CS 3510 Algorithms Last updated: October 16, 2023

Lecture 15: Dynamic Programming

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

The core idea of dynamic programming is that you'll solve large problems by solving smaller problems. This may remind you of divide and conquer, but more often here, we will do bottom-up approaches to problems, where we start at the bottom and build to the overall solution, as opposed to D&C where we recurse down to the base case. [1]

1 Longest Contiguous Sum (LCS)

In this problem you're given n numbers $a[0], \ldots, a[n-1]$ which are (not necessarily positive) integers. Your goal is to find some sub-interval $x, x+1, \ldots, y$ which maximizes

$$\sum_{i=x}^{y} a[i]$$

As an example, take the array of integers $[5, 15, -30, 10, -5, 40, 10]$. The maximum contiguous sum is from 10 to 10.

The naive solution is to try all sub-intervals, which will be $\theta(n^2)$.

Can we do it in linear time though? Here's the plan, if we know the solution to the first $n-1$ numbers, we'd like to build off of that to find the solution to the first n numbers. First, define M to be an array such that $M[i]$ is the longest contiguous sum in $a[0, \ldots, i]$ that must contain i. Then the maximum value in M is the maximum continuous subarray; this is because we know the overall solution must end somewhere. Then, let's show by induction that we can find $M[]$ in $\theta(n)$ time.

Proof. For our base case, we have that $M[0] = a[0]$. Of course, the maximum subarray that ends with the zeroth element must be exactly the array only containing the zeroth element. We claim that

$$M[i] = a[i] + \max\{0, M[i-1]\}$$

Why is this? Well, for each $M[i]$, we are forced to take the ith element, but we can either choose to take the best sum up to the $i-1$th element, or we do not, simply taking 0 because $M[i-1]$ is negative. Hence we compute $M[]$ in a single for loop in $\theta(n)$ time. [2]

Consider the example where $a = [5, 15, -30, 10, -5, 40, 10]$. We write M as follows:

- $M[0] = 5$ as a base case.
- $M[1] = 15 + M[0] = 20$

[1] Recall in D&C that we compute the solutions to large problems by breaking it down into smaller problems, then solving the smaller problems. In DP, we solve the smallest problems first, then attempt to solve the large problems using the answers to the smaller problems.

[2] We could trivially optimize this for space to get constant time, but we're not super interested in the space.

- $M[2] = -10$, this is because we're stuck with -30, but if we take everything on the left, it will increase the sum.
- $M[3] = 10$ in this case, we don't want the sum to the left because it's negative, so instead we will just take 10.
- $M[4] = 5$, since we must take -5, but we keep the sum $M[3]$ on the left since it is positive.
- $M[5] = 45$, here we will take everything to the left, and include 40.
- $M[6] = 45 + 10$.

Now, we compute the maximum over the entire array ($\max_i \{M[i], 0\}$), which is 55. Thus, the longest contiguous sum ends in the 6th element and is 55, and we are done.

As we've seen, in dynamic programming, we're solving subproblems and then storing the solutions to these problems for later. We can often use arrays to store answers to subproblems, then use the array to compute the final answer at the end.

If we want to find out what our actual subarray is, it's not to difficult either. There are a number of algorithms you could use specific to this algorithm, but in a more general case, you'll want to simply record the choices you made at each time step. So, for this problem, you'd create a second array C which records at $C[i]$ whether the algorithm took the suffix or not. Then after we've found the max LCS, we can backtrack.

2 Longest Increasing Subsequence

Problem: Given n positive numbers $a[0], \ldots, a[n-1]$, find the subset of indices (not necessarily contiguous) $0 \leq i_1 < i_2 < \cdots < i_k \leq n-1$ such that

$$a[i_1] < a[i_2] < \cdots < a[i_k]$$

Consider the example $a = [5, 2, 8, 6, 3, 6, 9, 7]$.

If we use a greedy approach, starting at 5 and always taking the next valid integer, we obtain the indices $i = \{0, 2, 6\}$ giving the subsequence $5, 8, 9$. This isn't optimal, however: consider the subsequence starting from 2 with subsequence $2, 3, 6, 7$ at indices $i = \{1, 4, 5, 7\}$.

Obviously there are 2^n possible subsets, so if we tried to bruteforce this it would take exponential time. Then we need a little more understanding of the problem to develop even a polynomial time algorithm. However, it turns out that we can do this in $\mathcal{O}(n^2)$ time.

One key idea in solving problems with dynamic programming is how to construct the DP array. Consider an array M such that the entry $M[i][v]$ defines the element at the ith row and the vth column.

We'll say $M[i][v]$ is the length of the increasing subsequence of $a[0, \ldots, i]$ where each element has value at most v. If all the numbers are at most 100, then we have $v = 100$. We can then compute each $M[]$ in $\Theta(nV_{max})$. For the base case, we have:

$$M[0, v] = \begin{cases} 0 \text{ if } a[0] > v \\ 1 \text{ if } a[0] \leq v \end{cases}$$

Then we define the DP equation to be $M[i, v] = max\{M[i-1, v], 1 + M[i-1, a[i]]\}$. These cases align with the choices you have at each step: you can either take the ith element and take the best solution less than the ith element, or you can skip the ith element and take the best element before it.