Exam II Practice Problem Set **Solutions**

CSE 373 Summer 2020

The purpose of this problem set is to help you prepare for Exam II by giving you examples of the types of skills you may be asked to use. Please note that it is not intended to be representative of the length of the exam nor the exact types of questions you will be asked. For a more thorough list of the topics that may be included, refer to the topics list and learning objectives list published on the exams page on the course website.

1. ADTs & Data Structures

Below are five computational tasks. For each one, choose the data structure or ADT that is *best* suited to the task from the following list:

Stack, Queue, Hash Map, AVL Tree, Priority Queue, Disjoint Sets, List

In addition, list the basic operations for the data structure or ADT that you chose and give asymptotic worst-case running times for those operations.

a) While processing a list of objects, check if you have processed a particular object before.
Hash Map. Insert, Lookup, and Remove all have constant runtime in the "In-Practice case" or linear runtime in the worst case.

b) Store a list of students and their grades. You must also provide an efficient way for a client to see all students sorted in alphabetical order by name. Give the running time for this operation as well.
AVL Tree. Insert, Lookup, and Remove all have logarithmic runtime. Getting all students in sorted order takes linear time (doing an in-order traversal).

c) Process a digital image to divide the image up into groups of pixels of the same color.
DisjointSets. Find has a O(log* n) runtime (the iterated log, which is close to constant), and Union has a O(1) runtime.

d) Compute a frequency analysis on a file. That is, count the number of times each character occurs in the file, and store the results.
Hash Map. Insert, Lookup, and Remove all have constant runtime in the "In-Practice case" or linear runtime in the worst case.

e) Store the activation records (i.e. objects containing the return address and local variable associated with a function call) for nested function calls.

Stack. Pop has a O(1) runtime and push has a O(n) runtime (if array-based, although it is O(1) when amortized “in practice”) or a O(1) runtime (if linked-list based).

2. PriorityQueues & Heaps

2.1 Max of Min-Heap

a) Given a binary min-heap containing n elements and stored in an array, what is the minimum number of items in the heap’s internal array you would need to inspect to find the one with the largest priority?

Since the heap invariant ensures that a leaf node must be larger than its parent, but gives no guarantee about the relative values of left and right children, to be certain we had found the largest priority item we would need to search all leaf nodes in the heap. As in any binary tree, the number of leaf nodes would be (n + 1) / 2.

b) What about the second-largest?

Finding the second-largest item would require searching the bottom-most layer (leaf nodes) *as well as* any node that is a parent of a leaf node, because the second-largest item could be the parent of the largest item or it could be a leaf node itself and as in part (a) we’d need to search all candidate locations to be certain. The number of leaf nodes would be (n + 1) / 2, and the number of nodes that would be parents of those leaf nodes would be (n + 2) / 4, so the total number of items to search would be (3n + 4) / 4 items.

1.2 Heaps & Sorts

Consider the following code:

```
List<Integer> reverseMinK(List<Integer> list, k) {
      List<Integer> output = minK(list, k);
      reverseSort(output);
      return output;
}

List<Integer> minK(List<Integer> list, k) {
      MinPQ pq = new ArrayHeapMinPQ();
      for (int n : list) {
            pq.add(n)
```

```
    }
    List<Integer> output = new ArrayList();
    for (int i = 0; i < k; i++) {
        int n = pq.removeMin();
        output.add(n);
    }
    return output;
}

void reverseSort(List<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        int n = list.get(i);
        for (int j = i; j > 0; j--) {
            int b = list.get(j);
            int a = list.get(j - 1);
            if (a >= n) {
                list.set(j, n);
                break;
            }
            list.set(j, a);
        }
    }
}
```

In terms of n (the size of list) and k: (assume k <= n and that the list is always an ArrayList)
What are the orders of growth of:

a) The best/worst-case runtimes of minK?
   Best: n. Worst: n log(n).

b) The best/worst-case runtimes of reverseSort?
   Best: n. Worst: n^2.

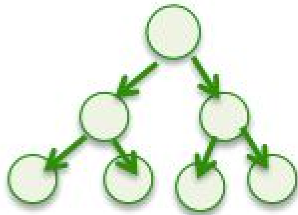
c) The best/worst-case runtimes of reverseMinK?
   Best: n. Worst: n log(n) + k^2.

d) The best/worst-case runtimes of reverseMinK, if all values in the list are guaranteed to be unique?
   Best: n + k log n + k^2. Worst: n log n + k^2.

3. Graphs

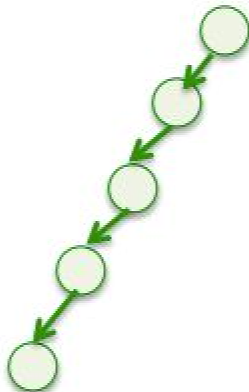
3.1 BFS & DFS

a) Draw a picture of a binary tree with N nodes where BFS takes Θ(N) auxiliary space. Briefly explain why your tree requires BFS to use Θ(N) auxiliary space.



Since BFS uses a queue for the set of "pending" or "perimeter" nodes to explore, it will store each level of the tree in the queue at some point. The largest level of this tree is the last one, which is a constant factor of the total number of nodes N.

b) Draw a picture of a binary tree with N nodes where DFS uses Θ(1) auxiliary space. Briefly explain why your tree only requires DFS to use Θ(1) auxiliary space.



As DFS proceeds through this tree, it will only have a single node stored in the "pending" or "perimeter" stack at each point before immediately popping that element off and proceeding with the algorithm.

3.2 Graph Properties

For each of the following statements, indicate whether it is ALWAYS true, SOMETIMES true (more information would be needed about the situation to determine), or NEVER true.

a) Dijkstra’s algorithm computes an incorrect result when there is a negative weight edge in a graph.
SOMETIMES -- It is still possible for Dijkstra’s algorithm to compute the correct result if there is a negative edge weight. For a trivial example, consider a graph with two vertices and a single edge with weight -1.

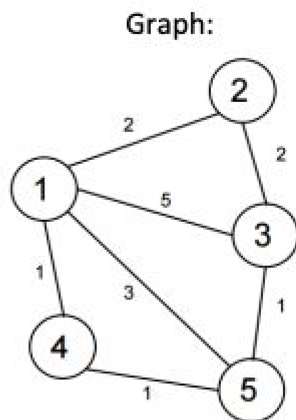
b) Dikstra’s algorithm computes an incorrect result when there is a positive self edge in a graph.
NEVER -- Dijkstra’s algorithm can always handle a positive self edge because that edge will never be a shorter path to the vertex than could already be found by going straight to the vertex itself.

c) If a weighted undirected graph has all unique edge weights, Prim’s and Kruskal’s algorithms will return the same result.
ALWAYS -- Prim’s and Kruskal’s both find a MST, and a graph with unique edge weights always has exactly one valid MST.

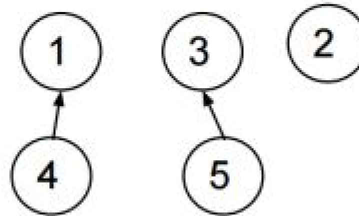
d) BFS will visit every vertex in a graph.
SOMETIMES -- Only in a connected graph is BFS guaranteed to visit every vertex. Otherwise, it’s possible that BFS will not be able to reach an isolated “island” of vertices.

4. MSTs & Disjoint Sets

You’re performing Kruskal’s algorithm for finding an MST on the following undirected weighted graph. After consider 2 edges and adding them to your MST, you have the following up-trees to represent the disjoint sets:



Up-Trees:



a) Give the array representation of the up-trees as specified for the WeightedQuickUnion data structure in lecture. Let each element with value x be stored in index x - 1 in the array.

```
[-2, -1, -2, 0, 2]
```

b) Suppose you have access to a variable arr that stores the up-trees as used in your part (a), and access to a variable map that stores a mapping from elements (of type T) to their corresponding indices in arr. Fill in the start of the following find method with pseudocode to perform the find operation without the path compression optimization, for the WeightedQuickUnion structure.

```
int find(T element) {
      int index = map.get(element);
      while (arr[index] >= 0) {
            index = arr[index];
      }
      return index;
}
```

c) Now, fill in the start of the following union method using your implementation of find in the previous problem, again for WeightedQuickUnion. You may assume that you still have access to arr and map.

```
void union(T x, T y) {
      int xRoot = find(x);
      int yRoot = find(y);
      if (xRoot != yRoot) {
            if (arr[xRoot] < arr[yRoot]) {
                  arr[xRoot] += arr[yRoot];
```

```
                arr[yRoot] = xRoot;
            } else {
                arr[yRoot] += arr[xRoot];
                arr[xRoot] = yRoot;
            }
        }
    }
```

d) What is the next edge in the graph that will be added to the MST by Kruskal's algorithm?
   The next edge will be the edge between 4 and 5, with weight 1.

5. Sorting

5.1 Sorting Design Decisions

At Third Street Elementary School, students can bring in boxes of tissues on the first day of class to earn extra credit towards their first assignment. At the end of the day, Miss Grotkey tallies up how many boxes each student had brought, and records the tallies next to each student's preferred name in the following order:

```
[
    (Gretchen, 22),
    (Spinelli, 18),
    (Randall, 17),
    (Vincent, 16),
    (Ashley, 16),

    .... 24 more students…

    (Erwin, 9),
    (Mikey, 4),
    (Anne, 4),
    (Bob, 3),
    (Gus, 0),
    (TJ, 0)
]
```

Assume that this ordering is rather peculiar; any trends that you notice will apply across the entire class of students, with only a couple out-of-place in the middle of the list. For the following questions, assume that you can represent the data as Objects that implement the Comparable interface with an appropriate compareTo() method customized to each problem.

a) Miss Grotkey needs to send the attendance office a list of her students in alphabetical order by preferred name; they want to make sure the roster they have matches up with the current students. What's a **good** sorting algorithm you can use to generate the roster for the attendance office? Defend your selection in 1-2 sentences, including an estimated runtime for the algorithm on this dataset.

   Quicksort or Merge sort, estimated runtime of n log(n). The list isn't already alphabetized in any way, so a sorting algorithm with a good average time complexity would be good.

b) Next, Miss Grotkey wants to order the class some snazzy name tags from NameBadge Inc. They give customers a 30% discount if they send a list of names sorted by length (e.g. shortest name at the beginning of the list, longest name at the end), since printing the name tags in that order saves material. What's the **worst** sorting algorithm you could use to sort this list for Name Badge Inc? Explain your choice in 1-2 sentences, including an estimated runtime for the algorithm on this data set.

   Insertion sort or Selection sort, with an estimated runtime of n^2. The list is in reverse order with respect to the length of names.

c) Finally, Miss Grotkey wants to sort the list in descending order of tissue boxes (e.g. students with the greatest number of boxes at the beginning of the list, students with zero boxes at the end) to make her grading easier. What's the **best** sorting algorithm you could use to sort this list? Explain your choice in 1-2 sentences, including an estimated runtime for the algorithm on this data set.

   Insertion sort, estimated runtime of n. The list is already sorted in descending order in terms of the amount of tissues each student brought.

5.2 Sorting Mechanics

a) Show the result of Selection Sort on the following array. When comparing elements, compare by number only -- the letters are just there to distinguish between elements that are considered equal to the sorting algorithm. Follow the pseudocode presented in lecture for Selection Sort, and if there is a tie for the smallest element, choose the first one found in the array.

   [2a, 5, 8a, 3, 2b, 8b, 1]

   Solution (shows each step, underlined element is the one that was just added to the sorted partition):

[1, 5, 8a, 3, 2b, 8b, 2a]
[1, 2b, 8a, 3, 5, 8b, 2a]
[1, 2b, 2a, 3, 5, 8b, 8a]
[1, 2b, 2a, 3, 5, 8b, 8a]
[1, 2b, 2a, 3, 5, 8b, 8a]
[1, 2b, 2a, 3, 5, 8b, 8a]
[1, 2b, 2a, 3, 5, 8b, 8a]

b) Show the result of Insertion Sort on the same array as in part (a), using the same comparison rules. Follow the pseudocode presented in lecture for Insertion Sort.

Solution (shows each step, underlined element is the one that was just added to the sorted partition):

[2a, 5, 8a, 3, 2b, 8b, 1]
[2a, 5, 8a, 3, 2b, 8b, 1]
[2a, 5, 8a, 3, 2b, 8b, 1]
[2a, 3, 5, 8a, 2b, 8b, 1]
[2a, 2b, 3, 5, 8a, 8b, 1]
[2a, 2b, 3, 5, 8a, 8b, 1]
[1, 2a, 2b, 3, 5, 8a, 8b]

c) When performing Selection Sort, what is the maximum number of times that any particular item could be moved in the array?

The maximum number of times a particular item could be moved is N times, because the Selection Sort algorithm makes at most N swaps to sort an array of N elements, and it's possible for the same item to be swapped all N times.

d) When performing the merge algorithm as part of the Merge Sort combine step , what is the maximum number of times that any particular item could be compared against another item while building up an array of size k?

The maximum number of times a particular item could be compared against another item is dictated by the size of the subarrays being merged to create the array of size k. At worst, an item in an array could be compared against every element in the array it is being merged with. Since the two subarrays combined to form an array of size k would each have size k/2, the maximum number of times an item could be compared would be k/2.