Section 6: Solutions

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.

Solution:

Consider the following instance:

$r_1 : h_1, h_2, h_3, h_4$
$r_2 : h_2, h_1, h_4, h_3$
$r_3 : h_3, h_4, h_1, h_2$
$r_4 : h_4, h_3, h_2, h_1$

$h_1 : r_2, r_1, r_4, r_3$
$h_2 : r_1, r_2, r_3, r_4$
$h_3 : r_4, r_3, r_2, r_1$
$h_4 : r_3, r_4, r_1, r_2$

This instance has four stable matchings:
$(r_1, h_1), (r_2, h_2), (r_3, h_3), (r_4, h_4)$
$(r_1, h_1), (r_2, h_2), (r_3, h_4), (r_4, h_3)$
$(r_1, h_2), (r_2, h_1), (r_3, h_3), (r_4, h_4)$
$(r_1, h_2), (r_2, h_1), (r_3, h_4), (r_4, h_3)$

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.

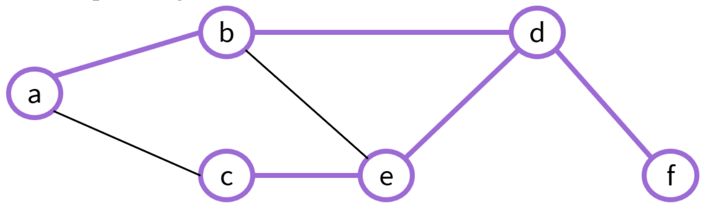
Solution:

(a) For (u, v) to be a cross-edge, we have to discover v first then get to u. But in an undirected graph, when we process v, we can view (v, u) as an edge and will discover u through it.

(b) Similarly, for (u, v) to be a forward edge, we must discover u, and (using some other edge(s)) discover v, then return to u and process the edge (u, v). In an undirected graph, when we found v, we would process all its edges, including (u, v) before popping it off the stack, so we can't get back to u to process.

(c) Let $e = (u, v)$ be a cut edge. Without loss of generality, let u be the first of u and v that DFS processes. When u is processed, it stays on the stack until all edges, including (u, v) are processed. (u, v) will be a tree edge unless DFS discovers v before (u, v) is processed. To discover v, we would need to follow a walk from u to v. If such a walk existed, then (u, v) would not be a cut edge, as u would still be connected to v. Thus we cannot discover v, and when (u, v) is processed, it will be a tree edge.

(d) There are many different examples you can come up with, but here is an example undirected graph with the corresponding DFS.



The edge (d,f) is a cut edge, while the other tree edges (a,b), (b,d), (d,e), and (e,c) are not.

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. **Solution:**

Forward direction: By contrapositive (part of a simple cycle $\rightarrow$ cut edge)
Suppose $e = (v_1, v_2)$ is part of a simple cycle $v_1, v_2, ..., v_k$. Every walk that uses e can have e replaced with $v_1, v_k, v_{k-1}, ...v_2$ and still be a valid walk (i.e., a trip "the long way" around the cycle), so the connected components of G cannot change when e is deleted.

Backward direction:
by contrapositive (not cut edge $\rightarrow$ not part of simple cycle) Suppose, for the sake of contradiction, that $e = (u, v)$ is not part of a simple cycle and (u, v) is not a cut edge. Then u and v must be able to reach each other in $G - e$. Let $v, w_1, ..., w_k, u$ be a (simple) path from v to u in $G - e$. Then adding e would give a simple cycle involving e in G, a contradiction.

(b) Define $\texttt{low}(u)$ to be the smallest $\texttt{pre}$ 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!) **Solution:**

(u, v) was a tree edge, so v was a new discovery, and there will be a DFS call starting from v made immediately. By the discovery property, v will discover every undiscovered vertex reachable from v before coming off the stack. Because (u, v) is part of a cycle, that must include returning to some already visited vertex in the cycle (possibly u itself, or possibly a predecessor of v). The `pre` number for that vertex is less than the `pre` for v since it was discovered first. Thus we have $\texttt{low}(v) \leq \texttt{u.pre}$

(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 $\texttt{u.pre} < \texttt{low}(v)$. **Solution:**

Since (u, v) is not part of a cycle, there is no way to return from u. Thus no back edge can reach any ancestor of u (if one did, combining with tree edges would give a cycle). We therefore have that $\texttt{low}(v)$ being at most $v.\texttt{pre}$. Since u discovered v, $u.\texttt{pre} < v.\texttt{pre} \leq \texttt{low}(v)$.

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

We just need to add some bookkeeping to keep track of low numbers.

```
DFS_Cut_Edge(u)
    Mark u as “seen”
    u.start = counter++
    u.low = u.start
    For each edge (u,v) //leaving u
        If v is not “seen”
            DFS(v)
            If(v.start < u.low)
                u.low = v.start
            Else
                mark (u,v) as a cut edge
            End If
        Else
            if(v.low < u.low)
                u.low = v.low
            End If
        End If
    End For
    u.end = counter++
```

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.

Solution:

Key Idea Taking the earliest-start-time (breaking ties with the longest interval) will guarantee all points are covered and will “stay ahead” of any other solution.

Algorithm

```
function IntervalCovering(()$\mathcal{X}$)
```

```
$\mathcal{Y} \leftarrow \varnothing$
Sort $\mathcal{X}$ by increasing start, breaking ties by decreasing end.
**while** $\mathcal{X} \neq \varnothing$ **do**
    Let $I$ be first element remaining in $\mathcal{X}$
    $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{I\}$
    Delete all elements of $\mathcal{X}$ that overlap with $I$
```

Correctness Let ALG $= a_1, a_2, ..., a_k$ be the list of intervals found by the algorithm, and let OPT $= o_1, ..., o_j$ be the list of intervals in an optimal cover. In both cases, let these lists be sorted by increasing start time.

We claim the following:

Lemma 1. *for all i,* $\texttt{END}(a_i) \geq \texttt{END}(o_i)$.

Proof. Base Case: Let ℓ be the left-most point of any interval in $\mathcal{X}$. To be valid covers, both ALG and OPT must cover ℓ. Since the algorithm starts by sorting $\mathcal{X}$, the first step of ALG chooses an interval covering ℓ. By the tie-breaking of the sort, $\texttt{END}(a_1)$ is the right-most point in any interval containing ℓ. Since OPT also covers ℓ, in sorted order o_1 must be an interval covering ℓ, thus $\texttt{END}(a_1) \geq \texttt{END}(o_1)$.

IH: Suppose $\texttt{END}(a_k) \geq \texttt{END}(o_k)$.

IS: Let ℓ_{k+1} be the left-most point in $\mathcal{X}$ not covered by any of $a_1, ..., a_k$. By IH, o_k also does not cover ℓ_{k+1}. Since OPT is a valid cover and sorted, o_{k+1} must cover ℓ_{k+1}. Now, consider the execution of the algorithm: when it added o_k, it deleted all elements overlapping with a_k (and already deleted everything earlier), thus it considered only intervals containing ℓ_{k+1} and chose the one which reached the farthest right, by the sorting order. Thus, o_k was an option for the algorithm, and it chose the farthest-right-reaching, so we have $\texttt{END}(a_{k+1}) \geq \texttt{END}(o_{k+1})$. □

With the Lemma proven, we observe that ALG is a minimum-sized cover. By construction, ALG covers every point in $\mathcal{X}$. Until the last interval a_k is added to ALG, there is still a point not covered by $\mathcal{Y}$ (as we delete all intervals that have all points covered); by the lemma $\texttt{END}(\text{ALG}_{k-1}) \geq \texttt{END}(\text{OPT}_{k-1})$, so OPT is not a cover until the final interval is added. Thus both OPT and ALG must contain the same number of intervals, and ALG is also optimal.

Running Time Note that the whole algorithm, including the deletion step, can be performed with a simple iteration — intervals in $\mathcal{X}$ will overlap with I if and only if their start time is before I's end-time. Since $\mathcal{X}$ is already sorted by start time, every element is either I for some interval or is deleted in $\mathcal{O}(1)$ time by the deletion command, so the function runs in $\mathcal{O}(n)$ time total.

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)?

Solution:

(a) Key idea: adapt binary search – by looking at three consecutive elements, we can see if we're on the "upward" or "downward" slope and find the peak.

```
**function** PeakFinder(A, i, j)
    **if** $j - i \leq 2$ **then**
```

```
        For each $i \leq k \leq j$, check if $A[k]$ satisfies the definition of peak in the range $i..j$.
        **return** the first element that does.
    Mid $\leftarrow i + \lfloor \frac{(j-i)}{2} \rfloor$
    **if** $A[\texttt{Mid} - 1] < A[\texttt{Mid}] \wedge A[\texttt{Mid}] < A[\texttt{Mid} + 1]$ **then**
        **return** PeakFinder($A$, Mid, $j$)
    **else if** $A[\texttt{Mid} - 1] > A[\texttt{Mid}] \wedge A[\texttt{Mid}] > A[\texttt{Mid} + 1]$ **then**
        **return** PeakFinder($A$, $i$, Mid)
    **else**
        **return** Mid
```

For correctness, observe that $A[\texttt{Mid}-1] > A[\texttt{Mid}] \wedge A[\texttt{Mid}] < A[\texttt{Mid}+1]$ is impossible in a mountain array, so in the "else" branch, $A[\texttt{Mid}]$ is greater than both $A[\texttt{Mid}+1]$ and $A[\texttt{Mid}-1]$. We will argue by induction that if the array $A[i..j]$ is a mountain, then the return value of `PeakFinder` is the peak. For the base case, we do a brute force search, so the

IH: Suppose for all arrays where $j - i < k$ and $A[i..j]$ is a mountain that `PeakFinder`(A, i, j) returns the peak. $(k \geq 3)$

IS: Let i, j be integers such that $j - i = k$ A be an array such that $A[i..j]$ is a mountain. Since $j - i = k \geq 3$, the code goes to the recursive case. If `Mid` is the peak, then we hit the else case, and return `Mid` as required. Otherwise, we have two cases:

Case 1: `Mid` is before the peak
Then since $A[i..j]$ is a mountain, $A[\texttt{Mid}-1] < A[\texttt{Mid}] \wedge A[\texttt{Mid}] < A[\texttt{Mid}+1]$. Thus we make a recursive all on `Mid`, j. By the assumption for this case, the peak is still in the range chosen by the recursive call. Thus, the remaining array is still a mountain with the peak in the desired range. Furthermore, since $k \geq 3$, `Mid` and i are different indices, so the subarray is smaller. By IH, the result of the recursive call is therefore the peak of the subarray (and thus also of $A[i..j]$), as required.

Case 2: `Mid` is after the peak
Is symmetric to case 1, with the code making the recusive call on i..`Mid`, where the peak will be.

In both cases, we have completed the inductive step.

Running Time: We do constant work (calculating `Mid`, checking inequalities, and setting up a recursive call) before making a recursive call. The recursive call is (up to rounding) $1/2$ the size of the original array.

Thus the running time has the recurrence $T(n) = \begin{cases} T(n/2) + O(1) & \text{if } n \geq 3 \\ O(1) & \text{otherwise} \end{cases}$ which (by recognizing it as the binary-search recurrence or solving) has a closed form of $\mathcal{O}(\log n)$.

(b) No. You need to examine **every** element of the array to see if it's a mountain. To see why, suppose that you have examined all elements except for the one at index u (the "unknown" element). For simplicity, assume that $u \neq 1$ and $u \neq n$. Furthermore, suppose that so far it is consistent with being a mountain. That is there is an index i such that $\forall 1 \leq j < i(A[j] \leq A[j+1] \vee j = u)$ and $\forall i \leq j < n(A[j] \geq A[j+1] \vee j = u)$.

We will consider two cases; in every case we show that depending on the value of $A[u]$, the array may or may not be a mountain.

Case 1: The index i is u
If $A[u]$ is set to be $\max\{A[u-1], A[u+1]\} + 1$, then all conditions will be met, as we've made u a peak (index u also satisfies $A[u] < A[u+1]$ and $A[u] > A[u+1]$, so the condition is met without needing the $j = u$ option).

If $A[u]$ is set to be $\min\{A[u-1], A[u+1]\} - 1$, then we will not satisfy the mountain property. u cannot be a peak, as $A[u-1] \not\leq A[u]$. No $i < u$ can be a peak, as $A[u] < A[u+1]$ violates the peak condition. Similarly, no $i > u$ can be a peak as $A[u-1] > A[u]$, which violates the first condition.

Case 2: The index $i \neq u$
Observe that there is only one such index i: for some i and i', with $i < i'$, for i to be a peak, $A[i]$ $A[i']$; for

i' to be a peak, $A[i'] > A[i]$, and only one of these can be true.

If $u < i$, we can make A not a mountain by setting $A[u] = A[u+1]+1$. Then $A[u] > A[u+1]$ and i is not a peak. Setting $A[u] = A[u+1]$ guarantees that u satisfies the conditions and makes it a peak. For $u > i$, setting $A[u] = A[u-1]+1$ or $A[u] = A[u-1]$ gives a symmetric argument to the $u < i$ case.

In all cases, until we examine $A[u]$ we cannot determine whether $A[]$ is a mountain or not. Thus, we will need at least $\Omega(n)$ time to determine if the array is a mountain.

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? **Solution:**

Let $\mathsf{OPT}(i, j)$ be the length of the longest palindromic non-contiguous substring among indices $i, ..., j$.

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

$$\mathsf{OPT}(i,j) = \begin{cases} \max\{\mathsf{OPT}(i+1,j), \mathsf{OPT}(i,j-1), 2+\mathsf{OPT}(i+1,j-1) \text{ if } s[i]=s[j]\} & \text{if } i<j \\ 1 & \text{if } i=j \\ 0 & \text{if } j<i \end{cases}$$

(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?)? **Solution:**

$\max_{i,j} \mathsf{OPT}(i, j)$.

(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. **Solution:**

For the first case, the maximum palindromic non-contiguous substring between i and j will be the maximum of up to 3 possibilities: it could be the length of the maximum palindromic non-contiguous substring between $i+1$ and j or between i and $j-1$, or if $s[i]$ and $s[j]$ are the same character it could also be 2 + the length of the maximum palindromic non-contiguous substring between $i+1$ and $j-1$. If $s[i]$ and $s[j]$ aren't the same character, then we just want the maximum length from one step previous (either $\mathsf{OPT}(i+1, j)$ or $\mathsf{OPT}(i, j-1)$). But if $s[i]$ and $s[j]$ are the same character, then we also need to consider the maximum length of from $i+1$ to $j-1$ and add 2 (for i and j).
For the base cases, if $i = j$ the substring has only one character, which is automatically a palindrome. If $j < i$ the substring has length 0.

5.2. Analyze the Dynamic Program

(a) Describe a memoization structure for your algorithm. **Solution:**

We need a (2D) array of size $n \times n$.

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

Outer loop i going from n to 1, inner loop j going from 1 to n.

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

Creating entry i, j requires checking at most 3 entries. Since we have n^2 entries, we need $\mathcal{O}\left(n^2\right)$ time.