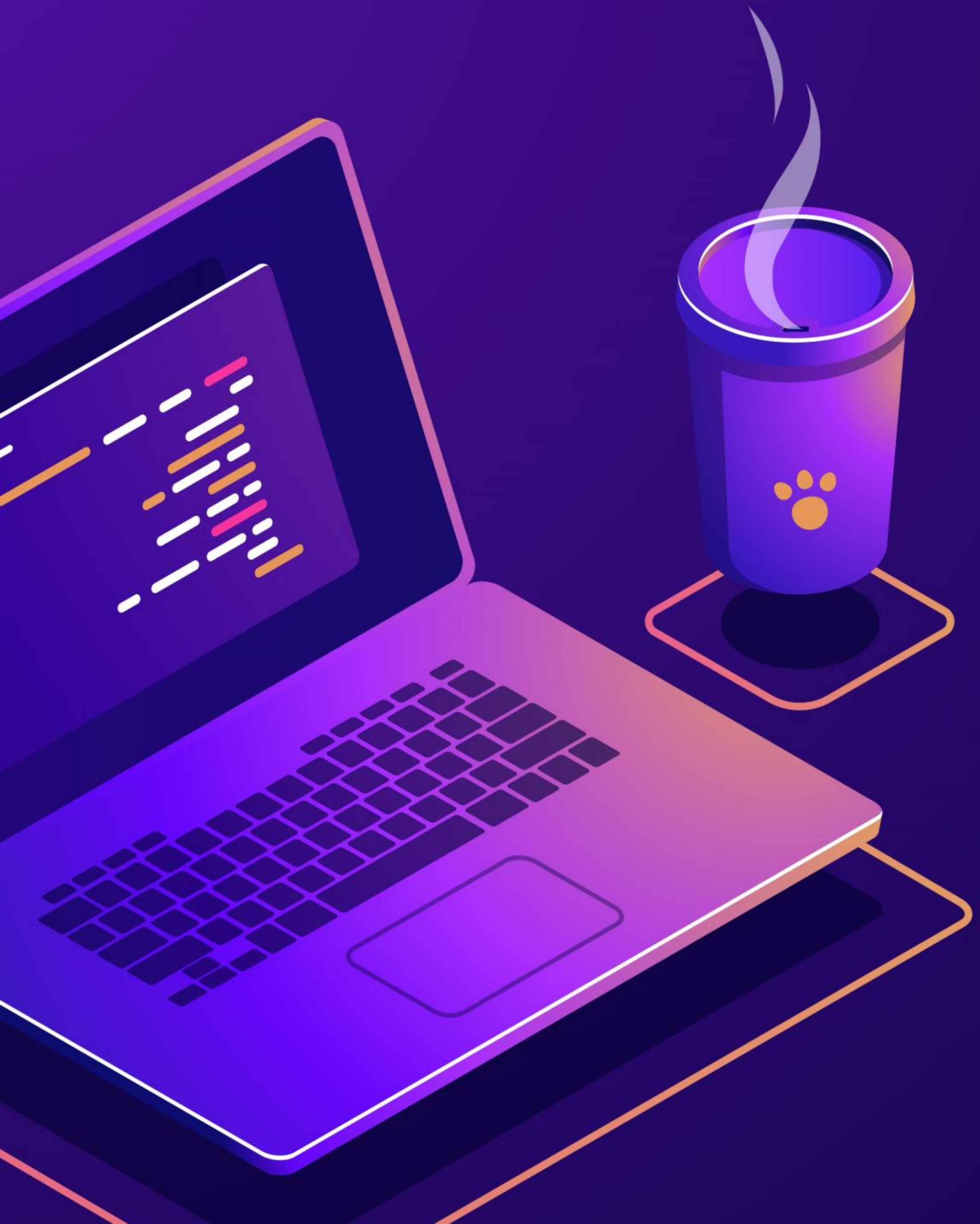


Daynamic Programming

Submitted by

Osama Al-Rashed
Hussam Hajjar
Mohannad Kf Alghazal

What is Daynamic Programming?

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

Why Daynamic Programming

Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again.

Optimal Substructure

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

DP are following TwoO Different ways

Memoization (Top Down)

The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions.

Tabulation (Bottom Up)

The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table.

What is Fibonacci

Fibonacci equation

fib (n) = fib (n-1) + fib (n-2)

where

fb (0) = 0

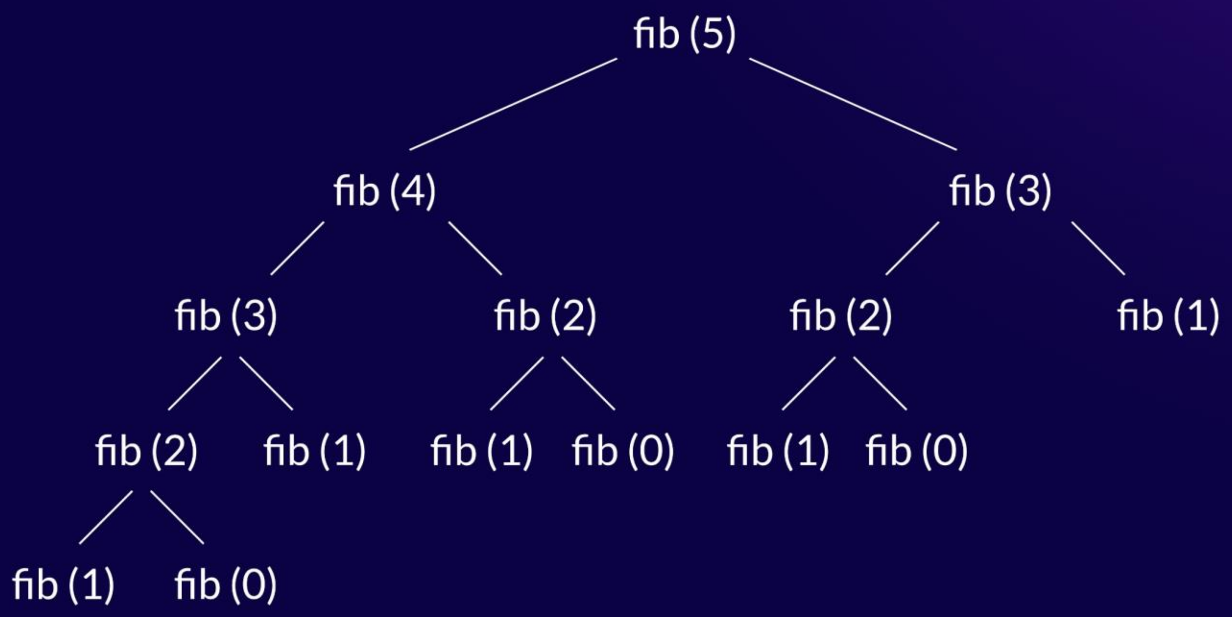
fib (1) = 1

Applying Fibonacci to

1- recursive

2 - Memoization (Top Down)

3 - Tabulation (Bottom Up)



Ex: Fibonacci (5)

Fibonacci Applying to Recursive

```
private int RecursionFib(int n)
{
    if (n <= 1)
        return n;

    return RecursionFib(n - 1) + RecursionFib(n - 2);
}
```

Fibonacci Applying to Memoization

```
private int MemoizationFib(int n)
    {
        if (lookup[n] == NIL)
        {
            if (n <= 1)
                lookup[n] = n;
            else
                lookup[n] = MemoizationFib(n - 1) + MemoizationFib(n - 2);
        }
        return lookup[n];
    }
```

Fibonacci Applying to Tabulation

```
private int TabulationFib(int n)
        {
            int[] f = new int[n + 1];
            f[0] = 0;
            f[1] = 1;
            for (int i = 2; i <= n; i++)
                f[i] = f[i - 1] + f[i - 2];
            return f[n];
        }
```

Time comparison between the three methods

N	Recursion	Memoization	Tabulation
10	456.83 ns	194.58 ns	**30.95 ns**
15	5,274.98 ns	250.98 ns	43.92 ns
20	57,433.44 ns	**299.68 ns**	56.30 ns
25	496,001.50 ns	**272.85 ns**	69.25 ns
30	6,941,941.89 ns	**346.08 ns**	65.11 ns

Comparison between the two types of DP

	Tabulation	Memoization
State	State transtion relation is difficulte to think	State transtion relation is easy to think.
Code	Code gets complicated when lot of condition are required	Code is easy and less complicated.
Speed	Fast, as we directly access previous states from the table.	Slow due to lot of recursive calls and return statements.
Subproblem	if all subproblems must be solved at least once, a bottom-up usually outperforms a top- down by constant factor	if all subproblems space need not be solved at all, the memoized solution has the advaantage of solving only that definitely...
Table Entries	Starting form the first enty, all entries are filled one by one	Unlike the tabulate version, all enties of the lookup table are not naecessarily filled...

What is Knapsack

Knapsack Applying to Recursive

```
private int KnapSackBruteForce(int W, int[] wt, int[] val, int n)
        {
            if (n == 0 || W == 0)
                return 0;

            if (wt[n - 1] > W)
                return KnapSackBruteForce(W, wt, val, n - 1);

            else
                return Math.Max(val[n - 1] + KnapSackBruteForce(W - wt[n - 1], wt, val, n - 1),
                                KnapSackBruteForce(W, wt, val, n - 1));
        }
```

Knapsack Applying to Memoization

```
private int KnapSackMemoization(int W, int[] wt, int[] val, int N)
        {

            int[,] dp = new int[N + 1, W + 1];

            for (int i = 0; i < N + 1; i++)
                for (int j = 0; j < W + 1; j++)
                    dp[i, j] = -1;

            return KnapSackRec(W, wt, val, N, dp);
        }
```

```
private int KnapSackRec(int W, int[] wt, int[] val,int n, int[,] dp)
{
    if (n == 0 || W == 0)
        return 0;
    if (dp[n, W] != -1)
        return dp[n, W];
    if (wt[n - 1] > W)
        return dp[n, W] = KnapSackRec(W, wt, val, n - 1, dp);

    else
        return dp[n, W] = Math.Max((val[n - 1] + KnapSackRec(W - wt[n - 1], wt, val,
                                   n,- 1, dp)) , KnapSackRec(W, wt, val, n - 1, dp));
}
```

Knapsack Applying to Tabulation

```
private int KnapSackTabulation(int W, int[] wt, int[] val, int n)
        {
            int i, w;
            int[,] K = new int[n + 1, W + 1];


            for (i = 0; i <= n; i++)
            {
                for (w = 0; w <= W; w++)
                {
                    if (i == 0 || w == 0)
                        K[i, w] = 0;

                    else if (wt[i - 1] <= w)
                        K[i, w] = Math.Max(val[i - 1] + K[i - 1, w - wt[i - 1]],K[i - 1, w]);

                    else
                        K[i, w] = K[i - 1, w];
                }
            }

            return K[n, W];
        }
```

Time comparison between the three methods

N	Recursion	Memoization	Tabulation
30	18, 757.45 ...	17.76 ...	**32.56 ...**

Thank you!