



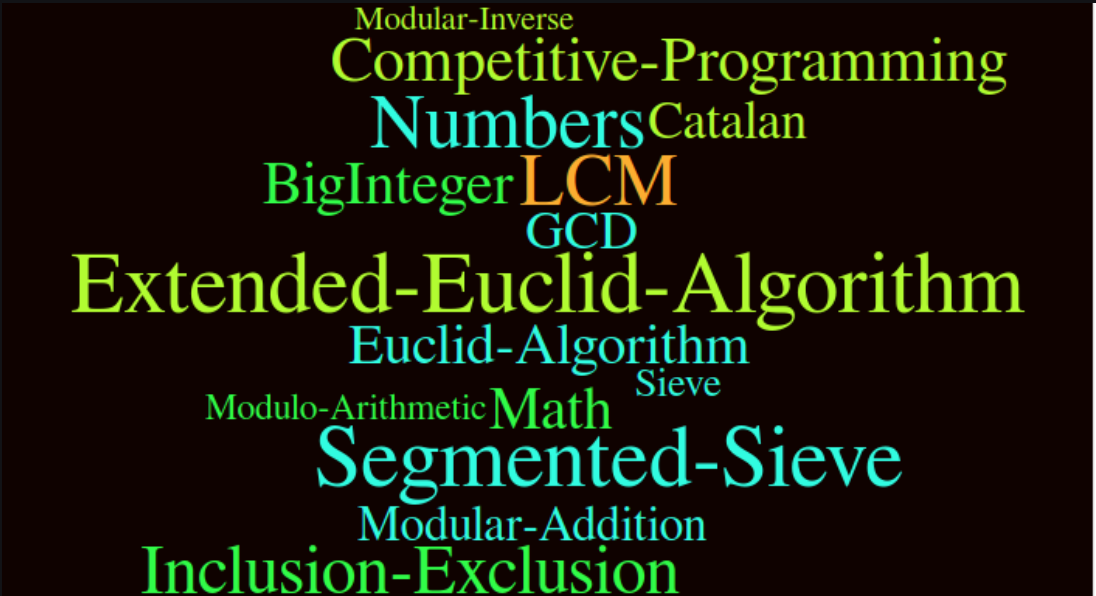
Must do Math for Competitive Programming

Difficulty Level : Medium ● Last Updated : 18 Jun, 2021

Competitive Programming (CP) doesn't typically require to know high-level calculus or some rocket science. But there are some concepts and tricks which are sufficient most of the times. You can definitely start competitive coding without any mathematical background. But maths becomes essential as you dive deep into the world of CP. A majority of the Competitive Coding problems that you'll encounter will have some mathematical logic or trick. All the algorithms that we learn are derived from a mathematical point of view. Most of the times, maths helps us solve the question within the necessary time constraints.

All of the topics can't be covered in a single article but we'll be looking into some of the most common mathematical concepts in competitive coding. Some of these concepts might look too difficult in first sight but applying them on problems will ease them for you.

One more thing is that I have only mentioned about the things you need to cover and some tricks to do that, but you can take help of the other online resources to learn and practice them.



A word cloud on a dark red background listing various mathematical and algorithmic topics. The words are in different colors (yellow, green, orange, blue) and sizes. The most prominent words are 'Competitive-Programming', 'Numbers', 'Extended-Euclid-Algorithm', 'Segmented-Sieve', and 'Inclusion-Exclusion'. Other visible words include 'Modular-Inverse', 'Catalan', 'LCM', 'GCD', 'Euclid-Algorithm', 'Sieve', 'Modulo-Arithmetic', 'Math', 'Modular-Addition', 'BigInteger', and 'Modular-Inverse'.

Modular-Inverse
Competitive-Programming
Numbers
Catalan
BigInteger
LCM
GCD
Extended-Euclid-Algorithm
Euclid-Algorithm
Sieve
Modulo-Arithmetic
Math
Segmented-Sieve
Modular-Addition
Inclusion-Exclusion



1. BigInteger

For e. g. [Calculating factorials of large numbers](#) (lets say 100) or take large numbers of input around 100000 digits in length. In c++, it is not possible to store these numbers even if we use long long int. One way to take this kind of number is, taking them into an array more wisely use vector ... each number will hold an index of array, like if the number is 12345 then $12345 \% 10 = 5$ will in `index[4]` and the number `now = 12345 / 10 = 1234`. now $1234 \% 10 = 4$ will be in `[3]` and so on to $1 \% 10 = 1$ is in `[0]`, or you can use string too, it is more easy since char array only allow 1 byte for each index so you don't need those modulation operation to fit number into index.

Java provides [BigInteger](#) class to handle this.

2. GCD, LCM, Euclidean Algorithm, Extended Euclidean Algorithm

The definitions of GCD and LCM are well-known, (and taught in middle school) I will skip the definitions. Also, since $\text{lcm}(a, b) * \text{gcd}(a, b) = a * b$, calculating GCD is equivalent to calculating LCM.

Now, how do we calculate the GCD of two numbers?

We can of course find the factors of the two numbers and then determine the highest common factor. As the numbers get bigger though (say 155566328819), factorization becomes ineffective.

This is where the Euclid's algorithm comes to our rescue. This algorithm uses the easy-to-prove fact $\text{gcd}(a, b) = \text{gcd}(b, r)$, where r is the remainder when a is divided by b , or just $a \% b$.

C

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

Can we find the numbers (x, y) such that $ux + vy = \text{gcd}(u, v)$? There exists infinitely many pairs – this is Bezout's Lemma. The algorithm to generate such pairs is called Extended Euclidean Algorithm.

3. Sieve of Eratosthenes and Segmented Sieve

Generating primes fast is very important in some problems. Let's cut to the chase and introduce Eratosthenes's Sieve. You can use the Sieve of Eratosthenes to find all the prime numbers that are less than or equal to a given number N or to find out whether a number is a prime number.

The basic idea behind the Sieve of Eratosthenes is that at each iteration one prime number is picked up and all its multiples are eliminated. After the elimination process is complete, all the unmarked numbers that remain are prime. Suppose we want to find all primes between 2 and 50. Iterate from 2 to 50. We start with 2. Since it is not checked, it is a prime number. Now check all numbers that are multiple of except 2. Now we move on, to number 3. It's not checked, so it is a prime number. Now check all numbers that are multiple of 3, except 3. Now move on to 4. We see that this is checked – this is a multiple of 2! So 4 is not a prime. We continue doing this.

C++

```
void sieve(int N)
{
    bool isPrime[N + 1];
    for (int i = 0; i <= N; ++i) {
        isPrime[i] = true;
    }

    isPrime[0] = false;
    isPrime[1] = false;

    for (int i = 2; i * i <= N; ++i) {
        // Mark all the multiples of i as composite numbers
        if (isPrime[i] == true) {
            for (int j = i * i; j <= N; j += i)
                isPrime[j] = false;
        }
    }
}
```

What if the number is large (say 10^{16}), in that case we require segmented sieve.

The idea of segmented sieve is to divide the range $[0..n-1]$ in different segments and compute primes in all segments one by one. This algorithm first uses Simple

Sieve to find primes smaller than or equal to \sqrt{n} . Below are steps used in Segmented Sieve.

1. Use Simple Sieve to find all primes up to square root of 'n' and store these primes in an array "prime[]". Store the found primes in an array 'prime[]'.
2. We need all primes in range $[0..n-1]$. We divide this range in different segments such that size of every segment is at-most \sqrt{n}
3. Do following for every segment $[low..high]$
 - Create an array $mark[high-low+1]$. Here we need only $O(x)$ space where x is number of elements in given range.
 - Iterate through all primes found in step 1. For every prime, mark its multiples in given range $[low..high]$.

In Simple Sieve, we needed $O(n)$ space which may not be feasible for large n . Here we need $O(\sqrt{n})$ space and we process smaller ranges at a time

4. Modulo arithmetic, Modulo exponentiation and Modulo inverse

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 10 and 3. $10\%3$ is 1 because when 10 is divided by 3, 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. $(a/b)\%c = ((a\%c) * (b\%c)^{-1})\%c$

Note: In the last property above, d is the multiplicative modulo inverse of b and c .

When are these properties used?

Assume that $a = 10^{12}$, $b = 10^{12}$, and $c = 10^9 + 7$. You have to find $(a*b)\%c$.

When you multiply a with b , the answer is 10^{24} , which does not conform with the standard integer data types. Therefore, to avoid this we used the properties.

$$(a*b)\%c = ((a\%c) * (b\%c))\%c$$

Fast Modulo exponentiation

Calculate a^b in modular m in $O(\log b)$,

It uses binary expansion of b , and is very straightforward.



C++

```
ll expo(ll a, ll b, ll m)
{
    if (b == 0)
        return 1;
    ll p = expo(a, b / 2, m) % m;
    p = (p * p) % m;

    return (b % 2 == 0) ? p : (a * p) % m;
}
```

Now, let us talk about [modular inverse](#).

By using Extended Euclidean Algorithm, we can get the inverse of a modulo m.

C++

```
// Returns modulo inverse of a with respect
// to m using extended Euclid Algorithm
// Assumption: a and m are coprimes, i.e.,
// gcd(a, m) = 1
int modInverse(int a, int m)
{
    int m0 = m;
    int y = 0, x = 1;

    if (m == 1)
        return 0;

    while (a > 1)
    {
        // q is quotient
        int q = a / m;
```

```

int t = m;

// m is remainder now, process same as
// Euclid's algo
m = a % m, a = t;
t = y;

// Update y and x
y = x - q * y;
x = t;
}

// Make x positive
if (x < 0)
    x += m0;

return x;
}

```

Fermat's Little Theorem gives $a^{(p-1)} \equiv a \pmod{p}$ if $\gcd(a, p) = 1$, where p is a prime. Therefore, we can calculate the modular inverse of a as $a^{(p-2)}$, by fast exponentiation also.

5. Lucas Theorem

We can calculate nCr in modulo p (p is a prime) very fast using Lucas' Theorem. Lucas theorem basically suggests that the value of nCr can be computed by multiplying results of $n(i)Cr(i)$ where $n(i)$ and $r(i)$ are individual same-positioned digits in base p representations of n and r respectively. This is very efficient when p is small and n, r is huge. We can precalculate the factorials and inverse of factorials modulo p by using the above code.

6. Chinese Remainder Theorem

Two numbers (positive integers) a and b are relatively prime (prime to each other), if they have no common prime factors. The numbers m_1, m_2, \dots, m_r , are pair wise relatively prime if any two distinct numbers in that collection, are relatively prime. Chinese remainder theorem says that given any r pair wise relatively prime numbers m_1, m_2, \dots, m_r , and any numbers $b_1, b_2, b_3, \dots, b_r$, we can always find a number M which leaves the remainders $b_1, b_2, b_3, \dots, b_r$ when it is divided by m_1, m_2, \dots, m_r respectively.

Let us solve $x \equiv r \pmod{m_i}$, where m_i are pairwise coprime.

(If they are not coprime, break them into prime powers, and if some are contradictory, there are no solutions.)

7. Series and Sequences



You just need to know some basics like :

- What is a series and does it converge to some value?
- Know about famous series like trigonometric, hyperbolic...etc
- How to calculate the finite limit of famous series like (geometric series, harmonic series)

and basically the same thing for the sequences, you just need to know the basics.

(Trick: use [OEIS](#) site)

We sometimes land up in a situation when various coding problems can be simplified to a mathematical formula but often finding that formula isn't that straightforward .Here comes, OEIS for rescue. We can calculate the terms for initial indices i.e $n=0, 1, 2, 3, \dots$ and then may use OEIS to find the mathematical expression.

8. Catalan Numbers

Catalan numbers are a sequence of natural numbers that helps to solve many counting problem. Terms starting with $n=0$ are : 1, 1, 2, 5, 14, 42, 132, 429, 1430and so on.

Questions based on catalan number may appear in many coding competitions. So it is always a plus point to know in depth about catalan number.

Catalan numbers find extensive applications in forming closed solutions to combinatorics problems. Some of the examples are:

1. The number of binary search trees that can be formed using 'n' nodes is the nth Catalan number.
2. The number of ways that a convex polygon of $n+2$ sides, can be cut into 2 or more triangles by joining any 2 edges is the nth Catalan number.
3. The closed solution to the number of possible parantheses matching given 'n' pairs is the nth Catalan number.

9. Pigeonhole Principle

The pigeonhole principle is a powerful tool used in combinatorial maths. But the idea is simple and can be explained by the following peculiar problem. Imagine that 3 pigeons need to be placed into 2 pigeonholes. Can it be done? The answer is yes, but there is one catch. The catch is that no matter how the pigeons are placed, one of the pigeonholes must contain more than one pigeon.

The logic can be generalized for larger numbers. The pigeonhole principle states that if more than n pigeons are placed into n pigeonholes, some pigeonhole must contain more than one pigeon. While the principle is evident, its implications are



astounding.

For example consider this statement "If you pick five numbers from the integers 1 to 8, then two of them must add up to nine."

Explanation: Every number can be paired with another to sum to nine. In all, there are four such pairs: the numbers 1 and 8, 2 and 7, 3 and 6, and lastly 4 and 5. Each of the five numbers belongs to one of those four pairs. By the pigeonhole principle, two of the numbers must be from the same pair—which by construction sums to 9.

10. Inclusion Exclusion Principle

Inclusion Exclusion principle is a very basic theorem of counting and many problems in various programming contests are based on it, a formal explanation of inclusion exclusion principle goes as follows:

Consider A as a collection of objects and $|A|$ as the number of objects in A and similarly for B, then the cardinality of collection of objects of both sets A and B (when both A and B are disjoint) can be stated as (for 2 finite sets) :

$$|A \cup B| = |A| + |B|$$

But what if the sets are not disjoint?

Then we need to subtract the common objects counted twice while calculating the cardinality of both A and B and new form will become:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

This is the most basic form of inclusion-exclusion principle.

But what if there are more than 2 sets, let's say n sets.

Then it can be stated as :

(Include=add, exclude=subtract)

$|A_1 \cup A_2 \cup A_3 \dots \cup A_n|$ = (Include count of each set, Exclude count of pairwise set, Include count of triplet sets, exclude count of quadruplet sets.....till nth tuple is included(if odd) or excluded(if even))

i. e., $|A_1 \cup A_2 \cup A_3 \dots \cup A_n| = (|A_1| + |A_2| + |A_3| + |A_4| \dots + |A_n|) - (|A_1 \cap A_2| + |A_1 \cap A_3| + |A_1 \cap A_4| \dots + \text{all combinations}) + (|A_1 \cap A_2 \cap A_3| \dots \text{all combinations}) \dots \dots \dots$ and so on.

This list is not exhaustive but the concepts will be very useful in contests in codeforces, codechef etc.. So grab your pen, paper and laptop and start practicing.

Happy coding!

