Section 6: Solutions

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.

Solution:

Key Idea Take the next interval that helps, i.e. that covers a new point; among all such intervals (if more than one) take one that goes the farthest right.

Algorithm

function IntervalCovering($\mathcal{X}$)
    $\mathcal{Y} \leftarrow \varnothing$
    Sort $\mathcal{X}$ by increasing start, breaking ties by decreasing end.
    Let y be the start time of the first element of $\mathcal{X}$
    while $\mathcal{X} \neq \varnothing$ **do**
        Let $I = [s, e]$ be the element remaining in $\mathcal{X}$, with latest end time among those starting y or earlier.
        $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{I\}$
        Delete all elements of $\mathcal{X}$ with endtime e or earlier
        $y \leftarrow$first covered point past e

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 a_k, it deleted all elements that would not cover a new point, thus it considered only intervals containing ℓ_{k+1} and chose the one with the latest end time. 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 $\texttt{END}(\mathsf{ALG}_{k-1}) \geq \texttt{END}(\mathsf{OPT}_{k-1})$, 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.

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)?

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.
  $\texttt{Mid} \leftarrow i + \lfloor \frac{(j-i)}{2} \rfloor$
  if $A[\texttt{Mid}-1] < A[\texttt{Mid}] \wedge A[\texttt{Mid}] < A[\texttt{Mid}+1]$ **then**
    return $\texttt{PeakFinder}(A, \texttt{Mid}, j)$
  else if $A[\texttt{Mid}-1] > A[\texttt{Mid}] \wedge A[\texttt{Mid}] > A[\texttt{Mid}+1]$ **then**
    return $\texttt{PeakFinder}(A, i, \texttt{Mid})$
  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 $\texttt{PeakFinder}(A, i, j)$ 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 call on $\texttt{Mid}, j$, which is a subarray forming a mountain of smaller length that contains the peak. 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 $i..\texttt{Mid}$, 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 for all entries other than $A[u]$, it is known that $A[0], A[1], ..., A[i]$ is strictly increasing and $A[i], A[i+1], ..., A[n]$ is strictly decreasing.

We will show that no matter which index u is, there is a choice for $A[u]$ where the array is a mountain, and a choice where it is not. If $u = i$, then setting $A[u] = \max\{A[u-1], A[u+1]\} + 1$, $A[u]$ is larger than both of its neighbors and choosing this value makes u a peak, and thus makes A a mountain. If $u \neq i$, then set $A[u] = \frac{A[u-1]+A[u+1]}{2}$. If u was before the peak, then $A[u-1], A[u], A[u+1]$ is a strictly increasing sequence. If u was after the peak, then it is strictly decreasing. Either way, the condition for A to be a mountain is satisfied.

Thus no matter where u is, there is a choice for $A[u]$ where the array is a mountain. We may also choose $A[u] = \min\{A[u-1], A[u+1]\} - 1$. Since a mountain cannot have an entry that is less than its neighbors, this always makes A not a mountain, regardless of u's placement.

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.

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? **Solution:**

Let $\mathsf{OPT}(i, j)$ be the largest net gain possible in taking a route from $(0, 0)$ to (i, j) in the grid if one exists and $-\infty$ otherwise. The parameter i ranges from 0 to m, and the parameter j ranges from 0 to n.

(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:**

$$\mathsf{OPT}(i,j) = \begin{cases} -\infty & \text{if } R[i,j]=0 \\ R[i,j] + \max(\mathsf{OPT}(i-1,j), \mathsf{OPT}(i,j-1)) & \text{if } i>0,\ j>0 \text{ and } R[i,j]\neq 0 \\ R[i,0] + OPT[i-1,0] & \text{if } i>0 \text{ and } j=0 \text{ and } R[i,j]\neq 0 \\ R[0,j] + OPT[0,j-1] & \text{if } i=0 \text{ and } j>0 \text{ and } R[i,j]\neq 0 \\ R[0,0] & \text{otherwise} \end{cases}$$

where we assume that $-\infty + v = -\infty$ and $\max(-\infty, v) = v$ for every integer v.

(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:**

$\mathsf{OPT}(n,m)$.

(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:**

If location indexed by (i,j) is impassable ($R[i,j]=0$) then no path can use it and the value should be $-\infty$ as indicated in the first line of the recurrences. Otherwise there are at most two directions that the last step might have taken to get there from intersection $(0,0)$: north or east. If it is north then one previously was at intersection $(i-1,j)$ and if east then one previously was at $(i,j-1)$. In that case the best reward is the individual reward (or toll) at intersection (i,j) itself plus the larger of the optimal net gains from getting to intersection $(i-1,j)$ and $(j,i-1)$, whichever is not $-\infty$. The last three cases are for the boundaries of this section of the street grid where one or none of the prior directions is possible.

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. **Solution:**

There are subproblems for $i = 0,\ldots,m$ and $j = 0,\ldots n$ which could be stored in a (2D) array of size $(m+1)\times(n+1)$.

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

Outer loop i from 0 to m.
Inner loop j from 0 to n.
    Compute answer for (i,j)

(c) Write the pseudocode for an iterative algorithm **Solution:**

```
if R[0,0]==0 then OPT[0,0] = -∞
       else OPT[0,0]= R[0,0]
for i=1 to m
   if R[i,0]==0 then OPT[i,0] = -∞
       else OPT[i,0]= R[i,0] + OPT[i-1,0]
for j=1 to n
   if R[0,j]==0 then OPT[0,j]= -∞
```

```
            else OPT[0,j] = R[j,0] + OPT[0,j-1]

    for i=1 to m
        for j=1 to n
            if R[i,j]==0 then OPT[i,j] = -∞
                  else OPT[i,j] = R[i,j] + max(OPT[j-1,i], OPT[j,i-1])
    return(OPT[m,n])
```

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

Creating entry i, j requires checking at most 2 prior values, which each require $\mathcal{O}(1)$ time. Since we have nm entries, we need $\mathcal{O}(nm)$ time.

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 intuition 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.

Solution:

(a) Let each pair be a vertex. (a_i, b_i) has a directed edge to (a_j, b_j) if and only if $b_i = a_j$ and neither are null. Run DFS to find a cycle in the graph. If there is a cycle, return ”Not Possible” and the pairs in the cycle If there isn't a cycle, then there must be a topo sort. Run DFS to find a topo sort, and return that order.

(b) With this encoding, a topological sort of the nodes is equivalent to a sort that satisfies the desired condition.

(c) Building the graph is the tricky part. We create n vertices $1, \ldots, n$ associated with $a_1, \ldots, a_n$ respectively. Since the numbers $a_1, \ldots, a_n$ are arbitrary when we see a b_j we don't necessarily know which a_i if any it is associated with. To do this we create a dictionary with the a_i values where the entry for a_i stores i. For each b_j we look up its value in the dictionary. If it is there and has index i stored then we put an edge from vertex j to vertex i. We can do this dictionary creation and all the look-ups in heuristic $O(n)$ time using hash tables but worst case we can use binary search trees and do it $O(n \log n)$ time in the worst case.

Since the non-null a_i's and the non-null b_i's are distinct, each pair, as a vertex, has at most 2 connections: one for its first entry, and one for its second. Therefore there are at most $2n$ edges. Running two DFS costs $2O(V+E) = O(2(n+2n)) = O(6n) = O(n)$.

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)

Solution:

We will create a `Basic` instance with $3h$ agents representing horses and $3h$ agents representing riders.

For each horse h_i in the original instance, create three agents $h_i^{(1)}, h_i^{(2)}, h_i^{(3)}$ with copies of h_i's preference list among the riders. Each starts with h_i's original list (we will need to add some more preferences to fill that list out to length $3h$ as we see below).

For each rider r_j, create a list as follows: from r_j's original list, put $h_i^{(1)}$ followed by $h_i^{(2)}$ in place of h_i in the original list. Then at the end, add another copy of the original list with each h_i replaced by $h_i^{(3)}$.

To make the total number of riders equal to $3h$, add "dummy" riders[a] $d_1, \ldots, d_\ell$ until the number of riders and horses is equal. Each dummy will have a list of the $h_i^{(3)}$, followed by the $h_i^{(2)}$ and $h_i^{(1)}$ (the h_i can be in any order relative to each other, as long as the time-of-day ordering is followed). Finally add the dummies to the end of the preference lists of all horses (in any order).

We now have an instance with $6h$ in two groups of $3h$ agents each, representing riders and horses respectively, and every list contains all the agents in the other group. Run the `BasicStableMatching` algorithm, then delete the dummy riders, and leave any horse whose partner was deleted unmatched.

The `Basic` algorithm doesn't produce unstable pairs, so we won't either (once we delete the dummies)

(b) We claim the result is a correct assignment. First, observe that each (real) rider is matched, and no horse is free on the first two rides. Since each horse prefers the real riders to the dummies and each rider prefers any of the first two rides to the third, a dummy rider matched with a horse on the first two rides would have created an unstable pair (the horse on the first two rides with any rider assigned to the third ride). Thus no horse is free on the first two rides.

It remains to show there is no unstable pair among matched agents. Suppose, for contradiction, there is a pair r, h_i where r and h_i would both prefer to be paired on ride j (over their current state). Then, by construction of the lists, r prefers $h_i^{(j)}$ on its preference list and $h_i^{(j)}$ prefers r on its preference list. This would have been an unstable pair for the `Basic` instance. But the algorithm produces a stable matching, which by definition has no such unstable pairs, a contradiction!

(c) $\Theta(h^2)$. We have $3h$ agents on each side, so the guarantee on `BasicStableMatching` gives a $\Theta(h^2)$ guarantee for that call. All the other operations (copying lists, creating agents, etc.) can be done in time linear in the size of the final instance (since it's just copy-pasting) which is also $\Theta(h^2)$ ($\Theta(h)$ agents, each with lists of length $\Theta(h)$).

[a] a "dummy" is like a dummy for clothing (a mannequin) it *looks like* a real rider, but doesn't actually represent a real rider. Just like a mannequin looks like a real person but isn't one. Dummies are a very common tool in reductions.