BOSSCODER ACADEMY

Dynamic Programming

It's just a fancy way of saying

“Remember stuff now, to save time later.”

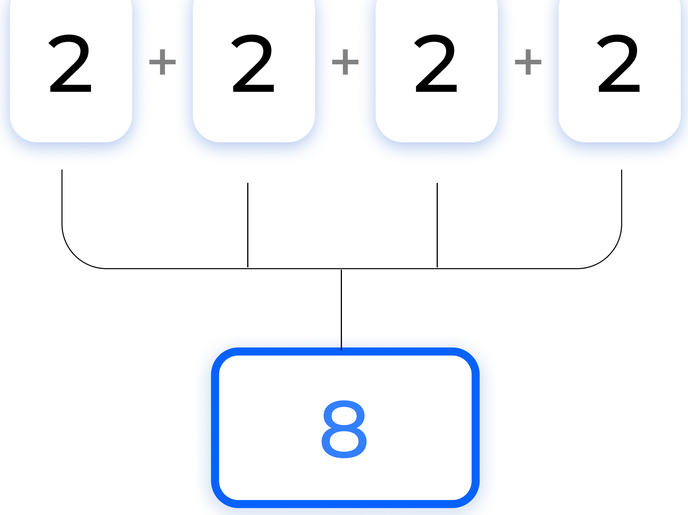
BOSSCODER ACADEMY

Problem Statement

To understand dynamic programming in the simplest terms, consider the following example.

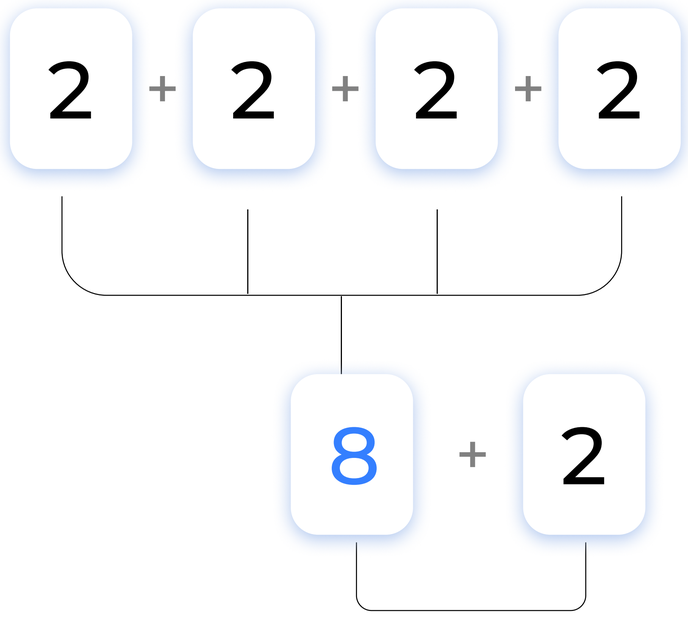
What is ?

2 + 2 + 2 + 2 = ?



Now, What is ?

2 + 2 + 2 + 2 + 2 = ?



We can arrive at the answer super quickly -

As we already know the answer to the previous question and by just adding another 2,

We can say the answer is 10!

BOSSCODER ACADEMY

Explanation

This can be represented by a question of the format, what is the sum of 2 taken n times .

To solve this we can consider an array f, where f[n] represents the sum of 2 taken n times, that is 2*n.

Mathematically, since we know that multiplication is repeated addition, the sum of 2 taken n times will always be equal to the sum of 2 taken n-1 times and 2 itself.

That is f(n)=f(n-1)+2

is a resultant recurrence relation for this problem

So if we have say n=4,

sum of 2 taken 3 times is 2+2+2=6

f(n-1)=f(3)=6

And f(4)=f(3)+2=6+2=8

BOSSCODER ACADEMY

To solve this question through code, we have to first initialise the base cases or values of f(n) for small values of n.

```
f(0)=0
( sum of 2 taken 0 times is zero)
f(1)=2
(sum of 2 taken 1 time is two)

int count2(int n)
{
    //creating an array to store the previous values
    int f[n+1];

   //initialise the base cases
    f[0]=0;
    f[1]=2;
    //iterate through 2 to n
    for(int i=2;i<=n;i++)
    {
        f[i]=f[i-1]+2;
    }
    return f[n];
}
```

BOSSCODER ACADEMY

Complexity Analysis:

Time Complexity: O(n).

The array is traversed completely until n. So Time Complexity is O(n).

Space Complexity: O(n).

To store the values in the f array, ‘n’ extra space is needed.

BOSSCODER ACADEMY

Dynamic programming vs Recursion

The basic concepts of dynamic programming are similar to recursion. You first calculate the answers to the different sub-problems and then combine it to get the final answer.

Dynamic programming trades space for time.It uses more space to store the results of sub-problems to reduce time taken rather than taking a lot of time to calculate sub-problem solutions and saving space.

For Example

Consider the problem of finding out the nth Fibonacci number.

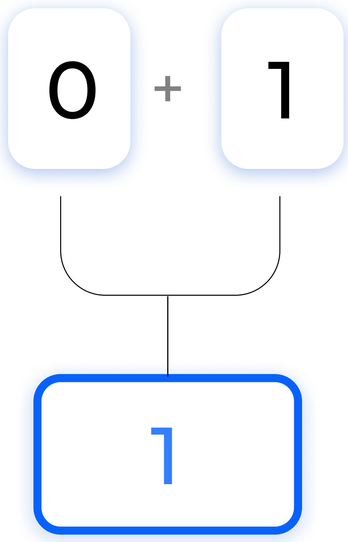
A fibonacci series is defined as a series of numbers, where each number is the sum of the previous two numbers.

So if we start of with 0 and 1, the next number in the series would be 1, since 0+1=1
Similarly the next number would 1+1=2

BOSSCODER ACADEMY

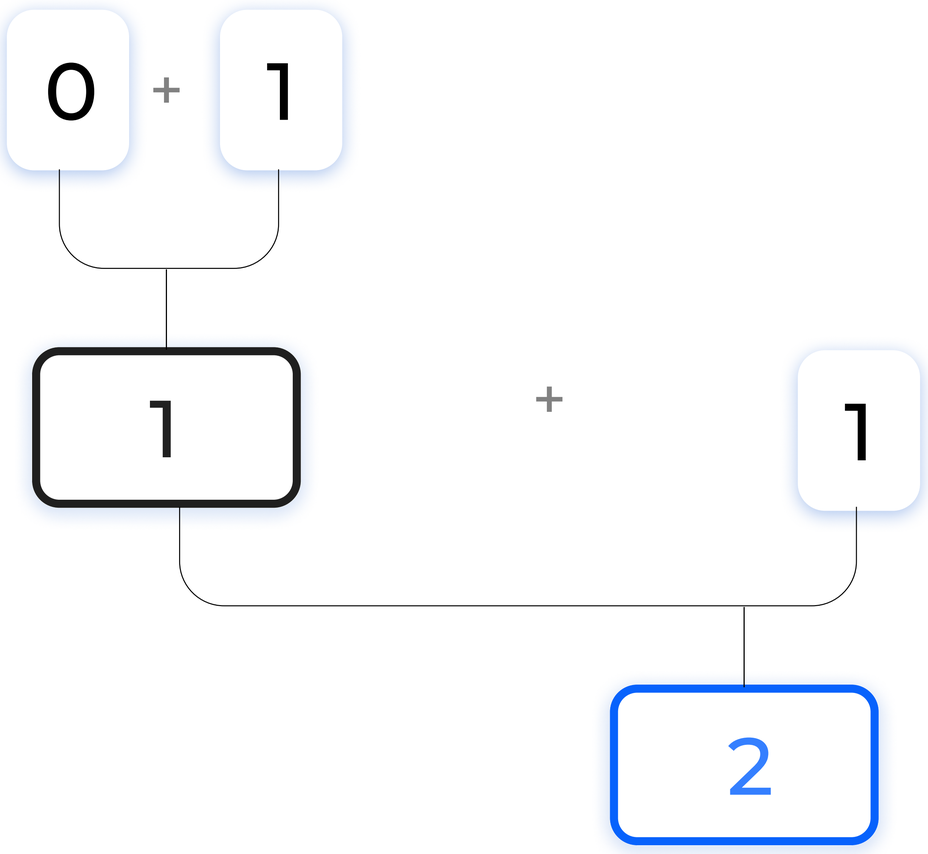
What is ?

0 + 1 = ?



Now, what is ?

0 + 1 +1 = ?



0,1,1,2,3,5,8.....
In general we get a recurrence relation as
fib(n) = fib(n-1) + fib(n-2)

BOSSCODER ACADEMY

To calculate the nth number of Fibonacci series, we can either use a **recursive approach** or **dynamic programming approach**.

```
int  fib(int  n)
{
     if (n   <=  1)
        return n;
     return fib(n-l)  +  fib(n-2) ;
}
```

Recursion :
Exponential

```
f[0] = 0;
f[1] = 1;

for (1 = 2; 1 <= n; i++)
{
   f[i] = f[i-1] + f[i-2];
}

return f(n];
```

Dynamic Programming :
Linear

If we use the recursive approach, it would take exponential time for large values of n.

By using dynamic programming, the time complexity reduces to linear, as the data is stored in an array.

Computation time is reduced as you simply need to access previously calculated values to generate the nth fibonacci number.

During recursion, sometimes solutions to subproblems may be calculated many times.

Consider the same example of calculating the nth fibonacci number.

- **fib(n) = fib(n-1) + fib(n-2)**
- **fib(n-1) = fib(n-2) + fib(n-3)**
- **fib(n-2) = fib(n-3) + fib(n-4)**

Here for different values of n, fib(n-3) is calculated multiple times.

In dynamic programming, by storing the result of fib[n-3] in an array, it can be accessed easily.

So in the dynamic programming approach we calculate the value of each element in the fib array only once and that value is used for the subsequent calculations.

BOSSCODER ACADEMY

Dynamic programming works when a problem has the following characteristics:

- Optimal Substructure: If an optimal solution contains optimal subsolutions, then a problem exhibits optimal substructure.
- Overlapping subproblems: When a recursive algorithm visits the same subproblems repeatedly, then a problem has overlapping subproblems.

In Divide and Conquer technique the subproblems are independent of each other.

In Dynamic Programming, the subproblems are dependent on each other and they overlap.

There are two different methods to store pre calculated values to save time.

Tabulation

In this method we follow a **bottom up approach.**

If you want to calculate the factorial of a given number (n)!, you can store the previous products (n-1)! and just multiply it with n.

The relation can be expressed as **f[n]=f[n-1]*(n)**

Here you first calculate the values of f[0], f[1], f[2]..... And then calculate the value of f[n], that is the lower or bottom values are calculated first and then higher values are derived from them.

```
int f[MAXN];
int f[0] = 1; //base value
for (int i = 1; i< =n; i++)
{
  f[i] = f[i-1] * i;
  //sequentially updating the table - tabulation
}
```

BOSSCODER ACADEMY

Memoization

Memoization is a form of caching and is used to **optimise recursion.**

It remembers the result of a function call with particular inputs in a lookup table (generally referred to as the **"memo"**) and returns that result when the function is called again with the same inputs.

Pseudocode for memoization method to calculate factorial:

```
If n== 0,
            return 1
Else if n is in the memo
            return the memo's value for n
Else
            Calculate (n−1)!×n
            Store result in the memo
            Return result
```

BOSSCODER ACADEMY

Algorithm

Dynamic Programming algorithm is designed using the following four steps

1. Characterise the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from the computed information.

Famous Applications of Dynamic Programming are:

- **0/1 Knapsack Problem**
- **Matrix Chain Multiplication**
- **All Pairs Shortest Path in Graphs**

BOSSCODER ACADEMY

Types of Dynamic Programming Problems:

There are different types of problems that can be solved using dynamic programming.

They usually vary on the type of tabulation approach used or by using in combination with methods like Bit masking.

Let’s look at each type of dynamic programming problems:

1. 1 Dimensional Dynamic Programming
In this method, the problem can be easily solved by using a one dimensional array.

Few examples for 1D Dynamic Programming:

- Stair Problem
- Pairing Problem
- Two Dimensional DP
- Bitwise Dynamic Programming

BOSSCODER ACADEMY

Stair Problem:

A person when running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time.

For Example

Find out how many possible ways the person can run up the stairs.

If there are 3 stairs in the staircase, the person can run up in 4 ways.

The four ways:

- 1 step + 1 step + 1 step
- 1 step + 2 step
- 2 step + 1 step
- 3 step

To implement a dynamic programming approach you consider a tabular approach where initial values are stored for a smaller number of steps.

BOSSCODER ACADEMY

Algorithm:

1. Create a 1d array dp of size n+1
2. Initialise the array with base cases as following dp[0]=1, dp[1]=1, dp[2]=2.
3. Run a loop from 3 to n.
4. For each index i, compute the value of ith position as dp[i] = dp[i-1] + dp[i-2] + dp[i-3].

That is to reach the ith stair, we count the number of ways to reach (i-1)th stair + (i-2)th stair + (i-3)th stair +

1. Print the value of dp[n], as the Count of the number of ways to run up the staircase.

Complexity Analysis:

Time Complexity: O(n).

The array is traversed completely. So Time Complexity is O(n).

Space Complexity: O(n).

To store the values in a dp array, ‘n’ extra space is needed.

BOSSCODER ACADEMY

Pairing Problem:

Given the number of people to be ‘n’,every person can either stay single or be paired with another person. Each person can be paired only once.

Find the total number of combinations of pairs and single people occurring for a given number of people.

For Example

Consider there are 3 people
n = 3

Possible combinations can be

{1}, {2}, {3} : all the people remain single
{1}, {2, 3} : 2 and 3 paired but 1 is single.
{1, 2}, {3} : 1 and 2 are paired but 3 is single.
{1, 3}, {2} : 1 and 3 are paired but 2 is single.

It should be noted that the order of pairings doesn't matter that is {1,2} is same as {2,1}
There are 4 total combinations with 3 people.

Again the first step is to obtain a mathematical relationship for ‘n’ and smaller values of n.

If you consider f(n) to denote the total number of combinations for n people then for any f(n) there are two possible cases:

The nth person can remain single and you have to calculate the combinations for remaining n-1 people.

or

The nth person can be paired with any of the remaining n-1 people, and you have to calculate the combinations for remaining n-2 people (as 2 people are paired)

So mathematically the relation obtained is
f(n)= f(n-1)+ (n-1)*f(n-2)
Next step, base cases have to be calculated and initialised.

f(0)=1
f(1)=1
(1 person has only one combination, remaining single)
f(2)=2
(2 people have 2 combinations, either be paired or both remain single)

The code can now be written as:

```
{  int countFriendsPairings(int n)
     {
       int f[n]; //array to store all pre
                    calculated values

                 //iterate through 3 to n
                 for (int i = 3; i <= n; i++) {
                         f[i] = f[i - 1] + (i - 1) * f[i - 2];
     }

       //for any n number of people, total
         combinations are stored in f[n]

         return f[n];
}
```

BOSSCODER ACADEMY

Two dimensional DP:

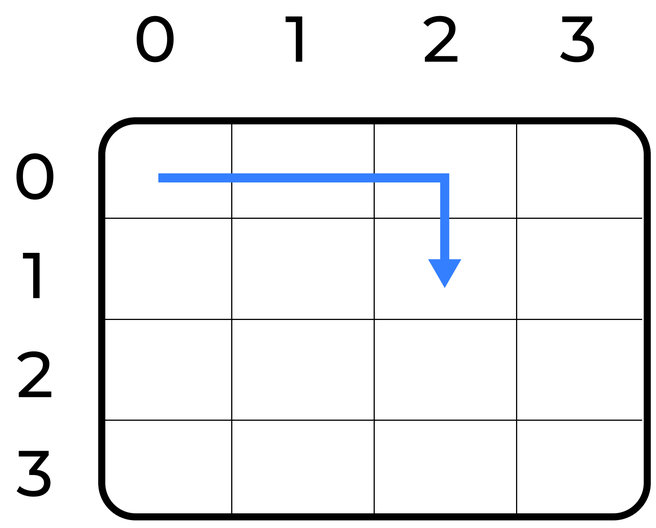
In two dimensional dynamic programming, we require a 2d array to store the precalculated values.

In many problems, based on grids or graphs, we use this kind of dynamic programming approach.

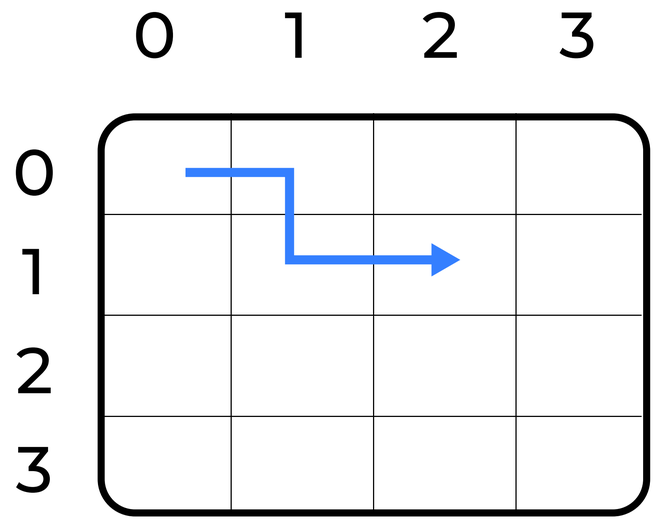
For Example

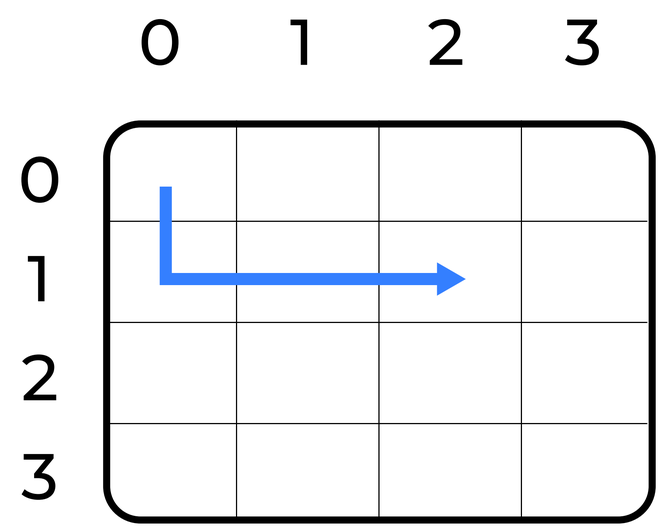
Given a grid or a two dimensional matrix with ‘m’ rows and ‘n’ columns. Starting from (0,0), if you can only move one step right or one step down, find the total number of ways to reach a position (i,j) in the matrix. As an example, consider that you want to reach the position (1,2) from (0,0).

There are 3 possible paths that can be taken.



(0,0) ⟶ (0,1) ⟶ (0,2) ⟶ (1,2)





Therefore the number of ways to reach cell (i,j) will be equal to the sum of number of ways of reaching (i-1,j) and number of ways of reaching (i,j-1).

If we consider f(i,j) to represent the number of ways to reach cell (i,j) in the matrix, the recurrence relation is formed as:

f(i,j)=f(i-1,j) + f(i,j-1)

Therefore the number of ways to reach cell (i,j) will be equal to the sum of number of ways of reaching (i-1,j) and number of ways of reaching (i,j-1).

If we consider f(i,j) to represent the number of ways to reach cell (i,j) in the matrix, the recurrence relation is formed as:

f(i,j)=f(i-1,j) + f(i,j-1)

Similar to 1d dynamic programming problems, we need to initialise the base cases.
In this problem, the base cases are the first row (in yellow) and leftmost column (in green).

All the cells in the leftmost column can only be reached in one way, by moving down from the cell above them.

1	1	1	1
1			
1			
1			

All the cells in the first row can only be reached in one way, by moving right from the cell to the left of them.

BOSSCODER ACADEMY

The code can now be written as:

```
int countnoways(m,n)
{
    //No of ways to reach (m,n)
    //We can optimise space by creating grid of
       m rows and n columns

    int f[m][n];//declare the f matrix
    f[0][0] = 1;

    //base cases
    //leftmost column

     for (int i = 1; i < m; i++)
     f[i][0] = 1;

    //firstrow

    for (int j = 1; j < n; j++)
    f[0][j] = 1;
    for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++){
```

```
        //recurrence relation iterating from 1

         f[i][j] = f[i - 1][j] + f[i][j - 1];
    }
    {
        //no of ways to reach m,n is stored in f[m-1]
           [n-1]
         return f[m - 1][n - 1];
    }
```

Complexity Analysis:

Time Complexity: O(m*n)
As we have iterated completely using two nested loops.

Auxiliary Space : O(m*n)
For storing the values we used a two dimensional array of m rows and n columns.

BOSSCODER ACADEMY

Bitwise Dynamic Programming

In this method, we generally use the concept of bit masking for getting efficient solutions.

Bit masking basically refers to using integers as a way to represent subsets of a given set of numbers using binary representation.

For example consider a set A {1,3,4,6,2}

A subset of this set {1,3,4} can be represented as 0000....1101 in the form of binary.

Here if ith digit from right is 1, then the corresponding ‘i’ is present in the set.

So since, 1st, 3rd, 4th digits are 1’s in our binary representation, the corresponding subset is {1,3,4}.

Now 000...1101 can be evaluated as the integer 13. So the integer 13 is used to represent this particular subset in bitmasking.

BOSSCODER ACADEMY

Problems that can be solved using Bit Masking and Dynamic Programming together are:

→**Count ways to assign unique cap to every person**

https://www.geeksforgeeks.org/bitmasking-and-dynamic-programming-set-1-count-ways-to-assign-unique-cap-to-every-person/

https://bit.ly/2PIUjJp

→**Number of ways to select n pairs of candies of distinct colors**

https://www.geeksforgeeks.org/count-ways-to-select-n-pairs-of-candies-of-distinct-colors-dynamic-programming-bitmasking/

https://bit.ly/3OycTiU

BOSSCODER ACADEMY

Must Do Problems in Dynamic Programming :

1. 0/1 Knapsack Problem
2. Count all increasing subsequences
3. Longest Palindromic Subsequence
4. Weighted job scheduling
5. Minimum Cost To Make Two Strings Identical
6. Longest Common Subsequence
7. Longest Increasing Subsequence
8. Coin Change Problem
9. Minimum Partition
10. Count all subsequences having product less than K
11. Maximum size square sub-matrix with all 1s
12. Largest Sum Contiguous Subarray

BOSSCODER ACADEMY

13. Weighted Job Scheduling

14. Make Array Strictly Increasing

15. Max Sum of Rectangle No Larger Than K

16. Boolean Parenthesization Problem

17. Maximum difference of zeros and ones in binary string

18. Largest Divisible Subset

19. Number of Submatrices That Sum to Target

20. Edit Distance

21. Interleaving String

22. Matrix Chain Multiplication

BOSSCODER ACADEMY

WHY BOSSCODER

200+ alumni placed at **Top product- based companies**

136% (Avg) hike for **working professionals**

22 LPA (Avg) package for **Bosscoder alumni**

Lavanya Meta

The syllabus is most up-to-date and the list of problems provided covers all important topics.

Rahul Google

Course is very well structured and streamlined to crack any MAANG company

Explore More →