sortuj(input, 0, input.size() - 1);
9
10
11   ASSERT_EQ(input, expected_output);
12 }
```

Listing 4. Czy zachowuje niezmienioną tablicę

```
1  TEST(MergeSortTest, SortsReversedArray) {
2
3    std::vector<int> input = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };
4    std::vector<int> expected_output = { 1, 2, 3, 4, 5, 6, 7, 8, 9
     };
5
6
7    MergeSort sorter;
8    sorter.sortuj(input, 0, input.size() - 1);
9
10
11   ASSERT_EQ(input, expected_output);
12 }
```

Listing 5. Czy posortuje odwróconą

```
1  TEST(MergeSortTest, SortsRandomArray) {
```

```
2
3     std::vector<int> input = { 7, 4, 2, 9, 3, 1 };
4     std::vector<int> expected_output = { 1, 2, 3, 4, 7, 9 };
5
6
7     MergeSort sorter;
8     sorter.sortuj(input, 0, input.size() - 1);
9
10
11    ASSERT_EQ(input, expected_output);
12  }
```

Listing 6. Czy posortuje losową

```
1   TEST(MergeSortTest, SortsNegativeNumbersArray) {
2     std::vector<int> input = { -4, -2, -7, -1, -9, -3 };
3     std::vector<int> expected_output = { -9, -7, -4, -3, -2, -1 };
4
5     MergeSort sorter;
6     sorter.sortuj(input, 0, input.size() - 1);
7
8     ASSERT_EQ(input, expected_output);
9   }
```

Listing 7. Czy posortuje ujemne

```
1   TEST(MergeSortTest, SortsMixedNumbersArray) {
2     std::vector<int> input = { -4, 2, -7, 1, 9, -3 };
3     std::vector<int> expected_output = { -7, -4, -3, 1, 2, 9 };
4
5     MergeSort sorter;
6     sorter.sortuj(input, 0, input.size() - 1);
7
8     ASSERT_EQ(input, expected_output);
9   }
```

Listing 8. Czy posortuje ujemne i dodatnie

```
1   TEST(MergeSortTest, HandlesEmptyArray) {
2     std::vector<int> input = {};
3     std::vector<int> expected_output = {};
4
5     MergeSort sorter;
6     sorter.sortuj(input, 0, input.size() - 1);
7
8     ASSERT_EQ(input, expected_output);
```

```
}
```

Listing 9. Test sprawdza, czy algorytm obsługuje pustą tablicę bez rzucania wyjątkiem

```
TEST(MergeSortTest, LeavesSingleElementArrayUnchanged) {
  std::vector<int> input = { 5 };
  std::vector<int> expected_output = { 5 };

  MergeSort sorter;
  sorter.sortuj(input, 0, input.size() - 1);

  ASSERT_EQ(input, expected_output);
}
```

Listing 10. Test sprawdza, czy algorytm nie zmienia tablicy, która zawiera tylko jeden element

```
TEST(MergeSortTest, SortsArrayWithDuplicates) {
  std::vector<int> input = { 4, 2, 7, 1, 9, 2, 3 };
  std::vector<int> expected_output = { 1, 2, 2, 3, 4, 7, 9 };

  MergeSort sorter;
  sorter.sortuj(input, 0, input.size() - 1);

  ASSERT_EQ(input, expected_output);
}
```

Listing 11. Test sprawdza, czy algorytm poprawnie sortuje tablicę z duplikatami liczb

```
TEST(MergeSortTest, SortsNegativeArrayWithDuplicates) {
  std::vector<int> input = { -4, -2, -7, -1, -9, -2, -3 };
  std::vector<int> expected_output = { -9, -7, -4, -3, -2, -2, -1
  };

  MergeSort sorter;
  sorter.sortuj(input, 0, input.size() - 1);

  ASSERT_EQ(input, expected_output);
}
```

Listing 12. Test sprawdza, czy algorytm poprawnie sortuje tablice ujemną z duplikatami

```
TEST(MergeSortTest, SortsMixedArrayWithDuplicates) {
  std::vector<int> input = { -4, 2, -7, 1, 2, 9, -3, 2 };
  std::vector<int> expected_output = { -7, -4, -3, 1, 2, 2, 2, 9
 };

```

```
    MergeSort sorter;
    sorter.sortuj(input, 0, input.size() - 1);

    ASSERT_EQ(input, expected_output);
}
```

Listing 13. Test sprawdza, czy algorytm poprawnie sortuje tablicę z liczbami ujemnymi, dodatnimi oraz duplikatami

```
TEST(MergeSortTest, SortsTwoElementArrayAscending) {
    std::vector<int> input = { 3, 1 };
    std::vector<int> expected_output = { 1, 3 };

    MergeSort sorter;
    sorter.sortuj(input, 0, input.size() - 1);

    ASSERT_EQ(input, expected_output);
}
```

Listing 14. Test sprawdza, czy algorytm poprawnie sortuje tablicę zawierającą tylko dwa elementy w kolejności rosnącej

```
TEST(MergeSortTest, SortsLargeArray) {

    const int arraySize = 105;
    std::vector<int> input = {
        // (elementy tablicy)
    };


    std::vector<int> expected_output = input;
    std::sort(expected_output.begin(), expected_output.end());

    MergeSort sorter;
    sorter.sortuj(input, 0, input.size() - 1);

    ASSERT_EQ(input, expected_output);
}
```

Listing 15. Test sprawdza, czy algorytm poprawnie sortuje dużą tablicę zawierającą ponad 100 elementów

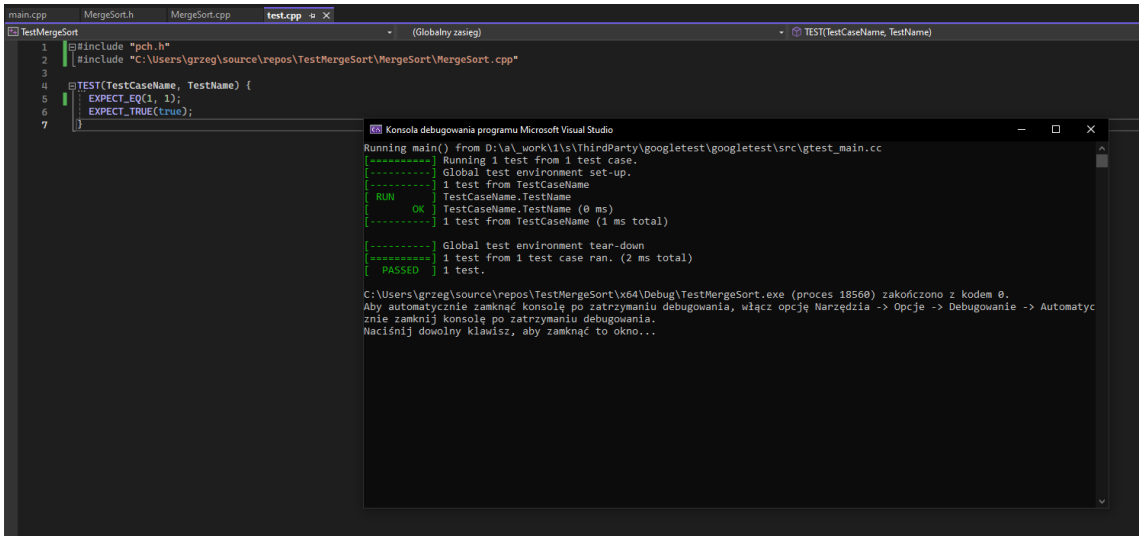
```
TEST(MergeSortTest, SortsLargeMixedArrayWithDuplicates) {

    const int arraySize = 105;
    std::vector<int> input = {
        // (elementy tablicy)
```

```
6       };
7
8       std::vector<int> expected_output = input;
9       std::sort(expected_output.begin(), expected_output.end());
10
11      MergeSort sorter;
12      sorter.sortuj(input, 0, input.size() - 1);
13
14      ASSERT_EQ(input, expected_output);
15  }
```

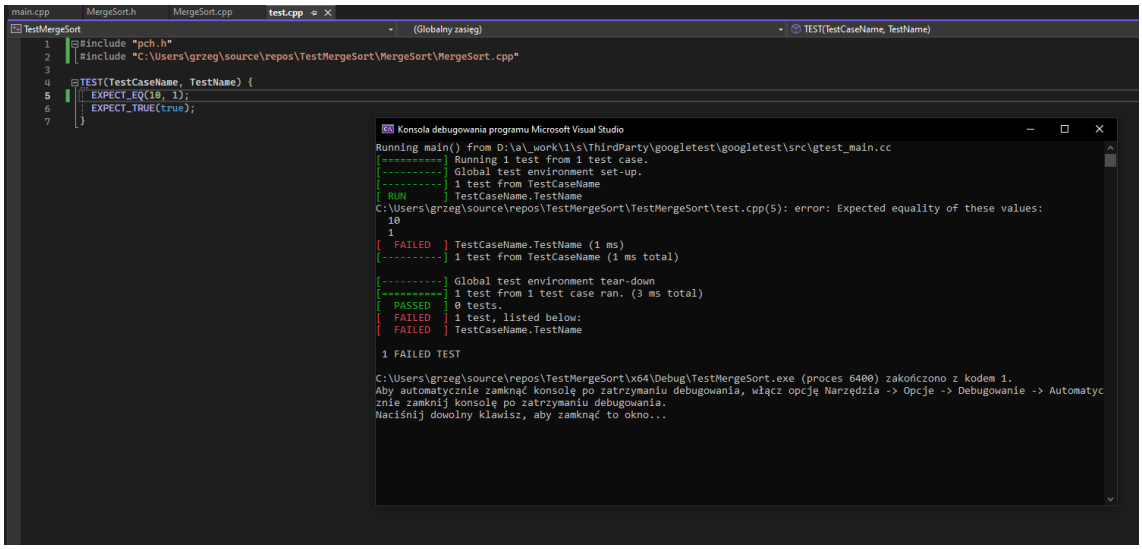
Listing 16. Test sprawdza, czy algorytm poprawnie sortuje dużą tablicę zawierającą ponad 100 elementów z liczbami ujemnymi, dodatnimi oraz duplikatami

4. Przeprowadzanie testu

Przykładowe działanie testów ukazane jest na obrazkach 4.1 i 4.2:



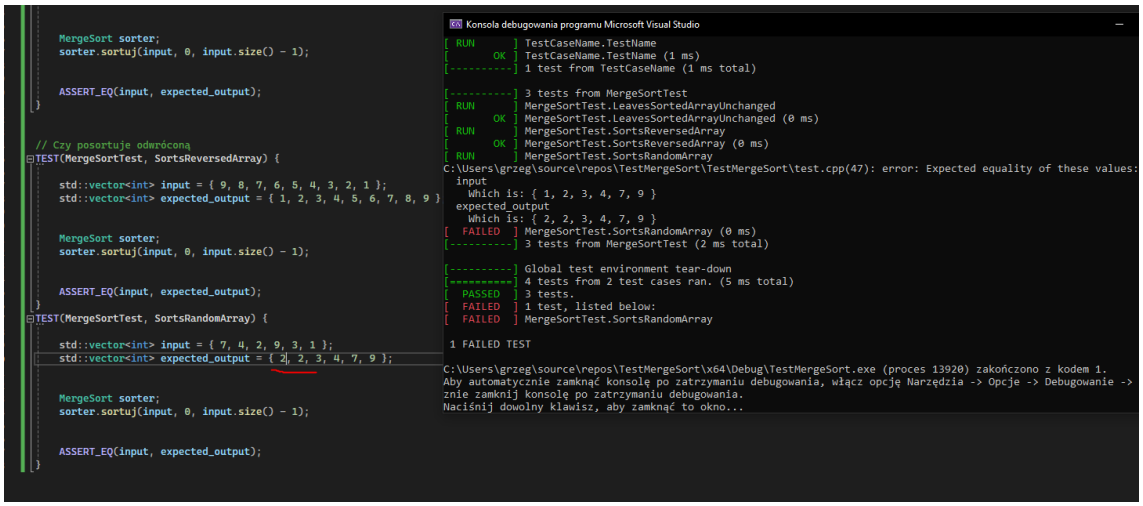
Rys. 4.1. Test zdany



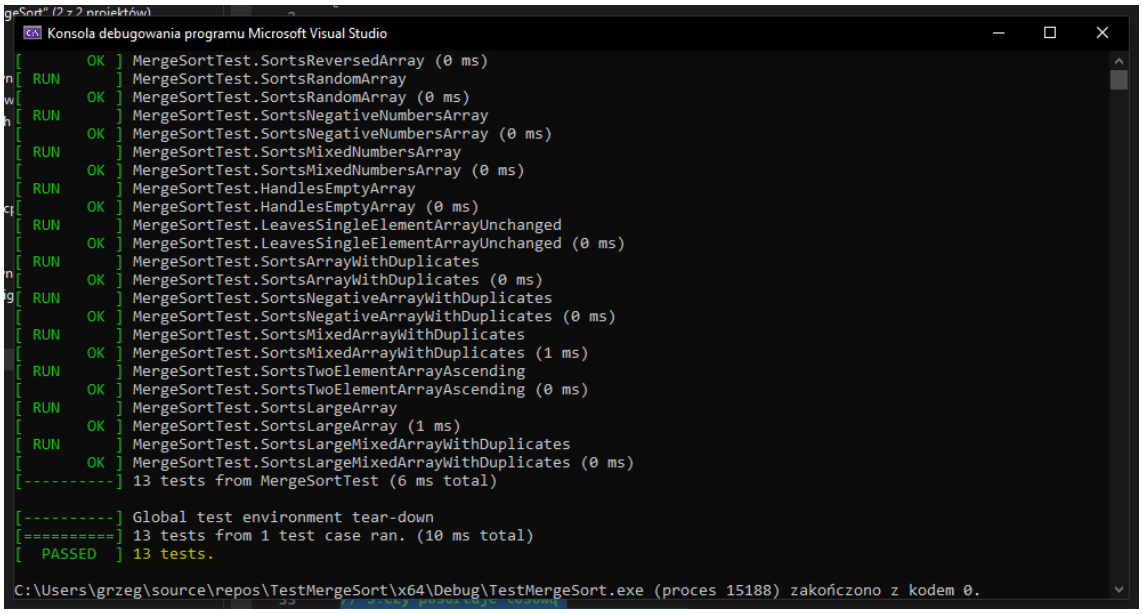
Rys. 4.2. Test niezdany

Widzimy, że program opisuje liczbę zdanych i niezdanych testów. Obrazek 4.3:

W przedstawionym wcześniej programie wszystkie 13 testów zakończyło się powodzeniem, co widać na obrazku 4.4:



Rys. 4.3. Szukanie drogi do elementu



Rys. 4.4. Wykonanie wszystkich testów

5. Podsumowanie i wnioski

Sortowanie przez scalanie jest używane do skutecznego i efektywnego sortowania dużych zestawów danych. Algorytm ten dzieli listę na mniejsze części, sortuje je osobno, a następnie scalając posortowane podlisty, tworzy pełną, posortowaną listę. Jest szczególnie przydatny przy sortowaniu dużych plików, w bazach danych oraz w aplikacjach wymagających stabilnego i wydajnego sortowania.

Testy w C++ (takie jak Google Test) pełnią kluczową rolę w procesie programowania, zapewniając szereg korzyści:

1. Zabezpieczają przed błędami: Testy automatyczne pomagają wykrywać błędy i problemy w kodzie źródłowym już na etapie jego tworzenia. Dzięki nim możliwe jest szybkie zidentyfikowanie i naprawienie potencjalnych problemów, co przekłada się na lepszą jakość kodu.

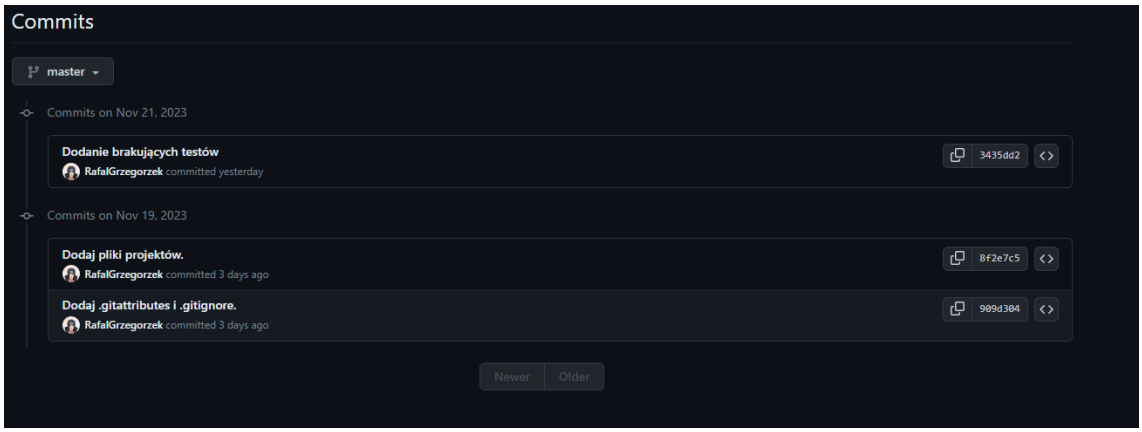
2. Utrzymywanie jakości kodu: Testy stanowią swoiste zabezpieczenie przed przypadkowym wprowadzeniem błędów podczas późniejszych modyfikacji kodu. Zapewniają, że nowe zmiany nie wpływają negatywnie na już istniejącą funkcjonalność.

3. Zwiększanie pewności działania kodu: Dzięki testom programista ma pewność, że istotne fragmenty kodu działają zgodnie z oczekiwaniami. To zwiększa zaufanie do kodu i ułatwia zarządzanie projektem.

4. Różne przypadki testowe: Testy pozwalają na sprawdzenie różnych przypadków użycia i sytuacji granicznych, co jest trudne do osiągnięcia tylko poprzez ręczne testowanie.

Projekt ten pomógł zrozumieć działanie i zastosowanie testów i uświadomił o ich użyteczności i funkcjonalności.

Cała praca została zapisana i zabezpieczona na platformie Git (rys. 5.1).



Rys. 5.1. Zapis plików - GitHUB

Spis rysunków

Spis tabel

Spis listingów