CS 4349 Lecture—December 4, 2017

Main topics are #review.

Prelude

- Homework 11 due December 6th. It has more questions, but they should be easier to answer than the usual ones.
- YOU MUST TURN THIS ONE IN AT THE BEGINNING OF LECTURE SO WE CAN TALK ABOUT IT (or any time Monday).

Final Exam

- This week is going to be mostly review and prep for the final.
- Feel free to ask lots of questions, the "lectures" are very flexible this week.
- The final will have more-or-less the same rules as the midterm.
- You may bring a single sheet of 8.5 x 11'' paper with whatever you want written on front and back. I'll provide separate question sheets and answer sheets.
- The only rule change is that you get the full 2 hours and 45 minutes to do the exam. It's from 8:00PM to 10:45PM in this room. I'm sorry it's so late.
- I'm going to have more multi-part "knowledge" check questions closer to what's in Homework 11, but there will still be a couple "design an algorithm" questions.
- I'm going to hold extra office hours. (2-3 on Tuesday like normal, 2-3 on Thursday, 11-12 on Friday, 3-4 next Monday?) **ask for opinions**
- So for the rest of today, I'd like to review material that may appear on the exam. Wednesday will be mostly going over specific homework and lecture notes problems that you're interested in discussing. Be sure to find some! I highly recommend the exams Jeff Erickson gave in the past. Those problems are should be harder than the ones I'm giving, but they're excellent practice.

Divide-and-Conquer

- We started the semester by discussing recursion and divide-and-conquer algorithms.
- In recursion, you reduce a problem to a smaller instance of the same problem. You only need to find a base case that works and a correct reduction. And that's it. It's actually more useful if you **don't** think too hard about what goes on in recursive subproblems. It just works for the same reason induction just works.
- Divide-and-conquer is a special form of recursion that usually has three steps.
  - **Divide** the input into smaller pieces.
  - **Conquer** the smaller pieces by solving the problem on them recursively.

    - **Combine** the recursive solutions.
- Binary search is one special case of divide and conquer. You divide the input into two equal halves. You conquer one of the halves by searching it. The combine step is trivial.
- Merge sort was another example. Given an array of numbers:
    - Divide the array into two equal halves.
    - Recursively sort both halves.
    - **Merge** the two recursively sorted subarrays in linear time.
- To analyze these algorithms we usually have to write a recurrence and solve it. For example, merge sort follows the recurrence $T(n) = 2T(n/2) + n$.
- Using the Master Theorem or recursion trees, we can solve the recurrence to see $T(n) = O(n \log n)$.
- I'm going to move on now, but again, feel free to stop me at any time. Today and Wednesday are pretty flexible.

Dynamic Programming

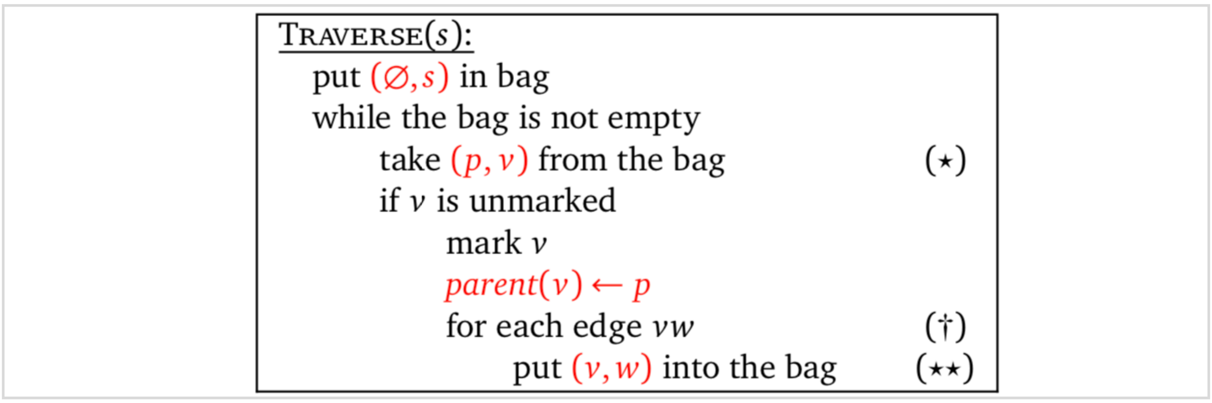
- The next major topic was dynamic programming.
- Here, you're given what is usually some kind of optimization problem.
- You first describe a function that tells you the optimal solution, and then find a recursive definition for that function.
- For example, suppose we want to compute length of the Longest Common Subsequence of $A[1 .. m]$ and $B[1 .. n]$ like on the first midterm. X is a common subsequence if it is a subsequence of both A and B.
- We can define $LCS(i, j)$ to be the length of the longest common subsequence of $A[1 .. i]$ and $B[1 .. j]$. The answer we want is $LCS(m, n)$.
- And then $LCS(i, j) =$
    - 0 if $i = 0$
    - 0 if $j = 0$
    - $\max\{LCS(i-1, j), LCS(i, j-1)\}$ if $A[i] \neq B[j]$
    - $\max\{LCS(i-1, j), LCS(i, j-1), 1 + LCS(i-1, j-1)\}$ otherwise
- Next, we describe a data structure to store the solutions to all the recursive subproblems. In this case, we can use a 2-dimensional array $LCS[0 .. m][0 .. n]$.
- We then figure out the dependencies and what order we can fill in the array, and that's enough to describe the algorithm.
- The running time is the number of entries in the array * how long it takes to compute each entry. So $O(mn)$ in this case.

Greedy Algorithms

- We talked about greedy algorithms.
- In dynamic programming, you describe a recurrence that considers the best outcome for all first decisions you might make, like whether that last character in A or B is going to be part of the longest common subsequence.
- In a greedy algorithm, you make your first decision and then stick with it.
- For example, we considered finding a maximum cardinality set of non-conflicting classes. This was the same as finding a maximum cardinality subset of intervals from the real line that didn't overlap.
- The optimal greedy choice is to take the class with the earliest finish time. Then you recursively pick from the classes that don't conflict.
- To prove this greedy choice is optimal, we do an exchange argument: consider some arbitrary solution X' and show how to change X' so its "first" choice is the same as ours without decreasing the quality of the solution.
- For example, given a subset of classes X', if the earliest class to finish in X' is not the earliest class to finish from all the classes, we can replace X''s first class with the first one to finish from all classes. We won't create new conflicts, because anything we're conflicting with already conflicted with X''s first class. So we have just as good a solution. By induction, the rest of our class choices are good as well for getting an optimal solution.
- Again, though, greedy algorithms are probably wrong if you want optimal solutions. You probably want to do dynamic programming instead.

Graph Algorithms

- The final subject I want to talk about today is graph algorithms.
- We started by talking about graph traversals.
- The basic traversal algorithms were all the same. We maintain some data structure holding vertices. We iteratively remove a vertex, check if it is marked, and if not, we mark it and add its neighbors to the bag.



```
TRAVERSE($s$):
    put $(\varnothing, s)$ in bag
    while the bag is not empty
        take $(p, v)$ from the bag                  $(\star)$
        if $v$ is unmarked
            mark $v$
            $parent(v) \leftarrow p$
            for each edge $vw$                       $(\dagger)$
                put $(v, w)$ into the bag            $(\star\star)$
```

- Depending on the bag we use we get different types of searches.
  - Use a stack to get depth-first-search: useful in many applications including topological sorting of tasks that must be done in some order.

    - Use a queue to get breadth-first-search: We find paths to each vertex that use as few edges as possible. In other words, shortest paths where all edge weights are 1.
- The actual paths induced by the parents give us depth-first and breath-first spanning trees. Both traversals take O(V + E) time assuming you have a connected graph.

- The next few weeks we discussed edge-weighted graphs.
- We talked about algorithms for computing minimum spanning trees: spanning trees where the sum of edge weights is as small as possible.
- All the algorithms were based on the same concept of adding *safe edges* to a spanning forest. An edge is safe if it is the minimum weight edge leaving a connected component of the forest.
- Borvka and Kruskal take O(E log V) time. Jarnik/Prim's algorithm takes O(E + V log V) time, assuming you use the right data structures.
- We also talked about algorithms for shortest paths. If you have no negative weight edges, you can use Dijkstra's algorithm in O(E + V log V) time. With negative edge lengths but no negative cycles, use Shimbel in O(VE) time.
- We also discussed some algorithms for all-pairs shortest paths: Find the shortest paths between all pairs of vertices. Floyd-Warshall runs in O(V^3) time using dynamic programming.

- Finally, we discussed maximum flows and minimum cuts. Either send as much flow from vertex s to vertex t as possible without violating capacities on the edges, or partition the vertices into two halves to separate s and t while minimizing the capacity of edges spanning the cut.
- We can solve this problem in O(VE) time using a recent algorithm of Orlin.
- And max flows and min cuts have applications to the maximum matching and assignment problems.