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

⋈ ⋈ ⋈ CSCI 2300 — Introduction to Algorithms ⋈ ⋈ ⋈
Spring 2020 Final Exam (May 6, 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 seven hours from 5:00-11:59PM 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 about the exam during or for 24 hours after the exam.**
- Once we have graded your final exam, solutions will be posted along with final course grades in Rainbow Grades. The grade inquiry window for the final exam will be 24 hours, after which final course grades will be posted in SIS. If need be, contact `goldschmidt@gmail.com` directly after that window is closed.

Submitting your Exam Answers

- You must submit your exam file(s) by 11:59PM 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, 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: Saait Ahmed

Failure to submit this page will result in a grade of 0 on the final exam.

1. **(8 POINTS)** For each question below, compare the given pairs of functions $f(n)$ and $g(n)$. In each case, determine how the two functions compare to one another using $O()$, $\Omega()$, or $\Theta()$ as n grows very large.

   For each, clearly circle the **best** answer. (No partial credit will be awarded.)

   (a) $f(n) = n!$ and $g(n) = 2^n$

   i. $f = O(g)$
   ii. $f = \Omega(g)$
   iii. $f = \Theta(g)$
   iv. None of the above

   (b) $f(n) = 2^{n+1}$ and $g(n) = 2^n$

   i. $f = O(g)$
   ii. $f = \Omega(g)$
   iii. $f = \Theta(g)$
   iv. None of the above

   (c) $f(n) = 5^{\log_2 n}$ and $g(n) = n^{1/2}$

   i. $f = O(g)$
   ii. $f = \Omega(g)$
   iii. $f = \Theta(g)$
   iv. None of the above

   (d) $f(n) = n2^n$ and $g(n) = 3^n$

   i. $f = O(g)$
   ii. $f = \Omega(g)$
   iii. $f = \Theta(g)$
   iv. None of the above

2. **(6 POINTS)** We want to describe how many times an exclamation mark (!) is printed by the pseudocode below.

```
function f(n)
{
  if n > 1 then
  {
    print "!"
    f(n/2)
    f(n/2)
  }
}
```

(a) Define a recurrence relation $T(n)$ for the given pseudocode that represents the number of times an exclamation mark is printed out.

$T(n) = 2 \cdot T(n/2) + 1$

(b) Solve the recurrence relation from part (a). In other words, remove any and all recursive $T()$ terms from the right-hand side.

$2 \cdot T(n/2) \rightarrow$ tree each layer has $2^{\text{layer depth}}$ nodes, $\log_2(n)$ layer, sum up layer

$T(n) = \sum_{i=1}^{\log(n)} 2^{i-1} \Rightarrow 2^{\log(n)} - 1$

(c) If `f(n/4)` replaced each `f(n/2)` call in the given pseudocode, again define relation $T(n)$ for the number of times an exclamation mark is printed out. As with part (b) above, remove any and all recursive $T()$ terms from the right-hand side.

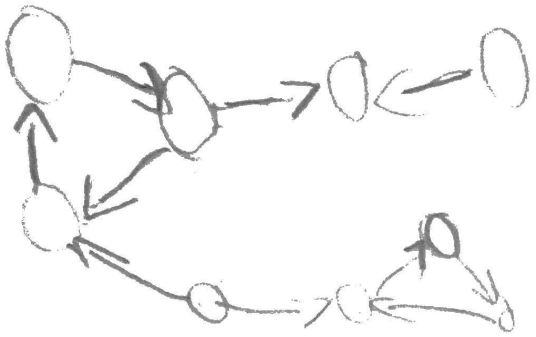
$T(n) = 2T(n/4) + 1$ depth of tree is $\log_4(n)$

Thus

$2^{\log_4(n)} - 1$

3. **(6 POINTS)** Draw a directed graph with two strongly connected components and at least six vertices for which it is impossible to add one edge and make the graph strongly connected.

For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.



4. **(10 POINTS)** Given an undirected graph $G = (V, E)$, let $V_{odd} \subseteq V$ contain all vertices with odd degree. Show that we can always find a way to pair up all vertices of V_{odd} so that the paths between each such pair of vertices have no edges in common. Note that these paths may share vertices (but not edges).

For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.

Proof by induction: Base case : 

We assume that in a graph there exists a set of unique pairs for the pairs of odd vertices.

If you add a vertex turning an odd node to even. Lets say Path(A,B) existed and you added some node P. The new path is the Path(A,P). This doesn't conflict as it's a new edge.

If you add a vertex making an even node odd then a new pair of odd nodes is created and you can just pair them together.

If you add an edge any newly turned even nodes are removed and if it creates 2 odd vertices then those can pair along the new edge.

Thus as we add edges and nodes the original claim is true.

5. **(8 POINTS)** Professor F. Lake claims that if $\{u, v\}$ is an edge in an undirected graph and during a DFS $\texttt{post}(u) < \texttt{post}(v)$, then vertex v is an *ancestor* of vertex u in the DFS tree. Either prove this claim to be true or show that this claim is not possible.

For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.

Claim is true.

Direct Proof:

We look at the definition of DFS. When exploring down the path, once we stop exploring we move up. If $\{u, v\}$ is an edge and the post(u) < post(v) this means that we went up at u before V. meaning exploring down we went from V→u or prev(v) < prev(u). Thus V is the ancestor.

6. **(8 POINTS)** What is the ~~maximum~~ minimum number of colors needed to color an undirected graph with exactly one odd-length cycle? Remember that to color a graph is to assign colors to its vertices such that adjacent vertices do not have the same color.

In your answer, also describe *how* you would color such a graph.

For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.

You need at 3 for the odd cycle

Maximum number of colors needed: 3

Describe how to achieve this solution for any graph $G = (V, E)$:

Start from a node and color it 1.

Search every node starting from the start node

At each uncolored node check the color of adjacent nodes.

if they are all colored and the same, color current node neighbor_color + 1.

if they are all colored but colored differently color it number_of_unique_neighbor_colors + 1.

otherwise alternate between ancestor color +1 and ancestor color -1

Runtime: $O(V \cdot E)$, each node we check all the edges.

7. **(8 POINTS)** When multiple shortest paths exist in a given graph, Dijkstra's shortest path algorithm arbitrarily finds one of these shortest paths. In such cases, the most convenient of these paths is often the path with the fewest number of edges (e.g., fewest number of layovers for a series of flights).

In this problem, you are asked to extend Dijkstra's algorithm to also calculate the fewest number of edges in each shortest path from a start vertex s. More specifically, for each vertex u in a given graph, define:

$$\texttt{best}[u] = \text{minimum number of edges in a shortest path from } s \text{ to } u$$

Rewrite Dijkstra's algorithm (e.g., from Figure 4.8 of DPV) with the above addition. Be sure to define the inputs, the outputs, the modified pseudocode, and the runtime of your algorithm. Assume that `makequeue()`, `deletemin()`, and `decreasekey()` operations are already given.

For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.

Input: Graph $G=(V,E)$ and edge lengths L and start s.

Output: dist(u), the set of distances from s to all $u \in V$.
best, the minimum number of edges in a shortest path from s to u.

```
best[V]
for all u∈V:
    dist(u) = ∞
    prev(u) = null
    var(w) = ∞ // holds number of edges traversed
dist(s) = 0; var(s) = 0
H = makequeue(V)
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
        if dist(v) > dist(u) + l(u,v):
            dist(v) = dist(u) + L(u,v)
            prev(v) = u
            decreasekey(H,v)
        if var(v) > var(u) + 1:
            best[v] = var(u) + 1
return dist, best
```

Runtime: With a heap Dijkstra is $O((V+E)\log V)$. Our additions are $O(1)$. Thus the runtime is $O((V+E)\log V)$

8. **(8 POINTS)** Given a connected undirected graph $G = (V, E)$ as input, describe a linear-time $O(|V|)$ algorithm to determine if there is an edge that can be removed from G that still leaves G connected. Your algorithm merely needs to output either true or false (so do *not* attempt to identify the specific edge).

In your answer, be sure to show the input, the output, and the runtime of your algorithm, as well as a brief description of the algorithm itself.

For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.

We run a modified breadth first search.

```
let Q be a queue
let the start node S be discovered
    Q.insert(S)
While Q is not empty:
  V = Q.top()
    for all edges {V, W} in V.adjacent():
      if W is discovered:
        return true
      else:
        W is dicovered
        W.parent = V
        Q.insert(W)
    return false
```

Output is either true or false. If any node is found that means there were two ways to get to it so we can delete an edge.

Runtime: this is just BFS so $O(V+E)$

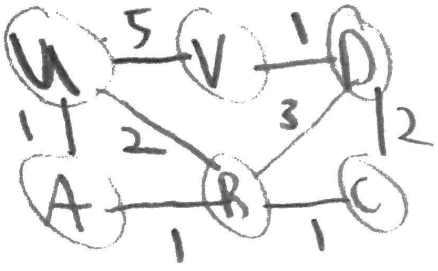
* Prof said it could be $O(V+E)$

9. **(12 POINTS)** Professor F. Lake has two new claims, shown below. For each claim, either prove the claim to be true or disprove the claim by describing a simple counterexample or contradiction.

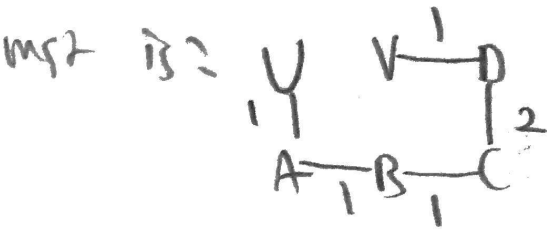
For all graph-related questions, no self-loops or multi-edges are allowed. In other words, all edges have distinct endpoint vertices.

(a) The shortest path between two vertices u and v is always part of a minimum spanning tree.

False (counter example: shortest path is (U, V)



MST is:



(b) First, for any $m > 0$, define an *m-path* to be a lightweight path with edge weights $w_i < m$. The claim is that if graph $G = (V, E)$ contains an m-path from vertex s to vertex t, then every minimum spanning tree of G must also contain an m-path from vertex s to vertex t.

This is true:

Direct Proof:

Imagine an m-path exists from s→t. If the M path is taken by the MST then clearly an m path exists. if the Mst does not take the M path that means that all the edges taken, individually, have better edge weights than the m path. Better is smaller.

However this is also an M path because if b < m and a < b then a < m.

Thus if a m-path exists then the MST will always have some m-path.

10. **(18 POINTS)** Given a rectangular sheet of metal with dimensions $H \times V$, where H and V are both positive integers, we are also given a list of n products that can be made from the sheet metal. For each product $i = 1, 2, \ldots, n$, we know that an $h_i \times v_i$ rectangle is required to make the product and that the selling price for the product is c_i. For all i, note that h_i, v_i, and c_i are all positive integers with $h_i \leq H$ and $v_i \leq V$.

The machine used to cut the sheet metal is limited in that it can only cut a rectangular piece into two smaller pieces by cutting either horizontally or vertically.

Use dynamic programming to develop an algorithm that determines the set of products we can make that will maximize the sum of the costs c_i. Note that we can make as many copies of the same product as we would like. And we do not need to make all n possible products.

Be sure to describe the subproblems, the recurrence relation, the order in which subproblems must be solved, how a solution is determined, and the runtime of your algorithm.

This is the knapsack problem where weight is area left.

the subproblem is $K(h,v)$, the maximum value achievable with area $h \times V$. If we do this till $K(H,V)$ we solve the problem. We have to modify K again to account for the limited cutting such that if we cut vertically it's different from cutting horizontally. So we have the relation of $K(h_i,V_i) = \max\{K(h_{i-1},V_i)+c_i,\ K(h_i,V_{i-1})+c_i\}$

Procedure :

$K(0,0)=0$

for ($h \leq H$ and $v \leq V$; h++ v++) : // h_i, V_i means ith object of n

$K(h_i,V_i) = \max\{K(h_{i-1},V_i)+c_i,\ K(h_i,V_{i-1})+c_i\}$

return $K(h,V)$

Runtime at most we check H against V against n objects

Thus runtime is $O(HVn)$. This is because we treat H and V as weights like in knapsack.

11. **(8 POINTS)** We define a new problem called the Intersect Every Set (IES) Problem as follows. Given a family of n sets $\{S_1, S_2, \ldots, S_n\}$ and a budget b, we wish to find a set I of size $s \leq b$ that intersects every set S_i, if such an I exists. More specifically, we wish to find a set I such that $I \cap S_i \neq \emptyset$ for all i.

Show that the IES Problem is in class NP-complete. As a hint, consider modeling this problem using a graph and remember that to show a problem is in class NP-complete, we must show a polynomial-time reduction from a known NP-complete problem to the new problem.

We represent the problem as finding a Rudrata path each vertex is representing the set S_i. Each edge represents a single element intersection. If set S_1 and S_2 share 1 value they get 1 edge, if they share 2 they get 1 distinct edge, etc. etc. Now we have the Rudrata path, finding a path that touches each node exactly once. Specifically we want to minimum rudrata path or just check if the rudrata path size is less than b. If there is no rudrata then you take one element from each of the outlier nodes and put it into I, and you take the edges currently traversed, each representing a value between those sets and put into I. If you find a Rudrata path then put the edges traversed, each representing a value into I. Check if it is $\leq b$. Return I.

Because we can reduce Rudrata to IES, it is NP complete.