Section 6: Midterm Review

1. Short answer

The exam will include some questions that are multiple choice, followed by a space to briefly explain your answer. This station gives you some practice, but certainly not comprehensively—review the lectures before the exam!

Some problems, marked with (KT), are from your textbook, *Algorithm Design* by Kleinberg and Tardos.

(a) (KT 1.2) Consider an instance of the Stable Matching Problem in which there exists a proposer p and a receiver r such that p is ranked first on the preference list of r and r is ranked first on the preference list of p. Then in every stable matching S for this instance, the pair (p, r) belongs to S.

◯ True

◯ False

Briefly justify your answer.

(b) Select all that apply. Running DFS on a directed acyclic graph may produce:

◯ Tree edges

◯ Back edges

◯ Forward edges

◯ Cross edges

Briefly justify your answer.

(c) A recursive algorithm for a problem that reduces the problem on inputs of size n to 2 sub-problems of the same type of size $\frac{n}{3}$ plus $\Theta(n^2)$ extra work has running time:

◯ $\Theta(n^2)$

◯ $\Theta(n^2 \log n)$

◯ $\Theta(n^{\log_2 3})$

◯ $\Theta(n^{\log_3 2})$

Briefly justify your answer.

(d) (KT 4.2(a)) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2 , thereby creating a new instance of the problem with the same graph but different costs. Then, T must still be a minimum spanning tree for this new instance.

◯ True

◯ False

Briefly justify your answer.

(e) (KT 4) Let $G = (V, E)$ be any weighted, undirected graph. Let $S \subseteq V$ be any cut, and let e be an edge of minimum weight across the cut S. Then every minimum spanning tree contains the edge e.

◯ True

◯ False

Briefly justify your answer.

2. Stable matching reduction: Horseback riding

A version of this problem appeared on the Section 1 handout.

Suppose there is a set of R riders and H horses with more riders than horses, in particular, $2H < R < 3H$. You wish to set up a set of 3 rounds of rides which will give each rider exactly one chance to ride a horse. To keep things fair among the horses, you wish for each to have exactly 2 or 3 rides.

Because it's winter, by the time the third ride starts it will be very dark, so every rider would prefer *any* horse on the first two rides over being on the third ride. Between the first two rides, each rider doesn't have a preference over time of day, and have the same preference over horses. If a rider must be on the third ride, it has the same preference list for that ride as well.

Each horse has a single list over riders, which doesn't change by ride. Since horses love their jobs, they prefer to being one of the horses on the third ride to one of the ones left home.

Design an algorithm which calls the following library *exactly once* and ensures there are no pairs (r, h) which would both prefer to change the matching and get a better result for themselves.

`BasicStableMatching`
Input: A set of $2k$ agents in two groups of k agents each with preference lists over the other group.
Output: A stable matching among the $2k$ agents.

3. Graph modeling: Moving apartments

In a certain apartment building, people are moving around. There are some people moving between units in the same building (for example, they want a bigger or smaller space), some new arrivals, and some people moving out entirely. Your job is to determine in what order everyone should move, so that for every unit, the person inside moves out before the next person moves in. Call this a "valid moving order".

In particular, you are given a list of pairs $P = [(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)]$ where each pair is the starting and ending unit of a particular person. (Apartment unit numbers are integers.) People who are moving in from somewhere else have $a_i =$ null and people who are moving out entirely have $b_i =$ null.

You may assume that in total, at most 1 person is leaving each unit, and at most 1 person is moving into each apartment. Give an efficient algorithm that returns:

- All of the pairs in a valid moving order, or
- "Not possible" and a minimal list of pairs preventing such an ordering from being possible.

Examples

- **Input**: $(1, 2)$, $(2, 3)$, $(\text{null}, 1)$, $(4, \text{null})$
  Output: $(2, 3)$, $(1, 2)$, $(\text{null}, 1)$, $(4, \text{null})$ (other solutions may exist)
- **Input**: $(1, 2)$, $(2, 3)$, $(3, 1)$, $(4, 1)$
  Output: "Not possible" because $(1, 2)$, $(2, 3)$, $(3, 1)$. (A valid moving ordering is not possible in this example because no one can be the first to move – they each need one of the others to vacate their unit first.)

(a) Describe an algorithm to solve this problem.

(b) Briefly justify the correctness of the algorithm.

(c) What the running time of your algorithm?

4. Greedy: Minimizing covers again

A version of this problem appeared on the Section 3 handout.

You have a set $\mathcal{X}$ of (possibly overlapping) integer intervals $[a, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$. You wish to choose a subset $\mathcal{Y}$ of the intervals to cover the full set. Here, cover means the union of all intervals in $\mathcal{X}$ is equal to the union of all intervals in $\mathcal{Y}$. Describe (and prove correct) an algorithm which gives you a cover with the fewest intervals.

5. Divide and conquer: Peak performance

A version of this problem appeared on the Section 4 handout.

Let $A[1\,..\,n]$ be an array of integers. Call A a *mountain* if there exists an index i called the "peak", such that $A[1] < \cdots < A[i-1] < A[i]$ and $A[i] > A[i+1] > \cdots > A[n]$. We allow the peak to be at index 1 or n (that is, a strictly increasing or strictly decreasing array is still a mountain). For simplicity, we are not allowing $A[j] = A[j+1]$ for any j. More formally, mountains satisfy:

- For all $1 \leq j < i$, we have $A[j] < A[j+1]$, and
- For all $i \leq j < n$, we have $A[j] > A[j+1]$.

(a) Given a mountain A, describe an algorithm to find the index of the peak in $O(\log n)$ time.

(b) (Extra credit) Prove that there is no deterministic algorithm for deciding in the same $O(\log n)$ running time: Given an array, whether or not it is a mountain.

(Deterministic means that we don't use randomness, like almost all algorithm we study in this class. In other words, the algorithm always has the same output when viewing the same input.)

6. Dynamic programming: Driving in Manhattan

Imagine this problem taking place in a city with a grid of one-way streets like Manhattan, but where each street only goes East or North (all routes lead to the Upper East Side). As usual, the city is under construction, so some intersections are blocked and impassible. At other intersections, as you pass through the intersection you may be able to collect a reward, or be asked to pay a toll. Turning at an intersection counts as passing through. You want to get the largest net gain possible while taking a route following the one-way streets from an intersection designated $(0, 0)$ to an intersection designated (m, n) that is m blocks North and n blocks East (if such a route even exists). (Your net gain is the sum of the rewards minus the sum of the tolls you need to pay.)

You are given this information in an 2D array of real numbers $R[1\,..\,m, 1\,..\,n]$. If $R[i, j] > 0$, then this is the value of the reward for going through this intersection. If $R[i, j] < 0$, you must pay a toll of $|R[i, j]|$ to go through the intersection. If $R[i, j] = 0$, there is no reward or toll. If $R[i, j] = -\infty$, then this intersection is impassible. You should return the maximum net gain, or if there is no valid path at all, your algorithm should return $-\infty$. Note that you do collect the rewards/pay the tolls at $(1, 1)$ and (m, n).

Design a dynamic programming solution to this problem.

(a) Define, in English, the quantities that you will use for your recursive solution.

(b) Given a recurrence relation for the quantities you have defined.

(c) Argue for the correctness of your recurrence relation.

(d) Describe the parameters for the subproblems in your recursion and how you will store their solutions.

(e) In what order can you evaluate them iteratively?

(f) Write the pseudocode for your iterative algorithm.

(g) Analyze the running time of your algorithm in terms of n and k.