**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: 13/02/2023**

**Experiment No. 7**

# Title: Exponentiation, Mod

**Theory/Description of the Problem Statement:**

**Exponential:**

The divide and conquer algorithm for calculating the exponential of an integer involves recursively breaking down the problem into smaller subproblems, solving them, and combining the results. In this algorithm, if the exponent is 0 or 1, the answer is returned directly. If the exponent is negative, the reciprocal of the result for a positive exponent is returned. If the exponent is even, the result is calculated recursively by squaring the base raised to half the exponent. If the exponent is odd, the result is calculated recursively by multiplying the base with the square of the base raised to half the exponent (rounded down). The time complexity of this algorithm is O(log n), where n is the exponent, due to the binary tree of recursive calls with a height of log n.

**Modulo:**

This C++ program calculates the modular exponentiation of a given base raised to a given exponent with respect to a given modulus using the binary exponentiation algorithm. The program defines a function called modExp that takes in three arguments: base, exponent, and modulus, and returns the result of (base^exponent) % modulus. The program then reads in the base and exponent from standard input, sets the modulus to a large prime number, and calls the modExp function to calculate the modular exponentiation.

The modExp function works by repeatedly squaring the base and reducing the exponent by half until the exponent is 0. At each iteration, if the current bit of the exponent is 1, the current result is multiplied by the base modulo the modulus. This effectively calculates the modular exponentiation by avoiding the computation of very large intermediate results and reducing the answer modulo the given modulus after each multiplication. The program then outputs the result of the modular exponentiation to standard output.

**Algorithm:**

**Algorithm : Exponential**

* Function pownew(int x, unsigned int n)
* if (n == 0)
* return 1;
* int y = pownew(x, n/2);
* if (n % 2 == 0) {
* return y \* y;
* Else
* return x \* y \* y;

**Algorithm : MOD**

* function mod(base, exponent, modulus)
* if exponent = 0 then
* return 1 mod modulus
* else if exponent = 1 then
* return base mod modulus
* else if exponent mod 2 = 0 then
* temp <- mod(base, exponent / 2, modulus)
* return (temp \* temp) mod modulus
* else
* temp <- mod(base, (exponent - 1) / 2, modulus)
* return (base \* temp \* temp) mod modulus
* end if
* end function

**Analysis of the Algorithm**

**Exponential**

Time Complexity: O(log n)

Auxiliary Space: O(log n), for recursive call stack

**MOD**

Time complexity : O(log exponent)

as each recursive call halves the exponent. At each step of the recursion, the algorithm checks the value of the exponent to determine which branch to follow.

**Experiment and result:**

**Exponentioa**

Code:

#include <iostream>

using namespace std;

int pow(int x, unsigned int n)

{

    int pow = 1;

    for (int i = 0; i < n; i++) {

        pow = pow \* x;

    }

    return pow;

}

int pownew(int x, unsigned int n)

{

    if (n == 0) {

        return 1;

    }

    int y = pownew(x, n/2);

    if (n % 2 == 0) {

        return y \* y;

    } else {

        return x \* y \* y;

    }

}

int main()

{

    cout << "pow is " << pow(2, 3) << endl;

    cout << "pownew is " << pownew(2, 3) << endl;

    return 0;

}

**MOD:**

#include <bits/stdc++.h>

using namespace std;

long long modExp(long long base, long long exponent, long long modulus) {

 int result = 1;

 base = base % modulus;

 while (exponent > 0) {

 if (exponent & 1) {

 result = (result \* base) % modulus;

 }

 exponent = exponent >> 1;

 base = (base \* base) % modulus;

 }

 return result;

}

int main() {

 int mod = (int)1e9+7;

 int base, exponent;

 cin >> base >> exponent;

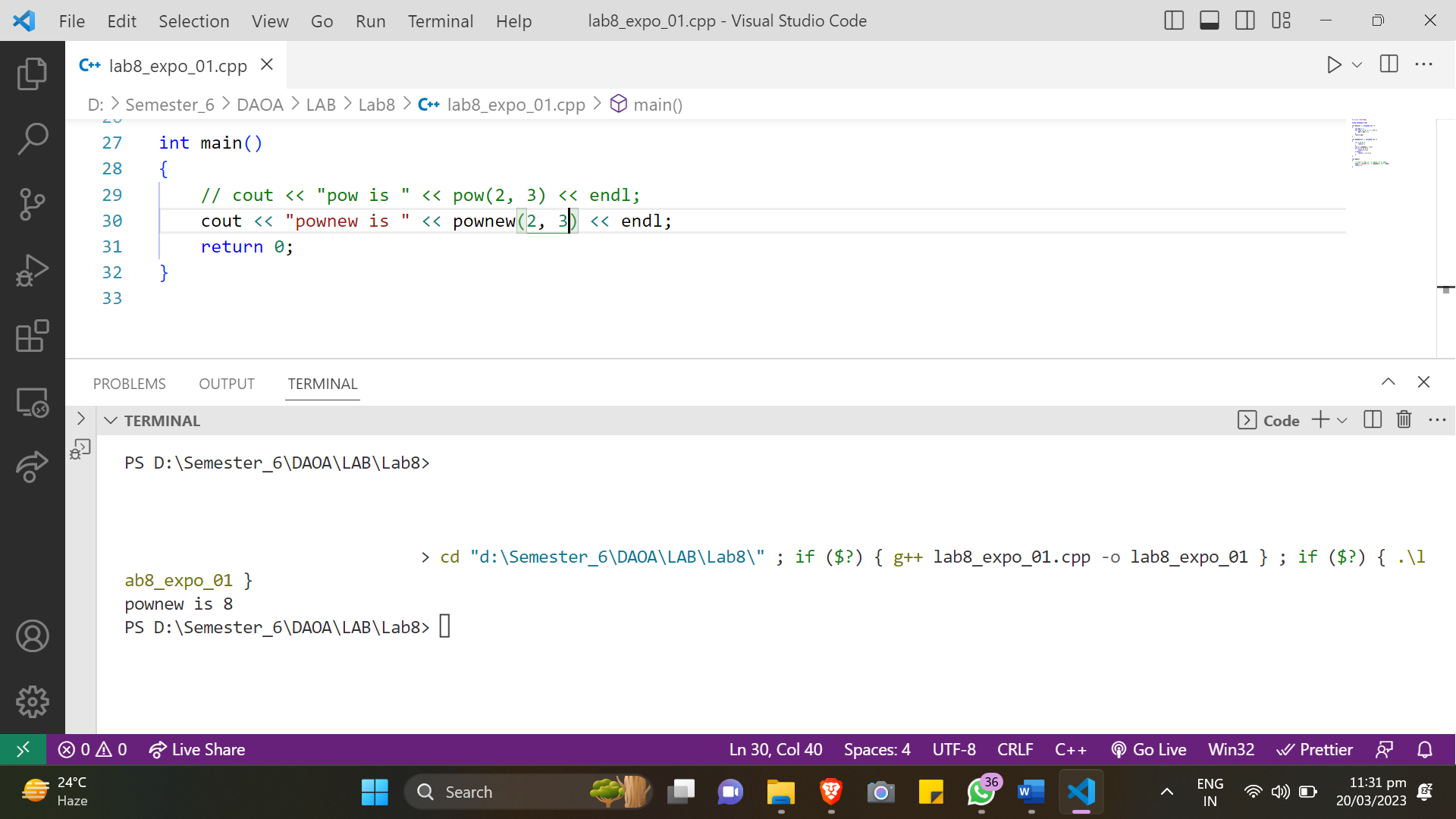
 cout << modExp(base, exponent, mod);

 return 0;

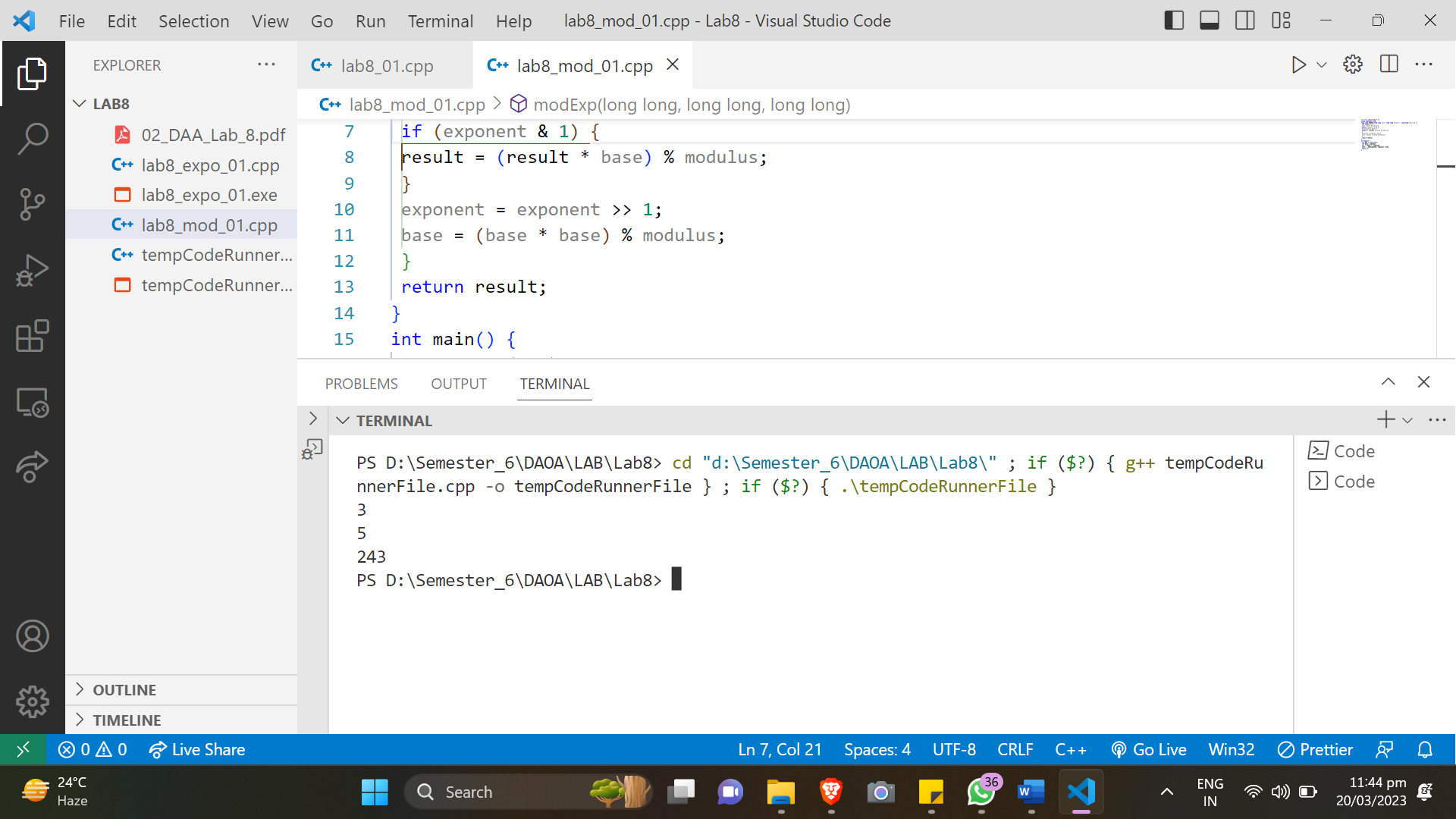
}

Output:

**Exponential**



**MOD:**



**Conclusions:**

Therefore, modular exponentiation and modular arithmetic algorithms are fundamental operations in many areas of computer science, particularly in cryptography as well as number theory. An algorithm most commonly used for modular exponentiation is binary exponentiation (x+y modulo p) in O(log y) time. In many cryptographic applications, it is a popular choice because of its efficiency and wide applicability.