What Have You Learned So Far?

Yining Liu

December 5, 2018

We started by learning about *asymptotics*, which is a way to classify functions. Given two functions $f(n)$ and $g(n)$, we hope to describe the relationship between how fast they grow. You know that you can take the $\lim_{n\to\infty} \frac{f(n)}{g(n)}$: if the limit is a real number, then $f(n)$ and $g(n)$ grow at the "same" rate [1]; if the limit is 0, then $g(n)$ grows much faster [2]; if the limit is ∞, then $f(n)$ grows much faster [3].

Oftentimes, we would write runtime in terms of a recurrence relation. Thus, it's important to know how to solve recurrence relations. If it's written as $T(n) = aT(n/b) + O(n^d)$, then congrats! All you need to do is plugging in *Master Theorem*. If you're less lucky, don't worry! You can always draw the recurrence tree, write out the work for each layer, and take the summation.

Here comes our first class of algorithms: *Divide-and-conquer*. Essentially, you would want to break the problem into smaller parts that have the same "structure" as your original problem, recursively solve the smaller problems, and then combine small solutions together. How do you get the runtime? You probably would want to write the runtime in terms of a recurrence relation, which you've learned how to solve. Two important examples use of Divide-and-conquer algorithms are matrix multiplication and FFT [4].

We now moved on to learning about *graphs*. You've learned about *Depth First Search* in 61B; in 170, you learned that you could do a lot [5] by wisely modify DFS. For example, you could use pre-order and post-order to determine the type of edges. In general, post-order is more useful than pre-order, since we have more information about the entire graph during post-visit. [6] We like *DAG*s because they're *linearizable* [7]. What if your graph isn't a DAG? It's okay - you can find the SCC, which would form a "linearizable meta-graph". How do you find SCCs? Reverse all edges and run DFS :P

[1] $f(n) = \Theta(g(n))$

[2] $f(n) = O(g(n))$

[3] $f(n) = \Omega(g(n))$

[4] Fast multiplication of two polynomials in $O(n \log n)$ with the help of roots of unity

[5] Useful thing to consider when you modify DFS: do you want to do the task from top down (during pre-visit) or from bottom up (during post-visit)?

[6] This is just the intuition. Make your own decision on which one to use depending on the problem :)

[7] Intuitively, a linearized graph is nice because then you can "process" the nodes in a more organized order.

Graphs without distance are kind of borning, so we spent some time looking at graphs with edge weights, and investigated *shortest paths algorithms*. You first learned about *Dijkstra's Algorithm*. Given a graph with positive edges weights, Dijkstra's Algorithm is guaranteed to find the shortest path from a source to all other vertices in linear time. Unfortunately, if the graph has negative weights, Dijkstra might not work; the expensive edges might prevent us from seeing the good negative edges.

What if the graph has negative weights? We also learned about how to deal with this! You can use *Bellman-Ford Algorithm*, which runs $|V|-1$ iterations [8] and updates all edges during each iteration. Bellman-Ford Algorithm runs in $O(|V||E|)$, but it's still great because it works on any graph. Moreover, it can detect negative cycles: just run for one more iteration and see if anything changes.

We then talked about another graph minimization problem: how do I connect all vertices with minimal costs [9]? You know two algorithms that can achieve this for you: *Kruskal's algorithm* [10] and *Prim's algorithm* [11]. They run pretty fast [12] and you know they work because of the *cut property* [13]. You might have already realized that these two algorithms are greedy: the solution is built up by choosing the next piece that offers the most immediate benefit, without worrying about the future.

We discussed three more examples of *greedy algorithm*. The first example is *Huffman encoding*: you assign more frequent symbol shorter codeword by combining the two nodes with lowest frequencies. The second example is *Horn Formula*: you set all variables to be false, reluctantly fix the unsatisfied implications [14] and check at the end that whether pure negations are still satisfied. The third example is *set cover*: given a collection of subsets of S, you want to pick some subsets that cover S. Although the greedy strategy might not always choose the smallest number of subsets [15], it's only off by a factor of $\ln n$. By the way, how do you prove the correctness of a greedy algorithm? You can use *exchange argument*: show that you can make any solution better by making it looks more like your greedy solution.

We also talked about *dynamic programming*. Whenever you see a recursive problem, you could use dynamic programming to improve the runtime. Instead of mindlessly calling yourself on small inputs like regular recursion does, dynamic programming solves the smallest subproblems first, stores the results in a table and then uses them to solve larger ones. In order to determine the runtime, first think about the size of the table, then think about how long it takes to fill

[8] You can implement with early stopping: if nothing gets changed during one iteration, you could stop.

[9] If you think of edge weights as costs, this is saying we want to find the minimum spanning trees.

[10] Picks the next cheapest edge that doesn't create a cycle.

[11] Grows the subtree by adding the cheapest edge from tree to another vertex.

[12] $O(|E|\log|V|)$

[13] Any edge of minimal weight in a cut is in some MST.

[14] "T'' $\implies$ "F''; can be fixed by setting the variable on the RHS to be true.

[15] Unlike our previous examples where greedy algorithms give the optimal solution

in each entry.

The last topic in algorithms is *linear programming*. You need to set up a linear objective function and some linear constraints; simplex would return an optimal solution in polynomial time on average [16]. If you want to solve the linear programs, you could plot the feasible regions and choose the optimal vertex. One important application of linear programming is *max flow*: use residual graph to keep track of the available options, and stop once there're no such path. You know the solution is optimal because max-flow equals to min-cut. In fact, for all linear program, you can find a dual program, whose solution would act as a certificate of optimality [17].

Up to this point, we've been trying to find *efficient algorithms*. Are there problems that are intrinsically harder than the other ones? So, you learned about *complexity*, which teaches you a more rigorous way of comparing the difficulty [18] of search problems [19]. In general, how do you show $A \leq_p B$? [20] You need to show there' a *polynomial time reduction* from A to B: in other words, find a polynomial time algorithm that changes instances of A into instances of B.

You spent some time with dealing with NP [21], P [22], NP-complete [23] and NP-hard [24]. How do you show a problem Q is NP-hard? Pick a NP-complete problem, and reduce it to Q.

(I hope you learned a lot from this class! Take good rest, and good luck on your final!)

[16] If optimal solutions exist. For example, there's no optimal solution if the linear program is infeasible or unbounded. In the worst case, simplex takes exponential time.

[17] bound the objective function by setting up a dual to find the "best" linear combinition of constraints.

[18] Difficulty in 170 is terms of runtime; a different type of difficulty (computability) was discussed in 70.

[19] A solution can be verified in polynomial time.

[20] Finding an efficient algorithm for B is at least as hard as finding an efficient algorithm for A

[21] Search problems.

[22] NP-problems that can be solved in polynomial time; at most as hard as NP-complete.

[23] NP-problems that all NP problems can be reduced to them in poly time.

[24] Problems that all NP-problems can be reduced to them in poly time; at least at hard as NP-complete.