Contents

1 Number theory **2**
- 1.1 Bigmod, Inverse Mod . . . 2
- 1.2 inclusion-exclusion - number of multiples of divs in [l, r] . . . 2
- 1.3 nCr if mod is small . . . 2
- 1.4 Linear Sieve . . . 2
- 1.5 Segmented sieve.cpp . . . 2
- 1.6 Phi Function[n]/ Phi Sieve[1 to n] . . . 2
- 1.7 Discrete Logarithm . . . 3
- 1.8 MatExpo . . . 3
- 1.9 CRT . . . 4
- 1.10 Miller Rabin Primality Test . . . 4
- 1.11 FFT . . . 5

2 Data Structure **5**
- 2.1 BIT . . . 5
- 2.2 Sparse Table . . . 5
- 2.3 LCA with Sparse Table . . . 6
- 2.4 PBDS/Ordered Set . . . 6
- 2.5 Mo's Algo/ Sqrt decomposition . . . 7
- 2.6 Centroid Decomposition . . . 7
- 2.7 Persistent Segment Tree . . . 8

3 Graph Theory **9**
- 3.1 0-1 BFS . . . 9
- 3.2 Bellman Ford Algorithm to find Negative Cycle . . . 9
- 3.3 Kruskal MST . . . 10
- 3.4 Articulation Bridge and Point . . . 10
- 3.5 Online Articulation Bridge Finding . . . 11
- 3.6 Max Flow(Dinic's Algo) . . . 12
- 3.7 Min Cost Max Flow . . . 13
- 3.8 Strongly Connected Components . . . 14
- 3.9 DSU with Rollback . . . 15
- 3.10 Heavy Light Decomposition . . . 16

4 Dynamic Programming **17**
- 4.1 LIS . . . 17
- 4.2 SOS DP . . . 17

5 String **18**
- 5.1 Hashing . . . 18
- 5.2 Trie . . . 18
- 5.3 Z-Algo //0-based Indexing . . . 19
- 5.4 KMP Algorithm . . . 19
- 5.5 Palindromic Tree . . . 20
- 5.6 Suffix Array . . . 20

6 Geometry **21**
- 6.1 Convex Hull . . . 21
- 6.2 Polar Sort . . . 22
- 6.3 Points on polygon . . . 22

7 Miscellaneous **23**
- 7.1 C++17 Sublime Build . . . 23
- 7.2 Test Case Generator with FASTIO . . . 23
- 7.3 Output Checker Bash Script . . . 24
- 7.4 Custom Hash for unordered map . . . 24
- 7.5 Release vector memory . . . 24
- 7.6 Graph and Bitmask Operation . . . 24

8 Notes **24**
- 8.1 Stars and Bars Theorem . . . 24
- 8.2 GCD . . . 25
- 8.3 Geometric Formula . . . 25
- 8.4 Series/Progression . . . 25
- 8.5 Combinatorial formulas . . . 25

1 Number theory

1.1 Bigmod, Inverse Mod

```
ll bigmod(ll a, ll b, ll mod){
    ll res = 1;
    while (b > 0){
        if (b & 1)  res = (res * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return res;
}

ll inverse_mod(ll a, ll b) {
 return 1 < a ? b - inverse_mod(b % a, a) * b / a : 1;
}
ll inv[N];  // inverse modulo pre calculate
void imod() {
 inv[1] = 1;
 for (ll i = 2; i < N; i++) inv[i] = (mod - (mod / i) * inv[mod % i]) % mod;
}
// another way to find inverse modulo of n is
ll inv_of_n = bigmod(n, mod - 2, mod);
```

1.2 inclusion-exclusion - number of multiples of divs in [l, r]

```
ll iep(int l, int r) {
 if (l > r) return 0;
 ll sum = 0, sz = divs.size();
 for (ll j = 1; j < (1LL << sz); j++) {
   ll gun = 1, one = 0;
   for (ll i = 0; i < sz; i++)
     if (j & (1LL << i)) {
       one++; gun *= divs[i];
     }
   ll mult = (r / gun) - ((l - 1) / gun);
   if (one % 2 == 1) sum += mult;
   else sum -= mult;
 }
 return sum;
}

```

1.3 nCr if mod is small

```
// /total complexity O(p+logn+logr)....
// /just convert  n and r to base p and multiply the ncr of all coefficients  of  n and r........
// /n=p^k*x1+p^(k-1)*x2+p^(k-2)*x3+......p^0*xk.
// /r=p^k*y1+p^(k-1)*y2+p^(k-2)*y3+......p^0*yk.
// /our ans= x1Cy1*x2Cy2*x3Cy3*................xkCyk
// /range of x and y is [0,p].....
```

1.4 Linear Sieve

```
const ll N = 10000000;
vector<ll> lp(N+1), pr;

for (ll i=2; i <= N; ++i) {
    if (lp[i] == 0) {
        lp[i] = i; pr.push_back(i);
    }
    for (ll j=0; j < (ll)pr.size() && pr[j] <= lp[i] && i*pr[j] <= N; ++j) {
        lp[i * pr[j]] = pr[j];
    }
}
```

1.5 Segmented sieve.cpp

```
int segmented_sieve(ll l, ll r) { //r-l <= 1e5
        bool mark2[r - l + 1] = {0};
        for (ll z : prime) {
                if (z * z > r) break;

                ll j = ((l + z - 1) / z) * z;
                if (j == z) j += z;
                for (; j <= r; j += z) mark2[j - l] = 1;
        }

        int ans = 0, i = max(2ll, l);
        for (; i <= r; i++)
                ans += !mark2[i - l];
        return ans;
}
```

1.6 Phi Function[n]/ Phi Sieve[1 to n]

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
```

```
        if (n % i == 0) {
            while (n % i == 0)  n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}

void phi_1_to_n(int n) {
    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;
}
```

1.7 Discrete Logarithm

```
int a,b,m;
///this find the minimum x.
int discreet_log(int a,int b,int m)
{
    if(a==0)
        return  b==0?1:-1;
    a=a%m,b=b%m;
    int n=sqrt(m)+1;
    int res=1;
    for(int i=0; i<n; i++)
        res=(res*a)%m;///a^n
    unordered_map<int,int>vals;///O(1)
    int cur=b;
    for(int q=0; q<n; q++)///a^q
    {
        vals[cur]=q;
        cur=(cur*a)%m;
    }
    cur=1;
    for(int p=1; p<=n; p++)
    {
        cur=(cur*res)%m;
        if(vals.count(cur))///O(1)
            return  n*p-vals[cur];
    }
    return -1;
}
///this works only if gcd(a,m)=1.
///Best implementation.....O(sqrt(m)).....using hashmap.
/**
a^x=b(mod m)
find x
or, find x=loga(b)(mod m)
log a base b mod m
*/
cin>>a>>b>>m;
int ans=discreet_log(a,b,m);
cout<<ans<<endl;
```

1.8 MatExpo

```
const ll N = 2, mod = 1000000007;
struct matrix {
        ll mat[N][N];
        matrix(int a, int b, int c, int d) {
                mat[][] = {{a, b}, {c, d}};
        }
        matrix operator * (const matrix &another) {
                matrix res(0, 0, 0, 0);

                for (int i = 0; i < N; i++)
                        for (int j = 0; j < N; j++) {
                                for (int k = 0; k < N; k++) {
                                        res.mat[i][j] = (res.mat[i][j] +
↪  mat[i][k] * another.mat[k][j]);
                                        if (res.mat[i][j] > 8 * mod * mod)
↪  res.mat[i][j] -= 8 * mod * mod; // to reduce mod operation, 8*mod*mod
                                }
                                res.mat[i][j] %= mod;
                        }
                // res.mat[i][j] = (res.mat[i][j] + mat[i][k] *
↪  another.mat[k][j]) % mod;
                return res;
        }
};

matrix expo(matrix a, ll n) {
        if (n == 1) return a;

        matrix ret = expo(a, n / 2);
        ret = ret * ret;
        if (n & 1) ret = ret * a;

        return ret;
}
```

```cpp

int main()
{
        ll a = 0, b = 1, n; // f[0] = a, f[1] = b, f[n] = f[n - 1] + f[n - 2]
        cin >> n; // check n = 1 or base/corner case

        matrix res(1, 1, 1, 0); res = expo(res, n - 1);
        ll ans = (res.mat[0][0] * b + res.mat[0][1] * a) % mod;
}
```

1.9 CRT

```cpp
long long GCD(long long a, long long b) { return (b == 0) ? a : GCD(b, a % b); }
inline long long LCM(long long a, long long b) { return a / GCD(a, b) * b; }
inline long long normalize(long long x, long long mod) { x %= mod; if (x < 0) x += mod; return x; }
struct GCD_type { long long x, y, d; };
GCD_type ex_GCD(long long a, long long b)
{
    if (b == 0) return {1, 0, a};
    GCD_type pom = ex_GCD(b, a % b);
    return {pom.y, pom.x - a / b * pom.y, pom.d};
}

pair <ll, ll> crt(vector <ll> &a, vector <ll> &n) { // 1 based indexing
    ll ans = a[1];
    ll lcm = n[1];
    for (int i = 2; i < a.size(); i++)
    {
        auto pom = ex_GCD(lcm, n[i]);
        int x1 = pom.x;
        int d = pom.d;
        if ((a[i] - ans) % d != 0) return { -1, -1}; // no solution

        ans = normalize(ans + x1 * (a[i] - ans) / d % (n[i] / d) * lcm, lcm * n[i] / d);
        lcm = LCM(lcm, n[i]); // you can save time by replacing above lcm * n[i] /d by lcm = lcm * n[i] / d
    }

    return {ans, lcm};
}

int main()
{
    int t; cin >> t;
    vector <ll> a(t + 1), n (t + 1);
    for (int i = 1; i <= t; i++) {
        cin >> a[i] >> n[i];
        normalize(a[i], n[i]);
    }
}
```

1.10 Miller Rabin Primality Test

```cpp
/* Miller Rabin Primality Test for <= 10^18 */
#define ll long long

ll mulmod(ll a, ll b, ll c) {
    ll x = 0, y = a % c;
    while (b)  {
        if (b & 1) x = (x + y) % c;
        y = (y << 1) % c;
        b >>= 1;
    }
    return x % c;
}
ll fastPow(ll x, ll n, ll MOD) {
    ll ret = 1;
    while (n) {
        if (n & 1) ret = mulmod(ret, x, MOD);
        x = mulmod(x, x, MOD);
        n >>= 1;
    }
    return ret % MOD;
}
bool isPrime(ll n) {
    if (n == 2 || n == 3) return true;
    if (n == 1 || !(n & 1)) return false;
    ll d = n - 1;
    int s = 0;
    while (d % 2 == 0) {
        s++;
        d /= 2;
    }

    int a[9] = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
    for (int i = 0; i < 9; i++)  {
        if (n == a[i]) return true;
        bool comp = fastPow(a[i], d, n) != 1;
        if (comp) for (int j = 0; j < s; j++)  {
                ll fp = fastPow(a[i], (1LL << (ll)j) * d, n);
                if (fp == n - 1) {
                    comp = false;
                    break;
                }
            }
        if (comp) return false;
```

```
    }
    return true;
}
```

1.11 FFT

```
/***
* Multiply (7x^2 + 8x^1 + 9x^0) with (6x^1 + 5x^0)
* ans = 42x^3 + 83x^2 + 94x^1 + 45x^0
* A = {9, 8, 7}
* B = {5, 6}
* V = multiply(A,B)
* V = {45, 94, 83, 42}
***/
/*** Tricks
* Use vector < bool > if you need to check only the status of the sum
* Use bigmod if the power is over same polynomial && power is big
* Use long double if you need more precision
* Use long long for overflow
***/
typedef vector<int> vi;
const double PI = 2.0 * acos(0.0);
using cd = complex<double>;
void fft(vector<cd> &a, bool invert = 0) {
  int n = a.size();
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1)
      j ^= bit;
    j ^= bit;

    if (i < j)
      swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI / len * (invert ? -1 : 1);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
        cd u = a[i + j], v = a[i + j + len / 2] * w;
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
        w *= wlen;
      }
    }
  }
  if (invert) {
    for (cd &x : a)
      x /= n;
  }
}

void ifft(vector<cd> &p) { fft(p, 1); }

vi multiply(vi const &a, vi const &b) {
  vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while (n < a.size() + b.size())
    n <<= 1;
  fa.resize(n);
  fb.resize(n);

  fft(fa);
  fft(fb);
  for (int i = 0; i < n; i++)
    fa[i] *= fb[i];
  ifft(fa);

  vi result(n);
  for (int i = 0; i < n; i++)
    result[i] = round(fa[i].real());
  return result;
}
```

2 Data Structure

2.1 BIT

```
ll bit[N]; //Add(0)--> RTE
void Add(ll x,ll v) { for(;x < N;x += x & (-x)) bit[x] = (bit[x] + v) ; }
ll Sum(ll x) { ll r = 0; for(;x;x -= x & (-x)) r = (r + bit[x]) ; return r; }

```

2.2 Sparse Table

```
const ll N = 2e5 + 20, M = 20; // M = log2(N)
ll n, q, a[N], spt[M][N], lg2[N];

ll f(int i, int j) { // do the main operation here
        return spt[i - 1][j] + spt[i - 1][j + (1 << (i - 1))];
}

void build() {
        lg2[1] = 0;
        for (int i = 2; i < N; i++)
                lg2[i] = lg2[i / 2] + 1;
```

```
        for (int i = 0; i < n; i++)
                spt[0][i] = a[i];
        for (int i = 1; i < M; i++) {
                for (int j = 0; j + (1 << i) <= n; j++) {
                        spt[i][j] = f(i, j);
                }
        }
}

ll querySum(int l, int r) {
        ll sum = 0;
        for (int i = M - 1; i >= 0; i--) {
                if ((1 << i) <= r - l + 1) {
                        sum += spt[i][l];
                        l += (1 << i);
                }
        }
        return sum;
}

ll queryMin(int l, int r) {
        int k = lg2[r - l + 1];
        return min(spt[k][l], spt[k][r - (1 << k) + 1]);
}
```

2.3 LCA with Sparse Table

```
const int N = 2e5, M = 20;
int n, level[N], par[N], root, spt[M][N];
vector <int> g[N];

void dfs(int u, int v, int c) {
    par[v] = u;
    level[v] = c;

    for(int &z : g[v]) if(z != u) dfs(v, z, c + 1);
}

void lca_build() {
    memset(spt, -1, sizeof spt);

    for(int i = 1; i <= n; i++) spt[0][i] = par[i];
    for(int i = 1; i < M; i++) {
        for(int j = 1; j <= n; j++) {
            int p = spt[i - 1][j];
            if(p >= root) spt[i][j] = spt[i - 1][p]; // 2^i-th parent of j
        }
    }
}

int lca_of(int u, int v) {
    int p = u, q = v;
    if(level[p] > level[q]) swap(p, q);  // lets level[p] <= level[q]
    int d = level[q] - level[p];
    for(int i = M - 1; i >= 0 and d; i--) {
        if(d >= (1 << i)) {
            d -= 1 << i;
            q  = spt[i][q];
        }
    }
    // now level[p] = level[q]
    if(p == q) return p;

    for(int i = M - 1; i >= 0; i--) {
        if(spt[i][p] >= root and spt[i][p] != spt[i][q]) {
            p = spt[i][p];
            q = spt[i][q];
        }
    }
    return spt[0][p];
}

int main()
{
    root = 1;
        // take input
        dfs(-1, root, 0);
        lca_build();
        cout << lca_of(root, n) << '\n';
}
```

2.4 PBDS/Ordered Set

```
#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>;

    ordered_set <pair<int,int>> st;
        st.insert({10, 1});

        cout << (st.find_by_order(2) -> first) << endl; //print element in k-th
↪  index
        cout << st.order_of_key({10, 2}) << endl; //print number of items < k
        auto print = [&] () {
                for (auto z : st) cout << z.first << ' ';
        };
```

```
        print();
        st.erase(st.lower_bound({0, 1}));

// #define ordered_set tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
// #define ordered_set tree<int, null_type, less_equal<int>, rb_tree_tag, tree_order_statistics_node_update>
//less than or equal -> use              less_equal<int>
// #define ordered_set tree<pair<int,int>, null_type, pair<int,int>, rb_tree_tag, tree_order_statistics_node_update>
//to use multiset -> use pair<int, int> where first element is value and second element is index

```

2.5 Mo's Algo/ Sqrt decomposition

```
void remove(idx);  // TODO: remove value at idx from data structure
void add(idx);     // TODO: add value at idx from data structure
int get_answer();  // TODO: extract the current answer of the data structure
int block_size;
struct Query {
    int l, r, idx;
    bool operator<(Query other) const
    {
        return make_pair(l / block_size, r) <
               make_pair(other.l / block_size, other.r);
    }
};
vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());

    // TODO: initialize data structure

    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will always reflect the range [cur_l, cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}

```

2.6 Centroid Decomposition

```
int n,k;
const int mx=5e4+5;
vector<int>adj[mx];
int vis[mx],sub[mx];
int tot;
int depth[mx];
int mx_depth;
int ans;
void find_sub(int node,int par)
{
    sub[node]=1;
    for(auto son:adj[node])
    {
        if(son==par || vis[son])
            continue;
        find_sub(son,node);
        sub[node]+=sub[son];
    }
}
int find_centroid(int node,int par)
{
    for(auto son:adj[node])
    {
        if(son==par || vis[son])
            continue;
        if(sub[son]>tot/2)
        {
            return  find_centroid(son,node);
        }
    }
    return node;
}
void calc_ans(int node,int par,int lev,int ok)
{
    if(lev>k)
        return;
```

```
    mx_depth=max(mx_depth,lev);
    if(ok)
        ans+=depth[k-lev];
    else ++depth[lev];
    for(auto son:adj[node])
    {
        if(son==par || vis[son])
            continue;
        calc_ans(son,node,lev+1,ok);
    }
}
void decompose(int node,int par)
{
    find_sub(node,par);
    tot=sub[node];
    int centroid=find_centroid(node,par);
    vis[centroid]=1;
    mx_depth=0;
    for(auto son:adj[centroid])
    {
        if(son==par || vis[son])
            continue;
        calc_ans(son,centroid,1,1);
        calc_ans(son,centroid,1,0);
    }
    ans+=depth[k];
    for(int i=0; i<=mx_depth; i++)
        depth[i]=0;
    for(auto son:adj[centroid])
    {
        if(son==par || vis[son])
            continue;
        decompose(son,centroid);
    }
}
int32_t main()
{
    /**Find number of pairs (u,v) such that their distance is exactly k. */
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cin>>n>>k;
    for(int i=0; i<n-1; i++)
    {
        int x,y;
        cin>>x>>y;
        adj[x].pb(y);
        adj[y].pb(x);
    }
    decompose(1,0);
    cout<<ans<<endl;
}
```

2.7 Persistent Segment Tree

```
/// Count of numbers of [a,b] range in [L, R] index
#define nsz 100010
#define tsz 6000010 ///take 4n + qlogn
ll a[nsz];
ll NEXT_FREE;
ll version[nsz];
ll val[tsz], Left[tsz], Right[tsz];
void build(ll node, ll lo, ll hi)
{
    if(lo == hi) /// leaf node
    {
        val[node] = 0;
        return;
    }
    Left[node] = NEXT_FREE++;
    Right[node] = NEXT_FREE++;
    ll mid = (lo + hi) >> 1;
    build(Left[node], lo, mid);
    build(Right[node], mid+1, hi);
    val[node] = val[Left[node]] + val[Right[node]];
}
ll update(ll node, ll lo, ll hi, ll idx, ll v)
{
    if(lo > idx || hi < idx)
        return node; /// Out of range, use this node.

    ll nnode = NEXT_FREE++; ///Creating a new node, as idx is in [l, r]

    if (lo == hi)    /// Leaf Node, create new node and return that.
    {
        val[nnode] = val[node]; ///cloning current old leaf node's value to new
↪   leaf node
        val[nnode] += v; /// adding or subtracting or replacing as needed
        return nnode;
    }
    ll mid = (lo + hi) >> 1;
    /// Left[nnode] is new node's Left child, it might end up being the old one
↪   too
    /// Left[node] is current old node's Left child.
    /// So we call to update idx in Left child of old node.
    /// And use it's return node as new node's Left child. Same for Right.
    Left[nnode] = update(Left[node], lo, mid, idx, v);
    Right[nnode] = update(Right[node], mid+1, hi, idx, v);
    val[nnode] = val[Left[nnode]] + val[Right[nnode]]; /// Update value.
    return nnode; /// Return the new node to parent.
```

```
}
ll query(ll lnode, ll rnode, ll lo, ll hi, ll l, ll r)
{
    if(lo > r || hi < l)
        return 0;

    if (lo >= l && hi <= r)
        return val[rnode] - val[lnode];

    ll mid = (lo + hi) >> 1;

    return query(Left[lnode], Left[rnode], lo, mid, l, r)
        +  query(Right[lnode], Right[rnode], mid+1, hi, l, r);
}
/// NEXT_FREE = 0
/// version[0] = NEXT_FREE++
/// build(version[0], 1, n)
/// upd(ara[i], 1) -> increment the frequency
/// So, version[i] = update(version[i-1], 1, n, i, 1)
/// query: Count of numbers of [a,b] range in [L, R] index
/// So, ans = query(version[l-1], version[r], 1, n, L, R)
```

3 Graph Theory

3.1 0-1 BFS

```
vector<int> d(n, INF);
d[s] = 0;
deque<int> q;
q.push_front(s);
while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;
        if (d[v] + w < d[u]) {
            d[u] = d[v] + w;
            if (w == 1)
                q.push_back(u);
            else
                q.push_front(u);
        }
    }
}
```

3.2 Bellman Ford Algorithm to find Negative Cycle

```
//bellman ford with negative cycle print
struct edge {
  ll v, w;
};
const ll N = 3e3 + 6, inf = 1LL << 60;
ll n, m, dis[N], par[N];
vector<edge> g[N];

int bellman_ford() {
  lop(n + 1) dis[i] = inf;
  dis[1] = 0;
  int cy;

  lop(n + 1) {
    cy = -1;
    for (int u = 1; u <= n; u++) {
      for (auto z : g[u]) {
        ll v = z.v, w = z.w;
        if (dis[u] + w < dis[v]) {
          dis[v] = dis[u] + w;
          par[v] = u;
          cy = v; // if(u == n) negative cycle;
        }
      }
    }
  }
  return cy; //cy is a adjacent node or a node of negative cycle
}

int main() {
  cin >> n >> m;
  lop(m) {
    ll u, v, w;
    cin >> u >> v >> w;
    g[u].pb({v, w});
  }

  int x = bellman_ford();
  if (x == -1) {
    //no negative cycle
    return 0;
  }

//x can be not a part of cycle, so if we go through //path sometimes, x will be a node of cycle
  lop(n) x = par[x];
  vector<int> cycle;
  int i = x;
```

```
  while (i != x or cycle.size() <= 1) {
    cycle.pb(i);//retrieving cycle
    i = par[i];
  }
  cycle.pb(i);
  reverse(all(cycle));
  for (int z : cycle)
    cout << z << ' ';
  return 0;
}
```

3.3 Kruskal MST

```
struct edge {
        int u, v, w;
        bool operator < (const edge &b) const {
                return w > b.w;
        }
};

const int N = 2e5;
int n, m, par[N];
vector <edge> eg;

int findpar(int x) {
        return par[x] = par[x] == x ? x : findpar(par[x]);
}

void Union(int u, int v) {
        par[findpar(u)] = findpar(v);
}

int kruskal() {
        sort(eg.begin(), eg.end());
        iota(par, par + n, 0);

        int cost = 0, connected = 0;
        while (connected != n - 1) {
                edge z = eg.back();
                eg.pop_back();

                int x = findpar(z.u), y = findpar(z.v);
                if (x != y) {
                        connected++;
                        cost += z.w;
                        Union(x, y);
                }
        }

        return cost;
}


int main()
{
        ios_base::sync_with_stdio(0); cin.tie(0);

        cin >> n >> m;
        for (int i = 0; i < m; i++) {
                int u, v, w;
                cin >> u >> v >> w;
                eg.push_back({u, v, w});
        }

        cout << kruskal() << '\n';

        return 0;
}
```

3.4 Articulation Bridge and Point

```
const int N = 1e4 + 10;
int n, m, root, Time, low[N], d[N];
bool vis[N], is_arti[N]; // is articulation point
vector <int> g[N];
vector <pair <int, int>> arti_bridge;

void dfs(int p, int u) {
        Time++;
        vis[u] = 1;
        d[u] = low[u] = Time;

        int child = 0;
        for (int v : g[u]) {
                if (v == p) continue;

                if (vis[v]) low[u] = min(low[u], d[v]);
                else {
                        child++;
                        dfs(u, v);

                        low[u] = min(low[u], low[v]);
                        if (low[v] >= d[u] and u != root)
                                is_arti[u] = 1;

                        if (d[u] < low[v])
                                arti_bridge.push_back({min(u, v), max(u, v)});
                }
```

```
        }

        if (u == root and child > 1)
                is_arti[u] = 1;
}

int main () {
        cin >> n >> m;
        for (int i = 0; i < m; i++) {
                int u, v; cin >> u >> v;
                g[u].push_back(v);
                g[v].push_back(u);
        }

        Time = root = 1;
        memset(vis, 0, sizeof vis);
        memset(is_arti, 0, sizeof is_arti);
        dfs(root, root);
}
```

3.5 Online Articulation Bridge Finding

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;

void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    lca_iteration = 0;
    last_visit.assign(n, 0);
    for (int i = 0; i < n; ++i) {
        dsu_2ecc[i] = i;
        dsu_cc[i] = i;
        dsu_cc_size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}

int find_2ecc(int v) {
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
}

int find_cc(int v) {
    v = find_2ecc(v);
    return dsu_cc[v] == v ? v : dsu_cc[v] = find_cc(dsu_cc[v]);
}

void make_root(int v) {
    v = find_2ecc(v);
    int root = v;
    int child = -1;
    while (v != -1) {
        int p = find_2ecc(par[v]);
        par[v] = child;
        dsu_cc[v] = root;
        child = v;
        v = p;
    }
    dsu_cc_size[root] = dsu_cc_size[child];
}

void merge_path (int a, int b) {
    ++lca_iteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
        if (a != -1) {
            a = find_2ecc(a);
            path_a.push_back(a);
            if (last_visit[a] == lca_iteration) {
                lca = a;
                break;
            }
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            b = find_2ecc(b);
            path_b.push_back(b);
            if (last_visit[b] == lca_iteration) {
                lca = b;
                break;
            }
            last_visit[b] = lca_iteration;
            b = par[b];
        }

    }

    for (int v : path_a) {
        dsu_2ecc[v] = lca;
        if (v == lca)
```

```
            break;
        --bridges;
    }
    for (int v : path_b) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
}

void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);
    if (a == b)
        return;

    int ca = find_cc(a);
    int cb = find_cc(b);

    if (ca != cb) {
        ++bridges;
        if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = dsu_cc[a] = b;
        dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
        merge_path(a, b);
    }
}
```

3.6 Max Flow(Dinic's Algo)

```

#define ll long long
const ll maxnodes = 10005;

ll nodes = maxnodes, src, dest;
ll dist[maxnodes], q[maxnodes], work[maxnodes];

struct Edge
{
    ll to, rev;
    ll f, cap;
};

vector<Edge> g[maxnodes];

void addEdge(ll s, ll t, ll cap)
{
    Edge a = {t, g[t].size(), 0, cap};
    Edge b = {s, g[s].size(), 0, 0};
    g[s].push_back(a);
    g[t].push_back(b);
}

bool dinic_bfs()
{
    fill(dist, dist + nodes, -1);

    dist[src] = 0;
    ll index = 0;
    q[index++] = src;

    for (ll i = 0; i < index; i++)
    {
        ll u = q[i];
        for (ll j = 0; j < (ll) g[u].size(); j++)
        {
            Edge &e = g[u][j];
            if (dist[e.to] < 0 && e.f < e.cap)
            {
                dist[e.to] = dist[u] + 1;
                q[index++] = e.to;
            }
        }
    }
    return dist[dest] >= 0;
}

ll dinic_dfs(ll u, ll f)
{
    if (u == dest)
        return f;

    for (ll &i = work[u]; i < (ll) g[u].size(); i++)
    {
        Edge &e = g[u][i];

        if (e.cap <= e.f) continue;

        if (dist[e.to] == dist[u] + 1)
        {
            ll flow = dinic_dfs(e.to, min(f, e.cap - e.f));
            if (flow > 0)
            {
```

```
                e.f += flow;
                g[e.to][e.rev].f -= flow;
                return flow;
            }
        }
    }
    return 0;
}

ll maxFlow(ll _src, ll _dest)
{
    src = _src;
    dest = _dest;
    ll result = 0;
    while (dinic_bfs())
    {
        fill(work, work + nodes, 0);
        while (ll delta = dinic_dfs(src, inf))
            result += delta;
    }
    return result;
}


// addEdge(u, v, C);    edge from u to v. Capacity is C
// maxFlow(s, t);  max flow from s to t
```

3.7 Min Cost Max Flow

```
const ll maxnodes = 10005;

ll nodes = maxnodes, src, dest;
ll dist[maxnodes], exist[maxnodes];
pll par[maxnodes];

struct Edge
{
    ll to, rev;
    ll f, cap, cost;
};

vector<Edge> g[maxnodes];

void addEdge(ll s, ll t, ll cap, ll cost)
{
    Edge a = {t, g[t].size(), 0, cap, cost};
    Edge b = {s, g[s].size(), 0, 0, -cost};
    g[s].push_back(a);
    g[t].push_back(b);
}

bool spfa()
{
    fill(dist, dist + nodes, inf);
    fill(exist, exist + nodes, 0);

    dist[src] = 0, exist[src] = 1;
    queue <ll> q;
    q.push(src);

    while(!q.empty()) {
        ll u = q.front();
        q.pop();
        exist[u] = 0;

        for(ll i = 0; i < g[u].size(); i++) {

            Edge e = g[u][i];

            if(dist[e.to] > dist[u] + e.cost && e.f < e.cap) {

                dist[e.to] = dist[u] + e.cost;
                par[e.to] = mp(u, i);

                if(!exist[e.to]) {
                    q.push(e.to);
                    exist[e.to] = 1;
                }

            }
        }
    }

    return dist[dest] != inf;
}

pll minCostMaxFlow(ll _src, ll _dest)
{
    src = _src;
    dest = _dest;
    pll result = mp(0,0);
    while (spfa())
    {
        ll cur = _dest, flow = inf;
        while(cur != _src) {
            ll p = par[cur].first;
            Edge e = g[p][ par[cur].second ];
            flow = min(flow, e.cap - e.f);

```

```
            cur = p;
        }
        result.first += flow;

        cur = _dest;
        while(cur != _src) {
            ll p = par[cur].first;
            Edge &e = g[p][ par[cur].second ];

            e.f += flow;
            g[e.to][e.rev].f -= flow;

            result.second += flow * e.cost;

            cur = p;
        }
    }
    return result;
}
// addEdge(u, v, C, cst); edge from u to v. Capacity=C, Cost=cst.
// minCostMaxFlow(s, t); min cost max flow from s to t
```

3.8 Strongly Connected Components

```
const int N=1e5+;
vector<int>adj[N];
int n,m;
int vis[N];
stack<int>st;
int on_stack[N];
int in[N];
int lo[N];
int tme;
int cnt;
int scc_num[N];
void dfs(int node)
{
    in[node]=lo[node]=++tme;
    vis[node]=1;
    on_stack[node]=1;
    st.push(node);
    for(auto son:adj[node])
    {
        if(on_stack[son] && vis[son])
        {
            lo[node]=min(lo[node],in[son]);
        }
        else if(!vis[son])
        {
            dfs(son);
            if(on_stack[son]) lo[node]=min(lo[node],lo[son]);
        }
    }
    if(in[node]==lo[node])///From Where the SCC started.
    {
        ++cnt;
        while(1)
        {
            int x=st.top();
            st.pop();
            scc_num[x]=cnt;///Marked the scc num for graph condensation....
            on_stack[x]=0;
            if(x==node) break;
        }
    }
}
int32_t main()
{
    /**
    SCC means the largest subset of nodes where we can go from any node to other
    ↪ nodes.
    SCC may contain multiple loops.
    Condensed graph does not contain any loop.
    */
    ///Complexity : O(V+E).
     ios_base::sync_with_stdio(false);
     cin.tie(NULL);
     cin>>n>>m;
     for(int i=0; i<m; i++)
     {
         int x,y;
         cin>>x>>y;
         adj[x].pb(y);
     }
     for(int i=1; i<=n; i++)
     {
         if(vis[i]) continue;
         dfs(i);
     }

     ///Graph condensation
     vector<int>v[cnt+3];
     for(int i=1; i<=n; i++)
     {
         for(auto j:adj[i])
         {
             if(scc_num[i]==scc_num[j]) continue;
             v[scc_num[i]].pb(scc_num[j]);///scc_num will be the node numbers
         }
```

```
    }
}
```

3.9 DSU with Rollback

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
#define pb push_back
#define pii pair<int,int>
#define endl '\n'
const int N=2e5+5;
int n,m,q;
vector<int>adj[N];
int a[N],b[N];
int block_size;
int ans[N];
vector<pair<pii,int>>blocks[500];
struct dsu_save
{
    int v, rnkv, u, rnku;

    dsu_save() {}

    dsu_save(int _v, int _rnkv, int _u, int _rnku)
        : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};

struct dsu_with_rollbacks
{
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;

    dsu_with_rollbacks() {}

    dsu_with_rollbacks(int n)
    {
        p.resize(n);
        rnk.resize(n);
        for (int i = 0; i < n; i++)
        {
            p[i] = i;
            rnk[i] = 0;
        }
        comps = n-5;
    }

    int find_set(int v)
    {
        return (v == p[v]) ? v : find_set(p[v]);
    }

    bool unite(int v, int u)
    {
        v = find_set(v);
        u = find_set(u);
        if (v == u)
            return false;
        comps--;
        if (rnk[v] > rnk[u])
            swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])
            rnk[u]++;
        return true;
    }

    void rollback(int sz)
    {
        while(op.size()>sz)
        {
            if (op.empty())
                return;
            dsu_save x = op.top();
            op.pop();
            comps++;
            p[x.v] = x.v;
            rnk[x.v] = x.rnkv;
            p[x.u] = x.u;
            rnk[x.u] = x.rnku;
        }
    }
};
void all_clear()
{
    for(int i=0; i<=block_size; i++)
    {
        blocks[i].clear();
    }
}
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int t;
    cin>>t;
    while(t--)
```

```
    {
        cin>>n>>m>>q;
        for(int i=1; i<=m; i++)
            cin>>a[i]>>b[i];
        block_size=450;
        for(int i=1; i<=q; i++)
        {
            int l,r;
            cin>>l>>r;
            blocks[l/block_size].pb({{r,l},i});
        }

        for(int i=0; i<=block_size; i++)
        {
            if(blocks[i].empty())
                continue;
            dsu_with_rollbacks d(n+5);
            sort(blocks[i].begin(),blocks[i].end());
            int curr_r=min(m,(i+1)*block_size);
            for(auto x:blocks[i])
            {
                int l=x.first.second;
                int r=x.first.first;
                int idx=x.second;
                int curr_l=l;
                int mn=min(r,curr_r);
                while(curr_r<=r)
                {
                    d.unite(a[curr_r],b[curr_r]);
                    ++curr_r;
                }
                int sz=d.op.size();
                //if(mn!=curr_r)
                while(curr_l<=r)
                {
                    if(curr_l==min(m,(i+1)*block_size))
                        break;
                    d.unite(a[curr_l],b[curr_l]);
                    ++curr_l;
                }
                ans[idx]=d.comps;
                d.rollback(sz);
            }
        }

        for(int i=1; i<=q; i++)
            cout<<ans[i]<<endl;
        all_clear();
    }
}
```

```
///Debug tips : Look for corner logic that is not handled.
```

3.10 Heavy Light Decomposition

```
/*
CHANGE i ti : change the cost of the i-th edge to ti
QUERY a b : ask for the maximum edge cost on the path from node a to node b
*/
struct edge {
        int v, w, id;
};

const int N = 1e4 + 10;
int a[N], b[N], c[N], subtree[N], par[N], head[N], in[N],  out[N];
int n, tim, tr[4 * N], nd[N]; // segment tree
vector <edge> g[N];

inline void dfs_subtree(int p, int u) {
        subtree[u] = 1; par[u] = p;

        for (auto &x : g[u]) {
                int v = x.v, w = x.w;
                if (v == p) continue;

                dfs_subtree(u, v);
                subtree[u] += subtree[v];
                b[v] = w; // assigning cost of u-v at node v
                nd[x.id] = v;

                if (subtree[g[u][0].v] <= subtree[v])
                        swap(g[u][0], x);
        }
}

inline void dfs_HLD(int p, int u) {
        if (p == -1) head[u] = u;

        in[u] = ++tim;
        for (auto [v, w, id] : g[u]) {
                if (v == p) continue;

                head[v] = (v == g[u][0].v ? head[u] : v);
                dfs_HLD(u, v);
        }
        out[u] = tim;
}

// basic segment tree starts
inline void build(int lo, int hi, int node) {
```

```
        if (lo == hi) {tr[node] = a[lo]; return;}
        int mid = (lo + hi) >> 1, lft = node << 1, rgt = lft | 1;
        build(lo, mid, lft);
        build(mid + 1, hi, rgt);
        tr[node] = max(tr[lft], tr[rgt]);
}

inline void update(int lo, int hi, int idx, int v, int node) {
        if (lo > idx || hi < idx) return;
        if (lo == hi) { tr[node] = v; return;}
        int mid = (lo + hi) >> 1, lft = node << 1, rgt = lft | 1;
        update(lo, mid, idx, v, lft);
        update(mid + 1, hi, idx, v, rgt);
        tr[node] = max(tr[lft], tr[rgt]);
}

inline int query(int lo, int hi, int l, int r, int node) {
        if (lo > r || hi < l) return 0;
        if (lo >= l && hi <= r) return tr[node];
        int mid = (lo + hi) >> 1, lft = node << 1, rgt = lft | 1;
        return max(query(lo, mid, l, r, lft), query(mid + 1, hi, l, r, rgt));
}
// basic segment tree ends

inline void update(int u, int x) {
        update(1, tim, in[nd[u]], x, 1);
}
inline int query(int l, int r) {
        return query(1, tim, in[l], in[r], 1);
}

inline bool isAnchestor(int u, int v) { // is u a anchestor of v?
        return in[u] <= in[v] and out[u] >= out[v];
}

inline int query_tree(int u, int v) {
        int res = 0;
        while (!isAnchestor(head[u], v)) {
                int x = query(head[u], u);
                res = max(res, x);
                u = par[head[u]];
        }
        swap(u, v);
        while (!isAnchestor(head[u], v)) {
                int x = query(head[u], u);
                res = max(res, x);
                u = par[head[u]];
        }

        if (in[v] < in[u]) swap(u, v);
        if (u != v) res = max(res, query(1, tim, in[u] + 1, in[v], 1));
        return res;
}

int main() {
        // take input
        tim = 0;
        dfs_subtree(-1, 1); // 1 is root
        dfs_HLD(-1, 1);
        for (int i = 1; i <= n; i++)
                a[in[i]] = b[i];
        build(1, tim, 1);

        // call update(u, x) to update node in tree
        // call query_tree(u, v) to find the result of query
}
```

4 Dynamic Programming

4.1 LIS

```
int lis(vector<int> const& a) {
 int n = a.size();
 const int INF = 1e9; //INF must be > max(a)
 vector<int> d(n + 1, INF);
 d[0] = -INF;
 for (int i = 0; i < n; i++) {
   int j = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
   if (d[j - 1] <= a[i] && a[i] <= d[j]) d[j] = a[i];
 }
 int ans = 0;
 for (int i = 0; i <= n; i++) {
   if (d[i] < INF) ans = i;
 }
 return ans;
}

```

4.2 SOS DP

```
/// iterate over all the masks
for (int mask = 0; mask < (1<<n); mask++)
{
    F[mask] = A[0];
    /// iterate over all the subsets of the mask
    for(int i = mask; i > 0; i = (i-1) & mask)
    {
        F[mask] += A[i];
```

```
    }
}
///memory optimized, super easy to code.
for(int i = 0; i<(1<<N); ++i)
    F[i] = A[i];
for(int i = 0; i < N; ++i)
    for(int mask = 0; mask < (1<<N); ++mask)
    {
        if(mask & (1<<i))
            F[mask] += F[mask^(1<<i)];
    }
```

5 String

5.1 Hashing

```
const ll N = 1e6 + 10, mod = 2e9 + 63, base1 = 1e9 + 21, base2 = 1e9 + 181;
ll pw1[N], pw2[N], hp1[N], hp2[N], hs1[N], hs2[N], n, q;
string s;

void pw_cal() {
  pw1[0] = pw2[0] = 1;
  for (int i = 1; i < N; i++) {
    pw1[i] = (pw1[i - 1] * base1) % mod;
    pw2[i] = (pw2[i - 1] * base2) % mod;
  }
}
void init() {
  hp1[0] = hp2[0] = hs1[n + 1] = hs2[n + 1] = 0;
  for (int i = 1; i <= n; i++) {
    hp1[i] = (hp1[i - 1] * base1 + s[i - 1]) % mod;
    hp2[i] = (hp2[i - 1] * base2 + s[i - 1]) % mod;
  }
  for (int i = n; i > 0; i--) {
    hs1[i] = (hs1[i + 1] * base1 + s[i - 1]) % mod;
    hs2[i] = (hs2[i + 1] * base2 + s[i - 1]) % mod;
  }
}

ll hashp(int l, int r) {
  ll hash1 = ( hp1[r] - hp1[l - 1] * pw1[r - l + 1]) % mod;
  if (hash1 < 0) hash1 += mod;
  ll hash2 = ( hp2[r] - hp2[l - 1] * pw2[r - l + 1]) % mod;
  if (hash2 < 0) hash2 += mod;
  return (hash1 << 32) | hash2;
}
ll hashs(int l, int r) {
  ll hash1 = (hs1[l] - hs1[r + 1] * pw1[r - l + 1]) % mod;
  if (hash1 < 0) hash1 += mod;
  ll hash2 = (hs2[l] - hs2[r + 1] * pw2[r - l + 1]) % mod;
  if (hash2 < 0) hash2 += mod;
  return (hash1 << 32) | hash2;
}

bool ispal(int l, int r) {
  int mid = (r + l) / 2;
  ll x = hashp(l, mid), y = hashs(mid, r);
  return x == y;
}
```

5.2 Trie

```
const int N = 1e5+10, M = 26;
int trie[N][M], nnode;
bool isword[N];

void reset(int k) {
        for (int i = 0; i < M; i++)
                trie[k][i] = -1;
}


void Insert(string &s) {
        int n = s.size(), node = 0;
        for (int i = 0; i < n; i++) {
                if (trie[node][s[i] - 'a'] == -1) {
                        trie[node][s[i] - 'a'] = ++nnode;
                        reset(nnode);
                }
                node = trie[node][s[i] - 'a'];
        }
        isword[node] = 1;
}

bool Search(string &s) {
        int n = s.size(), node = 0;
        for (int i = 0; i < s.size(); i++) {
                if (trie[node][s[i] - 'a'] == -1) return 0;
                node = trie[node][s[i] - 'a'];
        }
        return isword[node];
}
//find maximum subarray xor sum
int doxor(int s) {
  int nw = 0, t = 0;
  for (int i = 31; i >= 0; i--) {
    bool p = (1 << i) & s;
    if (node[nw][p ^ 1] != -1) {
```

```
      t |= 1 << i;
      nw = node[nw][p ^ 1];
    } else
      nw = node[nw][p];
  }
  return t;
}
//minimum subarray xor sum
int doxor2(int s) {
  int nw = 0, t = 0;
  for (int i = 31; i >= 0; i--) {
    bool p = (1 << i) & s;
    if (node[nw][p] != -1) nw = node[nw][p];
    else {
      t |= 1 << i;
      nw = node[nw][p ^ 1];
    }
  }
  return t;
}
//at first insert(0), then calculate xor before inserting each element of the array
//calculate number of subarray having xor>=k
int doxor(int s) {
  int nw = 0, t = 0;
  for (int i = 31; i >= 0; i--) {
    bool p = (1 << i) & s;
    bool q = (1 << i) & k;
    if (!q) {
      t += (node[nw][p ^ 1] != -1 ? word[node[nw][p ^ 1]] : 0);
      nw = node[nw][p];
    } else
      nw = node[nw][p ^ 1];
    if (nw == -1)
      break;
  }
  if (nw != -1)
    t += word[nw];
  return t;
}
//insert(0), sum returned value, insert prefix xor

int main() {
        reset(0);
        int n; cin >> n;
        for (int i = 0; i < n; i++) {
                string s;
                cin >> s;
                Insert(s);
        }
        int q; cin >> q;
        while (q--) {
                string s;
                cin >> s;
                cout << Search(s) << endl;
        }
}
```

5.3 Z-Algo //0-based Indexing

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

- The Z-function for this string s is an array of length n where the i-th element is equal to the greatest number of characters starting from the position i that coincides with the first characters of s.

5.4 KMP Algorithm

```
#define ll long long
const ll MAX_N = 1e5+10;
char s[MAX_N], pat[MAX_N];  /// 1-indexed
ll lps[MAX_N];     /// lps[i] = longest proper prefix-suffix in i length's prefix
void gen_lps(ll plen)
{
    ll now;
    lps[0] = lps[1] = now = 0;
    for(ll i = 2; i <= plen; i++) {
        while(now != 0 && pat[now+1] != pat[i])
            now = lps[now];
        if(pat[now+1] == pat[i]) lps[i] = ++now;
        else lps[i] = now = 0;
    }
}
ll KMP(ll slen, ll plen)
{
    ll now = 0;
```

```
    for(ll i = 1; i <= slen; i++) {
        while(now != 0 && pat[now+1] != s[i])
            now = lps[now];
        if(pat[now+1] == s[i]) ++now;
        else now = 0;
        /// now is the length of the longest prefix of pat, which
        /// ends as a substring of s in index i.
        if(now == plen) return 1;
    }
    return 0;
}
///Find if pat exists in s as a substring
/// slen = length of s, plen = length of pat
/// call gen_lps(plen); to generate LPS (failure) array
/// call KMP(slen, plen) to find pat in s
```

5.5 Palindromic Tree

```
const char CH = 'a'; // base character
struct pal_tree {
        int n, node, t; string s;
        vector <int> len, link, cnt;
        vector <vector<int>> tree;

        pal_tree(string &ss) {
                s = "0" + ss; n = s.size(); // ss is 0-based but s is 1-based string
                cnt.resize(n + 2); // cnt[i] = count of pal. substring at node i in string s
                len.resize(n + 2); link.resize(n + 2);
                tree.resize(n + 2, vector <int> (26));

                len[1] = -1, len[2] = 0; // len[i] = length of palindrome substring at node i
                link[1] = link[2] = 1; // link[i] = suffix link of node i
                node = t = 2;

                for (int i = 1; i < n; i++) add(i);
                for (int i = node; i > 2; i--) cnt[link[i]] += cnt[i];
        }

        inline int up(int x, int p) {
                while (s[p - len[x] - 1] != s[p]) x = link[x];
                return x;
        }

        void add(int p) {
                t = up(t, p);
                int x = up(link[t], p), c = s[p] - CH;
                if (!tree[t][c]) {
                        tree[t][c] = ++node;
                        len[node] = len[t] + 2;
                        link[node] = len[node] == 1 ? 2 : tree[x][c];
                }
                t = tree[t][c];
                cnt[t]++;
        }
};
```

5.6 Suffix Array

```
#define MAX_N 1000020
int n, t;
// char s[500099];
string s;
int SA[MAX_N], LCP[MAX_N];
int RA[MAX_N], tempRA[MAX_N];
int tempSA[MAX_N];
int c[MAX_N];
int Phi[MAX_N], PLCP[MAX_N];
// second approach: O(n log n)
// the input string, up to 100K characters
// the length of input string
// rank array and temporary rank array
// suffix array and temporary suffix array
// for counting/radix sort
void countingSort(int k) {    // O(n)
    int i, sum, maxi = max(300, n);
    // up to 255 ASCII chars or length of n
    memset(c, 0, sizeof c);
    // clear frequency table
    for (i = 0; i < n; i++)
        // count the frequency of each integer rank
        c[i + k < n ? RA[i + k] : 0]++;
    for (i = sum = 0; i < maxi; i++) {
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (i = 0; i < n; i++)
        // shuffle the suffix array if necessary
        tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];

    for (i = 0; i < n; i++)
        // update the suffix array SA
        SA[i] = tempSA[i];
}

void buildSA() {
    int i, k, r;
```

```
    for (i = 0; i < n; i++) RA[i] = s[i];
    // initial rankings
    for (i = 0; i < n; i++) SA[i] = i;
    // initial SA: {0, 1, 2, ..., n-1}
    for (k = 1; k < n; k <<= 1) {
        // repeat sorting process log n times
        countingSort(k); // actually radix sort: sort based on the second item
        countingSort(0);
        // then (stable) sort based on the first item
        tempRA[SA[0]] = r = 0;
        // re-ranking; start from rank r = 0
        for (i = 1; i < n; i++)
            // compare adjacent suffixes
            tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
                (RA[SA[i]] == RA[SA[i - 1]] && RA[SA[i] + k] == RA[SA[i - 1] +
↪   k]) ? r : ++r;
        for (i = 0; i < n; i++)
            // update the rank array RA
            RA[i] = tempRA[i];

        if (RA[SA[n - 1]] == n - 1) break;
        // nice optimization trick
    }
}

void buildLCP() {
    int i, L;
    Phi[SA[0]] = -1;
    // default value
    for (i = 1; i < n; i++)
        // compute Phi in O(n)
        Phi[SA[i]] = SA[i - 1];
    // remember which suffix is behind this suffix
    for (i = L = 0; i < n; i++) {
        // compute Permuted LCP in O(n)
        if (Phi[i] == -1) { PLCP[i] = 0; continue; }
        // special case
        while (s[i + L] == s[Phi[i] + L]) L++;
        // L increased max n times
        PLCP[i] = L;
        L = max(L - 1, 0);
        // L decreased max n times
    }
    for (i = 0; i < n; i++)
        // compute LCP in O(n)
        LCP[i] = PLCP[SA[i]];
    // put the permuted LCP to the correct position
}
// n = string length + 1
// s = the string
// memset(LCP, 0, sizeof(LCP));  setting all index of LCP to zero
// buildSA(); for building suffix array
// buildLCP(); for building LCP array
// LCP is the longest common prefix with the previous suffix here
// SA[0] holds the empty suffix "\0".

int main()
{
    s = "banana";
    s += "$";
    n = s.size();

    memset(LCP, 0, sizeof(LCP));
    buildSA();
    buildLCP();

    for (int i = 0; i < n; i++) cout << SA[i] << ' ' << s.substr(SA[i], n -
↪   SA[i]) << endl;;
    printf("\n");
    for (int i = 0; i < n; i++) printf("%d ", LCP[i]);
    printf("\n");

    return 0;
}
```

6 Geometry

6.1 Convex Hull

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

```

```cpp
void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i],
    include_collinear))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size()-2],
    down[down.size()-1], a[i], include_collinear))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
        reverse(a.begin(), a.end());
        return;
    }
    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

6.2 Polar Sort

```cpp
inline bool up (point p) {
  return p.y > 0 or (p.y == 0 and p.x >= 0);
}

sort(v.begin(), v.end(), [] (point a, point b) {
  return up(a) == up(b) ? a.x * b.y > a.y * b.x : up(a) < up(b);
});
```

6.3 Points on polygon

```cpp
const int N = 3e5 + 9;
const double inf = 1e100;
const double eps = 1e-9;
const double PI = acos((double)-1.0);
int sign(double x) { return (x > eps) - (x < -eps); }
struct PT {
    double x, y;
    PT() { x = 0, y = 0; }
    PT(double _x, double _y) : x(_x), y(_y) {}
    PT(const PT &p) : x(p.x), y(p.y)    {}
    PT operator + (const PT &a) const { return PT(x + a.x, y + a.y); }
    PT operator - (const PT &a) const { return PT(x - a.x, y - a.y); }
    PT operator * (const double a) const { return PT(x * a, y * a); }
    friend PT operator * (const double &a, const PT &b) { return PT(a * b.x, a *
    b.y); }
    PT operator / (const double a) const { return PT(x / a, y / a); }
    bool operator == (PT a) const { return sign(a.x - x) == 0 && sign(a.y - y) ==
    0; }
    bool operator != (PT a) const { return !(*this == a); }
    bool operator < (PT a) const { return sign(a.x - x) == 0 ? y < a.y : x < a.x;
    }
    bool operator > (PT a) const { return sign(a.x - x) == 0 ? y > a.y : x > a.x;
    }
    double norm() { return sqrt(x * x + y * y); }
    double norm2() { return x * x + y * y; }
    PT perp() { return PT(-y, x); }
    double arg() { return atan2(y, x); }
    PT truncate(double r) { /// returns a vector with norm r and having same
    direction
        double k = norm();
        if (!sign(k)) return *this;
        r /= k;
        return PT(x * r, y * r);
    }
};
inline double cross(PT a, PT b) { return a.x * b.y - a.y * b.x; }
inline int orientation(PT a, PT b, PT c) { return sign(cross(b - a, c - a)); }
/// returns true if  point p is on line segment ab
bool is_point_on_seg(PT a, PT b, PT p) {
    if (fabs(cross(p - b, a - b)) < eps) {
        if (p.x < min(a.x, b.x) || p.x > max(a.x, b.x)) return false;
        if (p.y < min(a.y, b.y) || p.y > max(a.y, b.y)) return false;
        return true;
    }
    return false;
}
 ///checks if convex or not
bool is_convex(vector<PT> &p) {
```

```
    bool s[3]; s[0] = s[1] = s[2] = 0;
    int n = p.size();
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        int k = (j + 1) % n;
        s[sign(cross(p[j] - p[i], p[k] - p[i])) + 1] = 1;
        if (s[0] && s[2]) return 0;
    }
    return 1;
}
bool is_point_on_polygon(vector<PT> &p, const PT& z) {
    int n = p.size();
    for (int i = 0; i < n; i++) {
            if (is_point_on_seg(p[i], p[(i + 1) % n], z)) return 1;
    }
    return 0;
}
/// returns 1e9 if the point is on the polygon
int winding_number(vector<PT> &p, const PT& z) { /// O(n)
    if (is_point_on_polygon(p, z)) return 1e9;
    int n = p.size(), ans = 0;
    for (int i = 0; i < n; ++i) {
        int j = (i + 1) % n;
        bool below = p[i].y < z.y;
        if (below != (p[j].y < z.y)) {
            auto orient = orientation(z, p[j], p[i]);
            if (orient == 0) return 0;
            if (below == (orient > 0)) ans += below ? 1 : -1;
        }
    }
    return ans;
}
/// -1 if strictly inside, 0 if on the polygon, 1 if strictly outside
int is_point_in_polygon(vector<PT> &p, const PT& z) { //// O(n)
    int k = winding_number(p, z);
    return k == 1e9 ? 0 : k == 0 ? 1 : -1;
}
```

7 Miscellaneous

7.1 C++17 Sublime Build

```
{
"cmd" : ["g++ -std=c++17 -Wall -fsanitize=address $file_name -o
    $file_base_name && timeout 10s ./$file_base_name <in>out"],
"selector" : "source.cpp",
"shell": true,
"working_dir" : "$file_path"
}
```

7.2 Test Case Generator with FASTIO

```
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx,avx2,fma")

//generator to generate testcase
#include <bits/stdc++.h>
using namespace std;
#define ll long long
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
// return a random number in [l, r] range
ll rand(ll l, ll r) {
  return uniform_int_distribution<ll>(l, r)(rng);
}

void tree() {
    int n = rand(1, 10); // number of nodes
    int t = rand(1, 5); // random parameter

    vector<int> p(n);
    for (int i = 1; i < n; i++) {
        for (int j = 0; j <= t; j++)
            p[i] = max(p[i], (int)rand(0, i - 1));
    }

    vector<int> perm(n);
    for (int i = 0; i < n; i++) perm[i] = i;
    shuffle(perm.begin(), perm.end(), rng);

    vector<pair<int, int> > edges;
    for (int i = 1; i < n; i++)
        if (rand(0, 1))
            edges.push_back(make_pair(perm[i], perm[p[i]]));
        else
            edges.push_back(make_pair(perm[p[i]], perm[i]));
    shuffle(edges.begin(), edges.end(), rng);

    printf("%d\n", n);
    for (int i = 0; i + 1 < n; i++)
        printf("%d %d\n", edges[i].first + 1, edges[i].second + 1);
}

int main(int argc, char* argv[]) {
  ios_base::sync_with_stdio(0); cin.tie(0);

  ll t = rand(1, 1);
  cout << t << endl;
  while (t--) {
```

```
    ll n = rand(1, 15);
    cout << n << endl;
  }
  return 0;
}
```

7.3 Output Checker Bash Script

```
//Bash script to auto check output
for((i = 1; ; ++i)); do
    echo $i
    ./gen $i > in
    # ./a < in > out1
    # ./brute < in > out2
    # diff -w out1 out2 || break
    diff -w <(./sol < in) <(./brute < in) || break
done
echo case
cat in
#create and build a bruteforce code named brute.cpp, main solution code sol.cpp and a random test case generator gen.cpp. To make this script runable, run this command chmod +x s.sh (s.sh is this bash script name). Then run the script by ./s.sh or bash s.sh
```

7.4 Custom Hash for unordered map

```
// To prevent collision in unordered_map
#include <bits/stdc++.h>
using namespace std;
struct custom_hash {
        static uint64_t splitmix64(uint64_t x) {
// http://xorshift.di.unimi.it/splitmix64.c
                x += 0x9e3779b97f4a7c15;
                x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
                x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
                return x ^ (x >> 31);
        }
        size_t operator()(uint64_t x) const {
                static const uint64_t FIXED_RANDOM =
                    chrono::steady_clock::now().time_since_epoch().
                    count();
                return splitmix64(x + FIXED_RANDOM);
        }
};
/// Declaration: unordered_map <int, int, custom_hash > numbers;
/// Usage: same as normal unordered_map
/// Ex: numbers[5] = 2;
/// *** To use gp_hash_table (faster than unordered_map) **** ///
/// Add these extra two lines:
/// #include <ext/pb_ds/assoc_container.hpp>
/// using namespace __gnu_pbds;
/// Declaration: gp_hash_table<int, int, custom_hash > numbers;
/// Usage: Same as unordered_map
```

7.5 Release vector memory

```
vector<int> Elements ;
// fill the vector up
vector<int>().swap(Elements);
```

7.6 Graph and Bitmask Operation

```
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx,avx2,fma")
/* Infos */
~ 4 Direction
int dr[] = {1, -1, 0, 0};
int dc[] = {0, 0, 1, -1};
~ 8 Direction
int dr[] = {1, -1, 0, 0, 1, 1, -1, -1};
int dc[] = {0, 0, 1, -1, 1, -1, 1, -1};
~ Knight Direction
int dr[] = {1, -1, 1, -1, 2, 2, -2, -2};
int dc[] = {2, 2, -2, -2, 1, -1, 1, -1};
~ Hexagonal Direction
int dr[] = {2, -2, 1, 1, -1, -1};
int dc[] = {0, 0, 1, -1, 1, -1};
~ bitmask operations
int Set(int n, int pos) { return n = n | (1 << pos); }
int reset(int n, int pos) { return n = n & ~(1 << pos); }
bool check(int n, int pos) { return (bool)(n & (1 << pos)); }
int toggle(int n, int pos) { return n = (n ^ (1 << pos));}
bool isPower2(int x) { return (x && !(x & (x - 1))); }
ll LargestPower2LessEqualX(ll x) { for (int i = 1; i <= x / 2; i *= 2) x = x | (x >> i); return (x + 1) / 2;}
// for unlimited stack, run the command in terminal and run the code in terminal
ulimit - s unlimited
```

8 Notes

8.1 Stars and Bars Theorem

1. The number of ways to put n identical objects into k labeled boxes is $\binom{n+k-1}{n}$
2. Suppose, there are n objects to be placed into k bins, ways $= \binom{n-1}{k-1}$

3. Statement of 1no. and empty bins are valid, ways = $\binom{n+k-1}{k-1}$

8.2 GCD

1. gcd(a, b) = gcd(a, a - b) [a > b]
2. gcd(F(a), F(b)) = F(gcd(a,b)) [F=fibonacci]
3. gcd(a,b)=$\sum \phi(k)$ where k are all common divisors of a and b

8.3 Geometric Formula

1. Point Slope Form: $y - y_1 = m \cdot (x - x_1)$
2. Slope, $m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$
3. Slope from line, $m = -\frac{A}{B}$
4. Angle, $tan\theta = \frac{m_1 - m_2}{1 + m_1 \cdot m_2}$
5. Distance from a Point (x_0, y_0) to a Line $(Ax + By + C = 0) = \frac{|Ax_0 + By_0 + C|}{\sqrt{A^2 + B^2}}$
6. Area of segment in radian angle: $A = \frac{1}{2} \cdot r^2(\theta - Sin\theta)$
7. The sum of interior angles of a polygon with n sides $= 180 \cdot (n - 2)$
8. Number of diagonals of a $n-$sided polygon $= \frac{n(n-3)}{2}$
9. The measure of interior angles of a regular $n-$sided polygon $= \frac{180(n-2)}{n}$
10. The measure of exterior angles of a regular $n-$sided polygon $= \frac{360}{n}$
11. Picks theorem: $A = I + \frac{B}{2} - 1$ where A = Area of Polygon, B = Number of integral points on edges of polygon, I = Number of integral points strictly inside the polygon.
12. Sine rule of a Triangle: $\frac{a}{sinA} = \frac{b}{sinB} = \frac{c}{sinC}$
13. Cosine Rule of a Triangle: $cosA = \frac{b^2 + c^2 - a^2}{2bc}$
14. Surface Area (SA) and Volumes(V):
    - Sphere: SA $= 4\pi r^2$, V$= \frac{4}{3}\pi r^3$
    - Cone: SA$= \pi r^2 + \pi rs$, V$= \frac{1}{3}\pi r^2 h$, side,s$= \sqrt{h^2 + r^2}$
    - Cylinder: SA$= 2\pi r^2 + 2\pi rh$, V$= \pi r^2 h$
    - Cuboid: SA$= 2(wh + lw + lh)$, V$= lwh$
    - Trapezoid: Area$= \frac{1}{2}(b1 + b2)h$

8.4 Series/Progression

1. Sum of first n positive number $= \frac{n(n+1)}{2}$
2. Sum of first n odd number $= n^2$
3. Sum of first n even number $= n \cdot (n + 1)$
4. Arithmetic Progression: $n-$th term $= a + (n - 1) \cdot d$, sum $= \frac{n}{2}\{2a + (n - 1) \cdot d\}$; a = first element, d = difference between two elements
5. Geometric Progression: $n-$th term $= a \cdot r^{n-1}$, sum $= a \cdot \frac{r^{n-1}}{r-1}$; a = fist element, r = ratio of two elements
6. Catalan Numbers: $1, 1, 2, 5, 14, 42, 132 \ldots C_n = \frac{(2n)!}{n! \cdot (n+1)!}; n >= 0$

8.5 Combinatorial formulas

1. $\sum_{k=0}^{n} k^2 = n(n + 1)(2n + 1)/6$
2. $\sum_{k=0}^{n} k^3 = n^2(n + 1)^2/4$
3. $\sum_{k=0}^{n} k^4 = (6n^5 + 15n^4 + 10n^3 - n)/30$
4. $\sum_{k=0}^{n} k^5 = (2n^6 + 6n^5 + 5n^4 - n^2)/12$
5. $\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x - 1)$
6. $\sum_{k=0}^{n} kx^k = (x - (n + 1)x^{n+1} + nx^{n+2})/(x - 1)^2$
7. $\binom{n}{k} = \frac{n!}{(n-k)!k!}$
8. $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$
9. $\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$
10. $\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1}$
11. $\binom{n+1}{k} = \frac{n+1}{n-k+1}\binom{n}{k}$
12. $\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$
13. $\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$
14. $\sum_{k=1}^{n} k^2\binom{n}{k} = (n + n^2)2^{n-2}$
15. $\binom{m+n}{r} = \sum_{k=0}^{r} \binom{m}{k}\binom{n}{r-k}$
16. $\binom{n}{k} = \prod_{i=1}^{k} \frac{n-k+i}{i}$
17. $a^{\phi(n)} = 1\%n$ where $\phi(n)$ is Euler Totient Function.
18. $a^b\%m = a^{b\%\phi(m)}\%m$ where a and m are coprime.