More Advanced Sorting Algorithms

Heapsort

Overview

Uses two main steps to sort a given array of sortable items:

1. Transform given array into a heap.
2. Make transformed array into the desired sorted array by repeatedly applying a procedure that partitions the array into two regions – "Heap" region and "Sorted" region

Step 1 (Transform Given Array into a Heap)

One way to do this is to take the elements of the given array one by one and build a heap. This is described in some detail in the textbook (beginning on page 636).

A more efficient method is described to some extent on page 644 of the textbook (Programming Project 5) and amplified here.

1. Treat the given array as a complete binary tree. (We have learned earlier that a complete binary tree can be conveniently represented using an array. Applying this knowledge "in reverse" allows us to view any given array as representing a complete binary tree.)
2. Make the complete binary tree into a heap by applying the *reheapification downward* procedure (that we also learned earlier) to *each node* of the complete binary tree as described in the following line, and annotated and further refined in lines thereafter:
   "Traverse the nodes *from the deepest level up* and, *from right to left at each level*."
   - This way of traversing the tree is equivalent to simply traversing the elements of the array one by one *from the last element to the first element*. (You may want to convince yourself on this by reviewing how a complete binary tree is represented using an array.)
   - This way of traversing the tree also ensures that the *leaf nodes* of the complete binary tree are *visited first*, followed by the *next progressively higher level parents* of the *nodes already visited*.
   - At any point during the traversal, *no reheapification* work is needed if the node currently visited is a *leaf*, for the simple reason that any *1-node tree is always a heap*. When a complete binary tree with `N` nodes (`N > 0`) is represented using an array, it can be shown that *all nodes with indexes greater than or equal to `N/2` are leaves*. Therefore, when the complete binary tree with `N` nodes (`N > 0`) is represented using an array, the efficiency of the algorithm can be improved by traversing the elements of the array one by one *from the element with index `N/2`* (instead of from the last element as given above) to the first element.
   - When the node currently visited is *not a leaf*, the tree rooted at that node *may not be a heap*, but we are *guaranteed that the child/children of the node is/are heaps*. (You may want to convince yourself that "to provide this guarantee" is in fact the reason why the nodes of the original tree have to be traversed in the fashion described above.) To facilitate further discussion, let's call such a tree (*i.e.*, a complete binary tree whose sub-trees are both heaps but whose root may be out of place) a *semiheap*. Our goal at each visit of a node during the traversal of the tree is then to make the semiheap rooted at that node into a heap. To accomplish this, as mentioned above, we apply the *reheapification downward* procedure that is shown in the following as an algorithm called `HeapifySemiheap`.

- Algorithm for `HeapifySemiheap`:

```
HeapifySemiheap(root):
If (root is leaf) // 1-node tree is always already a heap
  return
Else // root is not leaf --> must have left child, may have right child
  // first locate root's largest child...
  root_largest_child = root_left_child  // left child always exists
  If (root has right child)             // right child may exist
    If (value @ right child > value @ left child)
      root_largest_child = root_right_child
  // ...then swap root with its largest child if necessary
  If (value @ root < value @ root_largest_child)
    swap(root, root_largest_child)
    // tree rooted @ root_largest_child may not be a heap after swapping
    HeapifySemiheap(root_largest_child) // heapify to ensure that it is
```

Illustrate Step 1 using the array {6, 7, 5, 9, 2, 10}.

Step 2 (Repeatedly Partition Transformed Array into "Heap" and "Sorted" Regions)

This step works as follows:

1. Initially, "Heap" region is the entire array and "Sorted" region is empty. After each iteration of the step, "Sorted" region grows by one element and "Heap" region shrinks by one element. The process continues until "Sorted" region becomes the entire array and "Heap" region becomes empty.
2. Each iteration of the step is comprised of two sub-steps:
   - Move largest item from "Heap" region into "Sorted" region. (NOTE: Moved items are placed in the order in which they are moved – this way the items in "Sorted" region will be sorted when the process finishes).
   - Reheapify "Reduced" region (*i.e.*, the previously "Heap" region with the largest item removed, which may no longer remain as a heap as a result) that then becomes the new "Heap" region.
3. Observations
   - *Largest item in "Heap" region* for any iteration is at the *root of the complete binary tree*, which is at the *first element* of the array representation of the heap.
   - To move the largest item in "Heap" region then requires that the root be removed. To remove the root of the heap and reheapify the remaining tree as a heap (*reheapification downward* from earlier study of heap):
     - Copy the *root item* into a temporary store for it to be returned (in this case to be moved to "Sorted" region).
     - Copy the *rightmost node item at the deepest level* of the tree into root and remove that rightmost node from the tree. Note that this removal changes "Heap" region into "Reduced" region.
     - Apply reheapification downward procedure at root node (to maintain the reduced tree represented by "Reduced" region as a heap).
   - With "Heap" region represented as an array (which starts as the entire given array but thereafter becomes a subarray of decreasing size of the given array), the *rightmost node item at the deepest level* of the tree is the *last element* of the array. Note that as this last element is removed at each iteration, it happens also to be *just the right location* for storing the copy of root item already placed in temporary store – one author commented that this is perhaps the *most natural thing* he has seen in data structures.

4. Putting the above observations into good use, the two sub-steps mentioned above conveniently translates into the following (be sure to convince yourself of this):
    - (1st sub-step) *Swap* the items in the *first* and *last* elements of the array representing the *"Heap" region*. ("Heap" region is now shrunk by one to form "Reduced" region while "Sorted" region is now grown by one.)
    - (2nd sub-step) Apply `HeapifySemiheap` to the first element of the array representing the "Reduced" region (which is a semiheap) so that it becomes the new "Heap" region.

Illustrate Step 2 using the array {6, 7, 5, 9, 2, 10}.

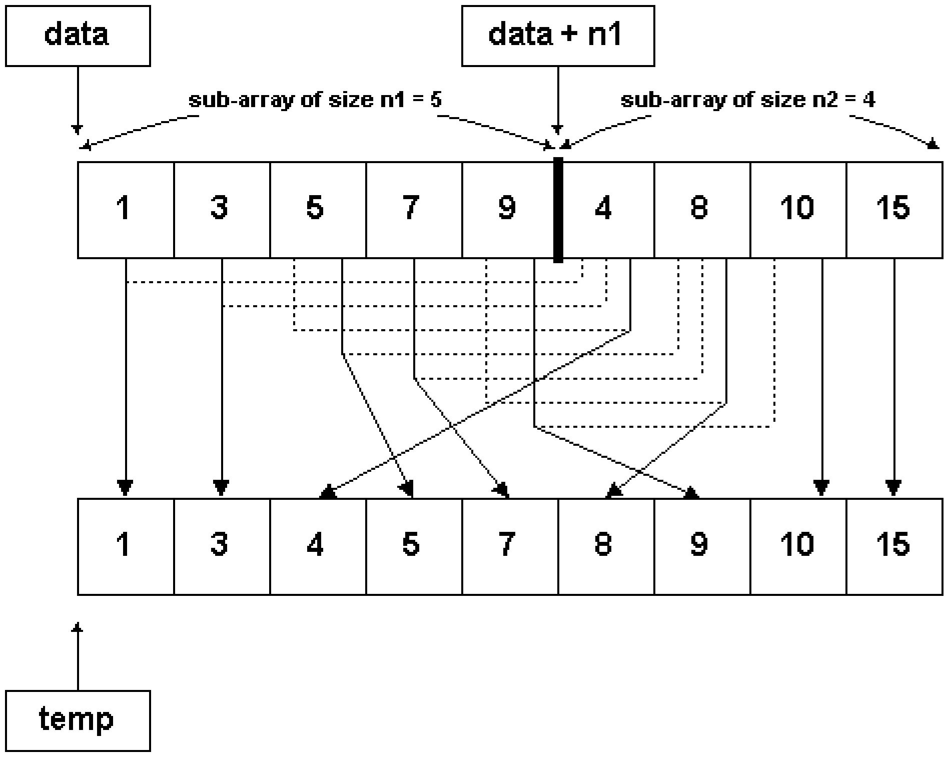
Mergesort

Overview

Given an array of sortable items, repeatedly apply the *usual procedure for merging 2 sorted subarrays into one sorted array* (referred to here as the ***basis for Mergesort***) to transform given array into a sorted array.

Basis for Mergesort (Merging 2 Sorted Subarrays into One Sorted Array)

The usual procedure for merging two sorted subarrays into one entire sorted array is pictorially shown below by way of an example. In the example, the given array (named **`data`**) is of size 9 and is comprised of 2 subarrays of sizes 5 and 4. The subarrays are sorted in themselves but their combination to form **`data`** as shown in the figure is not (note that the dark solid vertical bar separating the 2 subarrays does not really exist – it has been added as a visual aid).



Following are some notable points:

1. A temporary array (named **`temp`** in the example) of size equal to that of the given array is required to temporarily stored the sorted array. (This is perhaps the *biggest disadvantage* of Mergesort.)
2. Descriptively, the sorted array is obtained as follows: orderlily (*i.e.*, one by one from the first element to the last element) copy the *smaller of the leftmost "unaccounted for"* elements of the two subarrays into the temporary array. In the above figure, the element copied is indicated by a solid arrowed line, while the element in the other subarray that was compared is indicated by a dashed line. In case of a tie in value, either of the elements from the two subarrays can be copied. Also, once all the elements of one of the subarrays have been copied, no further comparison is necessary and any "unaccounted for" elements remaining in the other subarray are orderlily copied into the temporary array. When all the elements of the two subarrays are accounted for, the elements in the temporary array are then orderlily copied into the given array to give the desired sorted array.

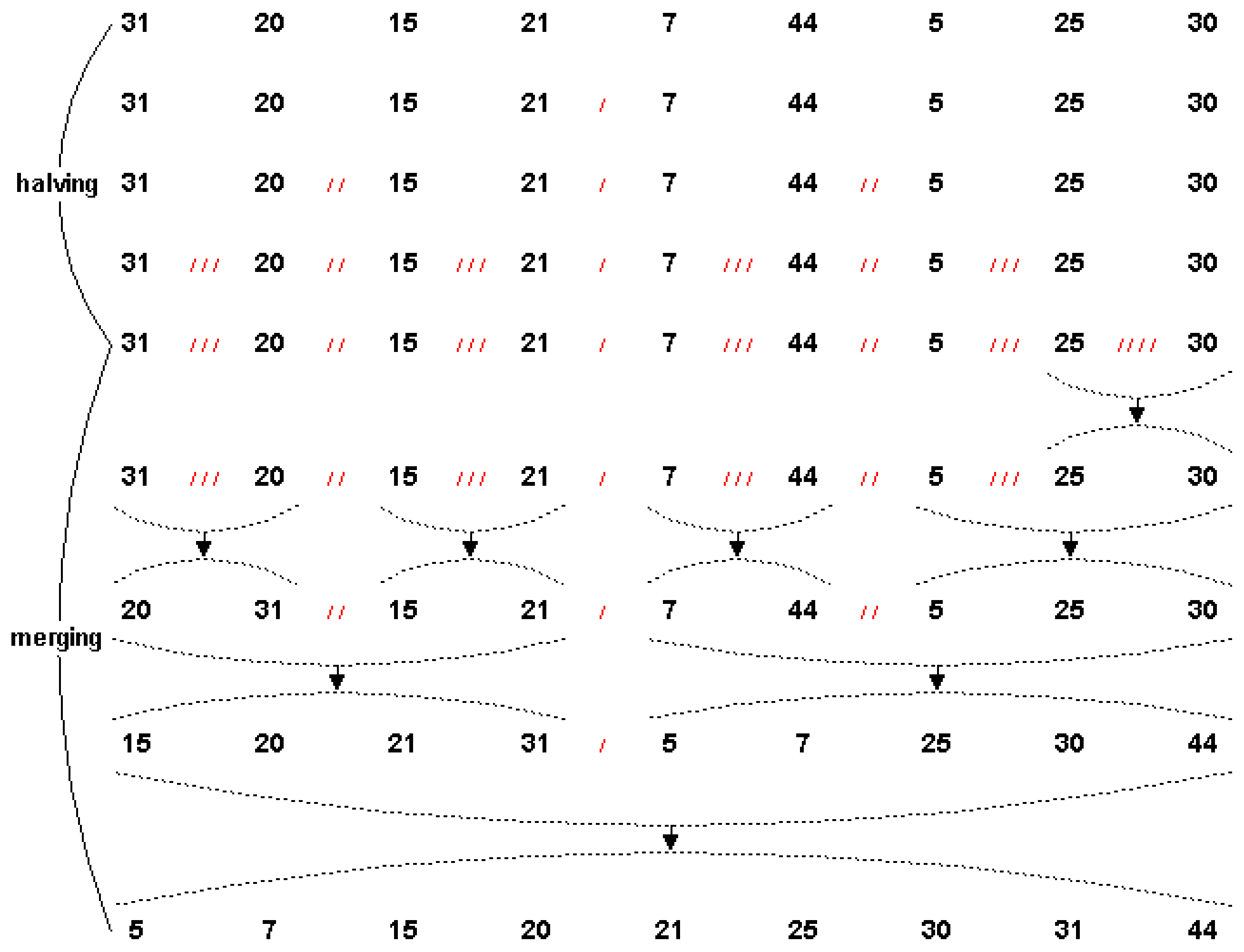
3. Typically, the temporary array is dynamically allocated, used and then deallocated.
4. In implementing the procedure using C++ (also in C), note that the starting addresses of the two subarrays are **`data`** and **`(data + n1)`** as shown in the above figure. This is but a simple application of *pointer arithmetic* that we have learned earlier.

A pseudocode algorithm for the above procedure is given below:

```
Merge(data, n1, n2):
data1 = data   // could have used data, but perhap more readable this way
data2 = data + n1
create temporary array (temp) of size (n1 + n2)
i = j = k = 0  // indexes for data1, data2 and temp, respectively
While (i < n1 && j < n2)
  If (data1[i] < data2[j])
    temp[k++] = data1[i++]
  Else
    temp[k++] = data2[j++]
While (i < n1) // take care of any "unaccounted for" items remaining in data1
  temp[k++] = data1[i++]
While (j < n2) // take care of any "unaccounted for" items remaining in data2
  temp[k++] =  data2[j++]
// copy sorted items from temporary array into given array
For (i = 0; i < (n1 + n2); i++)
  data[i] = temp[i]
release tempory array (where appropriate)
```

Using Basis for Mergesort to Sort a Given Array

How can the basis for Mergesort described above be used to sort a given array that is in general *not* comprised of two subarrays that are sorted? The strategy is to *divide and conquer*. First halve the given array into subarrays, then halve the resulting subarrays into yet smaller subarrays, and so on until no further halving can be performed (*i.e.*, when 1-element subarrays are obtained). Now, because any 1-element array is always sorted, the basis for Mergesort can now be applied to merge two 1-element subarrays into a 2-element subarray that is sorted, which in turn can be merged with another sorted subarray to form another sorted subarray that is even larger in size. This process continues orderlily (*i.e.*, from the deepest halving level upwards) until all subarrays resulting from halving have been merged and a sorted array with size equal to the given array is obtained. This strategy is pictorially illustrated in the following diagram. In the diagram, each *halving action* is indicated using a slash or a number of slashes, with the *number* of slashes representing the *level* of halving; each *merging action*, on the other hand, is indicated using dashed arcs that are connected with an arrow.



A pseudocode algorithm for performing Mergesort on a given array **`data`** of size **`n`** is given below:

```
Mergesort(data, n):
If (n == 1)     // base case
  return
Else            // inductive case
  n1 = n/2      // size of left-half subarray
  n2 = n - n1   // size of righ-half subarray is whatever that is left
  Mergesort(data, n1)       // sort left-half recursively
  Mergesort(data + n1, n2)  // sort right-half recursively
  Merge(data, n1, n2)       // merge 2 sorted halves
```

Quicksort:

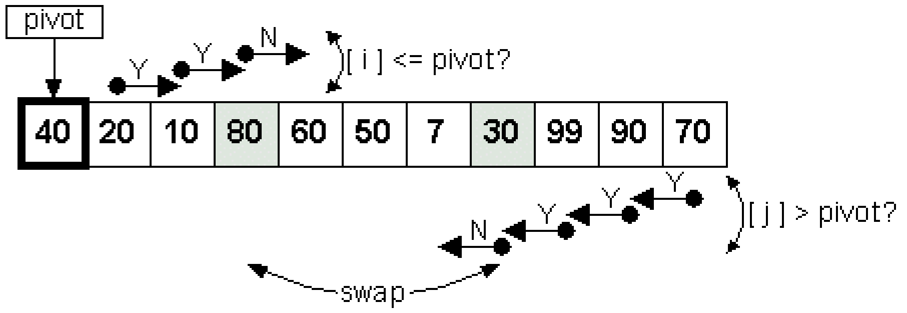
Overview

The goal is to sort a given array of sortable items and the basic idea is described below:

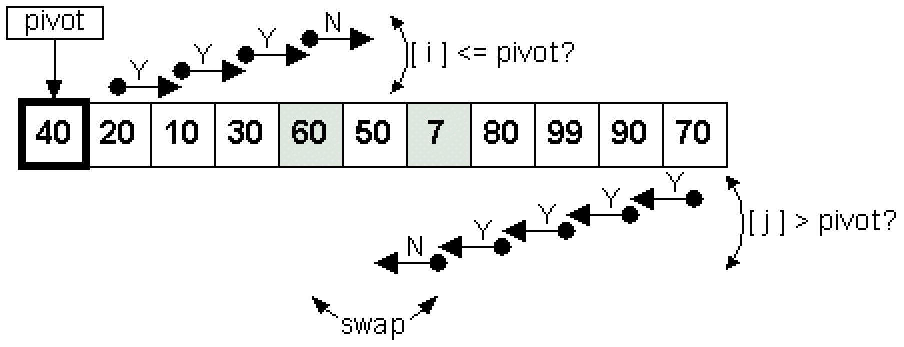
1. Suppose the given array (of size `n`) contains some random values.
2. Consider `[0]` (*i.e.*, the first array element); it is most likely that
    - the value of this element is *out of place* (*i.e.*, there are elements coming after it that have smaller values), and
    - these smaller-valued elements may be located *any where* in the rest of the array (*i.e.*, `[1] ... [n-1]`).
3. Now, suppose we can somehow filter or *partition* `[1] ... [n-1]` into 2 segments such that
    - all elements in 1st segment have values *less than or equal to* `[0]`, and
    - all elements in 2nd segment have values *greater than* `[0]`.
4. We can then insert `[0]` in between the 2 segments and be assured that this element is already *at where it should be* because all elements coming before it have smaller values and all elements coming after it don't have smaller values;
    - we are done for the element (which, by the way, is called the *pivot element*) and what is left to do is to sort the 2 segments that come before and after it.
5. Sorting each of the 2 segments is but smaller versions of the original problem of sorting the entire given array, and we can definitely use the same approach again; of course,
    - if a segment has only *one* element, there is *nothing left to do* (any 1-element array is always sorted), and
    - if a segment has only *two* elements, we only need to *swap the elements if necessary* to sort it.

Partition `[1] ... [n-1]` into Two Segments

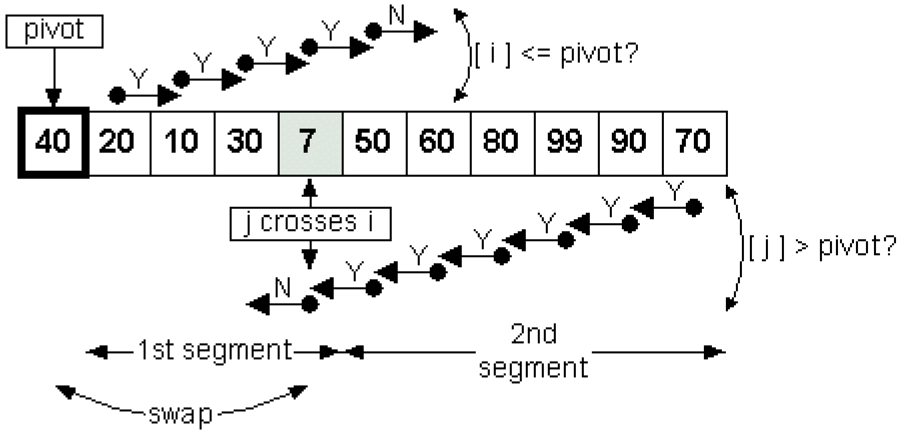
The big question is, of course: How are we going to partition `[1] ... [n-1]` into 2 segments as described above? Fortunately, it is conceptually also rather simple and is illustrated in the following series of figures.



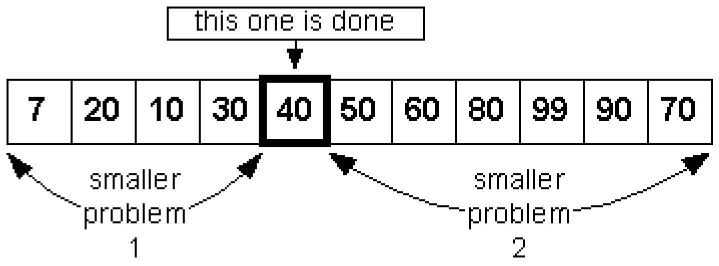
We want segment 1 to be the left portion and segment 2 the right portion of `[1] ... [n-1]`, so we search for an element that does not satisfy `[i] <= pivot` (out of place relative to segment 1) starting at `[1]` and moving forward; in the illustration, `[3]` was found. Similarly, we search for an element that does not satisfy `[j] > pivot` (out of place relative to segment 2) starting at `[n-1]` and moving backward; in the illustration, `[7]` was found. The elements found are swapped so that they are no longer out of place.



We then continue searching for more out of place elements; in the illustration, `[4]` and `[6]` are found and they are again swapped.



The searching process continues until the two search fronts cross each other (*i.e.*, `j < i`). When this happens, we stop because we have just established `[j]` to be the *tail element* of segment 1. (You may want to convince yourself on this by examining the case illustrated.) The partitioning is now complete. To insert the *pivot element* between the two segments (thus placing it at where it should be), we simply swap it with the *tail element* of segment 1 as indicated in the figure.



We are now left with two smaller problems similar to the original problem that we can solve using the same approach.

Algorithm for Partitioning `[1] ... [n-1]` into Two Segments

A pseudocode algorithm for partitioning an array `data` of size `n` (with `n > 1`) is given below. Note that `pivot_index` is the index of the array element into which the pivot element is *eventually* placed and the value of this index is returned to the caller (shown here returned indirectly, by making `pivot_index` a reference parameter in C++ for instance).

Partition(data, n, pivot_index):

```
pivot = data[0]  // value of pivot element
i = 1            // search-forward index
j = n - 1        // search-backward index
While (j >= i)   // do until search indexes cross
  While (i < n && data[i] <= pivot) // search-forward loop
    i++
  While (data[j] > pivot)           // search-backward loop
    j--
  If (i < j)                        // indexes have not crossed
    swap(data[i], data[j])
// set pivot_index & place pivot element at where it should be
pivot_index = j
data[0] = data[j]
data[j] = pivot
```

Algorithm for Quicksort

A pseudocode algorithm for sorting an array **`data`** of size **`n`** using Quicksort is given below.

Quicksort(data, n):

```
If (n > 1)                              // n == 1 is stopping case
  // partition array & set pivot index
  Partition(data, n, pivot_index)       // pivot_index assumed declared
  // compute sizes of subarrays (segment 1 & segment 2)
  n1 = pivot_index                      // size of segment 1
  n2 = n - n1 - 1                       // size of segment 2
  // sort subarrays recursively
  Quicksort(data, n1)                   // sort segment 1
  Quicksort(data + pivot_index + 1, n2) // sort segment 2
```

Choosing a Good Pivot Element

In the above discussion on Quicksort, we have assumed that the given array contains random values. In this case, the choice of using the first array element as the pivot element is perhaps as good as any others. The analysis of running times presented in the textbook (beginning on page 626) points out that the choice of a good pivot element is critical to the efficiency of Quicksort. According to the authors, if we can ensure that the pivot element is *near the median* of the array values, then Quicksort is very efficient. One technique that is often used to increase the likelihood of choosing a good pivot element is to choose three values from the array (such the first array value, the last array value, and the mid-array value) and then use the middle of these three values. Several other common techniques to speed up Quicksort are discussed in the Self-Test Exercises and Programming Projects of the textbook.

Performance Analyses/Comparisons (of sorting algorithms)

Although we will not get into the details of the analysis of the performance for the sorting algorithms discussed, such performance analyses and comparisons are important in practice for obvious reasons. You are strongly encouraged to read those sections in the textbook that describe the analyses of the sorting algorithms. Listed below are some summary items regarding the performance of the sorting algorithms taken from the textbook:

1. *Selectionsort* and *Insertionsort* have quadratic running times, i.e. ***O(n^2)***, in both the worst case and the average case. This is comparatively slow. However, if the arrays to be sorted are short, or if the sorting program will only be used a few times, then they are reasonable algorithms to use. *Insertionsort* is particularly good if the array is nearly sorted to begin with.
2. *Mergesort* and *Quicksort* are much more efficient than *Selectionsort* and *Insertionsort*. They each have an average-case running time that is ***O(n log n)***.
    - *Mergesort* has the advantage that its worst-case running time is O(n log n), but it has the disadvantage of needing to allocate a second array for use in the merging process.
    - *Quicksort* has the disadvantage of a quadratic, i.e. ***O(n^2)***, worst-case running time. However, with some care in the pivot selection process, this worst-case running time can usually be avoided.
3. *Heapsort* has O(n log n) running times in both the worst-case and the average case.

Textbook Readings: Sections 13.2 and 13.3 of Chapter 13.
