LU_Furious

Leading University, Sylhet

Contents

1 Basic **2**
1.1 Code Body . . . 2
1.2 Ordered Set & Map . . . 2
1.3 Random Generator & Time . . . 2
1.4 Coordinate Compression . . . 2
1.5 Bitwise Operations . . . 2

2 Equations **2**
2.1 Math . . . 2
2.2 Bitwise Operations . . . 3
2.3 Combinatorics . . . 4
2.4 Geomatry . . . 5
2.5 Probability . . . 5
2.6 Number Theory . . . 6

3 Maths **6**
3.1 Sieve . . . 6
3.2 Prime Factors . . . 7
3.3 Divisors . . . 7
3.4 Digits . . . 7
3.5 Euler's Totient Function . . . 7
3.6 Extended Euclid . . . 7
3.7 Discrete Logarithm . . . 7
3.8 Multiple . . . 7
3.9 Find XOR of 1-n . . . 7
3.10 Base 10 to K . . . 8
3.11 Base 2 to 10 . . . 8
3.12 Max Subarray Sum . . . 8
3.13 nCr Calculation . . . 8
3.14 Binomial Coefficient . . . 8

4 Data Structures **8**
4.1 Segment Tree . . . 8
4.2 Lazy Segment Tree . . . 9
4.3 Disjoint Set Union . . . 9
4.4 Sparse Table . . . 9
4.5 2D Sparse Table . . . 10
4.6 Trie Strings . . . 10
4.7 Trie XOR . . . 10

5 Graph Theory **11**
5.1 Grid Moves . . . 11
5.2 Depth First Search . . . 11
5.3 Breath First Search . . . 11
5.4 0-1 BFS . . . 11
5.5 Cycle Detection . . . 12
5.6 Strongly Connected Components . . . 12
5.7 Topological Sort . . . 12
5.8 Dijkstra . . . 12
5.9 LCA . . . 13
5.10 LCA Weighted + Max edge . . . 13
5.11 Articulation Bridge . . . 14
5.12 Articulation Point . . . 14
5.13 Euler Path & Circuit (Undirected) . . . 14
5.14 Euler Path & Circuit (Directed) . . . 15
5.15 Condensed Graph (using SCC) . . . 15
5.16 Dinic's Algorithm for max flow / min-cut . . . 16
5.17 Maximum Flow Minimum Cost (MCMF) . . . 16

6 String Theory **17**
6.1 Double Hashing . . . 17
6.2 Property Suffix Automata . . . 17
6.3 Suffix Automata . . . 18
6.4 Extra . . . 18
6.5 Z Algorithm . . . 19
6.6 Trie . . . 19
6.7 Manacher's Algorithm . . . 20
6.8 Lexi Kth Duplicate . . . 20
6.9 Lexi Kth Unique . . . 21

7 Dynamic Programming **21**
7.1 Digit DP . . . 21
7.2 Basic DP Types . . . 22

1 Basic

1.1 Code Body

```cpp
#include<bits/stdc++.h>
using namespace std;
#define nl '\n'
#define all(v) v.begin(), v.end()
#define Too_Many_Jobs int tts, tc = 1; cin >> tts; hell:
    while(tts--)
#define Dark_Lord_Binoy ios_base::sync_with_stdio(false);
    cin.tie(NULL);

#ifdef LOCAL
#include "debug/whereisit.hpp"
#else
#define dbg(...) 42
#endif
#define int long long

int32_t main() {
Dark_Lord_Binoy
#ifdef LOCAL
   freopen("input.txt", "r", stdin);
   freopen("output.txt", "w", stdout);
#endif
   // code here

   return 0;
}
```

1.2 Ordered Set & Map

```cpp
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template <typename T>
using o_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
// order_of_key (k) : Number of items strictly smaller than
    k . O(logn)
// find_by_order(k) : K-th element in a set (counting from
    zero). O(logn)
template <typename T, typename R>
using o_map = tree<T, R, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
```

1.3 Random Generator & Time

```cpp
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());
int getRand(int l, int r) {
   uniform_int_distribution<int> uid(l, r);
   return uid(rng);
}
vector<int> generateRandomNumbers(int n, int l, int r) {
   std::vector<int> randomNumbers;
   for (int i = 0; i < n; ++i) {
      randomNumbers.push_back(getRand(l, r));
   }
   return randomNumbers;
}

cout<<fixed<<setprecision(10);
cerr<<"Time:"<<1000*((double)clock())/(double)CLOCKS_PER_SEC
    <<"ms\n";
```

1.4 Coordinate Compression

```cpp
template <typename T> vector<T> Compression (vector<T>& a) {
   int n = a.size();
   vector<T> d = a;
   sort(d.begin(), d.end());
   d.resize(unique(d.begin(), d.end()) - d.begin());
   for (int i = 0; i < n; i++) {
      a[i] = lower_bound(d.begin(), d.end(), a[i]) - d.
          begin() + 1;
   }
   return d; // Original value of a[i] is d[a[i]]
}
```

1.5 Bitwise Operations

```cpp
int isSet(int n, int i) { return ((n & (1 << i)) != 0); }
int setBit(int n, int i) { return ((n | (1 << i))); }
int unsetBit(int n, int i) { return ((n & (~(1 << i)))); }
int toggleBit(int n, int i) { return ((n ^ (1 << i))); }
int bitCount(int n) { return __builtin_popcount(n); }
int bitCountll(int n) { return __builtin_popcountll(n); }
int lsb(int n) { return __builtin_ctzll(n); }
int msb(int n) { return 63 - __builtin_ctzll(n); }
```

2 Equations

2.1 Math

The sum of integers from p to q:

$$p + (p+1) + \cdots + q = \frac{(q+p)(q-p+1)}{2}$$

The sum of integers from 1 to n:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

The sum of the first n odd numbers:

$$1 + 3 + 5 + \cdots + (2n-1) = n^2$$

The sum of the first n even numbers:

$$2 + 4 + 6 + \cdots + 2n = n(n+1)$$

The sum of squares of the first n integers:

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

The sum of cubes of the first n integers:

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2$$

The sum of squares of the first n odd numbers:

$$1^2 + 3^2 + 5^2 + \cdots + (2n-1)^2 = \frac{n(4n^2-1)}{3}$$

The sum of cubes of the first n odd numbers:

$$1^3 + 3^3 + 5^3 + \cdots + (2n-1)^3 = n^2(2n^2-1)$$

The sum of the fourth powers of the first n integers:

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

The sum of a geometric series with common ratio c:

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c-1} \quad \text{for } c \neq 1$$

The sum of powers of 2 up to 2^{k-1}:

$$2^0 + 2^1 + \cdots + 2^{k-1} = 2^k - 1$$

If $F(n) = -1 + 2 - 3 + \cdots + (-1)^n \cdot n$,

$$F(n) = \begin{cases} \frac{N}{2} & \text{if } N \text{ is even} \\ \frac{N+1}{2} \times (-1) & \text{if } N \text{ is odd} \end{cases}$$

The N-th odd number:

$$\text{N-th odd number} = 2N - 1$$

The N-th even number:

$$\text{N-th even number} = 2N$$

The sum of a geometric series with first term a and common ratio k:

$$a + a \cdot k + a \cdot k^2 + \cdots + b = \frac{b \cdot k - a}{k - 1}$$

The sum of an arithmetic series with first term a and difference of 4:

$$a + (a + 4) + (a + 2 \cdot 4) + \cdots + b = \frac{n(a + b)}{2}$$

Basic properties of even and odd numbers in addition and multiplication:

$$\text{even} \pm \text{even} = \text{even}, \quad \text{even} \pm \text{odd} = \text{odd}, \quad \text{odd} \pm \text{odd} = \text{even}$$

$$\text{even} \times \text{even} = \text{even}, \quad \text{even} \times \text{odd} = \text{even}, \quad \text{odd} \times \text{odd} = \text{odd}$$

The number of digits in a number N:

$$\text{Number of digits in } N = \lfloor \log_{10}(N) \rfloor + 1$$

The number of trailing zeros in $N!$:

$$\text{Trailing zeros in } N! = \sum_{k=1}^{\infty} \left\lfloor \frac{N}{5^k} \right\rfloor$$

The total number of squares in an $N \times N$ grid:

$$\text{Total squares} = \frac{n(n + 1)(2n + 1)}{6}$$

The angle between the minute and hour hands of a clock:

$$\text{Angle between minute and hour} = |0.5 \times 11 \times m - 30 \times h|$$

For the smaller angle, if angle > 180, then:

$$\text{angle} = 360 - \text{angle}$$

The number of ways to select one or more items from N different items:

$$2^N - 1$$

The number of possible N-bit numbers:

$$2^N$$

The number of unique triplets from an array of length n:

$$\frac{n(n - 1)(n - 2)}{6}$$

Logarithmic base conversion formula:

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)}$$

Modular multiplication property:

$$(A \times B) \mod \text{Mod} = ((A \mod \text{Mod})$$

$$\times (B \mod \text{Mod})) \mod \text{Mod}$$

Modular division formula:

$$(A/B) \mod \text{Mod} = ((A \mod \text{Mod}) \times$$

$$(\text{BinExp}(B, \text{Mod} - 2) \mod \text{Mod})) \mod \text{Mod}$$

2.2 Bitwise Operations

Bitwise AND (&) : The AND operation compares each corresponding bit of two numbers and returns 1 if both bits are 1, otherwise 0.

$$(1\&1) = 1$$

Bitwise OR (—) : The OR operation compares each corresponding bit of two numbers and returns 1 if at least one of the bits is 1, otherwise 0.

$$(0|1) = 1, \quad (1|0) = 1, \quad (1|1) = 1$$

Bitwise XOR ($\hat{}$) : The XOR operation compares each corresponding bit of two numbers and returns 1 if the bits are different, otherwise 0.

$$(0^1) = 1, \quad (1^0) = 1$$

Bitwise NOT ($\tilde{}$) : The NOT operation inverts all the bits of the number. For example, if $a = 1001_2$, then

$$\sim a = 0110_2$$

Check if a number is odd or even: You can check if a number is odd or even by performing a bitwise AND with 1.

$$(N\&1) == 1 \quad \text{(N is odd)}, \quad (N\&1) == 0 \quad \text{(N is even)}$$

Shifting operations: Bitwise shift left (¡¡) and right (¿¿) allow you to multiply or divide a number by powers of 2. For example,

$$(N/2) = (N >> 1), \quad (N \times 2) = (N << 1)$$

Power of two: To check if a number is a power of two, use the following condition:

$$\text{is\_power\_of\_two}(val) \quad \text{if} \quad (val\&(val - 1)) == 0$$

Swapping two numbers: A quick way to swap two numbers a and b using XOR is:

$$a \oplus = b, \quad b \oplus = a, \quad a \oplus = b$$

Check a specific bit: To check the bit at position *pos* in a number *val*:

$$\text{CheckBit}(val, pos) = (val \& (1LL << pos))$$

Set a specific bit: To set the bit at position *pos* in a number *val*:

$$\text{SetBit}(val, pos) = (val \mid (1LL << pos))$$

Clear a specific bit: To clear the bit at position *pos* in a number *val*:

$$\text{ClearBit}(val, pos) = (val \& \sim (1LL << pos))$$

Flip a specific bit: To flip the bit at position *pos* in a number *val*:

$$\text{FlipBit}(val, pos) = (val \oplus (1 << pos))$$

Most Significant Bit (MSB): The position of the most significant bit (MSB) in a number *mask* can be found using:

$$\text{MSB}(mask) = 63 - \text{\_\_builtin\_clzll}(mask)$$

Least Significant Bit (LSB): The position of the least significant bit (LSB) in a number *mask* can be found using:

$$\text{LSB}(mask) = \text{\_\_builtin\_ctzll}(mask)$$

Counting set bits: To count the number of set bits (1's) in a 32-bit integer x, use:

$$\text{\_\_builtin\_popcount}(x)$$

For a 64-bit integer, use:

$$\text{\_\_builtin\_popcountll}(x)$$

Bitset functions: The bitset functions allow you to manipulate and convert binary data. For example:

$$\text{bitset¡64¿ b1(val)} \quad \text{or} \quad \text{bitset¡4¿ b2("1011")}$$

To convert a bitset to an unsigned long integer:

$$\text{int val} = \text{b1.to\_ulong()} \quad (\text{b1} = 1001)$$

To convert a bitset to a string:

$$\text{s1} = \text{b1.to\_string()} \quad (\text{s1} = \text{"1001"})$$

To count the number of set bits in a bitset:

$$\text{bit} = \text{b1.count()} \quad (\text{bit} = 2)$$

2.3 Combinatorics

The binomial coefficient, also known as "n choose k," represents the number of ways to choose k elements from n elements.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

A permutation represents the arrangement of k objects selected from n objects.

$$P(n,k) = \frac{n!}{(n-k)!}$$

The number of ways to choose k elements from n elements with repetition is given by:

$$\binom{n+k-1}{k}$$

Factorial of a number n, denoted $n!$, is the product of all positive integers less than or equal to n.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

The multinomial coefficient generalizes the binomial coefficient to more than two groups. It represents the number of ways to divide n objects into k groups.

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1! k_2! \ldots k_m!}$$

where $k_1 + k_2 + \cdots + k_m = n$.

The Stirling number of the first kind, denoted $S(n,k)$, counts the number of permutations of n elements with exactly k permutation cycles.

$$S(n,k) = \text{no of permu of n el with k cycles}$$

The Stirling number of the second kind, denoted $S(n,k)$, counts the number of ways to partition a set of n elements into k non-empty subsets.

$$S(n,k) = \text{no of ways to partition n el into k subsets}$$

The inclusion-exclusion principle is used to calculate the size of the union of multiple sets.

$$|A \cup B| = |A| + |B| - |A \cap B|$$

For three sets:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|$$

The pigeonhole principle states that if n objects are placed into m containers, where $n > m$, at least one container must contain more than one object.

A derangement is a permutation where no element appears in its original position. The number of derangements of n elements, denoted $D(n)$, is:

$$D(n) = n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + \frac{(-1)^n}{n!}\right)$$

The Catalan number C_n counts the number of ways to correctly match parentheses or to form a binary tree. The n-th Catalan number is given by:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

2.4 Geomatry

A circle is a set of points equidistant from a central point (the center). **Radius:** r **Diameter:** $D = 2r$ **Circumference:** $C = 2\pi r$ **Area:** $A = \pi r^2$

A rectangle is a quadrilateral with opposite sides equal and four right angles. **Length:** L **Width:** W **Perimeter:** $P = 2(L + W)$ **Area:** $A = L \times W$

A square is a rectangle with all sides equal. **Side length:** s **Perimeter:** $P = 4s$ **Area:** $A = s^2$)

A polygon is a closed figure with straight sides. **Perimeter (regular polygon):** $P = n \times s$ **Area (regular polygon):** $A = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}$

A hexagon is a polygon with six equal sides. **Side length:** s **Perimeter:** $P = 6s$ **Area:** $A = \frac{3\sqrt{3}}{2}s^2$

A triangle is a polygon with three sides. **Perimeter:** $P = a + b + c$ **Area (Heron's formula):**

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = \frac{a+b+c}{2}$

A pyramid is a polyhedron with a polygonal base and triangular faces. **Base Area:** B **Height:** h **Volume:** $V = \frac{1}{3} \times B \times h$ **Surface Area (square base):**

$$SA = B + \frac{1}{2} \times P \times l$$

where P is the perimeter of the base and l is the slant height.

When calculating the distance between two points (x_1, y_1) and (x_2, y_2) in a 2D plane:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where $s = \frac{a+b+c}{2}$ is the semi-perimeter.

The inradius r of a triangle (the radius of the inscribed circle) is:

$$r = \frac{A}{s}$$

where A is the area of the triangle, and s is the semi-perimeter.

The circumradius R of a triangle (the radius of the circumscribed circle) for side lengths a, b, and c is:

$$R = \frac{abc}{4A}$$

where A is the area of the triangle.

To calculate the length of the altitude h_a from vertex A to side a:

$$h_a = \frac{2A}{a}$$

where A is the area of the triangle, and a is the length of the side opposite vertex A.

For the median m_a from vertex A to the midpoint of side a in a triangle with sides a, b, and c:

$$m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$$

The formula for the area A of a triangle using its circumradius R and side lengths a, b, and c:

$$A = \frac{abc}{4R}$$

The formula for the area A of a triangle with an angle θ between two sides a and b:

$$A = \frac{1}{2}ab\sin(\theta)$$

To find the centroid (center of mass) (x, y) of a triangle with vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3):

$$x = \frac{x_1 + x_2 + x_3}{3}, \quad y = \frac{y_1 + y_2 + y_3}{3}$$

where $s = \frac{a+b+c}{2}$.

To find the area of a circle with radius r:

$$A = \pi r^2$$

The circumference of a circle with radius r is:

$$C = 2\pi r$$

To calculate the length of the hypotenuse c in a right triangle with legs a and b:

$$c = \sqrt{a^2 + b^2}$$

2.5 Probability

When calculating the probability of an event A occurring:

$$P(A) = \frac{\text{Number of favorable outcomes}}{\text{Total number of outcomes}}$$

For two independent events A and B, the probability of both occurring is:

$$P(A \cap B) = P(A) \times P(B)$$

When the order doesn't matter and repetition is not allowed, the number of ways to choose r items from n items is:

$$nCr = \frac{n!}{r!(n-r)!}$$

When the order doesn't matter and repetition is allowed, the number of ways to choose r items from n items is:

$$nHr = \frac{(n+r-1)!}{r!(n-1)!}$$

2.6 Number Theory

When performing modular addition, the sum of two numbers a and b modulo m is given by:

$$(a+b) \mod m = ((a \mod m) + (b \mod m)) \mod m$$

Similarly, for modular subtraction:

$$(a-b) \mod m = ((a \mod m) - (b \mod m)) \mod m$$

To implement modular addition in code:

ll add (x, y) {x += y; return x ¿= mod ? x - mod : x;}

And for modular subtraction:

ll sub (x, y) {x -= y; return x ¡ 0 ? x + mod : x;}

For modular multiplication, the result of multiplying two numbers a and b modulo m is:

$$(a \times b) \mod m = ((a \mod m) \times (b \mod m)) \mod m$$

Modulo division $\frac{a}{b} \mod m$ exists when b and m are coprime; otherwise, modulo division is undefined.

To find the modular multiplicative inverse of a modulo m, use:

$$a \times x \equiv 1 \pmod m$$

which gives the inverse $x = a^{m-2} \mod m$, valid when a is coprime with m.

Fermat's Little Theorem states that for a prime m and a coprime with m:

$$a^{m-1} \equiv 1 \pmod m$$

Euler's Totient Theorem gives:

$$a^{\phi(m)} \equiv 1 \pmod m$$

where $\phi(m)$ is the count of numbers less than m that are coprime with m.

For solving modular exponentiation, especially with large numbers, you can use the power tower approach:

$$a^b \mod m = a^{b \mod \phi(m)} \mod m$$

The Extended Euclidean Algorithm helps find solutions to equations of the form:

$$ax + by = \gcd(a, b)$$

which also gives the modular inverse of $a \mod m$.

For solving Diophantine equations of the form:

$$ax + by = c$$

the equation has integer solutions if and only if c is divisible by $\gcd(a, b)$. If $\gcd(a, b)$ divides c, then solutions exist.

The Euclidean algorithm is used to find the greatest common divisor:

$$\gcd(a, b) \to \gcd(b, a \mod b)$$

If $b = 0$, then a is the gcd.

BigMod is a method used to calculate $a^n \mod m$ efficiently:

$$f(n) = \begin{cases} f(n/2)^2 & \text{if } n \text{ is even} \\ f(n-1) \times a & \text{if } n \text{ is odd} \end{cases}$$

For modular multiplication via addition, to find $a \times n \mod m$, we use:

$$f(n) = \begin{cases} f(n/2) + f(n/2) & \text{if } n \text{ is even} \\ f(n-1) + a & \text{if } n \text{ is odd} \end{cases}$$

Addition overflow occurs when $(a+b) \mod m$ exceeds the data limit. In such cases, the result is calculated as:

$$r = \begin{cases} b - (m - a) & \text{if } m - a < b \\ a + b & \text{otherwise} \end{cases}$$

The sum of a geometric series modulo m can be calculated as:

$$S = x + x^2 + x^3 + \cdots + x^n = x \times \frac{x^n - 1}{x - 1} \mod m$$

This is efficient when m is prime.

The sum of powers modulo m can be computed by a recursive method for even n as:

$$f(n) = f(n/2) + f(n/2) \times x^{n/2}$$

For odd n, it is:

$$f(n) = f(n-1) + x^n$$

For the discrete logarithm problem, given a and b, we find the minimum x such that:

$$a^x \equiv b \pmod m$$

Shank's Baby Step Giant Step algorithm solves this with a time complexity of $O(\sqrt{m})$.

3 Maths

3.1 Sieve

```
const int SZ = 1e6 + 5;
int is_prime[SZ];
void sieve() {
    int maxN = 1e6;
    is_prime[0] = is_prime[1] = 1;
    // is_prime[x] == 0 means prime number
    for(int i = 2; i * i <= maxN; i++) {
        if(is_prime[i] == 0) {
            for(int j = i * i; j <= maxN; j += i) {
                is_prime[j] = 1;
            }
        }
    }
}
```

3.2 Prime Factors

```
vector<int> primeFactors(int n) {
    vector<int> pfs;
    while(n % 2 == 0) {
        pfs.push_back(2);
        n = n / 2;
    }
    for (int i = 3; i * i <= n; i += 2) {
        while (n % i == 0) {
            pfs.push_back(i);
            n = n / i;
        }
    }
    if(n > 2) pfs.push_back(n);
    return pfs;
}
```

3.3 Divisors

```
vector<int> divisors(int n) {
    vector<int> divs;
    for(int i = 1; i * i <= n; i++) {
        if(n % i == 0) {
            if(n / i == i) divs.push_back(i);
            else {
                divs.push_back(i);
                divs.push_back(n / i);
            }
        }
    }
    sort(divs.begin(), divs.end());
    return divs;
}
```

3.4 Digits

```
vector<int> digits(int n) {
    vector<int> ans;
    while(n) {
        int cur = n % 10;
        ans.push_back(cur);
        n /= 10;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

3.5 Euler's Totient Function

```
// Euler's totient function, also known as phi-function (n),
// counts the number of integers between 1 and n inclusive,
    which are coprime to n.
void phi_1_to_n(int n) { // O(loglogn)
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++) phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

3.6 Extended Euclid

```
// we can use pow(a, m-2) if m is prime; else use this
int _gcd(int a, int b, int &x, int &y) {
    if(b == 0) {
        x = 1, y = 0;
        return a;
    }
    int x1, y1;
    int d = _gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
int inverse(int a, int m) { // a^-1 % m
    int x, y;
    int g = _gcd(a, m, x, y); // must be coprime, to get x as
        inverse of a
    if(g != 1) return -1;
    return (x % m + m) % m;
}
```

3.7 Discrete Logarithm

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

// baby step - giant step algo; a and m are co-prime
// returns minimum integer x such that a^x = b (mod m)
int discrete_log(int a, int b, int m) { // O(m)
    a %= m, b %= m;
    int n = (int) sqrt(m + .0) + 1;
    int an = 1; // x = np - q; a^np = b  a^q (mod m);
    for (int i = 0; i < n; i++) an = 1LL * an * a % m;
    gp_hash_table<int, int> vals;
    for (int q = 0, cur = b; q <= n; q++) {
        vals[cur] = q;
        cur = 1LL * cur * a % m;
    }
    for (int p = 1, cur = 1; p <= n; p++) {
        cur = 1LL * cur * an % m; // (a^n)^p
        if(vals.find(cur) != vals.end()) {
            return n * p - vals[cur];
        }
    }
    return -1; // does not exist
}
```

3.8 Multiple

```
long long multiply(long long A, long long B) { // (A * B) %
    mod
    long long ans = 0;
    while(B) {
        if(B & 1) ans = (ans + A) % mod;
        A = (A + A) % mod;
        B >>= 1;
    }
    return ans;
}
```

3.9 Find XOR of 1-n

```
// XOR of numbers from 1 to n:
int findXOR(int n) {
    int mod = n % 4;
    if (mod == 0) return n;
    else if (mod == 1) return 1;
    else if (mod == 2) return n + 1;
    else if (mod == 3) return 0;
}
```

3.10 Base 10 to K

```
vector<int> base10tok(int n, int k) {
    vector<int> ans;
    while(n > 0) {
        ans.push_back(n % k);
        n /= k;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

3.11 Base 2 to 10

```
ll binaryToDecimal(string &binaryStr) {
    ll result = 0;
    int length = binaryStr.size();
    for (int i = 0; i < length; i++) {
        if (binaryStr[i] == '1') {
            result = result * 2 + 1;
        }
        else if (binaryStr[i] == '0') {
            result = result * 2;
        }
    }
    return result;
}
```

3.12 Max Subarray Sum

```
template<typename T>
T maxSubArraySum(vector<T> &a) {
    int left = 0, right = 0, j = 0, n = a.size();
    T maxSum = INT_MIN, cur = 0;
    for(int i = 0; i < n; i++) {
        cur += a[i];
        if(maxSum < cur) {
            maxSum = cur;
            left = j;
            right = i;
        }
        if(cur < 0) {
            cur = 0;
            j = i + 1;
        }
    }
    // return {maxSum, {left, right}}; // {sum, {start index,
         end index}}
    return maxSum;
};
```

3.13 nCr Calculation

```
// nCr using binomial coefficient:
ll binCof (ll n, ll r) {
    ll res = 1;
    for (int i = 0; i < r; i++) {
        res *= (n - i);
        res /= (i + 1);
    }
    return res;
}

// DP approach
int binomialCoeff(int n, int k) {
    vector<int> dp(k + 1);
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = min(i, k); j > 0; j--)
            dp[j] = dp[j] + dp[j - 1];
    }
    return dp[k];
}
```

3.14 Binomial Coefficient

```
const int mod = 1e9 + 7, N = 2e5 + 5;

long long power(long long a, long long b) { // (a ^ b) % mod
    long long res = 1;
    while (b) {
        if (b & 1) res = (res * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return res;
}

struct BinomialCoefficient {
    int fact[N], invFact[N];

    BinomialCoefficient() {
        fact[0] = invFact[0] = 1;
        for (int i = 1; i < N; i++) {
            fact[i] = 1LL * fact[i - 1] * i % mod;
        }
        invFact[N - 1] = power(fact[N - 1], mod - 2);
        for (int i = N - 2; i >= 1; i--) {
            invFact[i] = 1LL * invFact[i + 1] * (i + 1) % mod
                ;
        }
    }
    int nCr(int n, int r) {
        if(n < r) return 0;
        return fact[n] * invFact[n - r] % mod * invFact[r] %
            mod;
    }
    int nPr(int n, int r) {
        if(n < r) return 0;
        return fact[n] * invFact[n - r] % mod;
    }
}bc;
```

4 Data Structures

4.1 Segment Tree

```
const int N = 3e5 + 9;

int a[N];
struct Node { // change this
    int val;
    Node() {
        val = 0;
    }
};
struct SegmentTree {
    #define lc (n << 1)
    #define rc ((n << 1) | 1)
    #define out INT_MIN // change this
    vector<Node> t;
    SegmentTree() {
        t.resize(4 * N);
    }
    inline void pull(int n) { // change this
        t[n].val = max(t[lc].val, t[rc].val);
    }
    inline Node combine(Node a, Node b) { // change this
        if(a.val == out) return b;
        if(b.val == out) return a;
        Node n;
```

```
        n.val = max(a.val, b.val);
        return n;
    }
    void build(int n, int b, int e) {
        if (b == e) {
            t[n].val = a[b]; // change this
            return;
        }
        int mid = (b + e) >> 1;
        build(lc, b, mid);
        build(rc, mid + 1, e);
        pull(n);
    }
    void upd(int n, int b, int e, int i, int x) {
        if (b > i || e < i) return;
        if (b == e && b == i) {
            t[n].val = x; // update
            a[b] = x;
            return;
        }
        int mid = (b + e) >> 1;
        upd(lc, b, mid, i, x);
        upd(rc, mid + 1, e, i, x);
        pull(n);
    }
    Node query(int n, int b, int e, int i, int j) {
        if (b > j || e < i) {
            Node x;
            x.val = out;
            return x; // return approriate value
        }
        if (b >= i && e <= j) return t[n];
        int mid = (b + e) >> 1;
        return combine(query(lc, b, mid, i, j), query(rc, mid
             + 1, e, i, j));
    }
}t;
```

4.2 Lazy Segment Tree

```
const int N = 5e5 + 9;
int a[N];
struct LazySegmentTree {
    #define lc (n << 1)
    #define rc ((n << 1) | 1)
    long long t[4 * N], lazy[4 * N];
    LazySegmentTree() {
        memset(t, 0, sizeof t);
        memset(lazy, 0, sizeof lazy);
    }
    inline void push(int n, int b, int e) {
        if (lazy[n] == 0) return;
        t[n] = t[n] + lazy[n] * (e - b + 1);
        if (b != e) {
            lazy[lc] = lazy[lc] + lazy[n];
            lazy[rc] = lazy[rc] + lazy[n];
        }
        lazy[n] = 0;
    }
    inline long long combine(long long a,long long b) {
        return a + b;
    }
    inline void pull(int n) {
        t[n] = t[lc] + t[rc];
    }
    void build(int n, int b, int e) {
        lazy[n] = 0;
        if (b == e) {
            t[n] = a[b];
            return;
        }
        int mid = (b + e) >> 1;
        build(lc, b, mid);
        build(rc, mid + 1, e);
        pull(n);
    }
    void upd(int n, int b, int e, int i, int j, long long v)
         {
        push(n, b, e);
        if (j < b || e < i) return;
        if (i <= b && e <= j) {
            lazy[n] = v; //set lazy
            push(n, b, e);
            return;
        }
        int mid = (b + e) >> 1;
        upd(lc, b, mid, i, j, v);
        upd(rc, mid + 1, e, i, j, v);
        pull(n);
    }
    long long query(int n, int b, int e, int i, int j) {
        push(n, b, e);
        if (i > e || b > j) return 0; //return null
        if (i <= b && e <= j) return t[n];
        int mid = (b + e) >> 1;
        return combine(query(lc, b, mid, i, j), query(rc, mid
             + 1, e, i, j));
    }
}t;
```

4.3 Disjoint Set Union

```
const int N = 3e5 + 9;

struct DSU {
    vector<int> par, rank, sz;
    int c;
    DSU(int n) : par(n + 1), rank(n + 1, 0), sz(n + 1, 1), c(
         n) {
        for (int i = 1; i <= n; i++) par[i] = i;
    }
    int find(int v) { // finding root of v
        if(par[v] == v) return v;
        else return par[v] = find(par[v]);
    }
    bool same(int a, int b) {
        return find(a) == find(b);
    }
    int get_size(int v) {
        return sz[find(v)];
    }
    int count() {
        return c; // connected components
    }
    void merge(int a, int b) {
        a = find(a), b = find(b);
        if(a == b) return; // already in same component
        else c--;
        if(rank[a] > rank[b]) swap(a, b);
        par[a] = b;
        sz[b] += sz[a];
        if(rank[a] == rank[b]) rank[b]++;
    }
};
```

4.4 Sparse Table

```
const int N = 1e5 + 9;

int a[N];
struct SparseTable {
    int t[N][18][2], logs[N];
    SparseTable() {
        memset(t, 0, sizeof t);
        memset(logs, 0, sizeof logs);
        for (int i = 2; i < N; i++) logs[i] = logs[i >> 1] +
             1;
    }
    void build(int n) { // O(nlogn)
```

```
        for(int i = 1; i <= n; i++) t[i][0][0] = t[i][0][1] =
            a[i];
        for(int k = 1; k < 18; k++) {
            for(int i = 1; i + (1 << k) - 1 <= n; i++) {
                t[i][k][0] = min(t[i][k - 1][0], t[i + (1 << (
                    k - 1))][k - 1][0]);
                t[i][k][1] = max(t[i][k - 1][1], t[i + (1 << (
                    k - 1))][k - 1][1]);
            }
        }
    }
    int minQuery(int l, int r) { // O(1)
        // int k = 31 - __builtin_clz(r - l + 1);
        int k = logs[r - l + 1];
        return min(t[l][k][0], t[r - (1 << k) + 1][k][0]);
    }
    int maxQuery(int l, int r) { // O(1)
        int k = logs[r - l + 1];
        return max(t[l][k][1], t[r - (1 << k) + 1][k][1]);
    }
}st;
```

4.5 2D Sparse Table

```
const int N = 505, LG = 10;

int st[N][N][LG][LG];
int a[N][N], lg2[N];

int maxQuery(int x1, int y1, int x2, int y2) {
  x2++;
  y2++;
  int a = lg2[x2 - x1], b = lg2[y2 - y1];
  return max(
        max(st[x1][y1][a][b], st[x2 - (1 << a)][y1][a][b]),
        max(st[x1][y2 - (1 << b)][a][b], st[x2 - (1 << a)][
            y2 - (1 << b)][a][b])
      );
} // int call = maxQuery(0, 0, i, j); <top-left to bottom
    right>

void build(int n, int m) { // 0 indexed
  for (int i = 2; i < N; i++) lg2[i] = lg2[i >> 1] + 1;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      st[i][j][0][0] = a[i][j];
    }
  }
  for (int a = 0; a < LG; a++) {
    for (int b = 0; b < LG; b++) {
      if (a + b == 0) continue;
      for (int i = 0; i + (1 << a) <= n; i++) {
        for (int j = 0; j + (1 << b) <= m; j++) {
          if (!a) {
            st[i][j][a][b] = max(st[i][j][a][b - 1], st[i][j
                + (1 << (b - 1))][a][b - 1]);
          } else {
            st[i][j][a][b] = max(st[i][j][a - 1][b], st[i +
                (1 << (a - 1))][j][a - 1][b]);
          }
        }
      }
    }
  }
}
```

4.6 Trie Strings

```
struct Node {
    Node *nxt[26]; // 26 (a - z)
    int pref, words; // current prefix & words cnt
    Node() {
        for (int i = 0; i < 26; i++) nxt[i] = NULL;
        pref = words = 0;
    }
    bool exists(char ch) { // link already created
        return (nxt[ch - 'a'] != NULL);
    }
    void create(char ch, Node *node) { // create new link
        nxt[ch - 'a'] = node;
    }
};
class Trie {
private: Node *root;
public:
    Trie() {
        root = new Node();
    }
    void insert(string word) {
        Node *cur = root;
        for (auto &ch : word) {
            if(!cur->exists(ch)) {
                cur->create(ch, new Node());
            }
            cur = cur->nxt[ch - 'a'];
            cur->pref++;
        }
        cur->words++;
    }
    void remove(string word) {
        Node *cur = root;
        for (auto &ch : word) {
            cur = cur->nxt[ch - 'a'];
            cur->pref--;
        }
        cur->words--;
    }
    int search(string word) {
        Node *cur = root;
        for (auto &ch : word) {
            if(!cur->exists(ch)) {
                return false;
            }
            cur = cur->nxt[ch - 'a'];
        }
        return cur->words;
    }
    int startsWith(string prefix) {
        Node *cur = root;
        for (auto &ch : prefix) {
            if(!cur->exists(ch)) {
                return false;
            }
            cur = cur->nxt[ch - 'a'];
        }
        return cur->pref;
    }
};
```

4.7 Trie XOR

```
struct Node {
    Node *nxt[2];
    int pref; // number of elements with this prefix
    Node() {
        nxt[0] = nxt[1] = NULL;
        pref = 0;
    }
    bool exists(int bit) { // link already created
        return (nxt[bit] != NULL);
    }
    void create(int bit, Node *node) { // create new link
        nxt[bit] = node;
    }
};
class Trie {
private: Node *root;
```

```cpp
public:
    Trie() {
        root = new Node();
    }
    void insert(int num) {
        Node *cur = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if(!cur->exists(bit)) {
                cur->create(bit, new Node());
            }
            cur = cur->nxt[bit];
            cur->pref++;
        }
    }
    void remove(int num) {
        Node *cur = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            cur = cur->nxt[bit];
            cur->pref--;
        }
    }
    int maxXor(int num) { // returns max of trie ^ num
        Node *cur = root;
        int ans = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if(cur->exists(!bit) && cur->nxt[!bit]->pref > 0)
                 {
                ans = ans | (1 << i);
                cur = cur->nxt[!bit];
            } else {
                cur = cur->nxt[bit];
            }
        }
        return ans;
    }
    int minXor(int num) { // returns min of trie ^ num
        Node *cur = root;
        int ans = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if(cur->exists(bit) && cur->nxt[bit]->pref > 0) {
                cur = cur->nxt[bit];
            } else {
                ans = ans | (1 << i);
                cur = cur->nxt[!bit];
            }
        }
        return ans;
    }
};
```

5 Graph Theory

5.1 Grid Moves

```cpp
vector<pair<int, int>> kingMoves = {{1, 0}, {-1, 0}, {0, 1},
      {0, -1}, {1, 1}, {1, -1}, {-1, 1}, {-1, -1}};
vector<pair<int, int>> knightMoves = {{-2, -1}, {-2, 1},
     {-1, -2}, {-1, 2}, {1, -2}, {1, 2}, {2, -1}, {2, 1}};
inline bool isValid(int i, int j) { return i < n && j < m &&
      i >= 0 && j >= 0; }
```

5.2 Depth First Search

```cpp
const int N = 2e5 + 9;

int a[N];
struct Dfs {
    int n;
    vector<int> lvl;
    vector<vector<int>> g;
    Dfs(int _n) : n (_n) {
        lvl.assign(n + 1, 0);
        g.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void dfs(int v, int p = -1) {
        for(auto u : g[v]) {
            if(u == p) continue;
            lvl[u] = 1 + lvl[v];
            dfs(u, v);
        }
    }
};
```

5.3 Breath First Search

```cpp
const int N = 2e5 + 9;

int a[N];
struct Bfs {
    int n;
    vector<int> lvl;
    vector<bool> vis;
    vector<vector<int>> g;
    Bfs(int _n) : n (_n) {
        lvl.assign(n + 1, 0);
        vis.assign(n + 1, false);
        g.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void bfs(int st) {
        vis[st] = true;
        queue<int> q;
        q.push(st);
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            dbg(v, lvl[v], a[v]);
            for (auto u : g[v]) if (!vis[u]) {
                vis[u] = true;
                lvl[u] = lvl[v] + 1;
                q.push(u);
            }
        }
    }
};
```

5.4 0-1 BFS

```cpp
const int N = 1e5 + 10;
vector<pair<int, int>> g[N];
vector<int> level(N, INT_MAX);
void bfs(int source) {
    deque<int> q;
    q.push_front(source);
    level[source] = 0;
    while (!q.empty()) {
        int par = q.front();
        q.pop_front();
        for (auto &child : g[par]) {
            int u = child.first, w = child.second;
            if(level[par] + w < level[u]) {
                level[u] = level[par] + w;
```

```
            if(w==0) q.push_front(u);
            else q.push_back(u);
        }
    }
  }
}
```

5.5 Cycle Detection

```
const int N = 5e5 + 9;

vector<pair<int, int>> g[N];
int vis[N], par[N], e_id[N];
vector<int> cycle; // simple cycle, contains edge ids

bool dfs(int u) {
    if (!cycle.empty()) return 1;
    vis[u] = 1;
    for (auto it : g[u]) {
        int v = it.first, id = it.second;
        if (vis[v] == 0) {
            par[v] = u;
            e_id[v] = id;
            if (dfs(v)) return 1;
        }
        else if (vis[v] == 1) {
            // cycle here
            cycle.push_back(id);
            for (int x = u; x != v; x = par[x]) {
                cycle.push_back(e_id[x]);
            }
            return 1;
        }
    }
    vis[u] = 2;
    return 0;
}
```

5.6 Strongly Connected Components

```
struct SCC {
    int n, scc;
    vector<int> vis;
    vector<vector<int>> g, gT, com;
    stack<int> st;
    SCC(int _n) : n (_n), scc (0) { // 1-indexed
        vis.assign(n + 1, 0);
        g.assign(n + 1, vector<int>());
        gT.assign(n + 1, vector<int>());
        com.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        gT[v].push_back(u); // reversed edges graph
    }
    void _dfs(int v) {
        vis[v] = 1;
        for(auto u : g[v]) {
            if(!vis[u]) {
                _dfs(u);
            }
        }
        st.push(v);
    }
    void _dfs2(int v) {
        vis[v] = 1;
        com[scc].push_back(v); // node v is in scc'th
             component
        for(auto u : gT[v]) {
            if(!vis[u]) {
                _dfs2(u);
            }
        }
    }
    vector<vector<int>> getComponents() {
        return com;
    }
    int getScc() {
        for (int i = 1; i <= n; i++) {
            if(!vis[i]) {
                _dfs(i);
            }
        }
        vis.assign(n + 1, 0);
        while(!st.empty()) {
            int u = st.top();
            st.pop();
            if(!vis[u]) {
                scc++;
                _dfs2(u);
            }
        }
        return scc;
    }
};
```

5.7 Topological Sort

```
// graph must be DAG - Directed Acyclic Graph
int n;
const int N = 2e5 + 5;
vector<int> g[N]; vector<bool> vis; vector<int> ts;

void dfs(int u) {
    vis[u] = true;
    for(auto v : g[u]) {
        if(!vis[v]) {
            dfs(v);
        }
    }
    ts.push_back(u);
}
void topSort() {
    vis.assign(n + 1, false);
    ts.clear();
    for (int i = 1; i <= n; i++) {
        if(!vis[i]) {
            dfs(i);
        }
    }
    reverse(ts.begin(), ts.end());
}
```

5.8 Dijkstra

```
const int N = 2e5 + 9, inf = 2e9;

int a[N];
struct Dijkstra {
    int n;
    vector<bool> vis;
    vector<int> dis;
    vector<vector<pair<int, int>>> g;
    Dijkstra(int _n) : n (_n) {
        vis.assign(n + 1, false);
        dis.assign(n + 1, inf);
        g.assign(n + 1, vector<pair<int, int>>());
    }
    void addEdge(int u, int v, int w) {
        g[u].push_back({v, w}); // <node, weight>
        g[v].push_back({u, w});
    }
    void dijkstra(int st) {
        priority_queue<pair<int, int>, vector<pair<int, int
             >>, greater<pair<int, int>>> pq; // minheap
```

```
        dis[st] = 0;
        pq.push({0, st}); // <distance, node>
        while(!pq.empty()) {
            int u = pq.top().second;
            pq.pop();
            if(vis[u]) continue;
            vis[u] = true;
            for(auto adj : g[u]) {
                int v = adj.first;
                int cost = adj.second;
                if(dis[u] + cost < dis[v]) {
                    dis[v] = dis[u] + cost;
                    pq.push({dis[v], v});
                }
            }
        }
    }
};
```

5.9 LCA

```
const int N = 2e5 + 9, LOG = 19;

struct LCA {
    int n;
    vector<int> dep, sz;
    vector<vector<int>> g, par;
    LCA(int _n) : n (_n) {
        dep.assign(n + 1, 0);
        sz.assign(n + 1, 0);
        g.assign(n + 1, vector<int>());
        par.assign(n + 1, vector<int>(LOG + 1));
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void build(int u, int p = 0) {
        par[u][0] = p;
        sz[u] = 1;
        for (int i = 1; i <= LOG; i++) par[u][i] = par[par[u
            ][i - 1]][i - 1];
        for (auto v : g[u]) if(v != p) {
            dep[v] = dep[u] + 1;
            build(v, u);
            sz[u] += sz[v];
        }
    }
    int lca(int u, int v) {
        if(dep[u] < dep[v]) swap(u, v);
        u = kth(u, dep[u] - dep[v]); // move u to same depth
            as v
        if(u == v) return u;
        for (int i = LOG; i >= 0; i--) {
            if(par[u][i] != par[v][i]) u = par[u][i], v = par
                [v][i];
        }
        return par[u][0];
    }
    int kth(int u, int k) {
        assert(k >= 0);
        for (int i = 0; i <= LOG; i++) {
            if(k & (1 << i)) u = par[u][i];
        }
        return u;
    }
    int dist(int u, int v) {
        int l = lca(u, v);
        return dep[u] + dep[v] - (dep[l] << 1);
    }
    // kth node from u to v, 0th node is u
    int go(int u, int v, int k) {
        int l = lca(u, v);
        int d = dep[u] + dep[v] - (dep[l] << 1);
        assert(k <= d);
        if(dep[u] >= dep[l] + k) return kth(u, k);
        k -= dep[u] - dep[l];
        return kth(v, dep[v] - (dep[l] + k));
    }
};
```

5.10 LCA Weighted + Max edge

```
const int N = 2e5 + 9, LOG = 19;

struct LCA {
    int n;
    vector<int> dep, sz;
    vector<vector<int>> par, mx;
    vector<vector<pair<int, int>>> g;
    LCA(int _n) : n (_n) {
        dep.assign(n + 1, 0);
        sz.assign(n + 1, 0);
        g.assign(n + 1, vector<pair<int, int>>());
        par.assign(n + 1, vector<int>(LOG + 1));
        mx.assign(n + 1, vector<int>(LOG + 1));
    }
    void addEdge(int u, int v, int w = 1) {
        g[u].push_back({v, w});
        g[v].push_back({u, w});
    }
    void build(int u, int p = 0) {
        par[u][0] = p;
        sz[u] = 1;
        for (int i = 1; i <= LOG; i++) par[u][i] = par[par[u
            ][i - 1]][i - 1];
        for (int i = 1; i <= LOG; i++) mx[u][i] = max(mx[u][i
             - 1], mx[par[u][i - 1]][i - 1]);
        for (auto v : g[u]) if(v.first != p) {
            dep[v.first] = dep[u] + 1;
            mx[v.first][0] = v.second;
            build(v.first, u);
            sz[u] += sz[v.first];
        }
    }
    int lca(int u, int v) {
        if(dep[u] < dep[v]) swap(u, v);
        u = kth(u, dep[u] - dep[v]); // move u to same depth
            as v
        if(u == v) return u;
        for (int i = LOG; i >= 0; i--) {
            if(par[u][i] != par[v][i]) u = par[u][i], v = par
                [v][i];
        }
        return par[u][0];
    }
    int kth(int u, int k) {
        assert(k >= 0);
        for (int i = 0; i <= LOG; i++) {
            if(k & (1 << i)) u = par[u][i];
        }
        return u;
    }
    int dist(int u, int v) {
        int l = lca(u, v);
        return dep[u] + dep[v] - (dep[l] << 1);
    }
    // kth node from u to v, 0th node is u
    int go(int u, int v, int k) {
        int l = lca(u, v);
        int d = dep[u] + dep[v] - (dep[l] << 1);
        assert(k <= d);
        if(dep[u] >= dep[l] + k) return kth(u, k);
        k -= dep[u] - dep[l];
        return kth(v, dep[v] - (dep[l] + k));
    }
    int getMaxEdge(int u, int v) {
        int l = lca(u, v);
```

```cpp
        int ans = 0, k = dep[u] - dep[l];
        for (int i = 0; i <= LOG; i++) {
            if(k & (1 << i)) {
                ans = max(ans, mx[u][i]);
                u = par[u][i];
            }
        }
        k = dep[v] - dep[l];
        for (int i = 0; i <= LOG; i++) {
            if(k & (1 << i)) {
                ans = max(ans, mx[v][i]);
                v = par[v][i];
            }
        }
        return ans;
    }
};
```

5.11 Articulation Bridge

```cpp
struct Bridge {
    int n, timer;
    vector<int> dis, low;
    vector<vector<int>> g;
    vector<pair<int, int>> bridges;
    Bridge(int _n) : n (_n), timer (0) {
        dis.assign(n + 1, 0);
        low.assign(n + 1, 0);
        g.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void _inBridge(int u, int v) {
        bridges.push_back({u, v});
    }
    void _dfs(int u, int p = -1) {
        dis[u] = low[u] = ++timer;
        bool parent_skipped = false;
        for(auto v : g[u]) {
            if(v == p && !parent_skipped) {
                parent_skipped = true;
                continue; // in case of multi edge to parent -
                    skip once
            }
            if(dis[v] == 0) { // not yet visited
                _dfs(v, u);
                low[u] = min(low[u], low[v]); // using child
                    nodes
                if(low[v] > dis[u]) _inBridge(u, v);
            }
            else {
                low[u] = min(low[u], dis[v]); // using back
                    edge
            }
        }
    }
    vector<pair<int, int>> getBridges() {
        _dfs(1);
        return bridges;
    }
};
```

5.12 Articulation Point

```cpp
struct AP {
    int n, timer;
    vector<int> dis, low;
    vector<bool> art;
    vector<vector<int>> g;
    vector<int> articulationPoints;
    AP(int _n) : n (_n), timer (0) {
        dis.assign(n + 1, 0);
        low.assign(n + 1, 0);
        art.assign(n + 1, false);
        g.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void _dfs(int u, int p = -1) {
        dis[u] = low[u] = ++timer;
        int child = 0;
        for(auto v : g[u]) {
            if(v == p) continue;
            if(dis[v] == 0) { // not yet visited
                _dfs(v, u);
                low[u] = min(low[u], low[v]); // using child
                    nodes
                if(low[v] >= dis[u] && p != -1) art[u] = true;
                child++;
            }
            else {
                low[u] = min(low[u], dis[v]); // using back
                    edge
            }
        }
        if(child > 1 && p == -1) art[u] = true;
    }
    vector<int> getPoints() {
        _dfs(1);
        for (int i = 0; i <= n; i++) {
            if(art[i]) articulationPoints.push_back(i);
        }
        return articulationPoints;
    }
};
```

5.13 Euler Path & Circuit (Undirected)

```cpp
/*
 all the edges should be in the same connected component
 #undirected graph: euler path: all degrees are even or
     exactly two of them are odd.
 #undirected graph: euler circuit: all degrees are even
*/

// euler path in an undirected graph: start from odd deg ->
    finish in odd deg;
// it also finds circuit if it exists
struct Euler {
    int n, edges;
    vector<bool> vis;
    vector<int> done, path, deg;
    vector<vector<pair<int, int>>> g;
    Euler(int _n, int _edges) : n (_n), edges (_edges) {
        vis.assign(edges + 1, false);
        done.assign(n + 1, 0);
        deg.assign(n + 1, 0);
        g.assign(n + 1, vector<pair<int, int>>());
    }
    void addEdge(int u, int v, int idx) {
        g[u].push_back({v, idx});
        g[v].push_back({u, idx});
        deg[u]++, deg[v]++;
    }
    void dfs(int u) {
        while(done[u] < g[u].size()) {
            auto v = g[u][done[u]++];
            if(vis[v.second]) continue;
            vis[v.second] = true;
            dfs(v.first);
        }
        path.push_back(u);
```

```
    }
    bool hasEulerPath() {
        path.clear();
        int odd = 0, root = -1;
        for (int i = 1; i <= n; i++) {
            if(deg[i] & 1) odd++, root = i;
        }
        if(odd > 2) return false;
        if(root == -1) { // odd == 0
            for (int i = 1; i <= n; i++) if (deg[i]) { root =
                 i; break; }
        }
        if(root == -1) return true; // empty graph
        dfs(root);
        if(path.size() != edges + 1) return false;
        reverse(path.begin(), path.end());
        return true;
    }
    vector<int> getEulerPath() {
        if(hasEulerPath()) return path;
        return {};
    }
};
```

5.14 Euler Path & Circuit (Directed)

```
/*
  all the edges should be in the same connected component
  #directed graph: euler path: for all -> indeg = outdeg or
      nodes having indeg > outdeg = outdeg > indeg = 1 and
      for others in = out
  #directed graph: euler circuit: for all -> indeg = outdeg
*/

// euler path in a directed graph
// it also finds circuit if it exists
struct Euler {
    int n, edges;
    vector<int> done, path, in, out;
    vector<vector<int>> g;
    Euler(int _n, int _edges) : n (_n), edges (_edges) {
        done.assign(n + 1, 0);
        in.assign(n + 1, 0);
        out.assign(n + 1, 0);
        g.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        out[u]++, in[v]++;
    }
    void dfs(int u) {
        while(done[u] < g[u].size()) {
            int v = g[u][done[u]++];
            dfs(v);
        }
        path.push_back(u);
    }
    bool hasEulerPath() {
        path.clear();
        int cnt1 = 0, cnt2 = 0, root = -1;
        for (int i = 1; i <= n; i++) {
            if(in[i] - out[i] == 1) cnt1++;
            if(out[i] - in[i] == 1) cnt2++, root = i;
            if(abs(in[i] - out[i] > 1)) return false;
        }
        if(cnt1 > 1 || cnt2 > 1) return false;
        if(root == -1) { // all in == out degree's
            for (int i = 1; i <= n; i++) if(out[i]) { root =
                 i; break; }
        }
        if(root == -1) return true; // empty graph
        dfs(root);
        if(path.size() != edges + 1) return false; //
             connectivity issue
        reverse(path.begin(), path.end());
        return true;
    }
    vector<int> getEulerPath() {
        if(hasEulerPath()) return path;
        return {};
    }
};
```

5.15 Condensed Graph (using SCC)

```
// eleminates loops (into a single node)
struct SCC {
    int n, scc;
    vector<int> vis;
    vector<vector<int>> g, gT, com;
    stack<int> st;
    bool sccDone;
    SCC(int _n) : n (_n), scc (0), sccDone (false) { // 1-
         indexed
        vis.assign(n + 1, 0);
        g.assign(n + 1, vector<int>());
        gT.assign(n + 1, vector<int>());
        com.assign(n + 1, vector<int>());
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
        gT[v].push_back(u); // reversed edges graph
    }
    void _dfs(int v) {
        vis[v] = 1;
        for(auto u : g[v]) {
            if(!vis[u]) {
                _dfs(u);
            }
        }
        st.push(v);
    }
    void _dfs2(int v) {
        vis[v] = 1;
        com[scc].push_back(v); // node v is in scc'th
             component
        for(auto u : gT[v]) {
            if(!vis[u]) {
                _dfs2(u);
            }
        }
    }
    vector<vector<int>> getComponents() {
        getScc();
        return com;
    }
    int getScc() {
        if(sccDone) return scc;
        for (int i = 1; i <= n; i++) {
            if(!vis[i]) {
                _dfs(i);
            }
        }
        vis.assign(n + 1, 0);
        while(!st.empty()) {
            int u = st.top();
            st.pop();
            if(!vis[u]) {
                scc++;
                _dfs2(u);
            }
        }
        sccDone = true;
        return scc;
    }
    vector<vector<int>> getCondensedGraph() {
        getScc();
        vector<vector<int>> gC(n + 1, vector<int>());
```

```cpp
        vector<int> par(n + 1, 0);
        for(auto component : com) { // all component nodes ->
             1 node (par)
            if(component.empty()) continue;
            int root = *min_element(component.begin(),
                 component.end());
            for(auto v : component) par[v] = root;
        }
        for (int v = 1; v <= n; v++) {
            for(auto u : g[v]) {
                if(par[v] != par[u]) gC[par[v]].push_back(par[
                     u]);
            }
        }
        return gC;
    }
};
```

5.16 Dinic's Algorithm for max flow / min-cut

```cpp
#define sz(a) int((a).size())
struct Dinic {
    struct edge {
        int u, rev;
        ll cap, flow;
    };

    int n, s, t;
    ll flow;
    vector<int> lst;
    vector<int> d;
    vector<vector<edge>> g;

    Dinic() {}
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        g.resize(n);
        d.resize(n);
        lst.resize(n);
        flow = 0;
    }

    void add_edge(int v, int u, ll cap, bool directed = true)
         {
        g[v].push_back({u, sz(g[u]), cap, 0});
        g[u].push_back({v, sz(g[v]) - 1, directed ? 0 : cap,
             0});
    }

    ll dfs(int v, ll flow) {
        if (v == t) return flow;
        if (flow == 0) return 0;
        ll result = 0;
        for (; lst[v] < sz(g[v]); ++lst[v]) {
            edge& e = g[v][lst[v]];
            if (d[e.u] != d[v] + 1) continue;
            ll add = dfs(e.u, min(flow, e.cap - e.flow));
            if (add > 0) {
                result += add;
                flow -= add;
                e.flow += add;
                g[e.u][e.rev].flow -= add;
            }
            if (flow == 0) break;
        }
        return result;
    }

    bool bfs() {
        fill(d.begin(), d.end(), -1);
        queue<int> q({s});
        d[s] = 0;
        while (!q.empty() && d[t] == -1) {
            int v = q.front(); q.pop();
            for (auto& e : g[v]) {
                if (d[e.u] == -1 && e.cap - e.flow > 0) {
                    q.push(e.u);
                    d[e.u] = d[v] + 1;
                }
            }
        }
        return d[t] != -1;
    }

    ll calc() {
        ll add;
        while (bfs()) {
            fill(lst.begin(), lst.end(), 0);
            while((add = dfs(s, numeric_limits<ll>::max())) >
                 0)
                flow += add;
        }
        return flow;
    }
};
```

5.17 Maximum Flow Minimum Cost (MCMF)

```cpp
#define INF INT_MAX

struct Edge {
    int to, capacity, cost, rev;
};

struct MinCostMaxFlow {
    int n;
    vector<vector<Edge>> graph;
    vector<int> dist, parent, parentEdge;
    vector<bool> inQueue;

    MinCostMaxFlow(int n) : n(n), graph(n), dist(n), parent(n
         ), parentEdge(n), inQueue(n) {}

    void addEdge(int u, int v, int cap, int cost) {
        graph[u].push_back({v, cap, cost, (int)graph[v].size
             ()});
        graph[v].push_back({u, 0, -cost, (int)graph[u].size()
             - 1});
    }

    bool spfa(int s, int t) {
        fill(dist.begin(), dist.end(), INF);
        fill(inQueue.begin(), inQueue.end(), false);
        queue<int> q;

        dist[s] = 0;
        q.push(s);
        inQueue[s] = true;

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            inQueue[u] = false;

            for (int i = 0; i < graph[u].size(); ++i) {
                Edge &e = graph[u][i];
                if (e.capacity > 0 && dist[u] + e.cost < dist[
                     e.to]) {
                    dist[e.to] = dist[u] + e.cost;
                    parent[e.to] = u;
                    parentEdge[e.to] = i;
                    if (!inQueue[e.to]) {
                        q.push(e.to);
                        inQueue[e.to] = true;
                    }
```

```
            }
        }
    }
    return dist[t] != INF;
}

pair<int, int> minCostMaxFlow(int s, int t) {
    int flow = 0, cost = 0;
    while (spfa(s, t)) {
        int curr_flow = INF;
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            Edge &e = graph[u][parentEdge[v]];
            curr_flow = min(curr_flow, e.capacity);
        }
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            Edge &e = graph[u][parentEdge[v]];
            e.capacity -= curr_flow;
            graph[v][e.rev].capacity += curr_flow;
            cost += curr_flow * e.cost;
        }
        flow += curr_flow;
    }
    return {flow, cost};
}
};
```

6 String Theory

6.1 Double Hashing

```
// linear Hash
// ll hash(string const& s) {
//   const int p = 31;
//   const int m = 1e9 + 9;
//   ll hash_value = 0;
//   ll p_pow = 1;
//    for (char c : s) {
//      hash_value = (hash_value + (c - 'a' + 1) * p_pow) %
   m;
//      p_pow = (p_pow * p) % m;
//    }
//   return hash_value;
// }

ll power(ll x, ll y, ll p){
  ll r = 1;
  for(; y; y >>= 1, x = x * x % p){
  if(y & 1) {r = r * x % p;}}
  return r;}
ll inv(ll x, ll p) {return power(x, p - 2, p);}

class Hash{
  private:
  int length;
    const int mod1 = 1e9 + 7, mod2 = 1e9 + 9;
    const int p1 = 111111, p2 = 1111111;
    vector<int> hash1, hash2;
    pair<int, int> hash_pair;

public:
    vector<int> inv_pow1, inv_pow2;
    int inv_size = 1;
    Hash() {}
    Hash(const string& s) {
        length = s.size();
        hash1.resize(length);
        hash2.resize(length);

        int h1 = 0, h2 = 0;
        ll p_pow1 = 1, p_pow2 = 1;
        for(int i = 0; i < length; i++) {
            h1 = (h1 + (s[i] - 'a' + 1) * p_pow1) % mod1;
            h2 = (h2 + (s[i] - 'a' + 1) * p_pow2) % mod2;
            p_pow1 = (p_pow1 * p1) % mod1;
            p_pow2 = (p_pow2 * p2) % mod2;
            hash1[i] = h1;
            hash2[i] = h2;
        }
        hash_pair = make_pair(h1, h2);

        if(inv_size < length) {
            for(; inv_size < length; inv_size <<= 1);

            inv_pow1.resize(inv_size, -1);
            inv_pow2.resize(inv_size, -1);

            inv_pow1[inv_size - 1] = inverse(power(p1,
               inv_size - 1, mod1), mod1);
            inv_pow2[inv_size - 1] = inverse(power(p2,
               inv_size - 1, mod2), mod2);

            for(int i = inv_size - 2; i >= 0 && inv_pow1[i]
               == -1; i--) {
                inv_pow1[i] = (1LL * inv_pow1[i + 1] * p1) %
                   mod1;
                inv_pow2[i] = (1LL * inv_pow2[i + 1] * p2) %
                   mod2;
            }
        }
    }
    int size() {
        return length;
    }
    pair<int, int> prefix(const int index) {
        return {hash1[index], hash2[index]};
    }
    pair<int, int> substr(const int l, const int r) {
        if(l == 0) {
            return {hash1[r], hash2[r]};
        }
        int temp1 = hash1[r] - hash1[l - 1];
        int temp2 = hash2[r] - hash2[l - 1];
        temp1 += (temp1 < 0 ? mod1 : 0);
        temp2 += (temp2 < 0 ? mod2 : 0);
        temp1 = (temp1 * 1LL * inv_pow1[l]) % mod1;
        temp2 = (temp2 * 1LL * inv_pow2[l]) % mod2;
        return {temp1, temp2};
    }
    bool operator==(const Hash& other) {
        return (hash_pair == other.hash_pair);
    }
};
int main() {
    string str;
    cin >> str;
    int len = 5;
    auto hash = Hash(str);
    auto hash_pair = hash.substr(0, len - 1);
}
```

6.2 Property Suffix Automata

```
1) Substring Search:
bool searchSubstring(const string& substring) {
  int currentState = 0;
  for (char c : substring) {
    if (suffixAutomaton[currentState].next.find(c) ==
         suffixAutomaton[currentState].next.end()) {
        return false;
    }currentState = suffixAutomaton[currentState].next[c];
  }return true;
}int main(){
string substring = "babd";
if (searchSubstring(substring)){ok
```

```
} else {not ok}}

2) Longest common substring between 2 String
string longestCommonSubstring(const string& s1, const string
    & s2) {
  initialize();
  for (char c : s1) {
    extendAutomaton(c);
  }
  int currentState = 0;
  int maxLength = 0;
  int length = 0;
  int endIndex = -1;
  for (int i = 0; i < s2.length(); ++i) {
    char c = s2[i];
    while (currentState != -1 && suffixAutomaton[currentState
        ].next.find(c) == suffixAutomaton[currentState].next
        .end()) {
      currentState = suffixAutomaton[currentState].link;
      length = (currentState == -1) ? 0 : suffixAutomaton[
          currentState].length;
    }
    if (currentState != -1) {
      currentState = suffixAutomaton[currentState].next[c];
      length++;
    } else {
      currentState = 0;
      length = 0;
    }
    if (length > maxLength) {
      maxLength = length;
      endIndex = i;
    }
  }
  if (maxLength == 0) return "";
  return s2.substr(endIndex - maxLength + 1, maxLength);}
int main() {
  string s1 = "dddabcde";
  string s2 = "abbblllldllll";
  string lcs = longestCommonSubstring(s1, s2);
  if (!lcs.empty()) {
     "Longest Common Substring: " << lcs;
  } else {
    "No common substring found.";
  }
}

3) Count Different Substring
int countDifferentSubstrings() {
  int totalSubstrings = 0;
  for (int i = 1; i < suffixAutomaton.size(); ++i) {
      totalSubstrings += suffixAutomaton[i].length -
          suffixAutomaton[suffixAutomaton[i].link].length;
  }return totalSubstrings;}

4) Count total Length Of Different Substrings
ll totalLengthOfDifferentSubstrings() {
  ll tot = 0;
  for(int i = 1; i < suffixAutomaton.size(); i++) {
    ll shortest = suffixAutomaton[suffixAutomaton[i].link].
        length + 1;
    ll longest = suffixAutomaton[i].length;
    ll num_strings = longest - shortest + 1;
    ll cur = num_strings * (longest + shortest) / 2;
    tot += cur;
  }return tot;}
```

6.3 Suffix Automata

```
struct SuffixAutomatonNode {
  unordered_map<char, int> next;
  int length; int link;
};
vector<SuffixAutomatonNode> suffixAutomaton;
int last;
void initialize() {
  SuffixAutomatonNode initialNode;
  initialNode.length = 0;
  initialNode.link = -1;
  suffixAutomaton.push_back(initialNode);
  last = 0;
}
void extendAutomaton(char c) {
  SuffixAutomatonNode newNode;
  newNode.length = suffixAutomaton[last].length + 1;
  newNode.link = -1;
  int current = last;
  while (current != -1 && suffixAutomaton[current].next.find
      (c) == suffixAutomaton[current].next.end()) {
    suffixAutomaton[current].next[c] = suffixAutomaton.size()
        ;
    current = suffixAutomaton[current].link;
  }
  if (current == -1) {
    newNode.link = 0;
  } else {
    int next = suffixAutomaton[current].next[c];
    if (suffixAutomaton[current].length + 1 ==
        suffixAutomaton[next].length) {
      newNode.link = next;
    } else {
      SuffixAutomatonNode cloneNode = suffixAutomaton[next];
      cloneNode.length = suffixAutomaton[current].length + 1;
      suffixAutomaton.push_back(cloneNode);
      int cloneIndex = suffixAutomaton.size() - 1;
      while (current != -1 && suffixAutomaton[current].next[c
          ] == next) {
        suffixAutomaton[current].next[c] = cloneIndex;
        current = suffixAutomaton[current].link;
      }
      newNode.link = cloneIndex;
      suffixAutomaton[next].link = cloneIndex;
    }
  }
  suffixAutomaton.push_back(newNode);
  last = suffixAutomaton.size() - 1;
}
int main() {
  string input = "abab";
  initialize();
  for (char c : input) {
    extendAutomaton(c);
  }
}
```

6.4 Extra

```
// max product of sum by Changing atmost one subarray
long long ans = sum;
for (int i = 0; i < n; i++){
  long long tmp = sum;
  for (int j = 1; j <= min(i, n - 1 - i); j++){
    tmp -= (a[i - j] * b[i - j]) - (a[i + j] * b[i + j]);
    tmp += (a[i - j] * b[i + j]) + (a[i + j] * b[i - j]);
    ans = max(ans, tmp);
  }
}
for (int i = 0; i < n - 1; i++){
  long long tmp = sum;
  for (int j = 1; j <= min(i + 1, n - 1 - i); j++){
    tmp -= (a[i - j + 1] * b[i - j + 1]) - (a[i + j] * b[i +
        j]);
    tmp += (a[i - j + 1] * b[i + j]) + (a[i + j] * b[i - j +
        1]);
    ans = max(ans, tmp);
  }
}
```

```
// 2D SubRectangle Summation
// When C(i, j) = A(i) * B(j);
vector<ll> a(n + 1), b(m + 1), L(n + 1), R(m + 1);
for (int i = 1; i <= n; i++) cin >> a[i], a[i] += a[i - 1];
for (int i = 1; i <= m; i++) cin >> b[i], b[i] += b[i - 1];
for (int i = 1; i <= n; i++){
  L[i] = MX + 1;
  for (int j = i; j <= n; j++) L[i] = min(L[i], a[j] - a[j -
        i]);
}
for (int i = 1; i <= m; i++){
  R[i] = MX + 1;
  for (int j = i; j <= m; j++) R[i] = min(R[i], b[j] - b[j -
        i]);
}
for (int i = 1; i <= n; i++){
  for (int j = 1; j <= m; j++)
    C[i][j] = L[i] * R[j] == S
}

// Min/Max sum using swap same index value
dp[1][0] = a[0] * mn[1];
dp[1][1] = a[0] * mx[1];
for(int i = 2; i < n-1; i++){
 dp[i][0] = min(dp[i-1][0] + mx[i-1] * mn[i], dp[i-1][1] +
      mn[i-1] * mn[i]);
 dp[i][1] = min(dp[i-1][0] + mx[i-1] * mx[i], dp[i-1][1] +
      mn[i-1] * mx[i]);
}
cout << min(dp[n-2][0] + mx[n-2] * a[n-1], dp[n-2][1] + mn[n
     -2] * a[n-1]) << endl;

// check total pair a(i)+a(j)=x && a(i)*a(j)=y
cin >> x >> y;
ll D = x * x - 4LL * y;
ll R = sqrt(D);
if (R * R != D){
  cout << 0 << ' ';
} else {
  ll a = (x + R) / 2, b = (x - R) / 2;
  if (a == b) cout << (mp[a] * (mp[a] - 1)) / 2 << ' ';
  else cout << mp[a] * mp[b] << ' ';
}

// Find Bridge undirected graph
long long n, m, timer, mn = 1e18; // number of nodes
int const N = 2e5 + 10;
vector<pair<int, int>> edges;
vector<int> vis, tin, low, adj[N];
void dfs(int v, int p = -1){
  vis[v] = true;
  tin[v] = low[v] = timer++;
  bool parent_skipped = false;
  for (int to : adj[v]) {
    if (to == p && !parent_skipped) {
      parent_skipped = true;
      continue;
    }
    if (vis[to]) {
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      if (low[to] > tin[v])
        edges.push_back({v, to});
    }
  }
}
void find_bridges(){
  timer = 0;
  vis.assign(n + 1, false);
  tin.assign(n + 1, -1);
  low.assign(n + 1, -1);
  for (int i = 1; i <= n; i++){
      if (!vis[i])
          dfs(i);
  }
}
Note: Also possible to find cycle using bridge + dfs +
     hashing + if need

// Find Next Smaller Element
Input: [4, 8, 5, 2, 25]
Output: [2, 5, 2, -1, -1]
vector<int> result(arr.size(), -1);
stack<int> st;
for (int i = 0; i < arr.size(); ++i) {
  while (!st.empty() && arr[i] < arr[st.top()]) {
    result[st.top()] = arr[i];st.pop();
  }st.push(i);
}

// Summation of 2D Segment tree;
arr[i][j]+=arr[i][j-1];
dp[i][j]=arr[i][j]+dp[i-1][j];
rh = row_upperLim, rl = row_lowerLim;
ch = column_upperLim, cl = column_lowerLim;
// Exampe Cell[(2,3) to (4, 7)]
// dp[4][7]-dp[1][7]-dp[4][2]+dp[1][2];
Algorithm : dp[rh][ch]-dp[rl-1][ch]-dp[rh][cl-1]+dp[rl-1][cl
     -1];

// Given Tow number a and b; find out how many integers 1 <=
      i <= a
// are there such that gcd(i, a) == b.
vector<ll> p, prime(1001);
for (int i = 2; i < 1001; i++){
  if (prime[i] == 0){
    p.push_back(i);
    for (int j = 2; i * j < 1001; j++)
        prime[i * j]= 1;
  }
}
cin >> a >> b;
if (a % b != 0){cout << 0 << " ";continue;}
ll n = a / b;
ll ans = n;
for (int j = 0; j < p.size() && p[j] * p[j] <= n; j++){
  if (n % p[j]== 0){
    ans -= (ans / p[j]);
    while (n % p[j]== 0){
     n /= p[j];}
  }
}
if (n > 1) ans -= (ans / n);
```

6.5 Z Algorithm

```
string str;
int Z[N];
void getZarr() {
  int n = str.length();
  int L, R, k;
  Z[0] = n;
  L = R = 0;
  for (int i = 1; i < n; ++i) {
    if (i > R) {L = R = i;
      while (R < n && str[R - L] == str[R]){R++;}
      Z[i] = R - L;R--;
    } else {k = i - L;
      if (Z[k] < R - i + 1){Z[i] = Z[k];}
      else {L = i;
      while (R < n && str[R - L] == str[R]){R++;}
Z[i] = R - L;R--;}}}}
```

6.6 Trie

```
struct TrieNode {
    TrieNode* child[26];
    bool wordEnd;

    TrieNode() {
        wordEnd = false;
        for (int i = 0; i < 26; i++) {
            child[i] = nullptr;
        }
    }
};
void insertKey(TrieNode* root, const string& key) {
    TrieNode* curr = root;
    for (char c : key) {
        if (curr->child[c - 'a'] == nullptr) {
            TrieNode* newNode = new TrieNode();
            curr->child[c - 'a'] = newNode;
        }
        curr = curr->child[c - 'a'];
    }
    curr->wordEnd = true;
}
bool searchKey(TrieNode* root, const string& key) {
    TrieNode* curr = root;
    for (char c : key) {
        if (curr->child[c - 'a'] == nullptr)
            return false;
        curr = curr->child[c - 'a'];
    }
    return curr->wordEnd;
}
bool isEmpty(TrieNode* root) {
    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root->children[i])
            return false;
    return true;
}
TrieNode* remove(TrieNode* root, string key, int depth = 0)
 {
    if (!root)
        return NULL;

    if (depth == key.size()) {
        if (root->isEndOfWord)
            root->isEndOfWord = false;

        if (isEmpty(root)) {
            delete (root);
            root = NULL;
        }
        return root;
    }

    int index = key[depth] - 'a';
    root->children[index] = remove(root->children[index], key
        , depth + 1);

    if (isEmpty(root) && root->isEndOfWord == false) {
        delete (root);
        root = NULL;
    }
    return root;
}
int main() {
    TrieNode* root = new TrieNode();
    vector<string> arr = {"and", "ant"};
    for (const string& s : arr) {
        insertKey(root, s);
    }
    vector<string> searchKeys = {"do", "gee"};
    for (string& s : searchKeys) {
        cout << "Key : " << s << "\n";
        if (searchKey(root, s))
            cout << "Present\n";
        else
            cout << "Not Present\n";
    }
    remove(root, "heroplane");
}
```

6.7 Manacher's Algorithm

```
// when i is odd or T[i] == '#', L[i] - 1 is even palindrome
    length
// when is even or T[i] == 'a-z', L[i] - 1 is odd palindrome
    length
// Palindrome Start index = (i - L[i]) / 2; [0 index]
// Palindrome End index = (i + L[i]) / 2 - 2; [0 index]
std::vector<int>mc(const std::string&s){
  std::string T="$#";
  for(char c:s){T+=c; T+="#";}
    int m=T.length(); T+='&';
    int maxRight=0,mid=0;
    std::vector<int>L(m);
    for(int i=1;i<m;++i){
      L[i]=i<maxRight?std::min(L[2*mid-i],maxRight-i):1;
      while(T[i-L[i]]==T[i+L[i]]){++L[i];}
      if(maxRight<i+L[i]){mid=i; maxRight=i+L[i];}
}return L;}
int main(){string s;cin>>s;
std::vector<int>L=mc(s);}
```

6.8 Lexi Kth Duplicate

```
// Kth Lexicographically String (Duplicates Counted)
string res;
struct state {
    int len, link;
    map<int, int> next;
};
const int MAXLEN = 100001;
state st[MAXLEN * 2];
ll dp[2*MAXLEN];
ll cnt[2*MAXLEN];
int sz, last;
set< pair<int,int> > cal;
/// First state is the first one ,so len is zero
/// There is no suffix link of first state ,so link is -1
/// last is obviously zero for first state
void sa_init() {
 for (int i = 0; i <=sz; i++)
  {st[i].next.clear();}
 sz = last = 1;
 st[0].len = 0;
 st[0].link = -1;///++sz;
}
void sa_extend(int c) {
  int cur = ++sz;/// increase the size of sz
  st[cur].len = st[last].len + 1;
  cnt[cur]=1;
  cal.insert(make_pair(st[cur].len,cur));
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
    st[p].next[c] = cur;
    p = st[p].link;
  }
  if (p == -1) {
      st[cur].link = 1;
  } else {
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
        st[cur].link = q;
    } else {
      int clone = ++sz;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      cnt[clone]=0;
```

```
      cal.insert(make_pair(st[clone].len,clone));
      while (p != -1 && st[p].next[c] == q) {
          st[p].next[c] = clone;
          p = st[p].link;
      }
      st[q].link = st[cur].link = clone;
    }
}
  last = cur;
}

void calcul(int v) {
  if (dp[v])return;
  dp[v] = cnt[v];
  for (map<int, int>::iterator it = st[v].next.begin(); it
       != st[v].next.end(); it++){
  calcul(it->second);
  dp[v] += dp[it->second];}
}
void precal(){
  set< pair<int,int> >::reverse_iterator it;
  for(it=cal.rbegin(); it!=cal.rend(); it++ ){
  cnt[ st[ it -> second ].link ] += cnt[ it->second ];}}
void find_kth_lexicographical(int k){
  int p=1;
  while(k){
    int a=0;
    while( k>dp[st[p].next[a]] && a<26 ){
      if (st[p].next[a]) k-=dp[st[p].next[a]];
       a++;
    }
    res+=('a'+a); p=st[p].next[a];
    if(k>=cnt[p]) {k=k-cnt[p];}
    else {break;}
  }}
int main(){
  int n,k;
  string s;
  cin>>s>>k;
  ll len=s.length();
  len=len*(len+1)>>1;
  if(len<k) {cout<<"No such line."<<endl; return 0;}
  s="#"+s;
  sa_init();
  for(int i=1;i<s.length();i++) sa_extend(s[i]-'a');
  precal();
  calcul(1);
  res="";
  find_kth_lexicographical(k);
  cout<<res<<endl;
}
```

6.9 Lexi Kth Unique

```
//Lexicographically kth Unique Substring
void sa_init() {
  for (int i = 0; i <=sz; i++)
    {st[i].next.clear();}
 sz = last = 1;
 st[0].len = 0;
 st[0].link = -1;///++sz;
}
/// when we add new character then we will check by last
     pointer to which label with our new character c
void sa_extend(int c) {
    int cur = ++sz;/// increase the size of sz
    st[cur].len = st[last].len + 1;/// it's obviously
         increasing by one as new charcter is adding
    int p = last;
    while (p != -1 && !st[p].next.count(c)) { /// finding
         that state from where a state remains with edge of
         character same character c
        st[p].next[c] = cur;
        p = st[p].link;
    }
    if (p == -1) { /// If there is no state with such
         transition(edge) with character c , at last p will
         stop at first state
        st[cur].link = 1;/// now start from 1 node ,because
             it is start level now
    } else { /// state with label c has been found
    int q = st[p].next[c]; /// the state which has been
         attached to p by edge c has been stated by q
    if (st[p].len + 1 == st[q].len) {
        st[cur].link = q;
    } else {
      int clone = ++sz;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      while (p != -1 && st[p].next[c] == q) {
          st[p].next[c] = clone;
          p = st[p].link;
      }
      st[q].link = st[cur].link = clone;
    }
  }last = cur;}
/// it is i.e. how many paths can be started from here + 1 (
     empty string)
/// So , when find kth string it will help us to find kth
     path as we know how many paths from here can be
     possible
void calcul(int v) {
  if (dp[v])return;
  dp[v] = 1;
  for (map<int, int>::iterator it = st[v].next.begin(); it
       != st[v].next.end(); it++){
    calcul(it->second);
    dp[v] += dp[it->second];
  }
}
void find_kth_lexicographical(int k){
  int p=1;/// our starting node now start from 1 because
       string start from 1 0th is #
  while(k){
    int a=0;
    while( k>dp[st[p].next[a]] && a<26 ){
        if (st[p].next[a]) k-=dp[st[p].next[a]];
         a++;/// it is increased till there is edge from p i.
              e. st[p].next[a] is not empty
    }
  /// it has been checked that st[p].next[a] is not empty i.
       e.
  /// if it is empty so dp[0] (=0) is less than k :) ,so
       automatically increased by while loop
  /// so , we can take this path into account because we are
        finding kth path
  /// The one which path count of dp[i] is greater than k ,
       subtract those paths from k ,new k will be found
  res+=('a'+a);k--;
  p=st[p].next[a];}}
```

7 Dynamic Programming

7.1 Digit DP

```
long long dp[20][180][2];
void getDigits(long long x, vector<int>& digit){
while (x) {digit.push_back(x % 10);x /= 10;}}
long long digitSum(int idx, int sum, int tight, vector<int>&
      digit) {
  if (idx == -1) return sum;
  if (dp[idx][sum][tight] != -1 && tight != 1) {
  return dp[idx][sum][tight];}
  long long ret = 0;
  int k = (tight) ? digit[idx] : 9;
  for (int i = 0; i <= k; i++){
```

```
    int newTight = (digit[idx] == i) ? tight : 0;
    ret += digitSum(idx - 1, sum + i, newTight, digit);}
  if (!tight){dp[idx][sum][tight] = ret;}
return ret;}
int rangeDigitSum(int a, int b) {
    memset(dp, -1, sizeof(dp));
    vector<int> digitA;
    getDigits(a - 1, digitA);
    long long ans1 = digitSum(digitA.size() - 1, 0, 1, digitA
        );
    vector<int> digitB; getDigits(b, digitB);
    long long ans2 = digitSum(digitB.size() - 1, 0, 1, digitB
        );
    return (ans2 - ans1);}
int main() {ll a = 0, b = 1000;ll ans = rangeDigitSum(a, b)
    ;}
```

7.2 Basic DP Types

```
// LCS
vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
for (int i = 1; i <= m; ++i) {
for (int j = 1; j <= n; ++j) {
if (s1[i - 1] == s2[j - 1]){dp[i][j] = dp[i - 1][j - 1] +
    1;}
else{dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);}
}}ll ans = dp[m][n];

// Edit distance
int prev; // Stores dp[i-1][j-1]
vector<int> curr(n + 1, 0);
for (int j = 0; j <= n; j++){
  curr[j] = j;}
for (int i = 1; i <= m; i++){
  prev = curr[0]; curr[0] = i;
  for (int j = 1; j <= n; j++){
    int temp = curr[j];
    if (s1[i - 1] == s2[j - 1]){curr[j] = prev;}
    else{curr[j] = 1 + min({curr[j - 1], prev, curr[j]});}
    prev = temp;
  }
}return curr[n];

// 0/1 knapsack
int N = weight.size();
int W = 10; //Knapsack capacity
vector<int> dp(W + 1, 0);
for (int i = 0; i < N; ++i) {
  for (int w = W; w >= weight[i]; --w) {
    dp[w]=max(dp[w], dp[w - weight[i]] + value[i]);
  }
}return dp[W];
```