Dynamic Programming Tips

Sam McCauley
Algorithms, Spring 2025

Parts of a Dynamic Program

Let's go through the parts of a dynamic program, and look at some tips on how to fill in each.

Subproblem Definition: The subproblem definition is what the entries in your dynamic programming table mean. In other words, if you fill in an entry like $M[2] = 5$, the subproblem definition says what that 5 represents. Perhaps something like "the best solution using just the first two items has cost 5". The subproblem definition is often useful to help cross-check your work (we'll mention it many times later in this document).

First, let's talk about rules and some common pitfalls. The subproblem definition should always be a number—you should not store extra information in your dynamic programming table like "what set did I use" or "what was the last item I used" or anything along those lines.[1] Furthermore, the subproblem definition should always be an value representing optimal cost of some solution to the problem—if we want the longest increasing subsequence, we'll be storing a subsequence length; if we want the largest value subset of items we'll be storing a total value of a subset of items.

Now, let's talk about how to come up with a subproblem definition for a new problem. The first thing to try is to just have the ith subproblem be solving the exact same problem we're trying to solve, but limited to first i elements of the input. If that does not work, try making changes. Should we insist that the solution ends at the ith element, or that it includes the ith element? Should we add an extra variable to the subproblem?

Bear in mind the goal of the subproblem definition. Dynamic programming is ultimately a recursive technique: we "build up" solutions to larger problems based on solutions we already calculated for smaller problems. The subproblem definition states what problems we are solving—and, in general, says how we "build up" the solution over time.

Recurrence: The recurrence is the heart of a dynamic program: usually if you have this, the remaining parts can be filled out relatively quickly.

[1] You may want to store extra information in a separate table, like when doing backtracking to find the optimal solution. But this extra information should not be a part of the recurrence.

The recurrence states how to fill out an entry in your dynamic programming table, using information from the input or other dynamic programming table entries. Let's look at two example recurrences that we've seen already in the class.

$$OPT(i) = \max\{OPT(i-1), v_i + OPT(p(i))\}$$

$$M[i,j] = \min_{1 \le i' \le i} \max \left\{ M[i', j-1], \sum_{t=i'+1}^{i} s_t \right\}$$

In these examples, we have on the left an entry of the dynamic programming table we want to fill out. On the right, we have a way to compute the value of that entry using previous dynamic programming table entries and the input.

Before filling out a recurrence, double check your subproblem definition. Make sure that you understand exactly what this number represents.

Now, let's talk about how to come up with a recurrence. I often recommend splitting into *cases*. The idea is that while we don't (yet) know what the optimal solution looks like, we can split our analysis of the optimal solution into a few simple cases, calculate the cost of each, and find whichever has the best cost.

> **Example.** Recall that in the weighted interval scheduling problem, I want to find the highest-value set of n nonoverlapping intervals. Subproblem i is the cost of the highest-value subset of the first i intervals.
>
> There are two cases for the optimal solution of the first i intervals: either the ith interval is in the solution, or it is not.
>
> If the ith interval is in the solution, then $OPT(i)$ consists of two parts: first, the ith interval (with value v_i), and then, the best way to schedule the remaining intervals. Since i is in the solution, all other intervals in the solution cannot overlap with i; so the best way to schedule all remaining intervals costs $OPT(p(i))$; where $p(i)$ is the latest interval that does not overlap with i. Therefore, if the ith interval is in the solution, $OPT(i) = v_i + OPT(p(i))$.
>
> If the ith interval is not in the solution, then the best solution to $OPT(i)$ is the same as the best solution to $OPT(i-1)$; therefore, $OPT(i) = OPT(i-1)$.
>
> We want the highest-value subset, so we take whichever of these cases gives us a larger value:
>
> $$OPT(i) = \max\{OPT(i-1), v_i + OPT(p(i))\}.$$

When looking for cases, you are generally looking to *truncate* the optimal solution. Can you cut a piece

off the optimal solution, and split into cases about what it might look like? This often takes the form of something along the lines of:

- Is the last item in, or not in, the optimal solution?
- What is the second-to-last item in the optimal solution?
- Where is the last item in the solution from? (For example, in edit distance, we had multiple cases based on if the last column in the alignment had a character from string 1, string 2, or both strings.)

Base Case: The base cases are the values for small problems that you can solve *without using recursion*. You should state specifically how these values should be filled in.

This is in fact the best way to check that you have covered all base cases: you should look at your recursion. Any entry in your table that is not covered by a base case should have a "valid" recursion—in particular, no out of bounds table entries. You should cross-reference your subproblem definition to see what the bounds are of the table, and the evaluation order to see the order in which table entries are filled in.

> **Example.** Let's say that my recursion is
>
> $$M[i] = v_i + \max\{M[i-2], M[i-3]\}.$$
>
> I look at my subproblem definition and see that $M[i]$ is the value of the optimal solution for the first i vertices, including vertex i. This means that i ranges from 1 to n.
>
> I also see that the evaluation order is in increasing order of i, from 1 to n.
>
> Immediately, I have that I need $M[1]$ as a base case, since it's the first table entry we fill in. Then I also need $M[2]$ as a base case (otherwise it would access $M[0]$ and $M[-1]$; both out of bounds), and $M[3]$ as a base case (it accesses $M[0]$).

Memoization Structure: This one is usually straightforward; state if the table is one or two dimensional, and exactly how many rows/columns it has.

Evaluation Order: This is almost always in increasing order of i for one-dimensional tables, but for two-dimensional tables it is important to specify the order (usually row-major or column-major order).

The key here is to check the recursion. You want to ensure that when you are filling out a table entry, each entry you look up has already been filled in.[2]

Final Solution: Once you are done filling in the table, how can you obtain the solution to the problem we are trying to solve? The final solution is very closely tied to the subproblem definition; be sure to look it over carefully before filling this in.

Usually, the final solution is of one of two forms.

If the subproblem definition was extremely similar to the original problem—especially if it is along the lines of "solve exactly the same problem on the first i elements of the input"—the solution to the final subproblem is usually just a single entry in the table.

In some cases, the subproblem definition is slightly different from the original problem. In this case, you may need to search the table—perhaps you want to find the largest entry, or (perhaps) find the smallest entry in the last column.

Time and Space Analysis: The space is just the total space taken: usually this is just the number of entries in the dynamic programming table. (If you stored any other tables, those would also count towards the space.)

For the time taken, I strongly recommend starting by looking at the time spent to fill in a *single* entry in the table. What calculations do we need to do to fill in (say) $M[i, j]$? Then, sum this up over all entries in the table to obtain the total time.

It can (on relatively rare occasions) be useful to group cells when analyzing costs: in analyzing the Bellman-Ford algorithm, we showed that all cells in a given column take $O(m)$ time in total to calculate. Summing over the n columns gave us $O(nm)$ running time.

Finding the Solution Itself

The above techniques give us the *value* of the best possible solution to a problem. With only very rare exceptions, we can backtrack to get the solution itself.

In this class, we focus mostly on obtaining the value; please bear in mind that for problems in this class, **you do not need to find the solution itself** using the techniques explained below **unless you are asked to explicitly**.

The key idea of backtracking is that when we did our recurrence, we took the best of several "cases" for what the optimal solution might look like. If we look at the value of the recurrence—which case

[2] In particular, you don't want to "recurse" to fill in that table entry as well.

led to the best cost—we immediately know what the final portion of the optimal solution looked like. Doing this recursively gives us the final solution.

Example. Consider the bookshelf problem discussed in class. Our recurrence was

$$M[i,j] = \min_{1 \le i' \le i} \max \left\{ M[i', j-1], \sum_{t=i'+1}^{i} s_t \right\}$$

Recall that when we discussed this recurrence, we said that $i'+1$ is the first book assigned to the last worker.

Let's consider a relatively small instance, with 8 books and 3 workers. The number of pages in the books are (in order): $81, 99, 9, 81, 24, 45, 16, 73$. The dynamic programming table is as follows:

i (books)	M[i][1]	M[i][2]	M[i][3]
1	81	81	81
2	180	99	99
3	189	108	99
4	270	180	108
5	294	180	114
6	339	180	150
7	355	180	159
8	428	239	180

The optimal cost is $M[8][3] = 180$. Let's backtrack to get the optimal solution. We can see that the recurrence for $M[8][3]$ obtained its optimal value at $i' = 4$.[a] (Namely, $M[8][3] = \max\{M[4,2], 158\} = 180$). This means that books 5 through 8 are assigned to worker 3; the remaining 4 books are assigned to workers 1 and 2. So we recurse on $M[4,2]$.

We see that $M[4,2]$ takes its optimal value at $i' = 2$. (Namely, $M[4][2] = \max\{M[2,1], 90\}$). This means that books 3 and 4 are assigned to worker 2; the remaining 2 books are assigned to worker 1.

We are done! Worker 1 gets books 1 and 2 (180 pages); worker 2 gets books 3 and 4 (90 pages); worker 3 gets books 5 through 8 (158 pages). We can double check that the cost of this solution matches the cost from the DP table: the most-overworked worker gets assigned 180 pages, and this does indeed match.

[a] There is actually a tie! We can choose any of $i' = 4, 5, 6, 7$. Looking ahead to the final solution, we see that any of these solutions work: for example, if we move books 5, 6, and 7 to worker 2, we still obtain the same final cost.

In some cases, it may be useful to, when filling out the dynamic programming table, keep track of what the minimum was. This saves you extra work later (after all, figuring out what i' was best in the table in the example was a bit of a pain).