Lecture 6: Dynamic Programming

subject	TDT4121
topics	Asymptotic Running time, Bellman Ford's Algorithm, Dynamic Programming design method, Dynamic Programming tables, Sequence Alignment, Shortest Path in a Graph, The Knapsack Problem, The Longest Common Subsequence problem
created	@November 14, 2022 11:32 AM

Chapter 6 - Dynamic Programming

> The basic idea of Dynamic Programming is drawn from the intuition behind divide and conquer and is essentially the opposite of the greedy strategy: one implicitly explores the space of all possible solutions, by carefully decomposing things into a series of subproblems, and then building up correct solutions to larger and larger subproblems.

Goals 🏁

The student should be able to:

- Understand the Dynamic Programming design method
  - Understand when and how dynamic programming could be used
- Understand how dynamic programming tables are used
- Understand *The Knapsack Problem*

- Understand how dynamic programming can be used to solve the *Longest Common Subsequence* problem
- Understand how dynamic programming can be used to find *Sequence Alignment*
- Understand how dynamic programming can be used to find the *Shortest path in a graph*
- Find the asymptotic running time for Dynamic Programming Algorithms.

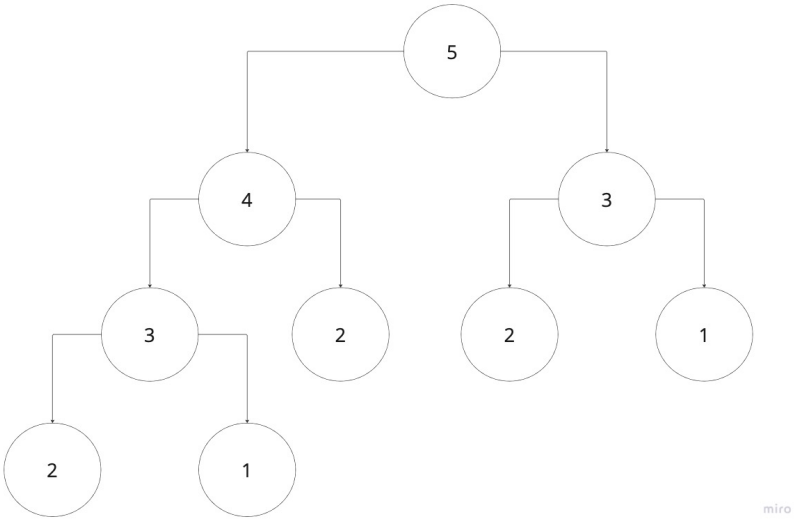
Dynamic Programming Design Method

Dynamic programming is **a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again**. The subproblems are optimised to optimise the overall solution is known as optimal substructure property.

- Dynamic programming helps us optimise solutions to problems by caching values for future use
- Dynamic programming helps us improve the runtimes of recursive algorithms

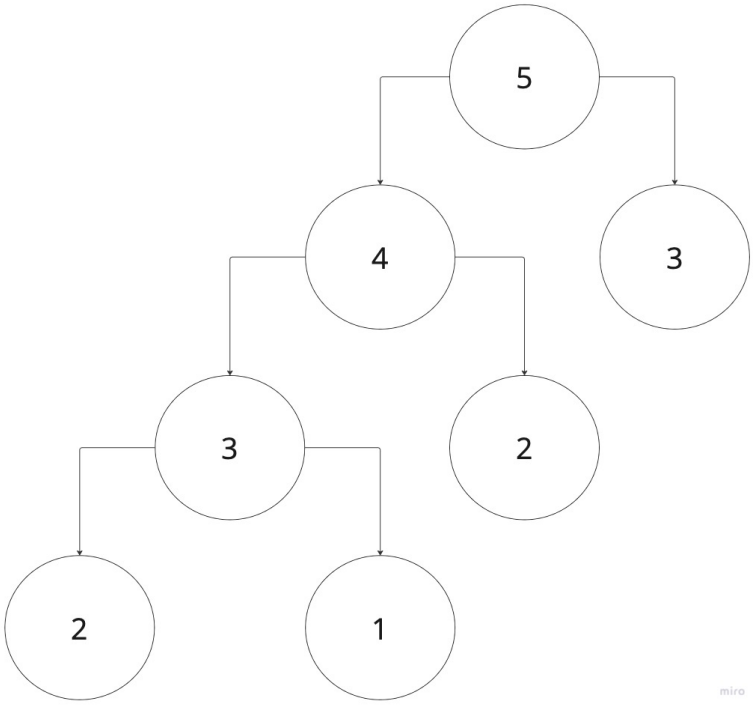
Example: tree of recursive calls for fib(5)

When we analyse the runtime of the recursion for calculating the 5th Fibonacci number by drawing up a tree, we notice the pattern that it has to analyse each number down to the the 1st or 2nd Fibonacci number for each recursive call. This means it has several duplicate subtrees. We also notice that the runtime of the algorithm will be $O(2^n)$.



Recursive calls for fib(5)

Dynamic programming practices will solve the runtime issue of the original algorithm to make sure we don’t have to calculate duplicate subtrees over and over. By, for example, implementing memoization will make the tree of recursive calls look like the figure below instead.



Recursive calls for fib(5) with memoization

Memoization: top-down approach

The top-down approach is generally recursive (but less efficient) and more intuitive to implement as it is often a matter of recognising the pattern in an algorithm and refactoring it as a dynamic programming solution.

It's called memoization because we create a *memo* for the values returned from solving each problem. When we encounter the same problem again, we simply check the memo, and, rather than solving the problem a second (or third or fourth) time, we retrieve the solution from our memo.

The time complexity of this solution is $O(n)$ because we don't solve the overlapping subproblems. The space complexity is also $O(n)$.

Tabulation: bottom-up approach

The bottom-up approach is generally iterative (and more efficient), but less intuitive and requires us to solve (and know!) the smaller problems first then use the combined values of the smaller problems for the larger solution. One can think of this solution as entering values in a table or spreadsheet, and then applying a formula to those values.

The time complexity of this algorithm is also $O(n)$ because a for loop is used to iterate over the value of n. The space complexity is however $O(1)$ because we have **no recursion**, we are only pushing one call to the stack.

When and how to use dynamic programming

In order for a problem to be solved by dynamic programming it needs have two properties: optimal substructure and overlapping sub-problems. If the problem does not have those two properties, it can't be solved with dynamic programming. Dynamic programming is similar to the divide-and-conquer approach as it also combines solutions to sub-problems. In dynamic programming the solution to a sub-problem is needed repeatedly, so the solutions are stored in a table so that they don't need to be recomputed. For this to be useful, the sub-problems need to overlap.

Dynamic Programming Tables

This is one of the most helpful visualisation techniques for designing **bottom-up** dynamic programming algorithms when the problem is a multi-prefix/multi-suffix or subsequence problem type.

The Knapsack problem

The **knapsack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Solving this with dynamic programming

Like other typical dynamic programming problems, re-computation of same subproblems can be avoided by constructing a temporary array `K[][]` in bottom-up manner.

Approach: In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a `DP[][]` table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.

The state `DP[i][j]` will denote maximum value of `j-weight` considering all values from '1 to ith'. So if we consider `wi` (weight in 'ith' row) we can fill it in all columns which have `weight values > wi`. Now two possibilities can take place:

- Fill `wi` in the given column.
- Do not fill `wi` in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' weight in 'jth' column then `DP[i][j]` state will be same as `DP[i-1][j]` but if we fill the weight, `DP[i][j]` will be equal to the value of 'wi'+ value of the column weighing `j-wi` in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualisation will make the concept clear:

Solving this with greedy algorithm

Since this resembles other scheduling problems we've seen before, it's natural to ask whether a greedy algorithm can find the optimal solution. It appears that the answer is no—at least, no efficient greedy rule is known that always constructs an optimal solution. One natural greedy approach to try would be to sort the items by decreasing weight—or at least to do this for all items of weight at most *W*—and then start selecting items in this order as long as the total weight remains below *W*. But if *W* is a multiple of 2, and we have three items with weights $\{W/2 +$

$1, W/2, W/2\}$, then we see that this greedy algorithm will not produce the optimal solution. Alternately, we could sort by *increasing* weight and then do the same thing; but this fails on inputs like
$\{1, W/2, W/2\}$.

The Longest Common Subsequence Problem

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily adjacent. For example, “*abc*”, “*abg*”, “*bdf*”, “*aeg*”, ”*acefg*”, .. etc are subsequences of “*abcdefg*”.

Solving this with dynamic programming

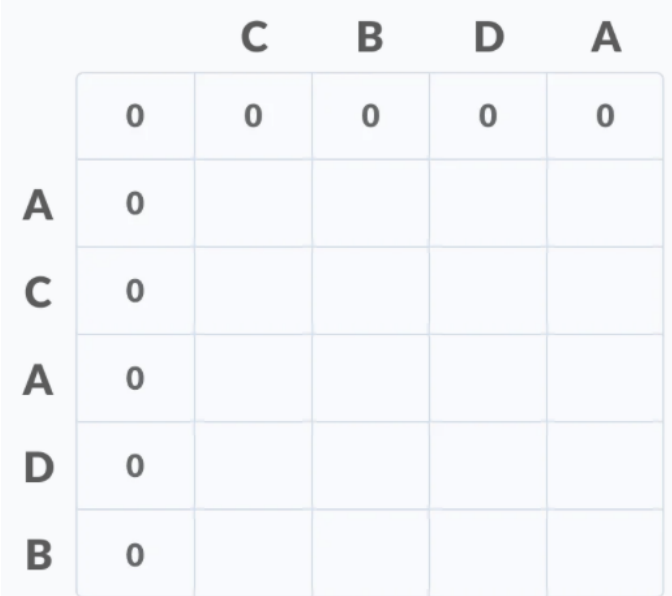
The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

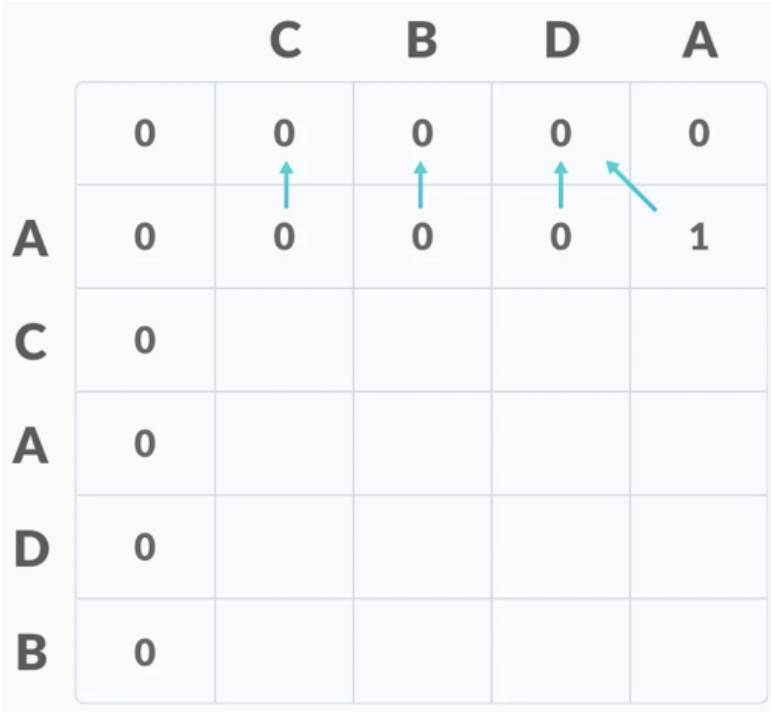
So, the time taken by a dynamic approach is the time taken to fill the table (ie. $O(mn)$). Whereas, the recursion algorithm has the complexity of $2^{max(m,n)}$.

Step-by-step:

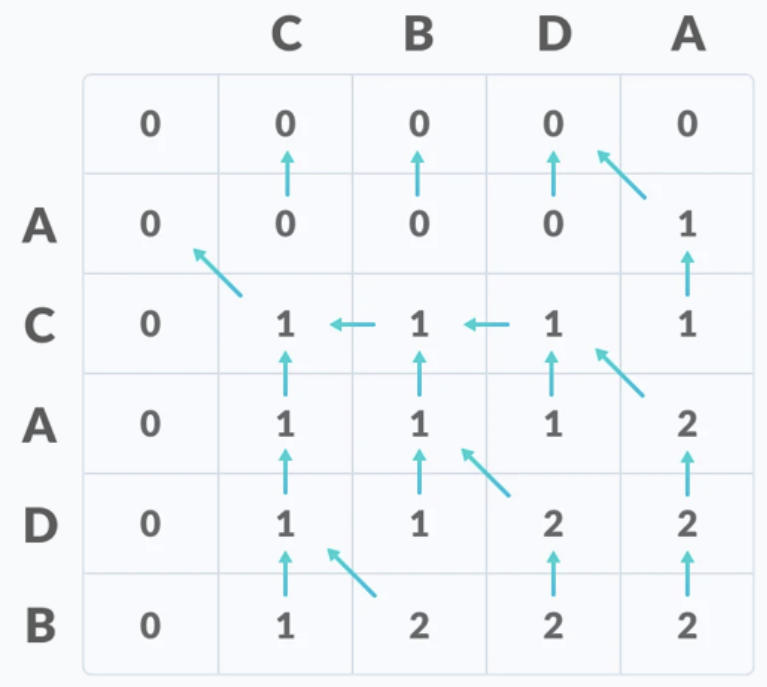
1. Create a table of dimension `n+1*m+1` where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.



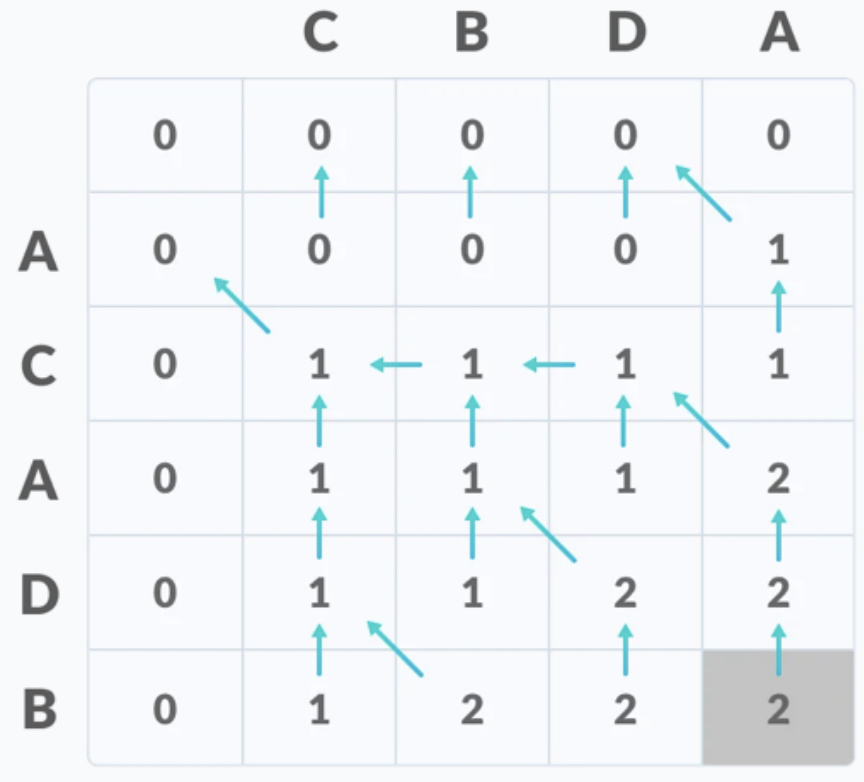
2. Fill each cell of the table using the following logic
    a. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
    b. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.



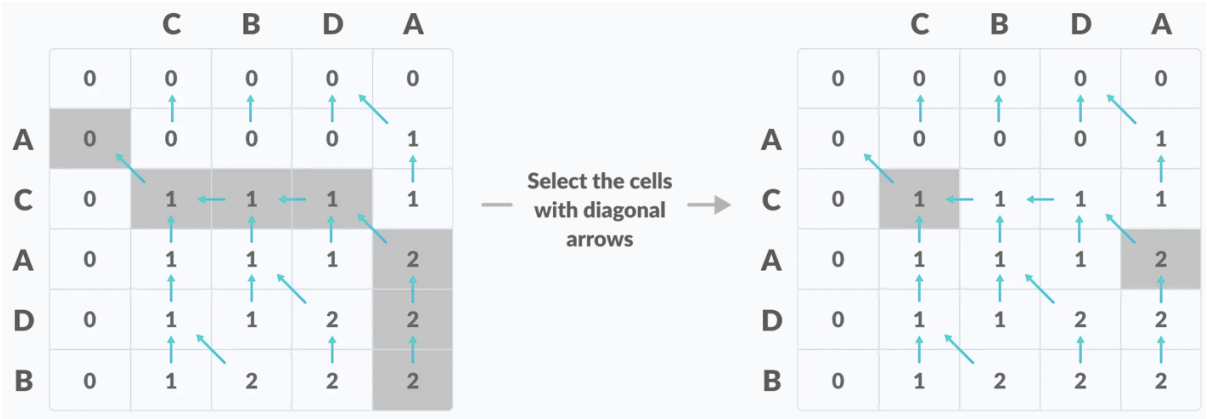
3. Repeat step 2 until the table is filled



4. The value in the last row and column is the length of the longest common subsequence



5. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Thus, the longest common subsequence is CA.

We can see here that we have a bottom-up type approach to the algorithm

```
# Pseudo-code
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
    If X[i] = Y[j]
        LCS[i][j] = 1 + LCS[i-1, j-1]
        Point an arrow to LCS[i][j]
    Else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Sequence Alignment

The sequence alignment problem is motivated by the notion of "lining up" two strings X and Y. An alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another. The parameter δ corresponds to the gap penalty. This gives a penalty at the cost of δ if a gap is inserted between the string. For each pair of letters p, q in our alphabet, we have a mismatch cost of α_{pq} for lining up p with q. There is no mismatch cost to line up a letter with another copy of itself.

This means that the cost for each alignment matching M is the sum of its gap and mismatch points, and we seek an alignment of minimum cost.

Example of alignments of “PALETTE” and “PALATE”

We assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# alignment 1
PALETTE
PALATE-
```

From alignment 1 we see that we have one gap and two mismatches which adds up to a cost of $M = 2\alpha_{pq} + \delta = 2 + 2 = 4$.

```
# alignment 2
PALETTE
PALAT-E
```

From alignment 2 we see that we have one mismatch and one gap which adds up to the cost of $M = \alpha_{pq} + \delta = 1 + 2 = 3$. This alignment has the lowest cost.

```
# alignment 3
P-ALETTE
-PALAT-E
```

From alignment 3 we see that we have one mismatch and and three gaps which adds up to the cost of $M = \alpha_{pq} + 3\delta = 1 + 6 = 7$

Finding the Shortest Path in a Graph

The shortest path problem has an optimal sub-structure. Suppose $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path from s to v. This implies that $s \rightsquigarrow u$ is a shortest path from s to u, and this can be proven by contradiction. If there is a shorter path between s and u, we can replace $s \rightsquigarrow u$ with the shorter path in $s \rightsquigarrow u \rightsquigarrow v$, and this would yield a better path between s and v. But we assumed that $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path between s and v, so have a contradiction.
Based on this optimal sub-structure, we can write down the recursive formulation of the single source shortest path problem as the following:

$$\delta(s, v) = min\{\delta(s, u) + w(u, v) | (u, v) \in E\}$$

Bellman Ford's Algorithm

The Bellman Ford algorithm computed the shortest paths in a bottom-up manner like other dynamic programming problems also does. First it calculates the shortest distances that have at-most one edge in the path. Then it calculates the shortest paths with at-most 2 edges and so on. After the i-th iteration of outer loop. the shortest path with at most i edges are calculated. Assuming that there is no negative weight cycle if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give the shortest path with at-most $(i + 1)$ edges.

Both Bellman Ford's algorithm and Djikstra's algorithm are single-source shortest path problems, meaning they both compute the shortest distance between each node in a graph from a single source node. However, they differ from each other in their approach for solving the problem. Bellman Ford's algorithm follow the dynamic programming approach, while Djikstra's algorithm follow the greedy algorithm approach.

Asymptotic Running Time