University of Pittsburgh
Department of Electrical and Computer Engineering
ECE 0301 – Spring 2021

In-Class Assignment #9

Programming concepts:

- Classes and objects, private member functions
- Pointers, functions, and arrays
- Dynamic memory allocation, array member variables
- Sorting algorithms and computational complexity
- The merge sort algorithm

ECE concepts:

- Sorting algorithms and computational complexity

Background: As we have learned, in the `C` language and early implementations of the `C++` language, we can use C-strings to store multiple characters and arrays to store multiple numeric values. In `C++11` and later implementations, we can use objects from the `string` class to store multiple characters and objects from the `vector` class to store multiple numbers. Objects from the `string` and `vector` classes are often a better choice, because they keep track of and can report their sizes, and have many useful functions to implement common operations, such as appending or deleting characters or elements.

In this assignment, you will develop a class for an array that can sort itself, by writing a member function that performs the sorting. You will implement the ***merge sort*** algorithm, which is more efficient than the bubble sort and selection sort algorithms presented in Chapter 8 of Gaddis. As you will see, the steps presented here lead you through development of the sorting algorithm first, followed by the development of the class that uses it.

We have also learned about the close relationship between arrays, pointers and functions in C++. In particular, we have learned that when you pass the name of an array to a function, you are really passing a pointer to the beginning of the array. When the function is instructed to modify an element of the array, it does so by taking the pointer to the beginning of the array, incrementing it by the proper number of memory locations to arrive at the location of the element to be modified, and then writing the new data to that location. For example, consider the following program. The `smooth2` function modifies the first `n-1` elements of the array by adding the neighboring element. The `main` function declares and initializes the array, calls the `smooth2` function, then prints the modified array to standard output. Note that, inside the `smooth2` function, the elements of the array are accessed and modified using pointer arithmetic and the dereferencing operator, or they can be accessed in the same way you access regular array elements.

```
#include <iostream>
using namespace std;

void smooth2(float *x, int n)
{
     for (int i=0; i<n-1; i++)
          *(x+i) += *(x+i+1); // or x[i] += x[i+1];

}

int main()
{
     const int size = 9;
     float a[size] = {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8};

     smooth2(a, size);
     for (int j=0; j<size; j++)
          cout << a[j] << " ";
     return 0;
}
```

The output from this program is

```
0.1 0.3 0.5 0.7 0.9 1.1 1.3 1.5 0.8
```

Merge Sort: The merge sort algorithm breaks down sorting a large array into a series of sorting operations on smaller and smaller arrays, until the arrays to be sorted have only one element. Then each pair of small arrays is *merged* into a larger array, in ascending order, and the merging process is repeated on successively larger arrays, until the original array has been sorted. This algorithm is naturally *recursive*, and follows a *divide-and-conquer* strategy by sorting arrays of increasing size until the task is complete.

Recall that the matrix determinant algorithm that we studied earlier is also recursive, and is computationally inefficient in comparison to other algorithms for solving systems of linear equations. By contrast, the merge sort algorithm is more efficient than many other sorting algorithms, including the bubble sort and selection sort in Gaddis, ch. 8. (More on this to follow.)

The Final Merge Step: To understand the algorithm, suppose we need to merge two sorted lists. This will be the case at the very last stage of our sorting algorithm, as illustrated in Figure 1. The upper row of boxes represents memory locations containing the array that must be merged, which consists of two sub-arrays that have somehow been sorted already. The arrow indicates the start of the second sub-array, consisting of the last five elements, and the first sub-array consists of the first four elements. The lower row of boxes in Figure 1 represents a second array that will receive the merged list; each element of this array may contain old data whose values don't matter, each indicated by a gray **X**.

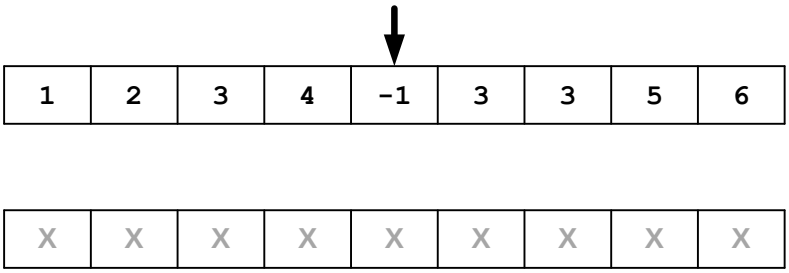

Figure 1: Merging two sorted sub-arrays that are stored in a single array

We must merge the two sub-arrays in the upper array into the lower array. One way to do this can be described in pseudocode:

As long as there are more elements in both sub-arrays:
Compare the first remaining elements in each sub-array
Copy the smaller element into the first open element of the lower array
Remove the copied element from its sub-array
Go back to the top of this loop
At this point, one of the sub-arrays will be empty
Copy the remaining elements from the other sub-array to the lower array

We will illustrate the pseudocode using the example from Figure 1. At the first iteration, $-1 < 1$, so -1 is copied into the first element of the lower array, and then this value is "removed" from the second sub-array. The configuration after completing this step is shown in Figure 2, where the gray shading denotes that this memory location still has its original contents, but is no longer considered part of the second sub-array.

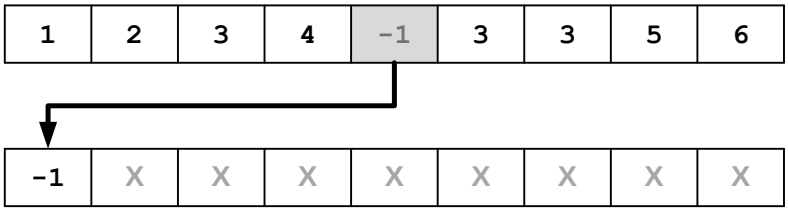

Figure 2: Merge, configuration after first iteration

For the next two iterations, the smaller element is drawn from the first sub-array. Figure 3 shows the array contents after copying these elements, and the first two elements are no longer part of the first sub-array.

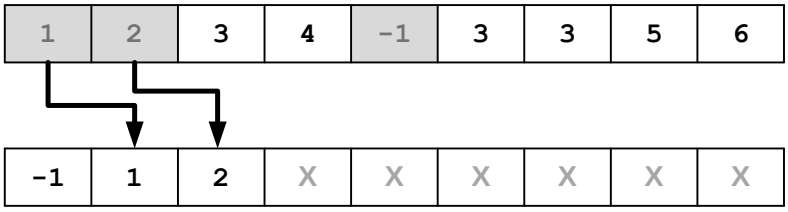

Figure 3: Merge, configuration after three iterations

After four more iterations, the memory contents will be as shown in Figure 4. The number 4 has just been copied to the lower array, leaving the first sub-array empty, which completes the loop in the pseudocode. Hereafter, the last two elements in the second sub-array will be copied to the lower array, completing the merge operation.

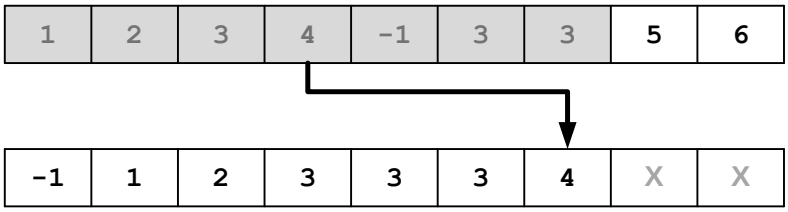

Figure 4: Merge, configuration when first sub-array is empty.

Figures 1-4 help to illustrate what will be needed to translate the pseudocode to C++. The function that implements this operation will need access to the array containing the two sub-arrays to be merged, and a second array to receive the merged elements. We also must supply this function with the index where the second sub-array begins, and the total number of elements.

The merging function must have index counters to keep track of the current position in each sub-array, and must increment these counters when an element is copied to the merged array, thereby "removing" it from its sub-array. There must also be an index to keep track of the write position in the merged array. Remember that these indices will be used in expressions in pointer arithmetic to access or modify array elements. The merging function must also be able to detect when one of the sub-arrays is empty, and then copy them remaining elements from the other array to the merged array.

Earlier Merge Steps: The algorithm described in the previous section assumes that we were working on the very last merge step, in which the array has already been arranged into two sub-arrays, each of which is already sorted. Now suppose that we are at an earlier stage of the algorithm, where we must merge two sub-arrays that together comprise only a portion of the whole array. This situation is depicted in Figure 5, using different data than in our previous example.

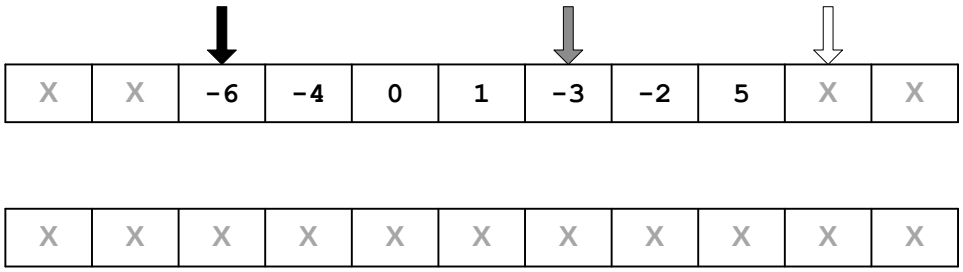

Figure 5: Merging two sub-arrays at an earlier step

In Figure 5, the black arrow marks the index of the first sub-array, the gray arrow marks the start of the second sub-array, and the white arrow marks the first array element after the end of the second sub-array. It should be clear to the reader that we can employ the same strategy used earlier to merge these two sub-arrays, but in this case the merge function must be provided with the three array indices denoted by the arrows in Figure 5.

An important point to take from all of this is that the sorting operation is implemented by merging, but it will only work if the sub-arrays to be merged are already sorted themselves. How can we guarantee that this will be the case when we implement the merge operation?

Splitting: The answer is that we will continually split the array into successively smaller sub-arrays until they are all single-element arrays, which by definition are already sorted. When the merging begins, the merge operation will be performed on pairs of single-element sub-arrays, and each pair will form a sorted two-element sub-array. These will be merged with other one-element and two-element arrays, to form larger sorted sub-arrays, and if the merging is continued, eventually the entire array will be sorted.

The complete algorithm is summarized by the example shown in Figure 6, with splitting operations shown in the upper half, and merging operations in the lower half. Note that the elements remain in the same order during splitting, and all of the sorting happens during the merging operations.

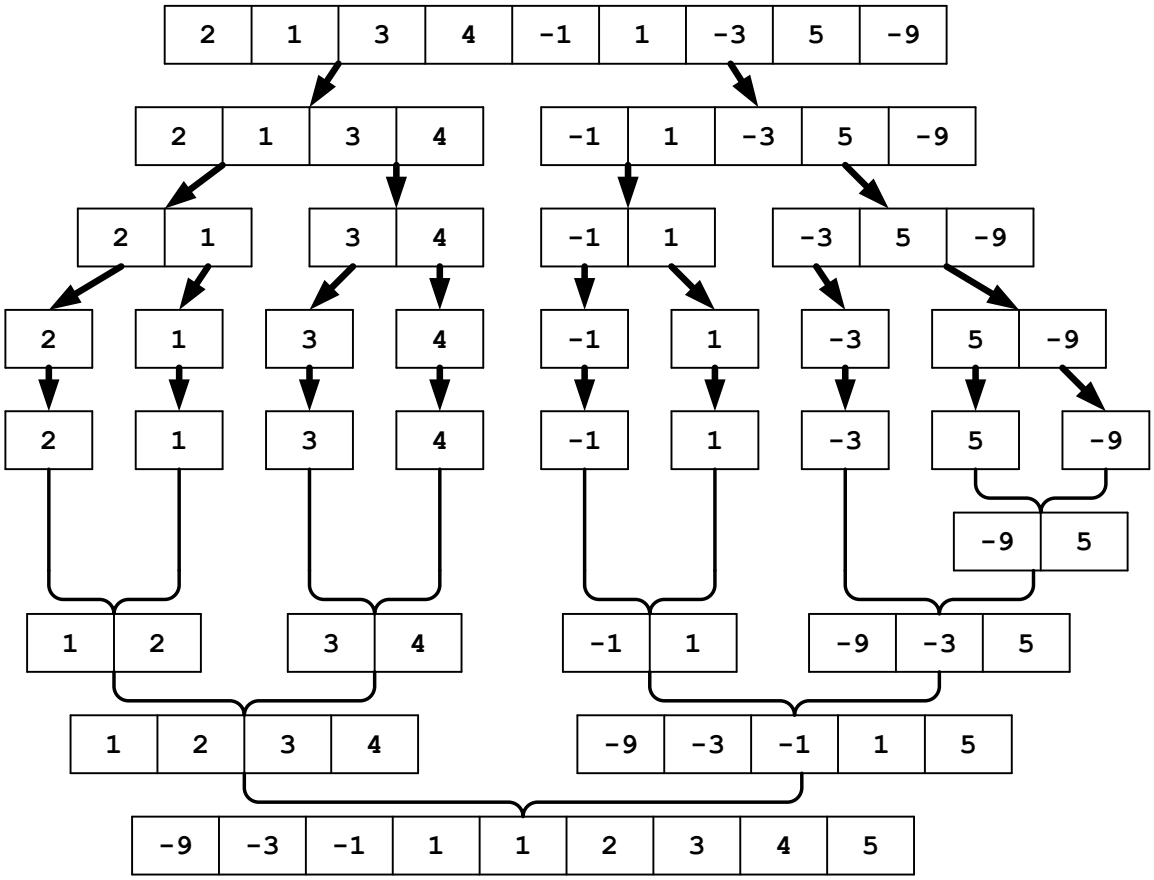

Figure 6: Merge Sort, illustration of complete algorithm

Several aspects of Figure 6 have implications for how we will implement the algorithm. First, we note that the splitting operation is recursive, as the operation is repeated on each sub-array until only one-element sub-arrays remain. Therefore, the function we write to perform the

splitting must call itself recursively, and return to the calling routine if the sub-array to be split has only one element.

Next, consider the vertical symmetry in Figure 6. For example, the sub-array `{5, -9}` is split during the last iteration of splitting, and the same elements are merged in the first iteration of merging. A similar symmetry exists for the splitting and merging of each sub-array. How can we guarantee that our algorithm will reproduce this symmetry? By having the last operation in the splitting function be to call the merge function for the elements that were previously split.

During the splitting, each sub-array is further divided into two smaller sub-arrays. This brings the question of where (at which index) to perform the splitting. If we want to minimize the number of recursions, and thereby minimize the number of merging operations that must be performed, then each division should split the sub-array in half, producing two new sub-arrays with equal numbers of elements, but this is possible only if the number of elements in the original sub-array is even. If we must split a sub-array of odd length, then one of the new sub-arrays will have one more element than the other. Where such cases occur in Figure 6, we have assigned the extra element to the second sub-array, but we could also choose to assign it to the first sub-array.

Figure 6 is somewhat misleading, because it makes it appear as if splitting an array produces two new smaller arrays, but this is not true. The question now is what does it mean to split an array then, if the elements of the sub-arrays will continue to be part of the original array? The splitting function must be provided with the starting and ending indices of the sub-array to be split, and the splitting operation merely consists of identifying the midpoint element that will be the location of the next split, then recursively calling the splitting function with each of the two sub-arrays. The last step in the splitting function is to call the merge function.

Armed with the illustration provided in Figure 6, and the implementation details we have derived from it, we are now ready to describe the splitting function in pseudocode.

Given:

- *An array containing elements to be sorted, and in which we have identified a sub-array to be split. We will refer to this as the "pre-split" array, even though several iterations of splitting may have already occurred.*
- *A second array containing the same elements as the first array, in which we will perform the splitting. We will refer to this as the "post-split" array.*
- *Indices that indicate the start and end of the sub-array to be split.*

Do:

If the sub-array to be split has only one element:
    Go back to the calling function
Otherwise:
    Find an index as close as possible to the midpoint of the sub-array to be split
    Call the splitting function on the sub-array that extends from the starting index to the midpoint index

Call the splitting function on the sub-array that extends from the midpoint index to the end index

At this point, the program will just have returned from the calls to the splitting function, and the two sub-arrays will be sorted

Call the merge function to merge the two sub-arrays

In-Place Computations: Each splitting operation takes the *pre-split* array and produces indices that identify two sub-arrays in the *post-split* array. Similarly, each merging operation takes elements from one array, the *pre-merge* array, and copies them into the proper elements of a *post-merge* array. The algorithms and figures have been presented in this way in order to explain the algorithm, and at this point the reader may think that it is therefore necessary to produce many copies of the original array, only the last of which will be sorted. Certainly Figure 6 seems to reinforce this perception, but it is false.

In fact, the entire algorithm can be carried out using only the original array to be sorted, and a second array containing a copy of the original. The key to doing so is alternating between two arrays that are split at each iteration, and then alternating between these same arrays as each iteration of merging is performed, as shown in Figures 7 and 8.

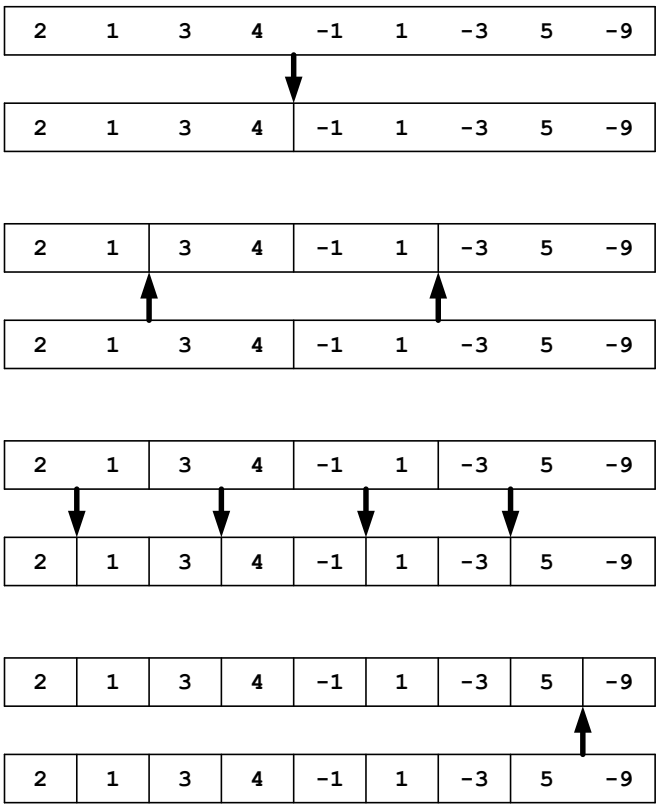

Figure 7: In-Place Splitting

To transfer the splitting operations shown in Figure 7 into C++ code, note that at every iteration after the first, we change our minds about which is the *pre-split* array and which is the *post-split* array. This is accomplished by swapping the array arguments when making the recursive function calls, placing the *pre-split* array in the position normally occupied by the *post-split* array, and vice versa.

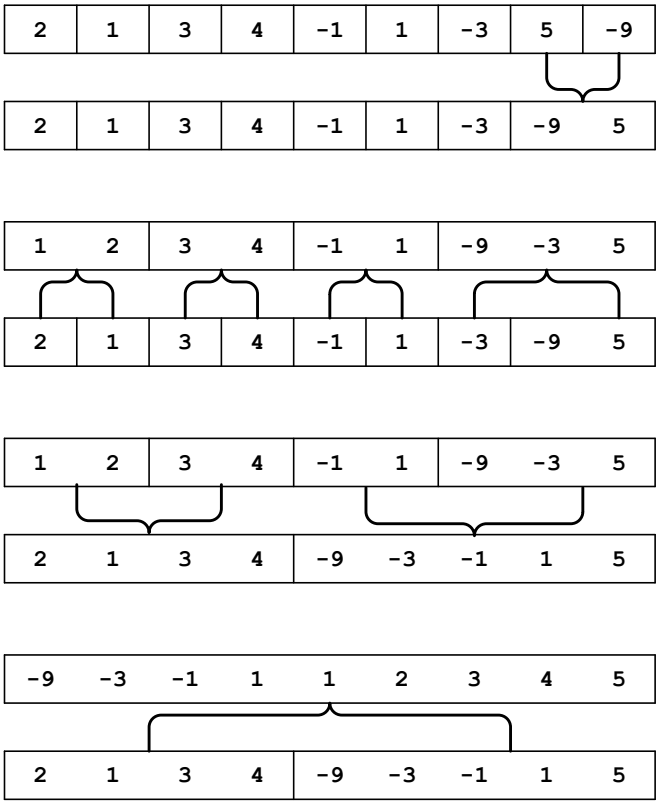

Figure 8: In-Place Merging

To follow the merging shown in Figure 8, we must also alternate at every iteration which array is the *pre-merge* array and which is *post-merge*. This is also accomplished by passing the proper arrays to the two parameters of the merge function. If you pass the arrays in the wrong order, your merge function will not accomplish sorting, because it will be drawing from two sub-arrays that are not already sorted themselves.

Merge Sort: Given an array to be sorted, the complete merge sort algorithm (or at least the version of it we are describing here) can be completed by copying the array to a second array, and then calling the splitting function, with the starting and ending indices set to the start and end of the array.

It is worth noting that one might be tempted to infer from Figures 6-8 that several splitting or merging operations are implemented in parallel, at what we have been referring to as one "iteration" of the corresponding functions. These operations do not occur simultaneously, of course, for they are due to recursive calls to the splitting function, executed sequentially. Each row in Figures 6-8 refers to a different level of recursion, and we leave it as an exercise for the

reader to work out the precise order in which each operation is performed, depending on which sub-array in each pair is processed first in the recursive function calls.

Building a Class: You will begin this assignment by developing a class for a sortable array, with the basic functionality needed to read the elements of an array from a text file, and write the elements to another text file. Several examples are provided of arrays that consist of two sub-arrays that are already sorted, and you will use these examples to develop the merge function. From there you will go on to develop the recursive splitting function, and the complete merge sort algorithm.

When your program is complete, the class you developed will have the following attributes:

- The class will include a member variable to store the number of elements in the array. Any member functions capable of initializing or changing the number of elements must also adjust this member variable accordingly.
- When an object from the class is defined, dynamic memory allocation will be used to allocate the memory required to store the desired number of elements, and release this memory when the object goes out of scope.
- The merge sort algorithm will be included as a public member function.
- The file reading and writing functions will allow your program to read an unsorted array from a text file, sort it, and then write the results to an output file.
- You will also write a member function to fill the array with pseudo-random numbers, as a way to test the merge sort algorithm.

Instructions: Develop software according to the following specifications and submit whatever portion you have completed to the class repository before midnight on the due date.

1. Write the definition for a class for a sortable array, named `SortableArray`. The private member variables are:
    - A pointer to type `double`, which will store the address of the start of the data array.
    - An `unsigned int` variable to store the number of elements in the array.

    Create the following public member functions for the class.
    - A default constructor that:
        - Sets the number of elements to 10.
        - Allocates memory for an array of type `double` with 10 elements, and assigns it to the data array pointer
        - Calls the member function to initialize all array elements to 0 (see below).
    - A destructor that de-allocates the memory that was previously allocated for the data array.
    - An accessor function that returns the pointer to the data array.
    - An accessor function that returns the number of elements in the array.
    - A member function that fills the data array with zeros. This function takes no parameters, and returns nothing.

- A member function that writes header information and an array to a text file, and returns nothing. The parameter for this function is an `ofstream` object for the output file, passed by reference. The format for the file is given below.

Write a `main` function that:

- Defines an instance of your sortable array class.
- Opens a file named `ece0301_merge_sort_results.txt` for writing.
- Calls the member function to write to the output file.
- Closes the output file.

Test your program. Since the array is initialized with ten zeros, the contents of the output file will be as shown below.

```
ECE 0301 - Sorting Arrays
Unsorted Array:
0
0
0
0
0
0
0
0
0
0
```

2. Write a second constructor for your sortable array class that accepts a parameter of type `unsigned int`. This constructor:
   - Sets the number of elements equal to the value of the parameter.
   - Allocates memory for an array of type `double` with the proper number of elements, and assigns it to the data array pointer.
   - Calls the member function to fill the array with zeros.

   Next, create a text file named `ece_0301_unsorted_array.txt`, and type the following lines into it:

```
LENGTH = 9
1
2
3
4
-1
3
3
5
6
```

   The first line of the file gives the number of elements in the array to be sorted, and the remaining lines each contain one element.

Write a function that is **outside the definition of your sortable array class** that reads an array from a text file. The parameter for this function is an `ifstream` object, passed by reference, and it returns an object from your sortable array class. This function must perform the following operations:

- Read the first line from the file. If this line does not begin with "`LENGTH = `", print the following error message to standard output, and exit the program with the failure termination code:
  ```
  ERROR! Invalid input file header.
  ```
- Extract the number that appears at the end of the first line, **which may have more than one digit**. Call your new constructor, and pass the number just read from the file to it, so that it creates a sortable array object with the proper number of elements. At this point, all elements will be initialized to zero.
- Use a loop to read a number of lines from the file that is equal to the number of elements in the sortable array. Convert each line to a `double`, and use the pointer to the data array to store the value just obtained from the input file in the proper element of the array. By doing this, you will fill the sortable array object with the numbers read from the file.
- Return the sortable array object to the calling function.

Write a `main` function that:

- Defines an `ifstream` object, and uses it to open the input file.
- Defines an object from your sortable array class, calls your function to read the input file, and assigns the object returned by the function to the object just defined.
- Defines an `ofstream` object, and uses it to open the output file.
- Calls the member function to write the sortable array to the output file.
- Closes the input and output files.

Very Important notice: assuming that the function that was created to read the data from the file is called `readArray`, and that it will take a variable called `inFile` as an input, then the `SortableArray` object created in the `main` has to be set equal to this function in the same line that it was defined, as follows:

```
SortableArray A = readArray(inFile);
```

If something else was done in the code, like this here:

```
SortableArray A;
A = readArray(inFile);
```

It will not work as intended, so please make sure to do this as mentioned above.

3. Write a member function for your sortable array class that merges two sub-arrays into a second array, by following the algorithm described in this document. Refer to Figures 1-4, and the pseudocode following Figure 1. The parameters for this function are:

- A pointer to a variable of type `double`, which will contain the address of the beginning of array from which we will copy elements. We will refer to this array as the *pre-merge* array.
- A pointer to a variable of type `double`, which will contain the address of the beginning of array that will receive the merged elements. We will refer to this array as the *post-merge* array.
- An `unsigned int` that holds the index of the start of the second sub-array.

This function returns nothing.

Figure 11 illustrates how the merge can be implemented using multiple pointers. The *pre-merge* array contains two sub-arrays, each of which is sorted. The black arrow represents a pointer to the beginning of the *pre-merge* array, and the gray arrow represents a pointer to the beginning of the second sub-array. One of these pointers will be incremented when data are copied from the corresponding sub-array into the *post-merge* object.

The red arrow represents a pointer that is also set to the beginning of the second sub-array, but it will not be incremented; instead, it will be used to tell when all elements of the first sub-array have been copied to the merged object. The blue arrow represents a pointer to the first memory location beyond the end of the second sub-array, and will be used to tell when all elements of the second sub-array have been copied to the merged object.

The green arrow represents a pointer to the beginning of the *post-merge* array. This pointer will be incremented each time an element is coped into the merged object.

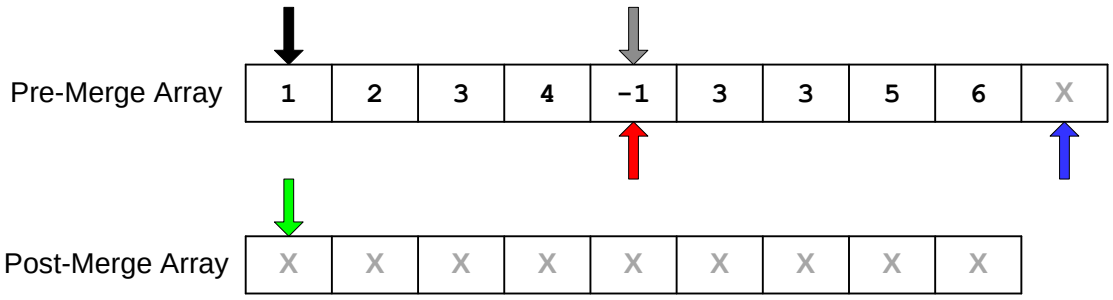

Figure 11: Merging with Multiple Pointers

Using Figure 11, the merge function should perform the following actions:

- Initialize the black, gray, red, blue and green pointers.
- As long as the black pointer is less than the red pointer, and the gray pointer is less than the blue pointer:
  - If the contents of the black pointer are less than the contents of the gray pointer, set the contents of the green pointer equal to the contents of the black pointer, and increment the black pointer.
  - Otherwise, set the contents of the green pointer equal to the contents of the gray pointer, and increment the gray pointer.
  - In either case, increment the green pointer.
  - Go back to the top of this loop.
- At this point, one of the sub-arrays will be empty, but we're not sure which one.

- As long as the black pointer is less than the red pointer:
  - Set the contents of the green pointer equal to the contents of the black pointer.
  - Increment the black pointer and the green pointer.
  - Go back to the top of this loop.
- As long as the gray pointer is less than the blue pointer:
  - Set the contents of the green pointer equal to the contents of the gray pointer.
  - Increment the gray pointer and the green pointer.
  - Go back to the top of this loop.

Modify your file-writing function so that it accepts a second parameter of type `bool`. If this parameter is `false`, then the function must write two lines of text to the output file,

```
ECE 0301 - Sorting Arrays
Unsorted Array:
```

Next, write the elements of the array from the calling object to the file, one per line.

If the `bool` parameter is `true`, this function writes one line of text to the output file,

```
Sorted Array:
```

followed by the elements of the array in the calling object, one per line.

Modify your `main` function so that it:

- Defines an `ifstream` object, and uses it to open the input file.
- Defines an object from the sortable array class, and assigns to it the object returned by the file-reading function. (make sure to follow the notice from step 2)
- Defines an `ofstream` object, and uses it to open the output file.
- Calls the file-writing function to write the original array to the output file, with the bool parameter set to `false`.
- Defines a second object from the sortable array class that has the same number of elements as the first object. You can do this by using the member function that returns the number of elements in the data array, and passing this value to the constructor.
- Uses the first object to call the merge function to merge the elements from the first object into the second. You must pass the data pointers from each of the two objects to this function, as well as the index of the start of the second sub-array (gray pointer).
- Calls the file-writing function again to write the merged array to the output file, with the bool parameter set to `true`.
- Closes the input and output files.

Test your program using the example from step 2. If your program is working correctly, the merge operation will produce a sorted array.

Test your program with the following additional examples. The element at the start of the second sub-array is **underlined**. In each case, you will have to edit the input file so that it contains the required data, and so that the first line shows the correct `LENGTH`. You will also

have to change the integer parameter that you pass to the merge function, to correctly identify the start of the second sub-array.

```
-11, -2, 6, 6, 8, -1, 0, 1
1, 2, 6, 6, 8, -11, -10, -10, -8, -5, -3
```

When you are certain your program is correct, save it in a file named:

`ece0301_ICA09_step03.cpp`

Submit this file to the class repository.

Important note: before the submitting your code to Gradescope, make sure that your step 3 file is able to sort this example correctly:

```
1, 2, 6, 6, 8, -11, -10, -10, -8, -5, -3
```

The testing case we are using for step 3 is this case, and since you have to *manually* change the start of the second sub-array in your `main` program, you have to make sure that the code and the autograder are both testing for this specific example in order to get the full credit for this step.

4. Modify your merge function so that it can merge two sub-arrays that jointly comprise only a portion of the entire array. Figure 12 is a modification of Figure 11 to illustrate this case, so that there are elements of the data array in the calling object before the beginning of the first sub-array, and after the end of the second sub-array.

   Your merge function will now need three `int` parameters instead of one, which represent the indices corresponding to the black/green, gray/red, and blue arrows in Figure 12.

   Note that the behavior of the merge function will be no different here than that described in step 3. The only difference will be in the initialization of the pointers.

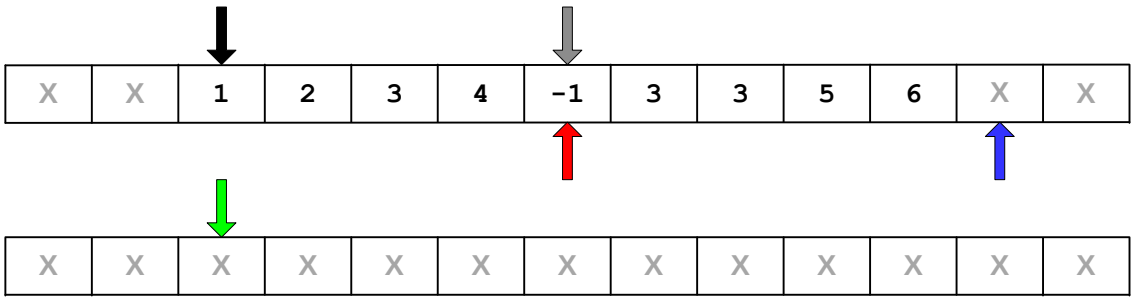

Figure 12: Generalization of Figure 11

   Modify your `main` function so that it calls your merge function properly. Test your program with the following examples. In each case, the data to be merged are non-zero, and there are two zero elements before the beginning of the first sub-array and after the end of the second sub-array. As in step 3, you will have to edit the input file for each example, and manually determine the proper values for three integer parameters.

```
0, 0, -6, -4, -1, 1, -3, -2, 5, 0, 0
```

```
0, 0, -1, 2, 3, 6, 1, 3, 4, 4, 5, 0, 0
    0, 0, 8, 9, 10, 3, 4, 5, 0, 0
```

If your program is correct, the merged arrays for each of the examples should be:

```
 0, 0, -6, -4, -3, -2, -1, 1, 5, 0, 0
0, 0, -1, 1, 2, 3, 3, 4, 4, 5, 6, 0, 0
    0, 0, 3, 4, 5, 8, 9, 10, 0, 0
```

When you are certain your program is correct, save it in a file named:

`ece0301_ICA09_step04.cpp`

Submit this file to the class repository.

Important note: before the submitting your code to Gradescope, make sure that your step 4 file is able to sort this example correctly:

```
0, 0, -1, 2, 3, 6, 1, 3, 4, 4, 5, 0, 0
```

The testing case we are using for step 4 is this case, and since you have to *manually* change the three pointers you pass to the merge function in your `main` program, you have to make sure that the code and the autograder are both testing for this specific example in order to get the full credit for this step.

5. Write a member function to perform splitting that **does not include** the recursive function calls. The parameters for this function are:
    - A pointer to the beginning data array that is to be split, i.e. the *pre-split* array.
    - A pointer to the beginning data array that will hold the result of the splitting operation, i.e. the *post-split* array.
    - An `unsigned int` that holds the index of the start of the first sub-array in the *pre-split* array.
    - An `unsigned int` that holds the index of the next memory location after the end of the second sub-array in the *pre-split* array.

    Since your function will not be recursive yet, it should perform only the following tasks:
    - Compute the midpoint index of the array to be split, where the second sub-array begins, by taking the average of the two integer parameters.
    - Call the merge function. Pass the pointer to the *pre-split* array as the *pre-merge* parameter and the pointer to *post-split* array as the *post-merge* parameter. The integer parameters for this call are the two integer parameters passed into the split function, and the midpoint just calculated, **in the proper order**.

    Modify your `main` function so that it:
    - Defines an `ifstream` object, and uses it to open the input file.
    - Defines an object from the sortable array class, and assigns to it the object returned by the file-reading function.
    - Defines an `ofstream` object, and uses it to open the output file.
    - Calls the file-writing function to write the original array to the output file, with the bool parameter set to `false`.

- Defines a second object from the sortable array class that has the same number of elements as the first object.
- Uses the first object to call the split function, with the pointer parameters set to the data pointers for the two objects that have been defined. Also, the first integer parameter must be set to 0 (the start of the array), and the second integer parameter must be set to the number of elements in the array.
- Calls the file-writing function again to write the split-and-merged array to the output file, with the bool parameter set to `true`.
- Closes the input and output files.

Test your program with the following examples:

```
1, 2, 3, 4, -1, 3, 3, 5, 6
1, 2, 6, 6, 8, -11, -10, -10, -8, -5, -3
```

If your splitting function calculates the midpoint correctly, then the merged array should be sorted correctly.

If you test your program with the other examples from steps 3 and 4, the merged array should not be sorted correctly, because the start of the second sub-array is not at the midpoint.

When you are certain your program is correct, save it in a file named:
`ece0301_ICA09_step05.cpp`
Submit this file to the class repository.

6. Modify your splitting function so that it implements all of the required steps:
   - If the second integer parameter is one more than the first integer parameter, then this means that the sub-array to be split has only one element. Return to the calling function (without taking any action).
   - Otherwise, compute the midpoint index of the array to be split, where the second sub-array begins, by taking the average of the two integer parameters.
   - Recursively call the split function on the first half of the data array, by setting the integer parameters appropriately. Also, you must reverse the order in which you pass the *pre-split* and *post-split* pointers to the split function.
   - Recursively call the split function on the second half of the data array, by setting the integer parameters appropriately. You must reverse the order in which you pass the *pre-split* and *post-split* pointers to the split function.
   - Call the merge function. Pass the pointer to the *pre-split* array as the *pre-merge* parameter and the pointer to *post-split* array as the *post-merge* parameter. The integer parameters for this call are the two integer parameters passed into the split function, and the midpoint just calculated, in the proper order.

   Test your program, using the `main` function from step 5. If you use the examples from step 5 to test your program, you will find that array is **not** sorted correctly in the output file. However, you can still use the output to determine if your program is working correctly. A correct program at this step will produce the following output when the examples from step 5 are used as the input:

```
-1, 0, 0, 1, 2, 3, 3, 3, 4
```

```
-11, -8, 0, 0, 0, 0, 0, 0, 1, 2, 6
```

The erroneous zeros are inserted in the output file, because the second sortable array object defined in `main` was initialized to all zeros.

7. Write a new member function that implements the complete merge sort algorithm. This member function accepts no parameters, and returns nothing. This function must complete the following tasks:
    - Define a pointer to `double`, and assign to it the data array pointer for the calling object. You can use the member function that returns the data pointer to do this.
    - Define a new sortable array object that that has the same number of elements as the calling object. We will refer to this object as the *working* array.
    - Define a pointer to `double`, and assign to it the data array pointer for the *working* array.
    - Copy the data from the calling object to the *working* array. Use the two pointers that were just defined to do this in a loop structure.
    - Call the split function. Pass the pointer to the *working* array as the *pre-split* pointer, and the pointer to the data array in the calling object as the *post-split* pointer. This is necessary so that the sorted data will end up in the calling object when the algorithm is complete.

    Modify your `main` function so that it:
    - Defines an `ifstream` object, and uses it to open the input file.
    - Defines an object from the sortable array class, and assigns to it the object returned by the file-reading function.
    - Defines an `ofstream` object, and uses it to open the output file.
    - Calls the file-writing function to write the original array to the output file, with the bool parameter set to `false`.
    - Calls the member function that implements the complete merge sort algorithm.
    - Calls the file-writing function again to write the (now sorted) array to the output file, with the bool parameter set to `true`.
    - Closes the input and output files.

    Test your program with the following examples. If your merge and splitting functions are correct, then the output array in each case should be sorted in ascending order.

```
2, 1, -3, 4, -1, 1, -3, -5, -9
6, -2, 1, 3, -3, -1, 0, 4, -5, 2, 5, -4
9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

    When you are certain your program is correct, save it in a file named:
    `ece0301_ICA09_step07.cpp`
    Submit this file to the class repository.

8. If you examine your program from step 7, you will notice that the merge and split functions are called only by other member functions. Modify these functions so that they are `private` member functions for the class, accessible only by other functions in the class. To do this, place the prototypes for these functions under the `private` access specifier.

   This will prevent other functions such as `main` from executing the merge or split operations. It is desirable to limit access to these functions, because they cannot perform a complete sorting of the data array by themselves, and allowing other functions to have access to them could corrupt the data array. However, by using the public member functions, one can fill a sortable array object with zeros or data from a file, and sort the data in ascending order.

   Test your program to make sure that the functionality has not changed.

9. Write a new member function that fills the data array with pseudo-random numbers of type `double` that take their values between -1 and +1. This function takes no parameters and returns nothing.

   To do this, follow the example for generating pseudo-random integers presented in Gaddis, Section 3.9. Your function should seed the random number generator with the output of the `time()` function, as shown in Program 3-25. You must include the `cstdlib` and `ctime` libraries to use the `rand()`, `srand()`, and `time()` functions.

   How can you convert a pseudo-random `int` produced by the `rand()` function into a pseudo-random `double` that takes its value between -1 and +1? The first step is to convert it to type `double` with value between 0 and 1. If `rand_01` has been declared of type double, the following statement will generate a pseudo-random integer, and then convert it to type double, with value between 0 and 1,

   ```
   rand_01 = static_cast<double>(rand()) / RAND_MAX;
   ```

   In this statement, `RAND_MAX` is a constant defined in the `cstdlib` library, with its value equal to 32767, the maximum integer produced by `rand()`. To convert this number so that it takes its value between -1 and +1, use the statement

   ```
   rand_pm1 = 2*rand_01 - 1;
   ```

   Modify your `main` function so that it:

   - Defines a sortable array object with 256 elements.
   - Fills the object with pseudo-random values between -1 and +1.
   - Defines an `ofstream` object, and uses it to open the output file.
   - Calls the file-writing function to write the original array to the output file, with the bool parameter set to `false`.
   - Calls the member function that implements the complete merge sort algorithm.
   - Calls the file-writing function again to write the (now sorted) array to the output file, with the bool parameter set to `true`.
   - Closes the output file.

If your program is working properly, the output file should contain a list of pseudo-random values, first in their original unsorted order, and then sorted in ascending order. Test your program several times, and ensure that both positive and negative values are produced, that all are between -1 and +1, and that they are sorted correctly.

10. Separate your program into three files:
    - A class specification file named `SortableArray.h`. This file must contain the class declaration, with prototypes and access specifiers for all members, and it must have an include guard.
    - A class implementation file named `SortableArray.cpp`. This file must contain the definitions for all of the member functions. It must also have a pre-processor directive to include the class specification file.
    - A program file named `ece0301_ICA09_step10.cpp` that contains only a pre-processor directive to include the class implementation file, and your `main` function.

    Submit these files to the class repository.