Lecture 13: Quiz Review

Nathan Brunelle

Please pick up a practice quiz!

Announcements

- HW 4 deadline extended to 10/29
- Quiz 1 in class on Friday 10/24

Practice Quiz

- Question 1: Divide and Conquer
- Question 2: Asymptotic Analysis
- Question 3: Algorithm Correctness
- Question 4: Stable Matchings

Algorithm Correctness

Algorithm Soundness

Q: When proving soundness, how to find out the exceptions? Could you please provide more detailed examples of what cases should be treated as exceptions and how to locate them, i.e. under what conditions are they likely to occur?

A: Some operations only have defined behavior for certain parameters. For those operations, we need to justify that when we do those operations, the arguments are within those parameters. For example, addition is defined for all integers, but division is not (it’s not defined for a denominator of 0). So we would need to justify that we never divide by zero, but the addition does not need to be addressed

Depth in Proofs

Q: Depth we need to go when proving algorithms correct.

A: The standard we’re going for is that steps should be obvious to others in the class. If the majority of the class would want further explanation of a step, then you should provide that explanation.

Proof Writing Tips

Q: Tips/advice for writing proofs would be great.

A: Really, a proof is just explaining how you know something to be true in a way that makes it clear that you know what you’re talking about. Start by convincing yourself that something is true, then carefully explain yourself using one of the strategies that we’ve discussed.

Selecting Loop Invariants

Q: How to determine loop invariants?

A: Loop invariants are simply statements that are true throughout the entire loop, so there’s a lot of flexibility in what a loop invariant might be. The important thing to keep in mind is that we’re trying to use the loop invariant to justify correctness, so we should pick one relevant for that goal. Start by thinking about what the algorithm needs to return, then consider what about the loop ensures that this value will be correct.

The idea is that loops are just doing a single action over and over again in a chain. Our loop invariant, therefore, is just trying to connect that chain from the input to the output. Keep in mind that for most loops, if you stop the loop sooner then you have an answer for a smaller version of the same problem. For this reason, loop invariants often take the form of a statement like “the algorithm is correct for the prefix i of the input”.

Algorithm Running Time

Running time and Data Structures

Q: When analyzing running time, shall we take into account all the time needed for initialing all data structures?

A: Well, the best answer is “it depends”. Recall that when doing running time we count specific operations. If initializing the data structures involves performing those operations, then it counts. If it doesn’t involve operations that you’re counting, then it does not.

Selecting Data Structures

Q: Selecting data structures for a given task (for example, the gale-shapley matching task).

A: I generally do this most as part of iterative improvements of an algorithm. I’ll have some algorithm that works, and then do a running time analysis to see where the most work is being done. After this, if there is some step that is asymptotically dominating, I’ll look to see if there’s a data structure that could make that step more efficient.

Divide and Conquer

Divide and Conquer Running Time

Q: Just want to go over the Complexity Analysis for divide and conquer.

A: Generally you want to identify 3 things: The number of subproblems you recursively solve, the size of each subproblem, and the amount of work required to identify those subproblems and combine their recursive solutions together.

For that last part (the time required to divide and combine), that's generally done using the same process as you would use for non-recursive running time analysis. You just ignore the time for the recursive steps, because that's taken care of by the recursive term in your recurrence.

D&C Recursion

Q: Could you go over the recursive part of divide and conquer algorithms?

A: Here’s the intuition I personally find most helpful for divide and conquer algorithms. It’s easier to solve problems on small inputs than larger ones, so if I can somehow express my large solution using smaller solutions then that should make things easier. When we’re dividing we’re picking smaller versions of the same problem. We then use potential solutions for those problems to help us find the answer for the big problem. Importantly, ANY way of finding solutions to those small problems is sufficient for finding the answer to the big problem. We could use some more direct way of doing it iteratively (this is the base case), or if we think our algorithm is really good we could just use that instead (recursively).

D&C Design Strategy

Q: I would like to know how to develop divide and conquer algorithms

A: Usually divide and conquer are going to be most helpful for improving the running time of algorithms. It will pretty much always be the case that some more straightforward approach will give us a correct answer, but more slowly.

For this reason, I think it's most helpful to first target a running time (that's faster than a naïve solution), then attempt to design an algorithm to achieve that running time. Once we have our running time, we can use the master theorem to identify a "pattern" / "budget" of our D&C algorithm. By this I mean that we can use that to guess how many subproblems we might have, how big each is, and how much non-recursive work we can afford to do.

If you do not know how where else to start, start by trying to break your problem in half and see where that takes you!

D&C Running Time

Q: How to analyze the time complexity in the combine part in divide and conquer algorithm?

A: This is done in the same way you would analyze the running time of any code. The combine step is likely going to be non-recursive, so you can consider that portion in isolation of the rest for running time analysis. To do this you'll want to consider the recursive output(s) as the "input(s)" to the combine part.

Karatsuba Running Time

Q: karatsuba method running time

A: The idea of our first divide and conquer multiplication algorithm is that we can take the multiplication algorithm that you learned in elementary school and adapt it to act on chucks of digits rather than on individual digits. We can then do the multiplications of the chunks recursively (i.e. using progressively smaller chunks). When doing this our algorithm defined multiplication in terms of chunks that were half the size of the original (so n/2 digits each), and we needed to find 4 different products of smaller chunks (then added together after adding some trailing zeros). Overall this gave us a recurrence of T(n)=4T(n/2)+n, for a running time of $O(n^2)$ using the master theorem

The Karatsuba method starts the same way, but just uses clever algebra to reduce the number of small-chunk multiplications from 4 down to 3, giving a recurrence of T(n)=3T(n/2)+n, for a running time of $O\left(n^{\log_2 3}\right)$. To achieve this, it needs to do more addition and subtraction compared to the previous algorithm, but that does not result in the asymptotic running time of the non-recursive steps changing (it increases the constant only).

Stable Matching

Level of Difficulty

Q: Will we being asked about the extension of the Gale-Shapley algorithm, like the hospital one in the homework, what level of difficulty will we test on Gale-Shapley algorithm

A: Level of difficulty is subjective, so I'm not going to be able to give a precise answer, really just an idea or motivation. In the syllabus we state

"Assessments will feature problems similar to standard homework, but are intended to be slightly easier given the limited time."

So the idea is that we will aim for the quiz to use the same "muscle" that you trained by completing and revising your homework. We will definitely scale down the difficulty of these tasks, however, because time is more limited, and we're trying to assess what you have already learned, rather than provide experiences to expand your understanding.