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.
- (Optional) Example: Matrix-chain multiplication.

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.

For instance, 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..i]` with T`[1..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 $T[j]$ so we have:

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

Case 2: what if $S[i] = T[j]$? Then the LCS of $S[1..i]$ and $T[1..j]$ 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. So, in this case we have:

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

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 right contains a value equal to the value in the current cell, then move to that cell (if both to, 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 correspondts 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 $O(mn)$.

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.

[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.

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 do this top down by starting with a simple recursive solution and then trying to memoize it. Let's start by just computing the best possible *total value*, and we afterwards can see how to actually extract the items needed.

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
{
  if (n == 0) return 0;
  if (s_n > S) result = Value(n-1,S); // can't use nth item
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
```

[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!

```
  return result;
}
```

Right now, this takes 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`.

```
Value(n,S)
{
  if (n == 0) return 0;
  if (arr[n][S] != unknown) return arr[n][S];  // <- added this
  if (s_n > S) result = Value(n-1,S);
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  arr[n][S] = 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[n][S]` = `arr[n-1][S]` then we *didn't* use the nth item so we just recursively work backwards from `arr[n-1][S]`. Otherwise, we *did* use that item, so we just output the nth item and recursively work backwards from `arr[n-1][S-s_n]`. 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 Independent Sets on Trees

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 v, which defines a notion of parents/children (and ancestors/descendents) for the tree.

Again, let us write down a naive resursive program to solve this problem. For a vertex v in the tree, let `MWIS(v)` be the weight of the maximum weight independent set in the subtree rooted at v. The naive recursive idea is simple: for each node v, either it is in the MWIS (in which case its children cannot be in the MWIS, and we look for the MWIS in its grandchildren. Or else v is not in the MWIS, and we look for the MWIS of its children.

```
MWIS(v) {
  if (v is a leaf) then result = w_v;
  else
    children_answer = sum_{u child of v} MWIS(u);
    grandchildren_answer = sum_{u grandchild of v} MWIS(u);
```

```
                // If no grandchildren, empty sum has value zero.
    result = max( w_v + grandchildren_answer, children_answer );
  return result;
}
```

Again, this is exponential-time in the worst-case. But fixing this using memoization is not difficult.

```
MWIS(v) {
  if (v is a leaf) then result = w_v;
  else
    if (arr[v] != unknown) return arr[v];  // <- added array check
    children_answer = sum_{u child of v} MWIS(u);
    grandchildren_answer = sum_{u grandchild of v} MWIS(u);
                // If no grandchildren, empty sum has value zero.
    result = max( w_v + grandchildren_answer, children_answer );
    arr[v] = result;                       // <- and this
  return result;
}
```

What is the runtime of this algorithm? Again, calls to the algorithm with any given vertex v can pass the array check only once. However, each recursive call no longer takes constant time. Indeed, we have recursive calls to all v's children and grand-children. There are n vertices, and there could be at most n such children/grand-children, so that is $O(n^2)$ runtime.

Actually, we can do a better calculation. For each v, `arr[v]` is unknown only once, and that particular recursive call takes time $O(1+$ number of v's children and grand-children$)$—the $O(1)$ accounts for other things done within the call. (Even if v is a leaf, the time taken is $\Theta(1)$.) When we make a call to v and the `arr[v]` is known, we take only $O(1)$ time; this can be charged to the parent recursive call. Summing over all v, the total time taken is

$$\sum_v O\bigg(1 + \text{number of children of } v + \text{number of grand-children of } v\bigg).$$

But this is $O(n)$, because each node u is counted at most twice in the sum, once for its parent and once for its grand-parent (if any).

> **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 (Optional) Example #4: Matrix product parenthesization

Our final example for Dynamic Programming is the *matrix product parenthesization* problem.

Say we want to multiply three matrices X, Y, and Z. We could do it like $(XY)Z$ or like $X(YZ)$. Which way we do the multiplication doesn't affect the final outcome but it *can* affect the running time to compute it. For example, say X is 100x20, Y is 20x100, and Z is 100x20. So, the end result will be a 100x20 matrix. If we multiply using the usual algorithm, then to multiply an ℓxm matrix by an mxn matrix takes time $O(\ell mn)$. So in this case, which is better, doing $(XY)Z$ or $X(YZ)$?

Answer: $X(YZ)$ is better because computing YZ takes 20x100x20 steps, producing a 20x20 matrix, and then multiplying this by X takes another 20x100x20 steps, for a total of 2x20x100x20. But, doing it the other way takes 100x20x100 steps to compute XY, and then multplying this with Z takes another 100x20x100 steps, so overall this way takes 5 times longer. More generally, what if we want to multiply a series of n matrices?

Definition 3 *The* **Matrix Product Parenthesization** *problem is as follows. Suppose we need to multiply a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$. Given the dimensions of these matrices, what is the best way to parenthesize them, assuming for simplicity that standard matrix multiplication is to be used (e.g., not Strassen)?*

There are an exponential number of different possible parenthesizations, in fact $\binom{2(n-1)}{n-1}/n$, so we don't want to search through all of them.[3] Dynamic Programming gives us a better way.

As before, let's first think: how might we do this recursively? One way is that for each possible split for the final multiplication, recursively solve for the optimal parenthesization of the left and right sides, and calculate the total cost (the sum of the costs returned by the two recursive calls plus the ℓmn cost of the final multiplication, where "m" depends on the location of that split). Then take the overall best top-level split.

For Dynamic Programming, the key question is now: in the above procedure, as you go through the recursion, what do the subproblems look like and how many are there? Answer: each subproblem looks like "what is the best way to multiply some sub-interval of the matrices $A_i \times \ldots \times A_j$?" So, there are only $O(n^2)$ *different* subproblems.

The second question is now: how long does it take to solve a given subproblem assuming you've already solved all the smaller subproblems (i.e., how much time is spent inside any *given* recursive call)? Answer: to figure out how to best multiply $A_i \times \ldots \times A_j$, we just consider all possible middle points k and select the one that minimizes:

$$\begin{array}{lll} & \textsf{optimal cost to multiply } A_i \ldots A_k & \leftarrow \textsf{already computed} \\ + & \textsf{optimal cost to multiply } A_{k+1} \ldots A_j & \leftarrow \textsf{already computed} \\ + & \textsf{cost to multiply the results.} & \leftarrow \textsf{get this from the dimensions} \end{array}$$

This just takes $O(1)$ work for any given k, and there are at most n different values k to consider, so overall we just spend $O(n)$ time per subproblem. So, if we use Dynamic Programming to save our results in a lookup table, then since there are only $O(n^2)$ subproblems we will spend only $O(n^3)$ time overall.

If you want to do this using bottom-up Dynamic Programming, you would first solve for all subproblems with $j - i = 1$, then solve for all with $j - i = 2$, and so on, storing your results in an n by n matrix. The main difference between this problem and the two previous ones we have seen is that any *given* subproblem takes time $O(n)$ to solve rather than $O(1)$, which is why we get $O(n^3)$ total running time. It turns out that by being very clever you can actually reduce this to $O(1)$ amortized time per subproblem, producing an $O(n^2)$-time algorithm, but we won't get into that here.[4]

[3]For the purposes of this course, you need not know how to derive this bound — it's only to suggest the futility of brute-force. But if you are interested, refer to 15-251 notes on counting, or to Catalan numbers.

[4]For details, see Knuth (insert ref).

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*".