Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology, Pune-37

(An Autonomous Institute Affiliated to Savitribai Pune University)

Department of Electronics & Telecommunication Engineering

ET3272: Design and Analysis of Algorithm

Name of the student: Aditya Shirude **Roll No.: 03**

Div: TY - A **Batch: B1**

Title: Implementation and Analysis of Exponentiation and Modulus.

Theory:

Exponentiation

Exponentiation is the mathematical operation of raising a number (the base) to a given power (the exponent). The result of exponentiation is the base multiplied by itself a number of times equal to the exponent. For example, 2^3 means "2 raised to the power of 3," which is equal to 2 multiplied by itself 3 times: 2 × 2 × 2 = 8. Exponentiation can be represented by the following formula: a^n = a × a × ... × a (n times) where a is the base and n is the exponent.

Modulus

In computer science, the modulus operation is commonly used in a variety of applications, including cryptography, hashing, and data compression. One particularly useful application of the modulus operation is in modular arithmetic, where the operations of addition, subtraction, multiplication, and exponentiation are performed modulo a given number.

Algorithm/Pseudocode:

Exponentiation

```
FUNCTION pow(base, exponent)
    result = 1
    FOR i = 1 TO exponent DO
        result = result * base
    END FOR
    RETURN result
END FUNCTION
```

Modulus

```
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 Algorithm:

Exponentiation

In terms of time complexity, the brute force exponentiation algorithm has a time complexity of O(n), where n is the exponent. This is because it requires n multiplication operations to compute the exponent. For example, to compute 3^4 using brute force, four multiplication operations are needed.

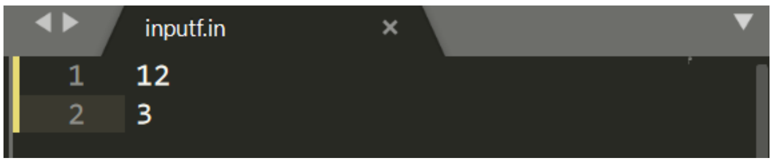
Modulus

It has a time complexity of O(log exponent) because 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. If the exponent is zero or one, it returns 1 modulo modulus or base modulo modulus, respectively, both of which take constant time. If the exponent is even, the algorithm recursively computes the value of base^(exponent/2) modulo modulus and squares the result. This takes one recursive call and one multiplication operation, both of which take constant time.

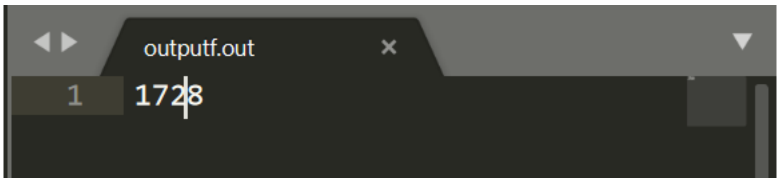
If the exponent is odd, the algorithm recursively computes the value of base^((exponent-1)/2) modulo modulus and multiplies it by base modulo modulus. This takes one recursive call and two multiplication operations, both of which take constant time.

Since each recursive call halves, the exponent, the maximum depth of the recursion is log exponent. Therefore, the time complexity of this algorithm is O(log exponent).

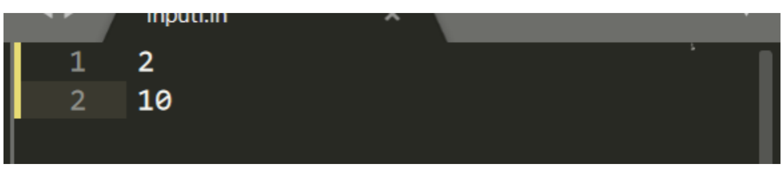
Results:



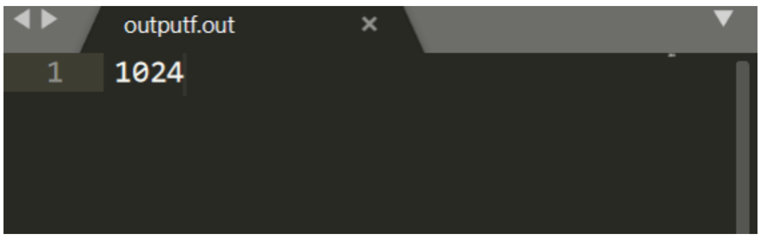
```
12
3
```



```
1728
```



```
2
10
```



```
1024
```

Conclusion:

In conclusion, modular exponentiation and modular arithmetic algorithms are fundamental operations in many areas of computer science, particularly in cryptography and number theory. The binary exponentiation algorithm, which computes x^y modulo p in O(log y) time, is one of the most commonly used algorithms for modular exponentiation. It is efficient and widely applicable, making it a popular choice for many cryptographic applications.

Code For Modulus:

```
#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;
}
```

Code For Exponentiation:

```
int pow(int base, int exponent) {
    int result = 1;
    for (int i = 0; i < exponent; i++) {
        result *= base;
    }
    return result;
}
```