# All about CP

## Table of Content


**1 Development** **2**

**2 FastIO** **2**
2.1 The Magic Line . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**3 Language** **3**
3.1 Return by reference . . . . . . . . . . . . . . . . . . . . . . . . . 3

**4 Maths** **3**
4.1 Summation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
4.1.1 Identities . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**5 String** **3**
5.1 Big Integers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**6 From A to B** **4**

**7 Bit Manipulation** **4**
7.1 Non-decimal Literal in C++ . . . . . . . . . . . . . . . . . . . . . . 4
7.2 How integers are stored . . . . . . . . . . . . . . . . . . . . . . . . 4
7.3 I/O with Non-Decimal Numbers . . . . . . . . . . . . . . . . . . . . 4
7.4 Signed and Unsigned Integers . . . . . . . . . . . . . . . . . . . . . 5
7.5 Index of Bit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.6 Terminologies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.7 Thinking in Binary . . . . . . . . . . . . . . . . . . . . . . . . . . 5
7.7.1 Position of bits . . . . . . . . . . . . . . . . . . . . . . . . 5
7.7.2 Converting to Decimal . . . . . . . . . . . . . . . . . . . . . 6
7.7.3 Maximizing/Minimizing bitstrings . . . . . . . . . . . . . . . 6
7.8 NOT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
7.8.1 `-x` in terms of `~x` . . . . . . . . . . . . . . . . . . . . . . 6
7.9 XOR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
7.9.1 How to Visualize XOR . . . . . . . . . . . . . . . . . . . . . 6
7.10 Thinking of Bitwise Operators: Fixing one operand . . . . . . . . 7
7.10.1 AND . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

7.10.2 OR . . . 7
7.10.3 XOR . . . 7
7.11 Visualizing n-1 . . . 7
7.12 Common Bit Operations and Checks . . . 8
7.12.1 Parity Check . . . 8
7.12.2 Left Shift as $\times 2$ . . . 8
7.12.3 Right Shift as $\div 2$ . . . 8
7.12.4 Set i-th Bit . . . 8
7.12.5 Clear i-th Bit . . . 8
7.12.6 Flip i-th Bit . . . 8
7.12.7 Check i-th Bit . . . 8
7.12.8 Unset Rightmost Set Bit . . . 9
7.12.9 Check if Power of Two . . . 9
7.12.10 Count Set Bits: Brian Kernighan's Algorithm . . . 9
7.12.11 Set Range of bits . . . 9
7.12.12 Clear Range of bits . . . 9
7.12.13 Flip Range of bits . . . 10
7.13 Builtin Functions . . . 10

**8 Ranges** **10**
8.1 Multiple Ranges . . . 10
8.1.1 Intersections of ranges . . . 10

**9 Number Theory** **11**
9.1 Divisibility of Integer N . . . 11
9.2 Binary Exponentiation . . . 11
9.3 Modular Arithmetic . . . 12
9.3.1 Basic Modular Operations . . . 12
9.3.2 Modular Exponentiation . . . 12

**10 Regular Expression** **12**
10.1 Matching Substring . . . 12

# 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

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

# 3 Language

## 3.1 Return by reference

Lets define an array to demonstrate return by reference

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

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

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

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

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

Demonstration:

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

**Problem:** This code causes TLE, but this gets AC

# 4 Maths

## 4.1 Summation

### 4.1.1 Identities

- $\sum c \times f(n) = c \times \sum f(n)$, $c$ 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 |

```
assert(13 == 0b1101);
assert(13 == 0xd);
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

```
int x;
cin>>hex>>x;  // takes input in hex
cout<<hex<<x; // prints output in hex
cin>>oct>>x;  // takes input in oct
cout<<oct<<x; // prints output in oct
cin>>dec>>x;  // takes input in dec
cout<<dec<<x; // prints output in dec

bitset<32> b;
cin>>b;       // takes 32 bit input in bin
cout<<b;      // prints 32 bit output in bin
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]

```
cout<<bitset<3>(5)<<"\n";
// Output: 101
cout<<bitset<32>(-1)<<"\n";
// 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 that 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
```

[1]cp-algorithms - bit manipulation

### 7.7.2 Converting to Decimal

To find decimal representation, we have to add up the positional values of set bits
As an example, lets 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 lenghts 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 significat bits that are unset

## 7.8 NOT

| Operation | Meaning |
|---|---|
| `~x` | 1's complement of `x` |
| `~x+1` | 2's complement of `x` |

### 7.8.1 -x in terms of ~x

```
assert(-x == ~x+1);
```

## 7.9 XOR

### 7.9.1 How to Visualize XOR

**Method-1: Non-equivalence Operator** `A^B` is `true` if truth value of `A` and `B` are different.
The following function uses this algorithm:

```
bool xor(bool a, bool b) {
  if (a!=b)
    return true;
  return false;
}
```

**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 ouput bit will be inverse of input bit, otherwise, the output bit will be the same as input bit.
In `A^B`, one bit decides if the other should be flipped.
The following function uses this algorithm:

```
bool xor(bool a, bool b) {
  if (a)
    return !b;
  return b;
}
```

## 7.10 Thinking of Bitwise Operators: Fixing one operand

### 7.10.1 AND

| Operation | Meaning |
|---|---|
| `& 1` | same |
| `& 0` | 0 |

### 7.10.2 OR

| Operation | Meaning |
|---|---|
| `\| 1` | 1 |
| `\| 0` | same |

### 7.10.3 XOR

| Operation | Meaning |
|---|---|
| `^ 1` | flip |
| `^ 0` | same |

## 7.11 Visualizing n-1

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

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

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

## 7.12 Common Bit Operations and Checks

### 7.12.1 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.

```
int n;
n=5;
assert((n&1) == 1);
assert((n%2) == 1);
n=-5;
assert((n&1) == 1);
assert((n%2) == -1);
```

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

### 7.12.2 Left Shift as $\times 2$

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

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

### 7.12.3 Right Shift as $\div 2$

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

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

### 7.12.4 Set i-th Bit

`n|(1<<i)`

### 7.12.5 Clear i-th Bit

`n&~(1<<i)`

### 7.12.6 Flip i-th Bit

We already discussed that, XOR works as a programmable inverter. `n^(1<<i)`

### 7.12.7 Check i-th Bit

`n&(1<<x) (n>>x)&1`

### 7.12.8 Unset Rightmost Set Bit

`n&(n-1)`

### 7.12.9 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

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

**Corner case:** 0

### 7.12.10 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

```
int countSetBits(int n) {
  int cnt=0;
  while(n)
    n=n&(n-1), cnt++;
  return cnt;
}
```

### 7.12.11 Set Range of bits

The concept is similar to setting i-th bit, except we will use a different bit mask.
`1<<i` is i 0s after one 1
`(1<<i)-1` is i 1s at the end, and rest are 0s
Now we can leftshift these 1s to fit into the range

```
int setRange(int n, int start, int stop) {
  int length = stop-start;
  int mask = (1<<length);
  mask = mask-1;
  mask = mask<<start;
  return n|mask;
}
```

### 7.12.12 Clear Range of bits

```
int clearRange(int n, int start, int stop) {
  int length = stop-start;
  int mask = (1<<length);
  mask = mask-1;
```

```
  mask = mask<<start;
  mask = ~mask;
  return n&mask;
}
```

#### 7.12.13 Flip Range of bits

```
int flipRange(int n, int start, int stop) {
  int length = stop-start;
  int mask = (1<<length);
  mask = mask-1;
  mask = mask<<start;
  return n^mask;
}
```

### 7.13 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:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 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

# 8 Ranges

## 8.1 Multiple Ranges

### 8.1.1 Intersections of ranges

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

**Edge Case:**
If $R < L$ , then the intersection is an empty range.

**Practice:**

- Problem 1
- Problem 2
- Problem 3

# 9 Number Theory

## 9.1 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 helpfull for those cases.

***N is***
1.Always divisible by 1.

2.Dibisible 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.

## 9.2 Binary Exponentiation

When we need to find $x^n$,

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

```cpp
#define ll long long
ll pow(ll x,ll n){//x^n
    if(n==0)return 1;
    ll z=modpow(x,n/2);
    z*=z;
    if (n&1)return x*z;
    return z;
}
```

Complexity:$O(logn)$

## 9.3 Modular Arithmetic

### 9.3.1 Basic Modular Operations

**Modular Addition:** (a+b)%m = ((a%m)+(b%m))%m
**Modular Multiplication** (aXb)%m= ((a%m) X (b%m))%m

### 9.3.2 Modular Exponentiation

**Calculate $x^n$ % m :**
Similar to Binary Exponentiation. But every time we need to use the modular multiplication formula.

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

Complexity:$O(logn)$

# 10 Regular Expression

## 10.1 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:**

```cpp
#include<bits/stdc++.h>
#include <regex>
using namespace std;
int countSubstrings(const string &s) {
```

```
  regex pattern("(?=(10+1))");
  sregex_iterator iter(s.begin(), s.end(), pattern);
  sregex_iterator end;
  int count = 0;
  while (iter != end) {
      ++count;
      ++iter;
  }
  return count;
}
```

**Python Code:**

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

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

**C++ Code:**

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

**Python Code:**

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

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