Contents

1 Longest Common Subsequence

1.1 Problem Definition

Given a sequence $s = \langle s_1, s_2, ..., s_n \rangle$ a **subsequence** is any sequence $\langle s_{i_1}, s_{i_2}, ..., s_{i_m} \rangle$ with i_j strictly increasing.

Applications: document compare, DNA analysis.

NOTE: This is to find *one* longest subsequence, not all! So mostly we focus on finding the size.

1.2 Top-Down Dynamic Programming

Optimal substructure! Assume $x = \langle x_1, ..., x_m \rangle$ and $y = \langle y_1, ..., y_n \rangle$. Let $z = \langle z_!, ..., z_k \rangle$ be an LCS of x and y. Then

- If $x_m = y_n$, then we must have $z_k = x_m = y_n$ and $\langle z_1, ..., z_{k-1} \rangle$ is an LCS of $\langle x_1, ..., x_{m-1} \rangle$ and $\langle y_1, ..., y_{n-1} \rangle$.
- If $x_m \neq y_n$ then z uses at most one of them. Specifically:
  - If $z_k \neq x_m$ then z is an LCS of $\langle x_1, ..., x_{m-1} \rangle$ and y
  - If $z_k \neq y_n$ then z is an LCS of x and $\langle y_1, ..., y_{n-1} \rangle$

Now we can define a recursion relation that tells us the length of the LCS! Let $c(i, j)$ be the length of the LCS of $\langle x_1, ..., x_i \rangle$ and $\langle y_1, ..., y_j \rangle$. We want $c(n, m)$.

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c(i-1, j), c(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \quad (1)$$

So now we can do it recurisively! We'll analyze the running time when we talk about it bottom up.

1.3 Bottom Up Dynamic Programming

Recursion is easy to write, but hard to think about... bottom up is much more natural to think about.

So let's think about it another way... If the problem is *really small* we can do it quickly! What does really small mean? If both sequences are size 1.

But if we know the answer where both subsequences are size 1, then we can in constant time figure out most of the cases where both subsequences are size 2... Specifically we draw a diagram that looks like the one from lecture. Go look at it. It is an nXm box and we can see that at box (r, c), if we know all squares (r_s, c_s) with $r_s < r$ or $c_s < c$ then we can fill in (r, c) in constant time.

Running Time: Now this is easy! We have to fill up the matrix. Filling in each square, given all squares below is constant time. There are $O(nm)$ squares. This was *exactly* how we analyzed the recursive running time - this diagram is just our "memo" - but it's much easier to see.

2 Bottom Up Dynamic Programming

2.1 Ordering Subproblems

We want to order subproblems such that we never have to "backtrack" and solve a smaller problem in order to solve a larger problem. This sounds like.... Topological Sort!

If problem u depends on v draw a directed edge from v to u. If this graph isn't acyclic then we can't use DP. If it is, we topologically sort it. Note that we can't sort longest simple path, for example, because the subproblems are inter-dependent.

2.2 Knapsack Problem

Given a set O of objects, each object has size and value. Want to maximize the value in a knapsack of finite size. NP-Hard in general. Solvable by DP when sizes and values are upper bounded integers... sort of.

Stupid Algorithm: $O(2^{|O|})$

Greedy Algorithms: Greedy on values or greedy on value/size. Doesn't give the optimal solution. Pretty clear if just greedy on values. For greedy on value/size, consider the following instance:

Knapsack size 50, Item 1 size 10 value 60, Item 2 size 20 value 100, Item 3 size 30 value 120.

Item 1 has the highest value/size, but the correct solution is actually items 2 and 3.

Recurrence: So given the best value we can obtain for items $i_{k+1}, ..., i_n$, what is the best value for items $i_k, ..., i_n$? That doesn't quite work. Think about the dependency graph. The "best value" for $i_{k+1}, ..., i_n$ that can be used with i_k depends on the size of i_k, but i_k depends on $i_{k+1}, ..., i_n$. So *that's* not going to work.

To decouple the problems, we need another dimension! Namely, the space. So we use $V(k, X)$ the maximum value for items $i_k, ..., i_n$ given X space. We want $V(0, S)$ where S is the size of the knapsack. So in order for this to work X must have a finite number of values! Then

$$V(k,X) = \begin{cases} 0 & \text{if } i = 0 or X = 0 or X > S \\ V(k+1, X) & \text{if } \textsc{Size}(i_k) > X \\ \max(V(k+1,X), \textsc{Value}(i_k) + V(k+1, X - \textsc{Size}(i_k))) & \end{cases} \quad (2)$$

Running Time: Size of memo is nS. Filling in each square takes constant time so $O(nS)$. Sadly... this isn't polynomial in the size of the input! S takes only $O(\log S)$ bits to specify.

2.3 Bottom Up Dynamic Programming to the... Oh, FAIL

For any problem, we can *always* create a memo where we can fill in each square in polynomial time! The problem is that the size of the memo may not be polynomial.

Longest Simple Paths Again: Consider the function $d(s, t, W)$ that stores the longest path from s to t using only vertices in W. We can use DP (I think) to calculate this but there are an exponential number of them.

3 Example Problems

3.1 Making Change

Problem: You are given n types of coins with values $v_1, ..., v_n$ and a cost C. You may assume $v_1 = 1$ so that it is always possible to make any cost. What is the smallest number of coins required to sum to C exactly?

For example, assume you coins of values 1, 5, and 10. Then the smallest number of coins to make 26 is 4: 2 coins of value 10, 1 coin of value 5, and 1 coin of value 1.

Give a recurrence relation, show how you solve this recurrence *bottom-up*, and analyze the runtime. (Note the runtime should be polynomial in the value of C but may not be polynomial in $\log C$.)

Solution: *Recursion*: We recurse on $M(j)$, the minimum number of coins required to make change for cost j.

$$M(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{v_i \in n}(M(j - v_i)) + 1 & \text{else} \end{cases} \quad (3)$$

Running Time: M has C elements and computing each element takes $O(n)$ time so the total running time is $O(nC)$.

3.2 Box Stacking

Problem: You are given a set of boxes $\{b_1, ..., b_n\}$. Each box b_j has an associated width w_j, height h_j and depth d_j. You wish to create the highest possible stack of boxes with the constraint that if box b_i is stacked on box b_j, the 2D base of b_i is larger in both dimensions than the base of b_j. You can of course, rotate the boxes to decide which face is the base, but you can use each box only once.

For example, given two boxes with $h_1 = 5, w_1 = 6, d_1 = 1$ and $h_2 = 4, w_2 = 4, h_2 = 2$, you should orient box 1 so that it has a base of 5x5 and a height of 1 and stack box 2 on top of it oriented so that it has a height of 4 for a total stack height of 5.

Give a recurrence relation, show how you solve this recurrence *bottom-up*, and analyze the runtime. (The runtime should be polynomial in n.)

Solution: *Recursion*: Memoize over $H(j, R)$, the tallest stack of boxes with j on top with rotation R.

$$H(j,R) = \begin{cases} 0 & \text{if } j = 0 \\ \max_{i<j \text{ with } w_i > w_j, d_i > d_j}(H(i,R) + h_j) & \text{if } j > 0 \end{cases} \tag{4}$$

Running Time: The size of H is $O(n|R|)$ where R is the number of possible rotations for a box. For our purposes, $|R| = 3$ (since we only care about which dimension we designate as the "height") so $|H| = O(n)$. Filling in each element of H is also $O(n)$ for a total running time of $O(n^2)$.