Find your way here

CSC 250: Foundations of Computer Science I

Fall 2023 - Lecture 5

Amitabha Dey

Department of Computer Science
University of North Carolina at Greensboro

Dynamic Programming
Fibonacci Series

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Let's take the example of the Fibonacci numbers. As we all know, Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, and 8, and they continue on from there.

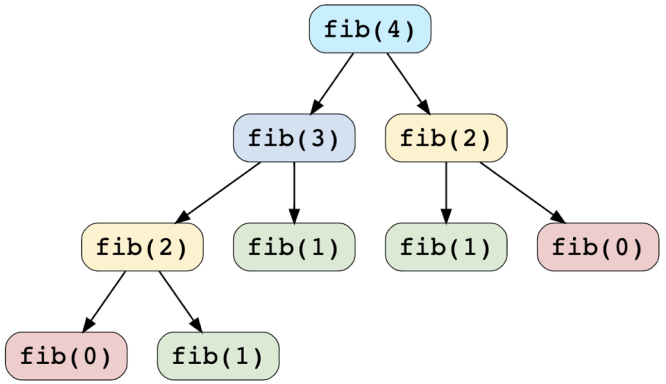
If we are asked to calculate the nth Fibonacci number, we can do that with the following equation,

```
Fib(n) = Fib(n-1) + Fib(n-2), for n > 1
```

As we can clearly see here, to solve the overall problem (i.e. Fib(n)), we broke it down into two smaller subproblems (which are Fib(n-1) and Fib(n-2)). This shows that we can use DP to solve this problem.

Subproblems are smaller versions of the original problem. Any problem has overlapping subproblems if finding its solution involves solving the same subproblem multiple times. Take the example of the Fibonacci numbers; to find the fib(4), we need to break it down into the following sub-problems:

Recursion tree for calculating Fibonacci numbers



Recursion tree for calculating Fibonacci numbers

Image credit - https://www.educative.io/

We can clearly see the overlapping subproblem pattern here, as fib(2) has been evaluated twice and fib(1) has been evaluated three times.

Dynamic Programming
Characteristics

There are two key attributes that a problem must have in order for dynamic programming to be applicable: **optimal substructure** and **overlapping sub-problems**.

If a problem can be solved by combining optimal solutions to **non-overlapping sub-problems**, the strategy is called "**divide and conquer**" instead.

- **Optimal substructure** means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of **recursion**.
- **Overlapping subproblems** means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

Every DP problem should have optimal substructure and overlapping subproblems.

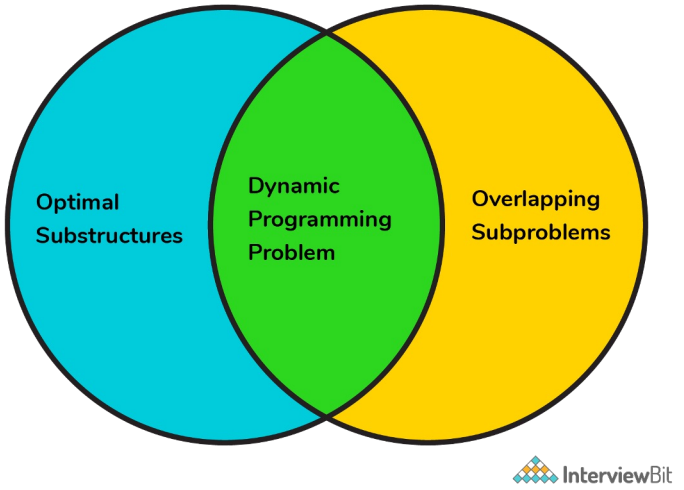
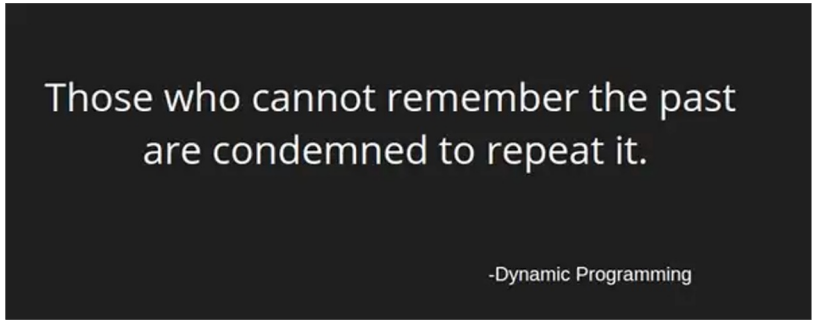


Image credit - logicmojo.com



Dynamic Programming
Optimal Substructure

A given problem is said to have Optimal Substructure Property 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. A given optimal substructure property if the optimal solution of the given problem can be obtained by finding the optimal solutions of all the sub-problems. In other words, we can solve larger problems given the solution of smaller problems.

Shortest Path Problem

Consider an undirected graph with vertices a, b, c, d, e and edges (a, b), (a, e), (b, c), (b, e),(c, d) and (d, a) with some respective weights. Find the shortest path between a and c.

This problem can be broken down into finding the shortest path between a & b and then shortest path between b & c and this can give a valid solution i.e. shortest path between a and c.

We need to break this for all vertices between a & c to check the shortest and also direct edge a-c if exists. So the following problem can be broken down into sub-problems and it can be used to find the optimal solution to the bigger problem(also the subproblems are optimal). So this problem has an optimal substructure.

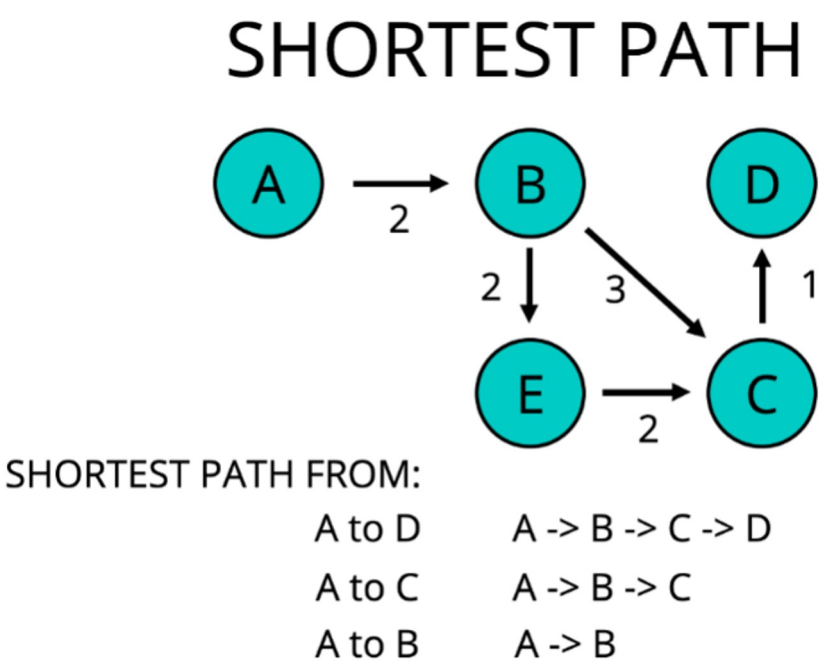


Image credit - Sogunle Omowunmi @medium.com

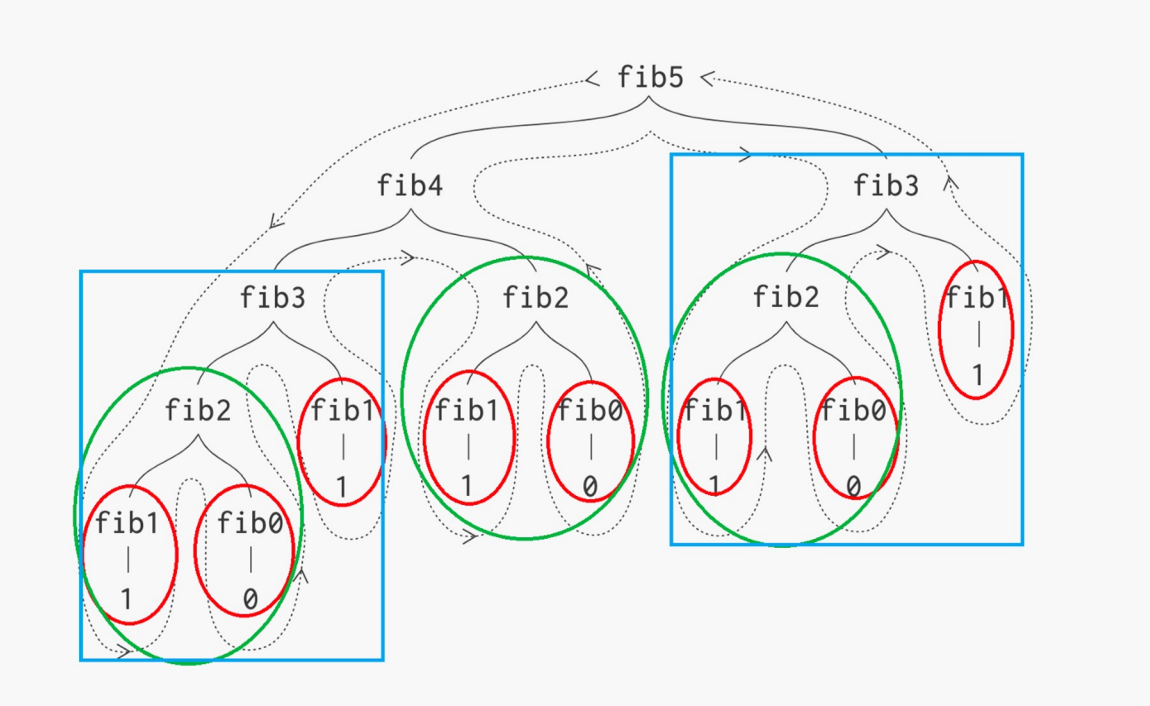
Dynamic Programming
Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming **combines solutions to sub-problems.** Dynamic Programming is mainly used when solutions to the same subproblems are needed again and again. In dynamic programming, **computed solutions to subproblems are stored in a table** so that these don't have to be recomputed.

So 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.

For example, Binary Search doesn't have common subproblems. If we take the example of following a recursive program for Fibonacci Numbers, there are many subproblems that are solved again and again.

Recursion tree highlighting the overlapping subproblems



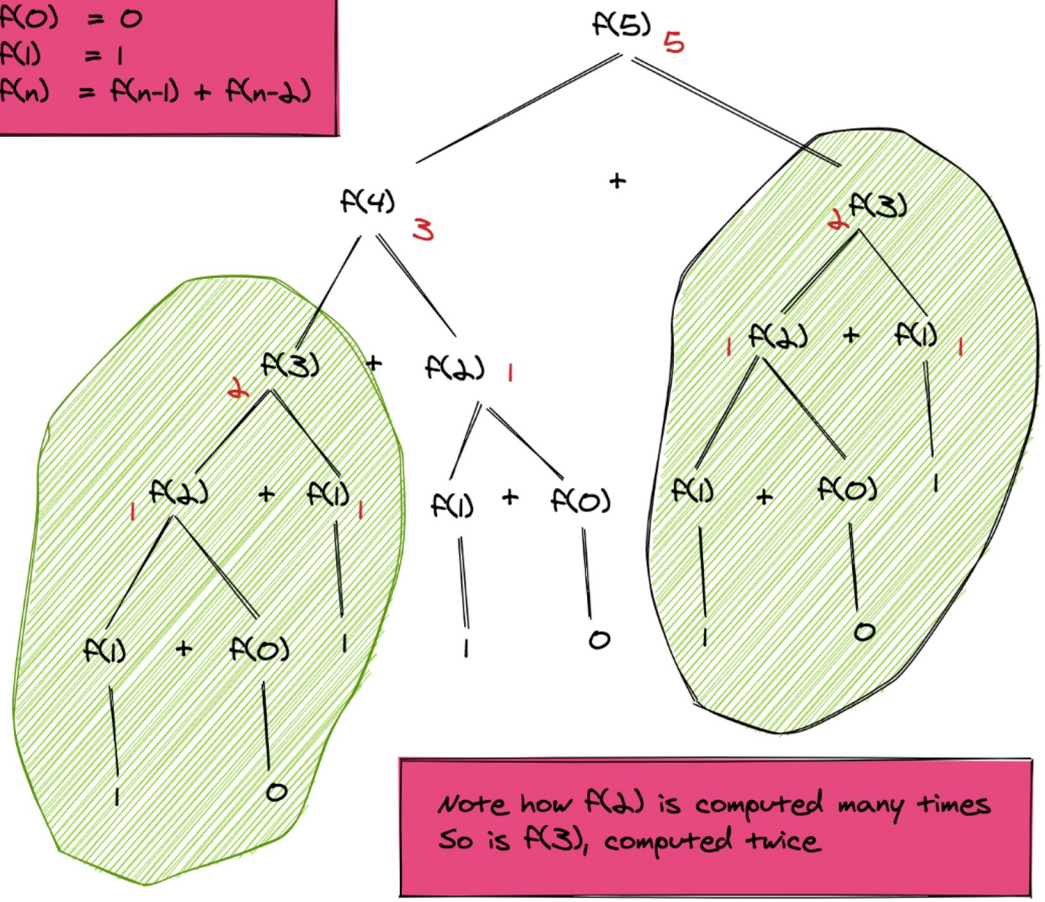
There are following two different ways to store the values so that these values can be reused:

- **Memoization (Top Down)**
  https://www.interviewcake.com/concept/java/memoization
- **Tabulation (Bottom Up)**
  https://programming.guide/dynamic-programming-vs-memoization-vs-tabulation.html

Dynamic Programming

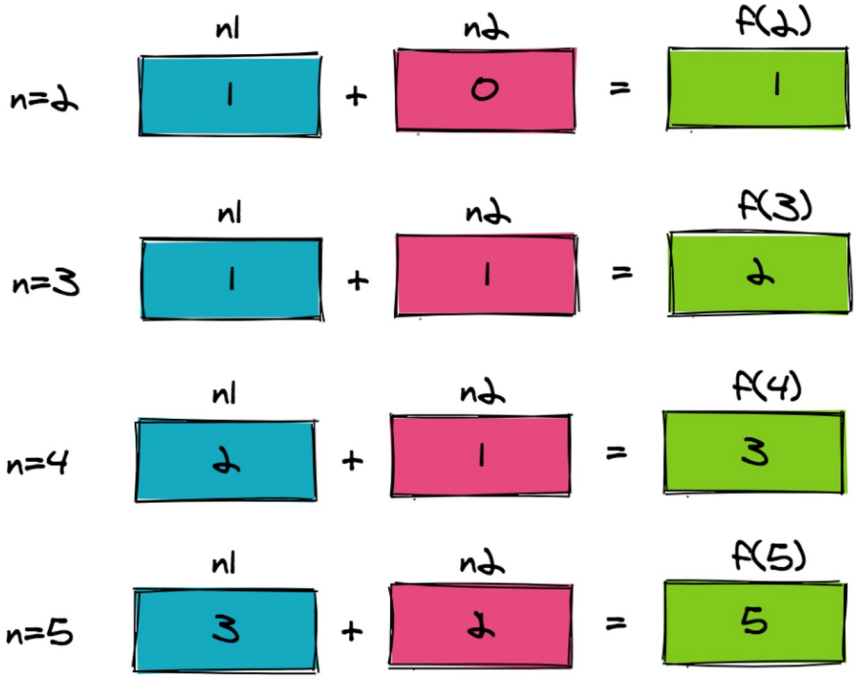
Memoization

Fibonacci Numbers
f(0) = 0
f(1) = 1
f(n) = f(n-1) + f(n-2)



Note how f(2) is computed many times
So is f(3), computed twice

f(0) = 0
f(1) = 1



Store the results for previous two numbers to achieve speed ups

Image credit - https://algodaily.com/

Dynamic Programming
House Robber Problem

A professional robber is planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping him from robbing each of them is that **adjacent houses have security system** connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money he can rob tonight without alerting the police.

```
Input: [1,2,3,1]
Output: 4
Explanation:
Option 1:
Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount he can rob = 1 + 3 = 4.
Option 2:
Rob house 2 (money = 2) and then rob house 4 (money = 1).
             Total amount he can rob = 2 + 1 = 3
So, option 1 maximizes the money stolen.
```

The problem can be cut down to sub-problem of finding whether to rob from i-th house or not.

Now, in case of taking decision whether to rob from i-th house or not, it completely depends on where the money maximizes. Since, adjacent houses are connected, the robber can't rob two adjacent house.

Either he can rob from (i-1)th house or he can rob from i-th house after robbing from (i-2)th house. To maximize the money simply find which results in maximum amount. In terms of recursive relation we can write:

```
f(i)=maximum(f(i-2)+money of ith house,f(i-1))
```

Image credit - https://www.includehelp.com/

Dynamic Programming
House Robber Problem

Image credit - https://www.includehelp.com/

Steps:

```
1.  Bases case:
    IF (n==0)
        Return 0;
    IF (n==1)
        Return money[0];
2.  Initialize DP matrix
    int house[n]; // DP matrix
    //initialize all elements with their respective money
    for(int i=0;i<n;i++)
        house[i]=money[i];

3.  IF there is only one house, then simply rob the money.
        house[0]=money[0];

4.  IF there is two house rob from the house having maximum amount
    house[1]=(money[0]>money[1])?money[0]:money[1];
5.  Formulate the recursive function:
    for(int i=2;i<n;i++){ //for more houses
        house[i]=max(house[i]+house[i-2],house[i-1]);
    }
    //house[i-2]= f(i-2) // as updated by DP matrix
    // house[i-2]= f(i-2) // as updated by DP matrix
    // house[i]( on R.H.S) = money of ith house // as not still updated by DP matrix
    // house[i] (on L.H.S) = f(i) // as it's going to be updated by DP matrix

6.  Return house[n-1]
```

Example with explanation:

```
Let's solve an example using the algorithm:

No of houses: 6
Money stashed at houses:
4, 2, 2, 10, 4, 2

Initially DP matrix
4 (0th) | 2 (1st) | 2(2nd) | 10 (3rd) | 4 (4th) | 2 (5th)

After step -3 & step - 4:
4 (0th) | 4 (1st) | 2(2nd) | 10 (3rd) | 4 (4th) | 2(5th)

At step-5:
House index starts from 0

Iteration 0:
house[2]=max(house[2]+house[0],house[1]);
house[2]=6 // house[2]+house[0]=6 &house[1]=4
that means rob at house 0 & then at house 2
4 (0th) | 4 (1st) | 6(2nd) | 10 (3rd) | 4 (4th) | 2 (5th)

Iteration 1:
house[3]=max(house[3]+house[1],house[2]);
house[3]=14 // house[3]+house[1]=14&house[2]=6
that means rob at house 1 & then at house 3 (decision changed)
4 (0th) | 4 (1st) |     6(2nd) | 14 (3rd) |     4 (4th) | 2 (5th)

Iteration 2:
house[4]=max(house[4]+house[2],house[3]);
house[4]=14 // house[4]+house[2]=10&house[3]=14
that means rob at house 1& then at house 3 (no change in last decision)
4 (0th) | 4 (1st) |     6(2nd) | 14 (3rd) |     10 (4th) | 2 (5th)

Iteration 3:
house[5]=max(house[5]+house[3],house[4]);
house[5]=16 // house[5]+house[3]=16&house[4]=10
that means rob at house 1 & then at house 3 & then at house 5(final)
4 (0th) | 4 (1st) |     6(2nd) | 14 (3rd) |     10 (4th) | 16 (5th)
```

Dynamic Programming
0-1 Knapsack Problem

For applying Dynamic programming to this problem we have to do three things in this problem:

- Optimal substructure
- Writing the recursive equation for substructure
- Whether subproblems are repeating or not

Now assume we have 'n' items 1 2 3 ... N. I will take an item and observe that there are two ways to consider the item either

- it could be included in knapsack
- or you might not include it in knapsack

Likewise, every element has 2 choices. Therefore we have 2X2X2X2... Upto n choices i.e **2^n choices**.

We have to consider the 2^n solution to find out the optimal answer but now we have to find that is there any repeating substructure present in the problem so that exempt from examining 2^n solutions.

The recursive equation for this problem is given below:

```
knapsack(i,w) = { max( Vi +knapsack(i-1,W-wi) , knapsack(i-1,W) )
                                    0,i=0 & W=0
                                    Knapsack(i-1,W)    , wi> W
}
```

Image credit - https://www.includehelp.com/

- **Knapsack(i-1,W)**: is the case of not including the ith item. In this case we are not adding any size to knapsack.
- **Vi + Knapsack(i-1,W-wi)**: indicates the case where we have selected the ith item. If we add i-th item then we need to add the value Vito the optimal solution.
- Number of unique subproblems in 0-1 knapsack problem is (n X W). We use tabular method using Bottom-up Dynamic programming to reduce the time from O(2^n) to O(n X W).

Dynamic Programming
0-1 Knapsack Problem

In the divide-and-conquer strategy, you divide the problem to be solved into subproblems. The subproblems are further divided into smaller subproblems. That task will continue until you get subproblems that can be solved easily. However, in the process of such division, you may encounter the same problem many times.

The basic idea of Knapsack dynamic programming is to use a table to store the solutions of solved subproblems. If you face a subproblem again, you just need to take the solution in the table without having to solve it again. Therefore, the algorithms designed by dynamic programming are very effective.

To solve a problem by dynamic programming, you need to do the following tasks:

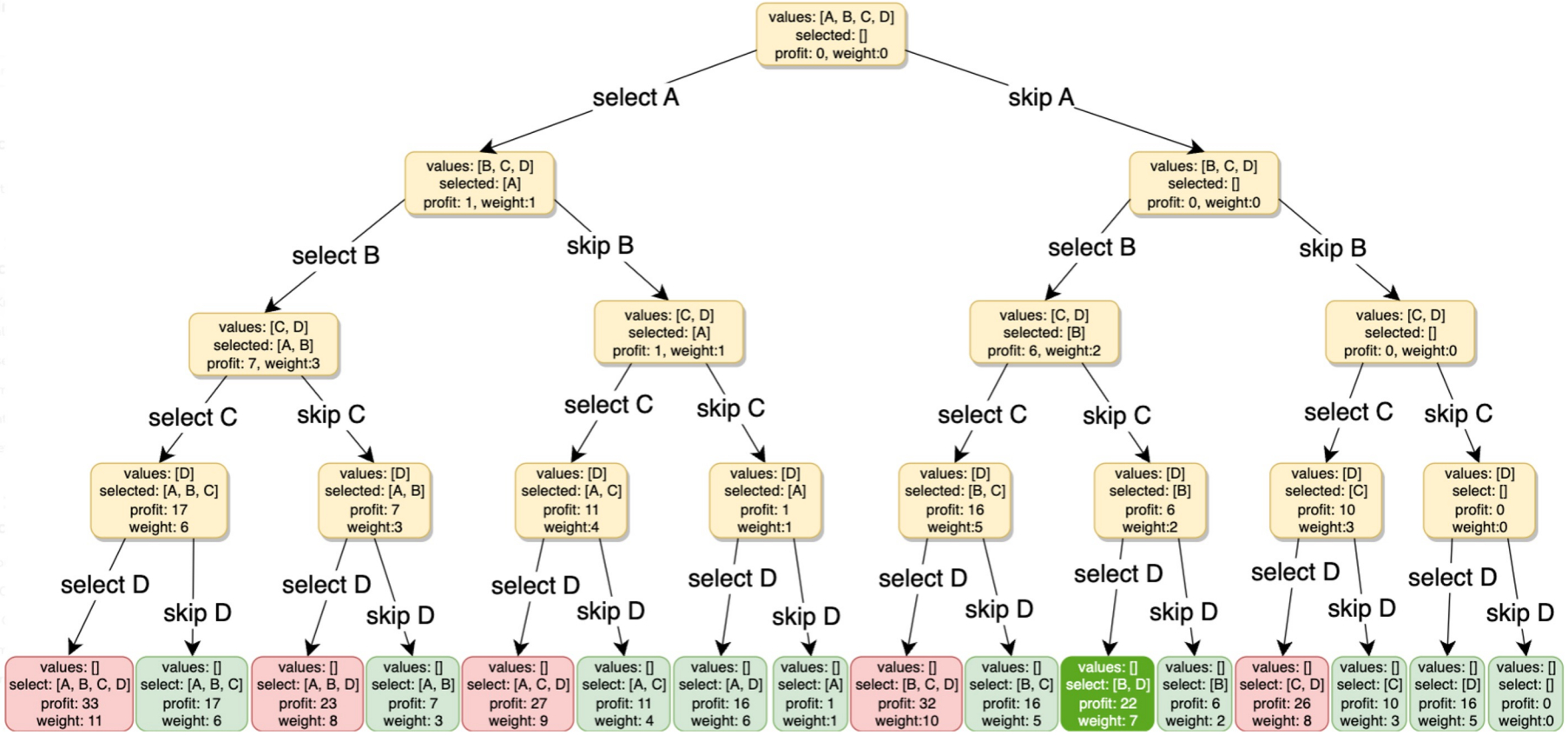
- Find solutions of the smallest subproblems.
- Find out the formula (or rule) to build a solution of subproblem through solutions of even smallest subproblems.
- Create a table that stores the solutions of subproblems. Then calculate the solution of subproblem according to the found formula and save to the table.
- From the solved subproblems, you find the solution of the original problem.

Image credit - https://www.educative.io/

Dynamic Programming
0-1 Knapsack Problem

items	A	B	C	D
profit	1	6	10	16
weight	1	2	3	5

Capacity: 7



Dynamic Programming
Word Break Problem

Given a dictionary, you have to split a given string into meaningful words.

Example:

```
Case-I:
If the dictionary contain the words:
{"like", "i" , "ice" , "cream", "is"};
Input : "ilikeicecream"
Output: "i like ice cream"

Case-II:
If the dictionary contain the words:
{"like", "i" , "ice" , "cream", "is"}
Input: "ilikeeicecream"
Output: False
(There is no combination possibleout from the dictionary)
```

We solve the problem using dynamic programming. The problem contains two parts one is detecting the words and the other one is retrieving the words.

Case-I

- We take a Boolean array of length the same as the length of the string and initialize it with false.
- We take a vector and initialize it with -1.
- We start comparing the string from position 0 and increment the length at each time by one.
- Whenever we find a meaningful word we push_back() that index into the vector and make that index in the boolean array true.
- After inserting the index then we start checking the next word from that index and also the previous index.
- Repeat step 2 to step 5 until we traverse the whole string.
- If the (length -1) index in the boolean array is true then we separate the string otherwise we don't.

Case-II

- We take the substring from the last element of the vector to the last of the string and check to the dictionary.
- If found then next time last will be the last element of the vector.
- If not found then only go to the next to last element of that vector.
- Repeat the process until we go to the first element of the vector.

Dynamic Programming
Word Break Problem

Examples:

Input:
s = "applepenapple"
words = ["apple", "pen"];

Output: True
Explanation:
The string "**applepenapple**" can be broken into "apple pen apple"

Input:
s = "catsandog"
words = ["cats", "dog", "sand", "and", "cat"]
Output: False

The intuition behind this approach is that the given problem (s) can be divided into subproblems s1 and s2.

If these subproblems individually satisfy the required conditions, the

e.g. "catsanddog" can be split into two substrings "catsand", "dog". The subproblem "catsand" can be further divided into "cats","and", which individually are a part of the dictionary making "catsand" satisfy the condition.

Going further backwards, "catsand", "dog" also satisfy the required criteria individually leading to the complete string "catsanddog" also to satisfy the criteria.

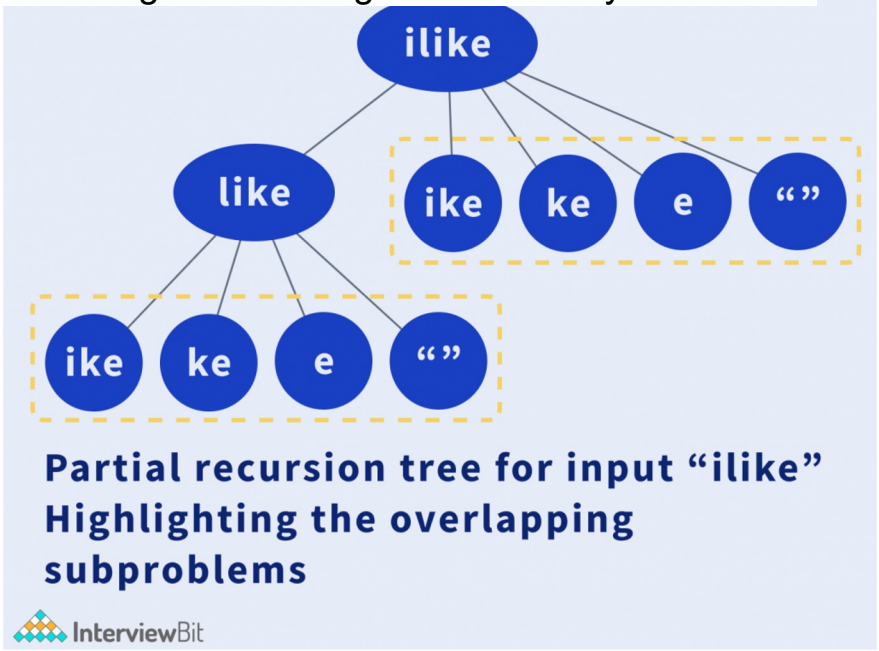


Image credit - https://www.interviewbit.com/

Dynamic Programming
Dice Throw Problem

Given **n** dice each with **m** faces, numbered from **1** to **m**, find the number of ways to get sum **X** dice are thrown.

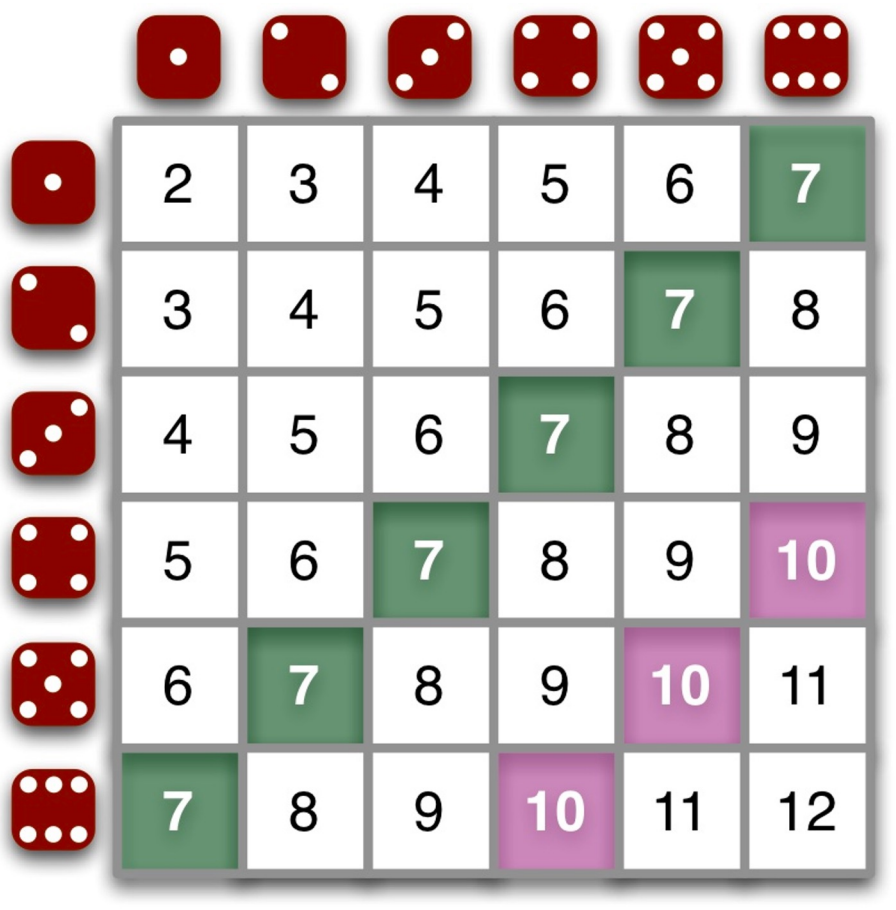
```
Input:
n=3
m=3
X=6

Output:
Total number of ways are: 7
```

Explanation:

```
Total number of dices: 3 say x1,x2,x3
Number of faces on each dice: 3 (1 to 3)
Total sum to be achieved: 6
We will write as xi(j)which means face value of dice xi is j
So sum 6 can be achieved in following ways:
6=x1(1)+x2(2)+x3(3)
6=x1(1)+x2(3)+x3(2)
6=x1(2)+x2(2)+x3(2)
6=x1(2)+x2(3)+x3(1)
6=x1(2)+x2(1)+x3(3)
6=x1(3)+x2(2)+x3(3)
6=x1(3)+x2(3)+x3(1)

This are total 7 ways to achieve the sum.
```



More details - https://www.includehelp.com/icp/dice-throw.aspx

Dynamic Programming
MIT OpenCourseware Notes

- Dynamic Programming I: Memoization, Fibonacci, Shortest Paths, Guessing
  https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/resources/mit6_006f11_lec19/
- Dynamic Programming II
  https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/resources/mit6_006f11_lec20/
- Dynamic Programming III
  https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/resources/mit6_006f11_lec21/
- Dynamic Programming IV
  https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/resources/mit6_006f11_lec22/
- Dynamic Programming Lecture (Video)
  https://ocw.mit.edu/courses/6-00sc-introduction-to-computer-science-and-programming-spring-2011/resources/lecture-23-dynamic-programming/