# CS 161 (Stanford, Winter 2025) Lecture 13

Adapted from Virginia Williams' lecture notes. Additional credits go to Wilbur Yang and Mary Wootters.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

# More Dynamic Programming

## 1 Overview

Last lecture, we talked about dynamic programming (DP), a useful paradigm and one technique that you should immediately consider when you are designing an algorithm. We covered the Bellman-Ford algorithm for solving the single source shortest path problem, and we talked about the Floyd-Warshall algorithm for solving the all pairs shortest path problem.

This lecture, we will cover some more examples of dynamic programming, and start to see a recipe for how to come up with DP solutions. We will talk about three problems today: longest common subsequence, knapsack, and maximum weight independent set in trees.

In general, here are the steps to coming up with a dynamic programming algorithm:

1. **Identify optimal substructure:** how are we going to break up an optimal solution into optimal sub-solutions of sub-problems? We're looking for a way to do this so that there are *overlapping* sub-problems, so that a dynamic programming approach will be effective.
2. **Recursively define the value of an optimal solution:** Write down a recursive formulation of the optimum, in terms of sub-solutions.
3. **Find the optimal value:** Turn this recursive formulation into a dynamic programming algorithm to compute the value of the optimal solution.
4. **Find the optimal solution:** Once we've figured out how to find the cost of the optimal solution, we can go back and figure out how to keep enough information in our algorithm so that we can find the solution itself.
5. **Tweak the implementation:**[1] Often it's the case that the solutions that we come up with in the previous steps aren't implemented in the best way. Maybe they are storing more than they need to, like we saw with our first pass at the Floyd-Warshall algorithm. In this final step (which we won't go into in too much detail in CS161), we go back through the DP solution we've designed, and optimize it for space, running time, and so on.

In this class, we'll focus mostly on 1, 2, and 3. We'll see a few examples of 4, and occasionally wave our hands about 5.

[1] We won't talk too much about this step in CS161, even though it is often important in practice.

# 2 Longest Common Subsequence

We now consider the longest common subsequence problem which has applications in spell-checking, biology (whether different DNA sequences correspond to the same protein), and more.

We say that a sequence $Z$ is a *subsequence* of a sequence $X$ if $Z$ can be obtained from $X$ by deleting symbols. For example, `abracadabra` has `baab` as a subsequence, because we can obtain `baab` by deleting `a`, `r`, `cad`, and `ra`. We say that a sequence $Z$ is a *longest common subsequence* (LCS) of $X$ and $Y$ if $Z$ is a subsequence of both $X$ and $Y$, and any sequence longer than $Z$ is not a subsequence of at least one of $X$ or $Y$. For instance, the LCS of `abracadabra` and `bxqrabry` is `brabr`.

Using the definition of LCS, we define the LCS problem as follows: Given sequences $X$ and $Y$, find the length of their LCS, $Z$ (and if we are proceeding to Step 4 of the outline above, output $Z$).

In what follows, suppose that the sequence $X$ is $X = x_1x_2x_3 \cdots x_m$, so that $X$ has length $m$, and suppose that $Y = y_1y_2 \cdots y_n$ as length $n$. We'll use the notation $X[1 : k]$ as usual to denote the *prefix* $X[1 : k] = x_1x_2 \cdots x_k$.

## 2.1 Steps 1 and 2: Identify optimal substructure, and write a recursive formulation

Our sub-problems will be to solve LCS on prefixes of $X$ and $Y$. To see how we can do this, we consider the following two cases.

- **Case 1:** $x_m = y_n$**.** If $x_m = y_n = \ell$, then any LCS $Z$ has $\ell$ as its last symbol. Indeed, suppose that $Z'$ is any common subsequence that does *not* end in $\ell$: then we can always extend it by appending $\ell$ to $Z'$ to obtain another (longer) legal common subsequence.

  Thus, if $|Z| = k$ and $x_m = y_n = \ell$, we can write

  $$Z[1 : k-1] = \text{LCS}(X[1 : m-1], Y[1 : n-1])$$

  and

  $$Z = Z[1 : k-1] \circ \ell,$$

  where $\circ$ denotes the concatenation operation on strings.

- **Case 2:** $x_m \neq y_n$**.** As above, let $Z$ be the LCS of $X$ and $Y$. In this case, the last letter of $Z$ (call it $z_k$) is either not equal to $x_m$ or it is not equal to $y_n$. (Notice that this or is not an exclusive or; maybe $z_k$ isn't equal to either $x_m$ or $y_n$). In this case, at least one of $x_m$ or $y_n$ cannot appear in the LCS of $X$ and $Y$; this means that either

  $$\text{LCS}(X, Y) = \text{LCS}(X[1 : m-1], Y)$$

  or

  $$\text{LCS}(X, Y) = \text{LCS}(X, Y[1 : n-1]),$$

whichever is longer. That is, we can shave one letter off the end of either $X$ or $Y$. In particular, the length of $\text{LCS}(X, Y)$ is given by

$$\text{lenLCS}(X, Y) = \max\{\text{lenLCS}(X[1 : m-1], Y), \text{lenLCS}(X, Y[1 : n-1])\}.$$

This immediately gives us our recursive formulation. Let's keep a table $C$, so that

$$C[i, j] = \text{length of LCS}(X[1 : i], Y[1 : j]).$$

Then we have the relationship:

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0, \\ \max\{C[i-1, j], C[i, j-1]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0. \end{cases}$$

Suppose we keep a table $C$, where $C[i, j]$ maintains the length of $LCS(X[1 : i], Y[1 : j])$, the longest common subsequence of $X[1 : i]$ and $Y[1 : j]$. Then, we can fill in the values of $C$ using the following recurrence:

$$C[i,j] = \begin{cases} 1 + C[i-1, j-1], & \text{if } X[i] = Y[j], \\ \max(C[i-1, j], C[i, j-1]) & \text{otherwise.} \end{cases}$$

Technically, we should do a proof here to show that this recurrence is correct. See CLRS for the details, but it is true that if we define $C[i, j]$ recursively as above, then indeed, $C[i, j]$ is equal to the length of $\text{LCS}(X[1 : i], Y[1 : j])$. (Good exercise: prove this for yourself using induction).

## 2.2 Step 3: Define an algorithm using our recursive relationship.

The recursive relationship above naturally gives rise to a DP algorithm for filling out the table $C$:

Note that there are only $n \times m$ entries in our table $C$. This is where the **overlapping subproblems** come in: we only need to compute each entry once, even though we may access it many times when filling out subsequent entries.

We can also see that $C[i, j]$ only depends on three possible prior values: $C[i-1, j]$, $C[i, j-1]$, and $C[i-1, j-1]$. This means that each time we compute a new value $C[i, j]$ from previous entries, it takes time $O(1)$.

Thus, we can start to see how to obtain an algorithm for filling in the table and obtaining the LCS. First, we know that any string of length 0 will have an LCS of length 0. Thus, we can start by filling out $C[0, j] = 0$ for all $j$ and similarly, $C[i, 0] = 0$ for all $i$. Then, we can fill out the rest of the table, filling the rows from bottom up ($i$ from 1 to $m$) and filling each row from left to right ($j$ from 1 to $n$). The pseudocode is given in Algorithm 1.

As mentioned above, in order to fill each entry, we only need to perform a constant number of lookups and additions. Thus, we need to do a constant amount of work for each of the $m \times n$ entries, giving a running time of $O(mn)$.

**Algorithm 1:** lenLCS$(X, Y)$

Initialize an $n+1 \times m+1$ zero-indexed array $C$.
Set $C[0, j] = C[i, 0] = 0$ for all $i, j \in \{1, \ldots, m\} \times \{1, \ldots, n\}$.
**for** $i = 1, \ldots, m$ **do**
    **for** $j = 1, \ldots, n$ **do**
        **if** $X[i] = Y[j]$ **then**
            $C[i, j] \leftarrow C[i-1, j-1] + 1$
        **else**
            $C[i, j] \leftarrow \max\{C[i-1, j], C[i, j-1]\}$
**return** $C$

## 2.3 Step 4: Recovering the actual LCS

Algorithm 1 only computes the *length* of the LCS of $X$ and $Y$. What if we want to recover the actual longest common subsequence? In Algorithm 2, we show how we can construct the actual LCS, given the dynamic programming table $C$ that we've filled out in Algorithm 1.

**Algorithm 2:** LCS$(X, Y, C)$

```
// C is filled out already in Algorithm 1
```
$L \leftarrow \emptyset$
$i \leftarrow m$
$j \leftarrow n$
**while** $i > 0$ *and* $j > 0$ **do**
    **if** $X[i] = Y[j]$ **then**
        Append $X[i]$ to the beginning of $L$
        $i \leftarrow i - 1$
        $j \leftarrow j - 1$
    **else if** $C[i, j] = C[i, j-1]$ **then**
        $j \leftarrow j - 1$
    **else**
        $i \leftarrow i - 1$

In this algorithm, we start from the end of $X$ and $Y$ and work backward, using our table $C$ as a guide. We start with $i = m$ and $j = m$. If at some point $(i, j)$, we see that $X[i] = Y[j]$, then decrement both $i$ and $j$. On the other hand, if $X[i] \neq Y[j]$, then we know that we need to drop a symbol from either $X$ or $Y$. The table $C$ will tell us which: if $C[i, j] = C[i, j-1]$, then we can drop a symbol from $Y$ and decrement $j$. If $C[i, j] = C[i-1, j]$, then we can drop a symbol from $X$ and decrement $i$. Of course, it might be the case that both of these hold; in this case it doesn't matter which we decrement, and our pseudocode will be default decrement $j$.

How long does this take? Notice that in each step, the sum $i + j$ is decremented by at least one (maybe two) and stops as soon as one of $i, j$ is equal to zero; this is at least before $i + j = 0$. Thus, the number of times we decrement $i + j$ is at most $m + n$, which was their total value to start.

Because at each step of Algorithm 2, the work is $O(1)$, the total running time is thus $O(n+m)$, which is subsumed by the runtime of $O(mn)$ necessary to fill in the table.

The conclusion is that we can find $\text{LCS}(X, Y)$ of a sequence $X$ of length $m$ and a sequence $Y$ of length $n$ in time $O(mn)$.

Interestingly, this simple dynamic programming algorithm is basically the best known algorithm for solving the LCS problem. It is conjectured that this algorithm may be essentially optimal. It turns out that giving an algorithm that (polynomially) improves the dependence on $m$ and $n$ over the $O(mn)$ strategy outlined above would imply a major breakthrough in algorithms for the boolean satisfiability problem – a problem widely believed to be computationally hard to solve.

# 3 The Knapsack Problem

This is a classic problem, defined as the following:

We have $n$ items, each with a value and a positive weight. The $i$-th item has weight $w_i$ and value $v_i$. We have a knapsack that holds a maximum weight of $W$. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let's say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

| Item | Weight | Value |
|---|---|---|
| $A$ | 6 | 25 |
| $B$ | 3 | 13 |
| $C$ | 4 | 15 |
| $D$ | 2 | 8 |

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it's having knapsacks of smaller capacities, or maybe it's having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we will see, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

## 3.1 The Unbounded Knapsack Problem

In the example above, we can pick two of item $B$ and two of item $D$. Then, the total weight is 10, and the total value 42.

We define $K(x)$ to be the optimal solution for a knapsack of capacity $x$. Suppose $K(x)$ happens to contain the $i$-th item. Then, the remaining items in the knapsack must have a total weight of at most $x - w_i$. The remaining items in the knapsack must be an optimum solution. (If not, then we could have replaced those items with a more highly valued set of items.) This gives us a nice subproblem structure, yielding the recurrence

$$K(x) = \max_{i:w_i \leq x} \left( K(x - w_i) + v_i \right).$$

Developing a dynamic programming algorithm around this recurrence is straightforward. We first initialize $K(0) = 0$, and then we compute $K(x)$ values from $x = 1, \ldots, W$. The overall runtime is $O(nW)$.

**Algorithm 3:** UnboundedKnapsack($W$, $n$, $w$, $v$)

```
K[0] ← 0
for x = 1, ..., W do
    K[x] ← 0
    for i = 1, ..., n do
        if w_i ≤ x then
            K[x] ← max{K[x], K[x − w_i] + v_i}
return K[W]
```

*Remark* 1. This solution is not actually polynomial in the input size because it takes $\log(W)$ bits to represent $W$. We call these algorithms "pseudo-polynomial." If we had a polynomial time algorithm for Knapsack, then a lot of other famous problems would have polynomial time algorithms. This problem is NP-hard.

## 3.2 The 0-1 Knapsack Problem

Now we consider what happens when we can take at most one of each item. Going back to the initial example, we would pick item $A$ and item $C$, having a total weight of 10 and a total value of 40.

The subproblems that we need must keep track of the knapsack size as well as which items are allowed to be used in the knapsack. Because we need to keep track of more information in our state, we add another parameter to the recurrence (and therefore, another dimension to the DP table). Let $K(x, j)$ be the maximum value that we can get with a knapsack of capacity $x$ considering only items at indices from $1, \ldots, j$. Consider the optimal solution for $K(x, j)$. There are two cases:

1. Item $j$ is used in $K(x, j)$. Then, the remaining items that we choose to put in the knapsack must be the optimum solution for $K(x - w_j, j - 1)$. In this case, $K(x, j) = K(x - w_j, j - 1) + v_j$.
2. Item $j$ is not used in $K(x, j)$. Then, $K(x, j)$ is the optimum solution for $K(x, j - 1)$. In this case, $K(x, j) = K(x, j - 1)$.

So, our recurrence relation is: $K(x, j) = \max\{K(x - w_j, j - 1) + v_j, K(x, j - 1)\}$. Now, we're done: we simply calculate each entry up to $K(W, n)$, which gives us our final answer. Note that this also runs in $O(nW)$ time despite the additional dimension in the DP table. This is because at each entry of the DP table, we do $O(1)$ work.

**Algorithm 4:** ZeroOneKnapsack($W, n, w, v$)

```
for x = 1, ..., W do
    K[x, 0] ← 0
for j = 1, ..., n do
    K[0, j] ← 0
for j = 1, ..., n do
    for x = 1, ..., W do
        K[x, j] ← K[x, j − 1]
        if w_j ≤ x then
            K[x, j] ← max{K[x, j], K[x − w_j, j − 1] + v_j}
return K[W, n]
```

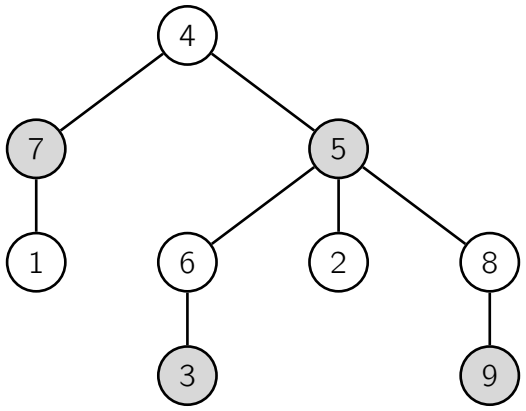
# 4 The Independent Set Problem

This problem is as follows:

Say that we have an undirected graph $G = (V, E)$. We call a subset $S \subseteq V$ of vertices "independent" if there are no edges between vertices in $S$. Let vertex $i$ have weight $w_i$, and denote $w(S)$ as the sum of weights of vertices in S. Given $G$, find an independent set of maximum weight $\text{argmax}_{S \subseteq V} w(S)$.

Actually, this problem is NP-hard for a general graph $G$. However, if our graph is a tree, then we can solve this problem in linear time. In the following figure, the maximum weight independent set is highlighted in blue.

*Remark* 2. Dynamic programming is especially useful to keep in mind when you are solving a problem that involves trees. The tree structure often lends itself to dynamic programming solutions.




As usual, the key question to ask is, "What should our subproblem(s) be?" Intuitively, if the problem has to do with trees, then subtrees often play an important role in identifying our subproblems. Let's pick any vertex $r$ and designate it as the root. Denoting the subtree rooted at $u$ as $T_u$, we define $A(u)$ to be the weight of the maximum weight independent set in $T_u$. How can we express $A(u)$ recursively? Letting $S_u$ be the maximum weight independent set of $T_u$, there are two cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children $v$ of $u$.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v A(v)$ for all grandchildren $v$ of $u$.

To avoid solving the subproblem for trees rooted at grandchildren, we introduce $B(u)$ as the weight of the maximum weight independent set in $T_u \setminus \{u\}$. That is, $B(u) = \sum_v A(v)$ for all children $v$ of $u$. Equivalently, we have the following cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children $v$ of $u$.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v B(v)$ for all children $v$ of $u$.

So, we can calculate the weight of the maximum weight independent set:

$$A(u) = \max\left\{ \sum_{v \in \text{Children}(u)} A(v), w_u + \sum_{v \in \text{Children}(u)} B(v) \right\}$$

To create an algorithm out of this recurrence, we can compute the $A(u)$ and $B(u)$ values in a bottom-up manner (a post-order traversal on the tree), arriving at the answer, $A(r)$. This takes $O(|V|)$ time.

**Algorithm 5:** MaxWeightIndependentSet($G$)

```
// G is a tree
```
$r \leftarrow \text{ArbitraryVertex}(G)$
$T \leftarrow \text{RootTreeAt}(G, r)$
**Procedure** SolveSubtreeAt($u$)
  **if** $\text{Children}(T, u) = \emptyset$ **then**
    $A(u) \leftarrow w_u$
    $B(u) \leftarrow 0$
  **else**
    **for** $v \in \text{Children}(T, u)$ **do**
      SolveSubtreeAt($v$)
    $A(u) \leftarrow \max\left\{\sum_{v \in \text{Children}(T,u)} A(v),\ w_u + \sum_{v \in \text{Children}(T,u)} B(v)\right\}$
    $B(u) \leftarrow \sum_{v \in \text{Children}(T,u)} A(v)$
SolveSubtreeAt($r$)
**return** $A(r)$