LU_SNS

Leading University

- **FastIO**: ios::sync_with_stdio(false); cin.tie(0);
- **File Handeling:**

```
#ifndef ONLINE_JUDGE
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
freopen("error.txt", "w", stderr);
auto st = clock(); // Current time should be placed on the first line
cerr << "Time = " << 1.0 * (clock() - st) / CLOCKS_PER_SEC << "\n";
#endif
```

- **next_permutation():** It is used to rearrange the elements in the range [first, last) into the next lexicographically greater permutation. {{1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}};

```
int arr[] = {1, 2, 3};                    => O(n*n!)
do{
    //Add any conditions;
   cout << arr[0] << " " << arr[1] << " " << arr[2] << "\n";
} while (next_permutation(arr, arr + 3));
```

- Erase Duplicate value in sorted vector**: v.erase(unique(v.begin(), v.end()), v.end());**
- **Better then rand()** function:

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count()); // mt19937_64 (long long)
auto my_rand(long long l, long long r)  { // random value generator [l, r]
   return uniform_int_distribution<long long>(l, r)(rng);
}
```

- **merge():** Merge two sorted arrays using merge present algorithm header file. **The Arrays must be sorted.**
  => O(vec1.size() + vec2.size() **)**

```
merge(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), back_inserter(finalVec));
merge(st1.begin(), st1.end(), st2.begin(), st2.end(), inserter(st[node], st.begin()));
```

- **Policy based DS**: The complexity of the **insert** and **erase** functions is **O(log n)**.

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

template <typename T> using ordered_set = tree<T, null_type, less<T>,
rb_tree_tag, tree_order_statistics_node_update>;
template <typename T, typename R> using ordered_map = tree<T, R, less<T>,
rb_tree_tag, tree_order_statistics_node_update>;

// *s.find_by_order(k): K-th element in a set (counting from zero).
// s.order_of_key(k): Number of items strictly smaller than k. (same as, lower_bound of k)
// less_equal<T> => for ordered_multiset or, ordered_multimap.
ordered_set<int> s;  ordered_map<int, ll>mp;  // we can change the data type.
```

- **gp_hash_table<int, int>:** Same as unordered_map, but **faster** than unordered_map.

```
struct custom_hash {
     static uint64_t splitmix64(uint64_t x) {
          static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().count();
          x += FIXED_RANDOM;
          x += 0x9e3779b97f4a7c15;
          x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
          x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
          return x ^ (x >> 31);
```

```
        }
        size_t operator()(uint64_t x) const { // x key
            return splitmix64(x);
        }
        size_t operator()(pair<uint64_t, uint64_t> x) const { // For, key = pair
            return splitmix64(x.first) ^ splitmix64(x.second);
        }
 };
```

- **priority_queue<**int**>max_heapPQ;** => In this queue elements are in non-increasing. [same as **multiset <int, greater<int> > s; But Priority Queue is more faster.**]
- **priority_queue<**int, vector<int>, **greater<int>>min_heapPQ;** => In this queue elements are in non-decreasing order. [similar to **multiset, but Priority Queue is more faster.**]].

Math:

- p+(p+1)+...+(q - 1)+q = **(q + p)(q – p + 1) / 2;** [Ex: **7**+8+9+10+**11**=(11+7)(11-7+1) / 2 =45]
- 1+2+3+...+(n - 1)+n = **(n * (n + 1)) / 2**; [Ex: 1+2+3+4+5=(5*(5+1))/2 = 15]
- 1+3+5+...+ (2n - 3) + (2n - 1) = **N^2**; [**N-> number of size**] [Ex: 1+3+5 = 3^2 = 9]
- 2+4+6+...+ (2n - 2) + 2n **= N * (N + 1)**; [**N-> number of size**] [Ex: 2+4+6 = 3*(3+1) =12]
- $1^2+2^2+3^2+ . . .+ (n - 1)^2 + n^2$ = **n (n + 1) (2n + 1) / 6;** [Ex: 1+4+9 = 3(3+1)(2*3 +1)/6 =14]
- $1^3 + 2^3 + 3^3 + . . . + (n - 1)^3 + n^3$ = **$\{n (n + 1) / 2\}^2$;** [Ex: 1+8+27= $\{3(3+1) / 2\}^2$ =36]
- $1^2 + 3^2 + 5^2 + . . . + (2n - 3)^2 + (2n - 1)^2$ = **$N*(4N^2 - 1) / 3$;** [Ex: 1+9+25 = $3*(4*3^2 - 1)/3$ = 35]
- $1^3 + 3^3 + 5^3 + . . . + (2n - 3)^3 + (2n - 1)^3$ = **$N^2 (2N^2 - 1)$;** [Ex: 1+27+125 = $3^2 (2*3^2 - 1)$ = 153]
- $1^4 + 2^4 + 3^4 + . . . + (n – 1)^4 + n^4$ = **$n(n+1) (2n+1) (3n^2 + 3n – 1) / 30$;**
  [Ex: 1+16+81+256 = $4(4+1) (2*4+1) (3*4^2 + 3*4 – 1) / 30$ = 354]
- $c^a + c^{a+1} + \cdots + c^b$ = **$(c^{b+1} – c^a) / (c – 1)$;** [c != 1]
- $2^0 + 2^1 + 2^2 + 2^3 + . . . + 2^{(k - 1)}$ = **$2^k – 1$;** [Ex: 1+2+4+8+16+32 = 2^6 - 1 = 63]
- If F(n) = -1 + 2 - 3 + . . .+ $(-1)^n * n$
  - If N **even** number, **ans = N/2;**
  - If N **odd** number, **ans = ((N + 1) / 2) * (-1));**
- N-th **Odd** number = **(2 * N) – 1**;
- N-th **Even** number = **2*N**;
- $a + a*k + a*k^2 + ... + b$ = **((b * k) - a) / (k - 1).** [ex: 3 + 6 + 12 + 24 = ((24 * 2) -3) / (2-1) = 45]
- a + (a+4) + (a+2*4) + ... + b= **(n * (a + b)) / 2.** [n-> number of size]
  [ex: 3 + 7 + 11 + 15 = (4 * (3 + 15)) / 2 = 36.]
- Number of digits in N = **floor(log10(**N**)) + 1;**
- Number of trailing **zeros** in **N!** => **while(N) sum+=N/5, N/=5; [**Ex: 10! = 36288**00;]**
- For a grid of size **(N x N)** the total number of **squares** formed: **((n*(n+1)) * (2n+1)) / 6;**
- **5 minutes** Clock Angular Value is **30°**. **[1 min = 6°]**
- Angle between clock minute and hour, **ans = abs ((0.5 * 11 * m) – (30 * h));**
  - For smaller angle, **if (ans >180) ans = 360 – ans;**
- The number of ways of selecting one or more things from N different things is given by
  2^N -1. (combination)
- Number of possible of N bits = 2^N. [4bits, 24 =16 => 0 to 15 number possible with using 4 bits] $(2^n -1)$ → highest value.
- The number of possible **unique triplet** for an array of length **n** formula: **n * (n-1) * (n-2) / 6;**
- N = **2^x =>** x = **log2(N).** Ex: 64 = 2^6 [log2(64) = **6**].
- **$\log_u(x)$=** $\frac{logk(x)}{logk(u)}$ [k-> any base (2,10)]; **$\log_a(k)$=** $\frac{1}{logk(a)}$; **a^x = b ;=> x = $\log_a$ b;**
- **(A * B) = ((A % Mod) * (B % Mod)) % Mod;** **<=** [Same As **+,-** Operator]

- **(A / B) = ((A % Mod) * (BinExp(B , Mod-2) % Mod)) % Mod;**

Bits:

- **Bitwise NOT(~):** inverts all bits of it. [a = 1001_2 **-> (~a) = 0110**]
- (**N / 2**) **== (N >> 1);** (**N * 2**) == (**N << 1);**
- (**2^N**) **== (1**LL **<< N);** **=> N = (1LL <<** (long long)**log2**(N) **);**
- **is_power_of_two(val)** => **(val & (val - 1)) == 0;**
- **CheckBit(val, pos) => (val & (1LL << pos));**
- **SetBit(val, pos) => (val |= (1LL << pos));**
- **ClearBit(val, pos) => (val &= ~(1LL << pos));**
- **FlipBit(val, pos) => (val ^= ~(1 << pos));**
- **MSB(mask) => 63 – __builtin_clzll(mask);** [Most Significant Bit position]
- **LSB(mask) => __builtin_ctzll(mask);** [Least Significant Bit position]
- **__builtin_popcount(**x**):** This function is used to count the number of one's(set bits) in an integer(32 bits). Similarly you can use **__builtin_popcountll(**x**)** for **long long** data types (64 bits). Ex: x = **5** (101) => ans=2 ;

Bitset Function:

bitset< highest_Bit_number **> name(data);**

- **bitset<**64**>** b1(val); or**, bitset<**4**>**b2("1011"); => auto-convert to binary;
- **to_ulong():** Converts the contents of the **bitset** to an **unsigned long integer**; [Ex: b1 = 1001, int val = b1.**to_ulong**(); => val = 9;]
- **to_string():** Converts the contents of the **bitset** to a **string**; [Ex: b1 = 1001, s1 = b1.to_string(); => s1= "1001";]
- **count():** returns the total number of **set bits**(1); [Ex: b1=1001; bit= b1.count(); => bit =2;]

Combination(C):

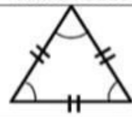
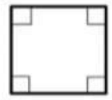
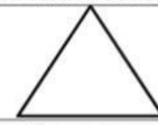
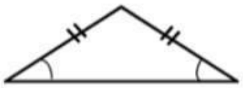
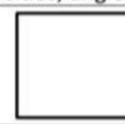
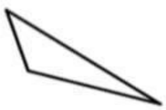
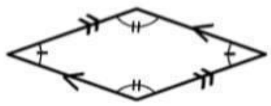
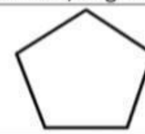
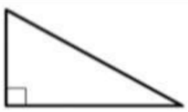
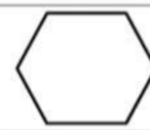
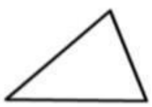
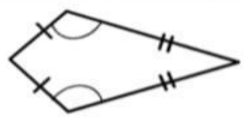
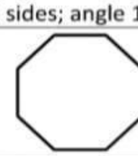
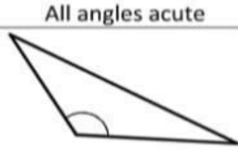
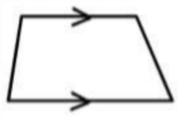
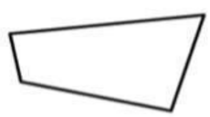
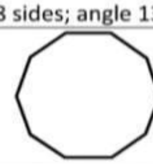
- If, **Order Doesn't Matter** and **Repetition Allowed** then, Possibilities**,** ${}^{n}C_{r} = \frac{n!}{r!(n-r)!}$
- If, **Order Doesn't Matter** and **Repetition Not Allowed** then, Possibilities, ${}^{n}C_{r} = \frac{(n+r-1)!}{r!(n-1)!}$

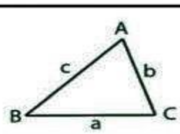
Permutation(P):

- If, **Order Matter** and **Repetition Allowed** then, Possibilities = n^r
- If, **Order Matter** and **Repetition Not Allowed** then, Possibilities = $\frac{n!}{(n-r)!}$

Geometry:

GEOMETRY QUICK GUIDE 2: 2D SHAPES (UK)

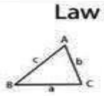
TRIANGLES	QUADRILATERALS	REGULAR POLYGONS
 **Equilateral triangle** All sides equal; interior angles 60°	 **Square** All sides equal; all angles 90°	 **Equilateral triangle** 3 sides; angle 60°
 **Isosceles triangle** 2 sides equal; 2 congruent angles	 **Rectangle** Opposite sides equal, all angles 90°	 **Square** 4 sides; angle 90°
 **Scalene triangle** No sides or angles equal	 **Rhombus** All sides equal; 2 pairs of parallel lines; opposite angles equal	 **Regular Pentagon** 5 sides; angle 108°
 **Right triangle** 1 right angle	 **Parallelogram** Opposite sides equal, 2 pairs of parallel lines	 **Regular Hexagon** 6 sides; angle 120°
 **Acute triangle** All angles acute	 **Kite** Adjacent sides equal; 2 congruent angles	 **Regular Octagon** 8 sides; angle 135°
 **Obtuse triangle** 1 obtuse angle	 **Trapezium** 1 pair of parallel sides  **Trapezoid** No pairs of parallel sides	 **Regular Decagon** 10 sides; angle 144°



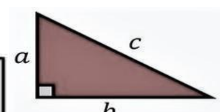
Law of sines

$$\frac{\sin(A)}{a} = \frac{\sin(B)}{b} = \frac{\sin(C)}{c}$$

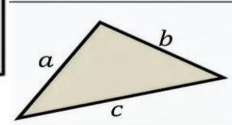
Law of Cosines



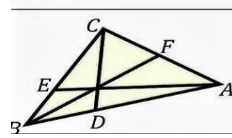
$$c^2 = a^2 + b^2 - 2ab\cos(C)$$
$$a^2 = b^2 + c^2 - 2bc\cos(A)$$
$$b^2 = a^2 + c^2 - 2ac\cos(B)$$



Pythagoras' Theorem
$a^2 + b^2 = c^2$



Heron's Formula
$Area = \sqrt{s(s-a)(s-b)(s-c)}$
$Semiperimeter, s = \dfrac{a+b+c}{2}$

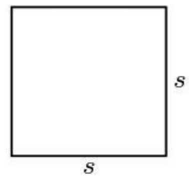


Ceva's Theorem
$Given\ AE, BF\ \&\ CD\ concurrent,$
$\dfrac{AD}{BD} \times \dfrac{BE}{CE} \times \dfrac{CF}{AF} = 1$

LU_SNS
Leading University

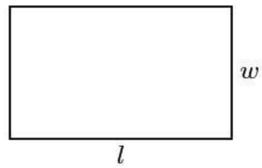
SQUARE

s = side
Area: $A = s^2$
Perimeter: $P = 4s$



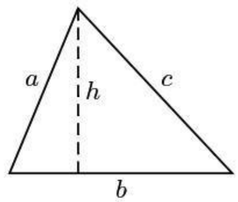
RECTANGLE

l = length, w = width
Area: $A = lw$
Perimeter: $P = 2l + 2w$



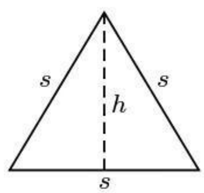
TRIANGLE

b = base, h = height
Area: $A = \frac{1}{2}bh$
Perimeter: $P = a + b + c$



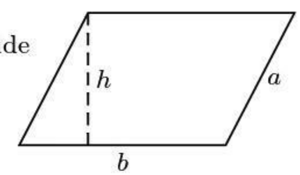
EQUILATERAL TRIANGLE

s = side
Height: $h = \frac{\sqrt{3}}{2}s$
Area: $A = \frac{\sqrt{3}}{4}s^2$



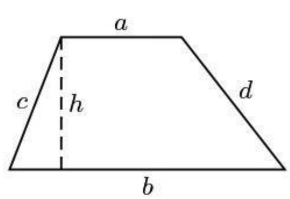
PARALLELOGRAM

b = base, h = height, a = side
Area: $A = bh$
Perimeter: $P = 2a + 2b$



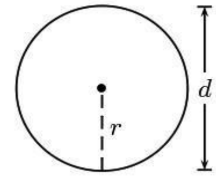
TRAPEZOID

a, b = bases; h = height;
c, d = sides
Area: $A = \frac{1}{2}(a + b)h$
Perimeter:
$P = a + b + c + d$



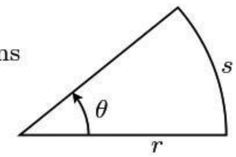
CIRCLE

r = radius, d = diameter
Diameter: $d = 2r$
Area: $A = \pi r^2$
Circumference: $C = 2\pi r = \pi d$



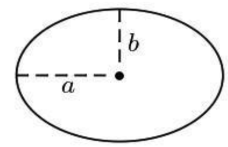
SECTOR OF CIRCLE

r = radius, θ = angle in radians
Area: $A = \frac{1}{2}\theta r^2$
Arc Length: $s = \theta r$



ELLIPSE

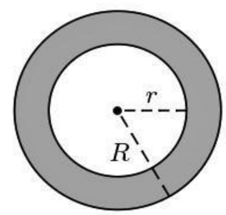
a = semimajor axis
b = semiminor axis
Area: $A = \pi ab$



Circumference:
$C \approx \pi\left(3(a + b) - \sqrt{(a + 3b)(b + 3a)}\right)$

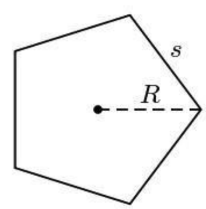
ANNULUS

r = inner radius,
R = outer radius
Average Radius: $\rho = \frac{1}{2}(r + R)$
Width: $w = R - r$
Area: $A = \pi(R^2 - r^2)$
or $A = 2\pi\rho w$



REGULAR POLYGON

s = side length,
n = number of sides
Circumradius: $R = \frac{1}{2}s\csc(\frac{\pi}{n})$
Area: $A = \frac{1}{4}ns^2\cot(\frac{\pi}{n})$
or $A = \frac{1}{2}nR^2\sin(\frac{2\pi}{n})$

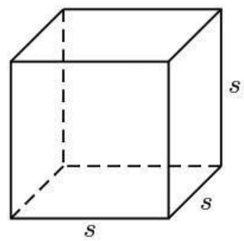


Rhombus: **Area =** (d1 * d2) / 2 **=** s^2 * sin(C)**;** **Perimeter =** 4*s ;
Kite: **Area** = (d1 * d2) / 2; **Perimeter =** 2(s1 + s2);
[d1 and d2= lengths of the diagonals, s = s1= s2 = length of side, C = interior angle;**]**

3D GEOMETRY FORMULAS

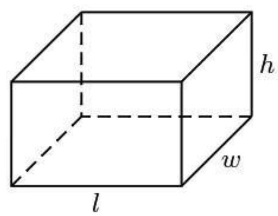
CUBE

s = side
Volume: $V = s^3$
Surface Area: $S = 6s^2$



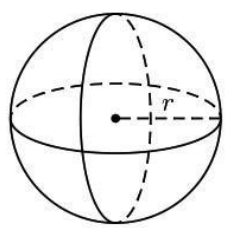
RECTANGULAR SOLID

l = length, w = width,
h = height
Volume: $V = lwh$
Surface Area:
$S = 2lw + 2lh + 2wh$



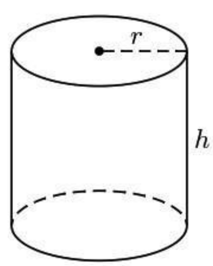
SPHERE

r = radius
Volume: $V = \frac{4}{3}\pi r^3$
Surface Area: $S = 4\pi r^2$



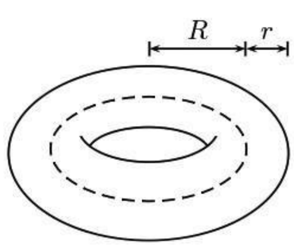
RIGHT CIRCULAR CYLINDER

r = radius, h = height
Volume: $V = \pi r^2 h$
Surface Area: $S = 2\pi rh + 2\pi r^2$



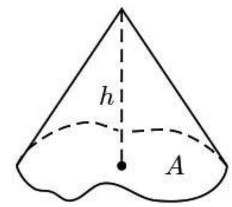
TORUS

r = tube radius,
R = torus radius
Volume: $V = 2\pi^2 r^2 R$
Surface Area: $S = 4\pi^2 rR$



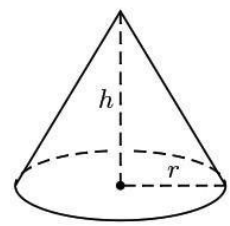
GENERAL CONE OR PYRAMID

A = area of base, h = height
Volume: $V = \frac{1}{3}Ah$



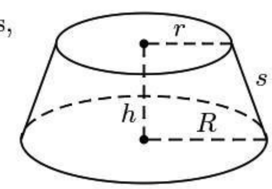
RIGHT CIRCULAR CONE

r = radius, h = height
Volume: $V = \frac{1}{3}\pi r^2 h$
Surface Area:
$S = \pi r\sqrt{r^2 + h^2} + \pi r^2$



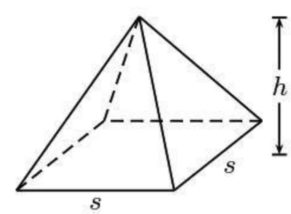
FRUSTUM OF A CONE

r = top radius, R = base radius,
h = height, s = slant height
Volume: $V = \frac{\pi}{3}(r^2 + rR + R^2)h$
Surface Area:
$S = \pi s(R + r) + \pi r^2 + \pi R^2$



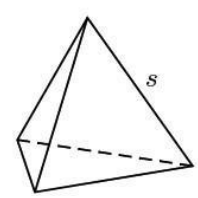
SQUARE PYRAMID

s = side, h = height
Volume: $V = \frac{1}{3}s^2 h$
Surface Area:
$S = s(s + \sqrt{s^2 + 4h^2})$



REGULAR TETRAHEDRON

s = side
Volume: $V = \frac{1}{12}\sqrt{2}s^3$
Surface Area: $S = \sqrt{3}s^2$



- **pi** = 2 * acos(0.0);
- Convert **Radian** to **Degree**: **sin(**val * **(pi / 180.0));** **asin(**val**) * (180.0 / pi);**

String:

Double Hashing:

```
const int N = 1e6 + 5;
const int Base1 = 137, Base2 = 277;
const int mod1 = 127657753, mod2 = 987654319;

bool isCalPow = 0;
pair<ll, ll> po[N];
void generatePower() // Storing the power of the Base.
{
  po[0].first = 1, po[0].second = 1;
  for (int i = 1; i < N; i++) {
    po[i].first = (po[i - 1].first * Base1) % mod1;
    po[i].second = (po[i - 1].second * Base2) % mod2;
  }
}
struct Hashing {
  vector<pair<ll, ll>> prefix, suffix;
  int n;
  void generatePrefixHash(string &s) {
    prefix[0].first = s[0], prefix[0].second = s[0];
    for (int i = 1; i < s.size(); i++) {
      prefix[i].first = ((prefix[i - 1].first * Base1) + s[i]) % mod1;
      prefix[i].second = ((prefix[i - 1].second * Base2) + s[i]) % mod2;
    }
  }
  void generateSuffixHash(string &s) {
    suffix[n - 1].first = s[n - 1], suffix[n - 1].second = s[n - 1];
    for (int i = n - 2; i >= 0; i--) {
      suffix[i].first = ((suffix[i + 1].first * Base1) + s[i]) % mod1;
      suffix[i].second = ((suffix[i + 1].second * Base2) + s[i]) % mod2;
    }
  }
  pair<ll, ll> generateHash(string &s) // return hash value of a string
  {
    pair<ll, ll> H = {0, 0};
    for (auto &c : s)  {
      H.first = ((H.first * Base1) + c) % mod1;
      H.second = ((H.second * Base2) + c) % mod2;
    }
    return H;
  }
  pair<ll, ll> getPrefixRangeHash(int l, int r) // return hash value of a range
  {
    if (l == 0) return prefix[r];
    pair<ll, ll> Hs;
    Hs.first = (prefix[r].first - (prefix[l - 1].first * po[r - l + 1].first % mod1) + mod1) % mod1;
    Hs.second = (prefix[r].second - (prefix[l - 1].second * po[r - l + 1].second % mod2) + mod2) % mod2;
    return Hs;
  }
  pair<ll, ll> getSuffixRangeHash(int l, int r) // return hash value of a range
  {
    if (r == n - 1) return suffix[l];
    pair<ll, ll> Hs;
    Hs.first = (suffix[l].first - (suffix[r + 1].first * po[r - l + 1].first % mod1) + mod1) % mod1;
```

```
    Hs.second = (suffix[l].second - (suffix[r + 1].second * po[r - l + 1].second % mod2) + mod2) % mod2;
    return Hs;
  }
  pair<ll, ll> concat(pair<ll, ll> &hash1, pair<ll, ll> &hash2, int len) //len = 2nd string size
  {
    return {((hash1.first * po[len].first) + hash2.first) % mod1, ((hash1.second * po[len].second) +
                                                                   hash2.second) % mod2};
  }
  void build(string &s)  {
    n = s.size();
    prefix.resize(n), suffix.resize(n);
    generatePrefixHash(s);
    // generateSuffixHash(s);
    if (!isCalPow) generatePower(), isCalPow = 1;
  }
} Hash;

void solve() {
  int n, m;
  string s1, s2;
  s1 = "abcabababc", s2 = "abc";
  // cin >> s1 >> s2;
  n = s1.size();
  Hash.build(s1);

  pair<ll, ll> hashOfS2 = Hash.generateHash(s2);
  for (int i = 0; i + s2.size() <= s1.size(); i++) {
    if (Hash.getPrefixRangeHash(i, i + s2.size() - 1) == hashOfS2) {
      cout << i << "\n";
    }
  }
  return;
}
```

String Hashing With Updates and Reverse:

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

int power(long long n, long long k, const int mod) {
 int ans = 1 % mod;
 n %= mod;
 if (n < 0) n += mod;
 while (k) {
  if (k & 1) ans = (long long) ans * n % mod;
  n = (long long) n * n % mod;
  k >>= 1;
 }
 return ans;
}

using T = array<int, 2>;
const T MOD = {127657753, 987654319};
const T p = {137, 277};
```

```
T operator + (T a, int x) {return {(a[0] + x) % MOD[0], (a[1] + x) % MOD[1]};}
T operator - (T a, int x) {return {(a[0] - x + MOD[0]) % MOD[0], (a[1] - x + MOD[1]) % MOD[1]};}
T operator * (T a, int x) {return {(int)((long long) a[0] * x % MOD[0]), (int)((long long) a[1] * x %
MOD[1])};}
T operator + (T a, T x) {return {(a[0] + x[0]) % MOD[0], (a[1] + x[1]) % MOD[1]};}
T operator - (T a, T x) {return {(a[0] - x[0] + MOD[0]) % MOD[0], (a[1] - x[1] + MOD[1]) %
MOD[1]};}
T operator * (T a, T x) {return {(int)((long long) a[0] * x[0] % MOD[0]), (int)((long long) a[1] * x[1]
% MOD[1])};}
ostream& operator << (ostream& os, T hash) {return os << "(" << hash[0] << ", " << hash[1] << ")";}

T pw[N], ipw[N];
void prec() {
pw[0] =  {1, 1};
  for (int i = 1; i < N; i++) {
    pw[i] = pw[i - 1] * p;
  }
  ipw[0] =  {1, 1};
  T ip = {power(p[0], MOD[0] - 2, MOD[0]), power(p[1], MOD[1] - 2, MOD[1])};
  for (int i = 1; i < N; i++) {
    ipw[i] = ipw[i - 1] * ip;
  }
}
struct Hashing {
  int n;
  string s; // 1 - indexed
  vector<array<T, 2>> t; // (normal, rev) hash
  array<T, 2> merge(array<T, 2> l, array<T, 2> r) {
  l[0] = l[0] + r[0];
    l[1] = l[1] + r[1];
    return l;
  }
  void build(int node, int b, int e) {
    if (b == e) {
      t[node][0] = pw[b] * s[b];
      t[node][1] = pw[n - b + 1] * s[b];
      return;
    }
    int mid = (b + e) >> 1, l = node << 1, r = l | 1;
    build(l, b, mid);
    build(r, mid + 1, e);
    t[node] = merge(t[l], t[r]);
  }
  void upd(int node, int b, int e, int i, char x) {
    if (b > i || e < i) return;
    if (b == e && b == i) {
      t[node][0] = pw[b] * x;
      t[node][1] = pw[n - b + 1] * x;
      return;
    }
```

```
    int mid = (b + e) >> 1, l = node << 1, r = l | 1;
    upd(l, b, mid, i, x);
    upd(r, mid + 1, e, i, x);
    t[node] = merge(t[l], t[r]);
  }
  array<T, 2> query(int node, int b, int e, int i, int j) {
    if (b > j || e < i) return {T({0, 0}), T({0, 0})};
    if (b >= i && e <= j) return t[node];
    int mid = (b + e) >> 1, l = node << 1, r = l | 1;
    return merge(query(l, b, mid, i, j), query(r, mid + 1, e, i, j));
  }
  Hashing() {}
  Hashing(string _s) {
  n = _s.size();
    s = "." + _s;
    t.resize(4 * n + 1);
    build(1, 1, n);
  }
  void upd(int i, char c) {
    upd(1, 1, n, i, c);
    s[i] = c;
  }
  T get_hash(int l, int r) { // 1 - indexed
    return query(1, 1, n, l, r)[0] * ipw[l - 1];
  }
  T rev_hash(int l, int r) { // 1 - indexed
    return query(1, 1, n, l, r)[1] * ipw[n - r];
  }
  T get_hash() {
    return get_hash(1, n);
  }
  bool is_palindrome(int l, int r) {
    return get_hash(l, r) == rev_hash(l, r);
  }
};
```

Number Theory

- **gcd():** Return a and b gcd(**Greatest Common Divisor**) value. => O(logn)
  [int gcd**= __gcd(a,b);** gcd(m*a, m*b) = m*gcd(a, b)**;** gcd(a/d, b/d) = gcd(a, b)/d**;**]
- **lcm()**: Return a and b lcm(**Least Common Multiple**) value. => O(logn)
  [int lcm**= (a*b)/__gcd(a,b);** lcm(m*a, m*b) = m*lcm(a, b)**;**]
  - lcm(a, b, c) = $\frac{abc * gcd(a,b,c)}{gcd(a,b) * gcd(a,c) * gcd(b,c)}$.

Extended Euclid: //=> **O(log(min(a, b)))**

// For this Eq. **(a*x) + (b*y) = gcd(a, b);**

```
ll extended_euclid(ll a, ll b, ll &x, ll &y) {
   if (b == 0) {
      x = 1, y = 0;
      return a;
   }
   ll x1, y1;
   ll gcd = extended_euclid(b, a % b, x1, y1);
   x = y1,  y = x1 - y1 * (a / b);
   return gcd;
}
```

Sum and Count of Divisor: **=>O(sqrt(n))**

Ex(sum): 20 => 22 (1+2+4+5+10+20).

Ex(count): 20 => 6 (1,2,4,5,10,20).

```
 void divisor()   {
   ll n, sum = 0, i, c = 0;
   cin >> n;
   vector<ll> divisors;
   for (i = 1; i * i <= n; i++) {
      if (n % i == 0) {
         sum += i, ++c;
         divisors.push_back(i);
         if (i != n / i)
            sum += n / i, ++c, divisors.push_back(n /
i);
      }
   }
   cout<<"Count = "<< c <<", Sum = "<< sum
<<endl;
   sort(divisors.begin(), divisors.end());
   for(auto &i: divisors) cout << i << " ";
}

int numberOfDivisors(LL n) {
   int sz = primes.size(), cnt = 1;

   for(int i = 0; i < sz && primes[i] * primes[i] <= n;
++i) {
      if(n % primes[i] == 0) {
         int pw = 0;
         while(n % primes[i] == 0) {
            ++pw;
            n /= primes[i];
         }
         cnt = pw + 1;
      }
   }
   if(n != 1) cnt <<= 1;
   return cnt;
}
```

Number of divisors: **=> O(nlog(n))**

Ex: 32 => 2 4 8 16 32

```
const int N = 1e5 + 10;
vector<int> divisor[N];
int main() {
   for (int i = 2; i < N; i++) {
      for (int j = i; j < N; j += i)
         divisor[j].push_back(i);
   }
   int n;  cin >> n;
   for (auto &it : divisor[n]) cout << it << " ";
   cout << endl;
}
```

Bitwise Sieve Algorithm (find prime number):
=> O(nloglogn)

```
const int N = 1e8 + 7;
int marked[N / 64 + 2];

#define on(x) (marked[x / 64] & (1 << ((x % 64)
/ 2)))
#define mark(x) marked[x / 64] |= (1 << ((x %
64) / 2))

void sieve() {
   for (int i = 3; i * i < N; i += 2) {
      if (!on(i)) {
         for (int j = i * i; j <= N; j += i + i) {
            mark(j);
         }
      }
   }
}
bool isPrime(int num) {
   return num > 1 && (num == 2 || ((num & 1) &&
!on(num)));
}
```

Sieve Algorithm (find prime number):
=> O(nloglogn)

```
const int N = 1e7 + 10;
bool marked[N];
```

```
void sieve() {
   for (int i = 3; i * i < N; i += 2) {
      if (marked[i] == false) // i is a prime {
         for (int j = i * i; j < N; j += i + i) {
            marked[j] = true;
         }
      }
   }
}
bool isPrime(int n) {
   if (n < 2) return false;
   if (n == 2) return true;
   if (n % 2 == 0) return false;
   return marked[n] == false;
}
```

Prime Factorization (Integer factorization):

=> O(sqrt(n))

Ex: 36 => 2 2 3 3

```
int main() {
   int n;
   cin >> n;
   vector<int> prime_factors;
   for (int i = 2; i * i <= n; i++) {
      while (n % i == 0) {
         prime_factors.push_back(i);
         n /= i;
      }
   }
   if (n > 1) prime_factors.push_back(n);
   for (auto &prime : prime_factors)
      cout << prime << " ";
}
```

Prime Factorization using Sieve algorithm:

=> O(log(n))

Ex: 50 => 2 5 5

```
vector<int> spf(N); // SPF : smallest prime factor
void sieve()          // => O( nloglogn)
{
   for (int i = 1; i < N; i++)  spf[i] = i;
   for (int i = 2; i * i < N; i++) {
      if (spf[i] == i) {
         for (int j = i * i; j < N; j += i)
            if (spf[j] == j) spf[j] = i;
      }
   }
}
int main() {
   sieve();
   int n;
   cin >> n;
   while (n != 1) {
      cout << spf[n] << " ";
      n /= spf[n];
   }
}
```

Binary Exponentiation using Iterative method:

=> O(log(b)).

Ex: $3^{13} => 3^{(8+4+0+1)} => 3^8 * 3^4 * 3^0 * 3^1$ => 1594323;

→(a^b)

```
const int Mod = 1e9 + 7;
long long BinExpIter(ll a, ll b) {
   ll ans = 1;
   while (b) {
      if (b & 1) ans = (ans * a) % Mod;
      a = (a * a) % Mod;
      b >>= 1;
   }
   return ans;
}
```

Binary Exponentiation for $N^{1/x}$:

=> O(x*log(N*10^d))

$3^{1/5}$ **=** 1.2457312346;

```
double eps = 1e-6;   // eps=1e-d; =>with d decimal accuracy
double BinExpPow (double n, int x) {
   double l = 0, r = n, m = (l + r) / 2;
   while (r - l > eps) {
      if (pow(m, x) > n) r = m;
      else l = m;
      m = (l + r) / 2;
   }
   return m;
}
```

Euler Totient Function:

```
// Find the co-prime between(1 to i);
// Time Complexity: O(NloglogN)
const int N = 1e6 + 7;
int coprimeCnt[N];
ll coprimeSum[N];

void generatePhi() {
   for (int i = 0; i < N; ++i) coprimeCnt[i] = i;
   for (int i = 2; i < N; i++) {
      if (coprimeCnt[i] == i) {
         for (int j = i; j < N; j += i)
            coprimeCnt[j] -= coprimeCnt[j] / i;
      }
   }
   // Sum of all coprime values (Ex: 10 => 1 + 3 + 7 + 9 = 20)
   coprimeSum[1] = 1;
   for (ll i = 2; i < N; ++i)
         coprimeSum[i] = (i * coprimeCnt[i]) >> 1;
}
```

Find the co-prime between(1 to i) => O(sqrt(n))

```
ll phi(ll n) {
   ll phiN = n;
   for(int i = 2; i * i <= n; i++) {
      if(n % i == 0) {
         phiN = phiN * (i - 1) / i; // for unique prime
         while (n % i == 0) n /= i;
      }
   }
   if(n > 1) phiN = phiN * (n - 1) / n;
   return phiN;
}
```

Find Combination(nCr): => O(r*log(n))

Ex: 5C2 = 10, 13C5= 1287;

```
void nCr( ll n, ll r) {
   ll p= 1, k=1, m;
   if (n – r < r)  r = n - r;
   if (r != 0) {
      while(r) {
         p*=n, k*=r;
         m=__gcd(p, k);
         p/=m, k/=m;
         n--, r--;
      }
   }
   else p=1;
   cout << p << endl;
}
```

Find Permutation (nPr): => O(n)

Ex: 5P2= 20, 6P3= 120;

```
ll fact(ll n) {
     if(n <= 1) return 1;
     return n * fact(n - 1);
}
ll nPr(ll n, ll r) {
     return fact(n) / fact(n - r);
}
// nCr and nPr using Modulo
const int Max = 2e5 + 5, mod = 998244353;
ll fact[Max], factInv[Max];
void build_fact() {
   fact[0] = 1;
   for(int i = 1; i < Max; i++) {
      fact[i] = 1LL * fact[i - 1] * i % mod;
   }
   factInv[Max - 1] = Pow(fact[Max - 1], mod - 2);
   for(int i = Max - 2; i >= 0; i--) {
      factInv[i] = 1LL * factInv[i + 1] * (i + 1) %
mod;
   }
   return;
}
int nCr_mod(int n, int r) {
   if(n < r or n < 0 or r < 0) return 0;
   return 1LL * fact[n] * factInv[r] % mod *
factInv[n - r] % mod;
}
int nPr_mod(int n, int r) // nPr = nCr * r!
{
   if(n < r or n < 0 or r < 0) return 0;
   return (1LL * nCr_mod(n, r) * fact[r]) % mod;
}
```

Principle of Inclusion and Exclution:

```
void solve()
{
   ll n, m;
   cin >> n >> m;
   vector<int> v(m);
   for (int i = 0; i < m; i++)
   {
      cin >> v[i];
      if (v[i] == 1)
      {
         cout << 0 << endl;
         return;
      }
   }
   long long ans = 0;
   for (int i = 1; i < (1 << m); i++) // loop from 1 to
2^m
   {
      vector<int> subset;
      int cnt = 0;
      for (int j = 0; j < m; j++) // loop through
binary representation of number(1 to 2^n)
      {
         if (i & (1 << j)) // checking ith bit is set(1) or
not
         {
            subset.push_back(v[j]);
            cnt++;
         }
      }
      int NumOfDiv, lcm = 1;
      for (auto it : subset) lcm = lcm * it / (gcd(lcm,
it));
      NumOfDiv = n / lcm;
     if (cnt & 1) // principle of inclusion and
exclution(A U B U C = n(A) + n(B) + n(C)-n(AUB)-
n(AUC)-
       n(BUC)+n(AUBUC));
          ans += NumOfDiv;
      else ans -= NumOfDiv;
   }
   cout << n - ans << endl;
}
```

// Modula Inverse using Extended Euclid (it does not matter mod is prime or not)

```
#define x first
#define y second
```

```
pair<ll, ll> extendedEuclid(ll a, ll b) // returns x, y;
ax + by = gcd(a,b)
{
   if (b == 0) return {1, 0};
   else
   {
      pair<ll, ll> d = extendedEuclid(b, a % b);
      return {d.y, d.x - d.y * (a / b)};
   }
}
ll modularInverseEE(ll a, ll Mod)
{
   pair<ll, ll> ret = extendedEuclid(a, Mod);
   return ((ret.x % Mod) + Mod) % Mod;
}

ll modularInverseFL(ll A, ll B) // (A / B) % mod
{
   ll inverse = ((A % mod) * (Pow(B, mod - 2) %
mod)) % mod; // (A * B^-1) % mod
   return (inverse + mod) % mod;
}
```

Segment Tree:

```
const int N = 3e5 + 9; // start
int a[N];
int tree[4 * N];
void build(int node, int st, int en)    //=> O(N)
{
   if (st == en)
   {
      tree[node] = a[st];
      return;
   }
   int mid = (st + en) / 2;
   build(2 * node, st, mid);
   build(2 * node + 1, mid + 1, en);
   tree[node] = tree[2 * node] + tree[2 * node + 1];
}
int query(int node, int st, int en, int l, int r)
 //=> O(logn)
{
   if (st> r || en< l) {
      return 0;
   }
   if (l <= st && en<= r) {
      return tree[node];
   }
   int mid = (st + en) / 2;
   int q1 = query(2 * node, st, mid, l, r);
   int q2 = query(2 * node + 1, mid + 1, en, l, r);
   return q1 + q2;
}
void update(int node, int st, int en, int idx, int val)
//=> O(logn)
{
   if (st == en) {
      a[st] = val;
      tree[node] = val;
      return;
   }
   int mid = (st + en) / 2;
   int left = 2 * node, right = 2 * node + 1;
   if (idx<= mid) update(left, st, mid, idx, val);
   else update(right, mid + 1, en, idx, val);
   tree[node] = tree[left] + tree[right];
}  // end
```

Lazy segment tree: **// start**

```
const int N = 5e5 + 9;
int a[N];
struct ST {
   #define lc (n << 1)
   #define rc ((n << 1) | 1)
   ll t[4 * N], lazy[4 * N];
   ST() {
      for (int i = 0; i< 4 * N; i++)
         t[i] = lazy[i] = 0;
   }
   inline void push(int n, int st, int en)
   {
      if (lazy[n] == 0) return;
      t[n] = t[n] + lazy[n] * (en - st + 1);
      if (st != en) {
         lazy[lc] = lazy[lc] + lazy[n];
         lazy[rc] = lazy[rc] + lazy[n];
      }
      lazy[n] = 0;
   }
   inline void pull(int n) {
      t[n] = t[lc] + t[rc];
   }
   void build(int n, int st, int en) {
      lazy[n] = 0;
      if (st == en) {
         t[n] = a[st];
         return;
      }
      int mid = (st + en) >> 1;
      build(lc, st, mid);
      build(rc, mid + 1, en);
      pull(n);
   }
   void update(int n, int st, int en, int l, int r, ll v)
   {
      push(n, st, en);  // push the value left and
                           right child
      if (r <st || en< l) return;
```

```
    if (l <= st && en<= r) {
      lazy[n] = v; // set lazy
      push(n, st, en);
      return;
    }
    int mid = (st + en) >> 1;
    update(lc, st, mid, l, r, v);
    update(rc, mid + 1, en, l, r, v);
    pull(n);
  }
  inline ll combine(ll a, ll b) {
    return a + b;
  }
  ll query(int n, int st, int en, int l, int r) {
    push(n, st, en);
    if (l >en || st> r) return 0; // return null
    if (l <= st && en<= r) return t[n];
    int mid = (st + en) >> 1;

     return combine(query(lc, st, mid, l, r),
                query(rc, mid + 1, en, l, r));
  }
} st; // end lazy
```

Binary Indexed tree(BIT):

```
/* 1'base indexing */ start
const int N = 3e5 + 9;
ll bit1[N];
ll bit2[N];
ll n;
void update(lli, ll x, ll *bit)            // O(logn)
{
while (i < N) {
  bit[i] += x;
  i += (i & (-i));
}
}
ll query(ll i, ll *bit)            // O(logn)
{
  ll sum = 0;
  while (i> 0) {
    sum += bit[i];
    i -= (i & (-i));
  }
  return sum;
}
void rupdate(ll l, ll r, ll val) {
   update(l, val, bit1);
   update(r + 1, -val, bit1);

   update(l, val * (l - 1), bit2);
   update(r + 1, -val * r, bit2);
}
ll rquery(ll l, ll r) {
ll sum1 = query(r, bit1) * r - query(r, bit2);
ll sum2 = query(l-1, bit1) * (l-1) - query(l-1, bit2);
return sum1 - sum2;
} // end
```

Mo's offline Query:

```
//=> O((N+Q)*sqrt(N))

const int N = 1e6 + 10;
int rootN;
struct Q {
  int l, r, idx;
};
Q q[N];
bool comp(Q q1, Q q2) {
  if (q1.l / rootN == q2.l / rootN) return q1.r >
q2.r;
  return q1.l / rootN< q2.l / rootN;
}
int main() {
  int n;
  cin>> n;
  int a[n];
  for (int i = 0; i< n; ++i) cin>> a[i];
  int query;
  cin>> query;
  rootN = sqrtl(n) + 1;
  for (int i = 0; i< query; ++i) {
    int l, r;
    cin>> l >> r;
    q[i].l = l;
    q[i].r = r;
    q[i].idx = i;
  }
  sort(q, q + query, comp);
  int curr_l = 0, curr_r = -1, l, r;
  llcurr_ans = 0;
  vector<ll>ans(query);
  for (int i = 0; i< query; i++) {
    l = q[i].l, r = q[i].r;
    --l, --r;
    while (curr_r< r) {
      ++curr_r;
      curr_ans += a[curr_r];
    }
    while (curr_l> l) {
      --curr_l;
      curr_ans += a[curr_l];
    }


    while (curr_l< l) {
      ++curr_l;
      curr_ans -= a[curr_l];
```

```
    }
    while (curr_r> r) {
      --curr_r;
      curr_ans -= a[curr_r];
    }
    ans[q[i].idx] = curr_ans;
  }
  for (int i = 0; i< query; i++) {
    cout<<ans[i] <<endl;
  }
  return 0;
} // end
```

Disjoint Set Union(DSU): // =>O(1)

Applications:1) Cycle detection. 2) Connected Components in graph. 3) MST(Minimum Spanning Tree).

```
const int N = 1e5 + 10;
int par[N];
int Size[N];

// returns the representative of the set that
contains the element v
int Find(int v) {
  if (par[v] == v) return v;
  return par[v] = Find(par[v]);
  // Path Compression
}

// merges the two specified sets(u & v)
void Union(int u, int v) {
  int repU = Find(u);
  int repV = Find(v);
  if (repU != repV)
  {
    // Union by size
    if (Size[repU] < Size[repV]) swap(repU, repV);
    par[repV] = repU;
    Size[repU] += Size[repV];
  }
}

int get_size(int i) {
  return Size[Find(i)];
}

int numberOfConnectedComponents(int n) {
  int ct = 0;
  for (int i = 1; i <= n; ++i)
  {
    if (Find(i) == i) ++ct;
  }
  return ct;
}

void build(int n) {
  for (int i = 0; i < n; i++) {
    par[i] = i;
    Size[i] = 1;
  }
}

int main() {
  int u, v, tc, n, k;
  cin >> n >> k;

  build(N); // Create a new set

  bool cycle = 0;
  for (int i = 1; i <= k; i++) {
    cin >> u >> v;
    /* //Finding Cycle
    if(Find(u)==Find(v)) cycle=1;   //Cycle is
                                        Found;
    else Union(u, v); */
    Union(u, v);
  }
  // if(cycle) cout<<"Found Cycle";

  cout << numberOfConnectedComponents(n) <<
endl; // Count Connected Components

  return 0;
}
```

Lowest Common Ancistor(LCA):

```
const int N = 1e5 + 10;
vector<int> g[N];
int table[N + 1][22];
int level[N];
int tin[N], tout[N];
int minLen[N + 1][22], maxLen[N + 1][22];
// maximum ans minimun weight of a tree
int n, lg, Time = 0, INF = 1e9 + 10;

void dfs(int v, int par = -1, int dep = 0, int mn=INF,
int mx=-1)
{
  tin[v] = ++Time;   //for find is_ancestor
  table[v][0] = par;
  level[v] = dep;
  minLen[v][0] = mn, maxLen[v][0] = mx;
  for (int i = 1; i<= lg; i++)
  {
    if (table[v][i - 1] != -1)
    {
      table[v][i] = table[table[v][i - 1]][i - 1];
      //minLen[v][i] = min(minLen[v][i - 1],
```

```
                minLen[table[v][i - 1]][i - 1]);
        //maxLen[v][i] = max(maxLen[v][i - 1],
                maxLen[table[v][i - 1]][i - 1]);
      }
   }
   for (auto &child : g[v])
   {
      if (child == par) continue;
      dfs(child, v, dep + 1);
      //dfs(child.first, v, dep + 1, child.second,
              child.second); //for max & min
   }
   tout[v] = ++Time;
}
void lca_build() //=> O(n*logn)
{
   dfs(1);
}
int lca_query(int a, int b) //=> O(logn)
{
   if (level[a] < level[b]) swap(a, b);
   // int dis = level[a] - level[b];
   // while (dis)  //a and b come to the same level
   // {
   //    int i = log2(dis);
   //    a = table[a][i], dis -= (1 <<i);
   // }
   for (int i = lg; i>= 0; i--)
   //a and b come to the same level
   {
      if (table[a][i] != -1 && level[table[a][i]] >=
level[b])
         a = table[a][i];
   }
   if (a == b) return a;
   for (int i = lg; i>= 0; i--)
   {
      if (table[a][i] != -1 && table[a][i] != table[b][i])
         a = table[a][i], b = table[b][i];
   }
   return table[a][0];
}
int dist(int u, int v)
// distance between two node
{
   int l = lca_query(u, v);
   return level[u] + level[v] - (level[l] << 1);
//level[l]*2
}
Int kth(int u, int k)
{
   for (int i = 0; i<= lg; i++)
         if (k & (1 <<i)) u = table[u][i];
   return u;
}
int findKth(int u, int v, int k)
// kth node from u to v, 0th node is u
{
   int l = lca_query(u, v);
   int d = level[u] + level[v] - (level[l] << 1);
   if (level[l] + k <= level[u]) return kth(u, k);
   k -= level[u] - level[l];
   return kth(v, level[v] - level[l] - k);
}
bool is_ancestor(int u, int v) //u is an ancestor of v
{
   return tin[u] <= tin[v] && tout[u] >= tout[v];
}

void reset() {
   for (int i = 0; i<= n; i++) {
      g[i].clear();
      level[i] = 0;
      for (int j = 0; j <= lg; j++) table[i][j] = -1;
   }
}
int main() {
   cin>> n;
   lg = log2(n) + 1;
   reset();
   // Input …. Query …
} // endl
```

Trie: //=> O(length)

```
struct Trie {
   static const int rangeSize = 26; // for lower_case
letter ('a' <= 'z')

   struct node {
      node *next[rangeSize];
      bool completedWord;
      int cnt;
      node() {
         completedWord = false;
         cnt = 0;
         for (int i = 0; i < rangeSize; i++)
            next[i] = nullptr;
      }
   } *root;

   Trie() {
      root = new node();
   }

   void trieInsert(const string &s) {
      node *cur = root;
      for (char ch : s) {
         int x = ch - 'a'; // for lowercase letter
         if (cur->next[x] == nullptr) {
            cur->next[x] = new node();
```

```
        }
        cur = cur->next[x];
        cur->cnt += 1;
    }
    cur->completedWord = true;
  }

  bool trieSearch(const string &s) {
    node *cur = root;
    for (char ch : s) {
      int x = ch - 'a'; // for lowercase letter
      if (cur->next[x] == nullptr) {
        return false;
      }
      cur = cur->next[x];
    }
    return cur->completedWord;
  }

  int prefixCount(const string &s) {
    node *cur = root;
    for (char ch : s) {
      int x = ch - 'a'; // for lowercase letter
      if (cur->next[x] == nullptr) {
        return 0;
      }
      cur = cur->next[x];
    }
    return cur->cnt;
  }

  void reset(node* cur) {
    for(int i = 0; i < rangeSize; i++)
      if(cur->next[i])
        reset(cur->next[i]);
    delete cur;
  }

  void clear() {
    reset(root); // Delete all nodes
    root = new node(); // Re-initialize root node
for reuse
  }

  ~Trie() { // Destructor
    reset(root);
  }
} trie;
```

Sparse Table:

```
// 0-based indexing, query finds in range [first,
last]
#define lg(x) (31 - __builtin_clz(x))
const int N = 1e5 + 7;
const int K = lg(N);

struct sparse_table {
  ll tr[N][K + 1];

  ll f(ll p1, ll p2) { // Change this function
according to the problem.
    return p1 + p2; // <===
  }

  void build(int n, const vector<ll> &a) { // O(N *
logN)
    for (int i = 0; i < n; i++) {
      tr[i][0] = a[i];
    }
    for (int j = 1; j <= K; j++) {
      for (int i = 0; i + (1 << j) <= n; i++) {
        tr[i][j] = f(tr[i][j - 1], tr[i + (1 << (j - 1))][j -
1]);
      }
    }
  }

  ll query1(int l, int r) { // find Sum, LCM =>
O(LogN)
    ll val = 0; // for sum => val = 0 and lcm => val
= 1
    for (int j = K; j >= 0; j--) {
      if ((1 << j) <= r - l + 1) {
        val = f(val, tr[l][j]);
        l += 1 << j;
      }
    }
    return val;
  }

  ll query2(int l, int r) { // find Min, Max, GCD,
AND, OR, XOR => O(1)
    int d = lg(r - l + 1);
    return f(tr[l][d], tr[r - (1 << d) + 1][d]);
  }
} spt;
```

Arthiculation Point:

```
const int N = 3e5 + 9;
int T, low[N], dis[N], art[N];
vector<int> g[N];
int n, m;

void dfs(int u, int pre = 0) {
  low[u] = dis[u] = ++T;
  int child = 0;
  for (auto &v : g[u]) {
    if (!dis[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
```

```
      if (low[v] >= dis[u] && pre != 0) art[u] = 1;
      ++child;
    }
    else if (v != pre) low[u] = min(low[u], dis[v]);
  }
  if (pre == 0 && child > 1) art[u] = 1;
}
```

Bridge:

```
const int N = 3e5 + 9;
int T, low[N], dis[N];
vector<int> g[N];
vector<pair<int, int>> bridge;
int n, m;

void dfs(int u, int pre = 0) {
  low[u] = dis[u] = ++T;
  int child = 0;
  for (auto &v : g[u]) {
    if (!dis[v])
    {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dis[u]) bridge.push_back({u, v});
      ++child;
    }
    else if (v != pre) low[u] = min(low[u], dis[v]);
  }
}
```

Bipartite:

```
const int N = 2e5 + 7;
vector<int> g[N];
vector<short> clr(N, -1); // 2 color(0 and 1)
bool isBipartite = 0; // Odd length cycle are
Bipartite graph

void dfs(int u, int c) {
  if(isBipartite) return;
  clr[u] = c;
  for(auto &v: g[u]) {
    if(clr[v] != -1) {
      if(clr[v] == c) {
        isBipartite = 1;
        return;
      }
      continue;
    }
    dfs(v, c ^ 1);
  }
}
```

Strongly Connected Components:

```
const int N = 1e5 + 7;
vector<int> g[N], r[N], comp[N]; // 1'Base
Indexing
bool vis[N];
stack<int> st;
int u, v, n, m, numOfComp;
void dfs(int i)
{
  vis[i] = 1;
  for(auto &j: g[i])
  {
    if(vis[j]) continue;
    dfs(j);
  }
  st.push(i);
}
void dfs2(int i)
{
  vis[i] = 1;
  comp[numOfComp].push_back(i);

  for(auto &j: r[i])
  {
    if(vis[j]) continue;
    dfs2(j);
  }
}
void solve()
{
  cin >> n >> m;

  for(int i = 1; i <= m; i++)
  {
    cin >> u >> v;
    g[u].push_back(v);
    r[v].push_back(u);
  }
  for(int i = 1; i <= n; i++)
  {
    if(vis[i]) continue;
    dfs(i);
  }
  memset(vis, 0, sizeof vis);
  numOfComp = 0;
  while(!st.empty())
  {
    int x = st.top();
    if(!vis[x])
    {
      ++numOfComp;
      dfs2(x);
    }
    st.pop();
  }
  for(int i = 1; i <= n; i++)
  {
    if(comp[i].size() == 0) continue;
    cout << i << "=> ";
```

```
        for(auto &j: comp[i]) cout << j << ", ";
        cout << endl;
    }
    return;
}
```

Topological sort: (Kahn's Algorithm)
=> only applicable for **acyclic** graph

```
void topoSortBFS(int v, vector<vector<int>> &adj)
{
    queue<int> nodes;
    vector<int> inDeg(v + 1, 0);
    vector<int> ans;
    for (int i = 1; i <= v; i++) {
        for (int j : adj[i]) inDeg[j]++;
    }

    for (int i = 1; i <= v; i++) {
        if (inDeg[i] == 0) nodes.push(i);
    }

    while (!nodes.empty()) {
        int n = nodes.front();
        ans.push_back(n);
        nodes.pop();

        for (int i : adj[n]) {
            inDeg[i]--;
            if (inDeg[i] == 0) nodes.push(i);
        }
    }

    // If Topological Sort does not exist then the
vector size will be less than the number of vertex
size
    if(ans.size() < v) cout << "Topological Sort does
not exist for this graph :)" << endl;
    else {
        for (auto &i: ans) cout << i << " ";
        cout << endl;
    }
}
```

Dijkstra: Used to find the shortest path between nodes in a graph with non-negative edge weights.
O((V + E) * Log(V))

```
const ll INF = LLONG_MAX;
vector<vector<pair<ll, ll>>> g;

void dijkstra(ll s, vector<ll> &d, vector<ll> &p)
{
    ll n = g.size();
    d.assign(n + 1, INF);
    p.assign(n + 1, -1);
    d[s] = 0;
    priority_queue<pair<ll, ll>, vector<pair<ll,
ll>>, greater<pair<ll, ll>>> q;
    q.push({0, s});

    while (!q.empty())
    {
        auto [d_v, v] = q.top();
        q.pop();
        if (d_v != d[v]) continue;
        for (auto &[to, len] : g[v])
        {
            if (d[v] + len < d[to])
            {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
vector<ll> restore_path(ll s, ll dest, vector<ll>
const &p)
{
    vector<ll> path;
    for (auto v = dest; v != s && p[v] != -1; v = p[v])
                        path.push_back(v);
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

KMP :

```
vector<int> ConstructLPSarray(string pattern) //
time complexity(O(n))
{
    int n = pattern.length();
    vector<int> lps(n);
    int j = 0;
    for (int i = 1; i < n;)
    {
        if (pattern[i] == pattern[j])
            lps[i] = j + 1, j++, i++;
        else
        {
            if (j != 0)   j = lps[j - 1];
            else  lps[i] = j, i++;
        }
    }
    return lps;
```

```
}
void KMP(string text, string pattern)
{
   vector<int> lps = ConstructLPSarray(pattern);
   int j = 0, i = 0; // i=text, j=pattern
   int n = text.length();
   int m = pattern.length();
   bool ok = false;
   while (i < n)
   {
      if (text[i] == pattern[j])  i++, j++;
      else
      {
         if (j != 0)  j = lps[j - 1];
         else  i++;
      }
      if (j == m)
      {
         cout << i - m << endl;
         j = lps[j - 1];
         ok = true;
      }
   }
   if (!ok)
      cout << endl;
}
void solve()
{
   int n;
   while (cin >> n)
   {
      if (n == 0) break;
      string pattern, text;
      cin >> pattern >> text;
      KMP(text, pattern);
   }
}
```

Others:

Sorting pair Using Compare Function:

=>O(n*log(n))

If vector<pair<ll, ll>>**vec**{{3, 4}, {1, 2}, {3, 5}, {3, 2}, {6, 1}};

```
bool cmp(pair<ll, ll> a, pair<ll, ll> b)
{
   if (a.first != b.first) return a.first<b.first;
   //=>first value increasing order;
   return a.second>b.second;
   //=> second value descending order;
}
sort(vec.begin(), vec.end(), cmp);
//=>vec={{1,2}, {3,5}, {3,4}, {3,2}, {6,1}};
struct Node
{
   ll val, id, cost;
   bool operator<(const Node &rhs) const
   {
      //your main logic for the comparator goes here
      return make_pair(val, id) < make_pair(rhs.val, rhs.id);
   }
};
```

Minimum fraction:

If **a/b = c/d** => ex: 12/18 = 2/3

c = **a /__gcd(a,b)**; d = **b /_gcd(a,b);**

Find N'th Fibonacci number using Binet's Formula: **=>O(1)**

```
int fib(int n){
   double phi = (sqrt(5) + 1) / 2;
   return round(pow(phi, n) / sqrt(5));
}
```

Count words in a string using stringstream:

```
#include<sstream>
#include<string>
int countWords(string str)
{
stringstream sf(str);
   string word;
   int count = 0;
  while (sf >> word)
         count++; // <= you can change statement
   return count;
}
```

__int128 Data-type:

```
__int128 read()
{
   __int128 x = 0, f = 1;
   char ch = getchar();
   while (ch< '0' || ch> '9')
   {
      if (ch == '-') f = -1;
      ch = getchar();
   }
   while (ch>= '0' &&ch<= '9')
   {
      x = x * 10 + ch - '0';
      ch = getchar();
   }
   return x * f;
}
void print(__int128 x)
{
   if (x < 0) putchar('-'), x = -x;
   if (x > 9) print(x / 10);
```

```
    putchar(x % 10 + '0');
}
string __int128toString(__int128 num)
{
    auto tenPow18 = 1000000000000000000;
    string str;
    do
    {
        long long digits = num % tenPow18;
        auto digitsStr = to_string(digits);
        auto leading0s = (digits != num) ? string(18 -
digitsStr.length(), '0') : "";
        str = leading0s + digitsStr + str;
        num = (num - digits) / tenPow18;
    } while (num != 0);
    return str;
}
bool cmp(__int128 x, __int128 y) { return x > y; }
```

// To find the **rectangular grid sum in a range** with complexity O(1)

```
class NumMatrix {
    private:
    vector<vector<ll>>  prefixSum;

    public:
    NumMatrix(vector<vector<int>> &matrix) {
        int m = matrix.size();
        int n = matrix[0].size();

        prefixSum = vector<vector<ll>>(m + 1,
vector<ll>(n + 1, 0));

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                prefixSum[i][j] = matrix[i - 1][j - 1] +
prefixSum[i - 1][j] + prefixSum[i][j - 1] -
prefixSum[i - 1][j - 1];
            }
        }
    }

    ll sumRegion(int row1, int col1, int row2, int
col2) {
        return prefixSum[row2 + 1][col2 + 1] -
prefixSum[row1][col2 + 1] - prefixSum[row2 +
1][col1] + prefixSum[row1][col1];
    }
};
```