Algorithms & Complexity 1/30/2017 – 2/3/2017

0145-344-001

Note Taker: Jai Punjwani

Topic: Internal Sorting

PowerPoint: <http://home.adelphi.edu/~siegfried/cs344/344l3.pdf>

**Bubble Sort** – impractical for large collections; compares each element with the following element until no swaps are made in a run-through (pass) or until we have finished all possible comparisons.

Best case: ***O***(n) [check array once without swapping]

Worst case: ***O***(n2) [(n-1)2 comparions]

**Cocktail Shaker Sort** – variation of bubble sort that sorts from both directions so that elements that are at opposite ends are found quicker. For large data sets, this is still inefficient.

Best case: ***O***(n)

Worst case: ***O***(n2)

**Selection sort** – inefficient in-place comparison sort in which we find the largest element and swap it with the last element, then we repeat with our unsorted list that is 1 element less each time.

All Cases: ***O***(n2) (better than bubble sort b/c it makes fewer unnecessary swaps)

**Insertion sort** – starts with a sorted array of the first element. Then it considers an array up till the second element, and eventually till the nth, inserting the new element in each iteration into its proper place in the preceding sub-array.

Best Case: ***O***(n) [when array is close to or is already sorted]

Worst Case: ***O***(n2)

**Quick sort** – divide and conquer sort. Basically, the algorithm divides an array into two smaller sub-arrays by first choosing a pivot, and then moving all elements less than the pivot into the array on the left, and all elements greater than the pivot into the array on the right. Elements that are equal to the pivot, and the pivot itself can go in either array. Then, for each array we choose another pivot and repeat the process recursively.

Useful Tutorial (recursive implementation) <http://www.vogella.com/tutorials/JavaAlgorithmsQuicksort/article.html#quicksort_implementation>

Average Case: ***O***(n log n)

Worst Case: ***O***(n2)

**Merge sort** – another divide and conquer algorithm. We repeatedly halve our arrays (by keeping track of the left and right array), sort the left half, right half, and then merge the left and right halves. Note that our base case is an array of size 2, which is sorted via a simple swap. Also note that this algorithm requires an auxiliary array to copy our merged array into on each step.

Useful Tutorial (recursive implementation):

<http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html#mergesort_implementation>

All Cases: ***O***(n log n) time complexity + ***O***(n) space complexity

**Heap sort** – heap is a tree-based data structure. In this algorithm, we try to obtain a sorted heap in which the father is greater than its sons (each root is greater than its sons). First we organize the array into a heap, and then we repeatedly take the first element and place it on the correct position on the heap.

All cases: ***O***(n log n) time complexity + ***O***(n log n) space complexity (to build heap)