CS161 Final Exam

Once you turn this page, your exam has officially started!
You have three (3) hours for this exam.

Instructions:

- This is a **timed**, **closed-book** exam.
- You must complete this exam within **180 minutes** of opening it.
- You may use two two-sided sheets of notes that you have prepared yourself. **You may not use any other notes, books, or online resources. You may not collaborate with others.**
- **If you have a question about the exam:** Try to figure out the answer the best you can, and clearly indicate on your exam that you had a question, and what you assumed the answer was.
- You may cite any result we have seen in lecture without proof, unless otherwise stated.
- This exam is printed two-sided. There are blank pages at the end for extra work. Do NOT tear off or unstaple the pages.
- **Please write your name at the top of all pages.**

Advice: If you get stuck on a problem, move on to the next one. Pay attention to how many points each problem is worth. Read the problems carefully.

Honor code: The following is a statement of the Stanford University Honor Code.

By signing your name below, you acknowledge that you have abided by the Stanford Honor Code while taking this exam.

Signature: ____________________

Name: ____________________

SUNetID: ____________________

1 Multiple Choices (26 pt.)

For the following problems, fill in the circle(s) for the correct choice(s). No explanation is needed.

1.1 (6 pt.) Looking Back :)

Which of the following is true? **Select all that apply.**

Ⓐ A divide and conquer algorithm can always use memoization to reduce runtime at the cost of increased memory usage.

Ⓑ A randomized algorithm's expected runtime will always be faster than or equal to its worst-case runtime.

Ⓒ For a problem that can be solved using a greedy algorithm, making any choice other than those of the algorithm leads to a non-optimal solution.

Ⓓ A top-down dynamic programming (DP) algorithm is similar to a bottom-up DP algorithm, except the bottom-up version stores the results of previously encountered sub-problems, while the top-down version does not.

SOLUTION:
B. (A) is false because the problem might not have overlapping sub-problems. (B) is true because expected runtime considers the expectation of the outcomes in the random parts of the algorithm, which would yield a runtime that lower-bounds the runtime when every random outcome is the worst outcome. (C) is false– multiple optimal solutions could exist, but greedy only finds one of them. (D) is false because all DP algorithms store the result of previous sub-problems.

1.2 (6 pt.) Hashing

Let U be a finite set and n be a positive integer. Let S be the set of all functions with domain U and range $\{0, 1, 2, \cdots, n-1\}$. Assume $|U|$ is much much larger than n. Suppose that we want to make a hash table with S as our hash family. Which of the following is true? **Select all that apply.**

Ⓐ S is a universal hash family.

Ⓑ When we choose a hash function $\mathcal{H}$ from S at random, there always exist two elements that would be hashed into the same bucket by $\mathcal{H}$.

Ⓒ It takes $O(|U|)$ bits to store a hash function in S.

Ⓓ Assume that we need to hash n elements in total and looking up each hash value takes constant time. In the hash table constructed using a random function from S, the operations Insert, Delete, and Search would run in expected $O(1)$ time.

SOLUTION:
ABD. (A) is true because S is the set of all hash functions from U to $\{0, 1, 2, \cdots, n-1\}$. (B) is true because $\mathcal{H}$'s domain is bigger than its range, so multiple elements will be mapped to

the same bucket; however, when we choose a hash function from S at random, the probability of collision for any two elements is $\leq \frac{1}{n}$, which is what we need. (C) is false because it takes $\Omega(|U| \log n)$ bits to store a hash function in S. (D) is true, since S is universal.

1.3 (4 pt.) Climb Stairs

Consider the CLIMB-STAIRS (CS) problem: there are n stairs to climb. Return the number of ways there are for a person standing at the bottom to climb to the top if they can climb either 1, 2, or 3 stairs at a time. What would be the recurrence relation for $\text{CS}(n)$, assuming that $n \geq 3$? **Select the single correct answer.**

(A) $\text{CS}(n) = \text{CS}(n-1) + \text{CS}(n-2) + \text{CS}(n-3)$

(B) $\text{CS}(n) = \max\{\text{CS}(n-1), \text{CS}(n-2), \text{CS}(n-3)\}$

(C) $\text{CS}(n) = \text{CS}(n-1) + \text{CS}(n-2) + \text{CS}(n-3) + 1$

(D) $\text{CS}(n) = \max\{\text{CS}(n-1), \text{CS}(n-2), \text{CS}(n-3)\} + 1$

SOLUTION:
A. # ways to stair n = # ways to stair n-1 (then take 1 step to stair n) + # ways to stair n-2 (then take 2 steps to stair n) + # ways to stair n-3 (then take 3 steps to stair n).

1.4 (6 pt.) Negative Edge Weights

Which of the following graph algorithms can return the correct result on a graph with potentially negative edge weights (but there are no negative cycles)? Assume that all four algorithms can take weighted graphs as valid inputs. **Select all that apply.**

(A) DFS for finding all vertices that are connected to a vertex in an undirected graph

(B) Dijkstra's algorithm for finding the shortest path from a vertex to all other vertices in a directed graph

(C) Floyd-Warshall for finding All-Pairs Shortest Paths in a directed graph

(D) Kruskal's algorithm for finding a Minimum Spanning Tree in an undirected graph

SOLUTION:
ACD. (A) is true because DFS does not take edge weights into account, but it does not need to do so to find a connected component. (B) is false–Dijkstra needs to be certain that the path weights from the source to the vertices it has marked as "sure" will not decrease in the future. (C) is true because FW will consider all edges, and update the weight of the paths with negative edges accordingly. (D) is true because Kruskal selects the edge weights in ascending order, no matter if they are negative.

1.5 (4 pt.) Flows and Cuts

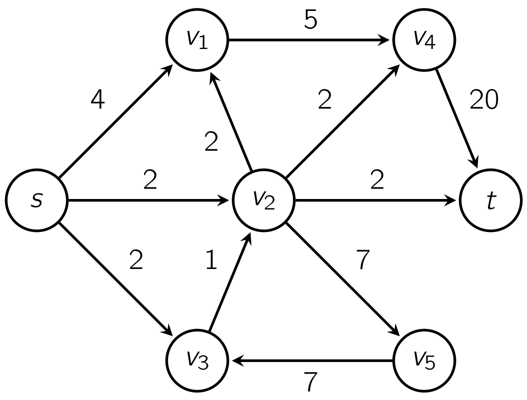
Consider a graph $G = (V, E)$ with a capacity c_e for every $e \in E$. Suppose there exists a flow from s to t (not necessarily the maximum) of value 5. What do we know about the size of the **minimum** s-t cut (denoted by $|C|$) in G? **Choose the single correct answer.**

(A) $|C| \leq 5$ (B) $|C| = 5$ (C) $|C| \geq 5$ (D) None of the above

SOLUTION:
C. The size of any cut $\geq$ of the value of any flow.

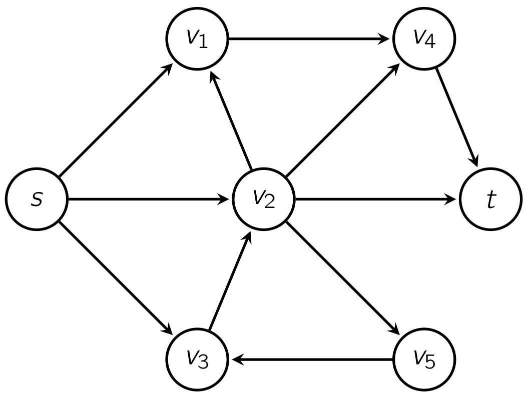
2 Short Answer (30 pt.)

2.1 (6 pt.)

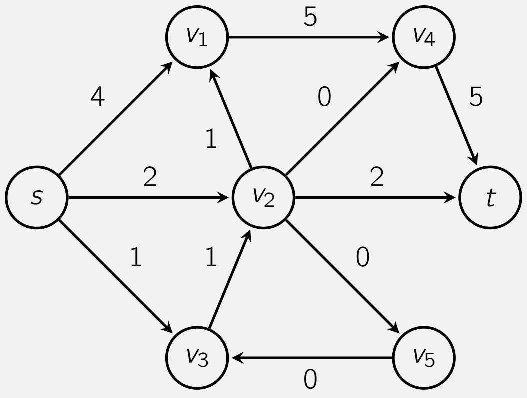


Above you see a graph with edge capacities indicated next to each edge. Find a maximum flow in the above graph from s to t. You may fill out the flow on every edge in the below picture.

[We are expecting: Fill in the graph below with a maximum flow.**]**



SOLUTION:
Here is one example. Any valid flow with value 7 is correct.



2.2 (6 pt.)

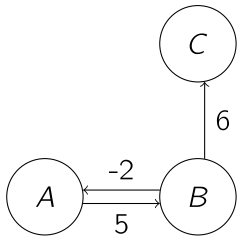
Suppose we are given an undirected, weighted, and connected graph G with n vertices and m edges. We wish to store the graph so that we can use Prim's algorithm to return the minimum spanning tree of the graph in the best possible asymptotic runtime (that is achievable by Prim's algorithm specifically). Do we want to use an adjacency list representation for our graph or an adjacency matrix representation? Justify your choice.

[We are expecting: A choice of representation and a short explanation**]**

SOLUTION:
We would want to use an adjacency list representation. The minimum spanning tree algorithm requires us to add every vertex to our tree and then iterate over its neighbors. Using an adjacency matrix would require us to iterate through the whole row/column associated with a vertex to find its neighbors, resulting in an $\Omega(n^2)$ runtime. On the other hand, an adjacency list would allow us to iterate just through the neighbors of each vertex and achieve the $O(n \log n + m)$ runtime. Since $m = O(n^2)$ is not a tight bound, this can be a strictly better runtime than $\Omega(n^2)$.

2.3 (6 pt.)

Suppose we have a directed graph with n vertices and m edges. You start at some specified vertex, and each time cross an edge going out of the current vertex. Each edge has an integer dollar amount attached, such that your earnings increase by that dollar amount if you traverse the edge (or decrease if the dollar amount is negative). If you traverse an edge multiple times, your earnings increase by the dollar amount multiple times. For example, in the graph below, if you start at A, then move to B, then move back to A, then move back to B, then move to C, you will earn $5 + (-2) + 5 + 6$ dollars.



You want to know whether your earnings are bounded (i.e., there is a path that earns the maximum amount of money, beyond which you cannot earn more). More specifically, you want an algorithm that can take in any graph and will return true if your earnings are bounded and false if there is the potential to earn unlimited money. Explain an algorithm to do this in $O(mn)$ time.

[We are expecting: An English description of your algorithm, likely referencing an algorithm from class, and a brief explanation of why it works**]**

SOLUTION:
You can run Bellman-Ford on a version of the graph where all the edge weights have been multiplied by -1. Then Bellman-Ford will determine whether there are negative cycles, which would translate to positive cycles in the original graph, which would give you unbounded rewards.

2.4 (6 pt.)

Suppose we have a directed acyclic graph with n vertices and m edges where each vertex has an associated integer score. We also care, however, about a vertex's descendantScore, which is the highest score of the vertex itself or of any of its descendants (i.e., the vertices it has a path to). Explain how to calculate the descendantScore of all the vertices in our graph in $O(n+m)$ time.

[We are expecting: Pseudocode OR a brief English description of your algorithm. You do not need to justify the runtime.**]**

SOLUTION:
There are a couple of similar ways to do this, but the simplest is through top-down dynamic programming. We can iterate through the vertices, keeping track of the scores for vertices that we have already calculated. When we encounter a vertex whose score we haven't calculated, we calculate it as the maximum of its own score and the descendantScores of its children, recursing as necessary to calculate the children's descendantScore.
It can also be implemented through a bottom-up dynamic programming search where we first calculate a topological sort on the graph and then iterate over the vertices in the reverse order.
We can also implement it through a modification of DFS, where we store the descendantScore at each node we encounter, and then calculate the descendantScore of a new vertex in terms of the descendantScores of its children.

2.5 (6 pt.)

Suppose for a given graph, we have access to a second graph representing the DAG of the strongly connected components of the first graph. We also have access to a function getSCC, which takes as input a vertex in the first graph and returns the vertex corresponding to its strongly connected component in the second graph. The function runs in $O(1)$ time. If there are b strongly connected components, devise an $O(b^2)$ algorithm that takes as input vertices u, v in the original graph and identifies whether there exists a path from u to v.

[We are expecting: Pseudocode OR a brief English description of your algorithm, as well as a brief justification of runtime. You do not need to justify the runtime of any algorithms from the lecture.**]**

SOLUTION:
We first access the strongly connected components of the two vertices. We can then run DFS or BFS from the strongly connected component of the first vertex and see if we reach the strongly connected component of the second vertex. This will run in $O(|V|+|E|)$ on the second graph, where $|V| = b$ and $|E| = O(b^2)$.

3 Greedy for Compensation (49 pt.)

3.1 (8 pt.)

Lucky recently discovered an opportunity in the psychology department, where students are paid to participate in studies. There are n ongoing studies numbered from 1 to n, with no new studies starting. Today is day 0, and each study i will end on day $d_i \geq 0$. Lucky is limited to participating in each study only once and has the flexibility to choose which day to participate, up to and including the study's end date. Each study requires only one day of participation, and Lucky can participate in only one study per day.

Payouts can differ between studies. Study i pays out $p_i \geq 0$ dollars. Lucky's goal is to maximize the total payout. Lucky has devised the following greedy strategy: every day, Lucky participates in the remaining study with the highest payout and adds that to his total. A pseudocode implementation of this strategy is shown below.

```
Algorithm 1: Lucky's algorithm
Input: A set of studies studies = $\{(p_i, d_i) \mid 1 \leq i \leq n\}$, where $p_i$ is the payout and $d_i$ is the end
       day for study $i$.
Output: Lucky's total payout.

payout ← 0
day ← 0
sortedStudies ← studies sorted by decreasing order of payout
for $(d, p)$ ∈ sortedStudies do
    if day ≤ $d$ then
        day ← day + 1
        payout ← payout + $p$
return payout
```

Prove that Lucky's strategy may not result in the maximum payout.

[We are expecting: A concise proof by counterexample which provides a list of (d_i, p_i) pairs on which the algorithm does not achieve maximum payout.**]**

SOLUTION:
As a counterexample, let $(d_1, p_1) = (1, 2)$ and $(d_2, p_2) = (0, 1)$. Lucky would participate in study 1 on day 0 and then be unable to participate in a study on day 1, leading to a total payout of 2. A higher payout of 3 is achievable by participating in study 2 on day 0 and study 1 on day 1.

3.2 (25 pt.)

Plucky suggests a fix to Lucky's algorithm. Plucky's strategy still processes studies in decreasing order of payout but schedules each at the last available day possible rather than the first. A pseudocode implementation of this strategy is shown below.

Algorithm 2: Plucky's algorithm

Input: A set of studies studies $= \{(p_i, d_i) \mid 1 \leq i \leq n\}$, where p_i is the payout and d_i is the end day for study i.

Output: Maximum payout from studies.

```
payout ← 0
studySchedule ← an array of length $\max\{d_1, \ldots, d_n\} + 1$ with all entries set to "available"
sortedStudies ← studies sorted by decreasing order of payout
for $(d, p)$ ∈ sortedStudies do
    for day = $d, d-1, \ldots, 0$ do
        if studySchedule[day] = "available" then
            studySchedule[day] ← "booked"
            payout ← payout + $p$
            break // from the inner for loop
return payout
```

Prove that Plucky's algorithm correctly returns the maximum payout Lucky can achieve.

[We are expecting: A formal proof of correctness.**]**

SOLUTION:
Suppose for contradiction that Plucky's algorithm fails to return the maximum payout. Then the schedule created by Plucky is suboptimal. Let S be the schedule created by Plucky and $S^\star$ be an optimal schedule which is consistent with S for the longest possible prefix of studies in sortedStudies. Let study i be the first study in *sortedStudies* at which S and $S^\star$ differ. One of the following three cases must occur.

- **S includes study i but $S^\star$ does not.** Consider the day at which study i is scheduled in S. Either $S^\star$ does not have study for this day, in which case it can be improved by adding study i, or it has a different study j which appears later in sortedStudies and can be replaced by i without reducing the total payout. In either case, there is a contradiction.
- **$S^\star$ includes study i but S does not.** Because S and $S^\star$ are consistent up to processing study i, Plucky would have found an available day for study i and scheduled it in S, giving a contradiction.
- **$S^\star$ assigns study i to a different day than S.** Plucky's algorithm would assigns study i to the latest available day D, so $S^\star$ must have study i assigned to an earlier day $D' < D$. The study (if one exists) at day D in $S^\star$ can swap places with study i. Study i would still be completed before its end date and the other study would only be moved forward, so the new schedule would be valid. This would create a new optimal strategy consistent with S through study i, a contradiction.

All cases lead to contradiction, so we conclude Plucky's algorithm must return the maximum payout.

Common Errors: Note that in any inductive proof it is important to prove the inductive hypothesis **exactly** for the next iteration in the inductive step. We illustrate a common proof idea that would fail for this question because of this reason:

Inductive Hypothesis: The algorithm works well for k total studies. **Base Case**: The algorithm trivially works when there are 0 studies. **Inductive Step:** Say we have $k + 1$ studies. We

schedule the study with the highest payout on a particular day. We can now use IH to prove optimality of the remaining k studies.
Note the above proof seems to work for 3.1 as well, even though we just proved that 3.1 is sub-optimal! The discrepancy comes from the following: the IH implicitly assumes that we have all the days available to book as we want. However, after booking the highest payout study in the IS, we don't have all the days left. By being imprecise in our IH, we ignore the implicit assumption and end up with an incorrect proof.
While different phrasings of the proof would have different loopholes, in general proofs by induction on the number of days left to schedule or the number of total studies, etc. would likely not work.
As a sanity check, you can always check if your proof, or a slightly modified version of it would work for 3.1. If it does, then the proof is obviously wrong.

3.3 (16 pt.)

Lucky has resolved to be a more ethical lemur. Lately, he's been reading up on utilitarianism, and decides that rather than pursuing only his own profit, he should instead participate in the studies that are most valuable by utilitarian standards.

3.3.1 (4 pt.)

How would a utilitarian calculate the value of each study? Would Lucky's personal payout (assuming he participates) factor into the calculation? Why or why not?

[We are expecting: (1) a 1 sentence description, in general terms, of how a utilitarian calculates value, and (2) a 1 sentence explanation of whether and why Lucky's personal payout does or does not factor into the calculation**]**

SOLUTION:
"A utilitarian would calculate the value of each study by adding up the happiness that that study would produce for everyone affected by it. Since Lucky is one of the people affected by the study, his personal payout would factor into the calculation as a consequence of the study that makes him happy." Students do not need to emphasize this, but of course, Lucky's personal payout is only one factor among many in the calculation.

3.3.2 (4 pt.)

What changes would Lucky need to make to Plucky's algorithm in order to return maximum utilitarian value rather than maximum personal payout?

[We are expecting: 2-3 sentences describing the ways in which Plucky's algorithm would need to change to output maximum utilitarian value**]**

SOLUTION:
"In the input set of studies, the p variables would have to be values of the total overall happiness produced by the study, rather than Lucky's personal payout from the study. Then the sort(studies)

method would have to sort the studies by these new p variables, so that the studies end up sorted by decreasing order of utilitarian value"

3.3.3 (4 pt.)

Why might it be difficult to actually do the kind of utilitarian calculation you described in 3.3.1? Please support your answer by giving at least one specific example of something relevant to the utilitarian calculation that could be difficult to measure.

[We are expecting: 2-3 sentences clearly describing a measurement-related barrier to doing the utilitarian calculation, either focusing on or supported by a specific example of a difficult-to-measure quantity that the calculation relies on**]**

SOLUTION:
There is more room for variability in correct answers here, as there are a number of measurement-related barriers to actually doing this kind of utilitarian calculation. There is the general problem that it is not clear how to measure happiness, especially in a way that allows for comparison across individuals. Students who go for this option should ideally make it more specific by at least naming a person or group of people whose happiness would be relevant but difficult to measure. So they might write, e.g., "It might be difficult to do the utilitarian calculation because it's not clear how to measure happiness. For example, how much happiness do Lucky or the other participants get from their payouts, and how does that compare to how much happiness future psychologists or their patients will derive from the study's useful findings?" Students might also point to the difficulty in measuring how much happiness the study will generate for future people, given that its results are at present uncertain. Again, students should ideally make this a bit more specific. So they might write, e.g., "It might be difficult to do the utilitarian calculation because much of the potential happiness generated by psychological studies depends on how useful their results will be, which is hard to know in advance. A study into a new method of treating depression, for instance, might generate enormous amounts of happiness for future people, but only if the study turns out to be successful." Enterprising students might point out even more remote consequences of a study, which matter to the utilitarian's calculation but are difficult to anticipate. For instance, if Lucky turns his small payout into millions by using it as an early-stage investment in the next Google or Apple, and then uses his millions to feed the hungry, then the happiness the study will produce by way of Lucky's payout alone is enormous, but this is something that is hard to measure in advance. Students might give other answers as well. Full credit should be given to any good-faith effort that discusses anything that could reasonably be considered a measurement-related barrier to adding up the happiness of everyone affected by a study.

3.3.4 (4 pt.)

Suppose that one of the studies violates the privacy rights of its participants, although the participants never find out about this violation. Would this fact affect a utilitarian's evaluation of the study (assuming the utilitarian was aware of it)? Why or why not?

[We are expecting: 2-3 sentences explaining why this fact is or is not relevant to the utilitarian's evaluation**]**

SOLUTION:
Here too a number of answers are acceptable. The most straightforward answer would look something like this: "If the participants in the study never find out about the violation of their privacy, then the violation won't affect their happiness. The violation would therefore make no difference to the utilitarian's evaluation of the study, since the utilitarian's evaluation depends only on the study's effects on happiness." But enterprising students could reasonably reach the opposite conclusion, for example, by arguing that a person's "true" happiness or well-being could be set back even by violations of their privacy that they're unaware of. Or they might argue that researchers who get away with conducting a privacy-violating study might be more likely to violate privacy again in the future in ways that would have a detrimental effect on the happiness of future study participants. Again, good-faith efforts should be rewarded; the main thing is that students recognize that privacy violation can only be relevant to the utilitarian calculation if it has an effect on someone's happiness. It's possible that a student might give an answer like this: "If the utilitarian is also a deontologist, the privacy violation might affect their evaluation even if no one is aware of it." Strictly speaking, you can't be both a utilitarian and a deontologist, but since this point was not emphasized in the lecture, an answer like this should be given full credit (but only if the student is explicitly considering a utilitarian who is also a deontologist. If they just say what a deontologist would say, they're not answering the question!)

4 Optimal Matrix Multiplication (35 pt.)

We consider the problem of deciding multiplication order for matrix multiplication. In particular, given n matrices $A_0, A_1, \dots, A_{n-1}$, we want to decide in which order to multiply them, to take the least time. For this problem, we assume the matrix dimensions are given as an $(n+1)$-sized array d, where the dimensions of the matrix A_i are $d[i] \times d[i+1]$ for $i \in \{0, 1, \dots, n-1\}$. Assume that it takes $i \cdot j \cdot k$ time to multiply two matrices of sizes $i \times j$ and $j \times k$; the resulting matrix is going to be of size $i \times k$.

For example, for $d = [1, 2, 3, 1]$, we have three matrices A_0, A_1, A_2 with dimensions 1×2, 2×3, 3×1, respectively. There are two possible ways of multiplying these three matrices which can be shown as $((A_0A_1)A_2)$ and $(A_0(A_1A_2))$. In other words, we could multiply the first two matrices first and the result with the third, or multiply the last two matrices first. It takes 9 time for the first way and 8 time for the second way, so we prefer the second way.

4.1 (5 pt.)

Find the optimal time (and how to achieve this time) for multiplication in the following examples:

Example 1: $d = [1, 10, 1, 10, 1]$ Example 2: $d = [1, 10, 1, 10, 10, 1]$

[We are expecting: Optimal time and order of multiplications for each of the examples (you can use parentheses to indicate the order).**]**

SOLUTION:
1. 21. We use the following way to multiply matrices: $((A_0A_1)(A_2A_3))$.
2. 121. We use the following way to multiply matrices: $((A_0A_1)(A_2(A_3A_4)))$.

4.2 (15 pt.)

Let us define $M(i, j)$ as the optimal time to multiply the matrices between i and j (both inclusive), i.e., the optimal time to compute $A_iA_{i+1} \cdots A_j$. Write a recursive relation for $M(i, j)$.

[We are expecting: Recursive formulation and 2-3 sentences of justification.**]**

SOLUTION:
We take various cases of where we could insert the first break. That is, for each k between i and $j-1$, we consider multiplying everything between i and k and everything between $k+1$ and j before multiplying the results with each other. This gives us the following recurrence relation.
$M(i, j) = \min\{M(i, k) + M(k+1, j) + d[i] \cdot d[k+1] \cdot d[j+1] \mid i \leq k < j\}$

4.3 (15 pt.)

Write an algorithm, with a runtime of $O(n^3)$, to find the optimal time to multiply our n matrices.

[We are expecting: Pseudocode AND English description, justification for runtime, and proof of correctness.**]**

SOLUTION:

```
/* We use memoization and initialize an $n \times n$ array $M$ of NULL values to
   store the results                                                        */
function getOptMult($(i, j)$):
    if $i = j$ then
        return 0
    if $M[i, j] \neq NULL$ then
        return $M[i, j]$
    $M[i, j] \leftarrow \infty$
    for $k \in \{i, \ldots, j-1\}$ do
        $M[i, j] \leftarrow \min\{M[i, j]$, getOptMult$(i, k)+$getOptMult$(k+1, j)+d[i]d[k+1]d[j+1]\}$
    return $M[i, j]$
/* We call getOptMult$(0, n-1)$ to find the optimal answer                   */
```

We use a DP approach and create a 2D array corresponding to $M(i, j)$. We can use a top-down recursive formulation to solve for $M(0, n-1)$ using the above recurrence relation.
For time complexity analysis, note that we need to fill in the entire n^2 length array and each entry (i, j) will make $j - i \leq O(n)$ recursive calls.

5 Graph Mania! (40 pt.)

For each of the following algorithm design tasks, please provide the following:

- A clear English description of how you would do the task, and
- A justification for why your proposed algorithm meets runtime requirements.

In this problem, all graphs will have n vertices and m edges.

Please note the following:

- Pseudocode is not required, but please feel free to include it to make your algorithm clearer. **It is not necessary to provide a formal proof of correctness.**
- Any result or algorithm seen in class is fair game to cite without justification.
- If you want to do a graph task using a graph algorithm from class, be very explicit about which algorithm/procedure you are using, what task you want to accomplish, and what your input/output is.
- All runtimes are worst-case, deterministic runtimes.
- Space is not constrained, but do not use prohibitively large amounts of space.
- It is not necessary, but if it helps, you may assume you can easily access any node's neighbors in constant time (in a directed graph, incoming and outgoing neighbors).

5.1 (10 pt.)

We define the **diameter** of a strongly connected directed graph $G = (V, E)$ as the largest distance between any two vertices in V. That is,

$$\text{diameter}(G) = \max\{\text{distance}(u, v) \mid u, v \in V\}.$$

Recall that distance(u, v) is the cost of the shortest directed path from u to v.

Take $G = (V, E)$ to be a weighted strongly connected directed graph. G may have negative edges, but you are guaranteed that G contains no negative cycles. **Design an algorithm that finds the diameter of G in time $O(n^3)$.**

[We are expecting: A clear English description (pseudocode optional) AND a brief justification of runtime.**]**

SOLUTION:

- Run Floyd-Warshall on G to retrieve the shortest paths between all pairs in the graph. This takes $O(n^3)$ time.
- Take one pass over the pairs to find the maximum distance between a pair. This takes $O(n^2)$ time.

5.2 (13 pt.)

Let $G = (V, E)$ be a directed unweighted graph (i.e., every edge has weight 1). Among a set of vertices S, we say $v \in S$ is the closest to u if, among those vertices in S that are reachable from u, vertex v is the one reachable via a path of the shortest total cost. **Given two distinct vertices $x, y \in V$, design an algorithm that finds the closest vertex to x that is also reachable by y. Your algorithm must run in $O(m)$ time.**

Please note the following:

- It may be the case that the closest vertex to x reachable by y is actually x itself.
- You may assume that at least one vertex is reachable by both x and y.
- You may assume that $m > 0$.

[We are expecting: A clear English description (pseudocode optional) AND a brief justification of runtime.**]**

SOLUTION:

- Run single-source tree-building BFS from node x to get lengths of shortest paths from x to every other node in the graph. The nature of BFS guarantees we get these shortest paths arranged in order of distance from x (the distance-0 node first, then the distance-1 nodes, and so on).
- Run single-source tree-building BFS from node y to get a set of nodes reachable from y. Enumerate the vertices and then maintain a bit array isReachable which stores 0 if not reachable by y, 1 if reachable by y. (Also acceptable: store reachable vertices as a hashset)
- Iterate through the nodes of the graph in the order they were returned by the node-x BFS run. Check whether each one is reachable by y (using either the bit array or hashset). Return the first value which is reachable.
- For runtime, we first run single-tree-building BFS twice. In any graph, this BFS runs in time $O(m)$ because every new node we encounter as we build our tree must have been reached by some distinct edge; that is, no matter how large n is in the original graph, the number of nodes we actually traverse in single-tree-building BFS is upper-bounded by the number of edges we go through. So these two runs take $O(m)$ time each. Then we iterate through the output of the node-x BFS run (which is again limited by $O(m)$ as before). To check whether each of these is reachable by y takes constant time (deterministically with a bit array, expected time with a hashset). So this also costs $O(m)$ time, and the overall runtime is also $O(m)$.

5.3 (17 pt.)

Let $G = (V, E)$ be a directed unweighted graph. We say G is "somewhat-connected" if for every pair of vertices $u, v \in V$, either there is a directed path from u to v in G, there is a directed path from v to u in G, or both.

Design an algorithm to determine if G is "somewhat-connected" in $O(m + n)$ time.

[We are expecting: A clear English description (pseudocode optional) AND a brief justification of runtime.**]**

SOLUTION:

- Run Kosaraju's on G to get the SCCs. Convert G into G' its SCC-DAG.
- Toposort G'.
- Go left to right through the toposort. If we find a pair of SCCs which are adjacent in the toposort but do not have an edge between them in G', return False.
- If we made it through that run without returning False, return True.
- For runtime, note that Kosaraju's is $O(m+n)$ time, the transformation into an SCC-DAG takes at most $O(m+n)$ time, toposort is just DFS and takes $O(m+n)$ time, and the last check of consecutive SCCs in the toposort is linear in the number of SCCs, or $O(n)$ in the worst case, so overall runtime is $O(m+n)$ as desired.
- (justification, which is unnecessary to receive credit: In order for G to be semi-connected, it must be true that consecutive nodes in our toposort are connected left-to-right. The reason for this is that if there were some pair in our toposort A, B such that A did not have an edge to B, then there would be no path for A to get to B because they are ordered consecutively in the toposort. Then the existence of this pair A, B would mean the graph cannot be somewhat-connected (since the nodes within the SCCs do not have paths between them) and we would return False. On the other hand, if every single consecutive pair in the toposort obeys this property, that the first in the pair has an edge leading directly to the second, then it confirms that we have a path that goes from one SCC through each one of the rest; that is, the SCCs are all connected from left to right in the toposort. Since the nodes within SCCs are strongly connected, and we see that every possible pair of SCCs has a path between them (by looking at the toposort and going from the more left of the two to the more right of the two), we can conclude that every pair of nodes has a path between them in some direction and therefore that the graph is somewhat-connected and we return True.)

This is the end!