Discussion Prompt Questions
Module 10
Brian Loughran
Johns Hopkins Data Structures

1. What are some key factors that differentiate the various sorts you have seen?
2. What would you describe as the core idea behind merge sort?
3. What are some good methods for choosing a quicksort pivot, and why?
4. How are subfiles created for shellsort?
5. In what way does a natural merge exploit existing order in the dataset?
6. Describe in your own words the notion of a stable sort.
7. What difference might it make if the records you are sorting are very large or very small?
8. In considering a sorting problem, how important is it to you that some of the data is likely to be ordered?
9. Have you been in a pre-demo development crunch? What were the key decisions you faced, and how did you address them?
10. What are the risks associated with having the wrong algorithm in place as a long-term solution?

ANSWERS:

1. There are many different sorts, as discussed in depth in this module. Some key differences that I have seen are in the time complexity of the sort. Some are O(n lg n), some are O(n^2), some are O(n^3/2), among others. Selecting a sort with a low time complexity can save you a large amount of time for long sorts. Another key factor is the complexity in implementing the sort. Insertion sort is one of the easiest to implement, and there are harder sorts like shell sort, etc. When dealing with short lists, it may be more prudent to select an easy algorithm to implement, especially if there is no appreciable time difference in the sorts. Some sorts work better with mostly sorted data, some sorts do not care if the data is mostly sorted. There are many more key factors differentiating the sorts we have gone over.
2. The core idea behind a merge sort is to split the list into partitions, sort the partitions, then merge the partitions into a fully sorted array. Since the partitions are sorted, you do not have to look at every element in the array to find the next element, you can simply look at the elements at the beginning of each partition.
3. There are many ways to select a pivot. Choosing a random pivot minimizes the chance you will encounter worst case (O(n^2)) performance. You can choose the middle element, this is acceptable in many cases. You can select the first or last element, however this is usually not recommended. You can also take a quick look at the data and select a pivot based on what you see in the array.
4. Subfiles for shell sort are created by partitioning the original array into even sized groups. It is smart not to select a subfile size of 1, or much greater than n/2, otherwise you run the risk of not getting much benefit over a simple insertion sort.
5. Natural merge partitions the data into groups of ascending order. Then the partitions are already sorted. Since the partitions are already sorted you do not have to deal with sorting them, therefore you can save some valuable computing time. The rest of the algorithm is to just merge the sorted partitions.

Discussion Prompt Questions
Module 10
Brian Loughran
Johns Hopkins Data Structures

6. A merge is stable if objects with equal value in the input are always in the same order as in the output. This means that there are no possible swaps of items with equal value. Some examples of stable sort are the insertion sort, bubble sort, merge sort, and radix sort.
7. In sorting very large records, you would want the most efficient algorithm possible to reduce the compute time. If the records you were sorting were very small, you might want to select a simple algorithm, that way you can save time on implementation. Or you may want to pick an algorithm with low overhead so you are not wasting memory, and can go faster than a complex sort with high overhead.
8. If the data is likely to be mostly ordered, you may want to select an algorithm which takes advantage of the natural order of the data, like natural merge. Natural merge can approach O(n) time complexity on a mostly sorted list. If the data is unlikely to be ordered, you may want to take advantage of an algorithm which does not care if the data is ordered, like quicksort.
9. I have been in a pre-demo development crunch. When you are crunched for time like that, it is very important to maintain good, working code that takes care of all edge cases. Not considering all edge cases can lead to a very embarrassing demo. When in a crunch such as this, algorithmic complexity is usually pretty low on the priority list, as long as you can get a working prototype out, you can stand to lose a bit in performance, employ a simple algorithm, and then tinker with a more complex algorithm later.
10. The wrong algorithm can put a large tax on the performance if it is in place long-term. If it is an in-house tool, you can expect that the efficiency of the company workforce will suffer due to slower runtimes. If it is a commercial tool, other companies may recognize your inefficiencies and take market share from you. Add that to the fact that the wrong algorithm can produce entirely wrong results, and you have pretty good motivation to implement the right algorithm in your projects.