Name: Jae Park RCS ID: parkj23 @rpi.edu

# ⋈ ⋈ ⋈ CSCI 2300 — Introduction to Algorithms ⋈ ⋈ ⋈
## Spring 2020 Final Exam (May 6,. 2020)

### Overview

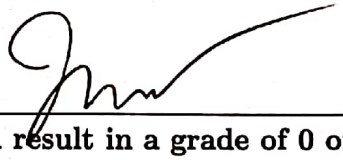
- This exam is open book(s), open notes; given that you are working remotely, you may use any and all of the posted course materials, including all previous questions and answers posted in the Discussion Forum.
- **Please do not search the Web for answers.** While we cannot stop you from doing so, such 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:** 

**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 $g(n) = n^{1/2}$

   $5^{\log_2 n} = \frac{5}{2} \cdot (2^{\log_2 n}) = \frac{5}{2} n$

   $\frac{5}{2} n > n^{1/2}$

   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)
  }
}
```

10 5 2 1 0.
1 1 1 1 1

1 1 1.

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

$T(n) = 2T(\frac{n}{2}) + O(1)$

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

$T(n) = O(n)$

n c
n/2 n/2 2c
n/4 n/4 n/4 n/4 4c.

$2^{\log_2 n} \cdot c = cn$

$T(n) = c + 2c + 4c \cdots cn$

$d = 0$

$\log_b a = \log_2 2 = 1$

(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.

⇒ num in tree
$= \Delta t(n-1)$
$= 2n-1$
$= O(n)$

$T(n) = 2T(\frac{n}{4}) + O(1)$.

(pg.49) master: $aT(\frac{n}{b}) + O(n^d)$, $a=2, b=4, d=0$.

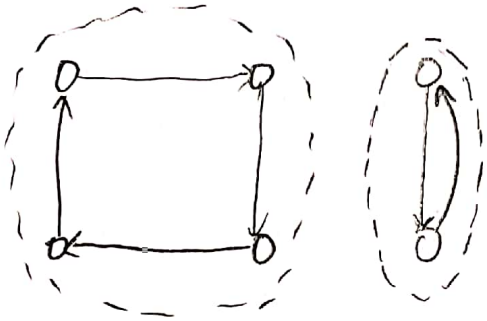
if $\log_b a > d$, $T(n) = (n^{\log_b a})$

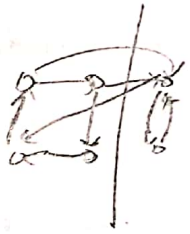
$= n^{\log_4 2} = n^{\frac{1}{2}}$

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

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.*





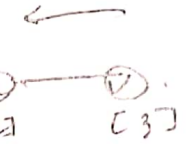
4. **(10 POINTS)** Given an undirected graph $G = (V, E)$, let $V_{odd} \subseteq V$ contain all vertices with odd degree. Show that we can always find a way to pair up all vertices of $V_{odd}$ so that the paths between each such pair of vertices have no edges in common. Note that these paths may share 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.*

We take advantage of finding the Euler path (DPV pg. 236, 237).
In order to satisfy the two conditions for there to exist a Euler path, (a) the graph is connected, (b) every vertex, with the exception of two vertices (start & final).

So, we take all vertices in Vodd and add some edges between them. The way to do this is, we arbitrarily take a pair of vertices of which there is no edge between them. Now, if we look at Vodd again, we have a graph that is connected and each vertex in it has even degrees (because we added an edge between pairs of odd-degree vertices). We know this graph satisfies the conditions to have a Euler path. So, we find such path. Now, we just get rid of the extra edges we added earlier.

From whatever remains in the Euler path after removing those edges, we can see that the edges that are left have their two ends being the vertices of odd degree. Each of these edges, therefore, gives a vertex-pairing without any repeated edges because we used Euler path.




5. **(8 POINTS)** Professor F. Lake claims that if $\{u, v\}$ is an edge in an undirected graph and during a DFS post$(u)$ < 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.*

(refer to DPV Pg. 89) During the explore( ), there are only three cases possible.

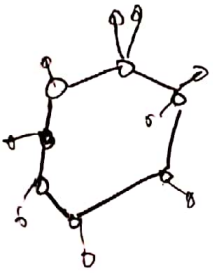
① pre(v) < pre(u) < post(u) < post(v) – normal case (tree/forward)

② pre(u) < pre(v) < post(v) < post(u) – back

③ pre(u) < post(u), pre(v) < post(v) – cross

② is unnecessary because post(v) < post(u). Case ③ cannot exist because it would mean that ⓤ and ⓥ are non-overlapping/crossed. Also, we must visit all the neighbors of v before marking it visited with a post[ ] number. Therefore, case ③ doesn't hold. However, ① can work and this shows a relationship between v and u, where v is an ancestor of u in the DFS tree. Thus, the statement can be true.

6. **(8 POINTS)** What is the maximum 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.*

min.
~~Maximum~~ number of colors needed: 3

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

So, inside our G, we have an odd-length cycle and the rest of the graph. The rest of the graph can be composed of even-length cycles and/or trees.

→ To color the odd-length cycle, we cycle back between red and blue, and the last remaining vertex will receive a new color, yellow, that will have a red vertex(start) and a blue vertex(end) as its neighbors.

→ To color any even-length cycle, we cycle back between red and blue, so two colors minimum.

→ Whatever that is left in the graph has to be a tree because by definition, a tree is a non-cycle graph. To color any trees, we need just two colors, each color for each height(level) of the tree.

Finally, max(3, 2, 2) = 3

So we need minimum 3 colors.

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 $G=(V,E)$ where $\forall\, e \in E$ is positive weight, start vertex $s$
output: best[u] for each $u \in V$,

<pseudocode>

```
for all u∈V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
best[u] = 0  for all u∈V.  // initially.

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)
            best(v) = best(u) + 1
            //prev(v) = u
            decreasekey(H, v)
        else if (dist(v) = dist(u) + l(u,v)):   // making a detour
            if best(v) < best(u) + 1:
                best(v) = best(u) + 1
                //prev(v) = u.
        else
            continue;
```

<runtime>

now that we don't need to keep track of which path we took, prev(v) = u is indeed unnecessary. So, if we get rid of the prev() operation and add best[] array computation, this in whole still yields constant time operation. The modified algorithm therefore still takes the same runtime as Dijkstra, $O(|V|^2)$, or $O(|V|\log|V| + E)$ with fibonacci heap.

|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.*

① input: graph G(V, E)

② output: True if ∃ e ∈ E that leaves G(V, E\{e}) still connected

False if such edge doesn't exist.

④ brief description:

First, we check if the graph has a cycle.
The simplest traversal of a graph (eg. recursive method) would still incur a linear time of O(|V|+|E|). However, we want linear of O(|V|).
However, if we look carefully, O(|V|) is hidden inside O(|V|+|E|). What I mean by that is, when traversing every single vertex, we go through |V| vertices and |V-1| edges.
By that point, if any cycles exists, then that means an removable edge e exists within the cycle. The time of checking if a cycle exists in a graph took |V|+|V-1| = 2|V|+c = O(n) where c is some constant. Notice that |V-1| can be significantly smaller than |E|, especially in a big graph.
However, if a cycle doesn't exist, then it means that the entire graph is a tree, and removing any edges will leave the graph unconnected.

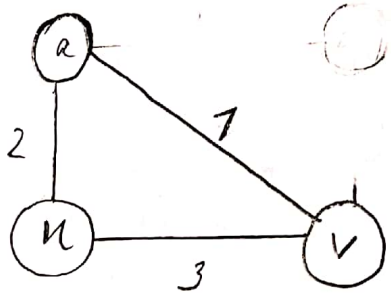
③ runtime: O(n) as discussed above.

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.

NO. counter example



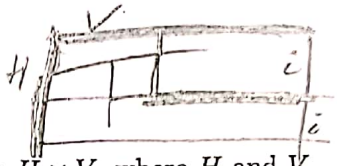

MST: (u, a)
(a, v)

shortest path: (u, v)

(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 $t$.

TRUE.

Assume for the sake of contradiction that there was an edge e along the path from s to t such that We > m (the weight of e exceeds m). Then one of the edges that comprise the original m-path must not have been included in the path. This is true, because if not, then we have an extra redundant edge between such edge and e.
However, removing edge e and adding this new edge can get us a better MST. Contradiction.




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.

① Subproblem: Given $1 \leq h_i \leq H$ and $1 \leq V_i \leq V$ ($h_i$ and $V_i$ are integers), let's define selling price of a rectangle sheet to be $C(h_i, V_i)$. Then we can find the subproblem as:

maximum $C(H, V)$ that can be acquired from the available rectangles. Such rectangles are that

$\Rightarrow$ max $(C(i, j))$ for all rectangles that $i = h_i$, $j = V_i$, if a product $h_i \times V_i$ exists, will yield a max price with all the available rectangles.

② recurrence:

$$C(h_i, V_i) = \max \text{ of } \begin{cases} C(h_i, V_i) \\ \max\left(C(j, V_i) + C(h_i - j, V_i)\right) \text{ for every } j : (1 \leq j \leq h_i) \\ \max\left(C(h_i, k) + C(h_i, V_i - k)\right) \text{ for every } k : (1 \leq k \leq V_i) \end{cases}$$

③ this way of subdiving the problem helps us access whether it earns more to just sell it by itself $C(h_i, V_i)$ or sell it with smaller pieces, smaller pieces in terms of each $h_i$ and $V_i$ ($H$ and $V$). So, bottom-up. Then the size of these smaller pieces increases as the computation goes on.

④ the final cost can be simply retrieved from $C(h_i, V_i)$ after the computation is done.

⑤ RUNTIME:

For $1 \le h_i \le H$ and $1 \le V_i \le V$, two cases. Either the rectangle itself can sell at the maximum price, or the not price given from recursion is higher. If the recursion is the case, then given by the inner max() functions in the recurrence relationship, the problem subdivides into $[(h_i - 1) + (V_i - 1)]$ different computation paths, and there are $(H * V)$ ways to divide up the original rectangle.

If you replace $h_i, V_i$ with index-less variables, $H, V$,

then we get

$$[(H-1)+(V-1)] \times (H*V)$$

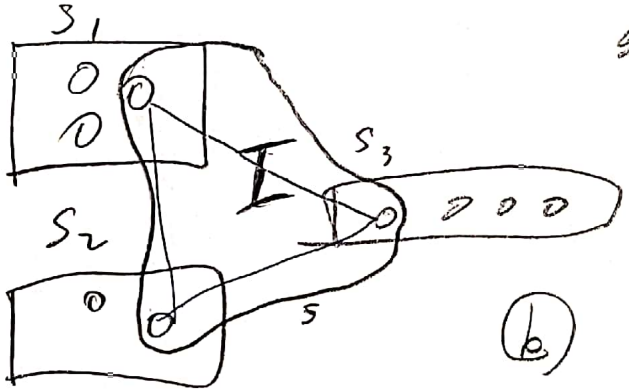
$$= [H+V-2] \times (HV)$$

$$= O(HV(H+V+c)) \quad \text{where } c \text{ is some constant}$$

$$= O(HV(H+V)+c') \quad \text{where } c' \text{ is some constant.}$$

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.




In chapter 5, we discussed the set-cover problem. This problem is ~~similar to~~ a subset of set-cover problem, a vertex-cover problem. We discussed the vertex-cover problem earlier in the lecture and lab.

So, this problem is essentially to find a vertex-cover of Graph $G=(V, E)$ where $V = S_1 \cup S_2 \cup \cdots \cup S_n$, with a condition that the size of vertices in the vertex-cover is at most $b$ ($s \leq b$).

Given by the NPV Index (pg. 252), it shows that independant set problem $\longrightarrow$ vertex cover problem.

Furthermore, from pg.243, we know that the independent set problem is in NP-complete.

Therefore, the vertex set cover problem, i.e., the Intersect Every Set(IES) problem is also in NP-complete.