Dynamic Programming I

Dynamic Programming is a powerful technique that often allows you to solve problems that seem like they should take exponential time in polynomial time. Sometimes it allows you to solve exponential time problems in slightly better exponential time. It is most often used in combinatorial problems, like optimization (find the minimum or maximum weight way of doing something) or counting problems (count how many ways you can do something). We will review this technique and present a few key examples.

Objectives of this lecture

In this lecture, we will:

- Review and understand the fundamental ideas of Dynamic Programming.
- Study several example problems:
    - Longest Common Subsequence
    - Knapsack
    - Independent Sets on Trees

1 Introduction

Dynamic Programming is a powerful technique that can be used to solve many combinatorial problems in polynomial time for which a naive approach would take exponential time. Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer", except that the subproblems will *overlap*.

We will assume that you have seen the idea of dynamic programming from your previous courses, but we will take a step back and review it in detail rather than diving straight into problems.

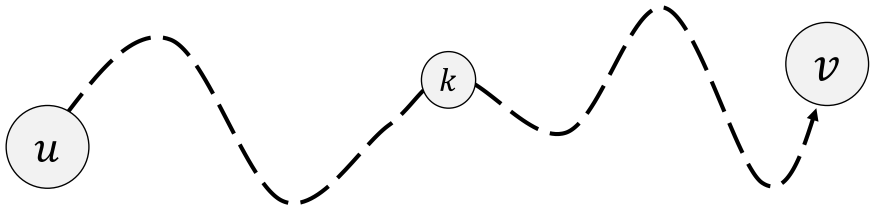
Key Idea: Dynamic programming

Dynamic programming involves formulating a problem as a set of *subproblems*, expressing the solution to the problem *recursively* in terms of those subproblems and solving the recursion *without repeating* the same subproblem twice.

The two key sub-ideas that make DP work are ***memoization*** (don't repeat yourself) and ***optimal substructure***. Memoization means that we should never try to compute the solution to the same subproblem twice. Instead, we should store the solutions to previously computed subproblems, and look them up if we need them again.

Optimal substructure is trickier, and is really where the challenge of dynamic programming comes from. A problem has optimal substructure if it can be broken into smaller problems such that the optimal solution to the big problem can be deduced from the optimal solution to the smaller problems. If the solutions to the smaller problems are completely unrelated to the solution to the bigger problem, then dynamic programming does not work. To make this a bit less vague and abstract, lets see an example.

Recall the *shortest path* problem that you may have seen before. We have a graph with two vertices u and v and we are interested in knowing the shortest path from u to v. Suppose the shortest path is P, and then consider any vertex k on the P somewhere between u and v. What can we say for certain about the path from u to k and the path from k to v? Here's a diagram to help picture this:



For certain, we can say that the path from u to k **must** be a shortest possible path from u to k, and similarly the path from k to v must be a shortest possible path from k to v. Why? We could argue by contradiction. Suppose that there was a shorter path from u to k. Then we could take that path, then follow the path from k to v and we would obtain a shorter path than P from u to v. This would contradict that u to v is a shortest path. This is exactly what we mean by optimal substructure. We know that an optimal solution to a big problem is made up from gluing together optimal solutions from the smaller subproblems!

With these key ideas in mind, lets give a high-level recipe for dynamic programming (DP). A high-level solution to a dynamic programming problem usually consists of the following steps:

1. **Identify the set of subproblems** You should **clearly and unambiguously** define the set of subproblems that will make up your DP algorithm. These subproblems must exhibit some kind of *optimal substructure* property. The smaller ones should help to solve the larger ones. This is often the **hardest part** of a DP problem, since locating the optimal substructure can be tricky.

2. **Identify the relationship between subproblems** This usually takes the form of a *recurrence relation*. Given a subproblem, you need to be able to solve it by combining the solutions to some set of smaller subproblems, or solve it directly if it is a *base case*. You should also make sure you are able to solve the original problem in terms of the subproblems (it may just be one of them)!

3. **Analyze the required runtime** The runtime is **usually** the number of subproblems multiplied by the time required to process each subproblem. In uncommon cases, it can be less

if you can prove that some subproblems can be solved faster than others, or sometimes it may be more if you can't look up subproblems in constant time.

This is just a *high-level* approach to using dynamic programming. There are more details that we need to account for if we actually want to implement the algorithm. Sometimes we are satisfied with just the high-level solution and won't go further. Sometimes we will want to go down to the details. These include:

4. **Selecting a data structure to store subproblems** The vast majority of the time, our subproblems can be identified by an integer, or a tuple of integers, in which case we can store our subproblem solutions in an array or multidimensional array. If things are more complicated, we may wish to store our subproblem solutions in a hashtable or balanced binary search tree.

5. **Choose between a *bottom-up* or *top-down* implementation** A bottom-up implementation needs to figure out an appropriate *dependency order* in which to evaluate the subproblems. That is, whenever we are solving a particular subproblem, whatever it depends on must have already been computed and stored. For a top-down algorithm, this isn't necessary, and recursion takes care of the ordering for us.

6. **Write the algorithm** For a bottom-up implementation, this usually consists of (possibly nested) for loops that evaluate the recurrence in the appropriate dependency order. For a top-down implementation, this involves writing a recursive algorithm with *memoization*.

2 Example 1: Longest Common Subsequence

Definition: Longest Common Subsequence

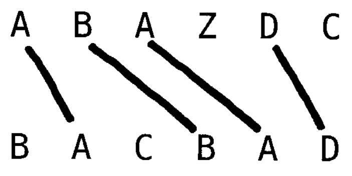
We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily contiguously) in both strings.

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.

Step 1: Identify the optimal substructure The first step is finding some optimal substructure that will inspire our subproblems. Lets take another look at the alignment in the picture above. Lets look at the last pair of aligned characters, the pair of `D` that are matched at the end. What can we say about the rest of the strings / the rest of the optimal alignment? Well, it must be the case that the optimal alignment consists of this pair of `D` plus whatever the optimal alignment is for everything that came before! In other words, the length of the optimal alignment is

$$1 + \text{LCS}(\text{“}ABAZ\text{''}, \text{“}BACDA\text{''})$$

This should hopefully provide some inspiration for our subproblems.

Step 2: Defining our subproblems Based on the above, it looks like recursively considering one less character in the string gives us some optimal substructure, so we will define our subproblems as

$$\text{LCS}[i, j] = \text{the length of the LCS of } S[1 \ldots i] \text{ and } T[1 \ldots j]$$

The solution to the original problem is the value of LCS$[n, m]$. Note that for now we are only looking for the value of the solution, i.e., *the length* of the LCS, rather than the actual string itself. We will see later that once we have the result, we can reconstruct the actual string.

Step 3: Deriving a recurrence The next step is to derive a recurrence relation between the subproblems LCS$[i, j]$ that we can use to solve the problem. When we found the optimal substructure, we already found an important piece of it: If we choose to match $S[i]$ to $T[j]$, then we know that we should optimally align the remaining characters $S[1 \ldots (i-1)]$ and $T[1 \ldots (j-1)]$, which would mean recursively solving for LCS$[i-1, j-1]$. This would give us

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

What if we don't want to match $S[i]$ and $T[j]$ though because they are just not the same character? In that case, we claim that the optimal alignment just ignores either $S[i]$ or $T[j]$ or both (it can not use both of them, because doing so would make the lines cross). This corresponds to the subproblems LCS$[i-1, j]$ and LCS$[i, j-1]$, of which we can just choose the better one.

$$\text{LCS}[i, j] = \max(\text{LCS}[i-1, j], \text{LCS}[i, j-1]) \qquad \text{if } S[i] \neq T[j]$$

Lastly, like any sensible recurrence, we need *base cases*. The base cases should be subproblems that can be solved easily without recursing. In this problem, any string has an LCS of length zero with an empty string, so that would make a good base case. Putting all of this together, we can write

Algorithm: Dynamic programming recurrence for LCS

The following recurrence gives the length of the LCS between the prefixes $S[1\dots i]$ and $T[1\dots j]$

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

Step 4: Analysis What would be the time complexity of evaluating this recurrence? Well, each evaluation of LCS(i, j) makes three recursive calls to problems of size 1 or 2 smaller, so a naive analysis would lead us to believe that the runtime is exponential. This is bad, but wait! Remember that one of the key ideas of DP was to *never repeat yourself*. If we only evaluate each subproblem once, then we get a much better algorithm. There are $O(nm)$ subproblems, and each of them does a constant amount of work to combine the results of the subproblems that it depends on recursively, so if we use dynamic programming, we get a runtime of $O(nm)$ which is pretty good!

That's it for the high-level solution. Lets take a moment to think about the lower-level details.

2.1 Possible implementation: Bottom-up

For the bottom-up strategy, we keep a table of values storing the solutions to LCS$[i, j]$ and compute them in a sensible order. What is a sensible order for this problem? Well, remember that each subproblem depends on the subproblems that are one character shorter, so we should solve them in increasing order of length. So, we can just do two loops (over values of i and j) , filling in the LCS using these rules.

```
LCS : (S: string, T : string) -> int = {
  n = size(S)
  m = size(T)
  lcs : array⟨int⟩[n+1][m+1] = {0}                      // this table is zero indexed
  for i = 1 to n do {
    for j = 1 to n do {
      if (S[i] == T[j]) lcs[i][j] = 1 + lcs[i-1][j-1]    // but the strings are one indexed!!
      else lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1])
    }
  }
  return lcs[n][m]
}
```

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 corresponts 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.

2.2 Possible implementation: Top-down

Since dynamic programming algorithms are usually defines by recurrences, it is sometimes most natural to think of implementing them recursively rather than iteratively.

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

```
LCS : (S : string, n : int, T : string, m : int) -> int = {
  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 the 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 a "memoization table" memo[i][j] to unknown for all $0 \le i \le n$ and $0 \le j \le m$ and then proceeds as follows:

```
LCS : (S : string, n : int, T : string, m : int) -> int = {
  if (n==0 || m==0) return 0;
  if (memo[n][m] != unknown) return memo[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) );
  memo[n][m] = result;                                   // <- and this line (**)
```

[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. But if you are interested, look up Catalan numbers.

```
  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 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.

2.3 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 $|S| - \mathrm{LCS}(S,T)$ deletes and $|T| - \mathrm{LCS}(S,T)$ 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.

3 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]

[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!

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

Definition: 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 $O(nS)$. Lets go through our recipe book for dynamic programming and see how we can solve this.

Step 1: Identify some optimal substructure Lets imagine we have some instance of the knapsack problem, such as our example $\{A, B, C, D, E, F, G\}$ above with total size capacity $S = 15$. Here's a seemingly useless but actually very useful observation: The optimal solution either does contain G or it does not contain G. How does this help us? Well, suppose it does contain G, then what does the rest of the optimal solution look like? It can't contain G since we've already used it, and it has capacity $S' = S - s_G$. What does the optimal solution to this remainder look like? Well, by similar logic to before, it must be the *optimal solution* to a knapsack problem of total capacity S' on the set of items not including G! (Formally we could prove this by contradiction again—if there was a more optimal knapsack solution for capacity S' without G, we could use it to improve our solution.) This is another case of *optimal substructure* appearing!

Step 2: Defining our subproblems Now that we've observed some optimal substructure, lets try to define some subproblems. Our observation seems to suggest that the subproblems should involve considering a *smaller capacity*, and considering one fewer item. How should we keep track of which items we are allowed to use? We could define a subproblem for every subset of the input items, but then we would have $\Omega(2^n)$ subproblems, and that's no better than brute force! But here's another observation: it doesn't really matter what order we consider inserting the items if for every single item we either use it or don't use it, so we can instead just consider subproblems where we are using items $1 \ldots i$ for $0 \le i \le n$. Combining these two ideas, both the capacity reduction and the subset of items, we define our subproblems as:

$$\text{V}(k, B) = \text{The value of the best subset of items from } \{1, 2, \ldots, k\} \text{ that uses at most } B \text{ space}$$

The solution to the original problem is the subproblem $V(n, S)$.

Step 3: Deriving a recurrence Now that we have our subproblems, we can use our substructure observation to make a recurrence. If we choose to include item k, then our knapsack has $B - s_k$ space remaining, and we can no longer use item k, so this gives us

$$v(k, B) = v_k + V(k-1, B - s_k) \qquad \text{if we take item } k$$

Otherwise, if we don't take item k, then we get

$$v(k, B) = V(k-1, B) \qquad \text{if we don't take item } k$$

Finally, we need some base case(s). Well, if we have no items left to use $k = 0$, that seems like a good base case because we know the answer is zero! So, putting this together, we can write the recurrence:

Algorithm: Dynamic programming recurrence for Knapsack

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

Note here that we had to check whether $s_k > B$. In this case, we can't choose item k even if we wanted to because it doesn't fit in the knapsack, so we are forced to skip it. Otherwise, if it fits, we try both options of taking item k or not taking item k, then use the best of the two choices.

Step 4: Analysis We have $O(nS)$ subproblems and each of them requires a constant amount of work to evaluate for the first time. So, using dynamic programming, we can implement this solution in $O(nS)$ time.

A top-down implementation This can be turned into a recursive algorithm. Naively this again would take exponential time. But, since 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 memoization table `memo[k][b]` to "unknown" for all $0 \le k \le n$ and $0 \le b \le S$.

```
V : (k : int, B : int) -> int = {
  if (k == 0) return 0;
  if (memo[k][B] != unknown) return memo[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)};
  memo[k][B] = result;                              // <- and this
  return result;
}
```

Since any given pair of arguments to V can pass through the memo 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 $O(nS)$.

Just like with LCS, 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 `memo[k][B]` = `memo[k-1][B]` then we *didn't* use the kth item so we just recursively work backwards from `memo[k-1][B]`. Otherwise, we *did* use that item, so we just output the kth item and recursively work backwards from `memo[k-1][B-s_k]`. One can also do bottom-up Dynamic Programming.

4 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 vertices are adjacent (i.e., 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 vertex r, which defines a notion of parents/children (and ancestors/descendents) for the tree. Lets go through our usual motions.

Step 1: Identify some optimal substructure Suppose we choose to include r (the root) in the independent set. What does this say about the rest of the solution? By definition, it means that the children of the root are not allowed to be in the set. Anything else though is fair game. In particular, for every *granchild* of the root, we would like to build a max-weight independent set rooted at that vertex. A proof by contradiction would as usual verify that this has optimal substructure.

On the other hand, if we choose to not include r in our independent set, then all of the children are valid candidates to include. Specifically, we would like to construct a max-weight independent set in all of the subtrees rooted at the children (which may or may not contain those children themselves).

Step 2: Define our subproblems The optimal substructure suggests that our subproblems should be based on *particular subtrees*. This general technique is often referred to as "tree DP" for this reason. Our set of subproblems might therefore be

$$W(v) = \text{the max weight independent set of the subtree rooted at } v$$

The solution to the original problem is $W(r)$.

Step 3: Deriving a recurrence Like many of our previous algorithms, we build the recurrence by casing on possible decisions we can make. Keeping with the spirit of that, it seems like the decision we can make at any given problem $W(v)$ is whether or not to include the root vertex v in the independent set. If we choose to not include it, then we should just recursively find a max-weight set in the children's subtrees. We let $C(v)$ be the set of children of vertex v. Then we have

$$W(v) = \sum_{u \in C(v)} W(u) \qquad \text{if we don't choose } v.$$

Suppose we do choose v, then what can we do? As discussed, we can no longer include any of v's children without violating the rules, but we can consider any of v's grandchildren and their subtrees. Let $GC(v)$ denote the set of v's grandchildren. Then we have

$$W(v) = w_v + \sum_{u \in GC(v)} W(u) \qquad \text{if we choose } v.$$

Finally, what about base cases? If v is a leaf then the max-weight independent set just contains v for sure. To write the full recurrence, we just take the best of the two choices, choose v or don't choose v.

Algorithm: Dynamic programming recurrence for max-weight independent set on a tree

$$W(v) = \max \left\{ \sum_{u \in C(v)} W(u), \quad w_v + \sum_{u \in GC(v)} W(u) \right\}$$

Wait, stop! Where's the base case? We must have forgotten it. Or did we? Suppose v is a leaf. Then both of the sums in the recurrence are empty because $C(v)$ and $GC(v)$ will both be empty. Therefore $W(v) = w_v$ for a leaf from the second case. This means that this particular recurrence doesn't need an explicit base case, because it sort of comes built in to the sum over the children. Cool!

Step 4: Analysis This is our first example of a DP where the runtime needs a more sophisticated analysis than just multiplying the number of subproblems by the work per subproblem. If we were to do that naive analysis, we would get $O(n^2)$ since there are $O(n)$ subproblems and each might have to loop over up to $O(n)$ children/grandchildren. However, we can do better. Note that each vertex is only the child or grandchild of exactly one other vertex (its parent or its grandparent respectively). Therefore, each subproblem is only ever referred to by at most two other vertices. So we can do the analysis "in reverse" or "upside down" in some sense, and argue that each subproblem is only used at most twice, and hence the total work done is just two times the number of subproblems, or $O(n)$.

Exercises: Dynamic Programming

Problem 1. One way in which top-down DP can occasionally be faster is when not all of the subproblems need to be solved in order to solve the original problem. Can you come up with an example input for the longest common subsequence problem in which the solution depends only on a linear number of subproblems?

Problem 2. We made a slightly *greedy* assumption in our algorithm for longest common subsequence. We assumed that whenever $S[i] = T[j]$, we should *always* match them. What if there was some more optimal matching where we didn't match them? Prove that this can not happen, i.e., that this greedy choice is optimal.

Problem 3. We showed how to find the weight of the max-weight independent set. Show how to find the actual independent set as well, in $O(n)$ time.

Problem 4. Give an example where using the greedy strategy for the 0-1 knapsack problem will get you less than 1% of the optimal value.

Problem 5. 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.