Modular Arithmetic

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^(c-2))%c))%c

   Note: For last property refer to fermat's little theorem

Exponentiation

Exponentiation is a mathematical operation that is expressed as x^n and computed as
x^n=x⋅x⋅...⋅x(n times).

O(logN) Solution

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

Primality Testing

A natural number N is said to be a prime number if it can be divided only by 1 and itself. Primality Testing is done to check if a number is a prime or not.

Code to check whether N is prime or not

```
int PrimeTest(int N)
{
    for (int i = 2; i*i <= N; ++i)
    {
        if(N%i == 0)
        {
            return 0;
        }
    }
    return 1;
}
```

Sieve of Eratosthenes:

This is a simple algorithm useful in finding all the prime numbers up to a given number. Time Complexity:O(Nlog(log(N)))

```
void SieveOfEratosthenes(int n)
{
    bool prime[n+1];
    memset(prime, true, sizeof(prime));

    for (int p=2; p*p<=n; p++)
    {
        if (prime[p] == true)
        {
            for (int i=p*p; i<=n; i += p)
                prime[i] = false;
        }
    }
 }
```

Euclid’s GCD Algorithm

The GCD of two or more numbers is the largest positive number that divides all the numbers that are considered.

NOTE:

1. GCD(A,0) = A

2. GCD(0,0) = A

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);
}
```

Fermat's Theorem

If p is a prime number and a is a natural number, then

(1) a^p=a (mod p).

Furthermore, if p does not divide a, then there exists some smallest exponent d such that

(2) a^d-1=0 (mod p)

and d divides p-1. Hence,

(3) a^(p-1)-1=0 (mod p)

So whenever we are asked to calculate anything modulo 1e9+7. The reason they give such a number is that it is prime and the constraints of the problem ensure that the answer is either smaller or not a multiple of 1e9+7.

So suppose we multiply a^(-1) in eqn. 3

We get, a^(p-2)-a^(-1) = 0 (mod p)
Which can be written as : (a^(p-2)) % p = (a^(-1)) % p

For better understanding refer :
https://mathworld.wolfram.com/FermatsLittleTheorem.html

Bitmasking

Bit masking is used to find all the subsets of an array. In this method, we create a one-to-one mapping between all subsets and the natural numbers.

Consider the first eight numbers and their binary representation.

0 - 000
1 - 001
2 - 010
3 - 011
4 - 100
5 - 101
6 - 110
7 - 111

Let’s say the contents of the array is {a,b,c}

a is at position 1
b is at position 2
c is at position 3

Consider a subset ac. In this subset, the element at first position of the array is taken. The element at second position is ignored. Again the element at third position is taken.

Let’s write it as : 101

where 1 at a position indicates the element in the array at that position is taken in the subset. 0 means it is ignored.

So, 110 will indicate :

element at first and second positions are taken, third one is ignored. So the subset it represents is ab.

The array {a,b,c} has three elements. So the number of subsets is 2^3 = 8

Let's write all the subsets in the above mentioned binary notation and let's write the binary numbers in their decimal representation

000 - empty set - 0
001 - c - 1
010 - b - 2
011 - bc - 3
100 - a - 4
101 - ac - 5
110 - ab - 6
111 - abc - 7

See all the subsets are covered and each subset is identified by a natural number (including 0) from the set [0,2^n-1], where n is the size of the array.

Now let's come to implementation part :

We can iterate through all numbers from 0 to 2^n-1 and check which bits are set in the number in its binary form. If a bit is set, we include the element of the array at that position in our present subset.

Now, how to check which bit is set? Consider the number 5 (101 in binary).

101 & 001 = 1 (ANDing with 2^0)
101 & 010 = 0 (ANDing with 2^1)
101 & 100 = 1 (ANDing with 2^2)

So,if ANDing with 2^x gives 1, then (x+1)th bit from right (counting starting at 1) is set.

C/C++, java gives shift operator (<<) (I don't know about other programming languages). 1<<x means 1 is shifted to the left x times. So,

1<<0 = 001 = 2^0
1<<1 = 010 = 2^1
1<<2 = 100 = 2^2, ans so on

So, by ANDing the number with 1<<x, where x can be from 0 to n-1, we can check which bit is set.

So, now that the basics are clear

Please read through this resource: BITMASKS — FOR BEGINNERS

Combinatorics

https://www.topcoder.com/community/competitive-programming/tutorials/basics-of-combinatorics/

https://www.hackerearth.com/practice/math/combinatorics/basics-of-combinatorics/tutorial/