Section 6: Solutions

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

1. Practice A Reduction

You have a set of r riders and h horses, but unfortunately, $2h < r < 3h$, i.e. there are many more riders than horses. You wish to setup a set of 3 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 has the same fixed 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 be one of the horses on the third ride instead of not being on a ride.

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 k horses and k riders. Each horse has a preference list of all k riders, and each rider has a preference list of all k horses.
Output: A stable matching among the k horses and k riders.

(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). You can assume `BasicStableMatching` has a runtime of $\Theta(k^2)$.

Solution:

(a) Create an instance where each horse has 3 copies (one per ride) and extra "fake" riders to balance the sides. Modify the preference list to represent what the agents want in the problem.

(b) We will create a `Basic` instance with $3h$ riders and $3h$ horses. For each horse h_i in the original instance, create three horses $h_i^{(1)}, h_i^{(2)}, h_i^{(3)}$. Each copy of the horse starts with h_i's original list.
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 $3h$, add "dummy" riders[a] $d_1, ..., d_\ell$ 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 lists of all horses (in any order).
We now have an instance with $3h$ riders and $3h$ horses, and every list contains all the other agents. Run the `BasicStableMatching` algorithm, then delete the dummy riders, and leave any horse whose partner was deleted unmatched.

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

(d) 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 because $r < 3h$. 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 a blocking pair (the horse on the first two rides with any rider assigned to the third ride). This dummy rider exists since $r < 3h$ and we know some rider must be matched with a third ride since $r > 2h$. Thus no horse is free on the first two rides.
It remains to show there is no blocking 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 a blocking pair for the `Basic` instance. But the algorithm produces a stable matching, which by definition has no such blocking pairs, a contradiction!

(e) $\Theta(h^2)$. 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.

2. Graph Modeling: Running Out of Rooms

In Madison, Wisconsin[1] essentially every apartment lease ends on August 14 with new leases beginning on August 15th. Dissatisfied with the current system (which leaves some people sleeping with their belongings in U-hauls overnight while waiting to move into their new apartment[2]), the City of Madison has given you the power to fix the problem. You will receive a list of all n people moving in the city, their current apartment (or `null` if they are newly moving to Madison from far away), and their new apartment (or `null` if they are moving far away from Madison).

Your goal is to find an ordering of people that will allow for *easy movement*. By that we mean, for every person, by the time you reach them in the list, the apartment they are moving to will have already been vacated by its current occupant (if any), ensuring no one ends up taking a nap in a U-haul. It may be that some apartments are currently vacant (or will become vacant after the move is completed); if an apartment appears only as someone's "new apartment" you may assume that it is currently vacant.

For simplicity, you may assume moving in or out of an apartment is instantaneous for each individual and that each apartment has only one tenant.

Given a list. you will return either

- `Easy Movement` and an ordering of all n people, allowing for easy movement.
- `No Easy Movement` and list of people who prevent easy movement.

For example,

Sample Input I
Swati [123 Fake St., 200 State St.]
Ewin [200 State St., 1000 Main St.]
Xin [`null`, 123 Fake St.]
Victor [567 Broadway, `null`]

You might return `Easy Movement` and [Victor, Ewin, Swati, Xin] (there are other valid lists to return here, you only need to give one).

[1] home of *the other* UW.
[2] Yes, really: https://onwisconsin.uwalumni.com/traditions/moving-day/.

Sample Input II
Swati [123 Fake St., 200 State St.]
Ewin [200 State St., 1000 Main St.]
Xin [1000 Main St, 123 Fake St.]
Victor [567 Broadway, `null`]

You would return `No Easy Movement` and [Swati, Ewin, Xin].
There is no easy movement in this example, because none of Swati, Ewin, and Xin can move 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.

Solution:

(a) Let each person be a vertex. Person A connects to person B if and only if person B is going to move into A's current apartment.
Run DFS to find a cycle in the graph. If there is a cycle, return No Easy Movement and people in the cycle If there isn't a cycle, then there must be a topo sort. Run DFS to find a topo sort, and that should be the order people move in
(b) INTUITION
(c) O(n). Because each tenant only moves to one apartment and each apartment only has one tenant move-in, so it has at most 2 connections. Therefore the total number of connections are at most 2n, and therefore the total number of edges are at most n. Running two DFS cost $2O(V+E) = O(2(n+n)) = O(4n) = O(n)$

3. Greedy Algorithms: Fractional Knapsack

You are given a weight capacity $W \in \mathbb{Z}_{>0}$ and n items. Each item i has a weight $w_i > 0$, and a value $v_i > 0$. You are allowed to take any fraction of an item. That is, for each item i, you may take any amount between 0 and 1 of the item, receiving proportional weight and value. Your goal is to choose a subset of items whose total weight is at most W and whose total value is maximized.

Design a greedy algorithm that always produces an optimal solution to the fractional knapsack problem. Describe the algorithm clearly and show that it is optimal.

Solution:

Greedy Algorithm. The optimal greedy strategy for the fractional knapsack problem is:

- Sort all items in decreasing order of density v_i/w_i.
- Initialize remaining capacity to W.
- Iterate through the sorted list:
  - If the current item fits entirely in the remaining capacity, take the whole item.
  - Otherwise, take exactly the fraction of the item that fills the remaining capacity and stop.

Correctness Since we can select fractional items, we can divide each item of weight w_i and value v_i into w_i individual item-chunks, each of weight 1 and value/density v_i/w_i. Now all item-chunks have equal weight. Therefore, an exchange argument tells us that the selection of item-chunks by decreasing density is optimal. Since the item-chunks derived from the same item have the same density, it follows that an iterative algorithm selecting item-chunks will continuously select as many chunks as possible from the same item. Observe that this is the greedy algorithm presented.

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`$(A, \texttt{Mid}, j)$
  else if $A[\texttt{Mid} - 1] > A[\texttt{Mid}] \wedge A[\texttt{Mid}] > A[\texttt{Mid} + 1]$ **then**
    return `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 `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 Subsequence

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

For example, the input "abcda" has a longest palindromic subsequence of length 3 ("aba", "aca", and "ada" are all different equally-long palindromic subsequences). The input "mnopqqrom" has a longest palindromic subsequence of length 6 ("moqqom" has 6 characters).

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 subsequence 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:

$\mathsf{OPT}(1, n)$ where n is the length of the input string s.

(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 subsequence between i and j will be the maximum of up to 3 possibilities: it could be the length of the maximum palindromic subsequence 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 subsequence 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 interval has only one character, which is automatically a palindrome. If $j < i$ the interval 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.