Name: Saqif Ahmed RCS ID: ahmeds7 @rpi.edu

CSCI 2300 — Introduction to Algorithms
Spring 2020 Exam 2 (April 23, 2020)

Overview

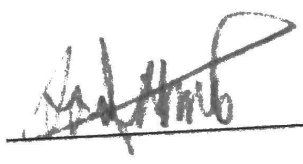
- This exam is open book(s), open notes; given that you are working remotely, you may use any and all of the posted course materials, including all previous questions and answers posted in the Discussion Forum.
- **Please do not search the Web for answers.** While we cannot stop you from doing so, such searching will likely lead you down the wrong path, answering the given questions using techniques we have not covered in this class. Therefore, some "correct" answers will not receive credit or partial credit, so please follow the instructions carefully and only use the techniques taught in this course.
- This exam is designed to take at most 120 minutes (for 50% extra time, the expected time is 180 minutes), but you can make use of the **full five hours from 5:00-9:59PM EDT**.
- Long answers are difficult to grade; **please be brief and exact in your answers**; the space provided should be sufficient for each question.
- **All work on this exam must be your own; do not even think of copying or communicating with others during or for 24 hours after the exam.**

Submitting your Exam Answers

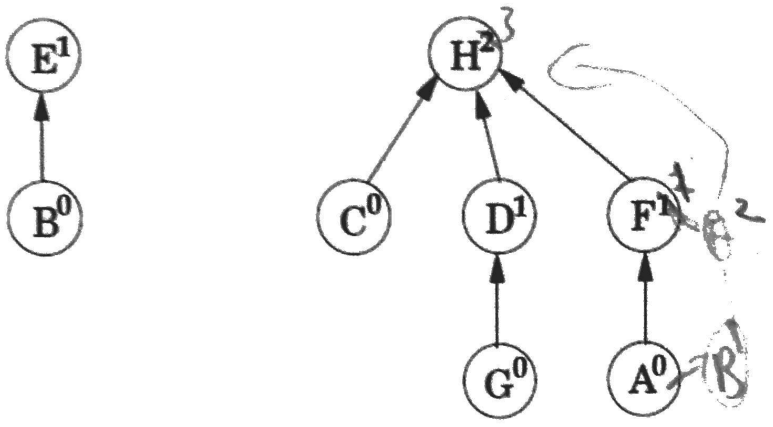
- You must submit your exam file(s) by 9:59PM EDT on Submitty.
- Please submit a single PDF file that includes this cover page and is called `upload.pdf`. This will help streamline the online grading process.
- If you are unable to submit everything as a single PDF file, submit files that have filenames that clearly describe which question(s) you are answering in each file. Do not submit a `README` or any other extraneous files.
- If you face any problems with the above instructions, please email `goldschmidt@gmail.com` directly with details.

Academic Integrity Confirmation

Please sign or indicate below to confirm that you will not copy or cheat on this exam, which in part means that you will not communicate with anyone under any circumstances about this exam:

Signature or Typed Name: 

For Questions 1-3, use the directed-tree representation of the two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$ (with rank values shown) below. And for each question, *start with the original representation depicted below*; in other words, any changes are *not* cumulative from one question to another.



1. **(2 POINTS)** Given the original directed-tree representation shown above, what is the output of the `find`(D) operation without path compression? Clearly circle the **best** answer.

   (a) π
   (b) E
   (c) set $\{E, H\}$
   (d) H
   (e) set $\{B, E\}$
   (f) set $\{A, C, D, F, G, H\}$
   (g) A, C, and G
   (h) E and F

2. **(2 POINTS)** Given the original directed-tree representation shown above, what is the output of the `find()` operation in the sequence `union`(A, B), `find`(C) without path compression? Clearly circle the **best** answer.

   (a) π
   (b) E
   (c) set $\{E, H\}$
   (d) H
   (e) set $\{B, E\}$
   (f) set $\{A, C, D, F, G, H\}$
   (g) A, C, and G
   (h) E and F

3. **(2 POINTS)** Given the original directed-tree representation shown above, what is the output of the last `find()` operation in the sequence `union(find`(C),`find`$(A))$, `find`(F), `find`(E), this time *with* path compression? Clearly circle the **best** answer.

   (a) π
   (b) E
   (c) set $\{E, H\}$
   (d) H
   (e) set $\{B, E\}$
   (f) set $\{A, C, D, F, G, H\}$
   (g) A, C, and G
   (h) E and F

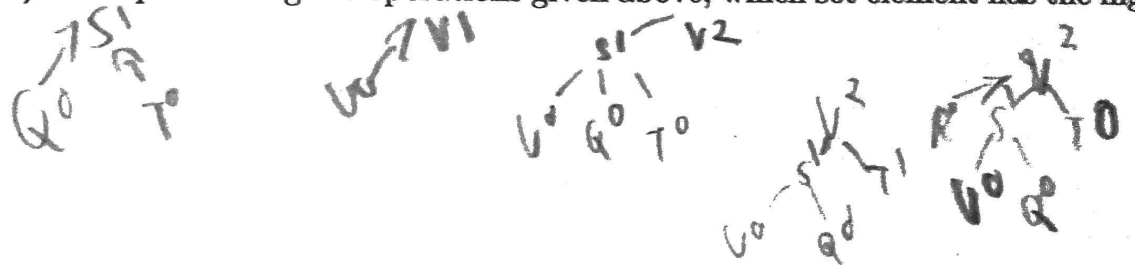
For Questions 4-6 below, first perform the following sequence of disjoint-sets operations, starting from singleton sets $\{Q\}$, $\{R\}$, $\{S\}$, $\{T\}$, $\{U\}$, $\{V\}$.

Use path compression for each operation, when applicable.

In case of any ties, use alphabetical order, i.e., always have the root closer to the beginning of the alphabet point to the root farther from the beginning of the alphabet (e.g., if there is a tie between S and T, have S point to T).

$$\texttt{union}(S, Q),\ \texttt{union}(T, Q),\ \texttt{union}(U, V),\ \texttt{union}(Q, U),\ \texttt{find}(T),\ \texttt{union}(U, R)$$

4. **(2 POINTS)** After performing the operations given above, which set element has the highest rank?



   (a) Q
   (b) R
   (c) S
   (d) T
   (e) U
   (f) V

5. **(2 POINTS)** After performing the operations given above, what is the highest numeric rank achieved?

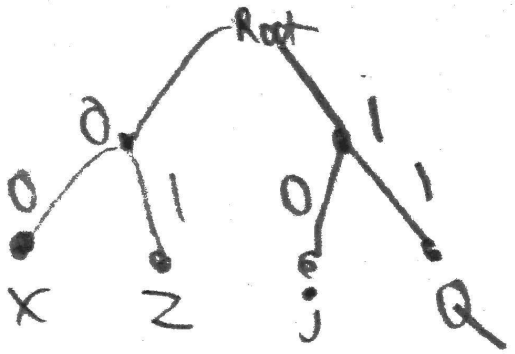
   (a) 0
   (b) 1
   (c) 2
   (d) 3
   (e) 4
   (f) 5

6. **(2 POINTS)** After performing the operations given above, how many elements have a rank of 0?

   (a) 0
   (b) 1
   (c) 2
   (d) 3
   (e) 4
   (f) 5

7. (12 POINTS) Given symbols j, q, x, and z occurring with frequencies 25%, 30%, 22%, and 23%, respectively, answers the questions below.

(a) **(5 POINTS)** Draw a valid tree that shows a prefix-free Huffman encoding for all four symbols.



(b) **(2 POINTS)** If a file contains exactly 1,000,000 symbols with the frequencies given above, how many bits are required to encode this file? Be exact.

2 bits per character. 2 million bits.

(c) **(5 POINTS)** Write an algorithm that converts the encoded file from part (b) **directly** into a newly encoded file in which all occurrences of symbol j are removed from the input. The conversion must be direct. Your two goals (to obtain full credit) are to minimize the resulting size of the newly encoded file and to minimize the time required to run your algorithm. Note that you can establish a new prefix-free Huffman encoding for the remaining three symbols if you would like.

input bit stream.
have two pointers i and j. j = i+1
Run through the bits:
    if i = 1 and j = 0:
        remove from list.
    increment i and j.
return bits.

8. **(12 POINTS)** Minimum spanning trees (MSTs) have many useful applications. Given undirected graph $G = (V, E)$ with edge weights $w_1, w_2, \ldots, w_{|E|}$, suppose we want to find in G a *light spanning tree* (*LST*), which has two constraints:

- Subset of vertices $U \subset V$ is given as an input, with all vertices in U required to be leaves of the identified LST.
- The LST must have a minimum total weight given the above constraint on U.

There might be other leaves in the LST, and the LST is not necessarily an MST.

Write a greedy algorithm for this problem that runs in $O(|E| \log |V|)$ time. Be sure to include a brief description of the inputs, the output, and the algorithm itself. Also be sure to show that your algorithm runs in $O(|E| \log |V|)$ time.

Note that you can make use of (call) Kruskal's or Prim's algorithm if you would like.

* We assume node and edge lookup is O(1)
* We run a modified version of Prim's.

Start from a random node not in U.
Run a modified version of Prim's algorithm.
Run Prims as normal until you take a node in U.
That node is now forbidden and cannot be searched from. I.E. No edges can be taken to and from that node anymore
Return the output of modified Prim.

Analysis: Because the definition of the LST is the MST with the constraints we use a greedy MST algo accounting for the constraints. Running Prim is $O(E \log V)$ and the modifications are O(1) so the overall algo is $O(E \log V)$
The output is also correct because we connect 1 edge only to each node in U.

9. **(12 POINTS)** Determine the runtime of the four divide-and-conquer algorithms described below. More specifically, write a recurrence relation for each, then express the runtime complexity using Big $O()$ notation. If you use the Master theorem, be sure to clearly indicate coefficients a, b, and d.

(a) **(3 POINTS)** Algorithm A solves problems of size n by dividing them into three subproblems, each of size $n/3$, recursively solving each subproblem, then combining results in $O(n^3)$ time.

$S(n) = 3 \cdot S(n/3) + O(n^3)$. Proof by Master's theorem: $S(n)$: $a=3, b=3, d=3$. $O(n^3) > \log_3(3)$

Runtime: $O(n^3)$

(b) **(3 POINTS)** Algorithm B solves problems of size n by dividing them into two subproblems, each of size $n/4$, recursively solving each subproblem, with no "conquer" step (similar to binary search).

$S(n) = 2(n/4) + O(1)$ Proof by Masters: $S(n)$: $a=2, b=4, d=0$ $\log_4 2 = \frac{1}{2}$ $\frac{1}{2} > 0$

Runtime: $O(\sqrt{n})$

(c) **(3 POINTS)** Algorithm C solves problems of size n by dividing them into four subproblems, each of size $n-2$, recursively solving each subproblem, then combining results in $O(n)$ time.

$S(n) = 4(n-2) + O(n)$

$n\left(\frac{4^{n+1}-1}{3}\right)$ ignore constants

is a sum

$n \cdot \sum_{i=0}^{n-2} 4^i \rightarrow n \cdot \left(\frac{4^{n-2+1}-1}{4-1}\right)$

runtime: $O(n \cdot 4^n)$

(d) **(3 POINTS)** Algorithm D solves problems of size n by dividing them into nine subproblems, each of size $n/3$, recursively solving each subproblem, then combining results in $O(\log_3 n)$ time.

Master's theorem:

$S(n) = 9(n/3) + O(\log_3 n)$ $a=9, b=3, d=<1$

$\log_3(a) = 2$ $O(n^2) + O(\log_3 n)$

$n^2 > \log_3 n$

Runtime: $O(n^2)$

10. **(16 POINTS)** Given unsorted array $Z[1 \ldots n]$ of n distinct integers, your goal is to develop a divide-and-conquer algorithm that re-orders the array to follow a *zigzag* order. Here, a zigzag order is one in which the array values go down, then up, then down, then up, etc.

More specifically, for even i, we have $Z[i] < Z[i+1]$; and for odd i, we have $Z[i] > Z[i+1]$.

Your divide-and-conquer algorithm must run in linear time. Be sure to include a brief description of the inputs, the output, and the algorithm itself. Also be sure to show/confirm that your algorithm runs in linear time.

input the unsorted array.

keep splitting the array until you have size 1 sub arrays.

look at the odd indexes.

Merge them such that they are greater than the element before it ($[i-1]$) and less than the element after it ($[i+1]$).

Continue until you have the whole array.

Return.

Analysis: the algo is linear $O(n)$ because we split the array once and merge linearly.

11. **(18 POINTS)** Suppose two teams C and L play games against one another in a series until one of the teams wins n games (e.g., $n = 4$ for a seven-game series). Assume that both teams are equally competent such that each has a 50% chance of winning any particular game.

Suppose that they have already played $m = i + j$ games, of which C has won i games and L has won j games.

Using dynamic programming, write an efficient algorithm to compute the probability that C will go on to win the series (i.e., reach n wins).

As an example, if $i = n - 1$ and $j = n - 3$, then the probability that C will win the series is $1/2 + 1/4 + 1/8 = 7/8$ since C must win *any* one of the next three games (whereas L must win *all* of the next three games).

For your dynamic programming algorithm, be sure to include a description of the subproblems, the recurrence relation, and the runtime of your algorithm.

define a subproblem: figure out the probability from winning from a certain i where $0 < i < n$ and j ($0 < j < n$). where i corresponds to C and j to L. We solve this from [0][0] to [n][n].

create Table[n][n]

Input base value:

for 1 to n:

Table[i][i] = 0.5.

For the other all positions [i->n][j->n]:

Table[i][j] = $\frac{1}{2}$(Table[i-1][j] + Table[i][j-1])

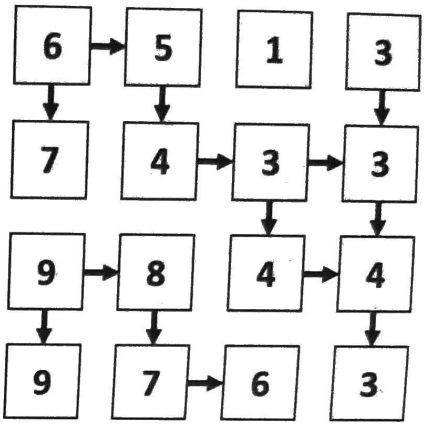
return Table[n][n]

Runtime is $O(n^2)$ for n^2 subproblems. Looking from the table is constant time. So overall it's $O(n^2)$

12. **(18 POINTS)** You are given a square $n \times n$ matrix containing positive integers in the range $[1, m]$ (endpoints included) that represent elevation levels across a vast area of treacherous land called *Summerarchia*. Assume $m > 9$.

Your goal is to write a dynamic programming algorithm to find a path through Summerarchia that maximizes the length of your path, but to form a path, there are some rules you must follow. From any given cell in the matrix, you can move right or you can move down, but only if the adjacent elevation values differ by at most 1 (so the elevation could stay the same).

An example matrix is shown to the right and has two solutions of maximum length 7, i.e., $6-5-4-3-4-4-3$ and $6-5-4-3-3-4-3$.



For your dynamic programming algorithm, be sure to include a description of the subproblems, the recurrence relation, and the runtime of your algorithm.

And note that your algorithm only needs to find one solution.

Define subproblem: for any spot in the matrix [i] [j], assuming it is a valid move, we need to take the maxPath: max_path(Matrix[i-1][j], Matrix[i][j-1]). Otherwise we just take the current path. We can store the subproblem answer in a table.

The algorithm is as follows:

Define a table[n][n] that holds our subproblem answers.

Run through every spot in the matrix [i->n] [j->n]:

```
if Matrix[i][j].is_valid():
    Table[i][j] = max_path(Table[i-1][j], Table[i][j-1])
else:
    Table[i][j] = current_path()
return Table[n][n]
```

Analysis: Algo is $O(n^2)$ because we solve n^2 subproblems and Table look up is constant. The is_valid() method checks to see if it can be reached by the previous nodes. at [n][n] we store one of the max paths.