15-451/651: Design & Analysis of Algorithms February 12, 2019
Lecture #8: Dynamic Programming I last changed: February 7, 2019

Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. In this lecture, we discuss this technique, and present a few key examples. Topics in this lecture include:

- The basic idea of Dynamic Programming.
- Example: Longest Common Subsequence.
- Example: Knapsack.
- Example: Independent Sets on Trees.
- Example: Optimum Binary Search Trees.

1 Introduction

Dynamic Programming is a powerful technique that can be used to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. (Usually to get running time below that—if it is possible—one would need to add other ideas as well.) Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer" is a general method, except that unlike divide-and-conquer, the subproblems will typically overlap. This lecture we will present two ways of thinking about Dynamic Programming as well as a few examples.

There are several ways of thinking about the basic idea.

Basic Idea (version 1): What we want to do is take our problem and somehow break it down into a reasonable number of subproblems (where "reasonable" might be something like n^2) in such a way that we can use optimal solutions to the smaller subproblems to give us optimal solutions to the larger ones. Unlike divide-and-conquer (as in mergesort or quicksort) it is OK if our subproblems overlap, so long as there are not too many of them.

2 Example 1: Longest Common Subsequence

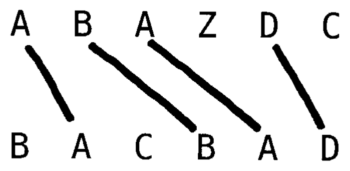
Definition 1 *The* **Longest Common Subsequence (LCS)** *problem is as follows. We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.*

For example, consider:

$S =$ `ABAZDC`

$T =$ `BACBAD`

In this case, the LCS has length 4 and is the string `ABAD`. Another way to look at it is we are finding a 1-1 matching between some of the letters in S and some of the letters in T such that none of the edges in the matching cross each other.



This type of problem comes up all the time in genomics: given two DNA fragments, the LCS gives information about what they have in common and the best way to line them up.

Let's now solve the LCS problem using Dynamic Programming. As subproblems we will look at the LCS of a prefix of S and a prefix of T, running over all pairs of prefixes. For simplicity, let's worry first about finding the *length* of the LCS and then we can modify the algorithm to produce the actual sequence itself.

So, here is the question: say LCS$[i,j]$ is the length of the LCS of $S[1\cdots i]$ with $T[1\cdots j]$. How can we solve for LCS$[i,j]$ in terms of the LCS's of the smaller problems?

Case 1: what if $S[i] \neq T[j]$? Then, the desired subsequence has to ignore one of $S[i]$ or $T[j]$. (If it were to use both of them, the result would be two matching lines in the diagram that cross.) So we have:

$$\text{LCS}[i,j] = \max(\text{LCS}[i-1,j], \text{LCS}[i,j-1]).$$

Case 2: what if $S[i] = T[j]$? Then the LCS of $S[1\cdots i]$ and $T[1\cdots j]$ might as well match them up. For instance, if I gave you a common subsequence that matched $S[i]$ to an earlier location in T, for instance, you could always match it to $T[j]$ instead to construct a solution that is the same length. (This is a *canonical form* argument.) So, in this case we have:

$$\text{LCS}[i,j] = 1 + \text{LCS}[i-1,j-1].$$

Combining these together and dealing with the base cases we get the following recurrence:

$$\text{LCS}[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{\text{LCS}[i-1,j], \text{LCS}[i,j-1]\} & \text{if } S[i] \neq T[j] \\ 1 + \text{LCS}[i-1,j-1] & \text{if } S[i] = T[j] \end{cases}$$

So, we can just do two loops (over values of i and j) , filling in the LCS using these rules. Here's what it looks like pictorially for the example above, with S along the leftmost column and T along the top row.

	B	A	C	B	A	D
A	0	1	1	1	1	1
B	1	1	1	2	2	2
A	1	2	2	2	3	3
Z	1	2	2	2	3	3
D	1	2	2	2	3	4
C	1	2	3	3	3	4

We just fill out this matrix row by row, doing constant amount of work per entry, so this takes $O(mn)$ time overall. The final answer (the length of the LCS of S and T) is in the lower-right corner.

How can we now find the sequence? To find the sequence, we just walk backwards through matrix starting the lower-right corner. If either the cell directly above or directly to the left contains a value equal to the value in the current cell, then move to that cell (if both do, then chose either one). If both such cells have values strictly less than the value in the current cell, then move diagonally up-left (this correspons to applying Case 2), and output the associated character. This will output the characters in the LCS in reverse order. For instance, running on the matrix above, this outputs `DABA`.

3 More on the basic idea, and Example 1 revisited

We have been looking at what is called "bottom-up Dynamic Programming". Here is another way of thinking about Dynamic Programming, that also leads to basically the same algorithm, but viewed from the other direction. Sometimes this is called "top-down Dynamic Programming".

Basic Idea (version 2): Suppose you have a recursive algorithm for some problem that gives you a really bad recurrence like $T(n) = 2T(n-1)+n$. However, suppose that many of the subproblems you reach as you go down the recursion tree are the *same*. Then you can hope to get a big savings if you store your computations so that you only compute each *different* subproblem once. You can store these solutions in an array or hash table. This view of Dynamic Programming is often called *memoizing*.

For example, for the LCS problem, using our analysis we had at the beginning we might have produced the following exponential-time recursive program (arrays start at 1):

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); // no harm in matching up
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  return result;
}
```

This algorithm runs in exponential time. In fact, if `S` and `T` use completely disjoint sets of characters (so that we never have `S[n]==T[m]`) then the number of times that `LCS(S,1,T,1)` is recursively called equals $\binom{n+m-2}{m-1}$.[1] In the memoized version, we store results in a matrix so that any given set of arguments to `LCS` only produces new work (new recursive calls) once. The memoized version begins by initializing `arr[i][j]` to `unknown` for all `i,j`, and then proceeds as follows:

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (arr[n][m] != unknown) return arr[n][m];  // <- added this line (*)
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  arr[n][m] = result;                          // <- and this line (**)
  return result;
}
```

All we have done is saved our work in line (**) and made sure that we only embark on new recursive calls if we haven't already computed the answer in line (*).

In this memoized version, our running time is now just $O(mn)$. One easy way to see this is as follows. First, notice that we reach line (**) at most mn times (at most once for any given value of the parameters). This means we make at most $2mn$ recursive calls total (at most two calls for each time we reach that line). Any given call of `LCS` involves only $O(1)$ work (performing some equality checks and taking a max or adding 1), so overall the total running time is $O(mn)$.

[1]This is the number of different "monotone walks" between the upper-left and lower-right corners of an n by m grid. For this course you don't need to know this bound or how to prove it—it just suggests that brute-force is not the way to go. But if you are interested, look up Catalan numbers.

Comparing bottom-up and top-down dynamic programming, both do almost the same work. The top-down (memoized) version pays a penalty in recursion overhead, but can potentially be faster than the bottom-up version in situations where some of the subproblems never get examined at all. These differences, however, are minor: you should use whichever version is easiest and most intuitive for you for the given problem at hand.

More about LCS: Discussion and Extensions. An equivalent problem to LCS is the "minimum edit distance" problem, where the legal operations are insert and delete. (E.g., the unix "diff" command, where S and T are files, and the elements of S and T are lines of text). The minimum edit distance to transform S into T is achieved by doing $|S| - \texttt{LCS}(S,T)$ deletes and $|T| - \texttt{LCS}(S,T)$ inserts.

In computational biology applications, often one has a more general notion of sequence alignment. Many of these different problems all allow for basically the same kind of Dynamic Programming solution.

4 Example #2: The Knapsack Problem

Imagine you have a homework assignment with different parts labeled A through G. Each part has a "value" (in points) and a "size" (time in hours to complete). For example, say the values and times for our assignment are:

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

Say you have a total of 15 hours: which parts should you do? If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points-per-hour (a greedy strategy).

But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?[2]

> **Exercise:** Give an example where using the greedy strategy will get you less than 1% of the optimal value (in the case there is no partial credit).

The above is an instance of the *knapsack problem*, formally defined as follows:

Definition 2 *In the* **knapsack problem** *we are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).*

We can solve the knapsack problem in exponential time by trying all possible subsets. With Dynamic Programming, we can reduce this to time $O(nS)$.

Let's begin, as all dynamic programming solutions begin, by writing a recurrence. And, as before, we begin by trying to compute the value of the best solution. Later we will figure out how to construct this solution.

Let $V(k, B)$ denote the value of the highest value solution that uses items from among the set $\{1, 2, \ldots, k\}$ and uses space at most B. Here is a recurrence for V:

[2]Answer: In this case, the optimal strategy is to do parts A, B, F, and G for a total of 34 points. Notice that this doesn't include doing part C which has the most points/hour!

$$V(k,B) = \begin{cases} 0 & \text{if } k=0 \\ V(k-1,B) & \text{if } s_k > B \\ \max\{v_k + V(k-1,B-s_k), V(k-1,B)\} & \text{otherwise} \end{cases}$$

Here is a derivation of this. If there are no items to consider the value is 0 (first base case). The rest of the recurrence is built around the idea that you can either use, or not use item k. If item k cannot fit into the alloted space B, then we disgard it and solve $V(k-1,B)$ (second base case). Finally in the third case we consider two options: using the kth item, or not using it. The above recurrence considers both of these cases and takes the maximum one.

This can be turned into a recursive function. To get our answer we ask it to evaluate $V(n,S)$. Unfortuantely this is exponential time. But, notice that there are only $O(nS)$ *different* pairs of values the arguments can possibly take on, so this is perfect for memoizing. As with the LCS problem, let us initialize a 2-d array `arr[i][j]` to "unknown" for all `i,j`.

```
V(k,B)
{
  if (k == 0) return 0;
  if (arr[k][B] != unknown) return arr[k][B];  // <- added this
  if (s_k > B) result = V(k-1,B);
  else result = max{v_k + V(k-1, B-s_k), V(k-1, B)};
  arr[k][B] = result;                          // <- and this
  return result;
}
```

Since any given pair of arguments to Value can pass through the array check only once, and in doing so produces at most two recursive calls, we have at most $2n(S+1)$ recursive calls total, and the total time is $O(nS)$.

So far we have only discussed computing the *value* of the optimal solution. How can we get the items? As usual for Dynamic Programming, we can do this by just working backwards: if `arr[k][B]` = `arr[k-1][B]` then we *didn't* use the kth item so we just recursively work backwards from `arr[k-1][B]`. Otherwise, we *did* use that item, so we just output the kth item and recursively work backwards from `arr[k-1][B-s_k]`. One can also do bottom-up Dynamic Programming.

Exercise: The *fractional* knapsack problem is the one where you can add $\delta_i \in [0,1]$ fraction of task i to the knapsack, using $\delta_i s_i$ space and getting $\delta_i v_i$ value. The greedy algorithm adds items in decreasing order of v_i/s_i. Prove that this greedy algorithm produces the optimal solution for the *fractional* problem.

5 Example #3: Max-Weight Indep. Sets on Trees (Tree DP)

Given a graph G with vertices V and edges E, an *independent set* is a subset of vertices $S \subseteq V$ such that none of the edges have both of their endpoints in S. If each vertex v has a non-negative weight w_v, the goal of the *Max-Weight Independent Set* (MWIS) problem is to find an independent set with the maximum weight. We now give a Dynamic Programming solution for the case when the graph is a tree. Let us assume that the tree is rooted at some node r, which defines a notion of parents/children (and ancestors/descendents) for the tree. We let $C(v)$ be the set of children of vertex v.

Again, let us write down a recurrence to solve this problem. For a vertex v in the tree, let $U(v)$ be the weight of the maximum weight independent set in the subtree rooted at v in which v is chosen to be in the independent set. Similarly let $N(v)$ be the maximum weight solution where vertex v is NOT used. One of the other of these two must happen, so the optimum final solution's value is $\max\{U(r), N(r)\}$. Now we can write a pair of recurrences for $U()$ and $N()$.

$$\begin{aligned} U(v) &= w_v + \sum_{u \in C(v)} N(u) \\ N(v) &= \sum_{u \in C(v)} \max\{N(u), U(u)\} \end{aligned}$$

The first recurrence holds because if we are using v then we cannot use any of its children. The second holds, because if we are not using v, then we can (for each child u) decide to use or not use it, and take the best of these two options. The general technique illustrated by these recurrences is called *Tree DP*.

Lemma 3 *When the above recurrence is memoized, the running time of the algorithm is $O(n)$ for a tree with n nodes.*

Proof: The total work done by the algorithm is proportional to the number of times the functions $U()$ or $N()$ are called. (Just as in sorting algorithms we can usually assign all the cost to the comparisons, in this case we can assign all the costs to the function calls.) So our goal is to count the number of function calls.

The first time we call $U(v)$ for a vertex v assign all of the function calls done inside it (at that level) to v. This assigns $|C(v)|$ to v. Any subsequent call to $U(v)$ does no other function calls, because of memoization. Therefore a vertex v is assigned precisely $|C(v)|$. The analysis of $N(v)$ is similar, except that it does $2 \times |C(v)|$ function calls.

Overall, $3 \times |C(v)|$ is assigned to v. When summed over the entire tree this is $3(n-1)$, because the calls to the root are not counted. So to compute the final answer we must do two additional calls $N(r)$ and $U(r)$. The final answer is $3n - 1$. ■

Exercise: We just showed how to find the weight of the MWIS. Show how to find the actual independent set as well, in $O(n)$ time.

Exercise: Suppose you are given a tree T with vertex-weights w_v and also an integer $K \geq 1$. You want to find, over all independent sets of cardinality K, the one with maximum weight. (Or say "there exists no independent set of cardinality K".) Give a dynamic programming solution to this problem.

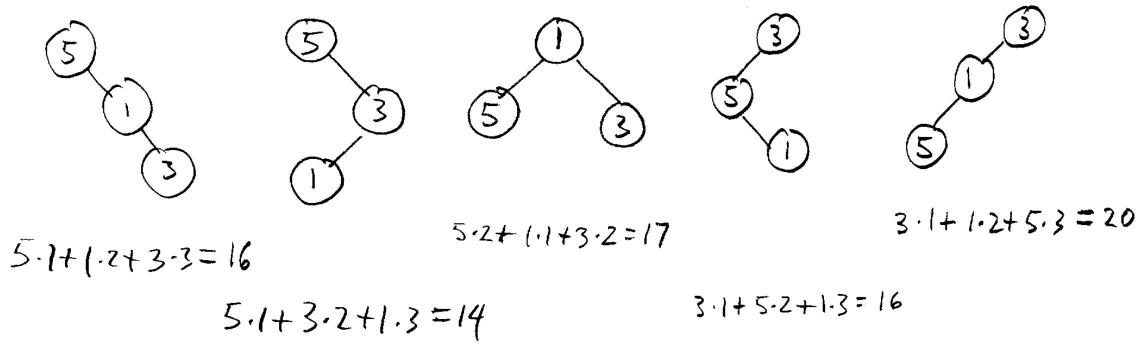
6 Example #4: Optimal Binary Search Trees

You're given frequencies $f_1, f_2, \ldots, f_n$ of the keys $\{1, 2, \ldots, n\}$ which are to be stored in a binary search tree in left to right order. The problem is to build the optimal static binary search tree for these frequencies.

Ground rules:

1. Comparisions are 3-way.
2. We'll count comparisons.
3. All searches are successful.

For example let $(f_1, f_2, f_3) = (5, 1, 3)$. Below are the five possible binary search trees of three nodes. Below each is the total search cost for that tree. The frequencies are shown inside of the nodes to help understand the costs involed.



In this case the best tree is the second one from the left.

The number of possible trees is exponential in n. Our goal is to use dynamic programming to do better than trying all trees. The trick is to consider ranges of keys $[i, j]$. Given such a range, along with the frequencies of all the keys in that range, there is an optimum binary search tree for that range of keys. This can be computed in isolation from the rest of the information. Let $C_{i,j}$ be the cost of that optimum tree in isolation using the frequencies of the keys in that range.

The optimum tree for that range must have some root k in that range. So we try all possible roots, and take the one that leads to the tree of minimum cost. Here's the resulting recurrence:

$$C_{i,j} = \begin{cases} 0 & \text{if } i > j \\ f_i & \text{if } i = j \\ \min\limits_{i \le k \le j} f_{i,j} + C_{i,k-1} + C_{k+1,j} & \text{otherwise} \end{cases}$$

I have used the notation $f_{i,j} = \sum_{k=i}^{j} f_k$. This term corresponds to all the comparisons done at the root of the tree representing the range $[i, j]$.[3] This does not depend on our choice of the root k. The terms $C_{i,k-1} + C_{k+1,j}$ take into account the rest of the search costs for the keys in this range.

What's the running time of computing this recurrence after it has been memoized? Well, the number of terms being computed is at most n^2, because there are n choices for i and n choices for j. For each of these we have to pay $j - i + 1 = O(n)$ work to find the best k. Thus the algorithm is $O(n^3)$ overall. This algorithm can be improved to $O(n^2)$.

[3]The identity $f_{i,j} = f_{1,j} - f_{1,i-1}$ allows us to compute $f_{i,j}$ in $O(1)$ time from the prefix sums. So these costs are negligible.

7 High-level discussion of Dynamic Programming

What kinds of problems can be solved using Dynamic Programming? One property these problems have is that if the optimal solution involves solving a subproblem, then it uses the *optimal* solution to that subproblem. For instance, say we want to find the shortest path from A to B in a graph, and say this shortest path goes through C. Then it must be using the shortest path from C to B. Or, in the knapsack example, if the optimal solution does not use item n, then it is the optimal solution for the problem in which item n does not exist. The other key property is that there should be only a polynomial number of different subproblems. These two properties together allow us to build the optimal solution to the final problem from optimal solutions to subproblems.

In the top-down view of dynamic programming, the first property above corresponds to being able to write down a recursive procedure for the problem we want to solve. The second property corresponds to making sure that this recursive procedure makes only a polynomial number of *different* recursive calls. In particular, one can often notice this second property by examining the arguments to the recursive procedure: e.g., if there are only two integer arguments that range between 1 and n, then there can be at most n^2 different recursive calls.

Sometimes you need to do a little work on the problem to get the optimal-subproblem-solution property. For instance, suppose we are trying to find paths between locations in a city, and some intersections have no-left-turn rules (this is particularly bad in San Francisco). Then, just because the fastest way from A to B goes through intersection C, it doesn't necessarily use the fastest way to C because you might need to be coming into C in the correct direction. In fact, the right way to model that problem as a graph is not to have one node per intersection, but rather to have one node per $\langle$*intersection, direction*$\rangle$ pair. That way you recover the property you need.

7.1 Why "Dynamic Programming"?

The term dynamic programming comes from Richard Bellman, who also gave us the Bellman-Ford algorithm from the next lecture. In his autobiography he says

> *"Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. [...] Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."*

Bellman clearly had a way with words, coining another super-well-known phrase, "*the curse of dimensionality*".