AiSD Laboratorium - Implementacja MergeSort

W tym dokumencie zaimplementujemy algorytm `MergeSort` oraz przetestujemy jego działanie. Dodatkowo,

- podstawowa wersja algorytmu zostanie zaimplementowana w taki sposób, aby mogłabyć użyta do sortowania `std::vector<unsigned short>`

Uwagi organizacyjne:

Zadanie podzielimy na kroki. Ostateczna wersja otrzymana w każdym kroku znajduje się w osobnym folderze `Step_i`.

Zadanie zrealizujemy w oparciu o pliki:

- `AiSD_MergeSort_demo.cpp` -- driver/plik demo, będzie służył do testowania i demonstracji; ten plik w każdym kroku jest inny, zwykle nie zawiera pomocniczych funkcji oraz kodu z zadania poprzedniego, w większości wypadków zalecam nie używać tego samego pliku w kolejnych zadaniach
- parę plików `mergesort.h/.cpp` -- zawierające deklaracje i definicje niezbędne do zrealizowania zadania; pliki te są dziedziczone pomiędzy krokami w tym sensie, że ostatecznia wersja z poprzedniego kroku jest punktem wyjścia do następnego. W większości przypadków, pliki zostały oczyszczone ze zbędnych komentarzy.
- parę plików `utils.h/.cpp` -- które zawierają funkcje pomocnicze, kßóre pochodzą z porpzedniego zadania; pliki te nie będa zmieniane, są konieczne w każdym kroku.
- Najlepszą metodą jest pisanie kodu samodzielnie.
- Dobrą metodą jest pisanie równolegle z tekstem, bez podglądania gotowego kodu, tak długo jak nie pojawią się problemy. Wtedy wręcz zajrzeć do odpowiednich plików.
- Niezłą metodą jest praca na załączonym kodzie, pamiętając że jest to wersja ostateczna, nie uwzględniająca zmian i uwag, które które pojawiły sie w trakcie.
- Najgorsza metodą jest poddanie się.

Zastrzeżenie o ograniczonej poprawności: Autor starał się, aby wszystko było poprawnie, zgodnie z dobrym stylem i działało. Czasami jednak mógł iść na skróty, zrezygnować z jakiegoś rozwiązania na rzecz prostoty lub wykazać się elementarną niewiedzą i ignorancją. W każdym takim przypadku zamiast narzekać lepiej zapytać. Literówki i błędy ort./gram. są winą Autora i jest mu wstyd.

Krok 0. Zapoznanie z szablonem

Zadanie 1. W folderze `Step_0` znajduje się szkielet projektu. Stwórz projekt zawierajacy pliki z folderu `Step_0`.

W plikach `utils.h/.cpp` znajdują się:

- alias `using T = unsigned short`
- funkcje `print` oraz `print_pretty`
- przeciążony oprator `<<`
- funkcja `is_sorted`
- klasa `RecordFactory`, która udostępnia funkcje `MakeRange` oraz `MakeRandom`

Wszystkie wymienione funckje działają tak jak w ostatecznej wersji z porpzedniego zadania. Niewielkie zmiany w ich implementacji w żaden sposób nie wpływają na sposób ich używania.

Dodatkowo, wszystkie własne funkcje znajdują się w przestrzeni nazw `AiSD`.

W dalszej częsci bedziemy się do niech odwoływać, ale jeżeli chcesz możesz użyć własnych funkcji.

Dla przypomnienia, w tej wersji pliku demo, podane są przykądy użycia każdej funkcji.

Zadanie 2. Skompiluj i uruchom program, aby upewnić się, że wszystkio działa prawidłowo.

Krok 1.

Zaimplementujemy następującą wersję algorytmu `MergeSort`:

```
MergeSort(A,first,last)
    if  first < last
        mid = (first+last)/2
        MergeSort(A,first,mid)
        MergeSort(A,mid+1,last)
        Merge(A,first,mid,last)
```

```
Merge(A,first,mid,last)
    iL = first,
    iR = mid + 1,
    k=1
    utwórz pomocniczą tablicę tmp rozmiaru tmp.length = last-first
    while iL <= mid and iR <= last                          //dopóki lewa i prawa podtablica zawiera elementy
    if A[iL] < A[iR]
        tmp[k] = A[iL]
        iL++, k++
    else
        tmp[k] = A[iR]
        iR++, k++

przekopiuj pozostałe elementy do tmp                       // jedna podtablica zawiera nieprzekopiowane elementy
przekopiuj zawartość tmp do A na odpowiednie miejsce
```

W powyższym pseudokodzie

- `A` oznacza tablicę do posortowania
- `first`, `last`, `mid` -- oznaczają indeksy

Zauważmy, że w żaden sposób nie jest powiedziana jak dokładnie jest reprezentowana tablica `A`. Widzimy jednak, że jedyne odwołania są realizowane przez operator `[]` i podanie indeksu. Do wyboru mamy tablicę w stylu C albo typ biblioteczny `std::vector`.

Decyzja: Chcemy, aby nasza implementacja mogłabyć użyta do sortowania `std::vector<T>`

Podjęta decyzja pozwala nam określić jak bedą wyglądać deklaracje funkcji `MergeSort` oraz `Merge`.

```
void MergeSort(std::vector<T>& A, size_t first, size_t last);

void Merge(std::vector<T>& A, size_t first,size_t mid, size_t last);
```

Zwróć uwagę, że pierwszy argument, tzn. tablica do posortowania,

- nie jest przekazywany jako stała bo chcemy modyfikować tablicę
- jest przekazywany przez referencję, gdyż chcemy aby modyfikacje nie były wykonywane na kopii

Zadanie: dodaj deklaracje do pliku `mergesort.h` oraz puste definicje do pliku `mergesort.cpp`. Sprawdź czy wszystko się poprawnie kompiluje.

Patrz folder `Step_1`

Krok 2.

Aby ułatwić sobie ewentualne szukanie błędów i lepiej zrozumiec jak przebiega proces sortowania, zdefiniujmy sobie dwie funkcje śledzące.

```
// w pliku mergesort.cpp

void trace_MergeSort(std::vector<T>& A, size_t first, size_t last){

    std::cout << "\nMergeSort: (first,last)  = (" << first << ", " << last
              << ")\narray: ";
    for(size_t i = first; i <= last; i++)
        std::cout << A[i] << " ";
    std::cout << '\n';

}

void trace_Merge(std::vector<T>& A, size_t first,size_t mid, size_t last){

    std::cout << "\n\tMerge: (first,mid,last)  = (" << first << ", " << mid << ", " << last << ")";

    std::cout << "\n\tleft  array: ";

    for(size_t i = first; i <= mid; i++)
        std::cout << A[i] << " ";
    std::cout << '\n';

    std::cout << "\tright array: ";
    for(size_t i = mid+1; i <= last; i++)
        std::cout << A[i] << " ";
    std::cout << '\n';

}
```

Zadanie: Co dokładnie robią te funkcje?

Funkcje śledzące wykorzystamy tak:

```
void MergeSort(std::vector<T>& A, size_t first, size_t last){

trace_MergeSort(A,first,last);
    // implementation
}

void Merge(std::vector<T>& A, size_t first,size_t mid, size_t last){

trace_Merge(A,first,mid,last);
    // implementation
}
```

Zadanie: Przetestuj `MergeSort` oraz `Merge` dla różnych tablic, z różnymi parametrami. Jakie są prawidłowe wartości `first`, `mid` oraz `last`?

Zadanie: W funkcji `MergeSort` będziemy uzywać `mid = (first+last)/2`. Przetestuj `Merge` dla róznych tablic, z róznymi parametrami `first` oraz `last` wg schematu:

```
    first = ...
    last = ...
    mid = (first+last)/2;
    Merge(A,first,mid,last);
```

Patrz folder `Step_2`

Krok 3.

Zadanie: Uzupełnij implementację funkcji `MergeSort` wg podanych niżej kroków. Do nastepnego kroku przejdź dopiero gdy zrozumiesz efekt. Pomocne jest wykonanie kilku rysunków. Zacznij od tablicy rozmiaru 5, potem sprawdź rozmiary 4 i 6.

- dodaj wyliczanie `mid` i wywołanie `Merge`

```
void MergeSort(std::vector<T>& A, size_t first, size_t last){

trace_MergeSort(A,first,last);

    if( first < last  ){
        size_t mid = (first + last)/2;

        Merge(A,first,mid,last);
    }
}
```

- dodaj wyliczanie `mid` i wywołanie `MergeSort` dla lewej tablicy

```
void MergeSort(std::vector<T>& A, size_t first, size_t last){

trace_MergeSort(A,first,last);

    if( first < last  ){
        size_t mid = (first + last)/2;

        MergeSort(A,first,mid);
    }
}
```

- dodaj wyliczanie `mid` i wywołanie `MergeSort` dla lewej prawej

```
void MergeSort(std::vector<T>& A, size_t first, size_t last){

trace_MergeSort(A,first,last);

    if( first < last  ){
        size_t mid = (first + last)/2;

        MergeSort(A,mid+1,last);

    }
}
```

- połącz powyższe kroki w pary
- użyj pełnej implementacji

Patrz folder `Step_ 3`.

Krok 4.

Zanim przejdziemy do implementacji `Merge` dokonamy pewnej modyfikacji.

Sposób w jaki podglądamy sobie działenie każdej z funkcji jest wygodny do śledzenia w jaki sposób wykonywane są wywołania podczs nauki i testowania, ale staje się zbędny po akończeniu tego etapu. Możliwości są dwie:

- (zły) przygotować dwie wersje funkcji
- użyć mechanizmu kompilacji warunkowej

- usunąć recznie wszystkie dodatki, których uzywalismy na etapie testowania -- tak powinno się zrobić w kodzie produkcyjnym

My zastosujemy drugi sposób. Opiera sie ona na makrach preprocesora. Preprocesor jest to program który uruchamiany jest na wstepnym etapie kompilacji. Komunikowac sie z nim można za pomocą tak zwanych dyrektyw. Omówimy cztery:

- dyrektywa `#define LABEL` mówi preprocesorowi że nazwa `LABEL` jest od teraz zdefiniowana
- dyrektywa `#undef LABEL` mówi preprocesorowi że nazwa `LABEL` jest od teraz nie zdefiniowana
- para dyrektyw `#ifdef ... #endif` pozwala na wykluczenie lub nie z kompilacji fragmentu

```
#ifdef LABEL
//kod wykonywany gdy LABEL jest zdefiniowany
#endif
```

- podobnie działa para dyrektyw `#ifndef ... #endif`

```
#infdef LABEL
//kod wykonywany gdy LABEL jest zdefiniowany
#endif
```

W naszym programie wykorzystamy preprocesor do wyłączenia śledzenia wykonania funkcji `MergeSort` oraz `Merge` w następujacy sposób:

```
void MergeSort(std::vector<T>& A, size_t first, size_t last){

#ifdef MERGESORT_DEBUG
trace_MergeSort(A,first,last);
#endif

    if( first < last  ){
        size_t mid = (first + last)/2;

        MergeSort(A,first,mid);
        MergeSort(A,mid+1,last);

        Merge(A,first,mid,last);
    }

}

void Merge(std::vector<T>& A, size_t first,size_t mid, size_t last){

#ifdef MERGE_DEBUG
trace_Merge(A,first,mid,last);
#endif
}
```

Teraz, w zależności od tego czy na poczatku pliku `mergesort.cpp` umieścimy bądź nie linie

```
#define MERGESORT_DEBUG
#define MERGE_DEBUG
```

Funkcje `MergeSort` oraz `Merge` będą wyświetlały bądź nie informacje diagnostyczne.

Zobacz folder `Step_4`

Krok 5.

W tym kroku zamiplementujemy i przetestujemy funkcję `Merge`. Dla przypomnienia:

```
Merge(A,first,mid,last)

1:    iL = first,
      iR = mid + 1,
      k=1

2:    utwórz pomocniczą tablicę tmp rozmiaru tmp.length = last-first

3:    while iL <= mid and iR <= last                          //dopóki lewa i prawa podtablica zawiera elementy
      if A[iL] < A[iR]
         tmp[k] = A[iL]
         iL++, k++
      else
        tmp[k] = A[iR]
        iR++, k++


4: przekopiuj pozostałe elementy do tmp                      // jedna podtablica zawiera nieprzekopiowane elementy

5: przekopiuj zawartość tmp do A na odpowiednie miejsce
```

Kod będziemy pisac na raty, dodamy do niego informacje debugujące (kompilacja warunkowa) i bedziemy na bierząco go testować.

Tablica na której bedziemy dokonywac testów wyglada następująco `{333,1,3,4,5,2,4,7,8,444}`. Będziemy obserwować działanie Merge dla parametrów `first = 2, last = size()-2, mid = 2,3,4,5`. Kod testowy wygląda następujaco:

```
    size_t first = 2;
    size_t last = A_test.size() - 2;

    A_test = {333,1,3,4,5,2,4,7,8,444};
    AiSD::Merge(A_test,first,2,last);

    A_test = {333,1,3,4,5,2,4,7,8,444};
    AiSD::Merge(A_test,first,3,last);

    A_test = {333,1,3,4,5,2,4,7,8,444};
    AiSD::Merge(A_test,first,4,last);

    A_test = {333,1,3,4,5,2,4,7,8,444};
    AiSD::Merge(A_test,first,5,last);
```

Oczywiście, jeżeli `Merge` będzie działać dla tej tablicy, to nie znaczy, że jest ona poprawna. Użycie tego przykładu ma nam pomóc wyłapać oczywiste błedy możliwie szybko.

Zadanie Prześledź na kartce jak przebiegną wywołania

- `AiSD::Merge(A_test,first,2,last)`
- `AiSD::Merge(A_test,first,3,last)`
- `AiSD::Merge(A_test,first,4,last)`
- `AiSD::Merge(A_test,first,5,last)`

Wykonajmy kroki `1:` i `2:`

```
void Merge(std::vector<T>& A, size_t first,size_t mid, size_t last){

#ifdef MERGE_DEBUG
trace_Merge(A,first,mid,last);
#endif

/* 1: i 2: */
    size_t Li = first;
    size_t Ri = mid+1;
    size_t Bi = 0;

    std::vector<int> B(last-first+1);
```

```
}
```

Aby zobaczyć jak wyglądają połowki tablic które będziemy łączyć dodamy na koniec nastepujacy kod

```
#ifdef MERGE_DEBUG
auto T_to_cout = [](auto& el){ std::cout << el << " ";  };

std::cout << "\t\tbefore Merge:\t";
std::for_each(A.begin(),A.begin()+first, T_to_cout );
std::cout << "|| ";
std::for_each(A.begin()+first,A.begin()+mid+1, T_to_cout );
std::cout << "| ";
std::for_each(A.begin()+mid+1,A.begin()+last+1, T_to_cout );
std::cout << "|| ";
std::for_each(A.begin()+last+1,A.end(), T_to_cout );
std::cout << std::endl;
#endif
```

Po uruchomieniu otrzymamy cos takiego (sprawdź czy się zgadza):

```
       Merge: (first,mid,last)  = (1, 2, 8)
       left  array: 1 3
       right array: 4 5 2 4 7 8
              before Merge:   333 || 1 3 | 4 5 2 4 7 8 || 444

       Merge: (first,mid,last)  = (1, 3, 8)
       left  array: 1 3 4
       right array: 5 2 4 7 8
              before Merge:   333 || 1 3 4 | 5 2 4 7 8 || 444

       Merge: (first,mid,last)  = (1, 4, 8)
       left  array: 1 3 4 5
       right array: 2 4 7 8
              before Merge:   333 || 1 3 4 5 | 2 4 7 8 || 444

       Merge: (first,mid,last)  = (1, 5, 8)
       left  array: 1 3 4 5 2
       right array: 4 7 8
              before Merge:   333 || 1 3 4 5 2 | 4 7 8 || 444
```

Skoczmy teraz na koniec i wyykonajmy krok `5`:

```
void Merge(std::vector<T>& A, size_t first,size_t mid, size_t last){

....



/* 5: */

    std::copy(B.begin() ,B.end(),A.begin()+first);

#ifdef MERGE_DEBUG
std::cout << "\t\tA after Merge:\t";
std::for_each(A.begin(),A.begin()+first, T_to_cout );
std::cout << "|| ";
std::for_each(A.begin()+first,A.begin()+mid+1, T_to_cout );
std::cout << "| ";
std::for_each(A.begin()+mid+1,A.begin()+last+1, T_to_cout );
std::cout << "|| ";
std::for_each(A.begin()+last+1,A.end(), T_to_cout );
std::cout << std::endl;
#endif
}
```

Po uruchomieniu mamy

```
       Merge: (first,mid,last)  = (1, 2, 8)
       left  array: 1 3
```

```
       right array: 4 5 2 4 7 8
              before Merge:   333 || 1 3 | 4 5 2 4 7 8 || 444

              after Merge:    333 || 0 0 | 0 0 0 0 0 0 || 444
... ...
```

Zadanie: po definicji `B` dodaj

```
    B.front() = 1000;
    B.back() = 9999;
    B[mid-1] = 666;
    B[mid] = 777;
```

Przetestuj.

Wróćmy do kroku `3:`. W tym kroku przekopiowywany jest mniejszy z elementów wskazywanych przez indeksy `Li` oraz `Ri`. dieje sie to tak długo, aż dojdziemy do końca jednej z tablic. Implementacja jest w tym wypadku bezpośrednia

```
    while( Li <= mid && Ri <=last  ){
        if( A[Li] <= A[Ri]){
            B[Bi++] = A[Li++];
        }
        else{
            B[Bi++] = A[Ri++];
        }
    }
```

W kluczowych momentach dodajmy informacje diagnostyczne

```
    while( Li <= mid && Ri <=last  ){
#ifdef MERGE_DEBUG
std::cout << "\t\tLi = " << Li << "  Ri = " << Ri
          << "\t  A[Li] = " << A[Li] << "  A[Ri] = " << A[Ri];
#endif
        if( A[Li] <= A[Ri]){
            B[Bi++] = A[Li++];
#ifdef MERGE_DEBUG
std::cout << "\t Copy from left to \tB:\t";
std::for_each(B.begin(),B.end(), T_to_cout );
std::cout << '\n';
#endif
        }
        else{
            B[Bi++] = A[Ri++];
#ifdef MERGE_DEBUG
std::cout << "\t Copy from right to \tB:\t";
std::for_each(B.begin(),B.end(), T_to_cout );
std::cout << '\n';
#endif
        }
    }
```

Po uruchomieniu otrzymamy coś takiego:

```
       Merge: (first,mid,last)  = (1, 2, 8)
       left  array: 1 3
       right array: 4 5 2 4 7 8
              before Merge:   333 || 1 3 | 4 5 2 4 7 8 || 444
              Li = 1  Ri = 3    A[Li] = 1  A[Ri] = 4   Copy from left to      B:      1 0 0 0 0 0 0 0
              Li = 2  Ri = 3    A[Li] = 3  A[Ri] = 4   Copy from left to      B:      1 3 0 0 0 0 0 0

              after Merge:    333 || 1 3 | 0 0 0 0 0 0 || 444

       Merge: (first,mid,last)  = (1, 3, 8)
       left  array: 1 3 4
       right array: 5 2 4 7 8
              before Merge:   333 || 1 3 4 | 5 2 4 7 8 || 444
              Li = 1  Ri = 4    A[Li] = 1  A[Ri] = 5   Copy from left to      B:      1 0 0 0 0 0 0 0
              Li = 2  Ri = 4    A[Li] = 3  A[Ri] = 5   Copy from left to      B:      1 3 0 0 0 0 0 0
              Li = 3  Ri = 4    A[Li] = 4  A[Ri] = 5   Copy from left to      B:      1 3 4 0 0 0 0 0
```

```
               after Merge:    333 || 1 3 4 | 0 0 0 0 0 || 444

       Merge: (first,mid,last)  = (1, 4, 8)
       left  array: 1 3 4 5
       right array: 2 4 7 8
               before Merge:   333 || 1 3 4 5 | 2 4 7 8 || 444
               Li = 1  Ri = 5    A[Li] = 1  A[Ri] = 2   Copy from left to      B:      1 0 0 0 0 0 0 0
               Li = 2  Ri = 5    A[Li] = 3  A[Ri] = 2   Copy from right to     B:      1 2 0 0 0 0 0 0
               Li = 2  Ri = 6    A[Li] = 3  A[Ri] = 4   Copy from left to      B:      1 2 3 0 0 0 0 0
               Li = 3  Ri = 6    A[Li] = 4  A[Ri] = 4   Copy from left to      B:      1 2 3 4 0 0 0 0
               Li = 4  Ri = 6    A[Li] = 5  A[Ri] = 4   Copy from right to     B:      1 2 3 4 4 0 0 0
               Li = 4  Ri = 7    A[Li] = 5  A[Ri] = 7   Copy from left to      B:      1 2 3 4 4 5 0 0

               after Merge:    333 || 1 2 3 4 | 4 5 0 0 || 444

       Merge: (first,mid,last)  = (1, 5, 8)
       left  array: 1 3 4 5 2
       right array: 4 7 8
               before Merge:   333 || 1 3 4 5 2 | 4 7 8 || 444
               Li = 1  Ri = 6    A[Li] = 1  A[Ri] = 4   Copy from left to      B:      1 0 0 0 0 0 0 0
               Li = 2  Ri = 6    A[Li] = 3  A[Ri] = 4   Copy from left to      B:      1 3 0 0 0 0 0 0
               Li = 3  Ri = 6    A[Li] = 4  A[Ri] = 4   Copy from left to      B:      1 3 4 0 0 0 0 0
               Li = 4  Ri = 6    A[Li] = 5  A[Ri] = 4   Copy from right to     B:      1 3 4 4 0 0 0 0
               Li = 4  Ri = 7    A[Li] = 5  A[Ri] = 7   Copy from left to      B:      1 3 4 4 5 0 0 0
               Li = 5  Ri = 7    A[Li] = 2  A[Ri] = 7   Copy from left to      B:      1 3 4 4 5 2 0 0

               after Merge:    333 || 1 3 4 4 5 | 2 0 0 || 444
```

Zadanie: Po uruchomieniu dla `(1,5,8)` mamy linijkę `after Merge: 333 || 1 3 4 4 5 | 2 0 0 || 444`. Czy pojawienie się 2 po 5 oznacza, że mamy bład w implementacji?

Implementacja kroku `4`: jest już bardzo prosta:

```
    if( Li > mid){
        // elements left in right half
        std::copy(A.begin() + Ri,A.begin()+last+1,B.begin()+Bi);
    } else{
        // elements left in left half
        std::copy(A.begin() + Li,A.begin()+mid+1,B.begin()+Bi);
    }
```

Po dodaniu informacji diagnostycznych wygląda to tak

```
/* 4: */

    if( Li > mid){
        // elements left in right half
        std::copy(A.begin() + Ri,A.begin()+last+1,B.begin()+Bi);
#ifdef MERGE_DEBUG
std::cout << "\t\tElements left in right half:\t";
std::for_each(A.begin()+Ri,A.begin()+last+1, T_to_cout );
std::cout << '\n';
std::cout << "\t\t Copy from right to \tB:\t";
std::for_each(B.begin(),B.end(), T_to_cout );
std::cout << '\n';
#endif
    } else{
        // elements left in left half
        std::copy(A.begin() + Li,A.begin()+mid+1,B.begin()+Bi);
#ifdef MERGE_DEBUG
std::cout << "\t\tElements left in right half:\t";
std::for_each(A.begin()+Li,A.begin()+mid+1, T_to_cout );
std::cout << '\n';
std::cout << "\t\t Copy from right to \tB:\t";
std::for_each(B.begin(),B.end(), T_to_cout );
std::cout << '\n';
#endif
    }
```

Tym razem an wyjściu dostajemy

```
       Merge: (first,mid,last)  = (1, 2, 8)
       left  array: 1 3
       right array: 4 5 2 4 7 8
              before Merge:   333 || 1 3 | 4 5 2 4 7 8 || 444
... ...
              Elements left in right half:    4 5 2 4 7 8
               Copy from right to     B:      1 3 4 5 2 4 7 8

              after Merge:    333 || 1 3 | 4 5 2 4 7 8 || 444

       Merge: (first,mid,last)  = (1, 3, 8)
       left  array: 1 3 4
       right array: 5 2 4 7 8
              before Merge:   333 || 1 3 4 | 5 2 4 7 8 || 444
... ...
              Elements left in right half:    5 2 4 7 8
               Copy from right to     B:      1 3 4 5 2 4 7 8

              after Merge:    333 || 1 3 4 | 5 2 4 7 8 || 444

       Merge: (first,mid,last)  = (1, 4, 8)
       left  array: 1 3 4 5
       right array: 2 4 7 8
              before Merge:   333 || 1 3 4 5 | 2 4 7 8 || 444
... ...
              Elements left in right half:    7 8
               Copy from right to     B:      1 2 3 4 4 5 7 8

              after Merge:    333 || 1 2 3 4 | 4 5 7 8 || 444

       Merge: (first,mid,last)  = (1, 5, 8)
       left  array: 1 3 4 5 2
       right array: 4 7 8
              before Merge:   333 || 1 3 4 5 2 | 4 7 8 || 444
... ...
              Elements left in right half:    7 8
               Copy from right to     B:      1 3 4 4 5 2 7 8

              after Merge:    333 || 1 3 4 4 5 | 2 7 8 || 444
```

Zadanie: Tylko jedna tablica jest posortowana. czy to oznacza, że mamy gdzieś błąd?

wsk: niczego to nie dowoddzi ale uruchom

```
   A_test = {333,1,3,4,5,2,4,7,8,444};
   AiSD::MergeSort(A_test,0,A_test.size()-1);
   AiSD::print(A_test);

   A_test = {333,1,3,4,5,2,4,7,8,444};
   AiSD::MergeSort(A_test,first,last);
   AiSD::print(A_test);
```

Zadanie: Przetestuj do końca funkcję `Merge. Jakie przypadki bedą wię pojawiały w pełnym algorytmie `MergeSort`?

Koniec

Pozostało jeszcze przetestowanie czy po implementacji `Merge` funkcja `MergeSort` rzeczywiście sortuje. Jest to TWOJE zadanie.

(tip: Dla wiekszych tablic lepiej nie wyświetlać trace'ów.)