Dynamic Programming Techniques

Winter 2023

[[Have a running example]]

1 How to Build a Dynamic Program

1.1 Find Subproblems

The first thing that you need to do is to find an appropriate family of sub-problems. These are usually of the form: Find the value of the best partial solution to your problem satisfying some constraints. These constraints will encode important information about your subproblem that will be necessary to let you know how that partial solution can be extended to a full solution.

There is a balance that needs to be struck. If your subproblems don't encode enough information, it will be hard to find a recurrence because you will not know whether the best solution to your subproblem can be extended to a larger solution. However, if you encode too much information, you may end up with too many subproblems to solve, which will make your algorithm too slow.

1.2 Find a Recurrence Relation

Once you have your family of subproblems, you then need to find a recurrence relation (and a base case) that allows you to easily solve each problem in terms of even simpler sub-problems. Often this kind of recurrence involves looking at the partial solution that you would be left with after undoing the first or last choice in your final solution. Undoing this choice (which might have a few different possibilities) leaves you with a partial solution that hopefully corresponds to one of your other subproblems.

1.3 Notes

- In practice, you usually want to develop your set of subproblems and your recurrence relation in tandem. Building the problems to encode the information that your recurrence needs.
- Remember that you need to have a recurrence for *every* subproblem. Not just the big one at the end that gives your final answer.
- Remember that your recurrence needs to solve each problem in terms of *simpler* subproblems. This is necessary to ensure that there is some order to solve the problems in.
- Often the answers to your subproblems should only encode the *value* of your solution. If you want to know what solution gives this value, you can often back it out from your computation.

2 Writing Your Dynamic Program

The general outline for a dynamic program is now fairly simple:

```
Create a table to store answers to all of your subproblems
For each subproblem (starting with the simplest)
  Use your recursion to compute the answer and store it in the table
Return the entry in your table corresponding to the full problem
```

3 Runtime Analysis

The basic runtime analysis of a dynamic program is usually pretty simple. As the algorithm essentially consists of computing and tabulating the answers to a bunch of subproblems, we usually have:

$$\text{Runtime} = (\#\text{Subproblems})(\text{Time per subproblem}).$$

There is an interesting comparison to be made between this and divide and conquer algorithms. Both essentially work by producing formulations of the problem that can be solved using some recursive formula, and both need to be careful in order to avoid an exponential blowup in runtimes. For divide and conquer, this is accomplished by making sure that the recursive calls are *substantially smaller* than the original problem. This means that one will not need to have too many layers of recursive calls to deal with. This is why the standard divide and conquer runtime recurrences have recursive calls of size that is at least a constant fraction smaller than the original.

Dynamic programs by contrast often make recursive calls on problems that are nearly the same size as the original. They avoid exponential blowups via *subproblem reuse*. If in order to solve a problem of size n you need to solve two subproblems of size $n-1$, then the size of your tree of recursive calls will be of size roughly 2^n. However, if this giant recursion tree contains only a small number of *distinct* subproblems, we won't need to evaluate every branch separately. Once we have computed the value of a subproblem once, we can record the value and use that value on future lookups without having to recompute.

4 Proof of Correctness

The proof of correctness of a dynamic program is usually fairly straightforward once you have a recursive formulation that you know to be correct. You can essentially prove by induction that each entry of your table is filled out with the correct answer to the corresponding subproblem. This usually relies on two things:

- That when you do the necessary table lookups, those values have already been (correctly by the inductive hypothesis) computed. This requires being careful about the order in which you fill in your table (or at least knowing that such an order exists in the case of memoization).
- Knowing that your recursive formula is correct and thus, that these previously computed values are correctly combined to get the new one.