Note for Fall 2025: This is based on an old exam from a previous quarter; it may have covered slightly different things and had different HW problems, which would make these problems differently easy or hard than they are this quarter. But they are all good practice problems!

Instructions

- Answer all of the questions as well as you can. You have three hours.
- The exam is **non-collaborative**; you must complete it on your own.
- 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.

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 **105 + 3** (bonus) total points on this practice exam. There are **five problems** across **17 pages**.

Name and SUNet ID (please print clearly):

SOLUTION

---

1. **(20 pt.) [Multiple Choice!]** For each of the parts below, clearly shade in **all** of the answers that are correct:

●	○	🪐 ∅ ☺
Filled in means: "This is a true statement"	Empty means: "This is **not** true statement"	Anything else may be marked as incorrect!

Unless stated otherwise, we are always referring to worst-case analysis guarantees. Do not make any assumptions that are not stated in the problem.

Grading note: Each of the "main" answers is worth **one point**. The "None of the above" option on its own is not worth any points, it's just there so that you can register that you intentionally didn't select anything. If you select "None of the above" and any other answer, we will ignore your "None of the above". Ambiguously filled in answers will be marked as incorrect.

[**We are expecting:** *For each part, clearly filled-in answers. No justification is required or will be considered when grading. Ambiguously filled-in answers will be marked as incorrect.*]

(a) **(4 pt.)** Which of the following quantities are $O(n^2)$? Fill in all that apply.

- ◯ $g(n) = 2^{\log_4 n}$
- ◯ $T(n)$, where $T(n) = 2T(n-1) + 1$ for $n \geq 1$, and $T(0) = 1$.
- ◯ The worst-case running time of QuickSort
- ◯ The maximum number of items that can be inserted into a Red-Black tree before its height might (in the worst-case) exceed $100 \log n$
- ◯ None of the above.

SOLUTION: T, F, T, F.

- A is true since $g(n) = 2^{\log_2(n)/\log_2(4)} = 2^{\log_2(n)/2} = n^{1/2} = \sqrt{n}$, which is indeed $O(n^2)$.
- B is false. A quick way to get some intuition for this is to note that it looks like it scales similarly to $T(n) = T(n-1) + T(n-2)$ (aka, the Fibonacci numbers), and we saw in class that those grow exponentially quickly. To get

more confidence in our answer, we can unroll the recurrence to see:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2(T(n-2)+1)+1 \\ &= 4T(n-2)+3 \\ &= 4(2T(n-3)+1)+3 \\ &= 8T(n-3)+7 \\ &= 8(2T(n-4)+1)+7 \\ &= 16T(n-4)+15 \\ &\text{etc} \end{aligned}$$

and looking at the pattern, we can guess that

$$T(n) = 2^j T(n-j) + 2^j - 1.$$

(If we wanted to be really sure, we could prove this formally by induction). Plugging in $j = n$, we get that $T(n) = 2^n + 2^n - 1 = 2^{n+1} - 1$, which is way bigger than $O(n^2)$.

- C is true by our analysis of Quicksort.
- D is false, because we had a lemma that said that if N items are inserted into an RB tree, it will have height at most $2\log_2(N+1)$. This implies that we can insert, say, n^{49} items into an RBTree and get height at most $2\log_2(n^{49}+1) \leq 100\log_2 n$. So "the max number of items that we can insert into an RBTree before the height might exceed $100\log_2 n$" is at least n^{49}. In particular, it is *not* $O(n^2)$.

(b) **(4 pt.)** Recall the 0/1 Knapsack problem from lecture, where the objective is to maximize the value of the knapsack, given that we have a single copy of each of n items. Which of the following are true? Fill in all that apply.

- ◯ The optimal solution can always be obtained using a greedy algorithm that greedily takes the items with the best value-to-weight ratio.
- ◯ Let $K[k]$ denote the maximum value you can fit into a knapsack of capacity k. Then there is an $O(n)$-time DP solution to 0/1 Knapsack that uses the subproblems $K[k]$.
- ◯ Any correct DP solution to this problem *must* be implemented in a top-down way.
- ◯ Any correct DP solution to this problem *must* be implemented in a bottom-up way.
- ◯ None of the above.

SOLUTION: None of the above. A is false because the greedy algorithm may fail (we saw a counter-example in class). B is false since we need some way to "remember" which items have been used before – we saw this in class (and we saw a 2-dimensional DP way to do it). Both C and D are false: we can implement a DP solution as either top-down or bottom-up.

(c) **(4 pt.)** Let $G = (V, E)$ be a weighted directed graph. The shortest path from a node $s \in V$ to a node $t \in V$ will remain unchanged if:

- ◯ Each edge weight $w(v, u)$ is replaced by $C \cdot w(v, u)$ for a constant $C > 0$.
- ◯ Each edge weight $w(v, u)$ is replace by $w(v, u) + C$ for a constant $C > 0$.
- ◯ Each edge weight $w(v, u)$ is replace by $w(v, u) - C$ for a constant $C > 0$.
- ◯ Each edge weight $w(v, u)$ is replace by $w(v, u)/C$ for a constant $C > 0$.
- ◯ None of the above

SOLUTION: T,F,F,T. First, note that A and D are actually the same (by replacing C with $1/C$), so it's enough to just argue that A is correct. To see this, notice that if a path P had cost x, and P' had cost $x' > x$, then after we update the weights, the paths have cost Cx and Cx', respectively; in particular, the cost of P is still smaller than the cost of P'. So the shortest path stays the same.

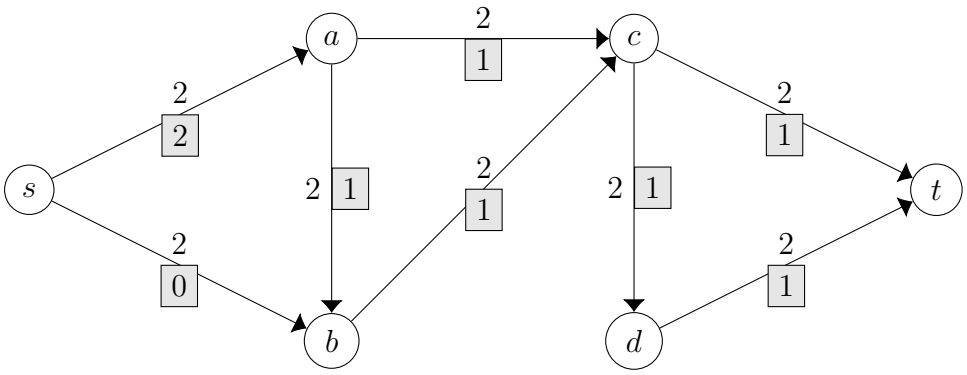
To see that B and C are false, notice that by adding or subtracting a constant from each edge weight, the cost of a path P will change by $C \cdot$ [number of edges in P]. So if we have a path P that has two edges with weights 1 and 1, and then we have a path P' with only one edge of weight 2.1, then P would be a shorter path than P' before adding C. But after adding $C = 1$ (say), P has cost $(1+1)+(1+1) = 4$, while P' has cost $2.1 + 1 = 3.1$. So now P' is shorter than P. (A similar example works when C is negative).

(d) **(4 pt.)** Which of the following algorithms runs in $O(n^2)$ time? (Note: A fully-connected graph is a graph in which there is an edge between every pair of nodes.)

- ◯ Bellman-Ford on a fully-connected directed graph with n nodes.
- ◯ Kosaraju's algorithm on a fully-connected graph with n nodes.
- ◯ Floyd-Warshall on a connected graph with n nodes, so that each node has degree exactly 3.
- ◯ Dijkstra's algorithm (implemented with a Red-Black tree) on a directed acyclic graph with n nodes and positive edge weights, so that each node has up to 7 outgoing edges.
- ◯ None of the above.

SOLUTION: F, T, F, T. A is false, since Bellman-Ford runs in time $\Theta(nm) = \Theta(n^3)$ in the worst-case. B is true, since Kosaraju's algorithm takes time $O(n + m) = O(n^2)$ (since $m = O(n^2)$). C is false, since the Floyd-Warshall algorithm takes time $\Theta(n^3)$ in the worst-case. D is true, since Dijkstra's algorithm takes time $O((n + m) \log n))$ with an RB-tree. Since each node has up to 7 outgoing edges, the number of edges is $m \leq 7n$, so this is $O(n \log n)$, which is $O(n^2)$.

(e) **(4 pt.)** Consider the graph G given below, where the labels without boxes are capacities and the labels with boxes are flows in a flow from s to t. Which of the following are true?



- ◯ $s \to a \to c \to t$ is an augmenting path from s to t.
- ◯ $s \to b \to a \to c \to t$ is an augmenting path from s to t.
- ◯ The flow shown here is a maximum flow.
- ◯ The minimum cut in this graph has value 4.
- ◯ None of the above.

SOLUTION: F, T, F, T. Explanation: A is false because the edge from s to a is full and so doesn't exist in the residual graph.

B is true by the definition of an augmenting path: none of the "forwards" edges are full, and the one "backwards" edge (a, b) is not empty.

C is false. One way to see this is that in B we found an augmenting path, so we must be able to increase the flow. Another way to see it is just to find a bigger flow: this flow has value 2, but the maximum flow has value 4 (set the flow on all edges to 2, except set $a \to b$ to 0).

D is true: we can prove this to ourselves by noticing that there is a flow of value 4 (mentioned above), and also a cut of cost 4 (e.g., cut both edges coming out of s), and so the min-cut-max-flow theorem (more precisely, the corollary of it that we saw in class) implies that the min cut and max flow are both 4.

2. **(20 pt.) [True or False?]** For each of the parts below, answer either True or False, and give a few sentences of explanation.

   [**We are expecting:** *Your answer (True or False) clearly stated, as well as a few sentences of justification. Ambiguous answers (e.g., where both True and False appear on the page) will be marked as incorrect.*]

   (a) **(4 pt.)** True or False (and explain): It is possible to detect whether or not a graph is bipartite in time $O(n + m)$.

   SOLUTION: This is true; we saw this in class as an application of BFS. (Note for Fall 2025: we skipped bipartite matching in class, so we probably wouldn't put this on an exam; but it's a good exercise to test your understanding of BFS even if you haven't seen it before! How would you detect this using BFS?)

   (b) **(4 pt.)** True or False (and explain): The SCCs returned by Kosaraju's algorithm can be different based on which node the algorithm starts at.

   SOLUTION: False. Any graph has a unique decomposition into SCCs; this follows from the fact that "there is a path from v to u and a path from u to v" is an equivalence relation on the vertices.

   (c) **(4 pt.)** Suppose that you are working on a "Maps" app for the Stanford campus. You are tasked with developing an algorithm to find the fastest route from any point A on campus to any other point B. To model the problem, you make a graph that represents all of the locations on campus, and an estimate of the time it takes to get between any points that are directly connected by a road. To get the estimates, you ride your bike between all the pairs of vertices on a sunny Saturday afternoon and time yourself. After you've generated this weighted graph, you run Dijkstra's algorithm to find shortest paths.

True or false (and explain): the scenario above involves *idealization*, as we defined it in the Embedded EthiCS lectures.

SOLUTION: True, this is a very idealized setting. For example, one assumption we are making is that everyone has a bike, and everyone rides their bike at the same speed that you do. A possible unintended consequence of these assumptions are that people without bikes (or who bike slower or faster) may not be able to use your tool as intended. Another thing that we are idealizing is that we assume that the conditions and amount of traffic are the same as on a sunny Saturday afternoon. However, on a rainy weekday during class-change periods, bike conditions and traffic may differ. An unintended consequence of this could be that people trying to use your app may either be late or bike too fast during worse conditions; the latter could be a safety hazard.

More parts on next page

(d) **(4 pt.)** Consider the following algorithm, which purports to find a minimum spanning tree in an unweighted, undirected graph G:

- Maintain a set $\mathcal{C}$ of "components", initialized to $\mathcal{C} = \{\{v\} : v \in V\}$.
- Maintain a set of edges S, which we initialize to the empty set $\emptyset$.
- Until $\mathcal{C}$ consists of just one big component:
  - Choose an arbitrary component $C \in \mathcal{C}$.
  - Let $e = \{u, v\}$ be a minimum-weight edge in E so that $u \in C$, $v \notin C$. Let C' be the component in $\mathcal{C}$ that contains v.
  - Add e to S, and merge C and C' in $\mathcal{C}$.
- Return S.

Note that this algorithm is similar to Kruskal's algorithm, except that instead of picking a minimum weight edge crossing *any* two components to add to S, we pick a minimum weight edge coming out of an *arbitrary* component.

True or False (and explain): this algorithm correctly returns a minimum spanning tree.

SOLUTION: This is true. The reason is basically the same as the reason that Kruskal's algorithm worked. The main step in the proof is to find a cut that respects S, and then show that the edge we choose greedily is light for that cut. In this case, we can use the cut between the component C and everything else: this respects S since no edges in S cross the cut, and the edge $\{u, v\}$ is light for this cut because of the way we chose it.

another part on next page!

(e) **(4 pt.)** Suppose that $a_1, \ldots, a_k$ are positive integers. For a positive integer x and $j \leq k$, let $D(x, j)$ denote the number of ways to write x as the sum of numbers in $\{a_1, \ldots, a_j\}$, where each number is used at most once and order doesn't matter.

> **For example**, if $a_1 = 1, a_2 = 2, a_3 = 3$ and $a_4 = 4$, then there are two ways to make $x = 5$ out of $\{a_1, a_2, a_3, a_4\}$, namely $5 = 2 + 3 = 1 + 4$. Thus, $D(5, 4) = 2$.

You want to use the subproblems $D(x, j)$ to design a dynamic programming algorithm to compute $D(x, k)$, the number of ways to write x as a sum of the numbers $a_1, \ldots, a_k$.

True or False (and explain): The following relationship is correct.

$$D(x, j) = D(x - a_j, j - 1) + D(x, j - 1)$$

If your answer is True, explain why; if it is False, explain what's wrong and how to fix it. (Don't worry about base cases).

SOLUTION: True, this is correct. The reason is that there are two things that can happen: either a_j is involved or it is not. If it is involved, the number of ways to make x out of $\{a_1, \ldots, a_j\}$, *including* a_j, is the same as the number of ways to make $x - a_j$ out of $\{a_1, \ldots, a_{j-1}\}$, aka $D(x - a_j, j - 1)$. On the other hand, if it's not involved, then the number of ways is $D(x, j - 1)$. Since we want the total number of ways, we add the two together.

3. **(20 + 3 BONUS pt.) [How to do it?]** For each of the following, describe how to accomplish it. You may use any result/algorithm from class as a black box, but you must state clearly how you are using it (e.g., what are the inputs). If you modify a result/algorithm from class, clearly state how you would modify it.

   We have done the first one for you to give you an idea of what we are expecting.

   [**We are expecting:** *For each, a clear paragraph explaining how you would do it, following the guidelines above. You may use pseudocode if it helps you be clear, but it is not required. An explanation about why your approach is correct may be considered for partial credit, but is not required.*]

   (-) **(0 pt.) (Example)** Given a directed, weighted graph G with n vertices and m edges, possibly with negative edge weights, detect whether or not there is a negative cycle in G in time $O(n^3)$.

   Answer: Run the Floyd-Warshall algorithm for n steps, and suppose that $D^{(n)}$ is our final array. For each vertex v, check to see if $D^{(n)}[v, v] < 0$. If you find such a v, output "negative cycle!" If there is no such v, output "No negative cycle!"

   Expanation for partial credit, just in case: This works because we showed in class that $D^{(n)}[v, v]$ is the length of the shortest path between v and v, using the vertices from $1, \ldots, n$. So $D^{(n)}[v, v]$ is negative if and only if there is a negative cycle containing v.

   (a) **(5 pt.)** Suppose that there are n cities, some of which are connected by roads. Each road (say from city A to city B) takes one hour to traverse, and costs $w(A, B)$ dollars in tolls. Find the cost of the cheapest path from city S to city T that takes at most four hours to traverse, or return `None` if no such path exists.

   SOLUTION: Run Bellman-Ford for four steps, starting at city S, and return $d^{(4)}[T]$, using the notation from Lectures 11/12. We saw in class that after step i, Bellman-Ford keeps track of the shortest path from S to any other vertex, among all of the paths that contain at most i edges. Since each edge takes exactly one hour, this is exactly the problem we want to solve, for $i = 4$.

More parts on next page!

(b) **(5 pt.)** Suppose that users on a social media platform are represented by a directed graph $G = (V, E)$ with n nodes and m edges. Users are the nodes. If there is an edge $(a, b) \in E$ from a to b, it means that user b follows user a on the social media platform. Say that b is *downstream* of a if there is a directed path from a to b in G is *downstream* of a if there is a directed path from a to b in G.

You create a new account and haven't followed anyone yet. Given G, give an algorithm that will find the smallest set of users you can follow to make sure that you are downstream of all users on the platform.

Your algorithm should run in $O(m + n)$ time.

SOLUTION: Run Kosaraju's algorithm to find the SCC DAG of G. Then iterate through all of the SCCs. If there are no outgoing edges from an SCC C, follow a user in C.

(c) **(5 pt.)** Let $G = (V, E)$ be a directed weighted graph, possibly with negative weights, and vertex set $V = [v_1, v_2, \cdots, v_n]$. The *all-pairs shortest path matrix* (APSPM) of G is an $n \times n$ matrix A where $A[i, j]$ is the shortest path distance in G between v_i and v_j.

Given G, its APSPM A, and a new edge $e = (v_a, v_b) \notin E$ with weight w, find the APSPM of G after adding e to E. Assume that there no negative cycles before or after e is added. Your algorithm should run in $O(n^2)$ time.

SOLUTION: For all $i, j \in \{1, \ldots, n\}$, do:

$$A[i, j] \leftarrow \min\{A[i, j], A[i, a] + w + A[b, j]\}.$$

The reason this works is that either the new shortest path uses e or it doesn't. If it doesn't, then $A[i, j]$ should stay the same, and if it does then the shortest path is the path from $v_i \to v_a \to v_b \to v_j$, which has cost $A[i, a] + w + A[b, j]$. So we take the minimum of these two things.

Note that this is very similar to our Floyd-Warshall update rule, except that we are adding a new edge, rather than a new vertex!

An alternative solution if you really want to match the Floyd-Warshall update rule is to add a new node v_{n+1}, along with edges (v_a, v_{n+1}) and (v_{n+1}, v_b), each with weight $w/2$, and then do the Floyd-Warshall update to add the vertex v_{n+1}.

Note that we don't have time to recompute A by running Floyd-Warshall from scratch, since that would take time $\Theta(n^3)$.

More parts on next page!

(d) **(5 pt.)** Suppose there are N students and N classes. Each student i has a set S_i of classes that they are interested in taking. Suppose that each student can take at most 4 classes, and each class can seat at most 30 students. Given the lists S_i, in time at most $O(N^5)$, find a way to match students to classes, so that:

- No student is in more than 4 classes and no class has more than 30 students.
- No student is in a class they aren't interested in.
- The assignment has as many student-class matches as possible.

SOLUTION: This is basically the same as the "matching ice cream to students" application of Ford-Fulkerson that we saw in class. Use the Ford-Fulkerson algorithm to find the maximum flow in the graph where:

- The vertex set consists of a source node s, a sink node t, one node per student and one node per class.
- There is an edge from s to each student i, with capacity 4.
- There is an edge from each class j to t, with capacity 30.
- For each student i, there is an edge from i to each class in S_i, with capacity 1.

Then if the maximum flow has one unit of flow from student i to class j, assign student i to class j.

The running time, if we use the $O(nm^2)$ running time for Edmonds-Karp that we discussed in class, is $O(N^5)$, since $n = O(N)$ and $m = O(N^2)$. In fact, since the max flow itself is at most N, Ford-Fulkerson will run a bit faster (time $O(m \cdot \text{flow}) = O(N^3)$), but you didn't need to say that for this problem.

(e) **(3 BONUS pt.)** Let $G = (V, E)$ be a directed unweighted graph. We say that G is "kind-of-connected" if for every $u, v \in G$, *either* there is a path from u to v, *or* there is a path from v to u (or possibly both). Given an algorithm that determines whether or not G is kind-of-connected, in time $O(n + m)$.

SOLUTION: First, run Kosaraju's algorithm to find the SCC DAG, call it G'. Topologically sort G' (in time $O(n+m)$), and suppose that the ordering of SCCs is $C_1, C_2, \ldots, C_r$. Then do:

- For $i = 1, \ldots, r-1$:

  If there is not an edge from C_i to C_{i+1} in G', return `False`.
- Return `True`

Explanation (not required for credit): To see why this works, suppose that the algorithm returns `True`. We claim that for any u, v, either u can reach v or v can reach u. Indeed, suppose without loss of generality that u appears before v in the topological sorting: say that $u \in C_i$, and $v \in C_j$ for some $j > i$. (Note that if they are in the same SCC, then they can both reach each other). Then, because the algorithm returned `True`, there is a path from u to v that goes through $C_i, C_{i+1}, \ldots, C_j$.

On the other hand, suppose that the algorithm returns `False`. We claim that there is some u, v so that neither can reach each other. Indeed, since the algorithm returned `False`, there is some i so that C_i is *not* connected to C_{i+1}. Let $u \in C_i$ and let $v \in C_{i+1}$. Then there is no way to get from v to u, since C_{i+1} appears later in the topological ordering that C_i, so there can't be any edges from v's SCC going "backwards" towards u's SCC. Also, there is no way to get from u to v, since there's no way to get from C_i to C_{i+1}: all of the edges leaving C_i must "skip" C_{i+1} and go on to a larger-indexed SCC, from which we then can't get back to C_{i+1}.

4. **(25 pt.) [Greedy Algorithms!]**

There are n final exams today at Stanford; exam i is scheduled to begin at time a_i and end at time b_i. Two exams which overlap cannot be administered in the same classroom; two exams i and j are defined to be *overlapping* if $[a_i, b_i] \cap [a_j, b_j] \neq \emptyset$ (including if $b_i = a_j$, so one starts exactly at the time that the other ends). Consider the following problem.

Input: Arrays A and B of length n so that $A[i] = a_i$ and $B[i] = b_i$.

Output: The smallest number of classrooms necessary to schedule all of the exams, and an optimal assignment of exams to classrooms.

For example: Suppose there are three exams, with start and finish times as given below:

i	1	2	3
a_i	12pm	4pm	2pm
b_i	3pm	6pm	5pm

Then the exams can be scheduled in two rooms; Exam 1 and Exam 2 can be scheduled in Room 1 and Exam 3 can be scheduled in Room 2.

(a) **(10 pt.)** Design a **greedy algorithm** that solves the following problem. Your algorithm should run in time $O(n \log(n) + nk)$, where k is the minimum number of classrooms needed.

[**We are expecting:** *Pseudocode* ***AND*** *a short English description, as well as a short justification of the running time. You do not need to prove that your algorithm is correct (yet).*]

SOLUTION: Our greedy algorithm will first sort the exams by start time. Then it will go through and assign each exam to a room where it will fit. If there is no such room, then the algorithm starts a new room:

```
def scheduleExams(A, B):
    n = len(A)
    Sort the exams by start time
    # assume that now we have inputs
    # A and B so that A is sorted.
    Rooms = []
    For i = 0, ..., n-1:
        foundARoom = False
        For r in Rooms:
             if r[-1][1] < A[i]: # if the last exam in
                                 # room r ends before
```

```
                                # exam i starts
                # schedule exam i in room r
                r.append([A[i], B[i]])
                foundARoom = True
                break
        If not foundARoom:
              # schedule this exam in a new room.
              Rooms.append([[A[i], B[i]]])
    return Rooms
```

The time this algorithm takes is the time to sort the items ($O(n \log n)$), plus the time to do two for-loops, one through the activities and one through all the rooms, so that's another $O(nk)$. Thus the total running time is $O(n \log n + nk)$, as desired.

More space on next page!

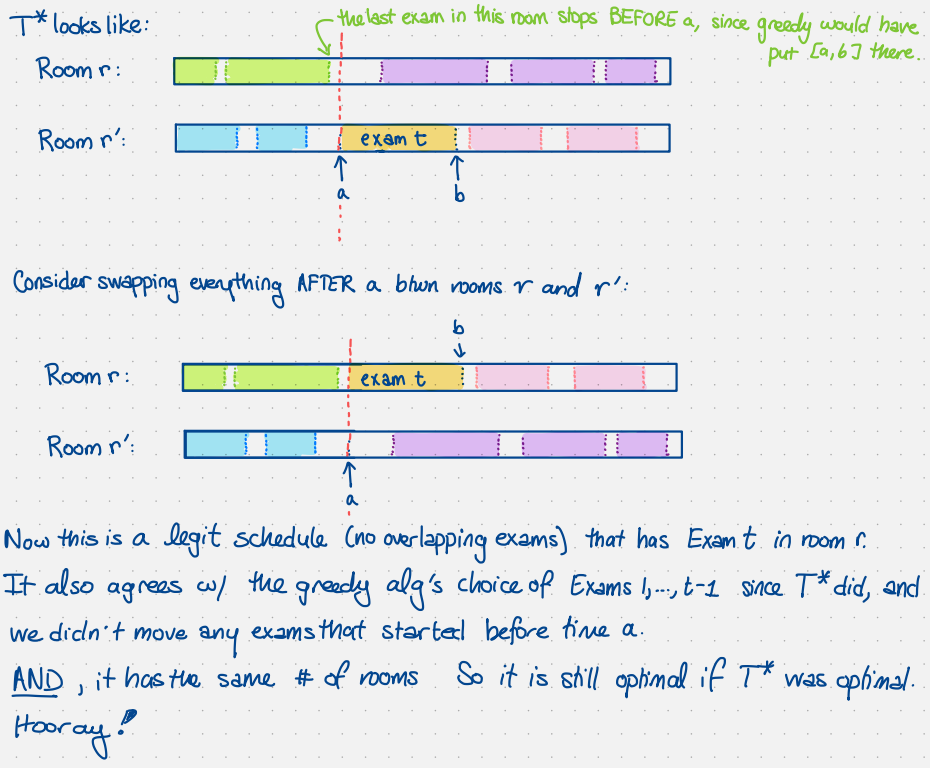
More space for your greedy algorithm!

Another part on next page!

(b) **(15 pt.)** Prove formally that your greedy algorithm from part (a) is correct. [**We are expecting:** *A formal proof. If you do a proof by induction, be sure to clearly state your inductive hypothesis, base case, inductive step, and conclusion.*]

SOLUTION: We follow the recipe that we saw from class: do a proof by induction to show that our greedy choices won't rule out success.

- **Inductive Hypothesis.** After we have assigned the t'th exam to a room, there is still an optimal solution that is consistent with the choices our algorithm has made so far.
- **Base case.** When we haven't made any assignments yet, any optimal solution is consistent with our choices so far! This establishes the IH for $t = 0$.
- **Inductive step.** Suppose that the IH holds for $t-1$; we want to show it for t. Suppose that there is an optimal solution, call it T^*, that uses k rooms, and that extends the first $t-1$ choices we have made so far. (Such a T^* exists by induction). Suppose that our t'th greedy assignment puts the exam $[a, b]$ into room r. If $[a, b]$ is scheduled in room r in T^*, then we are done, so suppose that it is not, and that T^* instead assigns $[a, b]$ to a room r'.
  At this point the clearest way to finish this step is by picture:



If we wanted to write all that out formally in words, we'd do the following, but the picture/explanation above would be enough for full credit (and would probably be preferred since it is clearer).

Suppose that in T^*, the schedule for room r looks like:

$$[x_1, y_1], [x_2, y_2], \ldots, [x_\ell, y_\ell]$$

and suppose that $[x_p, y_p]$, for some $p \leq \ell$, is the first exam that starts after time a. That is, we have

$$x_{p-1} \leq a < x_p.$$

Suppose that in T^*, the schedule for room r' looks like:

$$[w_1, z_1], [w_2, z_2], \ldots, [w_g, z_g],$$

and suppose that $[a, b] = [w_q, z_q]$ for some $q \leq g$. Now consider the schedule T that we get when we keep all of the rooms other than r, r' the same, and swap everything between rooms r and r' after time a. That is, in T, room r will now look like:

$$[x_1, y_1], \ldots, [x_{p-1}, y_{p-1}], [a, b], [w_{q+1}, z_{q+1}], \ldots, [w_g, z_g]$$

and the room r' will now look like:

$$[w_1, z_1], \ldots, [w_{q-1}, z_{q-1}], [x_p, y_p], \ldots, [x_\ell, y_\ell]$$

. We need to argue that both of these are legitimate schedules, with no overlapping exams. The only concerns are that $a \leq y_{p-1}$, or that $x_p \leq z_{q-1}$, so we need to show that neither of those occur.

To see that $a > y_{p-1}$, notice that our greedy algorithm would have put $[a, b]$ in room r, which already contained $[x_{p-1}, y_{p-1}]$, and thus $a > y_{p-1}$. Notice that we are using the assumption (from the IH) that T^* is consistent with the greedy choices we've made so far; since $x_{p-1} \leq a$ and our greedy algorithm sorts the exams by start time, we must have placed $[x_{p-1}, y_{p-1}]$ already, and thus it appears in room r in both T^* and in what our greedy algorithm did.

To see that $x_p > z_{q-1}$, notice that T^* assigned $[a, b]$ to room r' after $[w_{q-1}, z_{q-1}]$, which in particular means that $a > z_{q-1}$. But we chose p so that $x_p > a$, and we conclude that $x_p > z_{q-1}$.

Therefore T is a legitimate exam schedule, and it uses the same number of rooms as T^*. Since T^* was optimal, so is T. Finally, we observe that T is still consistent with the first $t-1$ greedy choices as well as the t'th, since T^* was consistent with those choices, and we didn't move any exams that started before time a.

Thus, there is an optimal exam schedule extending our first t greedy choices, and this establishes the IH for t.

- **Conclusion.** At the end of the algorithm, we have greedily scheduled all of the exams, and the IH tells us that there is an optimal solution extending our choices so far. The only assignment extending what we have so far is our final assignment itself, so we conclude that our assignment is optimal.

5. **(20 pt.) [Dynamic Programming!]** Suppose you have n coins with distinct integer values $c_0, c_1, \ldots, c_{n-1}$, so that $c_i > 0$ for all i. In this problem you will design a **dynamic programming** algorithm which takes as input the values $c_0, \ldots, c_{n-1}$, and an integer $k \geq 0$, and outputs the number of ways to divide the coins into two piles so that both piles have total value at least k. Your algorithm should run in time $O(nk^2)$.

> **For example**, if $n = 4$ and $k = 4$, and the four coins have values $c_0 = 1, c_1 = 2, c_3 = 4, c_4 = 5$, then the algorithm should output 8, since there are eight ways to split up the coins in to two piles, where each pile has total value at least 4: $\{1, 2, 4\}$ and $\{5\}$; $\{1, 2, 5\}$ and $\{4\}$; $\{1, 4\}$ and $\{2, 5\}$; $\{1, 5\}$ and $\{2, 4\}$; and then the same things again but swapping the piles.

(a) **(10 pt.)** What are the sub-problems you will use? What is the recursive relationship between the sub-problems, and what base cases do they satisfy?

[**We are expecting:** *A clear definition of your sub-problems, the recursive relationship that they satisfy, and a complete set of base cases. You should also give a clear explanation of why your recursive relationship and base cases hold.*]

SOLUTION: Note: Originally we posted a solution file with a solution to a different (but related) problem, where we were counting the number of ways to divide the coins into two piles with value exactly k, rather than at least k. Sorry if this caused any confusion!

Our subproblems will be $K[j, k_1, k_2]$ for $j = \{0, \ldots, n\}$ and $k_1, k_2 \in \{0, 1, \ldots, k\}$, which we define to be the number of ways to divide coins $c_1, \ldots, c_j$ into two piles so that the first one has value at least k_1, and the other has value at least k_2. (If $j = 0$, this is the number of ways to divide the empty set of coins).

The base cases are:

$$K[0, k_1, k_2] = 0 \quad \forall k_1, k_2 \text{ so that } k_1 + k_2 > 0$$

and

$$K[0, 0, 0] = 1.$$

The first base case is because there is no way to divide up the empty set into two piles so that the total value is positive. The second is because there's exactly one way to divide up the empty set into two piles so that each has at least zero value, namely that each pile is the empty set.

We also have the base cases

$$K[j, 0, 0] = 2^j,$$

because any way to distribute the j coins among two piles will result in value at least 0, and there are 2^j such ways.

In order to find the recursive relationship, we consider two cases for how we can count the ways counted by $K[j, k_1, k_2]$.

- Case 1: Coin c_j is in the first pile. The number of ways to do this is $K[j-1, k_1 - c_j, k_2]$.
- Case 2: Coin c_j is in the second pile. The number of ways to do this is $K[j-1, k_1, k_2 - c_j]$.

Since each piling counted by $K[j, k_1, k_2]$ lands in one of the above cases, we have

$$K[j, k_1, k_2] = K[j-1, k_1 - c_j, k_2] + K[j-1, k_1, k_2 - c_j],$$

with the convention that negative indexes should be treated as zero.

Another part on next page.

(b) **(5 pt.)** Write pseudocode for your algorithm.

[**We are expecting:** *Just pseudocode. You do not need to explain what it is doing or why it works, but it should use the sub-problems you defined in the previous part.*]

SOLUTION:

```
# C is an array so that C[i] = c_i
def countWays(C, k):
     Initialize an (n+1)-by-(k+1)-by-(k+1) array K
     For k_1 = 0, ..., k:
          For k_2 = 0, ..., k:
              K[0,k_1, k_2] = 0
     for j = 0,...,n:
         K[j,0,0] = 2^j
     For j = 1, ...., n:
          For k_1 = 0, ....., k:
              For k_2 = 0, ....., k:
                  K[j,k_1, k_2] = 0
                  K[j,k_1, k_2] += K[j-1, max(0, k_1 - C[j-1]), k_2]
                  K[j,k_1, k_2] += K[j-1, k_1 , max(0, k_2 - C[j-1])]
      Return K[n, k, k]
```

(c) **(5 pt.)** Now suppose that the coin values and the value k are not necessarily integers (but are still positive). Would your algorithm from part (b) still return the correct answer? If not, how would you fix it so that it would return the correct answer? Would the running time still be $O(nk^2)$?

[**We are expecting:** *The following things:*

- *Whether your algorithm from (b) would still work and an explanation.*
- *Pseudocode* ***OR*** *a high-level description of how to fix it if not.*

- *Whether or not the running time would still be $O(nk^2)$, and why or why not.*

]

SOLUTION: Our algorithm would not work, since we store things in an array, and do array accesses like $K[j, k_1, k_2]$, and if k_1 and k_2 aren't necessarily integers, this wouldn't work. One way to adapt our algorithm to still return the correct answer would be to do a top-down approach, and use a data structure like a Red-Black Tree (rather than an array) to keep track of the problems we've already solved. That is, we'd implement it like:

```
Initialize a global data structure (say, a Red-Black Tree), K,
that can store items with keys like (j, k1, k2),
and values which represent the solution to the
subproblems K[j,k_1, k_2] described above.
K should support INSERT, and SEARCH.

def topDownCoins(C, k1, k2):
    j = len(C)
    k1 = max(k1, 0)
    k2 = max(k2, 0)
    If (j, k1, k2) is in K.keys():
        return K[(j, k1, k2)]
    if k1 == 0 and k2 == 0:
        K[(j,k1,k2)] = 2^j
        return 2^j
    if j == 0:
        K[(j,k1,k2)] = 0
        return 0
    K[(j,k1,k2)] = topDownCoins( C[:-1], k1 - C[-1], k2 )
                   + topDownCoins( C[:-1], k1, k2 - C[-1] )
    return K[(j, k1, k2)]

return topDownCoins(C, k, k)
```

However, the worst-case running time might now be really slow in terms of n. Before, we knew that there were at most $k + 1$ possibilities for the arguments k_1 and k_2, because they were integers that were between 0 and k. However, now there may be as many as 2^n possibilities for each, if $k - \sum_{i \in S} c_i$ is different (and between 0 and k) for each set $S \subseteq \{0, \ldots, n-1\}$. In this case, the sub-problems wouldn't have much overlap, and our DP approach would not be very effective. Since there are $2^{\Omega(n)}$ such subsets S, that means that the running time of this algorithm could be as much as $2^{\Omega(n)}$, even if k is small.

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.

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.

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.

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.