CS 4820: Introduction to Analysis of Algorithms **Spring 2023**

Review Information for Dynamic Programming

Jenny Chen

1 Introduction, what we are looking for

DP is basically recursion. We break our problem into smaller subproblems and use their results to compute the final result. We use memoization to record the results of subproblems in the DP table to prevent exponential runtime. Therefore, your algorithm for DP problems is mostly you explaining the recurrence. We will look for your definition of the DP table entries (OPT(j) for a 1D DP table) and your recurrence relationship. For me, recurrence explains the relationship between subproblems and larger problems, written in the form of some formulas. For example, in the knapsack problem, the recurrence is the formula:

$$\text{OPT}(i, w) = \begin{cases} \text{OPT}(i-1, w) & \text{if } w < w_i \\ \max(\text{OPT}(i-1, w), w_i + \text{OPT}(i-1, w-w_i)) & \text{otherwise} \end{cases}$$

And your definition of $\text{OPT}(i, w)$ should be the value of the optimal solution using a subset of the items $\{1, ..., i\}$ with the maximum allowed weight w.

A general DP algorithm will consist of the following elements:

1. Explain the definition of your DP table.
2. State the recurrence formula and explain the notation you introduced. You can also briefly explain how you came up with this formula.
3. Explain all base cases.
4. Explain post-processing (if any). For example, is $\text{OPT}[n, k]$ your final result, or do you still need to sum up $\text{OPT}[i, k]$ for all i?

Once you have all these elements, you can start to analyze the runtime and its correctness.

2 General tips

You might ask, but how do I design the recurrence? I can't provide good tips for solving all DPs. I usually start by thinking about all choices I have there. In the knapsack, you either include the object or don't. In the RNA problem, you either pair the two bases now or leave them alone. Once you're convinced that you considered all cases, try to write all the cases in formula.

When you realize the problem is more complicated than you thought, always try to add another parameter to your recurrence (add one dimension to your DP table). For example, in the knapsack, you need an additional parameter to keep track of the weights; in the RNA problem, you need two parameters to keep track of the start and end of the strand. Although this sounds useless, it's true that the more problems you do, the better you know what you're looking for. So I highly recommend looking at the solved exercises at the end of each chapter.

3 Runtime analysis

Notice how in each subproblem/iteration, we fill one entry of the DP table. Therefore, the total runtime is just the number of DP table entries times whatever runtime you need to compute one subproblem. So here's a general formula.

$$\text{total runtime} = \text{number of subproblems} \cdot \text{runtime for each subproblem}$$

We can analyze runtime using two steps:

1. Find the number of subproblems
2. Find the runtime for one iteration

Let's look at the second half of recurrence from knapsack as an example:

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), w_i + \text{OPT}(i-1, w-w_i))$$

i ranges from 0 to n, w ranges from 0 to W, where W is the maximum weight we can take. Therefore, the total number of subproblems is just nW. (You might say it's $(n+1)(W+1)$, but normally the 1 part is taken care by the initialization. Also $O((n+1)(W+1)) = O(nW)$ so there's no need to be super precise here.)

Because we're using strong induction, we assume we already obtained correct answers for all smaller subproblems. Therefore, in each iteration, we just need to get the value of $\text{OPT}(i-1, w)$ and $\text{OPT}(i-1, w-w_i)$, calculate $w_i + \text{OPT}(i-1, w-w_i)$, and record the larger number between the two values. Because we store them in arrays, getting the result of smaller subproblems is always $O(1)$. We also assume the runtime for max() is $O(1)$. Therefore, the total runtime for each iteration is $O(1)$.

Using the formula above, the final runtime would be $O(nW * 1) = O(nW)$.

4 Proof of correctness

This outlines what you should do in your proof of correctness, using a 2D DP problem as an example. I stole this outline from some other review pdf :)

1. Somewhere in your proof, you should define what you think the value of each entry in the table means. (Example: "$\text{OPT}[i, j]$ denotes the minimum number of credits that must be taken to fulfill the prerequisites for course i by the end of semester j.")
2. The induction hypothesis in the proof is that the value of $\text{OPT}[i, j]$ computed by your algorithm matches the definition of $\text{OPT}[i, j]$ specified earlier. Sometimes, drawing out the DP table may help you visualize the dependency of the entries and develop the induction hypothesis.
3. Prove the correctness of the base case: show that the values filled in during the initialization phase are correct, i.e. they satisfy the definition of $\text{OPT}[i, j]$.
4. Prove the correctness of the induction case: the recurrence used to fill in the remaining entries of OPT is also correct, i.e. assuming all previous values were correct, why the current value we computed is the correct value to fill in the table.

5. Remaining important issue: prove that the table is filled in the correct order, i.e. when computing the value of OPT$[i, j]$, the algorithm only looks at table values that were already computed in earlier steps.

Steps 3 and 4 are the most important steps here. This is where you need to convince us that your algorithm is correct. Here are a few things to talk about:

1. What are the cases that you think the algorithm should cover? Are they all possible cases?
2. How does your recurrence cover all cases?
3. How does your formula correctly compute what you described?

There's no need to answer those questions exactly, but a vague statement like "because we assumed smaller problems are correct, so this formula gives the correct answer to a bigger problem" is not enough. In other words, the content is much more important than the structure.

If your proof of correctness has everything above, you're good to go!

5 Backtracking

Very often, we simplify problems to numeric problems first and then solve them using DP. For example, in the knapsack, instead of asking for the maximum value, we can also ask about which subset of the items will give the maximum value. You cannot just "read out" the subset we want by staring at the table since we only record the maximum value. Therefore, now we need backtracking to figure out our choices in each step.

Intuitively speaking, backtracking is just going back and checking which option we chose. For this recurrence,

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), w_i + \text{OPT}(i-1, w-w_i))$$

Since now we know all the entries of the DP table, we know OPT(i, W), OPT$(i-1, w)$, and OPT$(i-1, w-w_i)$. If we chose to include w_i before, then $w_i + \text{OPT}(i-1, w-w_i)$ must equal to OPT(i, W). Similarly, if OPT$(i, W) =$ OPT$(i-1, w)$, then we know we did not include w_i in our optimal solution. We can infer our choices in each step by examining the relationship between subproblems and larger problems.

Sometimes both choices give the same result (OPT$(i, w) =$ OPT$(i-1, w) = w_i +$ OPT$(i-1, w-w_i)$). In this case, the choice doesn't matter and you can pick a random one (include or not include w_i, probably won't happen in the knapsack problem).

I won't go into detail about backtracking. The textbook can explain much better than I do. Section 6.1 and 6.3 both talked about how they did backtracking.

DP proof is weird at the beginning, but ultimately it's (strong) induction. You will get better at it when you see more problems. I hope this helps, and good luck with the homework!