

All about CP

Table of Content

1	Development	3
2	FastIO	4
2.1	The Magic Line	4
3	Language	5
3.1	Return by reference	5
4	Maths	6
4.1	Pairwise Sum	6
4.2	Summation	6
4.2.1	Identities	6
5	String	7
5.1	Big Integers	7
6	From A to B	7
7	Bit Manipulation	8
7.1	Non-decimal Literal in C++	8
7.2	How integers are stored	8
7.3	I/O with Non-Decimal Numbers	8
7.4	Signed and Unsigned Integers	8
7.5	Index of Bit	9
7.6	Terminologies	9
7.7	Thinking in Binary	9
7.7.1	Position of bits	9
7.7.2	Converting to Decimal	9
7.7.3	Maximizing/Minimizing bitstrings	10
7.8	NOT	10
7.8.1	-x in terms of ~x	10
7.9	XOR	10
7.9.1	How to Visualize XOR	10
7.10	XOR with AND OR	11
7.11	Thinking of Bitwise Operators: Fixing one operand	11

7.11.1	AND	11
7.11.2	OR	11
7.11.3	XOR	11
7.12	Visualizing n-1	11
7.13	Common Bit Operations and Checks	12
7.13.1	Parity Check	12
7.13.2	Left Shift as $\times 2$	12
7.13.3	Right Shift as $\div 2$	12
7.13.4	Set i-th Bit	12
7.13.5	Clear i-th Bit	12
7.13.6	Flip i-th Bit	12
7.13.7	Check i-th Bit	13
7.13.8	Unset Rightmost Set Bit	13
7.13.9	Check if Power of Two	13
7.13.10	Count Set Bits: Brian Kernighan's Algorithm	13
7.13.11	Set Range of bits	13
7.13.12	Clear Range of bits	14
7.13.13	Flip Range of bits	14
7.14	Builtin Functions	14
7.15	Sum with bit operations	14
8	Ranges	16
8.1	Multiple Ranges	16
8.1.1	Intersections of ranges	16
9	Interactive Problem	17
10	Number Theory	18
10.1	Divisibility of Integer N	18
10.2	Binary Exponentiation	18
10.3	Modular Arithmetic	18
10.3.1	Basic Modular Operations	18
10.3.2	Modular Exponentiation	19
11	Regular Expression	20
11.1	Matching Substring	20

1 Development

This book is publicly developed on [GitHub](#). If you find anything confusing, or you think that there is a better way to express the idea, please make a pull request.

2 FastIO

2.1 The Magic Line

```
1 cin.tie(0)->sync_with_stdio(0);
```

3 Language

3.1 Return by reference

Lets define an array to demonstrate return by reference

```
1 int vals[] = {10, 12, 83, 122, 5, 34};
```

The following function returns the value of the i-th element

```
5 int getVal(int i) {  
6     return vals[i];  
7 }
```

The following function returns a reference to the i-th element

```
2 int& getRef(int i) {  
3     return vals[i];  
4 }
```

Demonstration:

```
8 int x = getVal(2);  
9 x++; // Doesn't change array  
10 cout<<getVal(2)<<"\n"; // Output: 83  
11 int y = getRef(2);  
12 y++; // Changes array  
13 cout<<getVal(2)<<"\n"; // Output: 84
```

Problem: This code causes TLE, but this gets AC

4 Maths

4.1 Pairwise Sum

Given sum of all pair of integers in an array of length $n > 3$, find the original array.

Let the array be $[a, b, c, \dots]$;

$$\begin{array}{r} (a+b) \\ + (b+c) \\ - (c+a) \\ \hline 2b \end{array}$$

4.2 Summation

4.2.1 Identities

$$\sum c \times f(n) = c \times \sum f(n), c \text{ is constant}$$

$$\sum (f(n) \pm g(n)) = \sum f(n) \pm \sum g(n)$$

$$\sum_{i=1}^n \sum_{j=1}^n a_i b_j = (\sum_{i=1}^n a_i) (\sum_{j=1}^n b_j)$$

Proof of this identity is interesting

Problem: [AtCoder ARC A - Simple Math](#)

5 String

5.1 Big Integers

1. Take input as string
2. Reverse the string

6 From A to B

[Digital Root](#)

Think out of the box: Convert from B to A instead

7 Bit Manipulation

7.1 Non-decimal Literal in C++

Base	Prefix
bin	0b
hex	0x
oct	0

```
1 assert(13 == 0b1101);
2 assert(13 == 0xd);
3 assert(13 == 015);
```

7.2 How integers are stored

Integers are stored as blocks of bytes

Data Type	No. of Bytes
char	1
short	2
int	4
long long	8

7.3 I/O with Non-Decimal Numbers

```
1 int x;
2 cin>>hex>>x; // takes input in hex
3 cout<<hex<<x; // prints output in hex
4 cin>>oct>>x; // takes input in oct
5 cout<<oct<<x; // prints output in oct
6 cin>>dec>>x; // takes input in dec
7 cout<<dec<<x; // prints output in dec
8
9 bitset<32> b;
10 cin>>b; // takes 32 bit input in bin
11 cout<<b; // prints 32 bit output in bin
12 cout<<b.to_ulong();
```

7.4 Signed and Unsigned Integers

Positive integers (both signed and unsigned) are just represented with their binary digits, and negative signed numbers (which can be positive and negative) are usually represented with the Two's complement ¹

¹[cp-algorithms - bit manipulation](#)


```

1 cout<<bitset<3>(5)<<"\n";
2 // Output: 101
3 cout<<bitset<32>(-1)<<"\n";
4 // Output: 11111111111111111111111111111111

```

7.5 Index of Bit

Bits in a bit string are indexed from right to left, starting with 0.

```

bit string: 1 0 1 1 0 1
index:      ... 3 2 1 0

```

In this text, i-th bit means bit with index i.

7.6 Terminologies

Terms	Meaning
Set bit	Make the bit 1
Unset/Clear bit	Make the bit 0
Flip bit	Make the bit opposite
Lower bit/ Higher bit	i-th bit is lower than j-th bit if $i < j$ MSB is highest bit, LSB is the lowest bit

7.7 Thinking in Binary

Fun fact: Bit stands for “Binary Digit”

7.7.1 Position of bits

Every position in a binary number has an index as mentioned [here](#).

Each position also has a positional value: 2^{index}

Here is an example:

```

bit string: 0 1 1 0 1
index:      4 3 2 1 0
value:      16 8 4 2 1

```

7.7.2 Converting to Decimal

To find decimal representation, we have to add up the positional values of set bits

As an example, let's convert 0b1101 to decimal:

```

bit string: 1 1 0 1
index:      3 2 1 0
value:      8 4 2 1

```

Therefore, 0b1101 in decimal is -
 $8 \times 1 + 4 \times 1 + 2 \times 0 + 1 \times 1$
 $= 8 + 4 + 1$
 $= 13$

7.7.3 Maximizing/Minimizing bitstrings

Bit string, A is greater than B if: - Length of binary representation of A is greater than that of B (ignoring leading zeros) - If their lengths are equal, then in the first position where they differ, A has 1 and B has 0.

This idea is important to solve problems related to maximizing/minimizing number with binary operations

These problems can be reduced to the following pattern: You are given a number n. Do some bitwise operations on n such that n is maximized.

The idea behind this pattern of problems is that you have to maximize the length of the bitstring (without leading zeros) and set more significant bits that are unset

7.8 NOT

Operation	Meaning
$\sim x$	1's complement of x
$\sim x + 1$	2's complement of x

7.8.1 $\sim x$ in terms of $\sim x + 1$

```
1 assert( $\sim x == \sim x + 1$ );
```

7.9 XOR

7.9.1 How to Visualize XOR

Method-1: Non-equivalence Operator $A \wedge B$ is true if truth value of A and B are different.

The following function uses this algorithm:

```
1 bool xor(bool a, bool b) {
2     if (a!=b)
3         return true;
4     return false;
5 }
```

Method-2: Programmable Inverter Think of XOR as a machine with an on/off button, that takes one bit as input and one bit as output. If the machine is on, the output bit will be inverse of input bit, otherwise, the output bit will be the same as input bit.

In $A \oplus B$, one bit decides if the other should be flipped.

The following function uses this algorithm:

```

1 bool xor(bool a, bool b) {
2     if (a)
3         return !b;
4     return b;
5 }
```

7.10 XOR with AND OR

$$a \oplus b = (a | b) - (a \& b)$$

$$a \oplus b = (a | b) \& \sim(a \& b)$$

7.11 Thinking of Bitwise Operators: Fixing one operand

7.11.1 AND

Operation	Meaning
$\& 1$	same
$\& 0$	0

7.11.2 OR

Operation	Meaning
$ 1$	1
$ 0$	same

7.11.3 XOR

Operation	Meaning
$\wedge 1$	flip
$\wedge 0$	same

7.12 Visualizing n-1

When we subtract 1 from a number, the rightmost set bit becomes unset and all the bits to its right become set.

```
n    = xxxx10000
n-1  = xxxx01111
```

There for value of a binary number with all 1s of length n is $2^n - 1$

7.13 Common Bit Operations and Checks

7.13.1 Parity Check

If n is an integer(positive or negative) then $n\&1$ represent parity of n . It is similar to $n\%2$ but better, because unlike $n\%2$, $n\&1$ works for both positive and negative numbers.

```
1 int n;
2 n=5;
3 assert((n&1) == 1);
4 assert((n%2) == 1);
5 n=-5;
6 assert((n&1) == 1);
7 assert((n%2) == -1);
```

In general, $x \% (1\<\<k)$ is equivalent to $x \& (1\<\<k)-1$

7.13.2 Left Shift as $\times 2$

$x\<\<y$ is equivalent to $x \times 2^y$

```
1 assert((5<<2) == 20);
```

7.13.3 Right Shift as $\div 2$

$x\>\>y$ is equivalent to $\lfloor \frac{x}{2^y} \rfloor$

```
1 assert((10>>2) == 2);
```

7.13.4 Set i-th Bit

$n|(1\<\<i)$

7.13.5 Clear i-th Bit

$n\&\sim(1\<\<i)$

7.13.6 Flip i-th Bit

We already discussed that, XOR works as a programmable inverter.
 $n\wedge(1\<\<i)$

7.13.7 Check i-th Bit

$n \& (1 \ll x)$ $(n \gg x) \& 1$

7.13.8 Unset Rightmost Set Bit

$n \& (n - 1)$

7.13.9 Check if Power of Two

n is power of two, if there is only one set bit. To check if there is only one set bit, unset the last set bit, and check if it becomes zero or greater. If it becomes zero its a power of two

```
1 bool isPowerOf2(int n) {
2     return n!=0 && (n&(n-1))>0;
3 }
```

Corner case: 0

7.13.10 Count Set Bits: Brian Kernighan's Algorithm

The idea is to count how many times we can unset the rightmost set bit, until we reach 0

```
1 int countSetBits(int n) {
2     int cnt=0;
3     while(n)
4         n=n&(n-1), cnt++;
5     return cnt;
6 }
```

7.13.11 Set Range of bits

The concept is similar to setting i-th bit, except we will use a different bit mask.

$1 \ll i$ is i 0s after one 1

$(1 \ll i) - 1$ is i 1s at the end, and rest are 0s

Now we can leftshift these 1s to fit into the range

```
1 int setRange(int n, int start, int stop) {
2     int length = stop-start;
3     int mask = (1<<length);
4     mask = mask-1;
5     mask = mask<<start;
6     return n|mask;
7 }
```

7.13.12 Clear Range of bits

```
1 int clearRange(int n, int start, int stop) {
2     int length = stop-start;
3     int mask = (1<<length);
4     mask = mask-1;
5     mask = mask<<start;
6     mask = ~mask;
7     return n&mask;
8 }
```

7.13.13 Flip Range of bits

```
1 int flipRange(int n, int start, int stop) {
2     int length = stop-start;
3     int mask = (1<<length);
4     mask = mask-1;
5     mask = mask<<start;
6     return n^mask;
7 }
```

7.14 Builtin Functions

The g++ compiler provides the following functions for counting bits:

- `__builtin_clz(x)`: the number of zeros at the beginning of the number
- `__builtin_ctz(x)`: the number of zeros at the end of the number
- `__builtin_popcount(x)`: the number of ones in the number
- `__builtin_parity(x)`: the parity (even or odd) of the number of ones

The functions can be used as follows:

```
1 int x = 5328; // 000000000000000000001010011010000
2 cout << __builtin_clz(x) << "\n"; // 19
3 cout << __builtin_ctz(x) << "\n"; // 4
4 cout << __builtin_popcount(x) << "\n"; // 5
5 cout << __builtin_parity(x) << "\n"; // 1
```

While the above functions only support int numbers, there are also long long versions of the functions available with the suffix ll.

Source: [CSES Book](#)

7.15 Sum with bit operations

$a+b=((a\&b)<<1)+(a\^b)$

Note: Since, $a \oplus b$ can be written in terms of and, or the sum can also be

written in terms of and,or
 $a+b=(a|B)+(a\&b)$

Practice: [CF](#)

8 Ranges

8.1 Multiple Ranges

8.1.1 Intersections of ranges

There are n ranges $[l_1, r_1], [l_2, r_2], [l_3, r_3], \dots [l_n, r_n]$
Now, the intersections of these ranges is $[L, R]$, where
 $L = \max(l_1, l_2, l_3, \dots, l_n)$
 $R = \min(r_1, r_2, r_3, \dots, r_n)$

Edge Case:

If $R < L$, then the intersection is an empty range.

Practice:

- [Problem 1](#)
- [Problem 2](#)
- [Problem 3](#)

9 Interactive Prpblem

Make a function called ask that takes the parameters of the question as parameter and returns the return value of the question

Example:

```
1 int ask(string s, int a, int b) {  
2     cout << s << ' ' << a << ' ' << b << '\n';  
3     int res;  
4     cin >> res;  
5     return res;  
6 }
```

10 Number Theory

10.1 Divisibility of Integer N

N can be very large number(containing more than **40** digits).Then, **N should be read as string**. The following properties can be helpful for those cases.

N is

- 1.Always divisible by 1.
- 2.Divisible by 2 if the last digit of N is divisible by 2 i.e., last digit is even.
- 3.Divisible by 3 if sum of digits is divisible by 3.
- 4.Divisible by 4 if the number containing only the last two digits of N is divisible by 4.
- 5.Divisible by 5 if last digit is 0 or 5.
- 6.Divisible by 6 if it is divisible by both 2 and three i.e.,last digit is even and the sum of all digits is divisible by 3.

10.2 Binary Exponentiation

x^n can be written as follows:

$$x^n = \begin{cases} 1, & n = 0 \\ x^{(n/2)} \cdot x^{(n/2)}, & n \% 2 = 0 \\ x^{(n-1)} \cdot x, & n \% 2 = 1 \end{cases}$$

C++ Code:

```
1 #define ll long long
2 ll pow(ll x,ll n){//x^n
3     if(n==0)return 1;
4     ll z=modpow(x,n/2);
5     z*=z;
6     if (n&1)return x*z;
7     return z;
8 }
```

Complexity: $O(\log n)$

10.3 Modular Arithmetic

10.3.1 Basic Modular Operations

Modular Addition: $(a+b)\%m = ((a\%m)+(b\%m))\%m$

Modular Multiplication $(a \times b)\%m = ((a\%m) \times (b\%m))\%m$

10.3.2 Modular Exponentiation

Calculate $x^n \% m$:

Just we need to use the modular multiplication formula in *Binary Exponentiation*.

$$x^n \bmod m = \begin{cases} 1 \bmod m, & n = 0 \\ ((x^{n/2} \bmod m) \cdot (x^{n/2} \bmod m)) \bmod m, & n \bmod 2 = 0 \\ ((x^{n/2} \bmod m) \cdot (x \bmod m)) \bmod m, & n \bmod 2 = 1 \end{cases}$$

```
1  #define ll long long
2  ll modpow(ll x, ll n, ll m){ //x^n mod m
3      if(n==0) return 1%m;
4      ll z=modpow(x, n/2, m)%m;
5      z=(z*z)%m;
6      if (n&1) return ((x%m)*z)%m;
7      return z;
8  }
```

Complexity: $O(\log n)$

11 Regular Expression

11.1 Matching Substring

Calculate how many times a certain pattern appears in a string(with duplicates).

Problem-1:

The pattern starts and ends with 1, and there are one or more 0s in-between.

C++ code:

```
1 #include<bits/stdc++.h>
2 #include <regex>
3 using namespace std;
4 int countSubstrings(const string &s) {
5     regex pattern("(?=(10+1))");
6     sregex_iterator iter(s.begin(), s.end(), pattern);
7     sregex_iterator end;
8     int count = 0;
9     while (iter != end) {
10         ++count;
11         ++iter;
12     }
13     return count;
14 }
```

Python Code:

```
1 def countSubstrings(s):
2     pattern=r'(?=(10+1))'
3     matches=finditer(pattern,s)
4     count=0
5     for match in matches:count+=1
6     return count
```

Now, if we want only the unique substrings we can store those substrings in a set.

C++ Code:

```
1 set<string> uniqueSubstrings;
2 while (iter != end){
3     uniqueSubstrings.insert((*iter)[1]);
4     ++iter;
5 }
```

Python Code:

```
1 uniqueSubstrings = set()
2 for match in matches:
3     uniqueSubstrings.add(match.group(1))
```

*The size of the set **uniqueSubstrings** is the number of unique substrings that matches the pattern.*