Name: Jae Park RCS ID: parkj23 @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

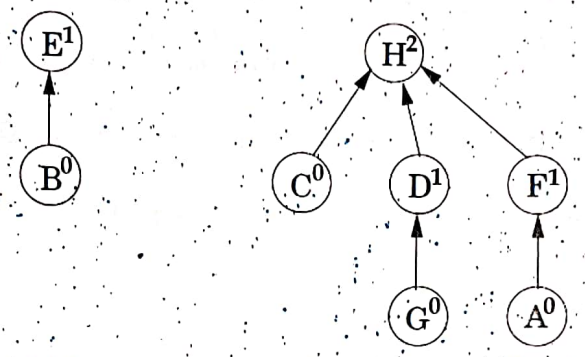
- You must submit your exam file(s) by 9:59PM EDT on Submitty.
- Please submit a single PDF file that includes this cover page and is called `upload.pdf`. This will help streamline the online grading process.
- If you are unable to submit everything as a single PDF file, submit files that have filenames 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: Jae Park

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

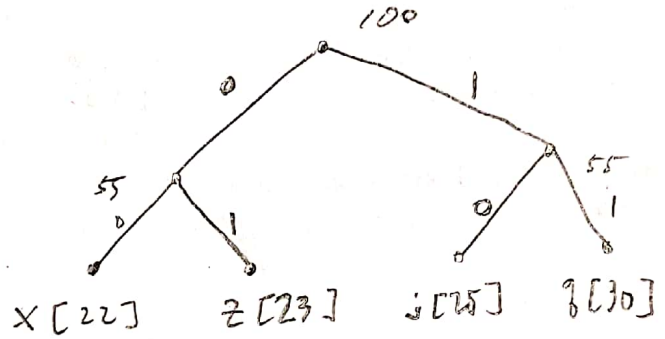
6. **(2 POINTS)** After performing the operations given above, how many elements have a rank of 0?

(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.

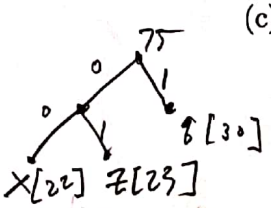
x	00
z	01
j	10
q	11



(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.

1,000,000 symbols × 2 bits per symbol

= 2,000,000.



x	00
z	01
q	1

(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.

String symbol = two characters read from the start of the file.

```
void function(symbol) {
    if(symbol != null)
        if (symbol = "00") //x
            don't do anything // x is still '00'
        if(symbol = "01") //z
            do nothing // z encoding is still '01'
        if(symbol = "10") //j
            get rid of the 'symbol' from the file.
        if (symbol = "11") //q
            change it to "1" // q: '11' → '1'
symbol = read two more characters;
function(symbol);
```

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.

Basically we use our graph with all the n nodes and the only edges connected to it.

Input: $G : (V - U, E_{new})$ where

$$E_{new} = \{(a,b) \mid a,b \in V-U \cap (a,b) \in E\}$$

output: LST

<algorithm>

create a graph $G'(V-U=V', E_{new})$.

② run kruskal's on $G'(V', E_{new}) = \underline{mst}$

if ($\underline{mst}$ does not exist)
    no LST.

else{ // if mst exists for G'

① $E' = \{(a,b) \mid a \in U, b \notin U\}$ // all the edges connected to leaves

② for($u : U$){
        makeset(u) can sort E' by the edge weights.

③ for ($(a,b) : E$) // this will be sorted.
        if find(a) ≠ find(b):
            add edge(a,b) to $\underline{mst}$
            union(a,b)

return mst; // by now mst will be LST.

}

④ is just running Kruskal's algorithm ($E \log V$) on a smaller graph.

① takes $|U|$ because we look at every node in U.
$|U| < |V| < |E|$.

② part of the Kruskal's algorithm.

③ if find(a) ≠ find(b), in whole looks at edge set E' and takes $O(E \log U)$

$\Rightarrow E \log V > E \log U > |U|$

$\Rightarrow O(E \log V)$

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.

pg.49 (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. $O(\log^p k)$

$T(n) = O(n^3) + 3T(n/3)$

$a = 3$
$b = 3$
$k = 3$
$p = 0$

$\therefore a < b^k$

$\therefore T_n = O(n^3 \cdot \log n^0)$

$= O(n^3)$

pg.49 (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).

$T(n) = 2T(n/4) + O(0)$

$a = 2$
$b = 4$
$d = 0$

$d < \log_b a$

$\therefore T(n) = O(n^{\log_b a})$

$= O(n^{\log_4 2})$

$= O(n^{\frac{1}{2}})$.

pg52 (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.

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

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

$\Rightarrow T(n) = O(n) + 4(O(n-2)) + 16(T-4)$

$\Rightarrow T(n) = 4^3 T(n-3 \cdot 2) + 4^2 O(n-2 \cdot 2) + 4O(n-2) + O(n)$

$= \sum_{i=0}^{n} 4^i O(n-3i) + C$ (where C is constant)

$= O(n \sum_{i=0}^{n/2} 4i)$

$= O(n \cdot 4^{n/2})$

pg47 (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.

$T(n) = 9T(n/3) + O(\log_3 n)$

$T(n/3) = 9(T(n/9)) + O(\log_3 \frac{n}{3})$

$a = 9$
$b = 3$
$k = 0$ from the master theorem,

then, $a > b^k$

so, $T(n) = O(n^{\log_b a}) = O(n^2)$.

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.

Until an array instance has size over 2, we divide it into two subarrays: $a[1 \ldots n] = a[1 \ldots m] + a[m \ldots n]$ where $m = \frac{(n-1+1)}{2}$, and check if the subarrays meet the conditions.

<pseudo>

```
zipsort(a[], start, end) {   // start initially 1, end initially n.
    if(start+1=end) // array size 2 ⇒ base case
        if(a[start] < a[end])
            swap a[start] and a[end]
        return; // end of the lowest base case
    m = ((end-start)+1)/2;
    zipsort(a[], start, m)
    zipsort(a[], m+1, end)
    if(a[m] > a[m+1]) // m is always even, so (m+1) is odd
        swap a[m] and a[m+1]
    return;
}
```

The recurrence relationship is given:

$T(n) = 2T(n/2) + O(C)$ where C is some constant time for aggregating ($\simeq O(1)$)

$= O(n^{\log_2 2}) = O(n)$.

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 $W(i,j)$ as the probability that A will win if A needs i more wins and j wins for B to win the series.

The subproblem then becomes computing the odds of winning the series with $7-(i+j)$ games left to play.

the recurrence relation is

$$W(i,j) = \frac{1}{2} W(i, j-1) + \frac{1}{2} W(i-1, j)$$

where $\frac{1}{2}$ represents the probability of each team winning, like the binomial distribution.

runtime. when $n=4$, i can range from 0 to 4 and j could also range from 0 to 4.

$\max(i)$ and $\max(j)$ is always floor of $\frac{n}{2}$ ($\lceil \frac{n}{2} \rceil$).

Imagine we employ $i \times j$ matrix. We have

$i \times j = \left(\lceil \frac{n}{2} \rceil\right)^2$ number of cells to compute in a matrix.

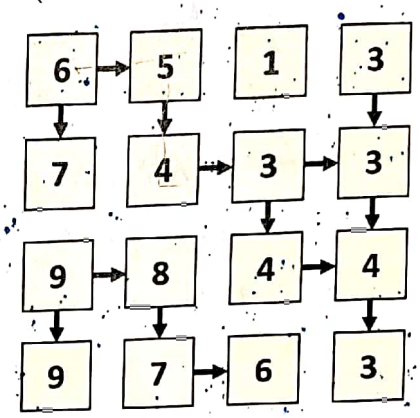
$O(\lceil \frac{n}{2} \rceil^2) \approx O(\frac{n^2}{2^2}) \approx O(n^2)$.

$\therefore O(n^2)$.

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.

From a cell, we can make two moves: right down

The subproblem can be defined as: from a cell, could you have gotten there from the cell left of you or above you. Then the longest possible path to that cell is same as examining the longest possible path to the cell left of it or above it.

define $c(i,j)$ as the longest path to (i,j)

recurrence $c(i,j) =$

$$\begin{cases} -1 & \text{if the cells next to you don't have a difference of 1 or less} \\ \max(c(i-1,j), c(i,j-1)) + 1 & \text{if both left and up paths can be taken} \\ c(i-1,j)+1 & \text{if only left cell has a valid path} \\ c(i,j-1)+1 & \text{if only the cell above can have a valid path} \end{cases}$$

Finally, from the bottom up approach, if there is any valid paths, return the path.
The runtime would be $O(n^2)$ because each value of $C(i,j)$ is computed at once, and there are $n * n = n^2$ cells.