All about CP

Table of Content

1 Development **3**

2 Fast I/O **4**
2.1 C++ 4
2.2 Python 4

3 C++ Specific **5**
3.1 Return by reference 5
3.2 Lambda Functions 5

4 Maths **7**
4.1 Summation 7

5 Constructive Algorithm **8**
5.1 From A to B 8

6 Searching **9**
6.1 Linear Search 9
6.2 Binary Search 10
6.3 Hash Table 12

7 Bit Manipulation **13**
7.1 Non-decimal Literal in C++ 13
7.2 How integers are stored 13
7.3 I/O with Non-Decimal Numbers 13
7.4 Signed and Unsigned Integers 14
7.5 Index of Bit 14
7.6 Terminologies 14
7.7 Thinking in Binary 14
7.8 NOT 15
7.9 XOR 16
7.10 XOR with AND OR 16
7.11 Thinking of Bitwise Operators: Fixing one operand 17
7.12 Visualizing n-1 17
7.13 Common Bit Operations and Checks 17

7.14 Builtin Functions 20
7.15 Sum with bit operations 20

8 Ranges **21**
8.1 Multiple Ranges 21

9 Interactive Problem **22**

10 Number Theory **23**
10.1 Binary Exponentiation 23
10.2 Modular Arithmetic 23
10.3 Big Integers 24

11 String **25**
11.1 Regular Expression 25

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.

Conventions

- Avoid comments: Name your variables and functions elaborately, instead
- Set indentation to two spaces
- Keep it concise
- Ranges should have inclusive start and exclusive stop: `[start,stop)`

Specific to C++:

- Make sure your code supports C++17 standard
- Use `ll` instead of `long long`
- Assume `bits/stdc++.h` is included

2 Fast I/O

2.1 C++

This will save you from TLEs in many problems

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

Caution: Either use `printf/scanf`, or use `cin/cout` with fast I/O. But don't use both

2.2 Python

Let's redefine the default input function to save your from TLEs

```
import sys
input = lambda: sys.stdin.readline().strip()
```

3 C++ Specific

3.1 Return by reference

Lets define an array for demonstration:

```
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
```

Practice:

- Leetcode - 77 - Combinations
  Solve this without declaring any helper function
  Note: This problem is not for beginners

3.2 Lambda Functions

Lambda functions are useful in two cases:

1. When you want a simple function without name
   Example: Sorting strings based on length

   ```
   string s[] = {"C++", "Python", "Java", "Rust"};
   sort(s, s+4, [&](string a, string b) -> bool {
     return a.size()<b.size();
   });
   ```

2. When you want to declare a function inside another function
   Example: Notice that the function is declared inside main function

```cpp
int main() {
  string s[] = {"C++", "Python", "Java", "Rust"};
  function<bool(string,string)> sortByLength =
    [&](string a, string b) -> bool {
      return a.size()<b.size();
    };
  sort(s, s+4, sortByLength);
}
```

Or, shorter

```cpp
int main() {
  string s[] = {"C++", "Python", "Java", "Rust"};
  auto sortByLength = [&](string a, string b) {
    return a.size()<b.size();
  };
  sort(s, s+4, sortByLength);
}
```

4 Maths

4.1 Summation

4.1.1 Identities

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

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

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

Practice:

- AtCoder - ARC A - Simple Math

5 Constructive Algorithm

5.1 From A to B

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

Practice:

- Codeforces - 1810 - Candies

6 Searching

6.1 Linear Search

Linear search sequentially checks each element of a range [1] until it finds the targeted value

```
int linearSearch(int a[], int n, int target) {
  for(int i=0; i<n; i++)
    if(a[i]==target)
      return i;
  return -1; // alternatively: return n;
}
```

Worst Case Time Complexity: $O(n)$

6.1.1 STL Algorithms with Linear Search

Let's define an integer vector for demonstration:

```
vector<int> v {3, 2, 1, 5, 3, 2, 6};
```

find
Returns the first occurance of the target in range

```
int i = find(v.begin(), v.end(), 2) - v.begin();
cout << "First occurance: " << i << '\n';

int j = find(v.rbegin(), v.rend(), 2) - v.begin();
cout << "Last occurance: " << n-j-1 << '\n';
```

count
Returns the number of elements that are equal to the target

```
int cnt = count(v.begin(), v.end(), 3); // 2
```

min_element, max_element, minmax_element
Code examples:

```
auto mn = min_element(v.begin(), v.end());
cout << "Minimum value: " << *mn << '\n';
cout << "Index of minimum value: " << mn-v.begin() << '\n';
```

```
auto mx = max_element(v.begin(), v.end());
cout << "Maximum value: " << *mx << '\n';
cout << "Index of maximum value: " << mx-v.begin() << '\n';
```

[1] A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers.
Source: https://cplusplus.com/reference/algorithm/

```
auto [mn,mx] = max_element(v.begin(), v.end());
cout << "Minimum value: " << *mn << '\n';
cout << "Index of minimum value: " << mn-v.begin() << '\n';
cout << "Maximum value: " << *mx << '\n';
cout << "Index of maximum value: " << mx-v.begin() << '\n';
```

min, max, minmax
Code examples:

```
auto mn = min({3, 2, 1, 5, 3, 2, 6});
cout << "Minimum value: " << mn << '\n';
```

```
auto mx = max({3, 2, 1, 5, 3, 2, 6});
cout << "Maximum value: " << mx << '\n';
```

```
auto [mn,mx] = max({3, 2, 1, 5, 3, 2, 6});
cout << "Minimum value: " << mn << '\n';
cout << "Maximum value: " << mx << '\n';
```

6.2 Binary Search

> *Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky*
>
> — Donald Knuth

Binary search compares the target value with the middle value of *sorted* range, and based on the comparison, it either finds the target value or chops off the range in half and again looks for the target value in the rest.

There are 2 categories of binary search problems:

1. Find exact match
2. Find smallest/largest solution

6.2.1 Exact Match

Examples:

1. Regular binary search
   $i : a_i = x$
2. Find peak/trough
   $i : (a_i > a_{i-1}) \land (a_i > a_{i+1})$
   $i : (a_i < a_{i-1}) \land (a_i < a_{i+1})$

For this category of problems, in each step there can be 1 of 3 possible outcomes:

1. Its a match!
2. Search in left half

3. Search in right half

All such problems can be simplified into this diagram, where > represents 'look right', < represents 'look left' and = represents 'its a match'.

`|>|>|>|=|<|<|<|<|`

The code for such problems is pretty straightforward

```
int l=0, r=n;
while(l<r) {
  int m=l+(r-l)/2;
  if(match) l=r=m;
  else if(look_right) l=m+1;
  else r=mid-1;
}
```

6.2.2 Smallest/Largest Solution

Examples:

1. Find `lower_bound`: first element greater than or equal to target
   min $i : a_i \geq x$
2. Find `upper_bound`: first element greater than ~~or equal to~~ target
   min $i : a_i > x$
3. Find last element less than or equal to target
   max $i : a_i \leq x$
   Same as `upper_bound-1`
4. Find last element less than target
   max $i : a_i < x$
   Same as `lower_bound-1`
5. Square root of N

For this category of problems, in each step there can be 1 of 3 possible outcomes:

1. Search for better solution in left half
2. Search for better solution in right half
3. There are no better solutions

All such problems can be simplified into this diagram, where `G` represents 'good' and `B` represents 'bad'. And, we have to find the position of last `B` or first `G`

`|B|B|B|B|B|G|G|G|`

Notice that the last `B` is always just before the first `G`
So, if we only learn the algorithm for finding the index of first `G`, then, we can substract 1 from it to find the index of last `B`

At this point you should be able to write a function(call it good), that takes the index as parameter and returns `true` for good and `false` otherwise.

For example, the good function of `lower_bound` is as follows:

```
auto good = [&](int i) {
  return a[i] >= target; // target is defined outside function
}
```

Then you can memorize the code for finding the first G

```
int firstGood(int l, int r, function<bool(int)> good) {
  while(l < r) {
    int m = l + (r - l) / 2;
    if(good(m)) r = m;
    else l = m + 1;
  }
  return l;
}
```

6.2.3 Overflow Error

`m=(l+r)/2` may cause overflow error. So use `m=l+(r-l)/2` instead

6.2.4 Characteristics of Binary Search Problems

If any of the following is true for a problem, then it is probably a binary search problem:

1. You will be given an array, and some queries on the array

6.3 Hash Table

7 Bit Manipulation

7.1 Non-decimal Literal in C++

To express integers in number systems other than decimal, use the following prefixes:

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 [2]

```
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
```

[2] 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 XOR with AND OR

A	B	$A \vee B$	$A \wedge B$	$A \oplus B$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

From the truth table it is clear that:
$A \oplus B = (A \vee B) - (A \wedge B)$
$A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$

This is also valid for bitwise operations:
`a^b=(a|b)-(a&b)`
`a^b=(a|b)&~(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
`^ 1`	flip
`^ 0`	same

7.12 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 $2^n - 1$

7.13 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.

```
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`

7.13.2 Left Shift as $\times 2$

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

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

7.13.3 Right Shift as $\div 2$

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

```
assert((11>>2) == 2);
```

7.13.4 Set i-th Bit

OR with a number with only i-th bit set

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

7.13.5 Clear i-th Bit

AND with a number with only i-th bit unset

```
n = n & ~(1<<i)
```

7.13.6 Flip i-th Bit

We already discussed that, XOR works as a programmable inverter.

```
n = n ^ (1<<i)
```

7.13.7 Check i-th Bit

AND with a number with only i-th bit set

```
n & (1<<i)
```

There is a chance of making overflow of bits, if `n` is of type `long long`. In general, it is better to write it this way:

```
(n>>i) & 1
```

7.13.8 Unset Rightmost Set Bit

See Visualizing `n-1`

```
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

```
n!=0 && (n&(n-1))>0;
```

Edge 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

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

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<<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

```
// Set bits in the range [start, stop)
int setRange(int n, int start, int stop) {
  int length = stop - start;
  int mask = (((1<<length) - 1) << start);
  return n | mask;
}
```

7.13.12 Clear Range of bits

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

7.13.13 Flip Range of bits

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

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:

```
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

7.15 Sum with bit operations

a&b represents the carry bit and a^b represents the last digit of a+b. Therefore the following formula holds:

```
a + b = ((a & b) << 1) + (a ^ b)
```

Since, `a ^ b` can be written in terms of AND, OR the sum can also be written in terms of AND, OR

```
a + b = ((a & b) << 1) + (a ^ b)
      = 2 * (a & b) + (a | b) - (a & b)
      = (a & b) + (a | b)
```

Practice:

- Codeforces - 1556D - Take a guess

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
$L = \max_{i=1}^{n} l_i$
$R = \min_{i=1}^{n} r_i$

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

Practice:

- Codeforces - 1282A - Temporarily unavailable
- Codeforces - 124A - The number of positions
- Codeforces - 714A - Meeting of Old Friends

9 Interactive Problem

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

Example:

```
int ask(string s, int a, int b) {
  cout << s << ' ' << a << ' ' << b << '\n';
  cout << flush;
  int res;
  cin >> res;
  return res;
}
```

10 Number Theory

10.1 Binary Exponentiation

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

C++ Code:

```
ll binPow(ll x, ll n) {
  if (n == 0) return 1;
  ll z = modpow(x, n/2);
  z *= z;
  if (n & 1) return x * z;
  return z;
}
```

Complexity: $O(logn)$

10.2 Modular Arithmetic

10.2.1 Basic Modular Operations

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

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

10.2.2 Modular Exponentiation

Calculate x^n % m :

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

```
ll modPow(ll x, ll n, ll 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: $O(logn)$

10.3 Big Integers

If N is a very large number (containing more than **40** digits), then **N should be read as string**.

10.3.1 Divisibility of Integer N

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 2 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.
7. Divisible by 7 or not, can not be checked efficiently
8. Divisible by 8 if the number containing only the last 3 digits of N is divisible by 8.
   Note: In general, N is divisible by 2^x if the number containing only the last x digits of N is divisible by 2^x.
9. Divisible by 9 if sum of digits is divisible by 9.
10. Divisible by 10, if …you've guessed it.

In Binary
N is divisible by 2^x if least significant x bits are 0

11 String

11.1 Regular Expression

11.1.1 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:

```
#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:

```
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:

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

Python Code:

```
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.*