All about CP

Table of Content

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

```
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_ib_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 Bit Manipulation

5.1 Non-decimal Literal in C++

Base	Prefix
bin	0b
hex	0x
oct	0

```
assert(13 == 0b1101);
assert(13 == 0xd);
assert(13 == 015);
```

5.2 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

5.3 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();
```

5.4 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
```

5.5 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.

5.6 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

5.7 Thinking in Binary

We are used to thinking of numbers in decimal system. But computers store everything in binary. So, to be good at programming (i.e. talking to computers) we have to learn how to think in binary
Fun fact: Bit stands for “Binary Digit”

5.7.1 Position of bits

Binary, Octal, Hexadecimal and Decimal are called Positional Number System. That’s just a fancy way of saying that the position of the digits in a number matters, unlike tally.
Every position in a binary number has an index as mentioned here.
Each position also has a 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
```

In later sections, we will refer to this index and value multiple times. Since value of a bit depends on the position, we will also refer to it as positional value.

5.7.2 Converting to Decimal

To convert a binary number to decimal, we have to multiplay each bit (0 or 1) with its positional value and then, sum up the products.
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

Shortcut: From the example, you can notice that we are just adding up the values of the positions where there is a 1 and ignoring the positions that hold a 0. This is because, 0 times the positional value is, 0. So, we can just ignore those. And, 1 times the positional value is the positional value itself. So, we can just add them up.

5.7.3 Maximizing numbers

A common problem among bit manipulation problems is that you will be given a number and the definition of some binary operations you can perform on the number, and you have to maximize the number.
To solve this type of problems, you have to understand what *maximizing a number* means.

To understand the concept better, think of positional values as *significance* of a bit. Since the

5.8 NOT

Operation	Meaning
`~x`	1’s complement of `x`
`~x+1`	2’s complement of `x`

5.8.1 `-x` in terms of `~x`

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

5.9 XOR

5.9.1 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;
}
```

5.10 Thinking of Bitwise Operators: Fixing one operand

5.10.1 AND

Operation	Meaning
`& 1`	same
`& 0`	0

5.10.2 OR

Operation	Meaning
`| 1`	1
`| 0`	same

5.10.3 XOR

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

5.11 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

5.12 Common Bit Operations and Checks

5.12.1 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`

5.12.2 Left Shift as \times 2

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

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

5.12.3 Right Shift as \div 2

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

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

5.12.4 Set i-th Bit

`n|(1<<i)`

5.12.5 Clear i-th Bit

`n&~(1<<i)`

5.12.6 Flip i-th Bit

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

5.12.7 Check i-th Bit

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

5.12.8 Unset Rightmost Set Bit

`n&(n-1)`

5.12.9 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

5.12.10 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;
}
```

5.12.11 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;
}
```

5.12.12 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;
}
```

5.12.13 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;
}
```

5.13 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

---

1. cp-algorithms - bit manipulation↩