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

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

3 Bit Manipulation

3.1 Binary Literal in C++

You can represent binary numbers in C++ by using the `0b` prefix. For example: 13 in binary is `0b1101`.

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

3.2 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]

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

3.3 Index of Bit

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

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

3.4 XOR

3.4.1 How to Visualize XOR

Method-1: Non-equivalence Operator

$A \oplus B$ 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 $A \oplus B$, 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;
}
```

3.5 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
```

3.6 Common Bit Operations and Checks

3.6.1 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);
```

3.6.2 Left Shift as $\times 2$

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

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

3.6.3 Right Shift as $\div 2$

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

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

3.6.4 Set i-th Bit

```
n|(1<<i)
```

3.6.5 Clear i-th Bit

`n&~(1<<i)`

3.6.6 Flip i-th Bit

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

3.6.7 Check i-th Bit

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

3.6.8 Unset Rightmost Set Bit

`n&(n-1)`

3.6.9 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&(n-1))>0;
}
```

3.6.10 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;
}
```

3.6.11 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;
}
```

3.6.12 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;
}
```

3.6.13 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;
}
```

---

1. cp-algorithms - bit manipulation↩