Math

Solve any problem to achieve a rank
View Leaderboard

Topics: Basic Number Theory-1

Basic Number Theory-1

TUTORIAL PROBLEMS

Introduction

This article discusses topics that are frequently used to solve programming problems based on math. It includes the following topics:

1. Modular arithmetic
2. Modular exponentiation
3. Greatest Common Divisor (GCD)
4. Extended Euclidean algorithm
5. Modular multiplicative inverse

1. Modular arithmetic

When one number is divided by another, the modulo operation finds the remainder. It is denoted by the $\%$ symbol.

Example

Assume that you have two numbers 5 and 2. $5\%2$ is 1 because when 5 is divided by 2, the remainder is 1.

Properties

1. $(a + b)\%c = (a\%c + b\%c)\%c$
2. $(a * b)\%c = ((a\%c) * (b\%c))\%c$
3. $(a - b)\%c = ((a\%c) - (b\%c) + c)\%c$
4. $(a/b)\%c = ((a\%c) * (b^{-1}\%c))\%c$

Note: In the last property, b^{-1} is the multiplicative modulo inverse of b and c.

Examples

If $a = 5, b = 3$, and $c = 2$, then:

?

- $(5+3)\%2 = 8\%2 = 0$
  Similarly, $(5\%2 + 3\%2)\%2 = (1+1)\%2 = 0$
- $(5*3)\%2 = 15\%2 = 1$
  Similarly, $((5\%2)*(3\%2))\%2 = (1*1)\%2 = 1$

If $a = 12, b = 15$, and $c = 4$, then the answer in some languages is $(12-15)\%4 = (12\%4 - 15\%4)\%4 = (0-3)\%4 = -3$. However, the answer of the $\%$ operator cannot be negative.

Therefore, to make the answer positive, add c to the formula and compute it as follows: $(12-15)\%4 = (12\%4 - 15\%4 + 4)\%4 = (0-3+4)\%4 = 1$

When are these properties used?

Assume that a = 10^{18}, b = 10^{18}, and c = 10^9+7. You have to find $(a*b)\%c$.

When you multiply a with b, the answer is 10^{36}, which does not conform with the standard integer data types. Therefore, to avoid this we used the properties.

$(a*b)\%c = ((a\%c)*(b\%c))\%c = (49*49)\%(10^9+7) = 2401$

2. Modular exponentiation
Exponentiation is a mathematical operation that is expressed as x^n and computed as $x^n = x \cdot x \cdot \ldots \cdot x$ (n times).

Basic method

While calculating x^n, the most basic solution is broken down into $x \cdot x^{n-1}$. The new problem is x^{n-1}, which is similar to the original problem. Therefore, like in original problem, it is further broken down to $x \cdot x \cdot x^{n-2}$.

This is a recursive way of determining the answer to x^n. However, sometimes an equation cannot be broken down any further as in the case of $n = 0$. A C++ code for this solution, considering $n \geq 0$ is as follows:

```
int recursivePower(int x,int n)
{
    if(n==0)
        return 1;
    return x*recursivePower(x,n-1);
}
```

The recursive method aligns with the explanation, however, the solution can also be written in an iterative format, which is quite ad hoc. A variable 'result', to which x is multiplied for n number of times, is maintained.

The iterative code is as follows:

```
int iterativePower(int x,int n)
{
    int result=1;
    while(n>0)
    {
         result=result*x;
         n--;
    }
    return result;
}
```

Time complexity

With respect to time complexity, it is a fairly efficient *O(n)* solution. However, when it comes to finding x^n, where n can be as large as 10^{18}, this solution will not be suitable.

Optimized method

While calculating x^n, the basis of Binary Exponentiation relies on whether n is odd or even.

If n is even, then x^n can be broken down to $(x^2)^{n/2}$. Programmatically, finding x^2 is a one-step process. However, the problem is to find $(x^2)^{n/2}$.

Notice how the computation steps were reduced from n to $n/2$ in just **one** step? You can continue to divide the power by 2 as long as it is even.

When n is odd, try and convert it into an even value. x^n can be written as $x \cdot x^{n-1}$. This ensures that $n-1$ is even.

- If n is even, replace x^n by $(x^2)^{n/2}$.
- If n is odd, replace x^n by $x \cdot x^{n-1}$. $n-1$ becomes even and you can apply the relevant formula.

Example

You are required to compute 3^{10}. The steps are as follows:

1. The power of 3 is 10, which is even. Break it down as follows:
$$3^{10} \Rightarrow (3^2)^5 \Rightarrow 9^5$$
2. Find 9^5. The power of 9 is 5, which is odd. Convert it into an even power and then apply the following formula:
$9^5 \Rightarrow 9 \cdot 9^4 \Rightarrow 9 \cdot (9^2)^2 \Rightarrow 9 \cdot (81^2)$
3. 81^2 is a one-step computation process

The result is $9 \cdot 81 \cdot 81 = 59049$.

This is an efficient method and the *ten-step process* of determining 3^{10} is reduced to a *three-step process*. At every step, n is divided by 2. Therefore, the time complexity is *O(log N)*.

The code for the process is as follows:

```
int binaryExponentiation(int x,int n)
{
     if(n==0)
          return 1;
     else if(n%2 == 0)          //n is even
          return binaryExponentiation(x*x,n/2);
     else                                 //n is odd
```

```
        return x*binaryExponentiation(x*x,(n-1)/2);
}
```

An iterative version of this method is as follows:

```
int binaryExponentiation(int x,int n)
{
    int result=1;
    while(n>0)
    {
        if(n % 2 ==1)
            result=result * x;
        x=x*x;
        n=n/2;
    }
    return result;
}
```

However, storing answers that are too large for their respective datatypes is an issue with this method. In some languages the answer will exceed the range of the datatype while in other languages it will timeout due to large number multiplications. In such instances, you must use modulus (%). Instead of finding x^n, you must find (x^n) % m.

For example, run the implementation of the method to find 2^{10^9}. The $O(n)$ solution will timeout, while the $O(logN)$ solution will run in time but it will produce garbage values.

To fix this you must use the modulo operation i.e. % M in those lines where a temporary answer is computed.

```
int modularExponentiation(int x,int n,int M)
{
    if(n==0)
        return 1;
    else if(n%2 == 0)        //n is even
        return modularExponentiation((x*x)%M,n/2,M);
    else                                 //n is odd
        return (x*modularExponentiation((x*x)%M,(n-1)/2,M))%M;

}
```

Similarly, the iterative binary exponentiation method can be modified as follows:

```
int modularExponentiation(int x,int n,int M)
{
    int result=1;
    while(n>0)
    {
        if(power % 2 ==1)
            result=(result * x)%M;
        x=(x*x)%M;
        n=n/2;
    }
    return result;
}
```

Recursive solution analysis

- Time complexity: O(log N)
- Memory complexity: O(log N) because a function call consumes memory and log N recursive function calls are made

Iterative solution analysis

- Time complexity: O(log N)
- Memory complexity: O(1)

3. Greatest Common Divisor (GCD)

The GCD of two or more numbers is the largest positive number that divides all the numbers that are considered. For example, the GCD of 6 and 10 is 2 because it is the largest positive number that can divide both 6 and 10.

Naive approach

Traverse all the numbers from min(A, B) to 1 and check whether the current number divides both A and B. If yes, it is the GCD of A and B.

```
int GCD(int A, int B) {
    int m = min(A, B), gcd;
    for(int i = m; i > 0; --i)
        if(A % i == 0 && B % i == 0) {
            gcd = i;
            return gcd;
        }
}
```

Time Complexity

The time complexity of this function is *O(min(A, B))*.

Euclid's algorithm

The idea behind this algorithm is $GCD(A, B) = GCD(B, A\%B)$. It will recurse until $A\%B = 0$.

```
int GCD(int A, int B) {
    if(B==0)
        return A;
    else
        return GCD(B, A % B);
}
```

Example

If a = 16 and B = 10, then the GCD is computed as follows:

- GCD(16, 10) = GCD(10, 16 % 10) = GCD(10, 6)
- GCD(10, 6) = GCD(6, 10 % 6) = GCD(6, 4)
- GCD(6, 4) = GCD(4, 6 % 4) = GCD(4, 2)
- GCD(4, 2) = GCD(2, 4 % 2) = GCD(2, 0)

Since B = 0, $GCD(2,0)$ will return 2.

Time complexity
The time complexity is *O(log(max(A, B)))*.

4. Extended Euclidean algorithm

This algorithm is an extended form of Euclid's algorithm. $GCD(A,B)$ has a special property so that it can always be represented in the form of an equation i.e. $Ax + By = GCD(A,B)$.

The coefficients (x and y) of this equation will be used to find the modular multiplicative inverse. The coefficients can be zero, positive or negative in value.

This algorithm takes two inputs as A and B and returns $GCD(A,B)$ and coefficients of the above equation as output.

Example

If A=30 and B=20, then $30*(1)+20*(-1)=10$ where 10 is the GCD of 20 and 30.

Key idea

A.x+B.y=GCD(A,B). ---(1)

You know that $GCD(A,B)=GCD(B,A\%B)$. Therefore, you can write the equation as follows: B.x_1 + (A % B).y_1=GCD(A,B). ---(2)

You can write $A\%B = A - B*\lfloor A/B \rfloor$ where $\lfloor\rfloor$ means floor value .B and substitute it in equation 2. Your equation will be as follows: B.x_1+ (A - $\lfloor A/B \rfloor$.B).y_1=GCD(A,B)

When you solve it further, your equation is as follows: B.(x_1 - $\lfloor A/B \rfloor$.y_1)+A.y_1=GCD(A,B). ---(3)

Comparing coefficients in equations 1 and 3, you get the following:

- x=y_1
- y=x_1 - $\lfloor A/B \rfloor$.y_1

These equations are key in understanding the extended Euclidean algorithm.

In this algorithm, recursive calls are made to $GCD(B,A\%B)$. The values that are returned from recursive calls are x_1 and y_1, which are used to get x and y.

Implementation

```
#include < iostream >

int d, x, y;
void extendedEuclid(int A, int B) {
    if(B == 0) {
        d = A;
```

```
            x = 1;
            y = 0;
        }
        else {
            extendedEuclid(B, A%B);
            int temp = x;
            x = y;
            y = temp - (A/B)*y;
        }
}

int main( ) {
extendedEuclid(16, 10);
cout << ”The GCD of 16 and 10 is ” << d << endl;
cout << ”Coefficients x and y are ”<< x <<  “and  “ << y << endl;
return 0;
}
```

Output

```
The GCD of 16 and 10 is 2.
Coefficients x and y are 2 and -3.
```

Initially, the extended Euclidean algorithm will run as Euclid's algorithm until you determine $GCD(A, B)$ or until B = 0. It will then assign x = 1 and y = 0.

In the current scenario, since B = 0 and $GCD(A, B)$ is A, the equation $Ax + By = GCD(A, B)$ will be changed to $A * 1 + 0 * 0 = A$.

The values of d, x, and y in the process of the extendedEuclid( ) function are as follows:

- $d = 2, x = 1, y = 0$
- $d = 2, x = 0, y = 1 - (4/2) * 0 = 1$
- $d = 2, x = 1, y = 0 - (6/4) * 1 = -1$
- $d = 2, x = -1, y = 1 - (10/6) * -1 = 2$
- $d = 2, x = 2, y = -1 - (16/10) * 2 = -3$

Time complexity

The time complexity of the extended Euclidean algorithm is $O(log(max(A, B)))$.

When is this algorithm used?

This algorithm is used when A and B are co-prime. In such cases, x becomes the multiplicative modulo inverse of A under modulo B, and y becomes the multiplicative modulo inverse of B under modulo A. This has been explained in detail in the ***Modular multiplicative inverse*** section.

5. Modular multiplicative inverse

What is a multiplicative inverse? If $A.B = 1$, you are required to find B such that it satisfies the equation. The solution is simple. The value of B is $1/A$ or A^{-1}. Here, B is the multiplicative inverse of A.

What is modular multiplicative inverse? If you have two numbers A and M, you are required to find B such it that satisfies the following equation:

$(A.B)\%M = 1$

Here B is the modular multiplicative inverse of A under modulo M.

Formally, if you have two integers A and M, B is said to be modular multiplicative inverse of A under modulo M if it satisfies the following equation:

$A.B \equiv 1(modM)$. where B is in the range [1,M-1]

This equation is a formal representation of the equation discussed earlier.

Why is B in the range [1,M-1]?

$(A * B)\%M = (A\%M * B\%M)\%M$

Since we have B%M, the inverse must be in the range [0,M-1]. However, since 0 is invalid, the inverse must be in the range [1,M-1].

Existence of modular multiplicative inverse

An inverse exists only when A and M are coprime i.e. $GCD(A, M) = 1$.

For example, if A=5 and M=12, then $(A * 5)\%M = (5 * 5)\%12 = 1$. Here, 5 is the modular multiplicative inverse of 5 under modulo 12.
Though $(5 * 17)\%12 = 1$, but since 17 > 12, it isn't considered.

Therefore, the answer is 5.

Approach 1 (naive approach)

```
int modInverse(int A,int M)
{
    A=A%M;
    for(int B=1;B<M;B++)
        if((A*B)%M)==1)
            return B;
}
```

Time Complexity

The algorithm mentioned above runs in *O(M).*

Approach 2

A and M are coprime i.e. $Ax + My = 1$. In the extended Euclidean algorithm, x is the modular multiplicative inverse of A under modulo M. Therefore, the answer is x. You can use the extended Euclidean algorithm to find the multiplicative inverse.

For example, if A=5 and M=12, then $GCD(A, B) = 1$. Therefore, the inverse exists.

$5 * (5) + 12 * (-2) = 1$ (which comes from the extended Euclidean algorithm). Therefore, 5 is the inverse of A=5 under modulo 12.

```
int d,x,y;
int modInverse(int A, int M)
{
    extendedEuclid(A,M);
    return (x%M+M)%M;    //x may be negative
}
```

Time complexity

O(log(max(A,M)))

Approach 3 (used only when M is prime)

This approach uses Fermat's Little Theorem.

The theorem specifies the following: $A^{M-1} \equiv 1(mod M)$

By multiplying with A^{-1} both sides,the equation can be rewritten as follows:

$A^{-1} \equiv A^{M-2}(mod M)$

The formula for A^{-1} is in the form of exponents. Therefore, modular exponentiation can be used to determine the result.

For example, if A=5 and M=11 then $A^{M-2}(mod M) = 5^9(mod 11) = 9$ is the inverse of 5 under modulo 11.

```
int modInverse(int A,int M)
{
    return modularExponentiation(A,M-2,M);
}
```

Time complexity *O(log M)*

When is modular inverse used?

Modular inverse is used to solve $(A/B)\%M$ as follows: $(A/B)\%M = (A * B^{-1})\%M$ After you find the inverse, you can solve this equation easily.

Contributed by: Shubham Gupta