

Session 8

Implementation of Binomial Coefficient Algorithm

This session compares two ways of computing binomial coefficients. The divide-and-conquer solution follows the mathematical recurrence directly, while the dynamic programming solution stores repeated subproblem results and is much faster for larger inputs.

Objectives

✕ Do not copy. Read for understanding and the viva

- Implement binomial coefficient using divide and conquer.
- Implement binomial coefficient using dynamic programming.
- Compare both approaches for small and large values of n and k .

Concept

✕ Do not copy. Read for understanding and the viva

The binomial coefficient $C(n, k)$ counts the number of ways to choose k items from n items.

It also follows Pascal's recurrence:

$$\begin{array}{l} 1 \quad C(n, k) = C(n-1, k-1) + C(n-1, k) \\ 2 \quad C(n, 0) = C(n, n) = 1 \end{array}$$

Divide and Conquer Approach

✕ Do not copy. Read for understanding and the viva

The recursive approach directly follows Pascal's recurrence. It is simple, but it recomputes many subproblems.

Metric	Value
Time complexity	Exponential without memoization
Space complexity	$O(n)$ recursion depth

Dynamic Programming Approach

✖ Do not copy. Read for understanding and the viva

The DP approach stores intermediate values in a table and avoids repeated computation.

Metric	Value
Time complexity	$O(nk)$
Space complexity	$O(nk)$ or $O(k)$ when optimized

Question 1

Problem Statement

■ Write in lab record

Implement a binomial coefficient problem using divide and conquer technique.

Explanation

✖ Do not copy. Read for understanding and the viva

The binomial coefficient, denoted as $\binom{n}{k}$ or $C(n, k)$, represents the number of ways to choose k elements from a set of n elements without regard to order. To implement this using the Divide and Conquer technique, we break the larger problem into smaller subproblems using Pascal's Identity.

To implement this using the Divide and Conquer technique, we break the larger problem into smaller subproblems using Pascal's Identity.

Mathematical Formulation

✕ Do not copy. Read for understanding and the viva

Pascal's Identity states that:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Base Cases:

✕ Do not copy. Read for understanding and the viva

- If $k = 0$, there is exactly 1 way to choose 0 items: $\binom{n}{0} = 1$
- If $k = n$, there is exactly 1 way to choose all n items: $\binom{n}{n} = 1$
- If $k > n$, it's impossible to choose more items than available: $\binom{n}{k} = 0$

By using this recurrence relation, we divide the problem of computing $\binom{n}{k}$ into two smaller subproblems: computing $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$, and then conquer them by adding their results.

Implementation

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

binomial-d-c-algo.py

```

1 # Binomial Coefficient using Divide & Conquer
2 class BinomialDnC:
3     def __init__(self):
4         self.call_count = 0 # Tracks recursive calls for analysis
5
6     def compute(self, n, k):
7         """
8         Divides problem using Pascal's Identity:
9          $C(n,k) = C(n-1, k-1) + C(n-1, k)$ 
10        """
11        self.call_count += 1
12
13        # Base Cases (Conquer step for trivial problems)
14        if k == 0 or k == n:
15            return 1
16        if k > n or k < 0:
17            return 0
18
19        # Divide: Split into two subproblems
20        left = self.compute(n - 1, k - 1)
21        right = self.compute(n - 1, k)
22
23        # Combine: Add results of subproblems
24        return left + right
25
26
27 def main():
28     solver = BinomialDnC()
29     n, k = 5, 2
30
31     print(f"Computing C({n}, {k}) using Divide & Conquer...\n")
32     result = solver.compute(n, k)
33
34     print(f"✅ Result: C({n}, {k}) = {result}")
35     print(f"📊 Total Recursive Calls: {solver.call_count}")
36     print(
37         "📝 Note: Pure D&C has overlapping subproblems. For n=5,k=2, it take
38     )
39     print("    Optimal approach: Add memoization → O(nk) time (Dynamic Progra
40
41
42 if __name__ == "__main__":
43     main()

```

Question 2

Problem Statement

■ Write in lab record

Implement a binomial coefficient problem using dynamic programming technique.

Explanation

× Do not copy. Read for understanding and the viva

To optimize the computation of the binomial coefficient $\binom{n}{k}$, we transition from the Divide and Conquer approach to Dynamic Programming (DP). As noted previously, the recursive formula $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ generates heavily overlapping subproblems. Dynamic programming solves each subproblem exactly once and stores the result in a table, completely eliminating redundant calculations.

The DP Strategy

× Do not copy. Read for understanding and the viva

We can implement this using a Bottom-Up (Tabulation) approach. We construct a 2D table `dp` of size $(n+1) \times (k+1)$, where the entry `dp[i][j]` will store the value of $\binom{i}{j}$.

Mathematical Dependencies:

× Do not copy. Read for understanding and the viva

- **Base Cases:** For any row i , `dp[i][0] = 1` (choosing 0 items) and `dp[i][i] = 1` (choosing all items).
- **Transitions:** For all other entries, `dp[i][j] = dp[i-1][j-1] + dp[i-1][j]`.

This structure directly mirrors how Pascal's Triangle is constructed row by row.

Implementation

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

`binomial-d-p-algo.py`

```
1 # Binomial Coefficient using Dynamic Programming (Bottom-Up Tabulation)
2
3
4 def binomial_dp(n, k, print_table=False):
5     """
6     Computes C(n,k) using DP tabulation.
7     Fills a 2D table iteratively to avoid recomputation.
8     """
9     # Handle invalid inputs
10    if k < 0 or k > n:
11        return 0
12    if k == 0 or k == n:
13        return 1
14
15    # DP table initialization: (n+1) rows, (k+1) columns
16    dp = [[0] * (k + 1) for _ in range(n + 1)]
17
18    # Fill table using Pascal's recurrence
19    for i in range(n + 1):
20        # Base case: C(i, 0) = 1
21        dp[i][0] = 1
22        for j in range(1, min(i, k) + 1):
23            # DP Transition: C(i,j) = C(i-1,j-1) + C(i-1,j)
24            dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
25
26    # Optional: Print DP table for lab verification
27    if print_table:
28        print(f"\n{'=' * 40}")
29        print(f"DP TABLE CONSTRUCTION (n={n}, k={k})")
30        print(f"{'=' * 40}")
31        print(f"{'i\j':<4}", end="")
32        for j in range(k + 1):
33            print(f"j={j:<3}", end="")
34        print()
35        print("-" * 40)
36        for i in range(n + 1):
37            print(f"i={i:<2}", end="")
38            for j in range(k + 1):
39                print(f"{dp[i][j]:<6}", end="")
40            print()
41        print(f"{'=' * 40}\n")
42
43    return dp[n][k]
```

```
44
45
46 def main():
47     n, k = 5, 2
48     print(f"Computing C({n}, {k}) using Dynamic Programming...\n")
49     result = binomial_dp(n, k, print_table=True)
50     print(f"Result: C({n}, {k}) = {result}")
51     print("DP Advantage: Each subproblem computed exactly once.")
52     print("    Time Complexity: O(n×k) | Space Complexity: O(n×k)")
53
54
55 if __name__ == "__main__":
56     main()
```

Question 3

Problem Statement

■ Write in lab record

Study the performance of both implementations using five problem instances in terms of efficiency for large and small values of n and k .

Answer

■ Write in lab record

To evaluate the empirical efficiency of the Divide and Conquer (Recursive) and Dynamic Programming (Tabulation) implementations, we analyze their performance metrics across five distinct problem instances. These instances are systematically chosen to reflect combinations of small and large values for n and k .

Experimental Setup & Test Cases

✗ Do not copy. Read for understanding and the viva

We evaluate the algorithms using the following five instances:

- Instance 1 (Small n , Small k): $\binom{5}{2}$ — Baseline verification.

- Instance 2 (Medium n , Small k): $\binom{25}{3}$ — Evaluates performance when k remains small but n grows.
- Instance 3 (Medium n , Balanced k): $\binom{26}{13}$ — Represents the worst-case scenario for a given n because the binomial coefficient peaks at $k = \lfloor n/2 \rfloor$.
- Instance 4 (Large n , Boundary k): $\binom{100}{1}$ — Evaluates behavior near the edge boundaries.
- Instance 5 (Large n , Large k): $\binom{100}{50}$ — The absolute worst-case threshold for evaluating large scaling structures.

Performance Breakdown by Scenario

✗ Do not copy. Read for understanding and the viva

Scenario A: Small n and Small k (Instance 1)

- **Divide & Conquer:** Performs acceptably well. With a tiny search space, the recursion tree is shallow ($n = 5$), and the redundant overlapping calculations are negligible to modern CPUs.
- **Dynamic Programming:** Allocates a small tracking grid and fills it linearly. Both implementations execute in microseconds.

Scenario B: Medium n and Balanced k (Instance 3)

- **Divide & Conquer:** Performance degrades exponentially. For $\binom{26}{13}$, the recursion tree branches out into over 20 million function calls to compute a final answer of just 10.4 million. The CPU wastes immense time re-evaluating the same sub-problems over and over.
- **Dynamic Programming:** Highly Efficient. The DP loop fills a small, predictable table. It computes the solution in exactly 260 additions, bypassing millions of redundant operations completely.

Scenario C: Large n and Boundary k (Instance 4)

- **Divide & Conquer:** Performs efficiently only because of the early exit condition ($k = 1$). The execution paths short-circuit quickly back up the call stack, limiting the total recursive operations to 199.
- **Dynamic Programming:** Keeps pace uniformly at 101 table updates.

Scenario D: Large n and Large k (Instance 5)

- **Divide & Conquer:** Total System Failure. The total operation count reaches $\approx 2 \times 10^{29}$. If a modern computer could process one trillion operations per second, it would still take over 6

billion years to complete this single computation. The program will crash due to a stack overflow or freeze indefinitely.

- **Dynamic Programming: Flawless.** Even with an extremely massive resulting number, the tabulation method completes the task in exactly 3,825 matrix step updates, rendering an instantaneous output.

Conclusion

✖ Do not copy. Read for understanding and the viva

- **Divide and Conquer Evaluation:** This approach is structurally unsuited for large entries. Its time complexity is fundamentally tied to the size of the final output ($2 \times \binom{n}{k} - 1$). As the solution value grows exponentially, the execution time mirrors that explosive growth. It is only practical for academic visualization or instances where $n \leq 20$.
- **Dynamic Programming Evaluation:** This approach remains completely immune to structural variations in the output value size. Its time complexity scales strictly based on the matrix area bounds ($\mathcal{O}(n \times k)$). DP converts an unmanageable exponential time problem into a predictable, highly efficient polynomial time calculation.

[✎ Edit this page](#)

< Previous
Session 7

Next >
Session 9