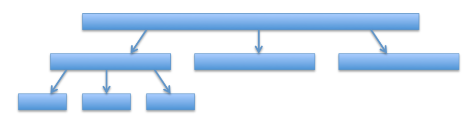
* **Due** Jun 4 by 11:59pm

This assignment will focus on writing an alternative to the standard merge sort algorithm. Instead of splitting our array into two sub-arrays, per the regular algorithm, we will implement a solution where we split our array into three sub-arrays.

**Three-way Merge Sort**

Recall that merge sort has two phases: divide and conquer. During the divide phase the array is recursively split into two sub-arrays of equal size. During the conquer phase each sub-array is merged into a new sorted array. A merge is performed after each set of recursive calls to a left sub-array and right sub-array.

Our assignment will perform a three-way split on our array.



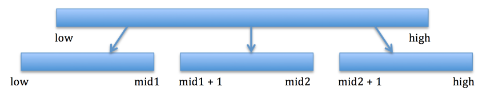
You can calculate the indices between index low and high as follows:

numberOfElements <-- high - low + 1   
mid1 <-- low + (numberOfElements / 3)  
mid2 <-- low + (2 \* numberOfElements / 3)

Each of the sub-arrays can be defined using the following range:

|  |  |  |
| --- | --- | --- |
| **Sub-array** | **Low index** | **High index** |
| left | low | mid1 |
| middle | mid1 + 1 | mid2 |
| right | mid2 + 1 | high |

These indices are displayed below:



**Divide step**

Begin by writing the recursive calls for the divide phase of merge sort.

* Your mergeSort() function should invoke itself recursively three times.

mergeSort(array, a, b);  
mergeSort(array, c, d);  
mergeSort(array, e, f);

* You will need to pass a new low and high index with each recursive call (from the table above).
* Your base case should be if you have fewer than three elements between low and high.
  + If you have two elements, simply swap them if they are out of order.
  + For one element, you can just return from the function call.

**Conquer step**

Write the merge() function for your mergeSort() function. The merge() function should be invoked after you have called mergeSort() recursively three times.

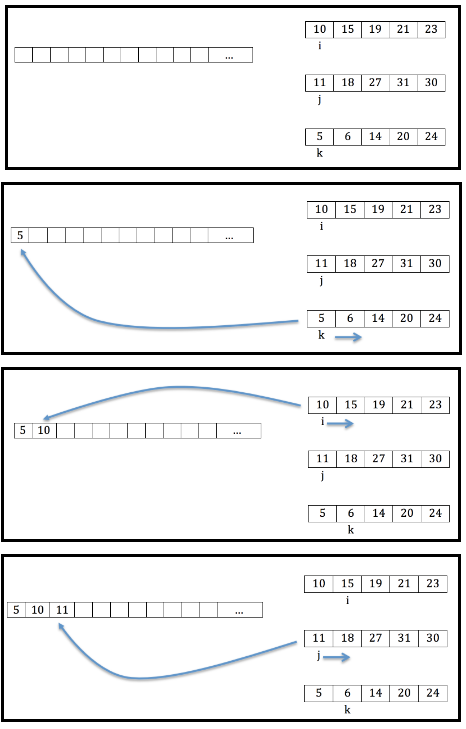
Your merge() function should accept the bounds for all three sub-arrays. For example:

merge(array, low1, high1, low2, high2, low3, high3);

It should then perform the merge operation as described in class:

* You will need a temporary array to hold the merged values during the operation
* You will need to keep track of three pointers, one for each sub-array, that can keep track of the next element to add to your temporary array.
* After all elements have been merged into your temporary array, copy all elements from the temporary array back into the original input array.

Here are a few steps of a hypothetical merge() operation:



*Note: You will need to take into account when any pointer (i, j or k) has exhausted the elements in a sub-array!*

*Note: You will also need to loop until you have exhausted all elements from each sub-array!*

**Testing**

Create a test class and thoroughly test your methods above. Part of your grade will be determined by how you verify the functionality of your mergeSort() function. Your tests should include at least the following cases:

* mergeSort() on arrays of size two or less.
* mergeSort() on an array of 10 elements, where each element is the same value. ie. [7, 7, 7, 7, 7, 7, 7, 7 ,7, 7]
* mergeSort() on a reverse sorted array. ie. [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
* mergeSort() on an array of n random numbers.
  + Test your sorts for n = 100, 1000, 10000, and 100000. How can you verify your results?

**Submission**

Submit the code for your mergeSort() to the dropbox on Canvas. Also, submit a few paragraphs that answer the following questions:

* What are the advantages/disadvantages of your design above?
* Would you use your three-way merge sort or the typical two-way merge sort? Why?
* What is the big-oh for your mergeSort() algorithm in terms of:
  + Time?
  + Space?