Section 6: Midterm Review

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

1. Stable Matching: A Quick Proof

Is it possible to have a stable matching instance with more than 2 stable matchings? If so, give an instance and at least 3 stable matchings. If not, prove that every instance has at most 2 stable matchings.

2. Graph Algorithms: Another DFS application

In an *undirected* graph, call an edge $e = (u, v)$ a **cut edge** if u and v cannot reach each other in $G \setminus e$.

Intuitively, a cut edge is one which, when "cut" (i.e., removed) causes the graph to have more components than it did before. Cut edges are also called *bridges*.

We'll adapt DFS to find all the cut edges in a graph.

2.1. Setting the Foundation

Usually the first step in designing an algorithm is making absolutely sure you know what you're trying to find! That is, to make sure you really understand the problem. And secondly, to try to get some intuition for what the thing you're looking for "usually" looks like. The best way to do both of these steps is to look at a few examples. We'll do that before we get to designing the algorithm.

(a) We're in an undirected graph now, so edge classification will be different from lecture. Look back at the edge classification for DFS for directed graphs. Convince yourself that it's not possible to have a cross edge in a DFS on an undirected graph. (1-3 sentences of intuition, not a proof).

(b) Convince yourself it is not possible to have a forward edge in a DFS on an undirected graph. (2-3 sentences of intuition, not a proof).

(c) NOW, let's try to recognize cut edges. Prove that a cut edge must be a tree edge.

(d) Draw a graph, and a corresponding DFS, which has at least one tree edge that is not a cut edge, and at least one tree edge that is a cut edge.

2.2. Building The Bridges

Now, we're ready to start designing an algorithm.

(a) Prove that an edge is a cut edge if and only if it is not part of a simple cycle.

(b) Define $\texttt{low}(u)$ to be the smallest `pre` number that can be reached by: starting at u, following any number of tree edges, and at most one back edge. Show that if (u, v) is a tree edge, discovered going from u to v, and (u, v) is part of a cycle, then $\texttt{low}(v) \leq \texttt{u.pre}$. (Notice we're comparing v's low to u's pre!)

(c) Show that if (u, v) is a tree edge, discovered going from u to v, and (u, v) is not part of a cycle then $\texttt{u.pre} < \texttt{low}(v)$.

(d) Now, modify DFS to detect every cut edge of an undirected graph.

3. 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.

4. 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)?

5. Dynamic Programming: Longest Palindromic Non-Contiguous Substring

Given an input string s, return the length of the longest palindromic non-contiguous substring in s.

For example, the input "abcda" has a longest palindromic non-contiguous substring of length 3 ("aba", "aca", and "ada" are all different equally-long palindromic non-contiguous substrings).

5.1. Write the Dynamic Program

(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.

5.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm.

(b) Describe a filling order for your memoization structure.

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