Competitive Programming I

Math

October3, 2024

Topics

There are several mathematical topics that are important to keep in mind.

- Bit representation
- Base conversion
- Modular arithmetic
- Fast exponentiation
- Sequences
- Primes
- Greatest Common Divisor
- Least Common Multiple

Topics

There are many more relevant topics that we will not discuss, but you are welcome to explore them.

Arithmetic Progression	Geometric Progression	Polynomial
Algebra	Logarithm/Power	Big Integer
Number Theory	Prime Number	Sieve of Eratosthenes
Miller-Rabin	Greatest Common Divisor	Lowest Common Multiple
Factorial	Euler Phi	Modified Sieve
Extended Euclidean	Linear Diophantine	Modular Inverse
Combinatorics	Fibonacci	Golden Ratio
Binet's Formula	Zeckendorf's Theorem	Pisano Period
Binomial Coefficients	Fermat's little theorem	Lucas' Theorem
Catalan Numbers	Inclusion-Exclusion	Probability Theory
Cycle-Finding	Game Theory	Zero-Sum Game
Decision Tree	Perfect Play	Minimax
Nim Game	Sprague-Grundy Theorem	Matrix Power

Table 5.1: List of *some* mathematical terms discussed in this chapter

Taken from: Competitive Programming 4 — Book 2 — Section 5.1

Bitwise operations

For some problems it can be relevant to perform bitwise operations on numbers, C++ offers some common operations out of the box

Code

```
int8_t i = 12; // 00001100
int8_t j =  9; // 00001001
cout << (i & j) << "\n"; // bitwise and -- 00001000
cout << (i | j) << "\n"; // bitwise or  -- 00001101
cout << (i ^ j) << "\n"; // bitwise xor -- 00000101
cout << ~i << "\n";      // bitwise not -- 11110011
```

Result

```
8
13
5
-13
```

Bit representations

The bitwise operations on numbers typically only allow us to perform operation on numbers with 8, 16, 32, 64 bits, and maybe 128 bits if `__int128` (gcc/clang) is supported.

If you need an arbitrary number of bits you can use `std::bitset` which also offers extra operations.

Bit representations

Code

```
bitset<4> b1(13); // 1101
bitset<4> b2("1101");
bitset<4> b3("BBAB", 4, 'A', 'B'); // char*, A=0, B=1
bitset<4> b4(std::string("BBAB"), 0, 4, 'A', 'B'); // A=0, B=1
cout << b1 << "\n";
cout << b2 << "\n";
cout << b3 << "\n";
cout << b4 << "\n";
```

Result

```
1101
1101
1101
1101
```

Bit representations

You can perform the same bitwise operations on bitsets

Code

```
bitset<4> b1(12); // 1100
bitset<4> b2(9);  // 1001
cout << (b1 & b2) << "\n"; // bitwise and -- 1000
cout << (b1 | b2) << "\n"; // bitwise or  -- 1101
cout << (b1 ^ b2) << "\n"; // bitwise xor -- 0101
cout << ~b1 << "\n";       // bitwise not -- 0011
```

Result

```
1000
1101
0101
0011
```

Bit representations

You also have some extra operations

- `b1.all()` – true if all bits are 1
- `b1.none()` – true if all bits are 0
- `b1.test(idx)` – true if the bit at index `idx` is 1
- `b1.set()` – set all bits to 1
- `b1.set(idx)` – set bit at index `idx` to 1
- `b1.reset()` – set all bits to 0
- `b1.reset(idx)` – set bit at index `idx` to 0
- `b1.to_string()` – returns a string of 0s and 1s
- `b1.to_string('A','B')` - returns a string of As (0) and Bs (1)

Base conversion

There are two common operations:

- Convert from a string representing a number in base X ($1 \leq X \leq 36$) to an integer number in base 10
- Convert from an integer number in base 10, to a string representing a number in base X

Usually, we have that $1 \leq X \leq 36$ since 36 is the largest base that can be represented by numbers and english letters. But there are exceptions with different symbols.

Base conversion — from base X to base 10

For the conversion from base X to base 10 we can use `stoll` to convert a string in base $[2, 36]$ to a `long long` in base 10. For bases larger than 10 it is assumed that `a (or A) = 10`, `b (or B) = 11`, and so on.

Code

```
string s = "a4";
cout << stoll(s, NULL, 11) << "\n"; // 10*11^1 + 4*11^0
cout << stoll(s, NULL, 12) << "\n"; // 10*12^1 + 4*12^0
cout << stoll(s, NULL, 16) << "\n"; // 10*16^1 + 4*16^0
```

Result

```
114
124
164
```

Base conversion — from base X to base 10

Otherwise you can perform this conversion manually.

To convert a number N in base X with M digits to base 10 you can do

$$\sum_{i=1}^{M} N_i \cdot X^{M-i}$$

Where N_i is the value of the i-th digit in base 10 (for example A in hexademical notation has value 10 in base 10).

Can you think of a solution to implement this in time $O(M)$?

Base conversion — from base X to base 10

Otherwise you can perform this conversion manually.

To convert a number N in base X with M digits to base 10 you can do

$$\sum_{i=1}^{M} N_i \cdot X^{M-i}$$

Where N_i is the value of the i-th digit in base 10 (for example A in hexademical notation has value 10 in base 10).

Can you think of a solution to implement this in time $O(M)$?

```
F(N, X, M):
  res = 0
  xpow = 1
  for i = M to 1:
    res += N[i] * xpow
    xpow *= X
  return res
```

Base conversion — from base X to base 10

What is the size of the integer (in bits) that you need to store the result of this conversion with respect to the base X and number of digits M?

Base conversion — from base X to base 10

What is the size of the integer (in bits) that you need to store the result of this conversion with respect to the base X and number of digits M?

The maximum number you can get is given by $X^M - 1$, therefore the number of bits you need is $\lceil \log_2(X^M - 1) \rceil$

This will let you know whether you should use 32-bit integer, 64-bit integer, or if you need to use a big integer (in which case it is easier to solve the problem in Python or Java)

Base conversion — from base 10 to base X

For the conversion from base 10 to base *X* we can use `to_chars` on C++17. Before C++17 we could use `iota` but it is not part of the standard and not always available.

Code

```
// for a string representation with maximum of 2 characters we
// need a vector with 3 chars to guarantee one \0 at the end
vector<char> aux(3); // could also be array<char, 3>
// convert to base 2
to_chars(aux.data(), aux.data()+aux.size(), 142, 16);
// save result in string (remove any '\0')
string s(aux.data());
cout << s << "\n";
```

Result

```
8e
```

Base conversion — from base 10 to base X

To perform this conversion manually we need to iteratively obtain the remainder and divide the number with respect to base *X*.

Code

```
int n = 142;
int x = 16;
deque<char> aux;
for (; n != 0; n /= x) {
  int rem = n % x;
  aux.push_front(rem < 10 ? ('0' + rem) : ('a' + rem - 10));
}
string s(aux.begin(), aux.end());
cout << s << "\n";
```

Result

```
8e
```

Base conversion — from base 10 to base X

Does the previous code work for $n = 0$?

Base conversion — from base 10 to base X

Does the previous code work for $n = 0$?

No. If $n = 0$ we need to define `s="0"` manually.

Base conversion — from base 10 to base X

Does the previous code work for $n = 0$?

No. If $n = 0$ we need to define `s="0"` manually.

Does it work for negative numbers?

Base conversion — from base 10 to base X

Does the previous code work for $n = 0$?

No. If $n = 0$ we need to define `s="0"` manually.

Does it work for negative numbers?

No. The result of the modulo operator for negative numbers will be a negative number so we need to be careful. The solution depends on the representation that is required for negative numbers by the problem description. Most often you will want a negative sign followed by the representation of the absolute value.

Base conversion — from base 10 to base X

Given a number N in base 10, how many characters do we need to represent it in base X?

Base conversion — from base 10 to base X

Given a number N in base 10, how many characters do we need to represent it in base X?

Since we consecutively divide the number by X we need $1 + \lfloor \log_X N \rfloor$ characters plus 1 character if we need to represent a negative sign.

Modular arithmetic

Problems where the answer is a very large number, often ask for the answer in modulo M (usually $M = 10^9 + 7$ or another large prime).

For example, given a number N in base 36 with at most 10^6 digits compute its decimal representation. Since the decimal representation can be very large, give the answer in modulo $M = 1000000007$. As an example:

`z8sn189z8x9v8ambydgft712631982ba`

has a decimal representation

`620107051854885676667423185665954717553650239884 06`

which in modulo M (our answer) is

`729915555`

Modular arithmetic

We cannot keep our number in a `long long` as we saw before. So how can we solve this?

One initial idea might be to use Python or Java since they have big integers. However, since the number of digits can be quite large, the operations on big integers can become computationally expensive and can give TLE.

So, how can we do it faster? Take advantage of the fact we want the answer in modulo M and use modular arithmetic. In particular, we can consider the following properties:

- $(a \times b) \mod M = ((a \mod M) \times (b \mod M)) \mod M$
- $(a + b) \mod M = ((a \mod M) + (b \mod M)) \mod M$

Modular arithmetic

Code

```
int64_t M = 1000000007, X = 36;
string s = "z8sn189z8x9v8ambydgft712631982ba";
int64_t xpow = 1;
int64_t ans = 0;
for (auto it = s.rbegin(); it != s.rend(); ++it) {
  int i = (*it <= '9') ? (*it - '0') : (*it - 'a' + 10);
  ans = (ans + ((i * xpow) % M)) % M;
  xpow = (xpow * X) % M;
}
cout << ans << "\n";
```

Result

```
729915555
```

Complexity?

Modular arithmetic

Code

```
int64_t M = 1000000007, X = 36;
string s = "z8sn189z8x9v8ambydgft712631982ba";
int64_t xpow = 1;
int64_t ans = 0;
for (auto it = s.rbegin(); it != s.rend(); ++it) {
  int i = (*it <= '9') ? (*it - '0') : (*it - 'a' + 10);
  ans = (ans + ((i * xpow) % M)) % M;
  xpow = (xpow * X) % M;
}
cout << ans << "\n";
```

Result

```
729915555
```

Complexity? $O(len(s))$

Modular arithmetic — Subtraction and division

Other important modular operations are:

- Subtraction — in this case we need to be careful that the number might become negative. Assuming that $0 \leq a, b < M$, we can do $(a - b + M) \mod M$.
- Division — to solve $(a/b) \mod M$ (only if $gcd(b, M) = 1$) we consider the multiplicate inverse of b

$$(a \cdot b^{-1}) \mod M$$

  If M is a prime number (which it usually is) we can use Fermat's little theorem and transform that equation into

$$((a \mod M) \cdot (b^{M-2} \mod M)) \mod M$$

Note: many problems involving probabilities require the use of fractions in modulo M.

Modular arithmetic — Fast exponentiation

In the previous slide we saw the need to compute b^{M-2} in modulo M where M is potentially a large number. A naive algorithm would take $O(M)$ complexity, which can be too slow when M is large.

Fortunately, if we want the answer in modulo M there is a faster way that is based on a Divide & Conquer technique (we divide the problem to speed up computation). In particular, we know that the following holds

- $b^0 = 1$
- $b^p \mod M = (b^{p/2} \cdot b^{p/2}) \mod M$ if p is even
- $b^p \mod M = (b^{p-1} \cdot b) \mod M$ if p is odd

When p is even we only need to compute $b^{p/2}$ and multiply the result by itself. As such, when p is even we divide the computational power needed in two. When p is odd, we reduce it by 1 to get an even number on the next step. In the worst case p will alternate between odd and even and we halve the value of p every other step. Therefore, we can compute the answer in $O(\log M)$.

Modular arithmetic — Fast exponentiation

A typical recursive approach could be

```
int64_t modPow(int64_t b, int64_t p, int64_t M) {
  if (p == 0) return 1;
  if (p % 2 == 0) {
    int64_t aux = modPow(b, p/2, M);
    return (aux * aux) % M;
  } else {
    return (modPow(b, p-1, M) * b) % M;
  }
}
```

Note: it is a good idea to use 64-bit integers for modulo operations, since usually $M \times M$ only fits in a 64 bit integer.

Note 2: this technique can also be used without the modulo. However, in that case p is usually small (unless the problem requires big integer) so it is not as relevant.

Sequences

There are several sequences that are important to know.

- Fibonacci — classic sequence that can appear as an exercise, good to recognize since it might appear without the name fibonacci attached. It also has some interesting properties (e.g., Zeckendorf's theorem, Pisano Period)
- Binomial coefficients — useful in various problems, especially related to probabilities and counting number of solutions
- Catalan numbers — shows up in many combinatorics problems, e.g., count the number of distint binary trees, count the number of valid parenthesis expressions, and so on

We will not explain them here since they are well documented online. However, you should explore them (and we will have some exercises related to them).

Prime numbers

We all know prime numbers. They have many interesting properties.

One important thing to know is how to compute prime numbers quickly. There is one important algorithm to know:

Sieve of Eratosthenes, which can be used to generate all prime numbers in the range $[0, N]$. It has complexity $O(N \log \log N)$. Up to $N = 10^7$ this is usually acceptable (< 1s).

Prime numbers — Sieve of Eratosthenes

Code

```
int64_t N = 30;
vector<int64_t> primes;
vector<bool> isprime(N+1, true);
isprime[0] = isprime[1] = false;
for (int64_t i = 2; i <= N; ++i) {
  if (isprime[i]) {
    for (int64_t j = i*i; j <= N; j += i) {
      isprime[j] = false;
    }
    primes.push_back(i);
  }
}

for (auto v : primes) cout << v << " ";
cout << "\n";
```

Prime numbers — Sieve of Eratosthenes

Result

```
2  3  5  7  11  13  17  19  23  29
```

Note that after computing this sieve we can also use the `isprime` vector to test whether a number up to N is prime.

If we don't need the `isprime` vector we can speed it up further.

Prime numbers — Testing primality

Another important thing is testing whether a number is prime or not.

- If you need to perform several tests, on numbers up to $N \leq 10^7$ using the sieve we saw to generate the `isprime` vector should be good enough.
- If you only need to check few numbers that may go up to a larger N then you can test all divisors up to $\sqrt{N}$ in $O(\sqrt{N})$.

```
bool is_prime(int64_t N) {
  if (N < 2) return false;
  if (N == 2) return true;
  if (N % 2 == 0 || N % 3 == 0) return false;
  for (int64_t i = 5; i*i <= N; i += 6) {
    if (N % i == 0 || N % (i+2) == 0) return false;
  }
  return true;
}
```

Prime numbers — Testing primality

Another option is compute all primes up to $\sqrt{N}$, and then only test on those

```
bool is_prime(int64_t N, vector<int64_t> const& primes) {
  // Assumes that 'primes' contains all primes up to sqrt(N)
  if (N < 2) return false;
  for (auto prime: primes) {
    if (N % prime == 0) return false;
  }
  return true;
}
```

Prime numbers — Testing primality

However, for very large numbers a better approach is to use probabilistic prime testing. In particular the Miller-Rabin algorithm is a commonly used algorithm that can quickly test primality of numbers up to 10^{18} with nearly 1 probability.

It takes a certainty parameter that trades off computational time with certainity of getting the right answer.

This algorithm is rarely needed, but it is something you can keep in mind that exists. Moreover, Java implements it out of the box on the BigInteger library (but it is possible to get more performance and consequently more certainty with a C++ implementation).

Greatest common divisor

The concept of the greatest common divisor (GCD) corresponds to finding the largest divisor that is common to two or more numbers. Several problems can be solved with the help of the GCD algorithm.

To compute the GCD, we can use the Euclid algorithm that takes $O(\log N)$ time, where $N = \max(a, b)$

Code

```
int64_t gcd(int64_t a, int64_t b) {
  return b == 0 ? a : gcd(b, a%b);
}

int main() {
  cout << gcd(15, 10) << "\n";
  return 0;
}
```

Greatest common divisor

Result

5

Finding the gcd of more than two numbers can be done by consecutive calls.

```
// Finds gcd of (a, b, c, d)
gcd(a, gcd(b, gcd(c, d)));
```

Least common multiple

The least common multiple (LCM) involves computing the the smallest number that is a multiple of two or more numbers.

The LCM can be defined as a function of the GCD

Code

```
int64_t gcd(int64_t a, int64_t b) {
  return b == 0 ? a : gcd(b, a%b);
}

int64_t lcm(int64_t a, int64_t b) {
  return a / gcd(a, b) * b;
}

int main() {
  cout << lcm(15, 10) << "\n";
  return 0;
}
```

Least common multiple

Result

```
30
```

To find the LCM of multiple numbers we can also chain calls

```
// Finds lcm of (a, b, c, d)
lcm(a, lcm(b, lcm(c, d)));
```

Conclusion

There are many more mathematical concepts we did not explore.

To explore other relevant concepts in more detail, you can start by looking at Chapter 5 of the Competitive Programming 4 book.

You will also come into contact with other concepts by participating in contests and checking out the solutions.