Lesson:

Dynamic Programming - 1

PW SKILLS

Prerequisites:

- Arrays
- Recursion

Today's Checklist

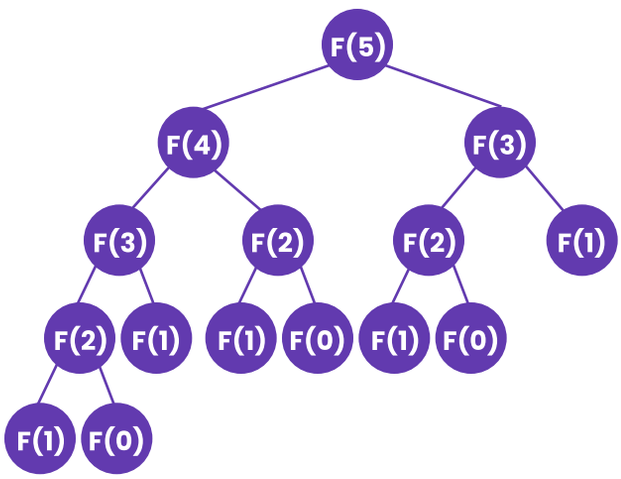
- What is Dynamic Programming?
- What are overlapping subproblems?
- What is an optimal substructure?
- Greedy vs DP
- Where to use dynamic programming?
- What are the various approaches of DP?
- What is memoization?
- Steps of solving a DP problem
- Problems of Memoization
- What is the Limitation of Top-DownProgramming?

TOPIC: What is Dynamic Programming

- **Dynamic Programming (DP)** is defined as a technique that solves some particular type of problems in Polynomial Time.
- It is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.
- It is mainly an optimization over plain recursion.
- Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later.
- This simple optimization reduces time complexities from exponential to polynomial.

TOPIC: What are overlapping subproblems?

- A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.
- **For example:** consider the recursive tree to calculate the fibonacci sequence upto n = 5.



The problem of computing the nth Fibonacci number F(n), can be broken down into the subproblems of computing F(n - 1) and F(n - 2), and then adding the two. The subproblem of computing F(n - 1) can itself be broken down into a subproblem that involves computing F(n - 2). Therefore, the computation of F(n - 2) is reused, and the Fibonacci sequence thus exhibits overlapping subproblems.

- Dynamic Programming is mainly used when solutions to the same subproblems are needed again and again.
- Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point in storing the solutions if they are not needed again.

TOPIC: What is an optimal substructure?

- A problem is said to have Optimal Substructure if the optimal solution of the given problem can be obtained by using the optimal solution to its subproblems instead of trying every possible way to solve the subproblems.
- For example: Consider finding a shortest path for traveling between two cities by car. Such an example is likely to exhibit optimal substructure. That is, if the shortest route from Seattle to Los Angeles passes through Portland and then Sacramento, then the shortest route from Portland to Los Angeles must pass through Sacramento too. That is, the problem of how to get from Portland to Los Angeles is nested inside the problem of how to get from Seattle to Los Angeles.
- Typically, a greedy algorithm is used to solve a problem with optimal substructure, otherwise Dynamic Programming is used.
- Whereas, a non-optimal substructure is like N-Queens problem, where you can't extrapolate solution of N-1 size chess set to get the solution of N size chess cell.

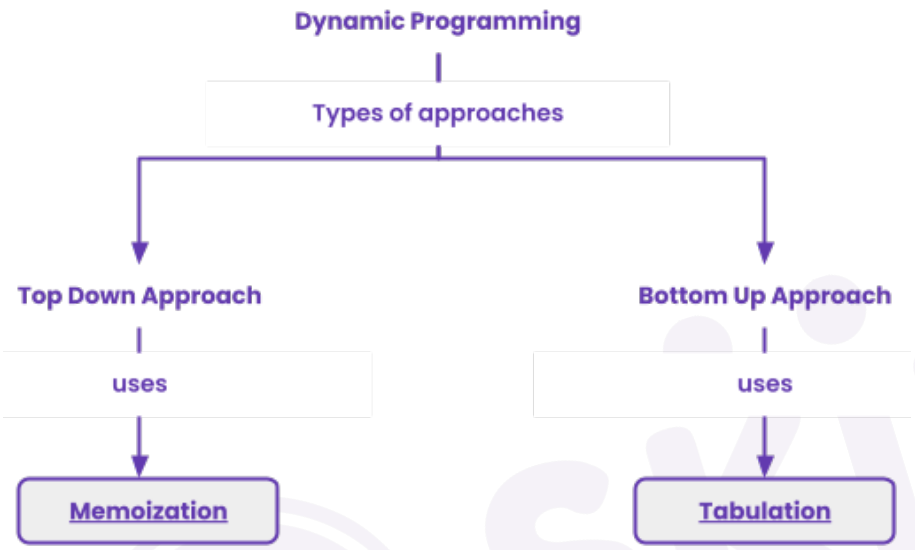
TOPIC: Greedy Algorithm VS Dynamic Programming Algorithm

GREEDY	DYNAMIC PROGRAMMING
A Greedy algorithm builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.	A dynamic programming algorithm builds up the solution to a problem by solving its subproblems recursively.
It is useful for solving problems where making locally optimal choices at each step leads to a global optimum.	It is used where the optimal solution can be obtained by combining optimal solutions to subproblems.
It does not necessarily consider the future consequences of the current choice.	Dynamic programming stores the solutions to subproblems and reuses them when necessary to avoid solving the same subproblems multiple times.
The greedy approach is generally faster and simpler, but may not always provide the optimal solution.	Dynamic programming guarantees the optimal solution but is slower and more complex.

TOPIC: Where to use Dynamic Programming?

- The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.
- Dynamic programming is useful for solving problems where the optimal solution can be obtained by combining optimal solutions to subproblems.

TOPIC: What are the various approaches of DP?



- **Top-Down(Memoization):** Break down the given problem in order to begin solving it. If you see that the problem has already been solved, return the saved answer. If it hasn't been solved, solve it and save it. This is usually easy to think of and very intuitive, This is referred to as Memoization.
- **Bottom-Up(Tabulation):** In this approach, the problem is solved by solving its subproblems in a bottom-up manner, starting from the smallest subproblem and progressively solving larger subproblems until the main problem is solved.
  The bottom-up approach typically uses iteration or loops to solve the problem and build up the solution step by step.

TOPIC: What is Memoization?

- Memoization is a specific form of caching that is used in dynamic programming that is used to speed up execution time by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.
- It basically stores the previously calculated result of the subproblem and uses the stored result for the same subproblem.
- This removes the extra effort to calculate again and again for the same problem.
- For example, consider the below recursive part of the code to calculate factorial of a number: **https://pastebin.com/HcEuxhfZ**

If you write the complete code for the above snippet, you will notice that there will be 2 methods in the code:

factorial(n) and main().
Now if we have multiple queries to find the factorial, such as finding factorial of 2, 3, 9, and 5, then we will need to call the factorial() method 4 times and each time the factorial method goes down to calculating the factorial of each number from n to 1 to compute the final result.
So for finding factorial of numbers K numbers, the time complexity needed will be O(N*K)where O(N) to find the factorial of a particular number, and O(K) to call the factorial() method K different times.

Let's see how we can optimize the time complexity here:
If we notice in the above problem, while calculation factorial of 9:
We are calculating the factorial of 2, We are also calculating the factorial of 3, and We are calculating the factorial of 5 as well.
Therefore if we store the result of each individual factorial at the first time of calculation, we can easily return the factorial of any required number in just O(1) time. This process is known as Memoization.

How Memoization works?
If we find the factorial of 9 first and store the results of individual sub-problems, we can easily print the factorial of each input in O(1).
Therefore the time complexity to find factorial numbers using memorization will be O(N): O(N) to find the factorial of the largest input and O(1) to print the factorial of each input.

TOPIC: Steps of solving a DP problem

1. **Identify if it is a Dynamic programming problem:**

- Typically, all the problems that require maximizing or minimizing certain quantities or counting problems that say to count the arrangements under certain conditions or certain probability problems
- Problems that satisfy the overlapping subproblems property and most of the DP problems also satisfy the optimal substructure property.

2. **Deciding the state**

What is the state?

- Since generally, a state is the particular condition that something is in at a specific point of time, similarly, in Dynamic Programming, a state is defined by a number of necessary variables at a particular instant that are required to calculate the optimal result.
- In other words, state is the minimum set of parameters needed to uniquely define a sub-problem.
- It is a combination of variables that will keep changing over different instants.
- Two states are the same, if all their corresponding variables have the same logical value.
- In order to verify if the state is minimal or not, try to calculate or derive one of the parameters of state with the help of others, if it's possible to do so, then remove that parameter.

3. **Formulating relation among the states**

- This is the most crucial and tough part of solving a DP problem.
- Let us understand it through an example:

Given 3 numbers {1, 3, 5}, the task is to tell the total number of ways we can form a number N using the sum of the given three numbers. (allowing repetitions and different arrangements).

Let's try to solve this problem for N = 6.

Let's look at the output first:
The total number of ways to form 6 is: 8
1+1+1+1+1+1
1+1+1+3
1+1+3+1
1+3+1+1
3+1+1+1
3+3
1+5
5+1

As we can only use 1, 3, or 5 to form a given number N, let us assume that we know the result for N = 1, 2, 3, 4, 5, 6.

Let us say we know the result for state (n = 1), state (n = 2), state (n = 3) ..... state (n = 6)
Now, we wish to know the result of the state (n = 7).

Now we can get a sum total of 7 in the following 3 ways:
Adding 1 to all possible combinations of state (n = 6)
Adding 3 to all possible combinations of state (n = 4)
Adding 5 to all possible combinations of state(n = 2)

Now, think carefully that the above three cases are covering all possible ways to form a sum total of 7.
Therefore, we can say that result for state(7) = state (6) + state (4) + state (2)
OR
state(7) = state (7-1) + state (7-3) + state (7-5)
In general,
state(n) = state(n-1) + state(n-3) + state(n-5)

This is how we pick a state and then formulate relationships amongst them.

Let's take a look at the recursive function for the above example:
https://pastebin.com/pfRynekw

4. **Adding memoization for the state**
   - The simplest portion of a solution based on dynamic programming is this.
   - Simply storing the state solution will allow us to access it from memory the next time that state is needed.
   - Let's add memoization to the above example
     https://pastebin.com/bmwa8bwZ

TOPIC: Problems on Memoization

Q1. The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n). 0 <= n < 30.

Input1: **n = 2**
Output1: **1**
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.

Input2: **n = 3**
Output2: **2**
Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.

Explanation:

- Recursively, the approach is easy. We just need to return fib(n-1) + fib(n-2) with the base case as if n == 0 || n == 1, return n.
- Now we optimize this solution by adding memoization to it.
- First we create a static array of size 30 as n can be maximum up to 30. This array will be static so that we don't have to pass it as a parameter to the recursive function.
- Initialize the array with -1 for all indices so that, we can spot if the answer for the current index has been computed already or not.
- Now instead of returning the answer in the recursive function, we store it in the array.
- First check if current index is computed yet or not.
- If not, for the base case, if n <= 1 meaning if n = 0 or 1, store the same value for the same index in the array.
- We call the recursive function as is for n-1 and n-2, just store it in the array at index N.
- If value for current index is already computed, return value of current index in the array.

CODE:
https://pastebin.com/fYBE775c

OUTPUT:

```
"C:\Program Files\Java\jdk-14.0.1\b
Enter the number:
3
2

Process finished with exit code 0
```

Q2. You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Input1: **nums = [1,2,3,1]**
Output1: **4**

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.

Input2: **nums = [2,7,9,3,1]**
Output2: **12**
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 **(money = 1).**
Total amount you can rob = 2 + 9 + 1 = 12.

EXPLANATION:

- There are two choices in front of each house: rob or not rob.
- Consider standing in front of a house. If you rob this house, then you definitely can't rob the adjacent houses, you can only start the next rob from the house after next.
- If you don't rob this house, then you can walk to the next house and continue making choices.
- When you walk past the last house, you don't have to rob. The money you could rob is obviously 0 **(base case).**
- In these two choices, you need to choose a larger result each time. You end up with the most money you can rob.
- To add memoization, we create an array of size N+1 with all indices as -1.
- In the recursive function, we pass the input array and an index variable as parameters.
- If index goes out of bounds, we return 0.
- If answer for current index has been computed, return that.
- Else, update answer for current index as max of i-1 or i-2 + houses[i].

CODE:
https://pastebin.com/tKBEMrZ5

OUTPUT:

```
"C:\Program Files\Java\jdk-14.0.1
Enter the number of houses:
4
Enter the money of each house:
1 2 3 1
4

Process finished with exit code 0
```

Q3. A message containing letters from A-Z can be encoded into numbers using the following mapping:
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"

To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:
"AAJF" with the grouping (1 1 10 6)
"KJF" with the grouping (11 10 6)
Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".
Given a string s containing only digits, return the number of ways to decode it.

Input1: s = "12"
Output1: 2
Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).

Input2: s = "226"
Output2: 3
Explanation: "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

EXPLANATION:

- Intuition behind the recursive solution: We take out 1 letter and 2 letters at a time from the string. We then check if the letters taken out are valid (can be decoded), then we recurse the function without those letter(s), otherwise assign a 0 to that component if its invalid.
- In the code below, our base case is defined as idx == s.length() which is the same as no characters left in the string anymore. We then check if we have a stored value of this substring and just return that. Otherwise, we first take out one letter, check the letter's validity and then call the function again without this letter that we took out. Once that step is completed, we then move on to doing the same thing by taking out 2 letters this time.
- Before working for 2 letters, we need to check if we have anymore letters remaining and also, if the first letter we took out was 1 or 2. This is because the numbers in our range are till 26 only.

CODE:
https://pastebin.com/pyNF9d8c

OUTPUT:

```
"C:\Program Files\Java\jdk-14.0.1\b
Enter the string:
12
2

Process finished with exit code 0
```

TOPIC: What is the Limitation of Top-DownProgramming?

- Since the process starts with the big picture, it can be difficult to make changes later on in the process.

- Breaking down the bigger problem into smaller subproblems can take a lot of time, especially if the design is complex.
- It requires massive amounts of expensive recursion.

Upcoming lectures:

- Tabulation