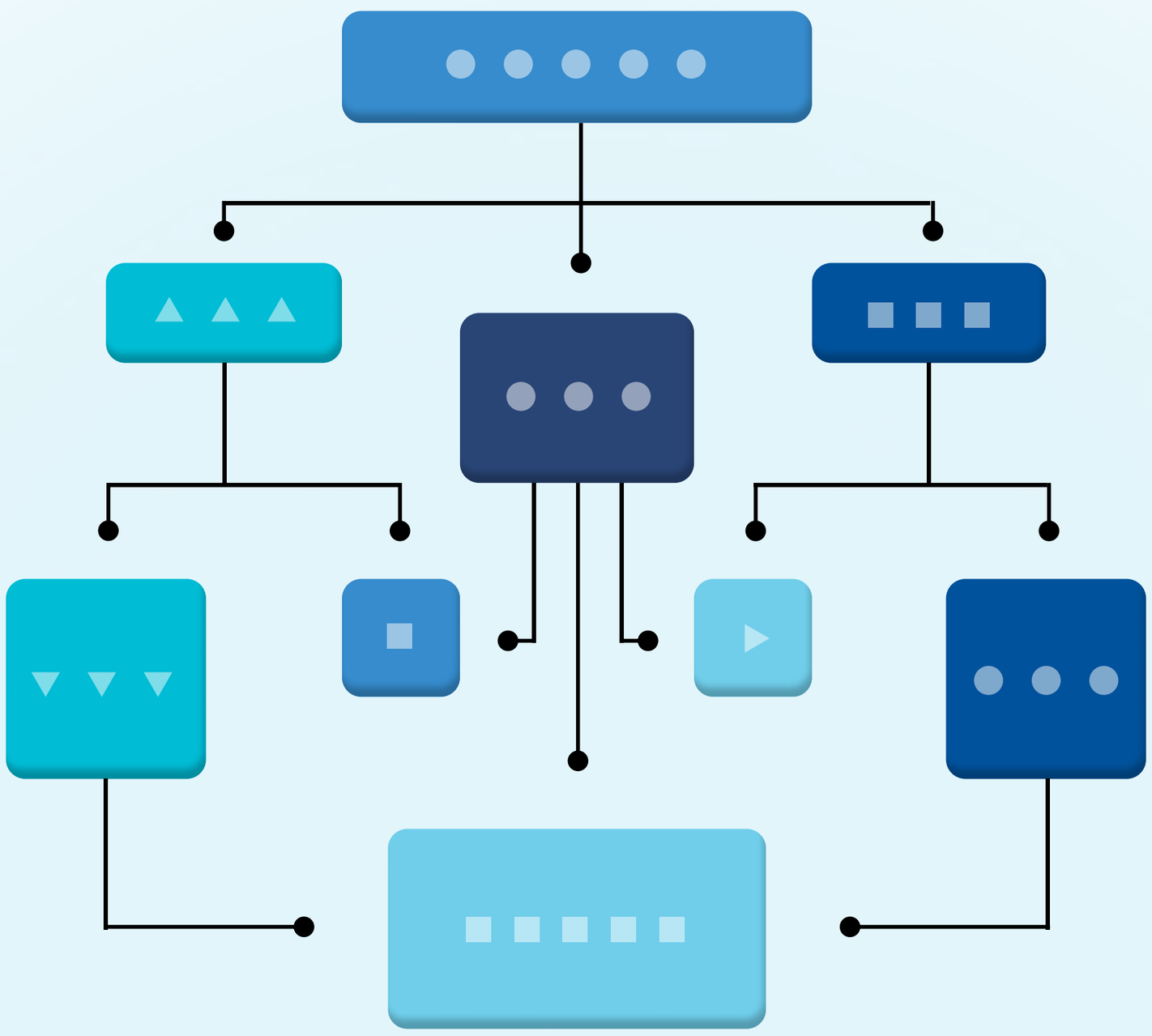
BOSSCODER ACADEMY

DYNAMIC PROGRAMMING

MADE EASY



Become a Master in 15 days!

DYNAMIC PROGRAMMING

It's just a fancy way of saying

“Remember stuff now, to save time later.”

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

What is 2+2+2+2?
→8

Now what is 2+2+2+2+2?
→10

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!

This can be represented by what is the sum of 2 taken n times?

To solve this we can consider an array f

f[n] represents the sum of 2 taken n times, that is 2*n.

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

For Example:

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

Let's solve this question through code

Initialising 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];
}
```

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.

DYNAMIC PROGRAMMING VS RECURSION

- The basic concepts of dynamic programming are similar to recursion.
- 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.

Example:

Consider the problem of finding out the nth Fibonacci number.

- A fibonacci series is defined as a series of numbers
- Each number is the sum of the previous two numbers.

Starting off 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
0,1,1,2,3,5,8.....

In general recurrence relation is
fib(n) = fib(n-1) + fib(n-2)

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

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

Recursion : Exponential

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

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

Dynamic Programming : Linear

In the recursive approach, it would take exponential time for large values of n.

In dynamic programming, the time complexity reduces to linear, as the data is stored in an array.
During recursion, 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 needs to be calculated only once

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.

COMPLEXITY ANALYSIS

It is a bottom up approach.

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)

```
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
}
```

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")

- That result is returned 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
```

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:

1. 0/1 Knapsack Problem
2. Matrix Chain Multiplication
3. All Pairs Shortest Path in Graphs

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

Example:

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.

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 +
5. 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.

Dynamic programming can only be mastered by practice. So, just keep practicing and you will start solving problems on your own in no time.

Happy Coding!

BOSSCODER ACADEMY

WHY BOSSCODER?

1000+ Alumni placed at Top Product-based companies.

More than **136% hike** for every **2 out of 3** working professional.

Average package of **24LPA.**

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

Lavanya
Meta

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

Rahul .
Google

EXPLORE MORE