Question 1: COMP 160 Overview Part I: Chart of Problems & Algorithms

Runtime of Insertion Sort: O(n^2) because worst case array is reverse sorted and you need to swap and compare every single item

Runtime of Bubble Sort: O(n^2) because worst case array is reverse sorted and you need to bubble up each item to the front

Runtime of Merge Sort: O(nlogn) This is a divide and conquer algorithm because we first split/halve the arrays and then merge the arrays back together at the end. We visit n items and logn is height of the binary tree we create. Recurrence relation is $T(n) = 2T(n/2) + n$

Runtime of Quick Sort: O(n^2) only when the worst pivot is chosen every single time you partition where only 1 element ends up being sorted. Average case is O(nlogn)

Runtime of Heap Sort: O(nlogn); build max-heap is O(n) time and heapify is O(logn), called n - 1 times. Heapsort has the advantage of being in place or constant memory.

Find min in unsorted array: O(n) simple brute force

Find min in min-heap: O(1)

Find min in max-heap: O(n) because we need to search all of the leaf nodes. Invariant of heap is that parents are greater than children so any child could be the min

Find min in BST: O(n) worst case if BST is a linked list

Find min in AVL: O(logn) this is because tree is balanced and will only have to traverse down one path

Find kth smallest in unsorted array using selection: O(n^2) in worst case where you choose the worst pivot at every point

Find kth smallest in unsorted array using selection with random pivot: O(n) as long as you don't partition out n - 1 elements with every pivot choice then average case is O(n)

Find kth smallest in min-heap: O(n) where k is the largest element so we would need to traverse the entire heap

Find kth smallest in BST: O(n) if linked list and k is the largest element

Find kth smallest in AVL: O(n)

Find kth smallest in AVL augmented with subtree size: O(logn)

Finding rank in unsorted array using mergesort: O(nlogn) to sort the array and then constant time access to a specifcic rank

Finding rank of element in heap: O(nlogn) because if we simply just keep extracting the min

Finding rank of element in BST: O(n)

Finding rank of element in AVL: O(n)

Finding rank of element in AVL augmented with subtree size: O(logn)

Sorting an unsorted array in a certain range of numbers signifies counting sort: Runtime is O(n + k) where n is number of items we are sorting and k is number of possible values

Sorting an unsorted array with length l and d digits is radixSort: runtime is applying counting sort times number of digits O(l(n + d))

Enumerate how many numbers are in a given interval using an AVL Tree augmented with span: O(n)

Finding MST on an unweighted, undirected, graph using Prim's algorithm is O((n + m)logn)

Finding MST on a weighted, directed but no negative edges graph using Kruskals algorithm is O((n + m)logn)

Finding SSSP on an unweighted graph using BFS is O(n + m)

Finding SSSP on a weighted graph using Dijkstra's is O((n + m)logn)

Finding sSSP on a weighted graph with negative edges using Bellman Ford is O(nm)

Finding cut vertices on an unweighted graph and outputting the number of cut vertices using DFS is O(n + m)

Inserting into unsorted array is O(1) since you can simply push at back, deleting from an unsorted array is O(1) if its from the back otherwise its O(n) since we have to find element to delete and move all of the elements over, search is O(n), no preprocessing build structure

Inserting into sorted array is O(n) since you have to search where to insert it, deleting an element is O(n) because we need to shift all elements over, searching is O(logn) because of binary search, no preprocessing build structure

Inserting into a BST is O(n) if its linked list, same for deleting and search as well, from an unsorted array it will take O(nlogn) to insert all of the elements into our BST and insert them in the correct place

Inserting into an AVL tree is O(logn) same for search and delete; to insert all elements into our AVL tree it will take O(nlogn)

Inserting into a hash table with chaining is O(1), deleting is O(n) and searching is O(n); to hash all of the elements it is O(n)

Inserting into a hash table with open addressing is O(n), deleting and searching is also O(n) and to hash all of the elements it is O(n)

Question 2: Does CountingSort work when we have an array of (potentially both positive and negative) integers? Do you need to do any modifications to the default algorithm? What about RadixSort

CountingSort would work with positive and negative integers it would just be tricky to implement. As long as each bucket is distinct and we know what number is supposed to be represented by it then we are fine. RadixSort also works with negative integers, you just have to treat the negative sign as another digit.

Question 3: What is a stable sorting algorithm? Which algorithms (as presented in class) are stable?

A stable sorting algorithm is one where order is maintained in regards to where it is so if there are two 2's in the dataset then the first 2 comes before the 2nd when it is sorted. Some examples of stable sorting algorithms are MergeSort, insertionsort, bubblesort, and radixsort.

Question 4: Difference between bottom up and top down in dynamic programming?

The difference is the method we use to calculate our result. The bottom-up approach consists in first looking at the smaller subproblems and then solving the larger subproblems using the solution to the smaller subproblems, whereas top-down is solving the problem regularly and checking to see if you have calculated a solution to a subproblem before. It is in the name one you start at the root and the other you start at the leaves.

Question 5: What is the key Lemma that guarantees correctness of Prim's algorithm? What about Kruskal?

The key lemma that guarantees correctness of Prim's and Kruskal's algorithm is the cut lemma. The cut lemma is that for any set a lightest weight edge between S and V - S is in the minimum spanning tree.