# CIS7 Unit 5 Lab: Binomial Coefficient

## Iteration- Binomial Coefficient

The **iteration** repeatedly execute the set of instructions using a loop. The loop repeatedly executes until the controlling condition becomes false.

A **binomial coefficient** C(n, r) can be defined as the coefficient of Xr in the expansion of (1 + X)n.

A **binomial coefficient C(n, r)** also gives the number of ways, disregarding order, that r objects can be chosen from among n objects; more formally, the number of r-element subsets (or r-combinations) of an n-element set.

Given two numbers n and r, find value of nCr

**Example A:**

Input : n = 5, r = 2

Output : 30

The value of 5C2 is 10 = (5!)/(2!\*(5-2)!)

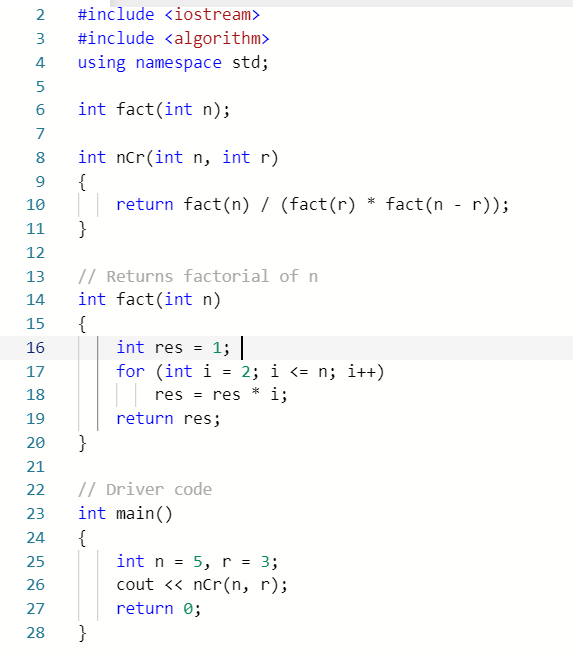
**Example B:**

Input : n = 3, r = 1

Output : 3

**nCr = (n!) / (r! \* (n-r)!)**

**Example 1 Program:** Binomial Coefficient program using function for standard formula.



1. Input Example 1 program into IDE, run the program and examine the result.
2. In Example 1, what is the value of n? What is the value of r? Verify if the output is correct.

N = 5 and R = 3 output should be equal to 5! / (3! \* 2!) = 120/ 12 = 10

1. Given C(7, 4), calculate the output for nCr. Hint: See formula on page 1.

Output = 7! / (4! \* 3!) = 5040 / 144 = 35

1. Use Example 1 program as a template, write C ++ program to display the result of nCr when n = 7, r = 4. Provide screen capture of code and output.

A screenshot of a computer

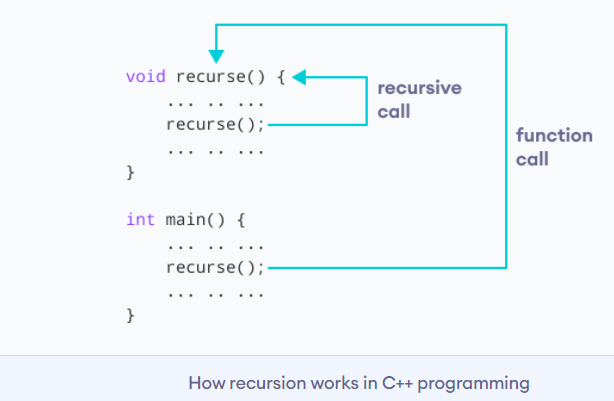
Description automatically generated

1. Explain how the formula **nCr = (n!) / (r!) \* (n-r)!)** is implemented in the program. Identify the type of functions and explain the tasks performed in these functions.

There is one helper function called fact which performs the factorial operation using iteration and returns and integer which is the result of the factorial. Then there is the function we are calling in main called nCr which contains the actual equation for calculating the number nCr. N and R are passed into the nCr function and the nCr function calls the fact function with N, R, and N-R as according to the formula. An integer value is returned which is then printed using cout and this value is our output.

## Recursion - Binomial Coefficient

The function which calls the same function, is known as **recursive function**.



The primary difference between **recursion** and **iteration** is that **recursion is a process**, always applied to a function. **Iteration** is applied to the set of instructions, which get repeatedly executed.

* Recursion uses selection structure.
* Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition (base case) and Infinite recursion can crash the system.
* Recursion terminates when a base case is recognized.
* Recursion is usually slower than iteration due to the overhead of maintaining the stack.
* Recursion uses more memory than iteration.
* Recursion makes the code smaller.
* Recursion is the better choice when the code is clearer, more concise, and more intuitive

**Example 2:** The value of C(n, r) can be recursively calculated using following standard formula for Binomial Coefficients.

**C(n, r) = C(n-1, r-1) + C(n-1, r)**

**C(5, 3) = C(4, 2) + C(4, 3)**

**C(5, 3) = (4!/(2! \* 2!)) + (4!/(3! \* 1!)) = 6 + 4 = 10**

**Base cases: C(n, 0) = C(n, n) = 1**



1. Input Example 2 program into IDE. Run the program and examine the result.
2. In example 2 program, what is the value of n? What is the value of r? Verify if the result is correct using **nCr = (n!) / (r!) \* (n-r)!).**

N = 5 and R = 3 Output = 5! / (3! \* 2!) = 120 / 12 = 10

1. Given C(8, 4), use the formula **nCr = (n!) / (r!) \* (n-r)!)** to calculate the result for **nCr.**

Output = 8! / (4! \* 4!) = 40320 / 576 = 70

1. Use example 2 as a template, write a C++ program using recursion for C(8, 4). Provide screen capture and output.

A screenshot of a computer

Description automatically generated

1. In comparison to Example 1 program, which program delivers higher efficiency? Explain your answer.

Example 1 program is more efficient because example 2 program is using recursion which is generally worse than iteration due to the need for storing the additional stack frames for each recursive call. Typically, we only use recursion for inherently recursive problems such as navigating through a graph or searching a binary tree or for sorting arrays of values. Furthermore, since we are making two function calls in this recursive function this function will exponentially take up more memory as N and R increase whereas iteration doesn’t need to store any additional values other than just the result making it far more efficient.

## Space-Time Complexity - Binomial Coefficient

While analyzing an algorithm, we mostly consider time complexity and space complexity.

* **Time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
* **Space complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and space complexity depends on lots of things, hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Suppose you are given an array A and an X integer and you have to find if exists in array. Simple solution to this problem is traverse the whole array A and check each element.

Each of the operation in computer take approximately constant time. Let each operation takes time. The number of lines of code executed is actually depends on the value of X . During analyses of algorithm, mostly we will consider **worst case scenario**, i.e., when is not present in the array A. In the worst case, the if condition will run N times where N is the length of the array A.

* **Order of growth** is how the time of execution depends on the length of the input.

In the above example, we can clearly see that the time of execution is **linearly depends** on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

* To denote asymptotic upper bound, we use **O-notation, “big-O”.**
  + While analyzing an algorithm, we mostly consider -notation because it will give us an upper limit of the execution time i.e. the execution time in the worst case.

int count = 0;

for (int i = 0; i < N; i++)

for (int j = 0; j < i; j++)

count++;

How many times **count++** will run?

When i=0, it will run 0 times.  
When i=1, it will run 1 times.  
When i=2, it will run 2 times and so on.

Total number of times **count++** will run is 0+1+2+...+(N−1)=N∗(N−1)/2. So the time complexity will be O(N2) There is usually a trade-off between optimal memory use and runtime performance.

**In general for an algorithm, space efficiency and time efficiency reach at two opposite ends and each point in between them has a certain time and space efficiency. So, if there is more time efficiency, then there is less space efficiency, vice versa.**

**Example 3:** The value of C(n, r) can be calculated in O(r) time and O(1) extra space**.** A loop has to be run from 0 to r. So, the time complexity is O(r). O(1), no extra space is required.

**C(n, r) = n! / (n-r)! \* r!**

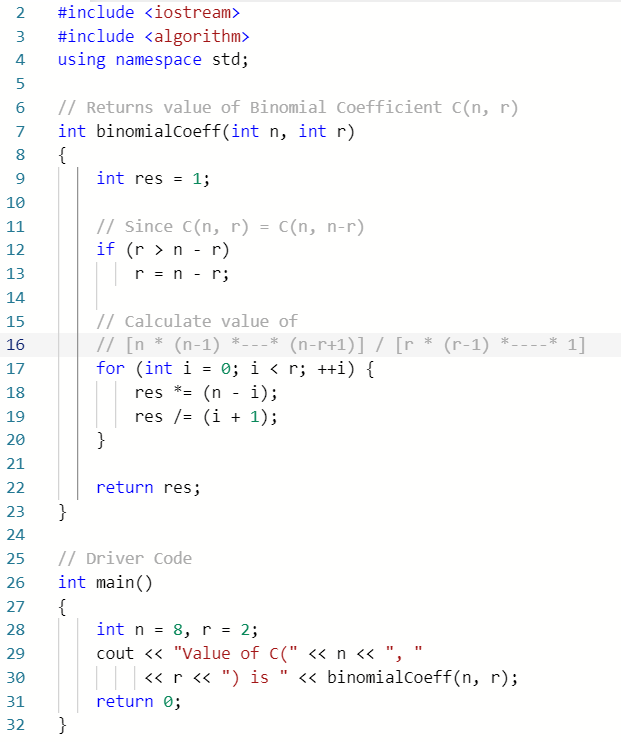
**= [n \* (n-1) \*....\* 1] / [ ( (n-r) \* (n-r-1) \* .... \* 1) \* ( r \* (r-1) \* .... \* 1 ) ]**

**After simplifying, we get**

**C(n, r) = [n \* (n-1) \* .... \* (n-r+1)] / [r \* (r-1) \* .... \* 1]**

**Also, C(n, r) = C(n, n-r)**

**r can be changed to n-r if r > n-r**



1. Input Example 3 program into IDE, run the program and evaluate the result.
2. In example 3, what is the value of n? What is the value of r? Verify if the result is correct using **nCr = (n!) / (r!) \* (n-r)!).**

N = 8 and R = 2 Output = 8! / (2! \* 6! ) = 40320 / 1440 = 28

1. Given C(9, 4), use **C(n, r) = n! / ((n-r)! \* r!)** to calculate the result.

Output = 9! / (4! \* 5!) = 362880 / 2880 = 126

1. Use Example 3 program as a template, write a C++ program for C(9, 4). Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. What is the length of input in this program? What is time of execution based on such input?

T = r, O(r)

1. Comparing Example 1 to Example 3, which program is more efficient in performance? Explain your answer.

Example 3 is more efficient in performance because we are only iterating R times rather than N times + R times + N-R times. We have a O(r) rather than O(N + R + N-R). Both are still linear, but O(r) is more efficient.

## Dynamic Programming - Binomial Coefficient

**Dynamic programming** is used where we have problems that can be divided into similar sub-problems, so their results can be re-used. These algorithms are used for **optimization**. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply **store the results of subproblems, so that we do not have to re-compute them when needed later**. This simple optimization reduces time complexities from exponential to polynomial. The intuition behind dynamic programming is that we **trade space for time**, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

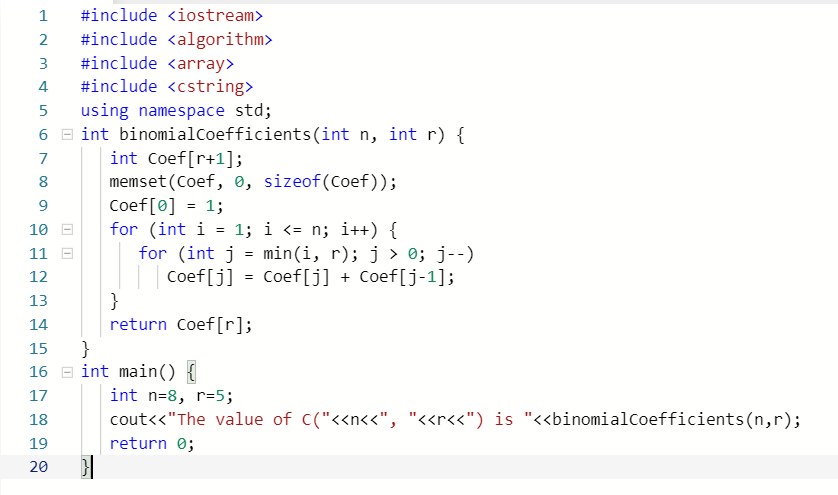
* An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
* Dynamic algorithms use **Memoization**, an optimization technique used primarily to speed up computer programs by storing the results of expensive [function calls](https://en.wikipedia.org/wiki/Subroutine) and returning the cached result when the same inputs occur again.

In **<cstring>, memset** sets the first number of bytes in the block of memory, pointed by ptr to the specified value.

***void \* memset ( void \* ptr, int value, size\_t num );***

* ptr: pointer to the block of memory to fill.
* value: value to be set. The value is passed as an int, but the function fills the block of memory using the unsigned char conversion of this value.
* num: number of bytes to be set to the value.
* size\_t is an unsigned integral type

**Example 4:**



1. Input the program into IDE, then run the program and evaluate the result.
2. In Example 4, what is the value of n? What is the value of r? Verify if the result is correct using **nCr = (n!) / (r!) \* (n-r)!).**

N = 8 and R = 5 Output = 8! / (5! \* 3!) = 40320 / 720 = 56

1. Given C(8,6), use **C(n, r) = n! / ((n-r)! \* r!)** to calculate the result.

Output = 8! / (6! \* 2!) = 40320 / 1440 = 28

1. Use Example 4 program as a template, write a C++ program for C(8, 6). Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Explain how dynamic programming is implemented to optimize this program. Refer to the specific area of the program.

Dynamic Programming is implemented using the Coef array on line 13 where values are being stored into an array each time rather than computing the values multiple times.

1. In comparison to Example 2 program, recursion, explain why this program is more efficient.

This program is more efficient compared to Example 2 because we are using an array to dynamically store the sub result values we have already computed rather than computing the same sub result multiple times leading to the program ultimately being more efficient and running faster.