CS 3510 Algorithms Last updated: October 17, 2023

Lecture 16: Dynamic Programming II

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

As a recap, in dynamic programming we define some table which will hold results for some sub-problem, and use a recurrence for a table entry to the other table entries. The primary difference between dynamic programming and divide and conquer is that since we pre-computed the table entries, we compute subproblems using look-ups rather than a recursive call.

1 Longest Increasing Subsequence, cont.

Recall that in the Longest Increasing Subsequence problem, we are given an array

$$a[0], a[1], \ldots, a[n]$$

. Then, we find a subset of indices of a

$$0 \leq i_1 < i_2 < \cdots < i_k \leq n-1$$

such that $a[i_1] \leq a[i_2] \leq \cdots \leq a[i_k]$. This states that the indices must be in increasing order (as it's a subsequence), and that such a construction must index a subsequence that increases or stays the same (i.e., [1,2,3,3,4,4] is a valid increasing sequence).

For our table, we'll construct a two-dimensional table where the rows represent the original elements, and the columns represent the original elements in sorted order. Then we define $M[i, v]$ which represents the length of the longest increasing subsequence of only the first i elements $[i]$, containing entries less than or equal to v.

The idea is that we arbitrarily impose a constraint such that all elements are less than some v, and we will see why this constraint allows us to compose an $\mathcal{O}(n^2)$ solution. Then, we compute the final solution by looking at the last (bottom right) entry $M[n-1, v_{max}]$. Looking at the definition of this last table entry it is an increasing subsequence with the first $n-1$th-indexed elements (all of the elements), and up to the maximum which does not exclude any elements. As a side note, we will show that every element in the matrix can be computed in constant time assuming we have already computed all of the solutions above and to the left of the element.

Now that we have a table, we'll define our base cases and the recurrence. In the first row, we only have the first element, so either we take it or we don't. 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

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

[1]

To explain this equation, we can describe the choice that we would like to make. If the value of the current element is greater than v (our self-imposed constraint), we cannot take this value, and we set $M[i, v] = M[i-1, v]$. Otherwise, we have the option of taking the element. We take the better of two options; we can either not take the element ($M[i, v] = M[i-1, v]$), or we can take the element ($1 + M[i-1, a[i]]$). $a[i]$ is written in this option because if we take $a[i]$, then every element in the subsequence we are adding to must be less than $a[i]$.

1.1 Runtime

The first point of confusion is whether or not this is pseudo-polynomial. While we are iterating over integer values, we are only iterating over integer values which are present in the array (which is done by sorting the list). Apart from that, the runtime may seem complicated, but it is actually quite simple. Since there's a constant time for each entry, and the size of the table is $\theta(n^2)$, the total running time is $\theta(n^2)$.

As we discussed last time, we can cheaply recover the choices we made at each step by storing the information.

1.2 Example

Let $a = [5, 2, 8, 6, 3, 9, 7]$. Then we write our DP table:

	2	3	5	6	6	7	8	9
5	0	0	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	2	2
6	1	1	1	2	2	2	2	2
3	1	2	2	2	2	2	2	2
6	1	2	2	3	3	3	3	3
9	1	2	2	3	3	3	3	4
7	1	2	2	3	3	4	4	4

Clearly, the longest increasing subsequence is 4, and we are done.

2 Max Independent Set on Trees

In this problem we are given a tree $G = (V, E)$, where a set $S \subseteq V$ is independent if no edge $(u, v) \in E$ for all $u, v \in S$.[2] The goal is to find the cardinality of the largest independent set.

[1] You may wonder if we were actually implementing this whether we should iterate over the rows or the columns, it turns out that this does not matter. The only thing requirement is that as you iterate, your dependencies (above and left from the element) have already been computed. We do not care about how this is done for this class.

[2] In other words, a set is independent if no two vertices share an edge.

On trees and graphs, our table may not be as clear because vertices do not always have an ordering.[3]

Instead, ignore ordering in the table and simple imagine storing a value for each vertex (so a table M is indexed by vertices $v \in V$). If we do this, when we want to find the entire solution, we can look at the value at the root of the tree as this will hold the result for the whole tree.

For each vertex $v \in V(G)$, we define $M[v]$ to be the size of the largest independent set in the subtree rooted at v. Again, we note that $M[root]$ is our final answer.

We note the base case as follows: for every leaf, we let $M[leaf] = 1$.

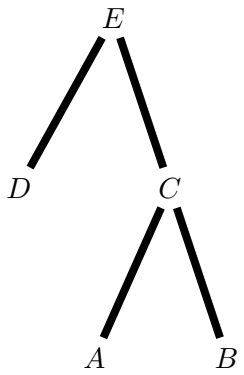
Then, for each vertex v, we have two choices. We can either take v, and add that to the independent set, or not take v. Then, our recurrence relation is as follows:

$$M[v] = \max \left\{ \sum_{u \in Child(v)} M[u], 1 + \sum_{u \in Grandchildren(v)} M[u] \right\}$$

where the first argument in max is the case if you do not add v to the independent set. In this case, the largest independent set is simply the sum of the largest independent sets of all of the children.[4] In the second case, we can take v and add it to the independent set. Then, we add 1 to the size of the independent set (v itself), and we sum the sizes of the independent sets of the grandchildren of v. This is because we assume that all of v's children are not taken, as otherwise the set would not be independent.

2.1 Example

Consider the following graph:



We first fill in $M[A] = M[B] = M[D] = 1$. For $M[C]$, we see that we can either take C and set $M[C] = 0$ (no grandchildren), or do not take C and set $M[C] = M[A] + M[B]$. Since the latter is bigger, we let $M[C] = 2$. Finally, we would like to compute $M[E]$. If

[3] Keep in mind the purpose of dynamic programming. As long as we have the dependent computations, we can assign a value/values to each node. In this case, we can imagine that we start with a leaf to be a base case, and then work our way up the tree.

[4] The union of the independent sets of the children is still independent, as it must pass through the vertex v, and a tree cannot have a cycle so there must not exist a path between any two subtrees.

we do not take E, we have that $M[E] = M[D] + M[C] = 3$. If we do, we take $M[E] = 1 + M[A] + M[B] = 3$. Since the values are equal, we set $M[E] = 3$. Thus, our final matrix M is

$$M = [1, 1, 2, 1, 3]$$

and since E is the root, we return $M[E] = 3$, and we are done.

2.2 Runtime

While each element may have been accessed a linear number of times, it may not seem like this algorithm is linear time. However, we can actually count the array accesses slightly differently to show that this algorithm, in fact, runs in linear time.

Consider an arbitrary vertex u. How many times is this accessed? We can see that u is only needed when computing the parent and the grandparent of u; u does not matter in the computation of the matrix entry of any other vertex. Since any vertex only has one parent and one grandparent, $M[u]$ is only accessed at most three times (once for computing its grandparent, once for computing its parent, and once for computing itself). Thus, the running time of this algorithm is $\mathcal{O}(|V(G)|)$, linear time in the number of vertices of G.[5]

3 Knapsack

We will introduce and briefly define the Knapsack Problem in class today, but will continue the discussion of this problem on Thursday.

We are given a budget $B \in \mathbb{N}$. Furthermore, we are given n jobs with sizes $s_1, s_2, \ldots, s_n$ with values of $v_1, v_2, \ldots, v_n$ respectively. Intuitively, we can imagine we have a list of tasks that we can choose to perform, each of which takes a certain amount of time (differs per job). If each job gives a certain reward, how do we maximize the reward if we are only given a certain amount of time?

To formalize this, the goal is to find $A \subseteq \{1, \ldots, n, \}$ such that

$$\sum_{i \in A} s_i \leq B \text{ and } \sum_{i \in A} v_i \text{ is maximized}$$

(that is, we choose a set such that we are under budget and we maximize reward). It turns out that this problem can be done in $\mathcal{O}(nB)$ with a dynamic programming approach, which we will discuss next class.

[5] As a side note, we remark that the largest independent set problem is not possible given a general graph G (this is actually NP-complete).