Section 6: Midterm Review

In this section, we review over topics from previous sections to help prepare for the midterm exam.

1. Greedy Algorithms: Interval Covering

You have a set, $\mathcal{X}$, of (possibly overlapping) intervals, which are (contiguous) subsets of $\mathbb{R}$. You wish to choose a subset $\mathcal{Y}$ of the intervals to cover the full set. Here, cover means for all $x \in \mathbb{R}$ if there is an $X \in \mathcal{X}$ such that $x \in X$ then there is a $Y \in \mathcal{Y}$ such that $x \in Y$.

Describe (and prove correct) an algorithm which gives you a cover with the fewest intervals.

2. Divide and Conquer: Binary Search Variant

Let $A[1..n]$ be an array of `ints`. Call an array a **mountain** if there exists an index i called "the peak", such that:
$\forall 1 \leq j < i (A[j] < A[j+1])$
$\forall i \leq j < n (A[j] > A[j+1])$

Intuitively, the array increases to the "peak" index i, and then decreases. Note that either of these conditions could be vacuous if the peak is index 1 or n (e.g., a decreasing array is still a mountain).

(a) Given an array `A[1..n]` that you are promised is a mountain, find the index peak index.

(b) Can you design an algorithm with the same running time that also **determines** whether a given array is a mountain (and if it is, finds the peak)?

3. Dynamic Programming: Orienteering on a Mutilated Grid

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, some intersections are blocked and impassible. At every other intersection, you either can collect a reward, or have to pay a toll to get through the intersection. 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 array R defined on $\{0, \ldots, m\} \times \{0, \ldots, n\}$. If $R[i, j] > 0$ then this is the value of the reward for going through this intersection. If $R[i, j] < 0$ this represents a toll that you need to pay for going through the intersection. If $R[i, j] = 0$ then this intersection is impassible and you can't go through it or be at it. If there is no path at all, your algorithm should return $-\infty$.

Design a dynamic programming solution to this problem.

3.1. Define and justify a recursive solution

(a) Formulate the problem recursively – what are you looking for (in English!!), and what parameters will you need as you're doing the calculation?

(b) Write a recurrence for solving the problem you defined in the last part (the recurrence is for the answer, **not** the running time).

(c) What is your final answer (e.g. what parameters for the recurrence do you need? Is it a single value or the max/min of a set of values?)?

(d) Give a brief justification for why your recurrence is correct. You do not need a formal inductive proof, but your intuition will likely resemble one.

3.2. Write and Analyze the Dynamic Program

(a) Describe the set of parameters for the subproblems in the recursive calls for your algorithm and how you could store their solutions.

(b) Describe a computation order for those subproblems that allows an iterative solution.

(c) Write the pseudocode for an iterative algorithm

(d) State and justify the running time of an iterative solution.

4. Graph Modeling: Running Out of Rooms

You are given a list of pairs $P = [(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)]$ where each entry in each pair is either an integer or null. It is guaranteed that the non-null a_i's are distinct and the non-null b_i's are distinct. Give an efficient algorithm that gives an ordering of the pairs where if $b_i = a_j$ and both are not null, the pair (a_i, b_i) is ordered before (a_j, b_j), or returns "Not Possible" and a minimal sublist of pairs preventing such an ordering from being possible.

Sample Input I
(1,2)
(2,3)
(null,1)
(4,null)

You might return [(null,1),(1,2),(2,3),(4,null)] (there are other valid lists to return here, you only need to give one).

Sample Input II
(1,2)
(2,3)
(3,1)
(4,1)

You would return "Not Possible" and [(1,2),(2,3),(3,1)].
Such an ordering is not possible in this example because none of the pairs can be first – they each need one of the others to go first.

(a) Describe an algorithm to solve this problem.

(b) Give some intution for why your algorithm is correct. (Don't write a full proof of correctness).

(c) If your list has n people, what is the worst-case running time. Briefly (1-2 sentences) explain.

5. Practice A Reduction

Suppose that is a set of r riders and h horses with many 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. Each agent has an ordered preference list of all k members of the other group,.
Output: A stable matching among the $2k$ agents.

(a) Give a 1-2 sentence summary of your idea.

(b) Give the algorithm you're going to run.

(c) Give a 1-2 sentence summary of the idea of your proof.

(d) Write a proof of correctness.

(e) Give the running time of your algorithm; briefly justify (1-3 sentences)