Merge Sort C++

1.0

Wygenerowano za pomocą Doxygen 1.15.0

Rozdział 1

Struktura katalogów

1.1 Katalogi

Rozdział 2

Indeks klas

2.1 Lista klas

Tutaj znajdują się klasy, struktury, unie i interfejsy wraz z ich krótkimi opisami:

Rozdział 3

Indeks plików

3.1 Lista plików

Tutaj znajduje się lista wszystkich plików wraz z ich krótkimi opisami:

Rozdział 4

Dokumentacja katalogów

4.1 Dokumentacja katalogu src

Pliki

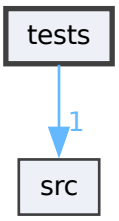
- plik main.cpp

  Plik główny demonstrujący działanie algorytmu Merge Sort.
- plik MergeSorter.h

  Implementacja szablonowa algorytmu sortowania przez scalanie (Merge Sort).

4.2 Dokumentacja katalogu tests

Wykres zależności katalogu dla tests:



Pliki

- plik test_mergesort.cpp

  Zestaw testów jednostkowych dla algorytmu Merge Sort.

Rozdział 5

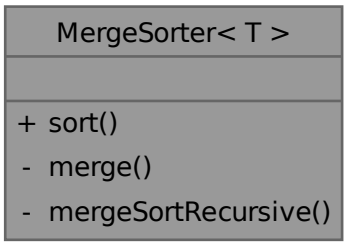
Dokumentacja klas

5.1 Dokumentacja szablonu klasy MergeSorter< T >

Klasa realizująca algorytm sortowania przez scalanie.

`#include <MergeSorter.h>`

Diagram współpracy dla MergeSorter< T >:



Metody publiczne

- void sort (vector< T > &array)

  Publiczna metoda do sortowania wektora.

Metody prywatne

- void merge (vector< T > &array, int left, int mid, int right)

  Scala dwie posortowane podtablice w jedną.
- void mergeSortRecursive (vector< T > &array, int left, int right)

  Rekurencyjna funkcja sortująca.

5.1.1 Opis szczegółowy

template<typename T>
class MergeSorter< T >

Klasa realizująca algorytm sortowania przez scalanie.

Klasa wykorzystuje strategię "dziel i zwyciężaj" (Divide and Conquer). Algorytm dzieli tablicę na połówki, sortuje je rekurencyjnie, a następnie scala.

Parametry Szablonu

T	Typ danych przechowywanych w wektorze. Musi posiadać operator porównania <=.

Definicja w linii 26 pliku MergeSorter.h.

5.1.2 Dokumentacja funkcji składowych

5.1.2.1 merge()

```
template<typename T>
void MergeSorter< T >::merge (
            vector< T > & array,
            int left,
            int mid,
            int right)  [inline], [private]
```

Scala dwie posortowane podtablice w jedną.

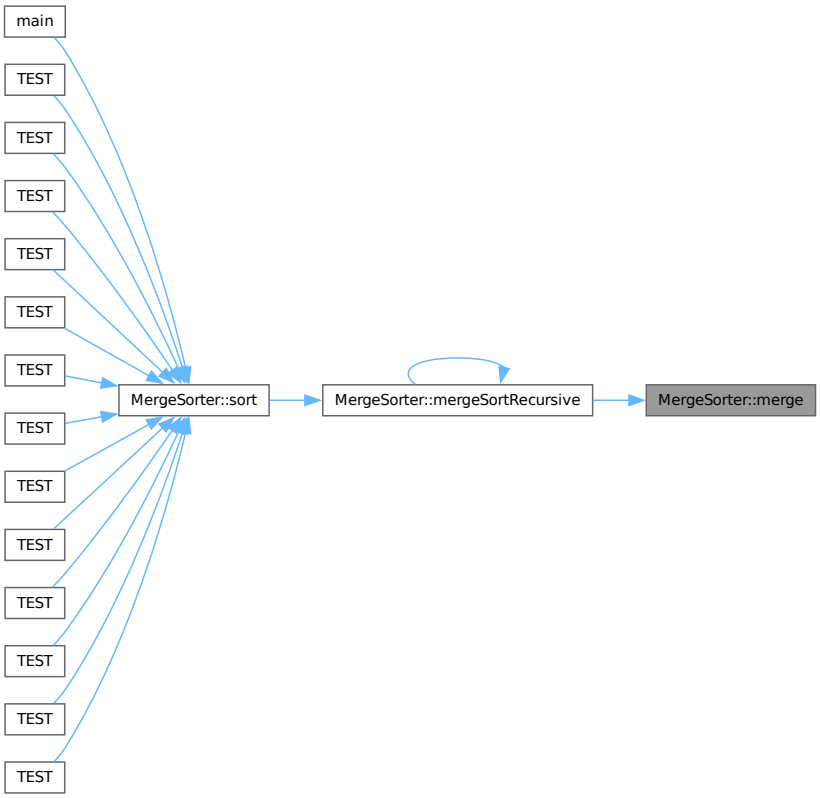
Funkcja pomocnicza, która łączy dwie podtablice tablicy[left..mid] oraz tablicy[mid+1..right] w jedną posortowaną sekwencję. Wykorzystuje dodatkową pamięć na tymczasowe tablice L i R.

Parametry

array	Referencja do wektora, na którym wykonywana jest operacja.
left	Indeks początkowy pierwszej podtablicy.
mid	Indeks środkowy (koniec pierwszej podtablicy).
right	Indeks końcowy drugiej podtablicy.

Definicja w linii 40 pliku MergeSorter.h.

Oto graf wywoływań tej funkcji:



5.1.2.2 mergeSortRecursive()

```
template<typename T>
void MergeSorter< T >::mergeSortRecursive (
            vector< T > & array,
            int left,
            int right)  [inline], [private]
```

Rekurencyjna funkcja sortująca.

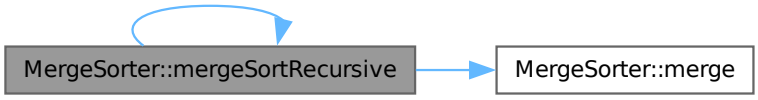
Dzieli tablicę na dwie połowy do momentu, gdy podtablica ma rozmiar 1, a następnie wywołuje funkcję merge() do scalenia posortowanych części.

Parametry

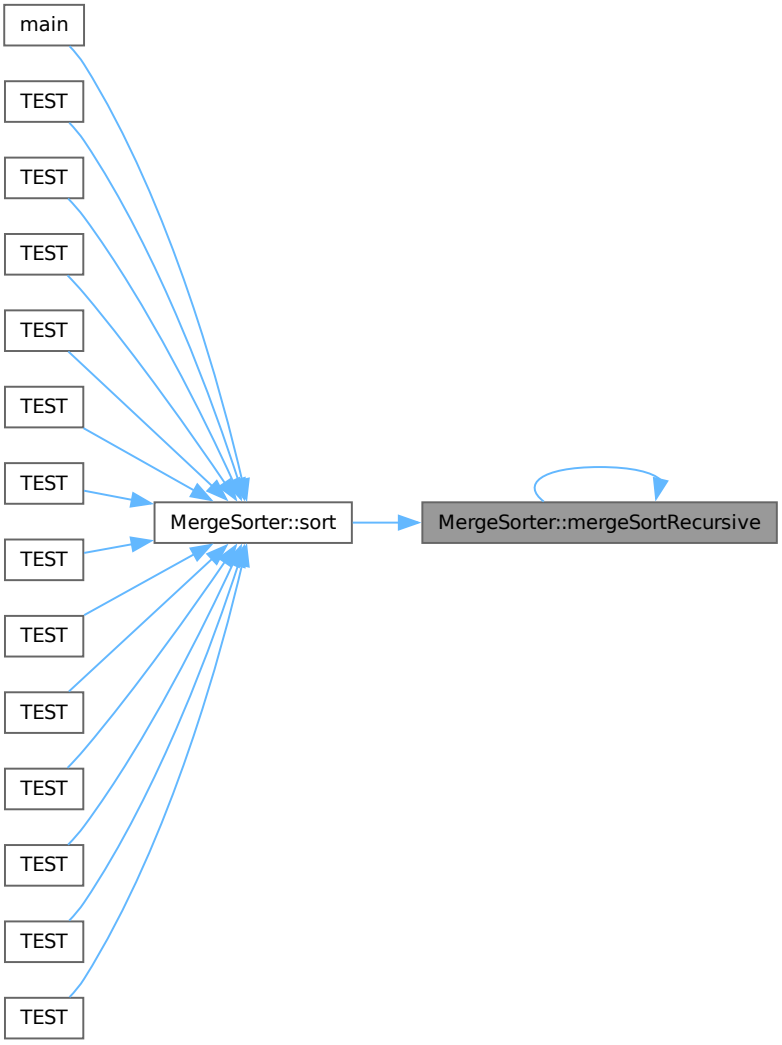
array	Referencja do sortowanego wektora.
left	Indeks lewego końca zakresu sortowania.
right	Indeks prawego końca zakresu sortowania.

Definicja w linii 95 pliku MergeSorter.h.

Oto graf wywołań dla tej funkcji:



Oto graf wywoływań tej funkcji:



5.1.2.3 sort()

```
template<typename T>
```

```
void MergeSorter< T >::sort (
            vector< T > & array)  [inline]
```

Publiczna metoda do sortowania wektora.

Jest to interfejs publiczny klasy. Sprawdza warunki brzegowe (pusta tablica lub jeden element) i uruchamia algorytm rekurencyjny.

Złożoność czasowa: $O(n \log n)$ we wszystkich przypadkach (średni, najgorszy, najlepszy). Złożoność pamięciowa: $O(n)$ (wymaga dodatkowej pamięci na scalanie).

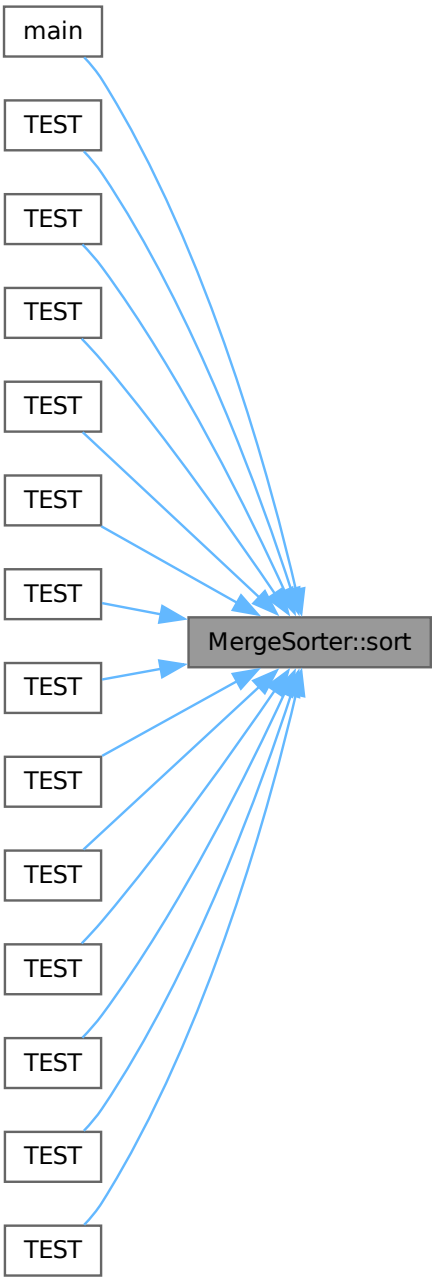
Parametry

array	Referencja do wektora, który ma zostać posortowany.

Definicja w linii 121 pliku MergeSorter.h.

Oto graf wywołań dla tej funkcji:

Oto graf wywoływań tej funkcji:



Dokumentacja dla tej klasy została wygenerowana z pliku:

- src/MergeSorter.h

Rozdział 6

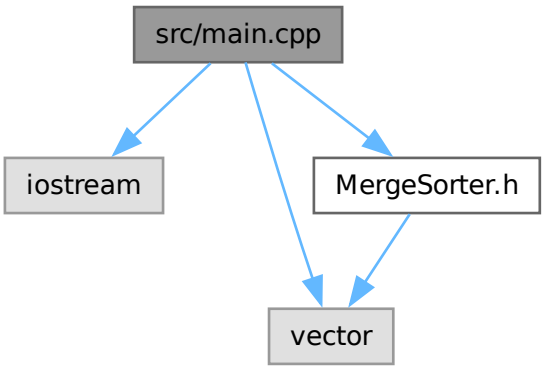
Dokumentacja plików

6.1 Dokumentacja pliku src/main.cpp

Plik główny demonstrujący działanie algorytmu Merge Sort.

```
#include <iostream>
#include <vector>
#include "MergeSorter.h"
```

Wykres zależności załączania dla main.cpp:



Funkcje

- template<typename T>
  void printArray (const vector< T > &arr)

  Wyświetla zawartość wektora na standardowym wyjściu.
- int main ()

  Główny punkt wejścia programu.

6.1.1 Opis szczegółowy

Plik główny demonstrujący działanie algorytmu Merge Sort.

Plik zawiera funkcję main, która tworzy instancje klasy MergeSorter dla różnych typów danych (int, double) i prezentuje wyniki sortowania na standardowym wyjściu.

Definicja w pliku main.cpp.

6.1.2 Dokumentacja funkcji

6.1.2.1 main()

```
int main ()
```

Główny punkt wejścia programu.

Funkcja przeprowadza demonstrację sortowania dla dwóch typów danych:

1. Liczb całkowitych (int).
2. Liczb zmiennoprzecinkowych (double).

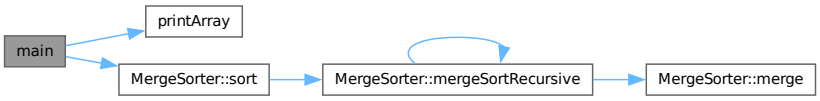
Dla każdego typu tworzony jest wektor, wyświetlany jest stan przed sortowaniem, następuje sortowanie, a na końcu wyświetlany jest wynik.

Zwraca

int Kod statusu zakończenia programu (0 oznacza sukces).

Definicja w linii 45 pliku main.cpp.

Oto graf wywołań dla tej funkcji:



6.1.2.2 printArray()

```
template<typename T>
void printArray (
            const vector< T > & arr)
```

Wyświetla zawartość wektora na standardowym wyjściu.

Funkcja pomocnicza iterująca przez elementy wektora i wypisująca je oddzielone spacją. Na końcu wypisywany jest znak nowej linii.

Parametry Szablonu

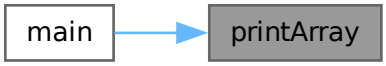
T	Typ danych przechowywanych w wektorze (musi obsługiwać operator<<).

Parametry

arr	Stała referencja do wektora, którego elementy mają zostać wyświetlone.

Definicja w linii 26 pliku main.cpp.

Oto graf wywoływań tej funkcji:



6.2 main.cpp

Idź do dokumentacji tego pliku.

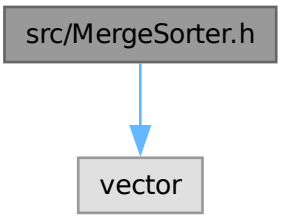
```
00001
00009
00010 #include <iostream>
00011 #include <vector>
00012 #include "MergeSorter.h"
00013
00014 using namespace std;
00015
00025 template <typename T>
00026 void printArray(const vector<T>& arr) {
00027     for (const auto& val : arr) {
00028         cout « val « " ";
00029     }
00030     cout « endl;
00031 }
00032
00045 int main() {
00046     // Demonstracja dla typu int
00047     vector<int> intTab = { 12, -5, 0, 7, 3, 12, -1, 49, 31 };
00048     MergeSorter<int> intSorter;
00049
00050     cout « "NO SORT | ";
00051     printArray(intTab);
00052
00053     intSorter.sort(intTab);
00054
00055     cout « "SORTED  | ";
00056     printArray(intTab);
00057
00058     // Demonstracja dla typu double
00059     vector<double> doubleTab = { 3.14, -1.1, 0.0, 2.5, 3.14, -0.5, -4.4, 5.96 };
00060     MergeSorter<double> doubleSorter;
00061
00062     cout « "NO SORT | ";
00063     printArray(doubleTab);
00064
00065     doubleSorter.sort(doubleTab);
00066
00067     cout « "SORTED  | ";
00068     printArray(doubleTab);
00069
00070     return 0;
00071 }
```

6.3 Dokumentacja pliku src/MergeSorter.h

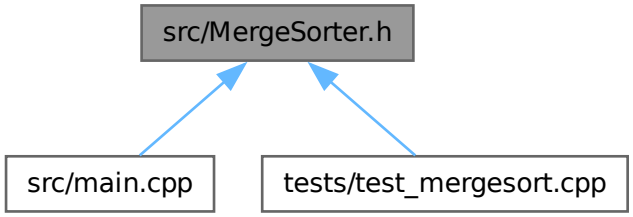
Implementacja szablonowa algorytmu sortowania przez scalanie (Merge Sort).

```
#include <vector>
```

Wykres zależności załączania dla MergeSorter.h:



Ten wykres pokazuje, które pliki bezpośrednio lub pośrednio załączają ten plik:



Komponenty

- class MergeSorter< T >

  Klasa realizująca algorytm sortowania przez scalanie.

6.3.1 Opis szczegółowy

Implementacja szablonowa algorytmu sortowania przez scalanie (Merge Sort).

Plik zawiera definicję klasy MergeSorter, która umożliwia sortowanie wektorów dowolnego typu, pod warunkiem, że typ ten obsługuje operatory porównania.

Definicja w pliku MergeSorter.h.

6.4 MergeSorter.h

Idź do dokumentacji tego pliku.

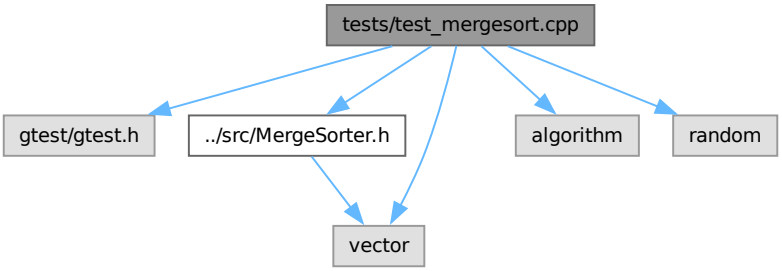
```
00001
00008
00009 #ifndef MERGESORTER_H
00010 #define MERGESORTER_H
00011
00012 #include <vector>
00013
00014 using namespace std;
00015
00025 template <typename T>
00026 class MergeSorter {
00027     private:
00040         void merge(vector<T>& array, int left, int mid, int right) {
00041             int n1 = mid - left + 1;
00042             int n2 = right - mid;
00043
00044             // Tworzenie tymczasowych wektorów
00045             vector<T> L(n1);
00046             vector<T> R(n2);
00047
00048             // Kopiowanie danych do tymczasowych wektorów L[] i R[]
00049             for (int i = 0; i < n1; i++)
00050                 L[i] = array[left + i];
00051             for (int j = 0; j < n2; j++)
00052                 R[j] = array[mid + 1 + j];
00053
00054             int i = 0; // Indeks początkowy pierwszej podtablicy
00055             int j = 0; // Indeks początkowy drugiej podtablicy
00056             int k = left; // Indeks początkowy scalonej tablicy
00057
00058             // Scalanie tymczasowych tablic z powrotem do array[left..right]
00059             while (i < n1 && j < n2) {
00060                 if (L[i] <= R[j]) {
00061                     array[k] = L[i];
00062                     i++;
00063                 } else {
00064                     array[k] = R[j];
00065                     j++;
00066                 }
00067                 k++;
00068             }
00069
00070             // Kopiowanie pozostałych elementów L[], jeśli są
00071             while (i < n1) {
00072                 array[k] = L[i];
00073                 i++;
00074                 k++;
00075             }
00076
00077             // Kopiowanie pozostałych elementów R[], jeśli są
00078             while (j < n2) {
00079                 array[k] = R[j];
00080                 j++;
00081                 k++;
00082             }
00083         }
00084
00095         void mergeSortRecursive(vector<T>& array, int left, int right) {
00096             if (left < right) {
00097                 // Obliczanie środka, chroni przed przepełnieniem (overflow) dla dużych int
00098                 int mid = left + (right - left) / 2;
00099
00100                 // Sortuj pierwszą i drugą połowę
00101                 mergeSortRecursive(array, left, mid);
00102                 mergeSortRecursive(array, mid + 1, right);
00103
00104                 // Scal posortowane połowy
00105                 merge(array, left, mid, right);
00106             }
00107         }
00108
00109     public:
00121         void sort(vector<T>& array) {
00122             if (array.size() <= 1) {
00123                 return; // Tablica jest już posortowana
00124             }
00125             mergeSortRecursive(array, 0, array.size() - 1);
00126         }
00127 };
00128
00129 #endif
```

6.5 Dokumentacja pliku tests/test_mergesort.cpp

Zestaw testów jednostkowych dla algorytmu Merge Sort.

```
#include <gtest/gtest.h>
#include "../src/MergeSorter.h"
#include <vector>
#include <algorithm>
#include <random>
```

Wykres zależności załączania dla test_mergesort.cpp:



Funkcje

- TEST (MergeSortTest, AlreadySorted)

  Sprawdza zachowanie dla tablicy już posortowanej.
- TEST (MergeSortTest, ReverseSorted)

  Sprawdza sortowanie tablicy posortowanej odwrotnie (malejąco).
- TEST (MergeSortTest, RandomArray)

  Testuje sortowanie losowej tablicy liczb.
- TEST (MergeSortTest, OnlyNegative)

  Weryfikuje sortowanie tablicy zawierającej tylko liczby ujemne.
- TEST (MergeSortTest, MixedNegativePositive)

  Sprawdza sortowanie mieszanki liczb dodatnich i ujemnych.
- TEST (MergeSortTest, EmptyArray)

  Testuje obsługę pustej tablicy.
- TEST (MergeSortTest, SingleElement)

  Sprawdza przypadek brzegowy z jednym elementem.
- TEST (MergeSortTest, Duplicates)

  Testuje poprawność sortowania tablicy z duplikatami.
- TEST (MergeSortTest, NegativeDuplicates)

  Sprawdza sortowanie duplikatów wśród liczb ujemnych.
- TEST (MergeSortTest, MixedDuplicates)

  Testuje mieszane wartości (dodatnie, ujemne, zero) z duplikatami.
- TEST (MergeSortTest, TwoElementsSorted)

  Sprawdza poprawność dla małej tablicy (2 elementy).
- TEST (MergeSortTest, LargeArray)

  Test wydajnościowy/poprawności dla dużej tablicy losowej.
- TEST (MergeSortTest, LargeComplexArray)

  Zaawansowany test dużej tablicy (ujemne, dodatnie, duplikaty).

6.5.1 Opis szczegółowy

Zestaw testów jednostkowych dla algorytmu Merge Sort.

Plik wykorzystuje framework Google Test do weryfikacji poprawności działania szablonu klasy MergeSorter. Testy pokrywają przypadki brzegowe, typowe scenariusze oraz testy obciążeniowe.

Definicja w pliku test_mergesort.cpp.

6.5.2 Dokumentacja funkcji

6.5.2.1 TEST() [1/13]

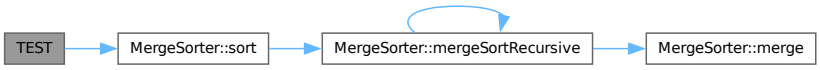
```
TEST (
            MergeSortTest ,
            AlreadySorted )
```

Sprawdza zachowanie dla tablicy już posortowanej.

Oczekiwane zachowanie: Tablica pozostaje niezmieniona.

Definicja w linii 23 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.2 TEST() [2/13]

```
TEST (
            MergeSortTest ,
            Duplicates )
```

Testuje poprawność sortowania tablicy z duplikatami.

Algorytm powinien zgrupować te same wartości obok siebie.

Definicja w linii 114 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:

6.5.2.3 TEST() [3/13]

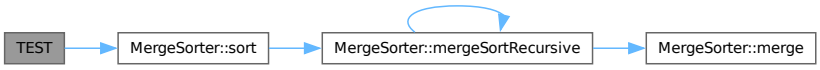
```
TEST (
            MergeSortTest ,
            EmptyArray )
```

Testuje obsługę pustej tablicy.

Algorytm nie powinien rzucić wyjątkiem ani naruszyć pamięci. Tablica po operacji powinna pozostać pusta.

Definicja w linii 89 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.4 TEST() [4/13]

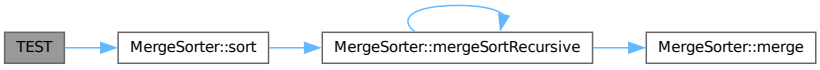
```
TEST (
            MergeSortTest ,
            LargeArray )
```

Test wydajnościowy/poprawności dla dużej tablicy losowej.

Tablica zawiera 150 losowych elementów generowanych funkcją rand().

Definicja w linii 164 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.5 TEST() [5/13]

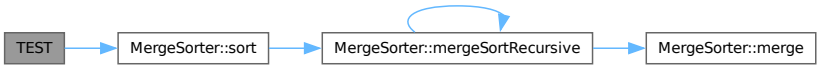
```
TEST (
            MergeSortTest ,
            LargeComplexArray )
```

Zaawansowany test dużej tablicy (ujemne, dodatnie, duplikaty).

Generuje ręcznie zestaw danych zawierający wszystkie typy liczb, a następnie miesza je losowo przed sortowaniem.

Definicja w linii 181 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.6 TEST() [6/13]

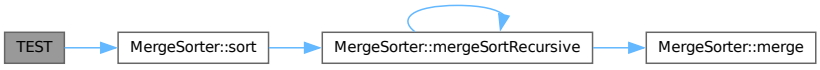
```
TEST (
            MergeSortTest ,
            MixedDuplicates )
```

Testuje mieszane wartości (dodatnie, ujemne, zero) z duplikatami.

Najbardziej złożony przypadek dla małej liczby danych.

Definicja w linii 138 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.7 TEST() [7/13]

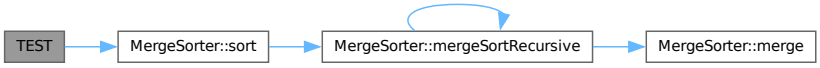
```
TEST (
            MergeSortTest ,
            MixedNegativePosition )
```

Sprawdza sortowanie mieszanki liczb dodatnich i ujemnych.

Testuje poprawne ustawienie liczb ujemnych przed zerem i liczbami dodatnimi.

Definicja w linii 75 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



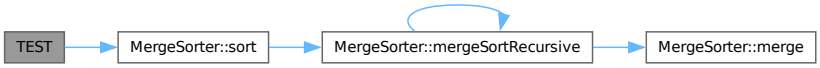
6.5.2.8 TEST() [8/13]

```
TEST (
            MergeSortTest ,
            NegativeDuplicates )
```

Sprawdza sortowanie duplikatów wśród liczb ujemnych.

Definicja w linii 125 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.9 TEST() [9/13]

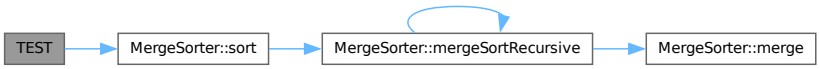
```
TEST (
            MergeSortTest ,
            OnlyNegative )
```

Weryfikuje sortowanie tablicy zawierającej tylko liczby ujemne.

Sprawdza, czy algorytm poprawnie interpretuje relacje mniejszości dla liczb ujemnych.

Definicja w linii 62 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.10 TEST() [10/13]

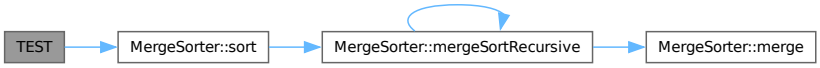
```
TEST (
            MergeSortTest ,
            RandomArray )
```

Testuje sortowanie losowej tablicy liczb.

Weryfikuje ogólną poprawność algorytmu na nieuporządkowanych danych. Wykorzystuje std::is_sorted do weryfikacji wyniku.

Definicja w linii 50 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.11 TEST() [11/13]

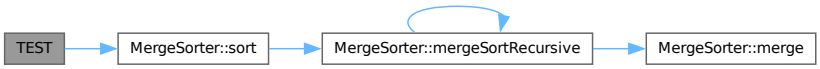
```
TEST (
            MergeSortTest ,
            ReverseSorted )
```

Sprawdza sortowanie tablicy posortowanej odwrotnie (malejąco).

Jest to jeden z przypadków, w których algorytm musi wykonać najwięcej przestawień (w scalaniu).

Definicja w linii 36 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.12 TEST() [12/13]

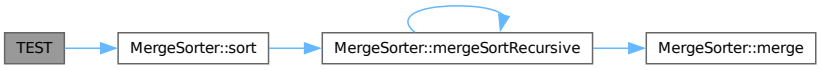
```
TEST (
            MergeSortTest ,
            SingleElement )
```

Sprawdza przypadek brzegowy z jednym elementem.

Tablica jednoelementowa jest z definicji posortowana. Algorytm nie powinien jej zmieniać.

Definicja w linii 101 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.5.2.13 TEST() [13/13]

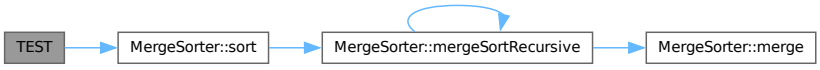
```
TEST (
            MergeSortTest ,
            TwoElementsSorted )
```

Sprawdza poprawność dla małej tablicy (2 elementy).

Podstawowy krok rekurencji (po podziale tablicy 4-elementowej).

Definicja w linii 151 pliku test_mergesort.cpp.

Oto graf wywołań dla tej funkcji:



6.6 test_mergesort.cpp

Idź do dokumentacji tego pliku.

```
00001
00009
00010 #include <gtest/gtest.h>
00011 #include "../src/MergeSorter.h"
00012 #include <vector>
00013 #include <algorithm>
00014 #include <random>
00015
00016 using namespace std;
00017
00023 TEST(MergeSortTest, AlreadySorted) {
00024     vector<int> arr = {1, 2, 3, 4, 5};
00025     vector<int> expected = {1, 2, 3, 4, 5};
00026     MergeSorter<int> sorter;
00027     sorter.sort(arr);
00028     EXPECT_EQ(arr, expected);
00029 }
00030
00036 TEST(MergeSortTest, ReverseSorted) {
00037     vector<int> arr = {5, 4, 3, 2, 1};
00038     vector<int> expected = {1, 2, 3, 4, 5};
00039     MergeSorter<int> sorter;
00040     sorter.sort(arr);
00041     EXPECT_EQ(arr, expected);
00042 }
00043
00050 TEST(MergeSortTest, RandomArray) {
00051     vector<int> arr = {10, 2, 8, 1, 5, 9};
00052     MergeSorter<int> sorter;
00053     sorter.sort(arr);
00054     EXPECT_TRUE(is_sorted(arr.begin(), arr.end()));
00055 }
00056
00062 TEST(MergeSortTest, OnlyNegative) {
00063     vector<int> arr = {-5, -1, -10, -3, -2};
00064     vector<int> expected = {-10, -5, -3, -2, -1};
00065     MergeSorter<int> sorter;
00066     sorter.sort(arr);
00067     EXPECT_EQ(arr, expected);
00068 }
00069
00075 TEST(MergeSortTest, MixedNegativePositive) {
00076     vector<int> arr = {-5, 10, 0, -3, 2};
00077     vector<int> expected = {-5, -3, 0, 2, 10};
00078     MergeSorter<int> sorter;
00079     sorter.sort(arr);
00080     EXPECT_EQ(arr, expected);
00081 }
00082
00089 TEST(MergeSortTest, EmptyArray) {
00090     vector<int> arr = {};
00091     MergeSorter<int> sorter;
00092     EXPECT_NO_THROW(sorter.sort(arr));
00093     EXPECT_TRUE(arr.empty());
00094 }
00095
00101 TEST(MergeSortTest, SingleElement) {
00102     vector<int> arr = {42};
00103     vector<int> expected = {42};
00104     MergeSorter<int> sorter;
00105     sorter.sort(arr);
00106     EXPECT_EQ(arr, expected);
00107 }
00108
00114 TEST(MergeSortTest, Duplicates) {
00115     vector<int> arr = {3, 1, 2, 3, 1};
00116     vector<int> expected = {1, 1, 2, 3, 3};
00117     MergeSorter<int> sorter;
00118     sorter.sort(arr);
00119     EXPECT_EQ(arr, expected);
00120 }
00121
00125 TEST(MergeSortTest, NegativeDuplicates) {
00126     vector<int> arr = {-3, -1, -2, -3, -1};
00127     vector<int> expected = {-3, -3, -2, -1, -1};
00128     MergeSorter<int> sorter;
00129     sorter.sort(arr);
00130     EXPECT_EQ(arr, expected);
00131 }
00132
00138 TEST(MergeSortTest, MixedDuplicates) {
00139     vector<int> arr = {-2, 3, 0, -2, 3, 1};
```

```
00140     vector<int> expected = {-2, -2, 0, 1, 3, 3};
00141     MergeSorter<int> sorter;
00142     sorter.sort(arr);
00143     EXPECT_EQ(arr, expected);
00144 }
00145
00151 TEST(MergeSortTest, TwoElementsSorted) {
00152     vector<int> arr = {1, 2};
00153     vector<int> expected = {1, 2};
00154     MergeSorter<int> sorter;
00155     sorter.sort(arr);
00156     EXPECT_EQ(arr, expected);
00157 }
00158
00164 TEST(MergeSortTest, LargeArray) {
00165     vector<int> arr(150);
00166     generate(arr.begin(), arr.end(), rand);
00167
00168     MergeSorter<int> sorter;
00169     sorter.sort(arr);
00170
00171     EXPECT_EQ(arr.size(), 150);
00172     EXPECT_TRUE(is_sorted(arr.begin(), arr.end()));
00173 }
00174
00181 TEST(MergeSortTest, LargeComplexArray) {
00182     vector<int> arr;
00183     for(int i=0; i<50; ++i) arr.push_back(i);       // Dodatnie
00184     for(int i=0; i<50; ++i) arr.push_back(-i);      // Ujemne
00185     for(int i=0; i<20; ++i) arr.push_back(5);       // Duplikaty dodatnie
00186     for(int i=0; i<20; ++i) arr.push_back(-5);      // Duplikaty ujemne
00187
00188     random_device rd;
00189     mt19937 g(rd());
00190     shuffle(arr.begin(), arr.end(), g);
00191
00192     MergeSorter<int> sorter;
00193     sorter.sort(arr);
00194
00195     EXPECT_GE(arr.size(), 140);
00196     EXPECT_TRUE(is_sorted(arr.begin(), arr.end()));
00197 }
```

Skorowidz