# Bansilal Ramnath Agarwal Charitable Trust’s

Vishwakarma Institute of Technology, Pune-37

*(Autonomous Institute of Savitribai Phule Pune University)*



|  |  |
| --- | --- |
| **Division** | C |
| **Batch** | B1 |
| **GR-no** | 12210343 |
| **Roll no** | 07 |
| **Name** | Vedant Rane |

Assignment 6: Using dynamic programming, find the nth Fibonacci number and kth binomial coefficient

# Title:

To calculate the nth Fibonacci number and the kth binomial coefficient using the following algorithms:

1. Brute force approach
2. Dynamic Programming (2D array approach)
3. Dynamic Programming (1D array approach)

# Theory:

Fibonacci Number: The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. Mathematically, it is defined as:

𝐹(0) = 0, 𝐹(1) = 1,

𝑎𝑛𝑑 𝐹(𝑛) = 𝐹(𝑛 − 1) + 𝐹(𝑛 − 2) 𝑓𝑜𝑟 𝑛 ≥ 2

The problem involves finding the nth Fibonacci number using different methods, ranging from recursive brute force to more optimized dynamic programming approaches.

Binomial Coefficient: The binomial coefficient C(n,k) represents the number of ways to choose k items from n without regard to order. It can be defined using the following relation:

𝐶(𝑛, 𝑘) = 𝑘! /(𝑛 − 𝑘)! 𝑛!

The recursive formula can be expressed as:

𝐶(𝑛, 𝑘) = 𝐶(𝑛 − 1, 𝑘 − 1) + 𝐶(𝑛 − 1, 𝑘)

The task is to compute C(n,k) using multiple techniques, comparing their efficiency.

# Algorithm and pseudocode

## nth Fibonacci number

* 1. **Naive approach**
     1. Function fibonacci(n):
     2. If n == 0 or n == 1, return n.
     3. Else, return fibonacci(n-1) + fibonacci(n-2)
* Explanation:

The naive method uses a simple recursive function to compute Fibonacci numbers. This is straightforward but leads to a large number of redundant calculations. The algorithm directly implements the definition of the Fibonacci series using recursion. However, it has an exponential time complexity due to repeated calculations of the same values.

## Using DP

* + 1. Function fibonacci(n):
    2. Create an array dp of size n+1. 3. Set dp[0] = 0 and dp[1] = 1.

4. For i from 2 to n:

5. dp[i] = dp[i-1] + dp[i-2]

6. Return dp[n].

* Explanation:

This approach uses an array (dp) to store previously calculated Fibonacci numbers, leading to significant improvements over the naive method. By storing intermediate results, dynamic programming avoids redundant calculations. This approach leverages a table where each cell stores a Fibonacci number.

## kth binomial coefficient

* 1. **Naive approach**
     1. Function binomial(n, k):
     2. If k == 0 or k == n, return 1.
     3. Else, return binomial(n-1, k-1) + binomial(n-1, k).
* Explanation: A recursive solution directly applying the recursive formula of the binomial coefficient. This straightforward recursive solution computes the value but leads to multiple recursive calls with overlapping subproblems.

## DP with 2D array

* + 1. Function binomial(n, k):
    2. Create a 2D array dp of size (n+1) x (k+1).
    3. For i from 0 to n:
    4. For j from 0 to min(i, k):
    5. If j == 0 or j == i, set dp[i][j] = 1.

6. Else, set dp[i][j] = dp[i-1][j-1] + dp[i-1][j].

7. Return dp[n][k].

* Explanation: Uses a 2D array to store previously computed values. This method leverages a table to store intermediate results, reducing the number of recursive calls drastically.

## DP with 1D array

* + 1. Function binomial(n, k):
    2. Create a 1D array dp of size k+1. 3. Set dp[0] = 1.

1. For i from 1 to n:
2. For j from min(i, k) down to 1:

6. dp[j] += dp[j-1].

7. Return dp[k].

* Explanation: Optimizes space usage by only maintaining a 1D array. The space requirement is minimized by updating the array from right to left, ensuring previous values are still accessible when needed.

# Analysis of the algorithm

## nth Fibonacci number

* 1. **Naive approach**
     + Time Complexity: 𝑛

𝑂(2)

* + - Space Complexity: 𝑂(𝑛)

## Using DP

* + - Time Complexity: 𝑂(𝑛)
    - Space Complexity: 𝑂(𝑛)

## kth binomial coefficient

* 1. **Naive approach**
     + Time Complexity: 𝑛

𝑂(2)

* + - Space Complexity: 𝑂(𝑛)

## DP with a 2D array

* + - Time Complexity: 𝑂(𝑛𝑘)
    - Space Complexity: 𝑂(𝑛𝑘)

## DP with a 1D array

* + - Time Complexity: 𝑂(𝑛𝑘)
    - Space Complexity: 𝑂(𝑘)

# Conclusion

Both the Fibonacci number and binomial coefficient problems demonstrate the power of dynamic programming. While brute force approaches lead to inefficient exponential time complexity, the use of dynamic programming reduces the problem to polynomial time by storing intermediate results, thus avoiding redundant calculations. The 1D array techniques further optimize space usage, demonstrating a trade-off between simplicity, time, and space efficiency.