PRACTICE Final Exam Solution CS161, Fall 2025

Instructions (that will appear on the real final exam)

- **DO NOT OPEN THE EXAM UNTIL YOU ARE INSTRUCTED TO.**
- Answer all of the questions as well as you can. You have **180 minutes.**
- The exam is **non-collaborative**; you must complete it on your own. If you have any clarification questions, please ask the course staff. We cannot provide any hints or help.
- This exam is **closed-book**, except for **up to three double-sided sheets of paper** that you have prepared ahead of time. You can have anything you want written on these sheets of paper.
- **Please DO NOT separate pages of your exam**. The course staff is not responsible for finding lost pages, and you may not get credit for a problem if it goes missing.
- There are a few pages of extra paper at the back of the exam in case you run out of room on any problem. If you use them, please clearly indicate on the relevant problem page that you have used them, and please clearly label any work on the extra pages.
- Please make sure to sign out of the roster when handing in your completed exam to the teaching team.
- **Please do not discuss the exam until after solutions are posted!** Of course for this practice exam, feel free to discuss the solutions with the course staff and anyone else!

General Advice

- If you get stuck on a question or a part, move on and come back to it later. The questions on this exam have a wide range of difficulty, and you can do well on the exam even if you don't get a few questions.
- Pay attention to the point values. Don't spend too much time on questions that are not worth a lot of points.
- There are **110** total points on this exam. There are **seven problems** across **22 pages**.

Name and SUNet ID (please print clearly):

SOLUTION

This page intentionally blank. Please do not write anything you want graded here.

Honor Code

The Honor Code is an undertaking of the Stanford academic community, individually and collectively. Its purpose is to uphold a culture of academic honesty. Students will support this culture of academic honesty by neither giving nor accepting unpermitted academic aid on this examination.

This course is participating in the proctoring pilot overseen by the Academic Integrity Working Group (AIWG), therefore proctors will be present in the exam room. The purpose of this pilot is to determine the efficacy of proctoring and develop effective practices for proctoring in-person exams at Stanford.

Unpermitted Aid on this exam includes but is not limited to the following: collaboration with anyone else; reference materials other than your cheat-sheet (see below); and internet access.

Permitted aid on this exam includes a "cheat-sheet:" two double-sided sheets of paper with anything written on them, which you have prepared yourself ahead of time.

I acknowledge and attest that I will abide by the Honor Code:

[signed] _____

Exam Break Sign-out

I pledge that during my exam break:

- I will not bring any paper, electronic devices (phone, smart watch, smart glasses, etc), or aid (permitted or unpermitted) *out of or into* the exam room.
- I will not communicate with anyone other than the course instructional staff about the content of the exam.

Signature Confirming Honor Code Pledge	Exit Time	Return Time	Proctor Initial	Length (min)

If you are feeling unwell and are not able to complete the exam, please connect with the proctor to discuss options.

Good Luck!

This page intentionally blank. Please do not write anything you want graded here.

Multiple Choice and Short Answer

1. **(29 pt.) [Multiple Choice!]** For each of the parts below, clearly fill in your answers. Ambiguously filled-in answers will be marked incorrect.

●	○	
Filled in means: "This is a correct choice"	Empty means: "This is **not** a correct choice"	Anything else may be marked as incorrect!

[**We are expecting:** *For each part, just clearly filled-in-answers. No explanation is required or will be considered while grading.*]

(a) **(5 pt.)** Which of the following algorithms can find shortest paths in weighted directed graphs, that may have negative edge weights (but no negative cycles). Select "Yes" for all that apply.

A. BFS

○ Yes ○ No

B. Dijkstra's Algorithm

○ Yes ○ No

C. Bellman-Ford Algorithm

○ Yes ○ No

D. Floyd-Warshall Algorithm

○ Yes ○ No

E. Ford-Fulkerson Algorithm

○ Yes ○ No

SOLUTION:
C and D. A is false since BFS doesn't work on weighted graphs, and B is false since Dijkstra doesn't work with negative edge weights. E is false since Ford-Fulkerson doesn't find shortest paths in any graph, it finds max flows.

(b) **(3 pt.)** In which of the following scenarios would Prim vs. Kruskal be asymptotically faster for finding a valid minimum spanning tree? Select all that apply.

A. On a graph $G = (V, E)$, where $|E| = \Theta(|V|)$ and the edge weights are all integers from 1 to 10. Assume Prim's algorithm uses a Fibonacci heap.

- ◯ Prim's is asymptotically faster
- ◯ Kruskal's is asymptotically faster
- ◯ They are asymptotically the same

B. On a graph $G = (V, E)$ where $|E| = \Theta(|V|^2)$, and the edge weights are arbitrary comparable items. Assume Prim's algorithm uses a Fibonacci heap.

- ◯ Prim's is asymptotically faster
- ◯ Kruskal's is asymptotically faster
- ◯ They are asymptotically the same

C. You have no guarantees on the graph (other than that it is connected), and you are using an RBTree in Prim's algorithm.

- ◯ Prim's is asymptotically faster
- ◯ Kruskal's is asymptotically faster
- ◯ They are asymptotically the same

SOLUTION:

In the explanations below, $n = |V|$ and $m = |E|$.

A. Kruskal is faster. Prim runs in time $O(n \log n + m) = O(n \log n)$ since $m = \Theta(n)$, and Kruskal runs in time approximately $O(m)$ since we can radixSort the edges.

B. Prim is faster. Prim runs in time $O(n \log n + m) = O(m) = O(n^2)$ if $m = \Theta(n^2)$. Kruskal runs in time $O(m \log m) = O(n^2 \log n)$ since the running time is dominated by sorting the edges.

C. They are asymptotically the same. Prim runs in time $O((n + m) \log n) = O(m \log n)$ with and RBTree. (For the equality, we are using that $n = O(m)$ in a connected graph). Kruskal runs in time $O(m \log m) = O(m \log n)$ (for the equality, we are using that $m \leq n^2$, so $\log(m) = O(\log n)$). So they both have running time $O(m \log n)$.

(c) **(3 pt.)** Suppose that $s \to v_1 \to v_2 \to \cdots \to v_\ell \to t$ is a shortest path from s to t in a connected graph with (possibly negative) edge weights, but no negative cycles.

Which of the following are true?

A. The path $s \to v_1 \to v_2 \to \cdots \to v_i$ is a shortest path for all $i = 1, \ldots, \ell$.

◯ True ◯ False

B. If all the edge weights in G are distinct, then this shortest path must be unique.

◯ True ◯ False

C. If you were to double all the edge weights in G, then $s \to v_1 \to v_2 \to \cdots \to v_\ell \to t$ would still be a shortest path in the transformed graph.

◯ True ◯ False

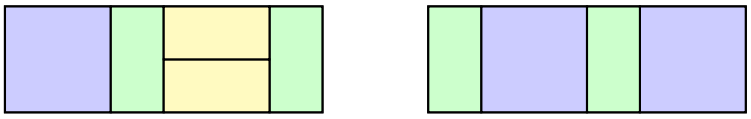
SOLUTION:

A is true (sub-paths of shortest paths are shortest paths)

B is false (for example, consider two paths, one with weights 2 and 3, and one with weights 1 and 4)

C is true. Let G' be G after all the edge weights have been doubled, and notice that the cost of a path in G' is twice the cost in G. So if this path (call it P) were not shortest in G', then there would be some other path P' so that $2 \cdot \text{cost}(P') < 2 \cdot \text{cost}(P)$. But then dividing by 2, we see that $\text{cost}(P') < \text{cost}(P)$, a contradiction.

(d) **(5 pt.)** Suppose you want to count the number of ways to tile a $2 \times n$ grid with tiles. You have (an infinite number of) tiles that are 2×2, 1×2, and 2×1. For example, if $n = 6$, here are two legitimate tilings:



Let $D[j]$ be the number of legitimate tilings of a $2 \times j$ board. Which of the following is a correct recursive relationship between these sub-problems?

A. ◯ $D[j] = D[j-2] + D[j-1]$

B. ◯ $D[j] = D[j-2] \times D[j-1]$

C. ◯ $D[j] = D[j-2] + 2 \cdot D[j-1]$

D. ◯ $D[j] = 2 \cdot D[j-2] + D[j-1]$

E. ◯ None of the above.

SOLUTION:
The correct answer is D. To see this, consider what possibilities there are for the j'th column. Either it is a 2×1 domino, in which case there $D[j-1]$ ways to fill in the rest. Or it is not, in which case the last 2×2 square is filled with either a 2×2 tile, or two 1×2 tiles stacked on top of each other. In that case, there are $2D[j-2]$ ways. So we add the two cases together to get the answer (D).

(e) **(5 pt.)** Suppose you have M identical items that you want to allocate to n people. Each person wants as many items as possible, up to some limit B, but they have diminishing returns. If person i already has $s-1$ items, then the value they get from the s'th item is $\Delta_{i,s}$, where

$$\Delta_{i,1} \geq \Delta_{i,2} \geq \cdots \geq \Delta_{i,B}.$$

There is no benefit to anyone getting more than B items.

You want to decide how to allocate items to maximize total value among all n people. You will give out the items one at a time, following a greedy strategy. Which greedy rule is correct for this problem? Choose the best answer. Assume that ties are broken arbitrarily.

A. ◯ Give the next item to the person with the smallest *total value so far*. (If person i has $s-1$ things so far, their *total value so far* is $\sum_{j=1}^{s-1} \Delta_{i,j}$).

B. ◯ Give the next item to the person who has the most *marginal benefit* from the next item. (If person i has $s-1$ things so far, their *marginal benefit* from the next item is $\Delta_{i,s}$).

C. ◯ Give the next item to the person who has the most *left to gain*. (If person i has $s-1$ things so far, the *amount they have left to gain* is $\sum_{j=s}^{B} \Delta_{i,j}$).

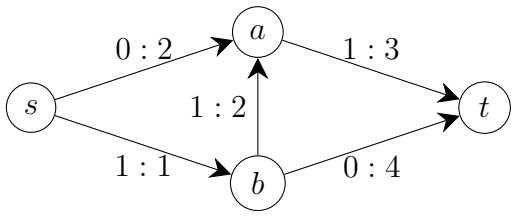
SOLUTION:

The solution is B (maximize marginal gain).

To see that A is not correct, suppose that there is one person, Alice, for whom $\Delta_{i,s} = 0$ for all s; and one person, Bob, with $\Delta_{i,s} > 0$ for $s \leq B$. Then after possibly giving one item to Bob, we'd give the rest to Alice, when in fact we should have given all to Bob.

To see that C is not correct, suppose that there are 3 people and 3 items, and that $B = 3$. Alice and Bob have $\Delta_{i,1} = 3$ and $\Delta_{i,s} = 0$ for $s > 1$. Carol has $\Delta_{\text{Carol},s} = 2$ for all s. The algorithm would give the first item to Carol (since she'd have remaining upside $6 > 3$); the second item to Carol (with remaining upside $4 > 3$) and the last item to one of Alice or Bob. This results in a value of $2 + 2 + 3 = 7$. However, the optimal solution is to give one item to each person, for value $2 + 3 + 3 = 8$.

(f) **(5 pt.)** Consider the following graph, which shows an s-t flow. Below, the notation $x : y$ means that the edge has flow x out of total capacity y.



A. Which of the following are augmenting paths in G? Fill in one bubble for each path.

Path	Augmenting Path	Not an augmenting path
$s \to a \to t$	◯	◯
$s \to a \to b \to t$	◯	◯
$s \to b \to a \to t$	◯	◯

B. What is the maximum flow in this graph? Choose exactly one.

◯ 2
◯ 3
◯ 4
◯ 5

SOLUTION:
A. $s \to a \to t$ and $s \to a \to b \to t$ are augmenting paths. $s \to b \to a \to t$ is not, since it uses the full edge $s \to b$.

B. The max flow is 3. You can see this because if we were to update along the augmenting path $s \to a \to t$ twice, we'd get flow 3, and we know this is optimal because we can find a cut with cost 3 (e.g., the cut that separates s from $\{a, b, t\}$).

(g) **(3 pt.)** Suppose Alice is designing an algorithm to decide which public school students should get need-based access to a free tutoring service. Alice's algorithm uses a student's GPA as an input. Barbara raises a concern that the algorithm does not consider whether the student is taking advanced, on-level, or remedial math courses, and a low grade in an advanced course may show less need than a low grade in a remedial course. What *best* describes the issue Barbara is raising?

A. ◯ Barbara is concerned about an abstraction
B. ◯ Barbara is concerned about an idealization
C. ◯ Barbara is concerned about an NP-hard problem

SOLUTION:
A is correct. Abstraction omits details of a real-world situation. In contrast, idealization *changes* details of a real-world situation, so B is not as good an answer. NP-hardness isn't relevant here, so C is not correct either.

2. **(9 pt.) [Recurrence Relations]** For each part below, give the best big-Oh bound you can for each $T(n)$.

   [**We are expecting:** *For each part, an expression "O(something in terms of n)." No explanation is required or will be considered. Your answer should be as simplified as possible.*]

   (a) **(3 pt.)** $T_1(n) = 4T_1(\lfloor n/2 \rfloor) + n^3$ for $n \geq 3$, $T_1(n) = 1$ for $n < 3$.

   $$T_1(n) = \_\_\_\_\_$$

   (b) **(3 pt.)** $T_2(n) = T_2(\lfloor n/2 \rfloor) + \log_2 n$ for $n \geq 2$, $T_2(n) = 1$ for $n < 2$.
   [**HINT:** *Recall that* $\sum_{j=0}^{T} j = \frac{T(T+1)}{2}$, *and* $\log_2(a/b) = \log_2(a) - \log_2(b)$.]

   $$T_2(n) = \_\_\_\_\_$$

   (c) **(3 pt.)** $T_3(n) = T_3(\lfloor n/2 \rfloor) + T_3(\lfloor n/4 \rfloor) + n$ for $n \geq 4$, $T_3(n) = 1$ for $n < 4$.

   $$T_3(n) = \_\_\_\_\_$$

SOLUTION:

(a) $T_1(n) = O(n^3)$ by the Master Theorem

(b) $T_2(n) = O((\log n)^2)$. To see this, we can use the tree method. There are $T \approx \log_2(n)$ levels, and at the j'th level we do $\log_2(n/2^j)$ work. Summing this us, we get

$$\begin{aligned}\sum_{j=0}^{T} \log_2(n/2^j) &= \sum_{j=0}^{T} \log_2(n) - \sum_{j=0}^{T} \log_2(2^j) \\ &= T \log_2(n) - \sum_{j=0}^{T} j \\ &\approx \frac{\log_2(n)^2}{2}.\end{aligned}$$

You could also use the substitution method to prove that this is the right answer, but "guessing" the answer first might be hard without the tree method.

(c) $T_3(n) = O(n)$. To see this, we can use the substitution method. We guess that $T_3(n) \leq Cn$, since we know that $O(n)$ is the smallest it can be (because of the "$+n$" term at the end). To check the base cases, when $1 \leq n \leq 4$, we get $1 = T_3(n) \leq Cn$

as long as $C \geq 1$. Next we do the inductive step. Assuming that $T_3(n) \leq Cn$ for all $n < j$, by induction we see that

$$T_3(j) \leq \frac{Cn}{2} + \frac{Cn}{4} + n = \left(\frac{3C}{4} + 1\right) n \leq Cn$$

as long as $C \geq 4$. So if we choose $C = 5$ or something, this proves that $T_3(n) = O(n)$.

3. **(20 pt.) [True or False and Why?]** For each of the parts below, say whether the statement is true or false, and give a *short* explanation saying why. You don't have to give a formal proof, but your explanation should be convincing to the grader. You can use any statements/algorithms we have seen in class.

   (-) **(Example:)** Let G be an undirected unweighted connected graph. True or False: Given s and t, it is possible to find a shortest path from s to t in G in time $O(n + m)$.

   Answer: True. We can run BFS starting at s to find the distance from s to any other vertex in an unweighted graph, in time $O(n + m)$.

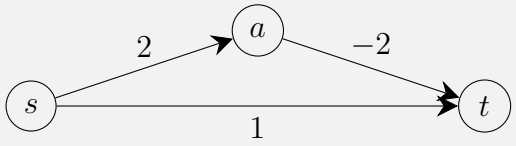
   (a) **(5 pt.)** Suppose that $f(n)$ and $g(n)$ are positive functions of n, and that $f(n) = O(g(n))$. True or False: $f(n) + g(n) = O(g(n))$.
   Explain your answer, using the definition of big-O.

   SOLUTION:
   True. If $f(n) = O(g(n))$, then there are constants C and n_0 so that $f(n) \leq Cg(n)$ for all $n \geq n_0$. This implies that $f(n) + g(n) \leq (C + 1)g(n)$ for all $n \geq n_0$, which shows that $f(n) + g(n) = O(g(n))$.

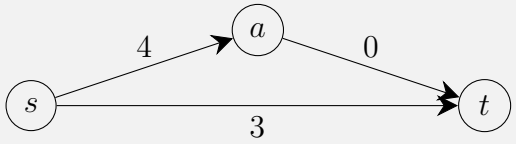
(b) **(5 pt.)** Your friend thinks they have a way to get Dijkstra's algorithm to work with negative edge weights. Suppose that G is a weighted directed graph, so that the weight of $(u, v) \in E$ is $w(u, v)$. Suppose that the smallest (most negative) edge weight is $-w^* < 0$, so $w^* = -\min_{(u,v)\in E} w(u, v)$. Let G' be the same as G, except with edge weights $w'(u, v) = w(u, v) + w^*$. By construction, G' has no negative edge weights, so we can run Dijkstra's algorithm on G' and return the resulting shortest paths.

True or false: this algorithm (constructing G' and running G on it) will correctly return shortest *paths* in G. (It's okay if it doesn't return the correct path lengths). If your answer is "True," explain why. If your answer is "False," give a counter-example.

SOLUTION:
False. For example, in the graph G =



the shortest path from s to t is $s \to a \to t$, which has cost 0. On the other hand, if we add 2 to all the edges to get G' with all non-negative edge weights, we get:



and now the shortest path is the one edge $s \to t$, which has cost 3.

(c) **(5 pt.)** Your friend has developed a new hash family $\mathcal{H}$ that maps the integers $1, \ldots, M$ to n buckets, for $n \geq 2$. Every function $h \in \mathcal{H}$ has the property that it maps any pair $x \neq y$ of integers in $\{1, \ldots, M\}$ to distinct buckets (that is, $h(x) \neq h(y)$ for all $h \in \mathcal{H}$), *except* for 3 and 5: for these two values, we will have $h(3) = h(5)$ for all $h \in \mathcal{H}$. However, your friend tells you that for different $h \in \mathcal{H}$, which bucket 3 and 5 land in is uniformly distributed over the buckets. (That is, for any $i \in \{1, \ldots, n\}$ $\Pr_{h\in\mathcal{H}}[h(3) = i] = 1/n$, and similarly for $h(5)$).

True or False: $\mathcal{H}$ is a universal hash family. Explain your answer, using the definition of universal hash family.

SOLUTION:
False. The definition of a universal hash family is that for all $x \neq y$, $\Pr_{h\in\mathcal{H}}[h(x) =$

$h(y)] \leq 1/n$. Your friend has just told you that

$$\Pr_{h \in \mathcal{H}}[h(3) = h(5)] = 1,$$

so this is *not* a universal hash family, since $1 > 1/n$.

(d) **(5 pt.)** In class, we saw a *minimum* spanning tree, which is a spanning tree of minimum cost. A *maximum* spanning tree is a spanning tree of maximum cost. Consider modifying Kruskal's algorithm by, instead of greedily taking the lowest-cost edge that doesn't form a cycle, you take the *highest*-cost edge that doesn't form a cycle. Call this new algorithm max-Kruskal.

True or false: max-Kruskal correctly finds a *maximum* spanning tree in a connected, undirected, weighted graph. Explain your answer.

SOLUTION:
True. There are (at least) two ways to see this.

Explanation 1: Imagine taking a graph G, and replacing every edge weight w with $-w$ to get a graph G'. Then running Kruskal on G' is the same as running max-Kruskal on G. Thus, max-Kruskal will return a *minimum* spanning tree in G', which is the same as a *maximum* spanning tree in G.

Explanation 2: When we were proving Kruskal's algorithm worked, we did it by showing that Kruskal always took a light edge crossing a cut that respected the choices made so far. Similarly, max-Kruskal takes the heaviest edge crossing a cut that respects the choices made so far. When analyzing Kruskal, we showed that taking a light edge means that we are still on track for the lightest tree; exactly the same logic shows that max-Kruskal stays on track for the heaviest tree.

Note: This next problem may be trickier, and is only worth 10 points out of 110 points total. (There are two independent 5-point parts). You may want to come back to this after you have finished the rest of the exam.

4. **(10 pt.) How to do it?** For each of the tasks below, explain briefly how you would do it. You may use any algorithm we have seen in class as a black box.

   [**We are expecting:** *For each, a clear English explanation of your algorithms. You do not need to include pseudocode, but you can if you think it makes your answer clearer. You do not need to justify correctness or running time.*]

   (a) **(5 pt.)** *[May be trickier.]* Let $G = (V, E)$ be an unweighted directed graph. Recall that G is *strongly connected* if for every $u, v \in V$, there is a path from u to v in G, *and* there is a path from v to u in G. Say that G is "*not-so-strongly-connected*" if, for every $u, v \in V$, *either* there is a path from u to v in G; *or* there is a path from v to u in G. Give an algorithm to decide if a graph is not-so-strongly-connected in time $O(n + m)$.

SOLUTION:

Run Kosaraju's algorithm from class in time $O(n + m)$ to find the SCC-DAG G' for G. Then topo-sort the vertices of G' (we can either do this in time $O(n + m)$ again using the topo-sort algorithm from class, or we can just return the SCCs in reverse order of finish time from the original DFS run in Kosaraju's algorithm). Let $S_1, S_2, \ldots, S_\ell$ be the sorted SCCs. Then we check if, for every $i \in \{1, 2, \ldots, \ell - 1\}$, there is an edge from S_i to S_{i+1} in G'. If there is, we return "Yes, this is not-so-strongly-connected!" If there is not, we return "Nope!"

Not required: To see why this works, suppose that the algorithm returns "Nope!" Then we claim that G is **NOT** not-so-strongly connected. Indeed, since the algorithm returned Nope, there must be some i so that there is no edge from S_i to S_{i+1}. Let u be any vertex in S_i and let v be any vertex in S_{i+1}. Then there is no path from v to u, since it would violate the topo-sorting of G'. There is also not a path from u to v: If there were it couldn't go through S_j for any $j > i + 1$ (or this would again violate the topo-sorting of G', since there would be a path from S_j to S_{i+1}). So there must be an edge from S_i to S_{i+1}, but we just said there wasn't. So there is no path from either u to v or from v to u, so G is not not-so-strongly-connected, as claimed.

On the other hand, suppose that the algorithm returns "Yes!" Then we claim that G is not-so-strongly-connected. Indeed, let u and v be any two vertices in G, and suppose that $u \in S_i$ and $v \in S_j$. Suppose without loss of generality that $i \leq j$. Then there is a path from u to v, by going through $S_i \to S_{i+1} \to \cdots \to S_j$. So indeed, G is not-so-strongly-connected.

Another part on next page

(b) **(5 pt.)** *[May be trickier.]* Let $G = (V, E)$ be a connected, unweighted, undirected graph. Let $s, t \in V$ be distinct vertices. Call a vertex $v \in V$ *essential* if every shortest path from s to t in G passes through v. Given G, s, and t, find a list of *all* of the essential vertices, in time $O(n + m)$.

[**HINT:** *Suppose that v lies on an shortest path from s to t. What can you say about the distance from s to v and the distance of t to v, relative to the distance from s to t?*]

SOLUTION:

Let $D_s[v]$ be the distance from s to v, and let $D_t[v]$ be the distance from t to v.

The algorithm is:

- Run BFS once from s to fill in D_s for all v.
- Run BFS again from t to fill in D_t for all v.
- Iterate through all $v \in V$, and create a set $S \subseteq V$ that consists of all v so that $D_s[v] + D_t[v] = D_s[t]$.
- Bucket-sort the vertices in S based on the key $D_s[v]$, in time $O(n)$. (Note that there are at most n values that $D_s[v]$ can take on).
- For each $i = 0, 1, 2, \ldots, n$, if the i'th bucket has only one vertex v in it (that is, if v is the unique vertex in S so that $D_s[v] = i$), then declare that v is essential.

The running time is $O(n+m)$ for the two runs of BFS, $O(n)$ for the bucket-sorting, and another $O(n)$ to step through the buckets, so $O(n + m)$ total.

Not required: To see why this works, notice first that v is on *any* shortest path from s to t if and only if $D_s[v] + D_t[v] = D_s[t]$. That's because sub-paths of shortest paths are shortest paths: if $D_s[t]$ is the length of a shortest path from s to t, then we can make it by taking a shortest path from s to v ($D_s[v]$) and a shortest path from v to t (distance $D_t[v]$, since G is undirected).

Thus, the set S that the algorithm creates is the set of all v that appear in *any* shortest paths. Now, we claim that $v \in S$ is essential if and only if there is no other $w \in S$ so that $D_s[w] = D_s[v]$.

To prove the claim, suppose first that there is some w so that $D_s[w] = D_w[v] =: d$. Then there is a shortest s-t path that uses d steps to go from s to w; and then $D_s[t] - w$ steps to go from w to t. This path can't involve v: If v appeared in the first part of the path (before w), then actually the distance between s and v was less than d. And if v appeared in the second part of the path (after w), then we could have made a shorter path from s to t by going to v rather than w. So v is not essential.

On the other hand, suppose that v is not essential. Then there must be some shortest path from s to t that doesn't involve v. Let $d = D_s[v]$, and let w be the d'th vertex in this path. But then $D_s[w] = d$ (sub-paths of shortest paths are shortest paths), so v is not alone in its bucket.

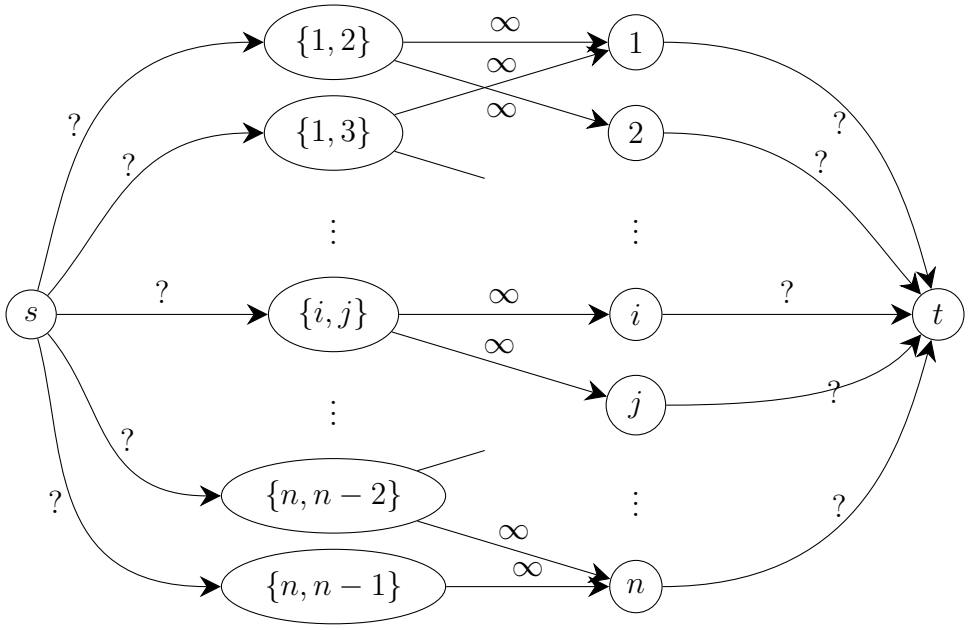
Algorithm Design

5. **(16 pt.) [Keeping the dream alive]** Suppose that you follow a sports league with $n+1$ teams, teams $0, 1, \ldots, n$. Team 0 (let's call them Team Cardinal) is your favorite team. Each team plays each other some number of times, and at the end of the season, the team who has won the most games is declared the champion. (If it's a tie, there are multiple champions).

   Right now, we are part-way through the season. Suppose that each team i has won w_i games so far. Team Cardinal (aka Team 0), has G games left to play in the season. For each other pair of teams $i, j \in \{1, 2, \ldots, n\}$ (*not* including Team Cardinal), there are $g_{i,j}$ games left in the season between Team i and Team j.

   As an avid fan, you want to know if it's still possible for Team Cardinal to become champion. You decide to solve this problem using the Ford-Fulkerson Algorithm.

   In more detail, consider the following graph:



   This graph has:

   - **Vertices:**
     - Special vertices s and t
     - One vertex for each (unordered) pair $\{i, j\}$ for $i, j \in \{1, \ldots, n\}$ of distinct teams (*not* including Team Cardinal)
     - One vertex for each team $i \in \{1, \ldots, n\}$ (*not* including Team Cardinal)
   - **Edges:**

- A directed edge from s to each "$\{i, j\}$" vertex. The weight of these edges is TBD.
- A directed edge from each "i" vertex to t. The weight of these edges is TBD.
- For each distinct pair $i, j \in \{1, \ldots, n\}$, there is a directed edge from vertex "$\{i, j\}$" to vertex "i"; and a directed edge from vertex "$\{i, j\}$" to vertex "j". All these edges have weight ∞.

(a) **(7 pt.)** Explain how to set the weights on the "?" edges (the ones with weight TBD) so that a maximum flow in this graph tells you whether or not Team Cardinal can still be champions.

That is, your goal is to decide if it is still possible for Team Cardinal to win at least as many games as any team $1, \ldots, n$, assuming they win all their remaining G games, as well as the w_0 games they have already won.

[**HINT:** *You'll need to use the parameters $g_{i,j}$, w_i, and G defined above.*]

[**HINT:** *Imagine that the flow on the edge $(\{i, j\}, i)$ represents how many games Team i will win against Team j going forward...*]

[**We are expecting:** *Just a description of the edge weights. No justification is required (yet)*]

SOLUTION:
For the edges $(s, \{i, j\})$, the edge weight should be $g_{i,j}$. For the edges (i, t), the edge weight should be $w_0 + G - w_i$.

(b) **(3 pt.)** Given your edge weights in the previous part, explain how to use the Ford-Fulkerson algorithm to tell if Team Cardinal can still be champions.

[**We are expecting:** *A clear English description of your algorithm.*]

SOLUTION:
If we have $w_0 + G - w_i < 0$ for any i (that is, if we have a negative capacity), then sadly it's not possible for Team Cardinal to win.

If all capacities are non-negative, run FF on the graph with the weights above to find a maximum flow from s to t. If the maximum flow is equal to $\sum_{i,j} g_{i,j}$, then it is still possible for Team Cardinal to win. Otherwise it's not.

(c) **(6 pt.)** Explain why your algorithm is correct.
[**We are expecting:** *A clear explanation; it doesn't have to be a formal proof.*]

SOLUTION:
Note: Such a long explanation would not be needed for full credit, but we want to make sure the solutions are clear.

If there are any negative edge weights, then $w_0 + G < w_i$ for some i. In this case, the maximum number of games Team Cardinal can possibly win, $w_0 + G$, is already less than the number of games team i has won. So Team Cardinal can't win. So assume that all the edge weights are non-negative, and we run Ford-Fulkerson.

Now, suppose it is possible for Team Cardinal to win. We will construct a flow with value $\sum_{i,j} g_{i,j}$, demonstrating that the max flow is at least this. (And since this is also the sum of the capacities coming out of s, this must be equal to the max flow).

First, we fill up each edge leaving s. Then, following the hint, we set the flow on $(\{i,j\}, i)$ to be the number of games that team i would win in the scenario where Team Cardinal is victorious. This means that the flow coming into node i is the number of games that Team i would win going forward; call this f_i. Thus, the total number of games that team i would win all season is $f_i + w_i$. Since Team Cardinal is victorious in this scenario, that means that

$$f_i + w_i \leq G + w_0,$$

or

$$f_i \leq w_0 + G - w_i.$$

The right hand side was our weight on the edge from node i to t, so we have just shown that the flow f_i coming out of node i can fit into this edge. Thus, this is a legit flow.

Going the other way, suppose that there is a maximum flow in this graph with value $\sum_{i,j} g_{i,j}$. Then all of the edges leaving s must be full. We know from class that all the flow values must be integers (since the capacities are integers – if you are worried about the ∞'s, note that you could replace them with a large integer like $\sum_{i,j} g_{i,j}$). Thus, we interpret the flow on the edge $(\{i,j\}, i)$ as "the number of games i will win against j going forward;" call this value f_i. Then, by the same logic above, the capacity of $w_0 + G - w_i$ on the edge (i,t) implies that, since the flow is legit, we must have

$$w_0 + G - w_i \geq f_i,$$

or

$$w_0 + G \geq f_i + w_i,$$

which implies that, in this set-up, Team Cardinal wins at least as many games as Team i, for every i. Hooray! Go Cardinal!

6. **(14 pt.) [Procrastination Optimization]** Over the upcoming break, Lucky the Lackadaisical Lemur is excited to kick back and binge-watch some shows. Lucky wants to watch k episodes over break; he's not very discerning, so he doesn't care what he watches.

   Lucky could access to m different platforms (LemFlix, Madagascar Prime-ate, Ring-tail+, etc). Suppose that the i'th platform has a fee of s_i to join. (Note: Lucky is planning on subscribing for one month during the break and then canceling, so it's a one-time fee for him). Once he's joined Platform i, each piece of content on Platform i costs some additional fee: Say that there are Q things on each platform, and that the j'th one costs $f_{i,j}$. You can assume that the pieces of content are ordered by price, so

$$0 \leq f_{i,1} \leq f_{i,2} \leq \cdots \leq f_{i,Q}.$$

   Design a DP algorithm to help Lucky find the minimum cost to watch k things across all these m platforms. Your algorithm should use the sub-problems:

   $D(i,j) =$ The least money Lucky can spend to watch j things across the first i platforms.

   If there aren't j distinct things to watch on the first i platforms, then $D(i,j) = \infty$.

   (a) **(2 pt.)** What is $D(i,0)$ for each $i \in \{1,\ldots,m\}$?
   [**We are expecting:** *Your answer, no explanation required.*]

   SOLUTION:
   $D(i,0) = 0$ for all $i \in \{1,\ldots,m\}$. That's because if I don't buy anything, it costs zero.

   (b) **(2 pt.)** What is $D(1,j)$ for each $j \in \{1,\ldots,k\}$?
   [**We are expecting:** *Your answer, and a sentence or two of explanation.*]

   SOLUTION:

$$D(1,j) = s_1 + \sum_{\ell=1}^{j} f_{1,\ell}$$

   for all $j \leq \min\{k, Q\}$, and $D(1,j) = \infty$ if $j > Q$. That's because I'm only allowed to use Platform 1; so I pay the cost s_1 to use that platform, and then I buy the first j (the cheapest) pieces of content.

(c) **(5 pt.)** Write a recurrence relation for the problems $D(i, j)$.
[**We are expecting:** *Your answer, and a sentence or two of explanation.*]

SOLUTION:
We start with the explanation. There are two cases, either Lucky's solution involves Platform i or it does not. If it does not, then $D(i, j) = D(i-1, j)$. If it does, then say that Lucky's solution buys t pieces of content from Platform i, for some $1 \leq t \leq \min\{Q, j\}$. Then the cost would be $D(i, j) = D(i-1, j-t) + s_i + \sum_{\ell=1}^{t} f_{i,\ell}$. Putting all these cases together, we get

$$D(i,j) = \min\left\{ D(i-1,i), \min_{1 \leq t \leq \min(j,Q)} D(i-1, j-t) + s_i + \sum_{\ell=1}^{t} f_{i,\ell} \right\}$$

(d) **(5 pt.)** Design an algorithm that runs in time $O(mQk)$ to compute the minimum cost that Lucky can spend. You algorithm should take as input k, and array L so that $L[i, j] = \ell_{i,j}$.
[**We are expecting:** *Clear pseudocode, and a justification of the running time.*]

SOLUTION:
Our pseudocode is:

Note: Whoops! "L" turned into "F" in the solution!

Function `StreamingLemur`(k, Q, F, S)**:**
  `// ` $F[i,j]$ `is the cost of the` j`-th cheapest item on platform` i
  `// ` $S[i]$ `is the cost to join platform` i
  Initialize array $D[0 \ldots m][0 \ldots k]$ so that all entries are ∞
  for $i \leftarrow 0$ **to** $m - 1$ **do**
    $D[i][0] \leftarrow 0$
  for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** k **do**
      $D[i][j] \leftarrow D[i-1][j]$
      $C \leftarrow 0$
      `// ` C `keeps track of` $\sum_{\ell=1}^{t} F[i,\ell]$ `as` t `varies`
      for $t \leftarrow 1$ **to** $\min(j, Q)$ **do**
        $C \leftarrow C + F[i,t]$
        $D[i][j] \leftarrow \min\Big(D[i][j],\ S[i] + C + D[i-1][j-t]\Big)$
  return $D[m][k]$

Note: we could have used the "base case" given in Part (b). In fact that's not

needed. The reason that Part (b) was there was mainly as a warm-up for part (c), but it's also correct to use it as a base case if you want.

Algorithm Analysis

7. **(12 pt.) [FIF Caching]**

Consider the following caching problem. Suppose we are given:

- A sequence $p_1, \ldots, p_n$ of pages in memory that an algorithm is going to ask for, in order. (We know this up-front). Some of the pages might repeat, so it's possible, say, that $p_3 = p_{17}$.
- A small fast cache, which can store at most $k < n$ pages.

At each timestep t, the page p_t arrives. If p_t is not in the cache at time t, then:

- We have a *cache miss*, and have to go find p_t in memory, which is costly.
- We add p_t to our cache.
- We *evict* another page from the cache (of our choice) to make room for p_t.

If p_t is in the cache at time t, then:

- We get a *cache hit*, and can retrieve the page quickly.
- If we like, we have the option to evict a page from cache and retrieve another one from memory, although this usually wouldn't be a good idea (since it would just be extra memory calls).

Our goal is to design an *eviction policy* (that is, a schedule for which page gets evicted at each time step, if any) to minimize the number of memory accesses over all n timesteps.

In this problem, we will analyze the FIF (farthest-in-the-future) policy. This policy works as follows. At time t, if p_t is not in the cache, we find the page x in the cache that next occurs furthest in the future. That is, for each x, we compute

$$\textsc{NextTime}(x) = \min\{i > t \,:\, x = p_i\},$$

and choose the x with the largest NextTime value. By convention, if x never appears again, $\textsc{NextTime}(x) = \infty$. The policy breaks ties arbitrarily.

(a) **(2 pt.)** To make sure that you understand the algorithm, suppose that the stream of items was a, b, c, a, b, a, c; $k = 2$; and that the original cache was $\{a, b\}$. At each time-step, what is the cache after FIF has evicted an element? Was there a cache hit or a cache miss? We have filled in the first three for you.

[**We are expecting:** *The rest of the table filled in.*]

Time t	Item p_t	Cache	Hit or Miss?
1	a	$\{a, b\}$	hit
2	b	$\{a, b\}$	hit
3	c	$\{a, c\}$	miss
4	a		
5	b		
6	a		
7	c		

SOLUTION:

Time t	Item p_t	Cache	Hit or Miss?
1	a	$\{a, b\}$	hit
2	b	$\{a, b\}$	hit
3	c	$\{a, c\}$	miss
4	a	$\{a, c\}$	hit
5	b	$\{a, b\}$	miss
6	a	$\{a, b\}$	hit
7	c	$\{a, c\}$ or $\{b, c\}$	miss

(b) **(10 pt.)** Our goal is to prove that the FIF policy is optimal (that is, it minimizes the number of memory accesses). To do this, we will follow our usual recipe of showing that our greedy choices don't rule out success. More precisely, we will prove the following claim.

Claim: *Suppose that, at time t, there is an optimal eviction policy (for $p_1, \ldots, p_n$ and cache size k) consistent with the choices that FIF has made so far. Then at time $t+1$, there is still an optimal eviction policy consistent with the choices FIF has made so far.*

We will walk you through the proof. Suppose that P is an optimal policy that is consistent with the choices made by FIF at time t. We will construct an optimal policy P' that is consistent with the choices made by FIF at time $t + 1$.

To begin, define P' to do exactly the same thing as P (and hence FIF) up to time t. Then define P' to do the same thing that FIF would have done at time $t + 1$.

i. **(1 pt.)** Suppose that there is no cache miss at time $t + 1$. Explain why we are done proving the claim in this case.

[**We are expecting:** *A sentence or two.*]

SOLUTION:
Since there is no decision to make at time $t+1$, and P is already consistent with all decisions up until time t, we can choose $P' = P$, and P' will be consistent with all decisions up to time $t+1$.

Given the previous part, suppose that there is a cache miss at time $t+1$, so FIF evicts the element x that appears furthest in the future. Since we defined P' to do the same thing as FIF at time $t+1$, P' evicts x at time $t+1$. Suppose that P evicts y at time $t+1$.

ii. **(1 pt.)** Suppose that $y = x$. Explain why we are done proving the claim in this case.
[**We are expecting:** *A sentence or two.*]

SOLUTION:
If $y = x$ then we can return $P' = P$, since it made all the same decisions as FIF up until time $t+1$.

Thus, for the rest of the problem, assume that $y \neq x$.

Let t^* be the first time after $t+1$ when we can't define P' to do the same thing that P does. For example, if P would evict some w at time t^*, but w is not in the cache of P' at time t^*, then we can't define P' to do the same thing that P does.

Define P' to do the same thing as P up until $t^* - 1$. Notice that, up until time $t^* - 1$, the caches only differ by one element: P has x (having evicted y), and P' has y (having evicted x). All other cached items are the same, since P and P' made all the same decisions up until time $t^* - 1$.

We need to figure out how to define P' at time t^*. There are three cases for what can happen. You will address them in the following sub-parts.

The goal is for P and P' to have the same cache after time t^*, and for P' to make no more trips to memory than P.

iii. **(2 pt.)** Suppose that $p_{t^*} = z$ is the request at time t^*, and that z is not in the cache for *either* P or P'. Suppose that P chooses to evict x and replace it z. How would you define P'' behavior in this case?
[**We are expecting:** *A clear description of what P' would do, and why it satisfies* ***the goal*** *above.*]

SOLUTION:
P' should evict y and replace it with z. After this change, P and P' have the same cache.

iv. **(3 pt.)** Suppose that $p_{t^*} = y$. Then P has a cache miss. Suppose that P chooses to evict an element w. How would you define P'' behavior in this case?
[**HINT:** *Recall that it's okay for a schedule to evict something even if there's not a cache miss.*]
[**We are expecting:** *A clear description of what P' would do, and why it satisfies* ***the goal*** *above.*]

SOLUTION:
If $w = x$, then P' should do nothing. Then P' has one fewer trip to memory than P, and the caches are the same (P just replaced x with y, and P' had y instead of x already).
If $w \neq x$, then P' should make an extraneous trip to memory; it should evict w and include x instead. Now P' and P both have the same number of trips to memory, and they have the same cache (they both have both x and y, and neither has w).

v. **(3 pt.)** The third case is that $p_{t^*} = x$. Explain why this can't happen.
[**We are expecting:** *A clear explanation about why this can't happen.*]

SOLUTION:
By the definition of FIF, the next time y appears is before the next time x appears. So this case can't happen at time t^*, since the previous case would have triggered t^* earlier.

Altogether, these cases show us how we can define a schedule P' at time t^* so that P and P' have the same cache at time $t^* + 1$. After that, we can define P' to do the same thing that P would have done, since P' will face all the same situations going forward.

Now that we have fully defined P', we see that P' is a policy with no more memory accesses than P, so it's still optimal. Moreover, at time $t + 1$, P' made the same decision as FIF. This proves the statement. So FIF is optimal – hooray!

This is the end of the exam!

This is the end of the exam! You can use this page for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for, and make a note on the problem page itself.

This page is for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for, and make a note on the problem page itself.

This page is for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for, and make a note on the problem page itself.

This page is for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for, and make a note on the problem page itself.