CSE 421 Section 6

Midterm Review

Administrivia

Announcements & Reminders

- HW4
  - If you think something was graded incorrectly, submit a regrade request!

- HW5
  - Was Due Yesterday, Wednesday Nov 1 @ 11:59 pm

- Midterm Exam: **Wednesday November 8@ CSE2 G20 @ 6-7:30 pm**
  - Make sure you have it saved on your calendar!
  - If you can't make it, let us know and we will schedule a conflict exam!
  - If you are sick on the day of, let us know and we will schedule a conflict exam!

Greedy Algorithms

Problem 1 – 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.

Work through this problem with the people around you, and then we'll go over it together!

Problem 1 – Interval Covering

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.

Problem 1 – Interval Covering

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.

```
function IntervalCovering(X)
    Y ← ∅
    Sort X by increasing start, breaking ties by decreasing end.
    Let y be the start time of the first element of X
    while X ≠ ∅ do
        Let I = [s, e] be the element remaining in X, with latest end time
                    among those starting y or earlier.
        Y ← Y ∪ {I}
        Delete all elements of X with endtime e or earlier
        y ← first uncovered point past e
```

Problem 1 – Interval Covering

Correctness:

Problem 1 – Interval Covering

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, END(a_i) ≥ 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, 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 END(a_1) ≥ END(o_1).

Problem 1 – Interval Covering

IH: Suppose END(a_k) ≥ END(a_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+1} 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, ok was an option for the algorithm, and it chose the farthest-right-reaching, so we have END(a_{k+1}) ≥ 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 END(ALG_{k-1}) ≥ END(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.

Problem 1 – Interval Covering

Running Time:

Problem 1 – Interval Covering

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.

Divide and Conquer

Problem 2 – 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).

Problem 2 – Binary Search Variant

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)?

Work through this problem with the people around you, and then we'll go over it together!

Problem 2 – Binary Search Variant

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

Problem 2 – Binary Search Variant

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

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 ≤ 2 then
    For each i ≤ k ≤ j, check if A[k] satisfies the definition of peak in i..j.
        return the first element that does.
    Mid ← i + ⌊(j−i)/2⌋
    if A[Mid − 1] < A[Mid] ∧ A[Mid] < A[Mid + 1] then
        return PeakFinder(A,Mid,j)
    else if A[Mid - 1] > A[Mid] ∧ A[Mid] > A[Mid + 1] then
        return PeakFinder(A, i, Mid)
    else
        return Mid
```

Problem 2 – Binary Search Variant

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

For correctness, observe that A[Mid − 1] > A[Mid] ∧ A[Mid] < A[Mid + 1] is impossible in a mountain array, so in the “else” branch, A[Mid] is greater than both A[Mid + 1] and A[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.

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 ≥ 3)

Problem 2 – Binary Search Variant

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

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 ≥ 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[Mid-1] < A[Mid] < A[Mid+1]. Thus we make a recursive call on 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..Mid, where the peak will be.

In both cases, we have completed the inductive step.

Problem 2 – Binary Search Variant

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

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

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{if } n < 3 \\ T(n/2) + \mathcal{O}(1) & \text{if } n \geq 3 \end{cases}$$

which (by recognizing it as the binary-search recurrence or solving) has a closed form of $\mathcal{O}(\log n)$

Problem 2 – Binary Search Variant

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)?

Problem 2 – Binary Search Variant

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)?

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.

Problem 2 – Binary Search Variant

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)?

A choice for A[u] where A is a mountain:

If u = I, if we set A[u] = max{A[u-1],A[u+1]} + 1, A[u] is chosen to be larger than both of its neighbors and choosing this value makes u a peak, and thus makes A a mountain.
If u ≠ 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, then it is strictly decreasing. Either way, in all cases, the condition for A to be a mountain is satisfied.

A choice for A[u] where A is not a mountain:

We may choose A[u] = min{A[u-1],A[u+1]} – 1. Since a mountain cannot have an entry that is less than both its neighbors, this 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 Ω(n) time to determine if the array is a mountain.

Dynamic Programming

Problem 3 –
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 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.

Problem 3.1 – Write 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.

Work through this problem with the people around you, and then we'll go over it together!

Problem 3.1 – Write 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

Problem 3.1 – Write 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?

Let 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.

Problem 3.1 – Write a Recursive Solution

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

Problem 3.1 – Write a Recursive Solution

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

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

Recall that $-\infty + v = -\infty$ and $\max(-\infty, v) = v$ for every integer v

Problem 3.1 – Write a Recursive Solution

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?)?

Problem 3.1 – Write a Recursive Solution

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?)?

$$\mathrm{OPT}(m, n)$$

Problem 3.1 – Write a Recursive Solution

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.

Problem 3.1 – Write a Recursive Solution

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.

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.

Problem 3.2 – Write and Analyze the Dynamic Program

a) Describe the parameters for the subproblems in the recursive calls for your algorithm and how you can 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 your iterative solution.

Work through this problem with the people around you, and then we'll go over it together!

Problem 3.2 – Write and Analyze the Dynamic Program

a) Describe the parameters for the subproblems in the recursive calls for your algorithm and how you can store their solutions.

Problem 3.2 – Write and Analyze the Dynamic Program

a) Describe the parameters for the subproblems in the recursive calls for your algorithm and how you can store their solutions.

The parameter i ranges from 0 to m, and the parameter j ranges from 0 to n.
We can use a (2D) array OPT[0..m, 0...n] of size $(m + 1) \times (n + 1)$

Problem 3.2 – Write and Analyze the Dynamic Program

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

Problem 3.2 – Write and Analyze the Dynamic Program

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

Outer loop i going from from 0 to n.
Inner loop j going from from 0 to n.
  Compute OPT[i,j]

Problem 3.2 – Write and Analyze the Dynamic Program

c) Write the pseudocode for an iterative algorithm

```
function BestNetGain(R, m, n)
   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])
```

Problem 3.2 – Write and Analyze the Dynamic Program

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

Problem 3.2 – Write and Analyze the Dynamic Program

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?)?

Creating entry i, j requires checking at most 2 prior entries and accessing 1 location in R.
Since we have mn entries, we need $\mathcal{O}(mn)$ time.

Graph Algorithms

Problem 4 – Running Out of Rooms

You are given a list of pairs $P = [(a_1,b_1),(a_2,b_2),\ldots, (a_n,b_n)]$ 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.

Problem 4 – Running Out of Rooms

Given a list, you will return either

- An ordering of the pairs satisfying the condition
- “Not Possible” and a minimal sublist of pairs not satisfying the condition

Examples:

(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.

(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 these pairs can be first – they each need one of the others to go first.

Problem 4 – Running Out of Rooms

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.

Work through this problem with the people around you, and then we'll go over it together!

Problem 4 – Running Out of Rooms

a) Describe an algorithm to solve this problem.

Problem 4 – Running Out of Rooms

a) Describe an algorithm to solve this problem.

Let each pair be a vertex. Give (a_i,b_i) a directed edge to (a_j,b_j) if and only if $a_j=b_i$ 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, there must be a topo sort. Run DFS to find a topo sort, and return that order.

Problem 4 – Running Out of Rooms

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

Problem 4 – Running Out of Rooms

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

Within this encoding, a topological sort of the nodes is equivalent to a sort that satisfies the desired condition.

Problem 4 – Running Out of Rooms

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

Problem 4 – Running Out of Rooms

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

$\mathcal{O}(n)$. Since the non-null a_i's are distinct 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

$$O(V + E) = O(2(n + 2n)) = O(6n) = O(n)$$

Stable Matching

Problem 5 – 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 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.

Problem 1 – Practice a Reduction

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 $\theta(k^2)$.

Work through this problem with the people around you, and then we'll go over it together!

Problem 1 – Practice a Reduction

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

Problem 1 – Practice a Reduction

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

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.

Problem 1 – Practice a Reduction

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

Problem 1 – Practice a Reduction

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

We will create a Basic instance with $3h$ agents for riders and $3h$ agents for horses. For each horse h_i in the original instance, create three agents 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 agents corresponding to riders equal $3h$, add "dummy" riders $d_1, \ldots, d_\ell$ until the number of agents corresponding to riders and to horses are 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 agents corresponding to horses (in any order).

We now have an instance with $k = 3h$, and every preference list contains all the k agents in the other group. Run the `BasicStableMatching` algorithm, then delete the dummy riders, and leave any horse whose partner was deleted unmatched.

Problem 1 – Practice a Reduction

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

Problem 1 – Practice a Reduction

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

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

Problem 1 – Practice a Reduction

d) Write a proof of correctness.

Problem 1 – Practice a Reduction

d) Write a proof of correctness.

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 agent prefers the real rider agents to the dummies and each rider agent prefers any of the first two rides to the third, a dummy rider agent matched with a horse agent on the first two rides would have created an unstable pair (the horse agent on the first two rides with any rider agent assigned to the third ride). This dummy rider agent exists since $r < 3h$ and we know that 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 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, the agent for r prefers h_i^j on its preference list and h_i^j prefers the agent for r on its preference list. This would have been an unstable pair for the `BasicStableMatching` instance. But the algorithm produces a stable matching, which by definition has no such unstable pairs, a contradiction!

Problem 1 – Practice a Reduction

e) Give the running time of your algorithm; briefly justify (1-3 sentences). You can assume `BasicStableMatching` has a runtime of $\theta(k^2)$.

Problem 1 – Practice a Reduction

e) Give the running time of your algorithm; briefly justify (1-3 sentences). You can assume `BasicStableMatching` has a runtime of $\theta(k^2)$.

$\theta(h^2)$. 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 $\theta(h)$).

That's All, Folks!

Thanks for coming to section this week!
Any questions?