Dynamic Programming

Makes algorithm more efficient by storing some of the intermediate results in an array and passing that array back to the function

Works well when you have repetitive steps

With Fibonacci sequence, you want to be able to write a function ‘fib(n)’ that returns the nth number in the Fibonacci sequation

1st step: Recursion

2nd step: Store (memoize)

3rd step: Bottom-up (if you don’t like recursion anymore)

Make an array of n+1 size. Set all values to null to start with

Make a function fib(n, memo) where memo is the array

Store the result in memo[n], not return it.

# From lecture:

We use dynamic programming if we can make the big problem into subproblems and it is more efficient. (subproblem optimality)

If you are solving a recursive problem and you realise you are working things out again and again, think, can you memoise it?

Difference between dynamic and greedy: greedy solve a problem by running forward and greedily choosing the best local item as the next one. Only 1 path is considered.

With dynamic, you solve a problem by considering multiple paths at each point and define the optimal solution.

Assembly line scheduling:

In a factory, theres some number of lines (2 in this example, top and bottom), there are stations along the way and each station has a worker. Everyone is variably as efficient. Theres a transition time to get from one line to another, but not if it stays on the same line. The question is trying to get the item from one end to the other in the shortest time.

We could use a brute force approach. If there are 2 lines, we can think of the chosen line stations as “on” and the not chosen stations as “off”. Therefore, brute force is *O(*2n)

Longest common subsequence of two strings. (matching gene sequences). Not necessarily unique

We want to see the longest overlap of 2 strings.

The longest string , S3, such that the character in S3 occur in both S1 and S2 in the same order, but not necessarily consecutively.