[How to get started with solving Ad-Hoc tasks on codeforces]

Miscellaneous

STL

v2.resize (4);
sort (v2 . rbegin () , v2 . rend ())
reverse (v2 . begin () , v2 . end ())

Set/Multiset
The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

s . insert (5) ;
// erase () removes :
// - the only instance of the argument from a set
// - all of the instances of the argument from a multiset .
s . erase (5) ;
s . count (3)

Creating vector from a set
vector < int > v = {1 ,1 ,2 ,3 ,1 ,5 ,8 ,8 ,2};
set < int > s2 (v . begin () , v . end ()) ;

Avoiding TLE

Codeforces systems can, usually, do a maximum of 10^8 operations/sec.

Let n be the main variable in the problem.

- If $n \le 12$, the time complexity can be O(n!).
- If $n \le 25$, the time complexity can be $O(2^n)$.
- If $n \le 100$, the time complexity can be $O(n^4)$.
- If $n \le 500$, the time complexity can be $O(n^3)$.
- If $n \le 10^4$, the time complexity can be $O(n^2)$.
- If $n \le 10^6$, the time complexity can be O(n log n).
- If $n \le 10^8$, the time complexity can be O(n).
- If $n > 10^8$, the time complexity can be O(log n) or O(1).

Memory in C++

Reference Variable:
int i = 5;
int &j = 5;
j is a reference variable, pointing to the same location as i. i++ or j++ is the same.
Terrible Practice/Compiler Warning:

```
int& func(int a) {
    int num = a;
    int& ans = num;
    return ans;
}

int* fun(int n) {
    int* ptr = &n;
    return ptr;
}
```

In both cases, you're returning a reference to a memory that is already dead once the function call is over. It's dumb.

Stack Memory

During compiling the code, compiler allocates memory in stack. Stack memory has a fixed size. if you write **int arr[n]** and take **n as input**, you will get to know **n** on run time and not compile time, what if **n** exceeds the stack memory?

- **Heap is bigger, stack smaller.**
- **Memory is released by itself**

<u>Heap Memory</u> (*new* keyword is necessary. always return address)

- **Whenever you allocate memory in heap, it's called Dynamic Memory Allocation.**
- **Need to manually release memory.**
- **delete c**
- **delete []arr**

int* arr = new arr[5];
char* c = new char;
delete c;
c is in stack. new char is in heap.

<u>Comparing Floating Point Values</u>

Float = 4 bytes, upto 7 digits
Double = 8 bytes, upto 15 digits

Floating point numbers will have internal precision errors. They won't be equal to what they should be, but will be close to it.

```
int main() {
    double a = 0.3;
    cout << setprecision(20);

    cout << a;
}
```

Output:

```
0.29999999999999998889nis
```

So, to check if two floating point numbers are equal, don't use **==**
Instead, do:

```
int main() {
    double a = 0.3*3 + 0.1;
    double b = 1;

    if (abs(a-b) < 1e-9) {
        cout << "Numbers are equal";
    }
    else {
        cout << "Numbers are unequal";
    }
}
```

ASCII Tricks

Converting upper-case to lower-case:

```
cout << char('D' - 'A' + 'a') << endl;
```

Converting char to int:

```
cout << '5' - '0' << endl;
```

Output: 5 (integer)

Sorting a Vector of Pairs

Based on 1st element:

```
sort(vect.begin(), vect.end());
```

Based on 2nd element:

If comparator(element1, element2) returns true, left waala is entered else right waala.

```
bool comp(pair<int, int> a, pair<int, int> b) {

     return a.second < b.second)
}

sort(vect.begin(), vect.end(), comp);
```

Iterators

map.begin() = first element (stores a pair)
auto it = map.end();
it--;
it = last element

Pointers

size of a pointer = 8 (on 64 bit), 4 (on 32 bit)

int num = 5;
int* p = #

Symbol table:

variable name : memory address
num : 120
p : 080

at address 080, 120 is stored.

int *dontDo; //garbage value can be anything, might be some important memory address.
//ALWAYS initialize.

int *doThis = 0; //right way to initialize, point to a memory add that DNE (0)
//albeit segmentation fault

p = p + 1;
or
p = &num + 1;
=> value of p (address) increases by 4 (reaches next integer)

When does a variable change?

When int/char/etc. is passed to a function, a copy is created.
When arrays are passed, they are actually passed by reference.
(*ptr)++ => changes
int c = *ptr;
c++ => doesn't change original

Mathematics

Basics

Any number, can be represented as:
$237 = 2*10^2 + 2*10^2$

Q. How to get all digits of an integer?
Q. Number of digits in a number in base 10? (a should be a decimal number)
=> $[\log_{10}(a)]+1$
Q. Number of digits in base 2?
=> $[\log_2(a)]+1$
Q. Return **x** if **x** is 0 to 9, and return 0 if **x** is 10.

Decimal to Binary:

Method 1 -
1. num/2, note the remainder (last digit).
2. num = quotient
3. repeat

Method 2 -
1. num&1 = last bit
2. num = num >> 1

Binary to Decimal:

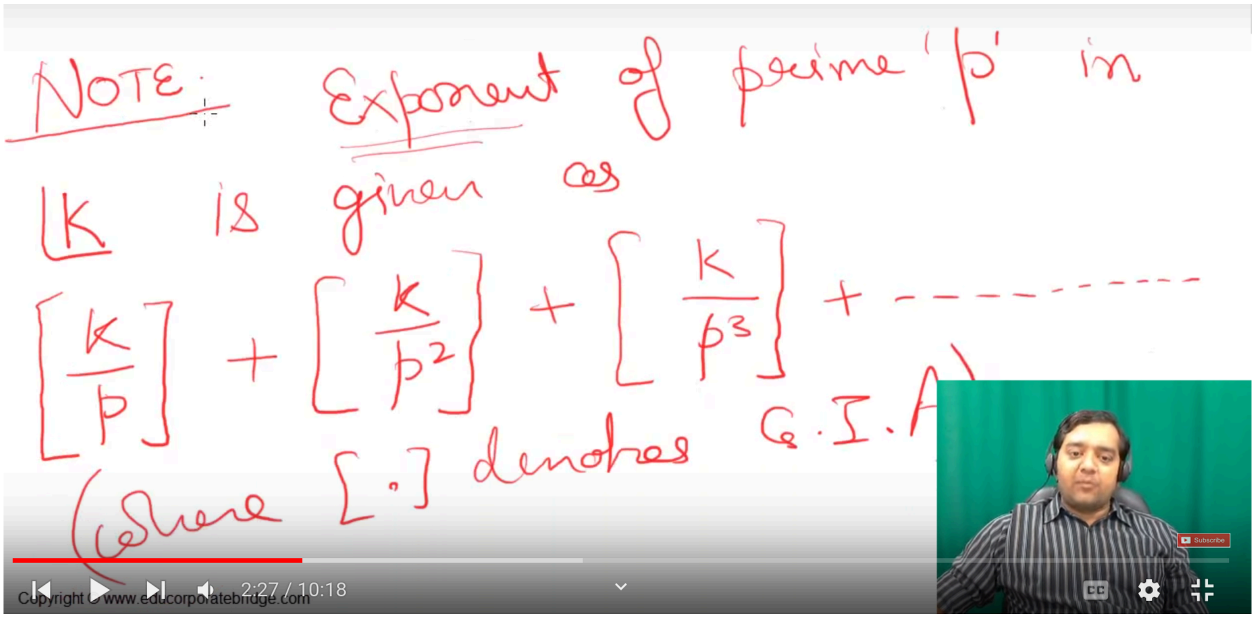
Normal multiplication.

Number Theory

- 0 ko har number divide karta hai.

Number of trailing zeroes in a factorial

Number of trailing zeroes = numebr of pairs of 2, 5. Power of 5 will always be less than that of 2. Hence, exponent of prime '5' is the answer.
Exponent of prime 'p' in fact(k)



Prime - A natural number with only two positive divisors, 1 and itself.
1 is neither prime nor composite. 2 is the lone even prime.

> Every positive integer has a unique prime factorization: a way of decomposing it into a product of primes, as follows:
>
> $n = {p_1}^{a_1} {p_2}^{a_2} \cdots {p_k}^{a_k}$
>
> where the p_i are distinct primes and the a_i are positive integers.
> For example:
> The prime factorization of 12 can be represented as
> $12 = 2^2 * 3^1$

Prime Factorization in O(√n)

Two facts:

- The smallest factor of any number (excluding 1) is a prime factor.
- This smallest factor lies before root(n), if the number is composite.

Har non prime ka pehla factor ek prime hoga. So, for every new ‘n’, the first factor is a prime. If not, then the number is prime itself.

```
void primeFactors(vector<int> & arr, int n) {               //O(√n)
```

```
    for (int i=2; i*i<=n;i++) {
    while (n%i==0) {
          arr.pb(i);
          n /= i;
    }
    }

    if (n>1) arr.pb(n);
}
```

This stores all the prime factors of a number. For 12, this stores 2,2,3.

<u>Checking whether a number is prime or not</u>

- Just check the size of this array.
- If any number from 2 to root(n) divides the number, it is not prime.

<u>All factors in O(√n)</u>

```
void allFactors(vector<int> & arr, int n) {                    //O(√n)

    for (int i=2; i*i<=n;i++) {
    if (n%i==0) {
          arr.pb(i);
          if (i*i!=n) arr.pb(n/i);
    }
    }

}
```

<u>Pairs? Factors?</u>

For example, the factors of 18 are $\{1, 2, 3, 6, 9, 18\}$. These can be paired up as $\{(1, 18), (2, 9), (3, 6)\}$.

For which of these integers can we pair them up like this?

For each factor below root(n), there will exist a corresponding factor after root(n). Hence, even number of factors.
But for numbers which are perfect squares, odd number of factors will exist.

Sum And Count of Divisors

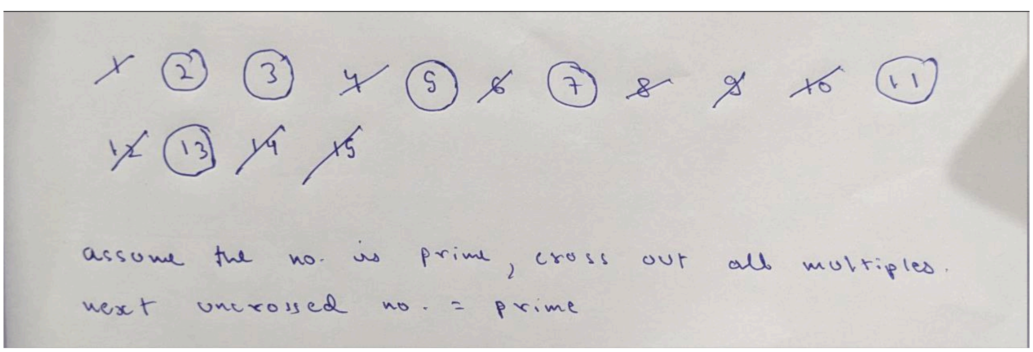
$x = p_1^{n1} p_2^{n2} p_3^{n3}$ (prime factorization of x)
n1 can have values from 0 to n1. each value of n1, n2, n3 gives a factor of x.

Number of divisors: **(n1 + 1)(n2 + 1)(n3 + 1)**
Sum of all divisors is given by:

$$(1 + p_1 + p_1^2 + \ldots + p_1^{n_1}) \times (1 + p_2 + p_2^2 + \ldots + p_2^{n_2}) \times \ldots$$

$$S(x) = \frac{p_1^{n_1+1} - 1}{p_1 - 1} \times \frac{p_2^{n_2+1} - 1}{p_2 - 1} \times \ldots$$

Sieve's Algorithm



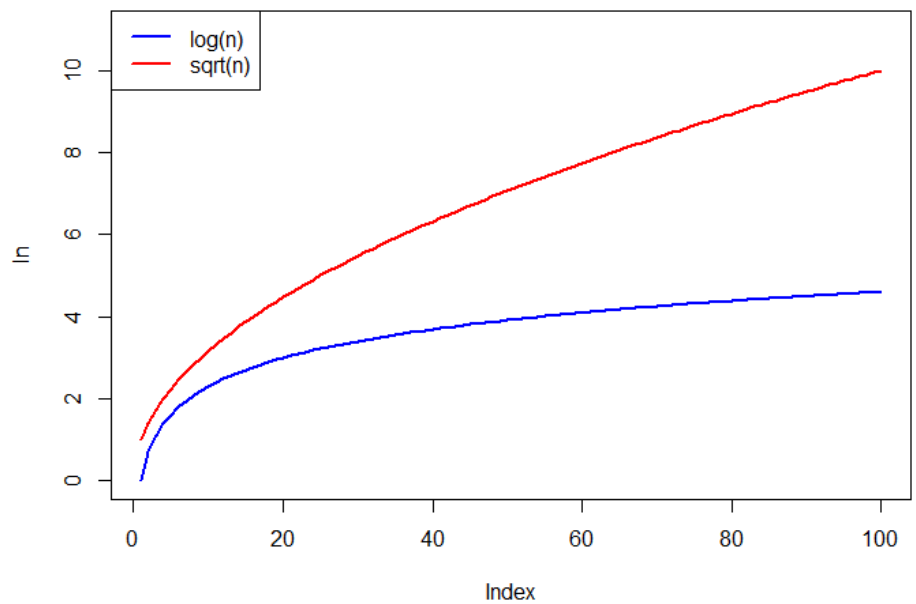

```
const int N = 1e7+10;
vector<bool> isPrime(N, true);
vector<int> lPrime(N, 0);
vector<int> hPrime(N);
// 24 = 2x2x2x3, lowest prime = 2, highest prime = 3
//O(log(log(N)). This is independent of base of log.
```

```
//O(log(N)) if we remove the if condition

void sieve() {
      isPrime[0] = isPrime[1] = false;
      for (int i=2;i<N;i++) {
      if (isPrime[i]) {
            lPrime[i] = hPrime[i] = i;
            for (int j=2*i;j<N;j+=i) {
                  isPrime[j] = false;
                  hPrime[j] = i;
                  if (lPrime[j]==0) lPrime[j] = i;
            }
      }
      }
}
```

Complexity:
Without the if condition, we sum:
N/2 + N/3 + N/4 + … N/N = Nlog(N)
With the if, we sum:
N/2 + N/3 + N/5 + …. = Nlog(log(N))

Prime Factorization using Sieve's Algorithm

```
void sievePrimeFact(vector<int> & arr, int n) {
//In the worst case, we keep dividing by 2. O(log2(n))

      while (n>1) {
            int factor = lPrime[n];
            while (n%factor==0) {
                  arr.pb(factor);
                  n /= factor;
            }
      }
}
```



Extra:
Numbers divisible by p = multiples of p
Numbers divisible by q = multiples of q
Numbers divisible by both p and q = multiples of lcm(p,q)

<u>GCD and LCM</u>

GCD = (a,b) = Largest positive common factor of a and b
LCM = [a,b] = Smallest positive number that can be divided by both a and b, or smallest common multiple of both

GCD = For each factor of a and b, take the lowest power and multiply. Even if the factor is not common.
LCM = Take the highest power and multiply.
Using the above, we can show that: ***gcd*lcm = product of numbers***

The LCM can be calculated with the GCD using this property:

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Warning!

Coding lcm as a * b / gcd(a, b) might cause integer overflow if the value of a * b is greater than the max size of the data type of a * b (e.g. the max size of int in C++ and Java is around $2 * 10^9$). Dividng a by gcd(a, b) first, then multiplying it by b will prevent integer overflow if the result fits in an int.

<u>Euclid's Algorithm</u>

Finding GCD(a,b) in O(log(a)), assuming a >= b.

A fact:

$$\mathbf{gcd(a, b) = gcd(b, a - b).}$$

Using it, we prove:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

a - qb = a%b (a mein se agar b ko quotient times ghataoge, to sirf remainder bachega)

<u>Proof of log2(a) complexity:</u>

- 2 is the smallest number we can divide by (worst case). The bigger number, a, can be divided at most log2(a) times.
- Or, proof:

Euclids Time Quest

We saw how to compute $\gcd(a, b)$, where $a \geq b$. But how much does it take? Let us try to count the number of steps that the algo takes.

Claim 1:
$a\%b \leq \frac{a}{2}$.

Proof:

- Case 1: If $b \leq \frac{a}{2}$, then, since $a\%b < b$ (from the definition of modulos), we have that $a\%b \leq \frac{a}{2}$.
- Case 2: If $b > \frac{a}{2}$, then $a\%b$ is just $a - b$ (since we can't subtract b twice from a), which is at most $\frac{a}{2}$. So, in this case also, we again have $a\%b \leq \frac{a}{2}$.

Thus, the claim is proved.

Now, let us see the first two steps of the Euclid's algo:

- We have (a, b), which gets converted into $(b, a\%b)$.
- We have $(b, a\%b)$, which gets converted into $(a\%b, b\%(a\%b))$.

So after two steps, the largest number, which was a has become $a\%b$ in the current pair, which as we have shown is $\leq \frac{a}{2}$.
Thus, in two steps, we have reduced the larger number by at least half. So, in at most $2 * \log_2 a$ steps, we will hit 0.
Thus, the time complexity of this algorithm is $O(log(a))$.

Results:

- gcd(a, b, c) = gcd(a, gcd(b, c)) = gcd(gcd(a, b), c) = gcd(gcd(a, c), b)
- 18/12 = 3/2 (Divide both by their GCD)

Modular Arithmetic

Instead of playing with numbers, we can play with number%m. Here, m will be a large prime, usually 10^9 + 7.

Why 10^9 + 7?

- A prime close to int maximum. Maximum an **int** can store is ~ 10^9. Hence, answer%m can be stored in an int.
- Modular Multiplicative Inverse of all numbers from 1 to prime can be found. And this is a prime.

Properties

- (a+b)%m = (a%m + b%m)%m
- (a-b)%m = (a%m - b%m + m)%m

- (a*b)%m = ((a%m) * (b%m))%m
- Different for division.
- If I say x is congruent to 2%3, then x%3 = 2%3.
- a and b are congruent mod c, so a%c = b%c.

<u>Why answer%m?</u>

We print answer%m so that it can be contained in the long long (or int) range. Sometimes, we can't even calculate the answer, in which cases we do **stepwise modulo operation**.

```
int main(){
    int n;
    cin >> n;
    int M = 47;
    long long fact = 1;
    for(int i = 2; i <=n; ++i){
        fact = (fact * i) % M;
    }
    cout << fact;
}
```

<u>Binary Exponentiation (a^b in log2(b) time)</u>

The reason we need to do a^b ourselves and can't use the pow(a,b) function is that this function returns **double**. While double can store large values, it has precision issues which might cause your code to not be accepted.

Method 1: Recursively
Divide and Conquer. Divide powers by 2 each time, only calculate even powers.

2^17 = 2*2^8*2^8
2^8 = 2^4*2^4
2^4 = 2^2 * 2^2
2^2 = 2 * 2
2 = 2 * 2^0

f(a,b) = f(a,b/2)*f(a,b/2) if b is even
f(a/b) = a*f(a,b/2)*f(a,b/2) if b is odd

```
int binaryExpRec(int a, int b) {
   if (b==0) return 1;
   int res = binaryExpRec(a, b/2);
   if (b&1) {
```

```
        return a * res * res;
    }
    else {
        return res * res;
    }
}
```

Method 2: Iterative

To break any number (here, power) in sum of 2, we convert it to binary and then write in powers of 2.

$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$ Since the number n has exactly $\lfloor \log_2 n \rfloor + 1$ digits in base 2, we only need to perform $O(\log n)$ multiplications, if we know the powers $a^1, a^2, a^4, a^8, \dots, a^{2^{\lfloor \log n \rfloor}}$.

```
//iterative is quicker

int binaryExp(int a, int b, int M) {            //in log2(b) time
      int res = 1;                              //What if a > 10^9?
      while (b > 0) {                           //Add a %= M in the start.
      if (b&1) {                                //(a^b)%M = (a%M)^b
            res = (res*1LL*a)%M;
      }
      b = b >> 1;
      a = (a * 1LL * a)%M;
      }
      return res;
}
```

Bit Manipulation

n = 101001100 => n-1 = 101001011 (right most 1 → 0, all zeroes to the right → 1)
LSB = Rightmost bit.

Bitwise operators compare each bit of both numbers to produce the result.

a^b (XOR/Exclusive OR) = Bits which are same will become zero, bits which are different will become 1.

a >> 3 => a is divided by 2^3
a << 4 => a is multiplied by 2^4. if you shift by a large number, you might make your number zero.

<u>Normal AND/OR/XOR Tricks</u>

- 0 ke saath | lene pe same hi rehjata hai bit.
- 1 ke saath & lene pe same hi rehjata hai bit.
- 1 ke saath ^ lene pe flip hojata hai bit.
- 0 ke saath ^ lene pe same rehjata hai bit.

<u>Bitwise Tricks</u>

- 1<<x = a number with the x’th bit set to 1 and all other 0.
- ~(1<<x) = a number with x’th bit set to 0 and all other 1.
- **Set/Flip/Clear a bit of a number ‘n’**
  Setting the x’th bit = n | (1<<x)
  Flipping the x’th bit = n ^ (1<<x)
  Clearing the x’th bit = n & ~(1<<x)
- **Checking if the x’th bit is set in ‘n’**
  Bring that bit to the end by n >> x;
  Do & 1.
- **Odd or even?**
  If (n&1==1) odd else even.
- **How to create 2^k?**
  2^k = 1 << k
- For a number to be divisible by 2^k, all bits to the right of 2^k banane wala bit should be zero. As, for a to be divisible by 2^k,
  a = 2^(k) + 2^(k+1) + 2^(k+2) + …
  Any smaller power is a problem.
- **Checking if a number is divisible by 2^k**
  if a&(2^k - 1) == 0, then it is.
- **Checking if ‘n’ is a power of (2, 4, 8, 16, 32…)**
  A power of 2 will only have one bit set, 32 = 0000000100000
  31 = 00000000111111
  32&31 == 0, that is,
  n&(n-1) == 0 if n is a power of 2. only exception is 0, which is not a power of 2.
- **Clearing the right-most set bit**
  n & (n-1)
- **Clear the trailing ones**
  n & (n+1)
  00110111 → 00110000
- **Set the rightmost 0 bit**

n | (n+1)
00110101 → 00110111

- **What is -n like?**
  Compared to ‘n’, all bits of ‘-n’ will be opposite, except the rightmost set bit which will be set in both.
  So, n&(-n) gives a number will all bits zero except the rightmost set bit.

Counting the number of set bits efficiently (Brian Kernighan’s Algorithm)
Worst case: O(log(n))
Jab tak rightmost bit exist karta hai tab tak ham usko zero kar payenge, and har baar count kar lenge.

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