1 Dynamic Programming

Consider the Fibonacci sequence

$$F_n = F_{n-1} + F_{n-2}$$

The naive algorithm would just recursively call F_{n-1} and F_{n-2}, which is very inefficient, especially since we would be making duplicate calls. Instead, we can *tabulate* the answers, thus saving us a bunch of time.

So, what is a dynamic program? The idea is as follows:

1. Relate your answer to some family of similar subproblems. In the case of Fibonacci, we had to relate the nth Fibonacci number to all of the Fibonacci numbers before it; that is, we had to compute F_t for all $t \in [t, n] \subseteq \mathbb{Z}$.
2. There should be a recurrence relation that gives the answer to each subproblem in terms of answers to simpler subproblems.
3. Create a *table*, compute the answers to all subproblems, and then tabulate them (store the answers in the table so that you can refer back to the table when you need the answer from the previous iteration for your current iteration). This is done in the simplest to most complicated order.

Remark: You can *usually* look up entries in the table in constant time. Generally speaking, the table is done using an array or a hash map.

A few notes about dynamic programming.

- The general correct proof outline is to prove by induction that each table entry is filled out correctly. Essentially, you want to make sure the base case is correct, and then make use of the recurrence relationship for the inductive hypothesis.
- The runtime of dynamic programming is *usually* the number of subproblems *multiplied by* the time per subproblem.
- For finding the recurrence, which is very important in a dynamic programming algorithm, you often look at the first or last choice and see what things look like without that choice.

1.1 Problem: Longest Common Subsequence

Given a sequence $a_1a_2\dots a_n$, we can get a subsequence by removing some entries in the sequence. For example, if we have a sequence `ABCD`, a subsequence would be `ACD` (by removing the `B`). Given two sequences $a_1a_2\dots a_n$ and $b_1b_2\dots b_m$, then the common subsequence is just some $c_1\dots c_k$ that is a subsequence of both $a_1a_2\dots a_n$ and $b_1b_2\dots b_m$.

So, the problem statement is as follows: Given two sequences, compute the longest common subsequences. That is, the subsequence with as many letters as possible.

1.1.1 Example: Longest Common Subsequence

Suppose $X =$ `ABCBA` and $Y =$ `ABACA`. Then, the longest common subsequence of X and Y is `ABCA`.

1.1.2 Case Analysis

How do we compute `LCSS`$(A_1A_2\dots A_n,\ B_1B_2\dots B_m)$? We need to consider several cases for the common subsequence.

1. It does not use A_n (it does not use the last letter in the sequence A). That is, if the common subsequence does not use A_n, it is actually a common subsequence of

   $$A_1A_2A_{n-1} \text{ and } B_1B_2\dots B_m$$

   Therefore, in this case, the longest common subsequence would be

   $$\texttt{LCSS}(A_1A_2\dots A_{n-1}, B_1B_2\dots B_m)$$

2. It does not use B_m (It does not use the last letter in the sequence B). If the common subsequence does not use B_m, it is actually a common subsequence of

   $$A_1A_2A_n \text{ and } B_1B_2\dots B_{m-1}$$

   Therefore, in this case, the longest common subsequence would be

   $$\texttt{LCSS}(A_1A_2\dots A_n, B_1B_2\dots B_{m-1})$$

3. It uses both A_n and B_m, and these characters are the same. If a common subsequence uses both A_n and B_m, then:

   - These characters must be the same.
   - Such a subsequence can be obtained by taking a common subsequence of $A_1A_2\dots A_{n-1}$ and $B_1B_2\dots B_{m-1}$ and adding a copy of $A_n = B_m$ to the end.
   - Therefore, the longest length of such a subsequence is given by

   $$\texttt{LCSS}(A_1A_2\dots A_n, B_1B_2\dots B_{m-1}) + 1$$

Remark: We can be in both case 1 and case 2.

1.1.3 Recursion

We've broken the longest common subsequence into three different cases, and for each of these cases we managed to compute the longest common subsequence of subsequences in that case by a longest common subsequence problem that is a little bit simpler than the one that we started at. Of course, the *best* longest common subsequence overall is just the best that we can get from any of these three cases.

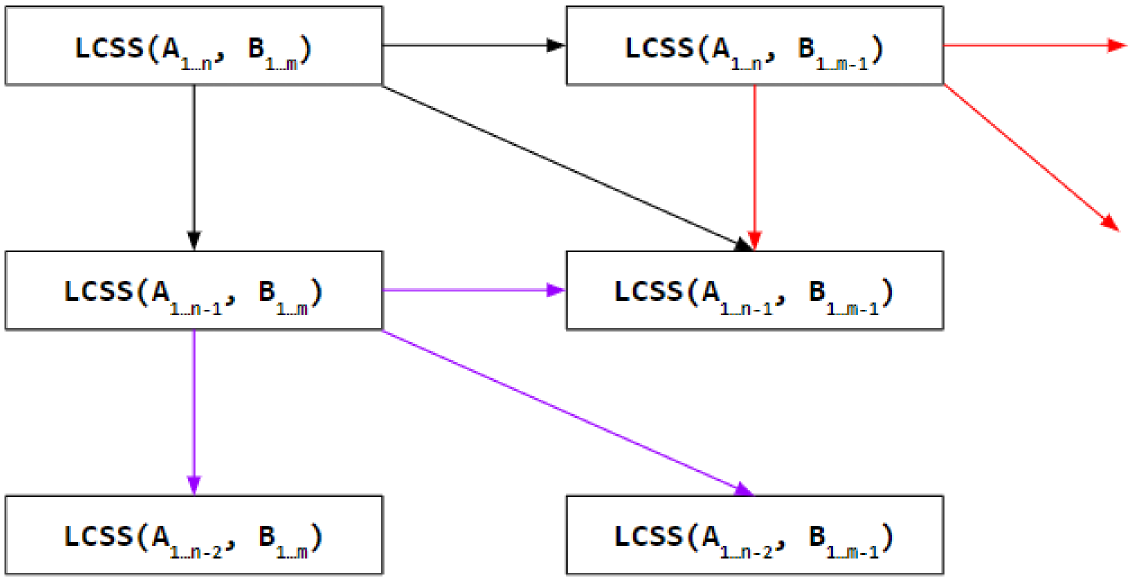
So, all we need to do is look at the three cases and take the *maximum* of the subsequences. That is

$$\texttt{LCSS}(A_1A_2\dots A_n, B_1B_2\dots B_m) = \max\begin{Bmatrix}\texttt{LCSS}(A_1A_2\dots A_{n-1}, B_1B_2\dots B_m)\\ \texttt{LCSS}(A_1A_2\dots A_n, B_1B_2\dots B_{m-1})\\ \texttt{LCSS}(A_1A_2\dots A_{n-1}, B_1B_2\dots B_{m-1})\end{Bmatrix}$$

where the last option is allowed only if $A_n = B_m$.

1.1.4 Analyzing Recursive Calls

Suppose we make a call to $\texttt{LCSS}(A_1\dots A_n,\ B_1\dots B_m)$. What recursive calls are made?



Here, many recursive calls are omitted. However, the key thing to note is that we're making a call to (LCSS)$(A_1A_2\dots A_{n-1},\ B_1B_2\dots B_{m-1})$ three times (there are three different paths to get to that function call).

Important Note 1.1

This is the key difference between a dynamic programming algorithm and a divide and conquer algorithm.

- The recursive subcalls for a divide and conquer algorithm are significantly smaller than the original problem. So, you never have to compute that many recursive subcalls even if you do the entire recursion tree.
- The recursive subcalls for a dynamic program algorithm are almost as big as the original call. *However*, the same subproblems will show up multiple times in the recursion tree. So, rather than recomputing each subproblem, we can compute each subproblem once and then *store* the result so we can refer to it later if we need it.

In the case of our problem above, we only ever see $\texttt{LCSS}(A_1A_2\dots A_k, B_1B_2\dots B_\ell)$ for some k and ℓ.

1.1.5 Base Case

Our recursion also needs a base case. So, our base case is:

$$\texttt{LCSS}(\emptyset, B_1B_2\dots B_m) = \texttt{LCSS}(A_1A_2\dots A_n, \emptyset) = 0$$

1.1.6 Algorithm

To take advantage of the fact that our dynamic program makes multiple *repeated* subcalls (i.e. subcalls with the same inputs), we use *tabulation* to *store* the results of one subcall and then, when needed, we can retrieve the results of the subcall.

Thus, our algorithm is[1]:

[1]Because the `verbatim` environment is limiting, I'll represent $A_1A_2\dots A_n$ as $A[1..n]$ and $B_1B_2\dots B_m$ as $B[1..m]$.

```
LCSS(A[1..n], B[1..m]):
    Initialize Array T[0..n, 0..m]
    // T[i, j] will store LCSS(A[1..i, 1..j])
    For i = 0 to n:
        For j = 0 to m:
            // Base Case
            If i == 0 OR j == 0:
                T[i, j] = 0

            // Case (3)
            Else if A[i] == B[j]
                T[i, j] = max(T[i - 1, j], T[i, j - 1], T[i - 1, j - 1] + 1)

            // Case (1) or (2)
            Else:
                T[i, j] = max(T[i - 1, j], T[i, j - 1])

    Return T[n, m]
```

The runtime is as follows:

- The outer loop runs in n time.
- The inner loop runs in m time.
- Everything inside the inner loop runs in constant time.

This gives us the runtime $\mathcal{O}(nm)$.

1.1.7 Example: Longest Common Subsequence Redux

Suppose $X =$ `ABCBA` and $Y =$ `ABACA`. Find the longest common subsequence.

We will tabulate the process. Call the table T.

	∅	A	AB	ABA	ABAC	ABACA
∅						
A						
AB						
ABC						
ABCB						
ABCBA						

We will go through each table one row at a time.

1. If we start at `i = 0`, then regardless of the value of `j`, the entire first row will be `0` by the first `if`-condition. So:

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A						
AB						
ABC						
ABCB						
ABCBA						

2. Now, when `i = 1`, then we need to consider the value of `j`.

- If `j = 0`, then the entry at `T[1, 0] = 0`.
- If `j = 1`, then we're comparing `A` with `A`, so we need to consider the second `if`-condition. In this case, we need to check the entry directly above (`T[i - 1, j]`), to the left (`T[i, j - 1]`), and directly adjacent in the top-left entry (`T[i - 1, j - 1] + 1`). Here, we see that the values are `0`, `0`, and `0 + 1 = 1`, respectively. So, we store the *maximum* of these values in `T[1, 1]`. Thus, `T[1, 1] = 1`.
- If `j = 2`, then we're comparing `A` with `AB`, so we need to consider the third `if`-condition. In this case, we check the entry directly above `T[i - 1, j]` and directly to the left (`T[i, j - 1]`), and take the maximum. In our case, we have `0` and `1` (which we computed from the previous bullet point), respectively. Thus, `T[1, 2] = 1`.
- If `j = 3`, then we're comparing `A` with `AB`, so we need to consider the second `if`-condition. In this case, we check the entry directly above, left, and adjacent (add one to the end result) of `[i, j] = [1, 3]`. We get the values `0`, `1`, and `0 + 1 = 1`, respectively. Thus, `T[1, 3] = 1`.
- Continuing on, we get `1`'s for the remaining cells in this row.

Therefore, the second row is:

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB						
ABC						
ABCB						
ABCBA						

Omitting the rest of the work, we see that the result is:

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB	0	1	2	2	2	2
ABC	0	1	2	2	3	3
ABCB	0	1	2	2	3	3
ABCBA	0	1	2	3	3	4

And so the answer is given by the bottom-right entry, or $\boxed{4}$.

1.1.8 Finding the Longest Common Subsequence

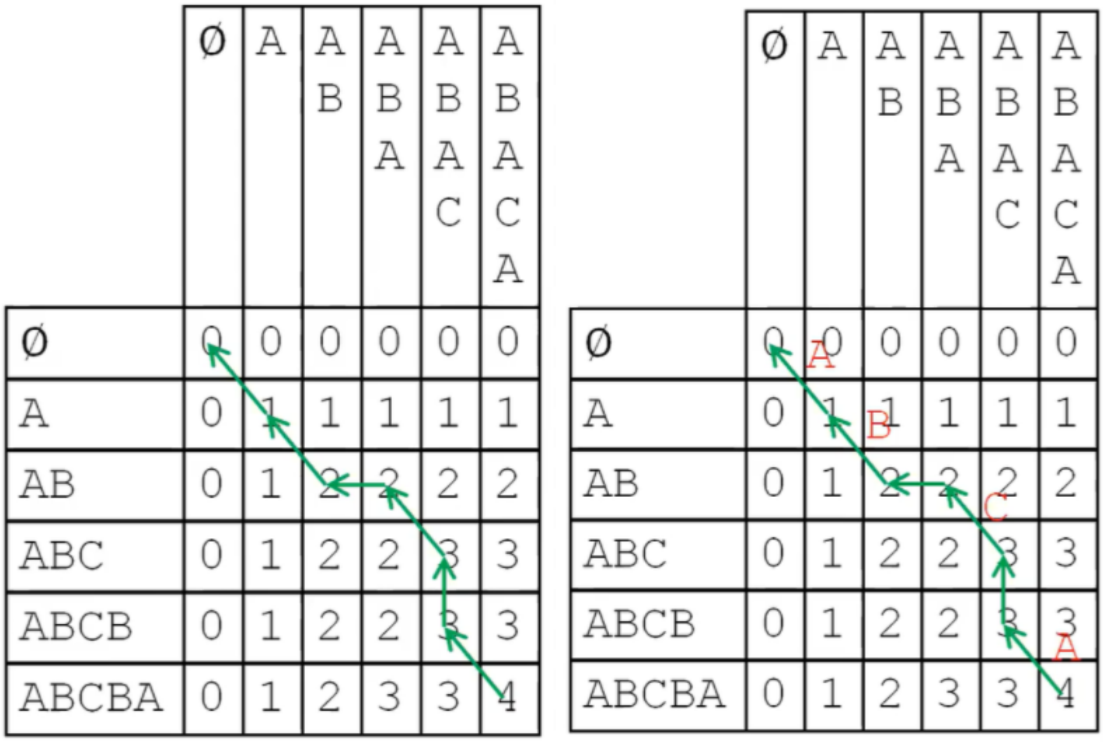
The above algorithm finds the *length* of the longest common subsequence. What if we wanted the actual subsequence? Well, we can *backtrack*. So, consider the table above.

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB	0	1	2	2	2	2
ABC	0	1	2	2	3	3
ABCB	0	1	2	2	3	3
ABCBA	0	1	2	3	3	4

We can use this trick, based on the recurrence, to find out what the subsequence is. If you're at `[i, j]`, consider the neighbors `[i - 1, j]`, `[i, j - 1]`, and `[i - 1, j - 1]`.

- If the values at all three of those entries are equal, then take the diagonal path; that is, take `[i - 1, j - 1]`.
- If `[i - 1, j - 1]` and `[i - 1, j]` are equal but `[i, j - 1]` isn't equal to any of those two, then take the `[i, j - 1]` path.
- If `[i - 1, j - 1]` and `[i, j - 1]` are equal but `[i - 1, j]` isn't equal to any of those two, then take the `[i - 1, j]` path.

Once you reach the beginning, then you backtrack again. For each *diagonal* path that was taken, take the last letter of one of the subsequences (the subsequences corresponding to the cell should end with the same letter). Once you reach back to the bottom-right square, then you can just concatenate the letters that you found.



	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB	0	1	2	2	2	2
ABC	0	1	2	2	3	3
ABCB	0	1	2	2	3	3
ABCBA	0	1	2	3	3	4

1.1.9 Proof of Correctness

We now need to prove, by induction, that this algorithm is correct.

Proof. We will use induction on i, j to show that the value assigned to $T[i, j]$ is the correct value for $\texttt{LCSS}(A_1 \ldots A_i, B_1 \ldots B_j)$.

- Base Case: When $i = 0$ or $j = 0$, then we assign `0` since the empty string has no common subsequence with a string of some length.
- Inductive Step: Suppose the previous values are assigned correctly. We note that, because of the recursive relationship and the inductive hypothesis (since we have previously filled out $T[i-1, j]$, $T[i, j-1]$, and $T[i-1, j-1]$ correctly), that $T[i, j]$ gets the correct values.

This completes the proof. □