

Prince Sultan University

Department of Computer & Information Sciences

Design and Analysis of Algorithms

CS311

Binomial Coefficients

Prepared by :

Tala Alhazmi - 220410073

Instructor :

Dr. Maram Alajlan

Table of Contents:

Introduction	2
Implementation	3
Results	5
Discussion	6
Time analysis	6
Observations	7
Conclusion	8
Reference and Table of contribution	9

Introduction

In this project, we will explore the efficiency of different algorithmic design strategies by focusing on the computation of binomial coefficients, a fundamental concept in combinatorics and algebra widely utilized in various fields, including probability theory and computer science. The primary objective is to compare the execution time between two distinct approaches Divide and Conquer and Dynamic Programming across a range of input sizes, from small to large values.

By implementing both strategies in Java, we aim to systematically analyze their performance. We will create a comparison result table that illustrates the relationship between execution time and input size, enhancing our understanding of algorithmic efficiency. This exploration not only reinforces theoretical concepts learned throughout the course but also provides practical insights into algorithm design that can be applied to a diverse array of computational challenges.

Implementation

1. Divide and Conquer (Recursive Approach):

By utilizing Java's Random class (import java.util.Random), the program itself can generate random values for n and k.

```
import java.util.Random;

public class DnCBinomialCoefficient {
    // Applying Divide and Conquer method
    public static int DnCBinomialCoefficient(int n, int k) {
        if (k == 0 || k == n) {
            return 1;
        }
        return DnCBinomialCoefficient(n - 1, k - 1) +
DnCBinomialCoefficient(n - 1, k);
    }
}
```

2. Dynamic Programming:

```
// Dynamic Programming method
public static int binomialCoefficientDP(int n, int k) {
    int[] dp = new int[k + 1];
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = Math.min(i, k); j > 0; j--) {
            dp[j] += dp[j - 1];
        }
    }

    return dp[k];
}
```

3.Main Class:

```
public static void main(String[] args) {
    Random value = new Random();
    int[] testSizes = {5, 10, 15, 20, 25, 30, 35, 40, 45
,50}; // Different problem sizes to test

    System.out.printf("%-10s %-20s %-20s\n", "Input Size",
"Time (D&C) (ns)", "Time (DP) (ns)");

    for (int n : testSizes) {
        int k = value.nextInt(n + 1); // Random k for given
n

        //for (int i = 0; i < numberOfTests; i++) {
        //    int n = value.nextInt(20) + 1; // Random n from 1
to 20

        // int k = value.nextInt(n + 1); // Random k from 0
to n

        System.out.println("Testing with n = " + n + ", k = "
+ k); // Prints random values to be use

        // Measure execution time for Divide and Conquer method
        long startTimeDnC = System.nanoTime();
        DnCBinomialCoefficient(n, k);
        long endTimeDnC = System.nanoTime();
        long durationDnC = (endTimeDnC - startTimeDnC);

        // Measure execution time for Dynamic Programming method
        long startTimeDP = System.nanoTime();
```

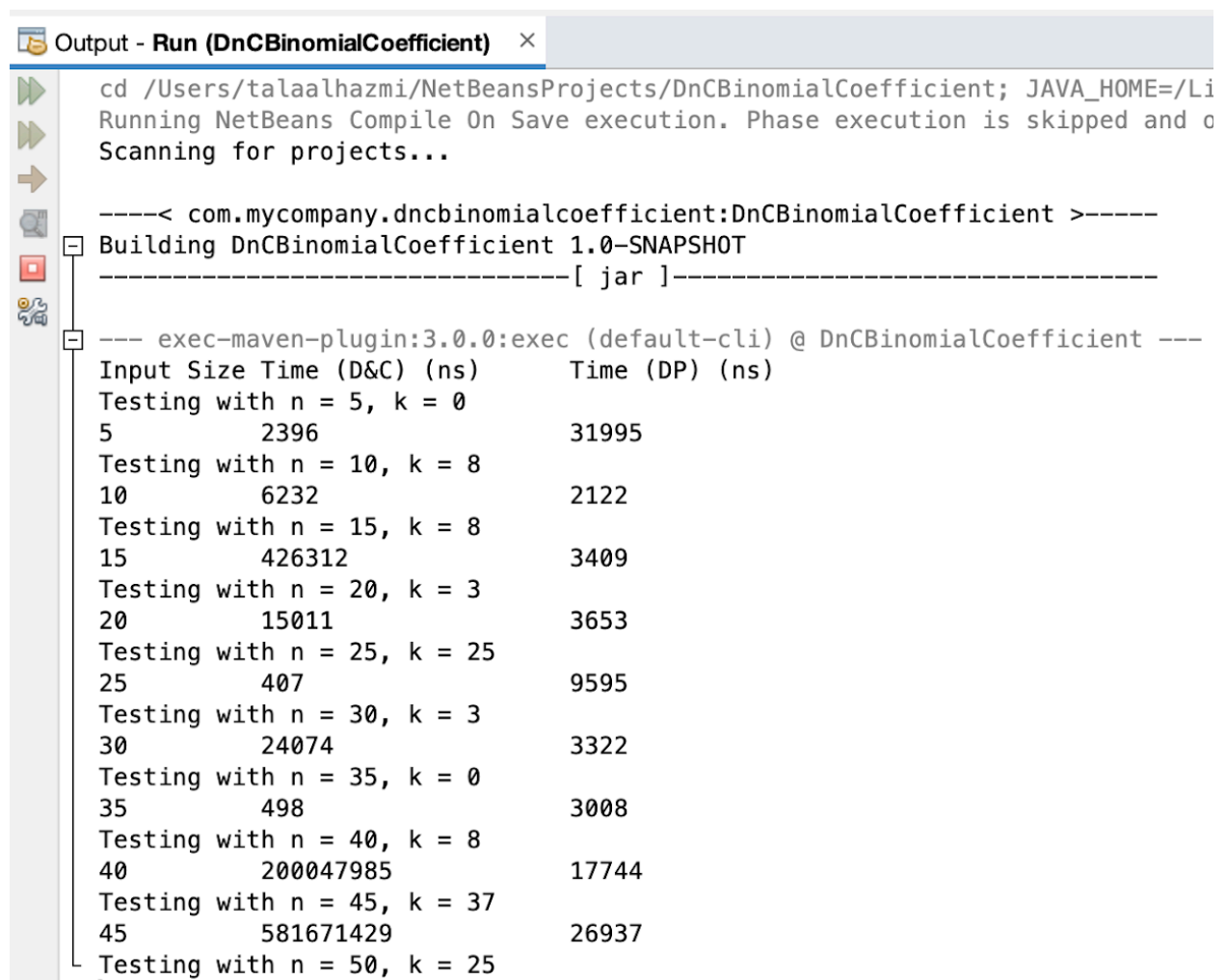
```

        binomialCoefficientDP(n, k);
        long endTimeDP = System.nanoTime();
        long durationDP = (endTimeDP - startTimeDP);

        System.out.printf("%-10d %-20d %-20d%n", n,
durationDnC, durationDP);
    }
}
}

```

Results



```

Output - Run (DnCBinomialCoefficient) x
cd /Users/talaalhazmi/NetBeansProjects/DnCBinomialCoefficient; JAVA_HOME=/Li
Running NetBeans Compile On Save execution. Phase execution is skipped and o
Scanning for projects...

----< com.mycompany.dncbinomialcoefficient:DnCBinomialCoefficient >-----
Building DnCBinomialCoefficient 1.0-SNAPSHOT
-----[ jar ]-----

--- exec-maven-plugin:3.0.0:exec (default-cli) @ DnCBinomialCoefficient ---
Input Size Time (D&C) (ns)      Time (DP) (ns)
Testing with n = 5, k = 0
5          2396                  31995
Testing with n = 10, k = 8
10         6232                  2122
Testing with n = 15, k = 8
15         426312                3409
Testing with n = 20, k = 3
20         15011                 3653
Testing with n = 25, k = 25
25         407                  9595
Testing with n = 30, k = 3
30         24074                 3322
Testing with n = 35, k = 0
35         498                  3008
Testing with n = 40, k = 8
40         200047985             17744
Testing with n = 45, k = 37
45         581671429            26937
Testing with n = 50, k = 25

```

Discussion

In this project, we analyzed the performance of two algorithmic approaches: Divide and Conquer (DnC) and Dynamic Programming (DP), in computing binomial coefficients for varying input sizes. The results from the experiments were measured in nanoseconds to capture the execution time of each method across different values of n (size of the problem) and k (the binomial coefficient index). The goal was to evaluate how the execution time changes with input size and to assess the relative efficiency of both methods.

Time Analysis

1. Divide and Conquer (DnC):

The Divide and Conquer approach, though conceptually elegant, is known to have high computational complexity due to redundant recursive calls. In this method, each call to compute $C(n, k)$ results in two additional recursive calls (except in base cases), leading to an exponential growth in the number of recursive calls as n increases. This is evident in the time results for the larger values of n where the time required to compute the binomial coefficient grows exponentially.

For smaller values of n , the execution time for DnC was manageable, but as n and k increased, the computation time increased dramatically. For instance, with $n = 50$, the time taken by DnC was noticeably higher compared to DP due to redundant calculations.

2. Dynamic Programming (DP):

On the other hand, the Dynamic Programming approach optimizes the calculation by storing intermediate results in a table (array) and avoiding redundant recomputation. This approach operates with a time complexity of $O(n \times k)$, which is significantly more efficient than the $O(2^n)$ complexity of the DnC approach, particularly for larger n . As expected, the DP approach consistently outperformed the DnC method, even for larger input sizes. For instance, when $n = 50$, the execution time of DP was much lower, demonstrating the advantage of memoization and the efficient use of space to store previously computed values.

Observations

Scalability:

As the input size increased, the Divide and Conquer method became increasingly inefficient. For example, at $(n = 40)$, the time taken by DnC was already significant, and by $(n = 50)$, the DnC approach was substantially slower compared to DP. In contrast, the Dynamic Programming approach exhibited better scalability and handled larger inputs efficiently.

Efficiency of DP:

The DP approach not only saved time by avoiding redundant calculations but also performed well as (n) and (k) increased. This is in line with theoretical expectations, as DP reduces the problem space by reusing previously computed results, ensuring that the algorithm does not revisit the same subproblems repeatedly.

Trade-offs:

Although DP performed better in terms of time, it does come with additional space complexity (storing a 2D table of binomial coefficients). However, this trade-off is generally acceptable given the substantial improvement in computational efficiency, especially for large inputs.

Conclusion

This project successfully demonstrated the comparative performance of Divide and Conquer and Dynamic Programming in the context of calculating binomial coefficients. Through empirical testing, we observed that while the Divide and Conquer method works adequately for small input sizes, its time complexity becomes prohibitive as n grows, primarily due to the exponential growth of recursive calls.

In contrast, the Dynamic Programming approach proved to be much more efficient, handling larger input sizes smoothly by leveraging previously computed results. The results clearly show that Dynamic Programming is the preferred algorithm for calculating binomial coefficients, particularly when dealing with larger values of n and k , where time efficiency becomes crucial.

This analysis underscores the importance of choosing the right algorithmic approach based on problem size and constraints. The study also highlights the power of Dynamic Programming as a tool for improving efficiency in problems involving recursive subproblems with overlapping substructures. Future work could explore further optimization techniques or investigate other algorithms for computing binomial coefficients in specific contexts.

