## Goal

In this lab you will explore the performance of quick sort and merge sort. Merge sort will be modified to implement a different algorithm for the merge operation. Quick sort will be modified to use a different partition algorithm.

## Resources

* Chapter 8: An Introduction to Sorting
* Chapter 9: Faster Sorting Methods

## Introduction

As mentioned in the first recursion lab, timing code introduces a number of difficulties. Unlike the number of comparisons, the time will vary from one test to the next. One reason for using timings in this lab instead of counting comparisons is that the performance of different versions of quick sort under examination will depend on factors besides the number of comparisons.

### Generic Recursive Sort

Both quick sort and merge sort are recursive algorithms and have essentially the same algorithm.

Sort(A)

1. If A is small enough, just return A.
2. Split A into A1 and A2.
3. S1 is Sort(A1).
4. S2 is Sort(A2).
5. Return combine(S1, S2).

Merge sort does its work on the way up the recursion tree. Its split is easy. It divides the array of values exactly in half. The split operation is easy and consists of just computing the index of the middle of the array. Combining the sorted sub-arrays, though, is where merge sort does all of its work. The merge method will combine two sorted arrays into one.

Quick sort, on the other hand, does its work on the way down the recursion tree. Its split is performed by the partition method, which will pick an element (pivot) in the array and then reorders the rest of the array with respect to the pivot. The combination method for quick sort, on the other hand, is easy. The sorted arrays just need to be spliced together. After the partition and recursive sorts, the values are in their final positions already and the combine operation is “do nothing.”

The split of quick sort depends on which data value is used as the pivot. Consequently, the performance of quick sort will vary depending on the order of values in the array to be sorted. Merge sort, on the other hand, will always split the array in half and is much less sensitive to the order of values in the array.

### Variations of Quick Sort

The basic version of quick sort presented by most other textbooks has a partition method that chooses a fixed element as the pivot. Usually, either the first or last value in the range to be split is chosen. Quick sort then proceeds according to the general algorithm.

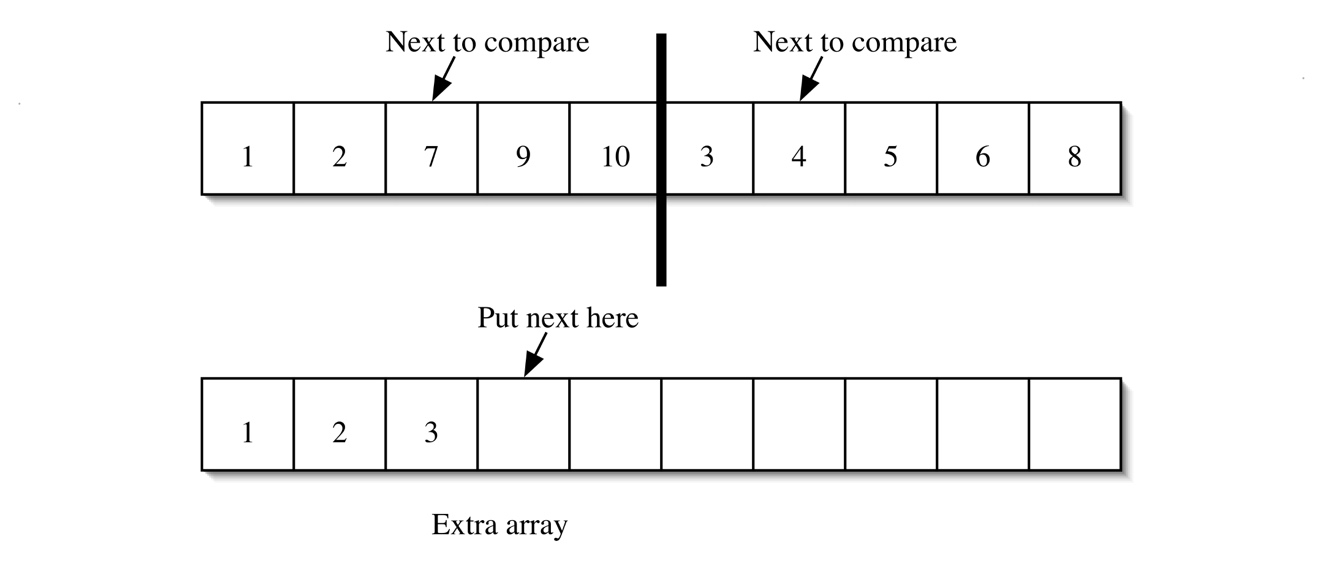
In practice, quick sort is the fastest general-purpose sort available, but there are a couple standard variations that improve the performance by a marginal amount.

**The first improvement (version2QuickSort in the lab) is to notice that for small size arrays, insertion sort is faster than quick sort. In part, the performance of quick sort is affected by the cost of doing a recursive call. An iterative insertion sort avoids this cost. Eventually, the increasing number of comparisons that insertion sort does will overtake the benefit of not doing the recursive method calls. In this improvement, the base case for quick sort is changed. For arrays that are small enough, use an insertion sort. Instead of making a lot of calls to insertion sort, one single call after quick sort is finished will accomplish the same task.**

The second improvement (version3QuickSort in the lab) is to choose a better pivot. (This is the strategy used by our book.) The closer the split is to an even split, the better the performance of quick sort will be. Median-of-Three looks at three fixed values, usually the first, middle, and last elements. The values are ordered, and the middle value is chosen as the pivot. This has two benefits. The first benefit is that it is more likely that there will be a good split and the average performance is improved. Even so, a good split is not guaranteed and the worst case performance is still O(n2). The second benefit is that the worst case for the basic algorithm is on arrays that are nearly sorted. Median-of-Three guarantees a good split on sorted data. Since nearly sorted data are fairly common, this shifts the worst case to orders that you expect to see less often.

### Merge Method

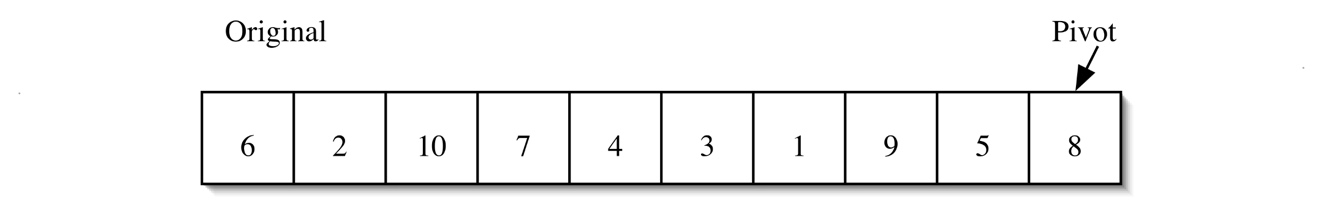
The basic version of merge uses an extra array during merging. The following picture shows an intermediate state of the merge.

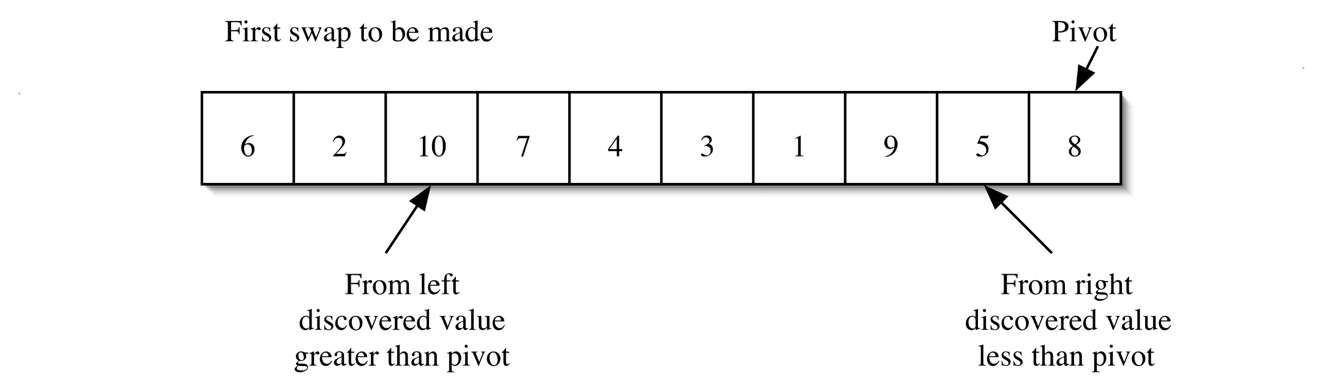


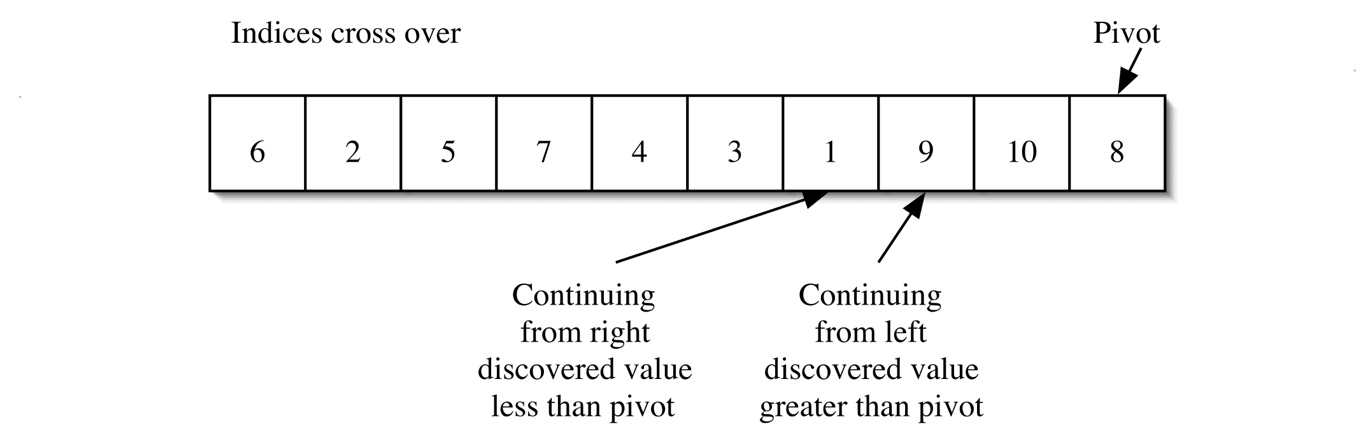
Once all the values have been compared and placed in the extra array, the sorted values are copied back. Doing an in place merge that still has O(n) behavior is a more challenging task.

### Partition Method

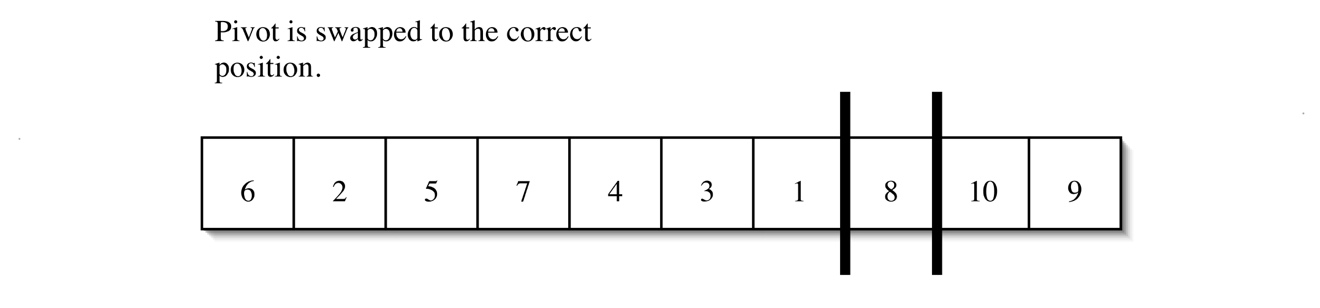
Partition is a surprisingly tricky algorithm. The basic version used in the lab will pick the last value as the pivot. Two indices are used to scan the array. The first index scans the array from the left, looking for a value that is greater than the pivot. The second index scans the array from the right, looking for a value that is less than the pivot. (The pivot will be not scanned.) As long as the indices have not crossed over, the found values will be swapped.







Notice that all values from 9 on are greater than or equal to the pivot. All values left of the 9 are less than or equal to the pivot. The final step is to swap the pivot to split the partition.



### Partition with Median-of-Three

With Median-of-Three, partition first orders the values in the first, middle, and last positions.







At this point, partition continues similarly to the basic version. Note that the first and last elements are guaranteed to be in the correct half of the partition.







All values from 7 on are greater than or equal to the pivot. All values left of the 7 are less than or equal to the pivot. The final step is to swap the pivot to split the partition.



# Experiment

1. Run the Ratios.java

This file Utilizes the implementation of QuickSort, MergeSort and InsertionSort to compare their performance.

1. Using MS Excel, draw a graph and compare the performance of the above sorting algorithms
2. Please test the code with array size 400, 800, 1200 and 2000