Team Notebook

September 29, 2022

Contents

1 2 Closest Points in 2D Plane (N $\log^2 N$)

```
struct Point{
 int x, y;
 Point operator -(Point p)
 {
  return {x-p.x, y-p.y};
 }
 int dist()
 {
  return x*x + y*y;
 }
};
bool by_x(Point &a, Point &b){
 return a.x < b.x;
}
bool by_y(Point &a, Point &b){
 return a.y < b.y;
}
const int N=1e5+10;
int n, ans=1e18;
Point pt[N];

//returns square of distance of closest two points in range
     [L,R] of pt.
int solve(int L, int R){
 if(L==R)
  return 1e18;
 int M=(L+R)/2;
 sort(pt+L, pt+R+1, by_x);
 int d=min(solve(L, M), solve(M+1, R));
 int midx=pt[L+(R-L+1)/2].x;
 vector<Point> v;
 for(int i=L;i<=R;i++){
  if(Point{pt[i].x-midx, 0}.dist()<d)
   v.push_back(pt[i]);
 }
 sort(v.begin(), v.end(), by_y);
 for(int i=0;i<v.size();i++){
  for(int j=i+1;j<v.size();j++){
   if(Point{0, v[i].y-v[j].y}.dist()>d)
    break;
   d=min(d, (v[i]-v[j]).dist());
  }
 }
 return d;
}
```

2 3-D geometry

```
struct point{
 ld x,y,z;
 point():x(0),y(0),z(0){}
 point(ld x,ld y,ld z): x(x),y(y),z(z){}
 point operator +(const point &rhs) const{
  return point(x+rhs.x,y+rhs.y,z+rhs.z);
 }
 point operator -(const point &rhs) const{
  return point(x-rhs.x,y-rhs.y,z-rhs.z);
 }
 point operator *(ld k) const{
  return point(x*k,y*k,z*k);
 }
};

/*
Useful points:
1) Rodrigues rotation formula for rotating a vector in 3-D
     about an axis of rotation:
If v is a vector in 3 and k is a unit vector describing an
     axis of rotation about
which v rotates by an angle  according to the right hand
     rule, the Rodrigues formula
for the rotated vector vrot is:
//vrot= v cos(theta)+ (k x v)sin(theta)+ k(k.v)(1-cos(theta
     )
*/
point cross(point a,point b){
 point tmp;
 tmp.x=a.y*b.z-b.y*a.z;
 tmp.y=b.x*a.z-a.x*b.z;
 tmp.z=a.x*b.y-a.y*b.x;
 return tmp;
}
ld dot(point a,point b){
 return a.x*b.x+a.y*b.y+a.z*b.z;
}
ld len(point v){
 ld len2=v.x*v.x+v.y*v.y+v.z*v.z;
 return sqrtl(len2);
}
ld dis(point a,point b){
 return len(a-b);
}
point rot(point v,point axis,ld theta){
 //using rodrigues rotation formula :
 //vrot= v cos(theta)+ (k x v)sin(theta)+ k(k.v)(1-cos(theta
     ))
 ld l=len(axis);
 axis.x/=l;
 axis.y/=l;
 axis.z/=l;
 point p;
 p=v*cosl(theta)+cross(axis,v)*sinl(theta);
 p=p+axis*(dot(axis,v)*(1-cos(theta)));
 return p;
}
```

3 Aho Corasick

```
constexpr int K = 26; //size
struct Vertex {
    std::array<int, K> next;
    bool terminal = false;
    int terminal_idx = -1;
    int p = -1;
    char pch;
    int depth = 0;
    int link = 0; //suffix link
    int next_terminal = 0; // output link
    std::array<int, K> go;

    Vertex(int p, char ch, int depth) : p(p), pch(ch), depth(
         depth) {
        std::fill(next.begin(), next.end(), 0);
        std::fill(go.begin(), go.end(), 0);
    }
};

class AhoCorasick {
public:
    AhoCorasick() : t(1, {-1, '$', 0}) {}

    void add_string(std::string const& s, int idx) {
        int v = 0;
        for (char ch : s) {
            int c = ch - 'a';
            if (!t[v].next[c]) {
                t[v].next[c] = t.size();
                t.emplace_back(v, ch, t[v].depth + 1);
            }
            v = t[v].next[c];
        }
        t[v].terminal = true;
        t[v].terminal_idx = idx;
    }

    void push_links() {
        std::queue<int> q;
        q.push(0);
        while (!q.empty()) {
            int v = q.front();
            auto& cur = t[v];
            auto& link = t[cur.link];
            q.pop();
            cur.next_terminal = link.terminal ? cur.link :
                 link.next_terminal;

            for (int c = 0; c < K; c++) {
                if (cur.next[c]) {
                    t[cur.next[c]].link = v ? link.next[c] :
                         0;
                    q.push(cur.next[c]);
                } else {
                    cur.next[c] = link.next[c];
                }
            }
        }
    }
    //to check output link add new recursive function
    int transition(int idx, char c) {
        return t[idx].next[c - 'a'];
    }

    Vertex const& getInfo(int idx) {
        return t[idx];
    }

private:
    std::vector<Vertex> t;
};
```

4 Articulation points in undirected graph

```
/*
   In DFS tree, a vertex u is articulation point if one of
        the following two conditions is true.
1) u is root of DFS tree and it has at least two children.
2) u is not root of DFS tree and it has a child v such that
     no vertex in subtree rooted with
   v has a back edge to one of the ancestors (in DFS tree)
        of u.
*/
const int N=2e5;
vector<int> adj[N];


bool is_AP[N];
int tim_AP=0;
bool vis_AP[N];
int disc_AP[N];
int low_AP[N];
void dfs_AP(int node,int par) // call dfs_AP(1,-1)
{
 vis_AP[node]=true;
 disc_AP[node]=++tim_AP;
 low_AP[node]=disc_AP[node];
 int child=0;
 for(auto x: adj[node]){
  if(!vis_AP[x]){
   dfs_AP(x,node);
   child++;
   low_AP[node]=min(low_AP[node],low_AP[x]);
   if(par==-1 && child>1)
    is_AP[node]=true;
   if(par!=-1 && low_AP[x]>=disc_AP[node])
    is_AP[node]=true;
  }
  else if(x!=par){
   low_AP[node]=min(low_AP[node],disc_AP[x]);
  }
 }
}
```

5 BIT - Binary Indexed Tree (Fenwick Tree)

```
struct BIT{
 int N;
 vector<int> bit;
 void init(int n){
  N = n;
  bit.assign(n + 1, 0);
 }
 void update(int idx, int val){
  while(idx <= N){
   bit[idx] += val;
   idx += idx & -idx;
  }
 }
 int pref(int idx){
  int ans = 0;
  while(idx > 0){
   ans += bit[idx];
   idx -= idx & -idx;
  }
  return ans;
 }
 int rsum(int l, int r){
  if(l > r)
   return 0;
  return pref(r) - pref(l - 1);
 }
};
```

6 Biconnected components

```
struct graph
{
 int n;
 vector<vector<int>> adj;

 graph(int n) : n(n), adj(n) {}

 void add_edge(int u, int v)
 {
  adj[u].push_back(v);
  adj[v].push_back(u);
 }

 int add_node()
 {
  adj.push_back({});
  return n++;
 }

 vector<int>& operator[](int u) { return adj[u]; }
};
vector<vector<int>> biconnected_components(graph &adj)
{
 int n = adj.n;

 vector<int> num(n), low(n), art(n), stk;
 vector<vector<int>> comps;
function<void(int, int, int&)> dfs = [&](int u, int p, int
     &t)
{
 num[u] = low[u] = ++t;
 stk.push_back(u);

 for (int v : adj[u]) if (v != p)
 {
  if (!num[v])
  {
   dfs(v, u, t);
   low[u] = min(low[u], low[v]);

   if (low[v] >= num[u])
   {
    art[u] = (num[u] > 1 || num[v] > 2);

    comps.push_back({u});
    while (comps.back().back() != v)
     comps.back().push_back(stk.back()), stk.pop_back();
   }
  }
  else low[u] = min(low[u], num[v]);
 }
};

for (int u = 0, t; u < n; ++u)
 if (!num[u]) dfs(u, -1, t = 0);

// build the block cut tree
function<graph()> build_tree = [&]()
{
 graph tree(0);
 vector<int> id(n);

 for (int u = 0; u < n; ++u)
  if (art[u]) id[u] = tree.add_node();

 for (auto &comp : comps)
 {
  int node = tree.add_node();
  for (int u : comp)
   if (!art[u]) id[u] = node;
   else tree.add_edge(node, id[u]);
 }

 return tree;
};

return comps;
}
```

7 Bridges in graph

```
int tim=0;
int u[N], v[N], vis[N];
int tin[N], tout[N], isBridge[M], minAncestor[N];
vector<pair<int, int> > g[N]; //vertex, index of edge

void dfs(int k, int par)
{
 vis[k]=1;
 tin[k]=++tim;
 minAncestor[k]=tin[k];
 for(auto it:g[k]){
  if(it.first==par)
   continue;
  if(vis[it.first]){
   minAncestor[k]=min(minAncestor[k], tin[it.first]);
   continue;
  }
  dfs(it.first, k);
  minAncestor[k]=min(minAncestor[k], minAncestor[it.first]);
  if(minAncestor[it.first]>tin[k])
   isBridge[it.second]=1;
 }
 tout[k]=tim;
}
```

8 DP Optimizations

```
// 1D-1D Optimization

//
     ------------------------------------------------------------------------


// Divide and conquer Dp optimization

/*The recurrence relation of following type:
dp(i,j)=min{dp( i1 ,k)+C(k,j)} -> where k<=j

The above recurrence relation can be solved using divide and
      conquer optimization of Dp.
Let opt(i,j) be the value of k that minimizes the above
     expression.
If opt(i,j) opt (i,j+1) for all i,j, then we can apply divide
     -and-conquer DP.
This known as the monotonicity condition.
The optimal "splitting point" for a fixed i increases as j
     increases.

// Pseudo code:

Original time complexity: O(n*n*k)
Reduced time complexity : O(n*k*log(n))
*/

const int N=
int dp[N][N];
int C(int i,int j);//Implementation differs every time.

void rec(int i,int l,int r,int optl,int optr)
{
    if(l>r)
    {
        return;
    }
    int mid=(l+r)>>1;
    int &ans=dp[i][mid];
    pair<int,int> best={inf,-1};
    for(int j=optl;j<=min(optr,mid);j++)
    {
        best=min(best,{dp[i-1][j]+C(j,mid),j});
    }
    ans=best.first;
    rec(i,l,mid-1,optl,best.second);
    rec(i,mid+1,r,best.second,optr);
}
int compute_full()
{
    dp[0][0]=0;
    for(int i=1;i<=k;i++)
    {
        rec(i,1,n,0,n);
    }
}
//
     ------------------------------------------------------------------------


//Knuth Optimization :
/*Applicable when the dp problem takes the following form:
 * dp[i][j]=min(dp[i][k]+dp[k+1][j])+Cost[i][j]; for every k
      in [i,j].
 * Now the conditions required to apply Knuth optimizations
      are as follows:
 * 1) Quadrangle inequality: Cost[a][c]+cost[b][d]<=cost[a][
     d]+cost[b][c] for a<=b<=c<=d;
 * 2) Monotonicity: cost[b][c]<=cost[a][d]; for a<=b<=c<=d;
 *
 * Let K=k[i][j] be the point at which dp[i][j]= min(dp[i][K
     ]+dp[i][K+1]+Cost[i][j]) is satisfied,
 * then according to knuth's optimization, k[i][j] lies in
     the range: [ k[i][j-1], k[i+1][j] ].
 *
 * So using this idea, knuth's optimization helps
     calculation of dp[i][j] over all i,j in O(n^2) instead
     of bruteforce dp O(n^3).
 * */


for(int i=1;i<=n;i++)
{
 dp[i][i]=;//initialize this.
 k[i][i]=i;
}
for(int len=2;len<=n;len++)
{
    for(int l=1;l+len-1<=n;l++)
    {
        int r=l+len-1;
        int kleft=k[l][r-1];
        int kright=k[l+1][r];
        dp[l][r]=1e18;
        k[l][r]=l;
  for(int kf=kleft;kf<=kright;kf++)
  {
   if(dp[l][r]>dp[l][kf]+dp[kf+1][r]+cum[r]-cum[l-1])
   {
    dp[l][r]=dp[l][kf]+dp[kf+1][r]+Cost(l,r);// change Cost(l
         ,r) according to the question.
    k[l][r]=kf;
   }
  }
   }
}

//
     --------------------------------------------------------------------

//Convex hull trick
```

9 DSU on tree

```
/*
This is Heavy light decomposition style of DSU on tree.
*/
// can answer queries of type:How many vertices in the
     subtree of vertex v has some property in time O(nlogn)
     (for all of the queries)?
const int maxn=;

int cnt[maxn];
bool big[maxn];
int sz[maxn];   // call szdfs for filling this.

void szdfs(int node,int par){
    sz[node]=1;
    for(auto x: adj[node]){
        if(x!=par){
            szdfs(x,node);
            sz[node]+=sz[x];
        }
    }
}

// implementation changes as per question.
void add(int v, int p, int x){
    cnt[ col[v] ] += x;
    for(auto u: adj[v])
        if(u != p && !big[u])
            add(u, v, x);
}

void dfs(int v, int p, bool keep)
{
    int mx = -1, bigChild = -1;
    for(auto u : adj[v])
       if(u != p && sz[u] > mx)
          mx = sz[u], bigChild = u;
    for(auto u : adj[v])
        if(u != p && u != bigChild)
            dfs(u, v, 0); // run a dfs on small childs and
                 clear them from cnt
    if(bigChild != -1)
        dfs(bigChild, v, 1), big[bigChild] = 1; // bigChild
             marked as big and not cleared from cnt
    add(v, p, 1);
    //now cnt[c] is the number of vertices in subtree of
         vertex v that has color c. You can answer the
         queries easily.
    if(bigChild != -1)
        big[bigChild] = 0;
    if(keep == 0)
        add(v, p, -1);
}
```

10 DSU

```
struct DSU{
    int parent = -1;
};
int find_parent(vector<DSU> &ds, int v){
    while (ds[v].parent >= 0)
        v = ds[v].parent;
    return v;
}
void merge_sets(vector<DSU> &ds, int a, int b){
    int p1 = find_parent(ds, a);
    int p2 = find_parent(ds, b);
    if (p1 == p2)
        return;
    if (abs(ds[p1].parent) < abs(ds[p2].parent)){
        swap(p1, p2);
        swap(a, b);
    }
    ds[p1].parent -= abs(ds[p2].parent);
    ds[p2].parent = p1;
}
```

11 FFT

```
template<typename S, typename T> struct FFT //Use typename S
      as long long int, T as the data type of input vector
     and result vector
{
 using cmplx = complex<long double>; // Can use long double
      for more precision but it will increase constant factor
      .
 const long double PI = acos(-1.0);
 void doDFT(vector<cmplx> &a, bool toInvert = false) {
  S n = a.size();
  for(S i = 1, j = 0; i < n; i ++){
   S bit = (n >> 1);
   for(; (j & bit); bit >>= 1) j ^= bit;
   j ^= bit;
   if(i < j) swap(a[i], a[j]);
  }
  for(S length = 2; length <= n; (length <<= 1)) {
   long double angle = ((2.0 * PI) / length) * (toInvert ?
        -1 : 1);
   cmplx omegaLen(cos(angle), sin(angle));
   for(S i = 0; i < n; i += length) {
    cmplx omega(1);
    for(S j = 0; j < (length / 2); j ++) {
     cmplx evenVal = a[i+j], oddVal = (a[i+j+(length/2)] *
          omega);
     a[i+j] = evenVal + oddVal;
     a[i+j+(length/2)] = evenVal - oddVal;
     omega *= omegaLen;
    }
   }
  }
  if(toInvert) {
   for(cmplx &i : a) i /= n;
  }
 }
 vector<T> doFFT(vector<T> &a, vector<T> &b) {
  vector<cmplx> newA(a.begin(), a.end());
  vector<cmplx> newB(b.begin(), b.end());
  S n = 1;
  while(n < (int)(a.size() + b.size())) n <<= 1;
  newA.resize(n);
  newB.resize(n);
  doDFT(newA, false);
  doDFT(newB, false);
  for(S i = 0; i < n; i ++) {
   newA[i] *= newB[i];
  }
  doDFT(newA, true);
  vector<T> result(n);
  for(S i = 0; i < n; i ++) {
   result[i] = roundl(newA[i].real());//is T is floating
        point, don't round it.
  }
  return result;
 }
};
/*
 How to use with int:
 FFT<int,int> ft;
    vector<int> a={1,1,1,1};
    auto v=ft.doFFT(a,a);
    for(auto x: v)
    {
        tr(x);
    }
 Same way with double, just remove the round function in
     line 54.
*/
```

12 Flows

```
/*--------------------Dinic's flow
     :----------------------------*/
/*
Time complexity:
general case : O((V^2)*E)
0-1 flow     : O(E*sqrt(V))
Bipartite graph: O(sqrt(v)*E)
*/
struct FlowEdge{
 int v, u;
 long long cap, flow = 0;
 FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap
     ) {}
};

struct Dinic{
 const long long flow_inf = 1e18;
 vector<FlowEdge> edges;
 vector<vector<int>> adj;
 int n, m = 0;
 int s, t;
 vector<int> level, ptr;
 queue<int> q;

 Dinic(int n, int s, int t) : n(n), s(s), t(t){
  adj.resize(n);
  level.resize(n);
  ptr.resize(n);
 }

 void add_edge(int v, int u, long long cap){
  edges.emplace_back(v, u, cap);
  edges.emplace_back(u, v, 0);
  adj[v].push_back(m);
  adj[u].push_back(m + 1);
  m += 2;
 }

 bool bfs(){
  while (!q.empty()){
   int v = q.front();
   q.pop();
   for (int id : adj[v]){
    if (edges[id].cap - edges[id].flow < 1)
     continue;
    if (level[edges[id].u] != -1)
     continue;
    level[edges[id].u] = level[v] + 1;
    q.push(edges[id].u);
   }
  }
  return level[t] != -1;
 }

 long long dfs(int v, long long pushed){
  if (pushed == 0)
   return 0;
  if (v == t)
   return pushed;
  for (int &cid = ptr[v]; cid < (int)adj[v].size(); cid++){
   int id = adj[v][cid];
   int u = edges[id].u;
   if (level[v] + 1 != level[u] || edges[id].cap - edges[id
        ].flow < 1)
    continue;
   long long tr = dfs(u, min(pushed, edges[id].cap - edges[
        id].flow));
   if (tr == 0)
    continue;
   edges[id].flow += tr;
   edges[id ^ 1].flow -= tr;
   return tr;
  }
  return 0;
 }

 long long flow(){
  long long f = 0;
  while (true){
   fill(level.begin(), level.end(), -1);
   level[s] = 0;
   q.push(s);
   if (!bfs())
    break;
   fill(ptr.begin(), ptr.end(), 0);
   while (long long pushed = dfs(s, flow_inf)){
    f += pushed;
   }
  }
  return f;
 }
};
```

```
/*--------------------MIN COST MAX FLOW ALGORITHM
    ------------------*/
// Time complexity: O(min(E^2 *V log V, E logV * flow))
const int mxN = ; // fill this everytime
const int INF = 2e9;
struct Edgee{
 int to, cost, cap, flow, backEdge;
};
struct MCMF
{
 int s, t, n;
 vector<Edgee> g[mxN];
 MCMF(int _s, int _t, int _n){
  s = _s, t = _t, n = _n;
 }
 void addEdge(int u, int v, int cost, int cap){
  Edgee e1 = {v, cost, cap, 0, (int)g[v].size()};
  Edgee e2 = {u, -cost, 0, 0, (int)g[u].size()};
  g[u].push_back(e1);
  g[v].push_back(e2);
 }
 pair<int, int> minCostMaxFlow(){
  int flow = 0, cost = 0;
  vector<int> state(n), from(n), from_edge(n), d(n);
  deque<int> q;
  while (true){
   for (int i = 0; i < n; i++)
    state[i] = 2, d[i] = INF, from[i] = -1;
   state[s] = 1;
   q.clear();
   q.push_back(s);
   d[s] = 0;
   while (!q.empty()){
    int v = q.front();
    q.pop_front();
    state[v] = 0;
    for (int i = 0; i < (int)g[v].size(); i++){
     Edgee e = g[v][i];
     if (e.flow >= e.cap || d[e.to] <= d[v] + e.cost)
      continue;
     int to = e.to;
     d[to] = d[v] + e.cost;
     from[to] = v;
     from_edge[to] = i;
     if (state[to] == 1)
      continue;
     if (!state[to] || (!q.empty() && d[q.front()] > d[to]))
      q.push_front(to);
     else
      q.push_back(to);
     state[to] = 1;
    }
   }
   if (d[t] == INF)
    break;
   int it = t, addflow = INF;
   while (it != s)
   {
    addflow = min(addflow,
         g[from[it]][from_edge[it]].cap - g[from[it]][
              from_edge[it]].flow);
    it = from[it];
   }
   it = t;
   while (it != s){
    g[from[it]][from_edge[it]].flow += addflow;
    g[it][g[from[it]][from_edge[it]].backEdge].flow -=
         addflow;
    cost += g[from[it]][from_edge[it]].cost * addflow;
    it = from[it];
   }
   flow += addflow;
  }
  return {cost, flow};
 }
};
```

13 Gauss modular solution of SLAE

```
ll poww(long long a, long long n, long long m){

        long long ans = 1;
        long long mul = a;
        while (n != 0){
                if (n % 2)
                        ans = (ans * mul) % m;
                mul = (mul * mul) % m;
                n /= 2;
        }
        return ans;
}
int inv(int x){
 return poww(x,MOD-2,MOD);
}
int gauss (vector<vector<int>> &a,vector<int> &ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (a[i][col] > a[sel][col])
                sel = i;
        if (a[sel][col]==0)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                int c = (a[i][col]*inv(a[row][col]))%MOD;
                for (int j=col; j<=m; ++j){
                 a[i][j] -= (a[row][j] * c)%MOD;
                 if(a[i][j]<0)
                        a[i][j] += MOD;
                }
            }
        ++row;
    }
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = (a[where[i]][m] * inv(a[where[i]][i]))%
                 MOD;
    for (int i=0; i<n; ++i) {
        int sum = 0;
        for (int j=0; j<m; ++j){
         sum += (ans[j] * a[i][j])%MOD;
         sum %= MOD;
        }
    }
}
```

14 Gauss solution of SLAE

```
/*
Following is an implementation of Gauss-Jordan.
Choosing the pivot row is done with heuristic: choosing
     maximum value in the current column.
```

```
The input to the function gauss is the system matrix a. The
     last column of this matrix is vector b.

The function returns the number of solutions of the system
     (0,1,or ).
If at least one solution exists, then it is returned in the
     vector ans.
*/

const int INF=1e9+100;
const double EPS=1e-6;
int gauss (vector < vector<double> > a, vector<double> & ans
     ) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

15 Graph theory important

```
/*
Minimum vertex cover: A vertex cover in a graph is a set of
     vertices that includes at least one endpoint of every
     edge, and a vertex cover is minimum if no other vertex
     cover has fewer vertices.

Maximum independent set: MIS is set of vertices such that no
      two nodes x,y in the set have an edge between them.

Maximum clique problem: Max clique is set of vertices such
     that for every x,y in the set, there is an edge between
      them.

If we have solution to MIS, we have solution to max clique
     problem: Just take complement of graph and run the algo
      and it gets us the required set of vertices. Same way
     we can solve MIS if we have soln to max clique problem.

If S is vertex cover, then V\S is independent set. therefore
     , (minimum vertex cover+ maximum independent set)= |V|
     of a particular graph.

A matching in a graph is a set of edges no two of which
     share an endpoint, and a matching is maximum if no
     other matching has more edges.

Konig's theorem states that:
In any bipartite graph, the number of edges in a maximum
     matching equals the number of vertices in a minimum
     vertex cover.

How to construct the minimum vertex cover set:
1) Find the max_matching of the bipartite graph.
2) let Z be the set of non matched vertices in L(left
     partition). Now add to Z all vertices reachable by some
      vertex from Z via an alternating path. A path  is
     called alternating if it alternates between edges from
     the matching and edges which are not in the matching.
 our required minimum vertex cover is set S:
 S=(L \setminus Z) \setunion (R \setintersection Z).

Largest independent set of any graph: In graph theory, an
     independent set, stable set, coclique or anticlique is
     a set of vertices in a graph, no two of which are
     adjacent. That is, it is a set {\displaystyle S}S of
     vertices such that for every two vertices in {\
     displaystyle S}S, there is no edge connecting the two.
The complement of a vertex cover in any graph is an
     independent set, so a minimum vertex cover is
     complementary to a maximum independent set.

The largest antichain in a DAG can be found by Dilwort's
     theorem along with Konig's theorem
Algorithm for finding largest antichain:
1) Make two partitions A(1 to n) and B(1 to n).
2) Make undirected edge from (a of A) to (b of B) iff a<b in
      DAG.
3) Find the maximum matching m in the above bipartite graph.
4) Largest antichain for above graph= n-m.

*/
```

16 Hashing(string)

```
struct Hash_string// make one struct for each string to be
     hashed;
{
 vector<ll> hash;
 vector<ll> inv;

 const int base=31;
 int m;

 ll cur_val=0;
 ll compute_hash(string &s,int mod){
  m=mod;
  int n=s.length();
  hash.resize(n+1);
  inv.resize(n+1);
  ll p_pow=1;
  for(int i=0;i<n;i++){
   cur_val=cur_val+(p_pow*(s[i]-'a'+1))%m;
   cur_val%=m;
   hash[i]=cur_val;
   inv[i]=poww(p_pow,m-2,m);
   p_pow=(p_pow*base)%m;
  }
  return cur_val;
 }
 ll get_hash(int l,int r){
  if(l==0)
   return hash[r];
  ll ret=hash[r]-hash[l-1];
  ret%=m;
  ret=(ret+m)%m;
  ret=ret*inv[l];
  ret%=m;
  return ret;
 }
};
// When u need double hash, make two struct object for the
     string(one with MOD=1e9+7, another with MOD=1e9+9).
```

17 Heavy-Light Decomposition(HLD)

```
const int N = 250005;
vector<int> adj[N];
vector<int> subtree_size(N);

int dfs_subtree_size(int node, int par){
    subtree_size[node] = 1;
    for (auto x : adj[node])
        if (x != par)
            subtree_size[node] += dfs_subtree_size(x, node);
    return subtree_size[node];
}

vector<int> chain_number(N), index_in_chain(N); //
     chain_number and index_in_chain for each node.
vector<vector<int>> chains(N);               // atmost N
     chains. each chain is collection of nodes in order from
      top to bottom by height.
map<pair<int, int>, int> chainInfo_to_node; //mapping from {
     chain_number, index in chain} -> node of tree.
vector<pair<int, int>> parent_chain(N);     // contains
     information of {parent_chain_no, index_in_parent_chain}
      from each chain. -1 for root chain.
int cur_chain = 0;

void dfs_hld(int node, int par, int chain_no)
{
    chain_number[node] = chain_no;
    index_in_chain[node] = chains[chain_no].size();
    chains[chain_no].push_back(node);
    chainInfo_to_node[make_pair(chain_number[node],
         index_in_chain[node])] = node;

    pair<int, int> bigChild = {0, -1}; //{subSize of child,
         child_node}
    for (auto x : adj[node])
        if (x != par)
            bigChild = max(bigChild, make_pair(subtree_size[x
                 ], x));

    if (bigChild.second != -1)
        dfs_hld(bigChild.second, node, chain_no);

    for (auto x : adj[node]){
        if (x != par && x != bigChild.second){
            cur_chain++;
            parent_chain[cur_chain] = make_pair(chain_no,
                 index_in_chain[node]);
            dfs_hld(x, node, cur_chain);
        }
    }
}
```

18 KMP

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
// pi[i] is the longest prefix of substring(0...i) which is
     also suffix of the same string.
```

19 LCA

```
const int N=1e5+100;
int n;
vector<int> adj[N];
const int LOGMAX=20;
int T[N];// T[i] returns father of node i
int P[N][LOGMAX]; // P[i][j]= 2^jth ancestor of i
int L[N]; // L[i]=level of node i

void dfs_lca(int node,int par,int level){
 L[node]=level;
 T[node]=par;
 for(auto x: adj[node]){
  if(x!=par){
   dfs_lca(x,node,level+1);
  }
 }
}
void preprocess(){
 dfs_lca(1,0,0);
 for(int i=0;i<=n;i++)
  for(int j=0;j<LOGMAX;j++)
   P[i][j]=-1;
 for(int i=1;i<=n;i++){
  P[i][0]=T[i];
 }
 for(int log=1;log<LOGMAX;log++){
  for(int i=1;i<=n;i++){
   if(P[i][log-1]!=-1){
    P[i][log]=P[P[i][log-1]][log-1];
   }
  }
 }
}
int LCA(int p,int q){
 //make sure level of p>= level of q
 if(L[q]>L[p])
  swap(p, q);
 int loglp;
 for(loglp=1; 1<<loglp <=L[p] ; loglp++)
 {}
 loglp--;

 // make p=(the ancestor of p which is at same level as that
       of q)
 for(int log=loglp;log>=0;log--)
 {
  if(L[p]-(1<<log)>=L[q])
  {
   p=P[p][log];
  }
 }
 // now p and q are at same level.
 if(p==q)
  return p;
 for(int i=loglp;i>=0;i--){
  if(P[p][i]!=-1 && P[p][i]!=P[q][i]){
   p=P[p][i];
   q=P[q][i];
  }
 }
 return T[p];
}
```

20 LIS

```
int LIS(vector<int> a)//Returns size of Longest increasing
     sequence.
{
 int n=a.size();
 vector<int> d(n+1,inf);
 for(int i=0;i<n;i++)
 {
  *(lower_bound(d.begin(),d.end(),a[i]))=a[i];
 }
 for(int i=0;i<=n;i++)
 {
  if(d[i]==inf)
  {
   return i;
  }
 }
 return 0;
}
<------------------------------------------------------------------------

Ishmeet code:

/*
 Note!
 1) Makes a dp array in which number represent to maximum
     length of sequence ending at the current index.
 2) Makes a predecessor array in which number represent the
     index of previous element. If it is equal to -1 then
     the current element is first element.
*/
template<typename S, typename T> /*vector<T>*/ /*pair<vector
     <S>, vector<S>>*/ S LIS(vector<T> a) {
 S sz = a.size();
 S maxLength = 1;
 vector<S> dp(sz, 1);
 vector<S> predecessor(sz, -1);
 set<pair<pair<T, S>, S>> cur;
 cur.insert({{a[0], 1}, 0});
 for(S i = 1; i < sz; i ++) {
  auto it = cur.upper_bound({{a[i], -1}, -1});
  if(it == cur.begin()) {
   cur.insert({{a[i], dp[i]}, i});
   continue;
  }
  it --;
  dp[i] += (*it).first.second;
  predecessor[i] = (*it).second;
  maxLength = max(maxLength, dp[i]);
  it ++;
  vector<pair<pair<T, S>, S>> toDelete;
  while((it != cur.end()) && (((*it).first.first >= a[i]) &&
       ((*it).first.second <= dp[i]))) {
   toDelete.push_back((*it));
   it ++;
  }
  for(auto i : toDelete) cur.erase(i);
  cur.insert({{a[i], dp[i]}, i});
 }
 return maxLength;
 //return dp;
 //return {dp, predecessor};
}
```

21 Linear diophantine equation

```
/*
A Linear Diophantine Equation (in two variables) is an
     equation of the general form:

ax+by=c
where a, b, c are given integers, and x, y are unknown
     integers.

In this article, we consider several classical problems on
     these equations:

finding one solution
finding all solutions
finding the number of solutions and the solutions themselves
      in a given interval
finding a solution with minimum value of x+y
*/
/* CODE FOR FINDING ONE SOLUTION OF ax+by=c.
The function will return false if any solution is not
     possible. x0 ,y0 and g are modified as passed by
     reference
*/
int gcd(int a, int b, int &x, int &y)
{
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd(b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0, int &y0
     , int &g)
{
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}

/*
To find any other solution:
x=x0+k*(b/g)
y= y0k *(a/g)
Where k is any integer.
*/
```

22 Linear programming solver

```
// Two-phase simplex algorithm for solving linear programs
     of the form
//
//     maximize     c^T x
//
//    subject to   Ax <= b
```

```
//
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be
     stored
//
// OUTPUT: value of the optimal solution (infinity if
     unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b,
     and c as
// arguments. Then, call Solve(x).


typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;//dont keep it too small as there is
      a huge precision loss in LPsolver
const DOUBLE INF=1e9;  //don't keep it too large .....
struct LPSolver {
 int m, n;
 VI B, N;
 VVD D;

 LPSolver(const VVD &A, const VD &b, const VD &c) :
  m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
   for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D
        [i][j] = A[i][j];
   for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D
        [i][n+1] = b[i]; }
   for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j];
        }
   N[n] = -1; D[m+1][n] = 1;
  }

 void Pivot(int r, int s) {
  DOUBLE inv = 1.0 / D[r][s];
  for (int i = 0; i < m+2; i++) if (i != r)
   for (int j = 0; j < n+2; j++) if (j != s)
    D[i][j] -= D[r][j] * D[i][s] * inv;
  for (int j = 0; j < n+2; j++) if (j != s) D[r][j] *= inv;
  for (int i = 0; i < m+2; i++) if (i != r) D[i][s] *= -inv;
  D[r][s] = inv;
  swap(B[r], N[s]);
 }

 bool Simplex(int phase) {
  int x = phase == 1 ? m+1 : m;
  while (true) {
   int s = -1;
   for (int j = 0; j <= n; j++) {
    if (phase == 2 && N[j] == -1) continue;
    if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] &&
         N[j] < N[s]) s = j;
   }
   if (s < 0 || D[x][s] > -EPS) return true;
   int r = -1;
   for (int i = 0; i < m; i++) {
    if (D[i][s] < EPS) continue;
    if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s]
         ||
      D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[
           r]) r = i;
   }
   if (r == -1) return false;
   Pivot(r, s);
  }
 }

 DOUBLE Solve(VD &x) {
  int r = 0;
  for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r =
        i;
  if (D[r][n+1] <= -EPS) {
   Pivot(r, n);
   if (!Simplex(1) || D[m+1][n+1] < -EPS) return -INF;
   for (int i = 0; i < m; i++) if (B[i] == -1) {
    int s = -1;
    for (int j = 0; j <= n; j++)
     if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s]
          && N[j] < N[s]) s = j;
    Pivot(i, s);
   }
  }
  if (!Simplex(2)) return INF;
  x = VD(n);
  for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n
       +1];
  return D[m][n+1];
 }
};
```

23 Lucas theorem

```
/*
Lucas theorem states that: [C(n,r)=PRODUCT(C(n(i),r(i)))]
     MOD p
                  where n=n(k)*p^k + n(k-1)*p^(k-1)+...n(0)
                        r=r(k)*p^k + r(k-1)*p^(k-1)+...r(0)
Therefore we can precompute all C(n,r) for all n,r in [0,p).
and use the recursive function nCrModpLucas to calculate C(n
     ,r) mod p
*/


// Returns nCr % p. In this Lucas Theorem based program,
// this function is only called for n < p and r < p.
int nCrModpDP(int n, int r, int p)
{
    // The array C is going to store last row of
    // pascal triangle at the end. And last entry
    // of last row is nCr
    int C[r+1];
    memset(C, 0, sizeof(C));

    C[0] = 1; // Top row of Pascal Triangle

    // One by constructs remaining rows of Pascal
    // Triangle from top to bottom
    for (int i = 1; i <= n; i++)
    {
        // Fill entries of current row using previous
        // row values
        for (int j = min(i, r); j > 0; j--)

            // nCj = (n-1)Cj + (n-1)C(j-1);
            C[j] = (C[j] + C[j-1])%p;
    }
    return C[r];
}

// Lucas Theorem based function that returns nCr % p
// This function works like decimal to binary conversion
// recursive function. First we compute last digits of
// n and r in base p, then recur for remaining digits
int nCrModpLucas(int n, int r, int p)
{
   // Base case
   if (r==0)
      return 1;

   // Compute last digits of n and r in base p
```

```
    int ni = n%p, ri = r%p;

    // Compute result for last digits computed above, and
    // for remaining digits. Multiply the two results and
    // compute the result of multiplication in modulo p.
    return (nCrModpLucas(n/p, r/p, p) * // Last digits of n
        and r
           nCrModpDP(ni, ri, p)) % p; // Remaining digits
}

// Time complexity of this solution is O(p^2 * Log n(to the
    base p)).
```

24 MoAlgorithm

```
void remove(int idx); // TODO: remove value at idx from data
     structure
void add(int idx);  // TODO: add value at idx from data
    structure
int get_answer(); // TODO: extract the current answer of the
     data structure

const int block_size=; // TODO: define the size of block

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;
}
```

25 NTT

```
template<typename T> T FastPower(T num, T raise, T modulo) {
 T result = 1;
 while(raise) {
  if(raise & 1) result = (result * num) % modulo;
  num = (num * num) % modulo;
  raise = (raise >> 1);
 }
 return result;
}
template<typename S> struct NTT {
 const S threshold = 400;
 S modulo, root, rootInverse, rootPower;

 NTT(S setMod = 998244353, S setRoot = 15311432, S
      setInverse = 469870224, S setPower = (1 << 23)) {
  setNewMod(setMod, setRoot, setInverse, setPower);
 }
 void setNewMod(S setMod, S setRoot, S setInverse, S
      setPower) {
  modulo = setMod;
  root = setRoot;
  rootInverse = setInverse;
  rootPower = setPower;
 }
 void doDFT(vector<S> &a, bool toInvert = false) {
  S n = a.size();
  for(S i = 1, j = 0; i < n; i ++) {
   S bit = (n >> 1);
   for(; (j & bit); bit >>= 1) j ^= bit;
   j ^= bit;
   if(i < j) swap(a[i], a[j]);
  }
  for(S length = 2; length <= n; (length <<= 1)) {
   S omegaLen = toInvert ? rootInverse : root;
   for(S i = length; i < rootPower; i <<= 1) {
    omegaLen = ((omegaLen * omegaLen) % modulo);
   }
   for(S i = 0; i < n; i += length) {
    S omega = 1;
    for(S j = 0; j < (length / 2); j ++) {
     S evenVal = a[i+j], oddVal = ((a[i+j+(length/2)] * omega
          ) % modulo);
     a[i+j] = (evenVal+oddVal < modulo) ? evenVal+oddVal :
          evenVal+oddVal-modulo;
     a[i+j+(length/2)] = (evenVal-oddVal >= 0) ? evenVal-
          oddVal : evenVal-oddVal+modulo;
     omega = ((omega * omegaLen) % modulo);
    }
   }
  }
  if(toInvert) {
   S nInverse = FastPower(n, (S)(modulo-2), modulo);
   for(S &x : a) x = ((x * nInverse) % modulo);
  }
 }
 void bruteMulti(vector<S> &a, vector<S> &b) {
  vector<S> res(a.size()+b.size()-1, 0);
  for(S i = 0; i < a.size(); i ++) {
   for(S j = 0; j < b.size(); j ++) {
    res[i+j] = (res[i+j] + ((a[i] * b[j]) % modulo) + modulo)
          % modulo;
   }
  }
  a = res;
 }
 void doNTT(vector<S> &newA, vector<S> &newB) {
  if((newA.size() + newB.size()) <= threshold) {
   bruteMulti(newA, newB);
   return;
  }
  S n = 1;
  while(n < (newA.size() + newB.size())) n <<= 1;
  newA.resize(n);
  newB.resize(n);
  doDFT(newA, false);
  doDFT(newB, false);
  for(int i = 0; i < n; i ++) {
   newA[i] = ((newA[i] * newB[i]) % modulo);
  }
  doDFT(newA, true);
 }
};

/*
Read abt primitive roots once if in doubt.
*/
```

26 Persistent data structures

```
int st[21*N],lpt[21*N],rpt[21*N];
int root[N];

int NODE=0;
int build(int l,int r)
{
    int curnode=NODE++;
    if(l==r)
        return curnode;
    int m=(l+r)/2;
    lpt[curnode]=build(l,m);
    rpt[curnode]=build(m+1,r);
    return curnode;
}
int insert(int oldnode,int l,int r,int w)
{
    if(w>=l && w<=r){
        int curnode=NODE++;
        st[curnode]=st[oldnode]+1;//change this according to
             problem.
        if(l==r)
            return curnode;
        int m=(l+r)/2;
        lpt[curnode]=insert(lpt[oldnode],l,m,w);
        rpt[curnode]=insert(rpt[oldnode],m+1,r,w);
        return curnode;
    }
    else
        return oldnode;
}
```

27 SOS DP

```
/*
*/


void SOS()
{
 for(int i = 0; i<(1<<max_bit); ++i)
  F[i] = A[i];
 for(int i = 0;i < max_bit; ++i)
  for(int mask = 0; mask < (1<<max_bit); ++mask){
   if(mask & (1<<i))
   {
    // F[mask] += F[mask^(1<<i)]; // use for sum over submask
    // F[mask^(1<<i)] += F[mask]; // Use for sum over
         supermask
   }
 }
}

vector<int> mobius_transform(int n, vector<int> f)
{
 /* Definition : mu[S] = + f[S]
       - (all subset of S with exactly one element removed)
       + (all subsets of S with exactly 2 elements removed)
       - ()
       + ()
  check definition in blog for more clarity.
 */
 assert(f.size() == (1<<n));
 // Apply odd-negation transform
 for(int mask = 0; mask < (1 << n); mask++) {
  if((__builtin_popcount(mask) % 2) == 1) {
   f[mask] *= -1;
  }
 }

 // Apply zeta transform
 for(int i = 0; i < n; i++) {
  for(int mask = 0; mask < (1 << n); mask++) {
   if((mask & (1 << i)) != 0) {
    f[mask] += f[mask ^ (1 << i)];
   }
  }
 }

 // Apply odd-negation transform
 for(int mask = 0; mask < (1 << n); mask++) {
  if((__builtin_popcount(mask) % 2) == 1) {
   f[mask] *= -1;
  }
 }
 vector<int> mu(1<<n);
 for(int mask = 0; mask < (1 << n); mask++) mu[mask] = f[
      mask];

 return mu;
}
```

28 Segment Tree

```
struct dat
{
 int mn;
 dat() : mn(1e9) {};
};

struct SegTree
{
 int N;
 vector<dat> st;
 vector<bool> cLazy;
 vector<int> lazy;

 void init(int n){
  N = n;
  st=vector<dat>(4 * N + 5);
  cLazy=vector<bool>(4 * N + 5, false);
  lazy=vector<int>(4 * N + 5, 0);
 }

 //Write reqd merge functions
 void merge(dat &cur, dat &l, dat &r) {
  cur.mn = min(l.mn, r.mn);
 }

 // Handle lazy propagation appriopriately
 //  think what should happen if clazy[2*node] is already 1
      before propagate is called
 void propagate(int node, int L, int R){
  if(L != R){
   cLazy[node*2] = 1;
   cLazy[node*2 + 1] = 1;
   lazy[node*2] = lazy[node];
   lazy[node*2 + 1] = lazy[node];
  }
  st[node].mn = lazy[node];
  cLazy[node] = 0;
 }
dat Query(int node, int L, int R, int i, int j){
 if(cLazy[node])
  propagate(node, L, R);
 if(j<L || i>R)
  return dat();
 if(i<=L && R<=j)
  return st[node];
 int M = (L + R)/2;
 dat left=Query(node*2, L, M, i, j);
 dat right=Query(node*2 + 1, M + 1, R, i, j);
 dat cur;
 merge(cur, left, right);
 return cur;
}

dat pQuery(int node, int L, int R, int pos){
 if(cLazy[node])
  propagate(node, L, R);
 if(L == R)
  return st[node];
 int M = (L + R)/2;
 if(pos <= M)
  return pQuery(node*2, L, M, pos);
 else
  return pQuery(node*2 + 1, M + 1, R, pos);
}

void Update(int node, int L, int R, int i, int j, int val){
 if(cLazy[node])
  propagate(node, L, R);
 if(j<L || i>R)
  return;
 if(i<=L && R<=j){
  cLazy[node] = 1;
  lazy[node] = val;
  propagate(node, L, R);
  return;
 }
 int M = (L + R)/2;
 Update(node*2, L, M, i, j, val);
 Update(node*2 + 1, M + 1, R, i, j, val);
 merge(st[node], st[node*2], st[node*2 + 1]);
}

void pUpdate(int node, int L, int R, int pos, int val){
 if(cLazy[node])
  propagate(node, L, R);
 if(L == R){
  change_here
  return;
 }
 int M = (L + R)/2;
 if(pos <= M)
  pUpdate(node*2, L, M, pos, val);
 else
  pUpdate(node*2 + 1, M + 1, R, pos, val);
 dat tleft=Query(2*node,L,M,L,M);
 dat tright=Query(2*node+1,M+1,R,M+1,R);
 merge(st[node],tleft,tright);
}

dat query(int pos){
 return pQuery(1, 1, N, pos);
}

dat query(int l, int r){
 return Query(1, 1, N, l, r);
}

void update(int pos, int val){
 pUpdate(1, 1, N, pos, val);
}

void update(int l, int r, int val){
 Update(1, 1, N, l, r, val);
}
};
```

29 Steiner tree

```
/*
n=number of steiner node(the optional intermediate nodes)
k=number of terminal nodes.(all nodes that has to be
     connected)
dp[i][mask]=minimum cost that the root is i and contains all
      terminal nodes which are set in mask.
g[i][j]=minimum cost from steiner node i to steiner node j.
     Run floyd warshall initially for finding g[][]
*/

for(int i=0;i<n;i++){
 for(int j=0;j<k;j++){
  dp[i][(1<<j)]= ;// distance from steiner node i to
       terminal node j, base case for dp.
 }
 dp[i][0]=0;
}
for(int mask=1;mask<(1<<k);mask++){
 for(int i=0;i<n;i++){
  for(int ss=mask;ss>0;ss=(ss-1)&mask){
   dp[i][mask]=min(dp[i][mask],dp[i][ss]+dp[i][mask-ss]);
  }
  if(dp[i][mask]<inf){
   for(int j=0;j<n;j++){
    dp[j][mask]=min(dp[j][mask],dp[i][mask]+g[i][j]);
   }
  }
 }
}
```

30 Suffix Automaton

```
class SuffixAutomaton{
    private:
        map <int,int> c;
    public:
        vector <map<char,int> > edges; vector <int> link;
        vector <int> length; map <int,int> terminals;
        vector <int> count;
        int last;

        SuffixAutomaton(string s) {
        edges.push_back(map<char,int>());
        link.push_back(-1);
        length.push_back(0);
        last = 0;
        for(int i=0;i<s.size();i++) {
            edges.push_back(map<char,int>());
            length.push_back(i+1);
            link.push_back(0);
            int r = edges.size() - 1;
            int p = last;
            while(p >= 0 && edges[p].find(s[i]) == edges[p].
                 end()) {
                edges[p][s[i]] = r;
                p = link[p];
            }
            if(p != -1) {
                int q = edges[p][s[i]];
                if(length[p] + 1 == length[q]) {
                    link[r] = q;
                }
                else {
                    edges.push_back(edges[q]);
                    length.push_back(length[p] + 1);
            link.push_back(link[q]);
            int qq = edges.size()-1;
            c[qq]=1;
            link[q] = qq;
            link[r] = qq;
            while(p >= 0 && edges[p][s[i]] == q) {
                edges[p][s[i]] = qq;
                p = link[p];
            }
        }
    }
    last = r;
}
set <pair<int,int> > base;c[0]=0;
for(int i=0;i<sz(length);i++){
    if(c[i]==1){
        count.push_back(0);
        base.insert(make_pair(length[i],i));}
    else{count.push_back(1);
        base.insert(make_pair(length[i],i));}
}
for(auto itr=base.rbegin();itr!=base.rend();itr++){
    count[link[itr->second]]+=count[itr->second];
}
int p = last;
while(p > 0) {
    terminals[p]=1;
    p = link[p];
}
    }
};
```

31 Trie

```
typedef struct data
{
 data* bit[2];
 int cnt = 0;
}trie;

void insert(trie* &head,int x){
 if(!head){
  head=new trie();
 }
 trie* cur = head;
 for(int i=30;i>=0;i--){
  int b = (x>>i) & 1;
  if(!cur->bit[b])
   cur->bit[b] = new trie();
  cur = cur->bit[b];
  cur->cnt++;
 }
}

//use only for persistent trie
void fork_new_trie(trie* &head,trie* par, int val){
 head=new trie();
 trie* cur=head;
 for(int i=30;i>=0;i--){
  int b=(val>>i)&1;
  cur->bit[b]=new trie();
  cur->bit[b]->cnt++;
  if(par){
   cur->bit[!b]=par->bit[!b];
  }
  cur=cur->bit[b];
  if(par){
   par=par->bit[b];
  }
 }
}
```

32 bitset doc

```
#define M 32
int main()
{
    // default constructor initializes with all bits 0
    bitset<M> bset1;

    // bset2 is initialized with bits of 20
    bitset<M> bset2(20);

    // bset3 is initialized with bits of specified binary
         string
    bitset<M> bset3(string("1100"));

    // cout prints exact bits representation of bitset
    cout << bset1 << endl; //
         00000000000000000000000000000000
    cout << bset2 << endl; //
         00000000000000000000000000010100
    cout << bset3 << endl; //
         00000000000000000000000000001100
    cout << endl;

// declaring set8 with capacity of 8 bits

bitset<8> set8; // 00000000

// setting first bit (or 6th index)
set8[1] = 1; // 00000010
set8[4] = set8[1]; // 00010010
cout << set8 << endl;

// count function returns number of set bits in bitset
int numberof1 = set8.count();

// size function returns total number of bits in bitset
// so there difference will give us number of unset(0)
// bits in bitset
int numberof0 = set8.size() - numberof1;

cout << set8 << " has " << numberof1 << " ones and "
     << numberof0 << " zeros\n";

// test function return 1 if bit is set else returns 0
cout << "bool representation of " << set8 << " : ";
for (int i = 0; i < set8.size(); i++)
    cout << set8.test(i) << " ";

cout << endl;

// any function returns true, if atleast 1 bit
// is set
if (!set8.any())
    cout << "set8 has no bit set.\n";

if (!bset1.any())
    cout << "bset1 has no bit set.\n";

// none function returns true, if none of the bit
// is set
if (!bset1.none())
    cout << "bset1 has some bit set\n";

// bset.set() sets all bits
cout << set8.set() << endl;

// bset.set(pos, b) makes bset[pos] = b
cout << set8.set(4, 0) << endl;

// bset.set(pos) makes bset[pos] = 1 i.e. default
// is 1
cout << set8.set(4) << endl;
```

```
    // reset function makes all bits 0
    cout << set8.reset(2) << endl;
    cout << set8.reset() << endl;

    // flip function flips all bits i.e. 1 <-> 0
    // and 0 <-> 1
    cout << set8.flip(2) << endl;
    cout << set8.flip() << endl;

    // Converting decimal number to binary by using bitset
    int num = 100;
    cout << "\nDecimal number: " << num
         << " Binary equivalent: " << bitset<8>(num);

    return 0;
}

int main()
{
    bitset<4> bset1(9); // bset1 contains 1001
    bitset<4> bset2(3); // bset2 contains 0011

    // comparison operator
    cout << (bset1 == bset2) << endl; // false 0
    cout << (bset1 != bset2) << endl; // true 1

    // bitwise operation and assignment
    cout << (bset1 ^= bset2) << endl; // 1010
    cout << (bset1 &= bset2) << endl; // 0010
    cout << (bset1 |= bset2) << endl; // 0011

    // left and right shifting
    cout << (bset1 <<= 2) << endl; // 1100
    cout << (bset1 >>= 1) << endl; // 0110

    // not operator
    cout << (~bset2) << endl; // 1100

    // bitwise operator
    cout << (bset1 & bset2) << endl; // 0010
    cout << (bset1 | bset2) << endl; // 0111
    cout << (bset1 ^ bset2) << endl; // 0101
}
```

33 centroid decomposition on tree

```
int nodes = 0;
int subtree[N], parentcentroid[N];
set<int> g[N];


void dfs(int u, int par){
 nodes++;
 subtree[u] = 1;
 for(auto &it:g[u]){
  if(it == par)
   continue;
  dfs(it, u);
  subtree[u] += subtree[it];
 }
}

int centroid(int k, int parent){
 for(auto it:g[k]){
  if(it==parent)
   continue;
  if(subtree[it]>(nodes>>1))
   return centroid(it,k);
 }
 return k;
}

void decompose(int u, int par){
 nodes = 0;
 dfs(u, u);
 int node = centroid(u, u);
 //do something
 parentcentroid[node] = par;
 for(auto &it:g[node]){
  g[it].erase(node);
  decompose(it, node);
 }
}


/*
Properties of Centroid Tree(VERY IMP): https://www.quora.com
     /q/threadsiiithyderabad/Centroid-Decomposition-of-a-
     Tree
*/
```

34 chinese remainder theorem

```
/*
Use #define int long long or BigInteger Library in case of
     Large numbers possible.
*/
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<int,int> CRT(vector<int> a,vector<int> b)// finds ans
     such that (ans=a[i]) mod(b[i]). IF NOT possible,
     returns {-1,-1}
{
    int t=a.size();
    for(int i=0;i<t;i++)
    {
        normalize(a[i],b[i]);
    }
    int ans=a[0];
    int lcm=b[0];
    for(int i=1;i<t;i++)
    {
        auto pom = ex_GCD(lcm, b[i]);
        int x1 = pom.x;
        int d = pom.d;
        if((a[i] - ans) % d != 0) return cerr << "No
             solutions" << endl, make_pair(-1,-1);
        ans = normalize(ans + x1 * (a[i] - ans) / d % (b[i] /
              d) * lcm, lcm * b[i] / d);
        lcm = LCM(lcm, b[i]); // you can save time by
             replacing above lcm * n[i] /d by lcm = lcm * n[i
             ] / d
    }
    return make_pair(ans,lcm);
}
```

35 combinatorics important

```
/*
1) n distinct objects in k distinct boxes: k^n.
2) n distinct objects in k identical boxes: S(n, r) or B_n ,
     stirling number of second kind or bell number.
    Description below
```

```
3) n identical objects in k distinct boxes: (n+k-1)C(k-1);
4) n identical objects in k identical boxes giving atleast
    one to each of the box:
 can be calculated by in O(n*k) dp using the following
     theorem:
 The number of partitions of an integer nn into rr parts is
     the same as the number of partitions of nn for which
     the largest part is rr. Intuition is : think of it as
     histogram where x is upto r and taking the lowest row
     each time will reduce it by r forming smaller
     subproblem.
 Thereafter the dp relation becomes: dp[n][r]=summation(dp[n
     -r][k]) for k=1 to k=r
 we can use prefix sum and calculate each dp[n][r] in O(1)
     giving the final complexity O(n*k).
 the whole idea is given on Brilliant.org


Stirling number of second kind : A Stirling number of the
    second kind, denoted as S(n, r) is the number of ways
    to partition n objects in r non-empty sets.

S(n, r) = r * S(n-1, r) + S(n-1, r-1)
Base cases :
S(n, r) = 1,if r == n
S(n, r) = 0 if r > n

Bell numbers : B_n is the number of ways of arranging n
    distinct objects into upto n identical non-empty bins.
B_n = S(n, 1) + S(n, 2) + ... + S(n, n) */
```

36 compression of undirected graph into tree

```
// Idea is to compress two way connected components of an
    undirected connected graph into a tree. So all the set
    of nodes that have two or more ways to reach one from
    another is compressed to one node.

// Algorithm:
// 1) Make a recursion function to create a directed graph:
void construct_directed_graph(int node,int prev){
    //t(node);
    if(vis[node])
     return;
    vis[node]=1;
    for(auto &x: undirected[node]){
        if(x==prev)
         continue;
        adj[node].push_back(x);
        adjtranspose[x].push_back(node);
        construct_directed_graph(x,node);
    }
}
// 2) This will create a directed graph.
// 3) we can compress all the components in this directed
     graph if they are Strongly connected components.
// Use condensation_of_directed_graph_using_SCC.cpp
// 4) To get back tree, modify the code from
     condensation_of_directed_graph_using_SCC.cpp
```

37 condensation of a directed graph using SCC

```
struct Condense_graph_SCC //WARNING: 1-based indexing
{
    int n;
    vector<vector<int>> adj, adjtranspose;
    vector<bool> visited1, visited2;
    stack<int> st;
    vector<int> color;
    vector<vector<int>> SCC; // Beware of Multi-edge between
         two SCC's
    int c=0;                  //number of colors
    void init(int r)          // pass the exact number of
         nodes n from 1 to n in consideration.
    {
        adj.clear();
        adjtranspose.clear();
        visited1.clear();
        visited2.clear();
        st.clear();
        color.clear();
        SCC.clear();
        n=r;
        adj.resize(r+5);
        adjtranspose.resize(r+5);
        visited1.resize(r+5,false);
        visited2.resize(r+5,false);
        color.resize(r+5,-1);
        c=0;
    }
    void add_edge(int u,int v){
        adj[u].push_back(v);
        adjtranspose[v].push_back(u);
    }
    void dfs1(int node){
        visited1[node]=true;
        for(auto x: adj[node]){
            if(!visited1[x]){
                dfs1(x);
            }
        }
        st.push(node);
    }
    void dfs2(int node,int c){
        visited2[node]=true;
        color[node]=c;
        for(auto x: adjtranspose[node]){
            if(!visited2[x])
                dfs2(x,c);
        }
    }
    void condense_directed_graph()// adj and adjtranspose
         should already be constructed before calling this
         function
    {
        for(int i=1;i<=n;i++){
            if(!visited1[i]){
                dfs1(i);
            }
        }
        while(!st.empty()){
            int x=st.top();
            st.pop();
            if(!visited2[x]){
                dfs2(x,c++);
            }
        }
        SCC.resize(c);
        for(int i=1;i<=n;i++){
            for(auto x: adj[i]){
                if(color[i]!=color[x]){
                    SCC[color[i]].push_back(color[x]);
                }
            }
        }
    }
};
```

38 convex hull trick

```
struct ConvexHullDynamic{
 static const int INF=1e18;
 struct Line{
  int a, b; //y = ax + b
  double xLeft; //Stores the intersection wiith previous
      line in the convex hull. First line has -INF

  enum Type {line, maxQuery, minQuery} type;
  int val;

  explicit Line(int aa=0, int bb=0): a(aa), b(bb), xLeft(-
      INF), type(Type::line), val(0) {}

  int valueAt(int x) const{
   return a*x + b;
  }
  friend bool isParallel(const Line &l1, const Line &l2){
   return l1.a == l2.a;
  }
  friend double intersectX(const Line &l1, const Line &l2){
   return isParallel(l1, l2)?INF:1.0*(l2.b-l1.b)/(l1.a-l2.a)
      ;
  }
  bool operator<(const Line& l2) const{
   if(l2.type == line)
    return this->a < l2.a;
   if(l2.type == maxQuery)
    return this->xLeft < l2.val;
   if(l2.type == minQuery)
    return this->xLeft > l2.val;
  }
 };

 bool isMax;
 set<Line> hull;

 bool hasPrev(set<Line>::iterator it){
  return it!=hull.begin();
 }
 bool hasNext(set<Line>::iterator it){
  return it!=hull.end() && next(it)!=hull.end();
 }
 bool irrelevant(const Line &l1, const Line &l2, const Line
     &l3){
  return intersectX(l1, l3) <= intersectX(l1, l2);
 }
 bool irrelevant(set<Line>::iterator it){
  return hasPrev(it) && hasNext(it) && (
   (isMax && irrelevant(*prev(it), *it, *next(it)))
   || (!isMax && irrelevant(*next(it), *it, *prev(it))));
}
//Updates xValue of line pointed by it
set<Line>::iterator updateLeftBorder(set<Line>::iterator it
    ){
 if(isMax && !hasPrev(it) || !isMax && !hasNext(it))
  return it;
 double val=intersectX(*it, isMax?(*prev(it)):(*next(it)));
 Line temp(*it);
 it=hull.erase(it);
 temp.xLeft=val;
 it=hull.insert(it, temp);
 return it;
}

explicit ConvexHullDynamic(bool isMax): isMax(isMax) {} //
    isMax : 0 for min-convex hull, 1 for max-convex hull

//Add ax + b in logN time
void addLine(int a, int b) {
 Line l3=Line(a, b);
 auto it=hull.lower_bound(l3);

 //If parallel liune is already in set, one of the lines
      becomes irrelevant
 if(it!=hull.end() && isParallel(*it, l3)){
  if(isMax && it->b<b || !isMax && it->b>b)
   it=hull.erase(it);
  else
   return;
 }

 it=hull.insert(it, l3);
 if(irrelevant(it)){
  hull.erase(it);
  return;
 }

 //Remove lines which became irrelevant after inserting
 while(hasPrev(it) && irrelevant(prev(it)))
  hull.erase(prev(it));
 while(hasNext(it) && irrelevant(next(it)))
  hull.erase(next(it));

 //Update xLine
 it=updateLeftBorder(it);
 if(hasPrev(it))
  updateLeftBorder(prev(it));
 if(hasNext(it))
  updateLeftBorder(next(it));
}
 int getBest(int x){
  Line q;
  q.val=x;
  q.type = isMax?Line::Type::maxQuery : Line::Type::minQuery
      ;

  auto bestLine=hull.lower_bound(q);
  if(isMax)
   --bestLine;
  return bestLine->valueAt(x);
 }
};
```

39 cp basefile

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;

template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     vector<T> & vec);
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     set<T> & vec);
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     unordered_set<T> & vec);
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     multiset<T> & vec);
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     unordered_multiset<T> & vec);
template<typename T1, typename T2>
std::ostream & operator << (std::ostream & os, const std::
     pair<T1,T2> & p);
template<typename T1, typename T2>
std::ostream & operator << (std::ostream & os, const std::
     map<T1,T2> & p);
template<typename T1, typename T2>
std::ostream & operator << (std::ostream & os, const std::
     unordered_map<T1,T2> & p);
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     vector<T> & vec){
    os<<"{";
    for(auto elem : vec)
        os<<elem<<",";
    os<<"}";
    return os;
}
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     set<T> & vec){
    os<<"{";
    for(auto elem : vec)
        os<<elem<<",";
    os<<"}";
    return os;
}
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     unordered_set<T> & vec){
    os<<"{";
    for(auto elem : vec)
        os<<elem<<",";
    os<<"}";
    return os;
}
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     multiset<T> & vec){
    os<<"{";
    for(auto elem : vec)
        os<<elem<<",";
    os<<"}";
    return os;
}
template <typename T>
std::ostream & operator << (std::ostream & os, const std::
     unordered_multiset<T> & vec){
    os<<"{";
    for(auto elem : vec)
        os<<elem<<",";
    os<<"}";
    return os;
}
template<typename T1, typename T2>
std::ostream & operator << (std::ostream & os, const std::
     pair<T1,T2> & p){
    os<<"{"<<p.first<<","<<p.second<<"}";
    return os;
}
template<typename T1, typename T2>
std::ostream & operator << (std::ostream & os, const std::
     map<T1,T2> & p){
    os<<"{";
    for(auto x: p)
        os<<x.first<<"->"<<x.second<<", ";
    os<<"}";
    return os;
}
template<typename T1, typename T2>
std::ostream & operator << (std::ostream & os, const std::
     unordered_map<T1,T2> & p){
    os<<"{";
    for(auto x: p)
        os<<x.first<<"->"<<x.second<<", ";
    os<<"}";
    return os;
}
/*---------*/
#warning TODOt1-5
#define t6(a,b,c,d,e,f) cerr<<#a<<"="<<a<<" "<<#b<<"="<<b<<"
      "<<#c<<"="<<c<<" "<<#d<<"="<<d<<" "<<#e<<"="<<e<<" "
     <<#f<<"="<<f<<endl
#define GET_MACRO(_1,_2,_3,_4,_5,_6,NAME,...) NAME
#ifndef ONLINE_JUDGE
#define tr(...) GET_MACRO(__VA_ARGS__,t6,t5, t4, t3, t2, t1)
     (__VA_ARGS__)
#else
#define tr(...)
#endif
/*-------*/
#define __ freopen("input.txt","r",stdin);freopen("output.
     txt","w",stdout);
#define fastio() ios::sync_with_stdio(0);cin.tie(0)
#define MEMS(x,t) memset(x,t,sizeof(x));
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch
     ().count());
/*
     --------------------------------------------------------------------------------------------
     */
// #define MOD 1000000007
// #define MOD 998244353
#define endl "\n"
#define int long long
#define inf 1e18
#define ld long double
/*
     --------------------------------------------------------------------------------------------
     */

signed main()
{
    fastio();
```

```
}
```

40 custom comparator for set map

```
/*
bool cmp(datatype a,datatype b){
  ...
}
set<datatype,decltype(cmp)*> se(cmp); // for set
map<datatype,decltype(cmp)*> mp(cmp); // for map

Note : don't use lambda function for cmp.
*/
```

41 dijkstra

```
vector<int> dijkstra(int source,int destination)
{
 int n=adj.size();
 vector<int> var(n, inf);
 vector<int> fix(n);
 set<pair<int,int>> s;
 s.insert({0,source});
 fix[source]=0;
 var[source]=0;
 while(s.size()){
  pair<int,int> cur=*(s.begin());
  s.erase(cur);
  fix[cur.second]=cur.first;
  for(auto x: adj[cur.second]){
   if(var[x.first]>cur.first+x.second){
    s.erase({var[x.first],x.first});
    var[x.first]=cur.first+x.second;
    s.insert({var[x.first],x.first});
   }
  }
 }
 return fix;
}
```

42 euler-totient(phi)

```
//phi(n) in O(sqrt(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;
}
/*
Properties of phi function:
1) phi(n)= count of numbers less than n that are co prime to
      n.
2) if p is prime, phi(p)=p-1;
3) if a and b are coprime, phi(ab)=phi(a)*phi(b)
4) in general: phi(ab)=phi(a)*phi(b)*(d/phi(d)) : where d=
     gcd(a,b)
5) if p is prime: phi(p^k)=p^k-p^(k-1), k>=1
6) let n= (p1^a1)*(p2^a2)*(p3^a3)....*(pk^ak), then phi(n)=n
     *(1-1/p1)*(1-1/p2)....*(1-1/pk);
7) summation((phi(d))(for all d divisible by n))=n;
*/

/*
algorithm for finding euler totient function till nlog(log(n
     )).
Logic is similar to linear sieve.
*/
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    phi[0] = 0;
    phi[1] = 1;
    for (int i = 2; 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;
        }
    }
}
```

43 extended gcd

```
int extgcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = extgcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
/*
 * It is used for getting the coefficients x, y such that ax
      +by=gcd(a,b)
 *
 * this is particularly useful in solving linear diophantine
       equation.
 * check out cpalgorithms.com if you want the proof how it
      works.
 * it works even for negative value.
 * /
```

44 floor sum

```
/*
returns summation_{i=0}^{i=n-1} floor((a * i + b) / m)
Time complexity : O(log(m))
*/
long long floor_sum(long long n, long long m, long long a,
     long long b) {
    long long ans = 0;
    if (a >= m) {
        ans += (n - 1) * n * (a / m) / 2;
        a %= m;
    }
    if (b >= m) {
        ans += n * (b / m);
        b %= m;
    }

    long long y_max = (a * n + b) / m, x_max = (y_max * m - b
         );
    if (y_max == 0) return ans;
    ans += (n - (x_max + a - 1) / a) * y_max;
    ans += floor_sum(y_max, a, m, (a - x_max % a) % a);
    return ans;
}
```

45 game theory

```
/*
How to check if our classification of winning and losing
     state is correct:
1) Final position should be losing position.
2) In winning position it's possible to go to atleast one
     losing position.
3)In losing position we are forced to go to winning position
     .


Minimum Excludant(MEX):
1) Operation performed on a set.
2) returns the smallest number not present in the set.

Grundy numbers:
Grundy is the MEX of a set of moves that we can play.
Defines the entire game state.
Calculated as follows:
 1)take the Mex of set.
 2)If set is empty: mex(null set)=0
  Grundy(losing condition=0
 3)Grundy(x)=MEX({Grundy of reachable states}).
  if a reachable state in (3) consist of multiple piles,to
       find grundy of that reachable state,
  take xor of grundy of each pile.
 4) a set of piles can be converted to a single pile with
      the value as the xor of the set


How to find if a state is winning state or losing state:

1) If there are independent sub-games involved, we can take
     xor sum of grundy numbers of subgames
 and the xorsum we get is the grundy number of overall game.
2) If Grundy is 0, losing state, else winning state.
*/
```

46 geometry

```
const long double eps = 1e-12;
const long double infl = 1e20;

template <class T>
inline int sgn(const T & x) {
 return (x > eps) - (x < -eps);
}

template <class T>
struct Point {
  T x, y;

 Point() : x(0), y(0) {}

 Point(const T & x, const T & y) : x(x), y(y) {}

 long double length() const {
  return sqrtl(x * x + y * y);
 }

 T length2() const {
  return x * x + y * y;
 }

 long double distance(const Point & rhs) const {
  return (rhs - *this).length();
 }

  T cross(const Point & lhs, const Point & rhs) const {
  return (lhs - *this) * (rhs - *this);
 }

 T dot(const Point & lhs, const Point & rhs) const {
  return (lhs - *this) & (rhs - *this);
 }

 long double distance_to_line(const Point & p, const Point &
       q) const {
  if (p == q) return distance(p);
  else return fabsl(cross(p, q) / p.distance(q));
 }

 long double distance_to_segment(const Point & p, const
      Point & q) const {
  if (p == q) return distance(p);
  else if (p.dot(q, *this) < 0) return distance(p);
  else if (q.dot(p, *this) < 0) return distance(q);
  else return distance_to_line(p, q);
 }
```

```
bool on_line(const Point & p, const Point & q) const {
 return sgn(cross(p, q)) == 0;
}

bool on_halfline(const Point & p, const Point & q, const
     bool & inclusive = true) const {
 if (*this == p) return inclusive;
 else return on_line(p, q) && sgn(p.dot(q, *this)) >= 0;
}

bool on_segment(const Point & p, const Point & q, const
     bool & inclusive = true) const {
 if (*this == p || *this == q) return inclusive;
 else return on_line(p, q) && sgn(dot(p, q)) <= 0;
}

bool in_triangle(const Point & u, const Point & v, const
     Point & w, const bool & inclusive = true) const {
 Point p[3] = {u, v, w};
 if (sgn(u.cross(v, w)) < 0) reverse(p, p + 3);
 for (int i = 0; i < 3; i++) {
  if (on_segment(p[i], p[(i + 1) % 3])) return inclusive;
  else if (sgn(cross(p[i], p[(i + 1) % 3])) < 0) return
       false;
 }
 return true;
}

bool in_angle(const Point & o, const Point & p, const Point
      & q, const bool & inclusive = true) const {
 if (on_halfline(o, p) || on_halfline(o, q)) return
      inclusive;
 int vs = sgn(o.cross(p, q)), ss = sgn(o.dot(p, q));
 if (vs == 0 && ss > 0) return false;
 int vp = sgn(o.cross(p, *this)), vq = sgn(o.cross(*this, q
      ));
 if (sgn(o.cross(p, q)) >= 0) return vp > 0 && vq > 0;
 else return vp > 0 || vq > 0;
}

Point operator + (const Point & rhs) const {// returns a+b(
     vector addition)
 return Point(x + rhs.x, y + rhs.y);
}

Point operator - (const Point & rhs) const {// Returns a-b(
     vector subtraction)
 return Point(x - rhs.x, y - rhs.y);
}
```

```
T operator * (const Point & rhs) const {  // cross product
 return x * rhs.y - y * rhs.x;
}

T operator & (const Point & rhs) const {  // dot product
 return x * rhs.x + y * rhs.y;
}

Point operator * (const T & rhs) const { // multiplication
     with constant
 return Point(x * rhs, y * rhs);
}

Point operator / (const T & rhs) const { // division with
     constant
 return Point(x / rhs, y / rhs);
}

bool operator == (const Point & rhs) const {// returns true
      if points are located at the same point.
 return x == rhs.x && y == rhs.y;
}

bool operator != (const Point & rhs) const {// returns true
      if points are not located at same point.
 return x != rhs.x || y != rhs.y;
}

inline int quadrant() const {    // returns quadrant
     defined from 0 to 7 (-1 if origin). Used in operator <
 if (x == 0 && y == 0) return -1;
 else if (x < 0 && y < 0) return 0;
 else if (x == 0 && y < 0) return 1;
 else if (x > 0 && y < 0) return 2;
 else if (x > 0 && y == 0) return 3;
 else if (x > 0 && y > 0) return 4;
 else if (x == 0 && y > 0) return 5;
 else if (x < 0 && y > 0) return 6;
 else return 7;
}

bool operator < (const Point & rhs) const { //used for
     sorting the points in counter clockwise order.
 int lq = quadrant(), rq = rhs.quadrant();
 if (lq != rq) {
  return lq < rq;
 } else {
  int s = sgn(*this * rhs);
  return s != 0 ? s > 0 : sgn(length2() - rhs.length2()) <
      0;
  }
 }
};

template <class T>
inline bool xycmp(const Point<T> & lhs, const Point<T> & rhs
     ) {
 if (lhs.x != rhs.x) return lhs.x < rhs.x;
 else return lhs.y < rhs.y;
}

template <class T>
void convex_hull(vector<Point<T> > p, vector<Point<T> > &
     ans) // ans[0] is the leftmost and downmost point,
     order is ACW.
{
 const int n = p.size();
 Point<T> o = *min_element(p.begin(), p.end(), xycmp<T>);
 for(int i = 0; i < n; i++) p[i] = p[i] - o;
 sort(p.begin(), p.end());
 for (int i = 0; i < n; i++) p[i] = o + p[i];
 ans.resize(n);
 int sz = 0;
 ans[sz++] = p[0];
 for(int i = 1; i < n; i++) {
  for(; sz > 1 && sgn(ans[sz - 2].cross(ans[sz - 1], p[i]))
       <= 0; sz--);
  ans[sz++] = p[i];
 }
 ans.resize(sz);
}

template <class T>
bool point_in_convex_polygon(const vector<Point<T> > & p,
     const Point<T> & q, const bool & inclusive = true)
{
 const int n = p.size();
 if (n == 1) {
  return inclusive && p[0] == q;
 } else if (n == 2) {
  return inclusive && q.on_segment(p[0], p[1]);
 } else if (!q.in_angle(p[0], p[1], p[n - 1])) {
  return false;
 } else if (q.on_line(p[0], p[1]) || q.on_line(p[0], p[n -
      1])) {
  return inclusive && (q.on_segment(p[0], p[1]) || q.
       on_segment(p[0], p[n - 1]));
 } else {
  int low = 1, high = n - 1;
  for (; low < high - 1; ) {
   int mid = low + (high - low) / 2;
   (sgn(q.cross(p[0], p[mid])) < 0 ? high : low) = mid;
  }
  if (q.on_segment(p[low], p[high])) {
   return inclusive;
  } else {
   return q.in_triangle(p[0], p[low], p[high]);
  }
 }
}

template <class T, class U>
inline T & minimize(T & a, const U & b) {
 if (b < a) a = b;
 return a;
}

template <class T>
Point<T> line_line_intersection(const Point<T> & p1, const
     Point<T> & p2, const Point<T> & q1, const Point<T> & q2
     ) {
 T pv1 = p1.cross(p2, q1), pv2 = p1.cross(p2, q2);
 return q1 * (-pv2 / (pv1 - pv2)) + q2 * (pv1 / (pv1 - pv2))
      ;
}

template <class T>
struct Line {
 Point<T> u, v;

 Line() {

 }

 Line(const Point<T> & u, const Point<T> & v) : u(u), v(v) {

 }

 Line(const T & px, const T & py, const T & qx, const T & qy
      ) : u(px, py), v(qx, qy) {

 }

 T operator * (const Line & rhs) const {
  return (v - u) * (rhs.v - rhs.u);
 }

 T operator & (const Line & rhs) const {
  return (v - u) & (rhs.v - rhs.u);
 }

 bool operator < (const Line & rhs) const {
  int vs = sgn((v - u) * (rhs.v - rhs.u)), ss = sgn((v - u)
       & (rhs.v - rhs.u));
  if (vs == 0 && ss > 0) return on_left(rhs.u, false);
  else return v - u < rhs.v - rhs.u;
 }

 template <class U>
 bool on_left(const Point<U> & p, const bool & inclusive =
      true) const {
  return p.cross(u, v) > 0;
 }

 template <class U>
 bool on_right(const Point<U> & p, const bool & inclusive =
      true) const {
  return p.cross(u, v) < 0;
 }
};

typedef vector<Point<ld>> Polygon;
long double signed_area(Polygon p)
{
 long double area=0;
 for(int i=0;i<p.size();i++)
 {
  int j=i+1;
  j%=p.size();
  area+=p[i].x*p[j].y-p[j].x*p[i].y;
 }
 return area/2.0;
}
long double area(const Polygon &p)
{
 return fabsl(signed_area(p));
}

template <class T>
Point<T> line_line_intersection(const Line<T> & lhs, const
     Line<T> & rhs) {
 return line_line_intersection(lhs.u, lhs.v, rhs.u, rhs.v);
}

/*
Useful points:
 Brahmagupta formula: area of quadrilateral with sides s1,s2
      ,s3,s4 is given by:
 s=(s1+s2+s3+s4)/2.0;
 Area=sqrt((s-s1)*(s-s2)*(s-s3)*(s-s4));
```

```
*/
```

47 hierholzer euler tour

```
/* For Directed Graph:*/
vector<int> ptr;//remembers the current ptr to each node
     adjacency list
vector<vector<int>> adj;
vector<int> tour;//the tour is stored in reverse order. Don'
     t forget to reverse it after completing dfs.
void dfs(int node)
{
 while(ptr[node]<adj[node].size())
 {
  int nxt=adj[node][ptr[node]++];
  dfs(nxt);
 }
 tour.push_back(node);
}

/* For Undirected Graph: */
vector<int> ptr;
vector<vector<pair<int,int>>> adj; //adj[u].push_back({v,
     edge_no});adj[v].push_back({u,edge_no});
vector<int> tour;
vector<bool> mark;// remembers whether we have visited the
     undirected or not.

void dfs(int node)
{
 while(ptr[node]<adj[node].size())
 {
  auto nxt=adj[node][ptr[node]++];
  if(mark[nxt.second])continue;
  mark[nxt.second]=1;
  dfs(nxt.first);
 }
 tour.push_back(node);
}
```

48 isprime

```
bool isPrime(int n)
{
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n%2 == 0 || n%3 == 0) return false;
    for (int i=5; i*i<=n; i=i+6)
        if (n%i == 0 || n%(i+2) == 0)
           return false;
    return true;
}
```

49 matrix exponentiation

```
struct Matrix_exponentiation
{
 int siz;
 int m_phi=0;
 vector<vector<int>> matrix;


 Matrix_exponentiation(int si,vector<vector<int>> matr): siz
      (si),matrix(matr){};

 void multiply(vector<vector<int>> &a, vector<vector<int>> &
      b) {
  int mul[siz][siz];
  for (int i = 0; i < siz; i++) {
   for (int j = 0; j < siz; j++) {
    mul[i][j] = 0;
    for (int k = 0; k < siz; k++) {
     mul[i][j] += (a[i][k]*b[k][j]+m_phi)%MOD;
     if(mul[i][j] >= MOD)
      mul[i][j] -= MOD;
    }
   }
  }

  // storing the multiplication resul in a[][]
  for (int i=0; i<siz; i++)
   for (int j=0; j<siz; j++)
    a[i][j] = mul[i][j]; // Updating our matrix
 }
 // Function to compute F raise to power n.
 void power(vector<vector<int>> &F, int n)
 {
  vector<vector<int>> Ma=matrix;
  if (n==1)
   return;

  power(F, n/2);

  multiply(F, F);

  if (n%2 != 0)
   multiply(F, Ma);
 }
 vector<vector<int>> findNthTerm(int n)
 {
  vector<vector<int>> F=matrix;
  power(F,n);
  return F;
 }
};
```

50 min cost bipartite matching

```
 // Min cost bipartite matching via shortest augmenting
      paths
//
// This is an O(n^3) implementation of a shortest augmenting
      path
// algorithm for finding min cost perfect matchings in dense
// graphs. In practice, it solves 1000x1000 problems in
     around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right
     node j
//   Lmate[i] = index of right node that left node i pairs
     with
//   Rmate[j] = index of left node that right node j pairs
     with
//
// The values in cost[i][j] may be positive or negative. To
     perform
// maximization, simply negate the cost[][] matrix.

//TODO: Initialize Lmate and Rmate as -1 before passing as
     argument.(uncommented so that i see)

// NOTE: Remove #define int long long if not necassary as
     the time taken here might be large when n=1000.

typedef vector<int> vi;
typedef vector<vi> vvi;

int MinCostMatching(const vvi &cost, vi &Lmate, vi &Rmate)
{
 int n = (int)(cost.size());
vi u(n);
vi v(n);
for (int i = 0; i < n; i++){
 u[i] = cost[i][0];
 for (int j = 1; j < n; j++){
  u[i] = min(u[i], cost[i][j]);
 }
}
for (int j = 0; j < n; j++){
 v[j] = cost[0][j] - u[0];
 for (int i = 1; i < n; i++){
  v[j] = min(v[j], cost[i][j] - u[i]);
 }
}
Lmate = vi(n, -1);
Rmate = vi(n, -1);
int mated = 0;
for (int i = 0; i < n; i++){
 for (int j = 0; j < n; j++){
  if (Rmate[j] != -1)
   continue;
  if (abs(cost[i][j] - u[i] - v[j]) == 0){
   Lmate[i] = j;
   Rmate[j] = i;
   mated++;
   break;
  }
 }
}
vi dist(n);
vi dad(n);
vi seen(n);
while (mated < n){
 int s = 0;
 while (Lmate[s] != -1)
  s++;
 fill(dad.begin(), dad.end(), -1);
 fill(seen.begin(), seen.end(), 0);
 for (int k = 0; k < n; k++)
  dist[k] = cost[s][k] - u[s] - v[k];
 int j = 0;
 while (true){
  j = -1;
  for (int k = 0; k < n; k++){
   if (seen[k])
    continue;
   if (j == -1 || dist[k] < dist[j])
    j=k;
  }
  seen[j] = 1;
   if (Rmate[j] == -1)
    break;
   const int i = Rmate[j];
   for (int k = 0; k < n; k++)
   {
    if (seen[k])
     continue;
    const int new_dist = dist[j] + cost[i][k] - u[i] - v[k];
    if (dist[k] > new_dist){
     dist[k] = new_dist;
     dad[k] = j;
    }
   }
  }
  for (int k = 0; k < n; k++)
  {
   if (k == j || !seen[k])
    continue;
   const int i = Rmate[k];
   v[k] += dist[k] - dist[j];
   u[i] -= dist[k] - dist[j];
  }
  u[s] += dist[j];
  while (dad[j] >= 0){
   const int d = dad[j];
   Rmate[j] = Rmate[d];
   Lmate[Rmate[j]] = j;
   j = d;
  }
  Rmate[j] = s;
  Lmate[s] = j;
  mated++;
 }
 int value = 0;
 for (int i = 0; i < n; i++)
  value += cost[i][Lmate[i]];
 return value;
}
```

51 mobius function

```
const int MAXN=1e7+100;
vector <int> prime;
bool is_composite[MAXN];
int mobius[MAXN];
void sieve (int n) {
 mobius[1] = 1;
 for (int i = 2; i < n; ++i) {
  if (!is_composite[i]) {
   prime.push_back (i);
   mobius[i] = -1;// according to definition.
  }
  for (int j = 0; j < prime.size () && i * prime[j] < n; ++j
      ) {
   is_composite[i * prime[j]] = true;
   if (i % prime[j] == 0) {
    mobius[i * prime[j]] = 0;
    break;
   } else {
    mobius[i * prime[j]] = mobius[i] * mobius[prime[j]];
   }
  }
 }
}

/*
O(n) implementation of calculating the mobius value from 1
    to n-1.

mobius[p^k] = [k=0] - [k=1] // where [] is boolean function
mobius[p1^k1 * p2^k2 * p3^k3 ... ] = mobius[p1^k1] * mobius[
    p2^k2] .... // mobius is multiplicative function

Mobius inversion states that :
if g(n) = summation_(d|n) f(d), then
f(n) = summation_(d|n) g(d) * mobius(n/d)

Most important property for mobius inversion :

(summation_(d|n) mobius(d)) = [n=1] , that is summation of
    mobius(d) for (divisors d of n) is 1 iff [n=1],
    otherwise 0

Few results :
1) Number of co-prime pair of integers (x, y) in range [1, n
    ] := summation_{d=1 to d=n} mobius(d) * floor(n/d)^2
2) Sum of gcd of every pair of integers (x, y) in range[1, n
    ] := summation_{k=1 to n} (number of co-prime pair of
    integers in range [1,n/k])

*/
```

52 pick's theorem

```
/*
```

```
Given a simple polygon constructed on a grid of equal
     distanced points
(i.e., points with integer coordinates) such that all
     polygon's vertices are
grid points, Pick's theorem provides a simple formula for
     calculating the
Area A of this polygon in terms of the number of interior
     points i and
boundary points b.

A= i + (b/2.0) -1;
i=number of interior point in polygon.
b=number of boundary point in polygon.

*/
```

53 power

```
long long poww(long long a, long long n, long long m)
{
        long long ans = 1;
        long long mul = a;
        while (n != 0){
                if (n % 2)
                        ans = (ans * mul) % m;
                mul = (mul * mul) % m;
                n /= 2;
        }
        return ans;
}
/*
1) (a/b)%mod= (a*(poww(b,phi(mod)-1,mod)))%mod iff a%b==0
2) (a^b)%mod= (a^(b%(mod-1)))%mod iff mod is prime
3) Fermat's little theorem:
        a^(phi(n))=1 mod(n) when a and n are relatively prime
        .
*/
```

54 simpson integration

```
const int N_steps = 3e7; // number of steps (already
     multiplied by 2)

ld f(ld x) //changes every time according to integration.
{}

ld simpson_integration(ld a, ld b){
    ld h = (b - a) / N_steps;
    ld s = f(a) + f(b); // a = x_0 and b = x_2n
    for (int i = 1; i <= N_steps - 1; ++i) { // Refer to
         final Simpson's formula
        ld x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

55 suffix array and LCP

```
vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;
    vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)
                 ) % n]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1
                 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
    }
    return p;
}
vector<int> suffix_array(string s) {
    s += "$";
    vector<int> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
/*
Uses of suffix_array:
1)Finding the smallest cyclic shift: The function
     sort_cyclic_shift(string s)
sorts all the cyclic shifts. Use that function and the p[0]
     gives the position
of smallest cyclic shift.

2) Finding a substring in a string:
The task is to find a string s inside some text t online -
     we know the text t
beforehand, but not the string s. We can create the suffix
     array for the text
t in O(|t|log|t|) time. Now we can look for the substring s
     in the following way.
The occurrence of s must be a prefix of some suffix from t.
     Since we sorted all the
suffixes we can perform a binary search for s in p.
     Comparing the current suffix and
the substring s within the binary search can be done in O(|s
     |) time, therefore the complexity
for finding the substring is O(|s|log|t|). Also notice that
     if the substring occurs multiple
times in t, then all occurrences will be next to each other
     in p. Therefore the number
of occurrences can be found with a second binary search, and
      all occurrences can be printed easily.
```

```
Returns lcp[i] = LCP(string starting at sa[i], '' '' '' sa[i
     +1])
where sa=suffix_array(string s)
*/
vector<int> lcp_construction(string const& s, vector<int>
     const& p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++)
        rank[p[i]] = i;
    int k = 0;
    vector<int> lcp(n-1, 0);
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
    return lcp;
}
/*
Use of LCP:
1) To find the LCP of string starting from sa[i] and sa[j],
find the min(lcp[i],lcp[i+1], ... , lcp[j-1]).
For efficiency, use segtree or sparse table for RMQ of lcp
     array.

2) Number of different substrings of a string of length n:
    get sa=suffix_array and lcp array. Then:
    int ans=0;
    ans+=(n*(n+1))/2;
    ans-=summation(lcp[0],lcp[1], ..., lcp[n-2]);
    return ans;
    check the logic from cp-algorithms if forgotten.
*/
```

56 tree order statistics

```
// Dont use #define int long long when working with these.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
#define T // Type of data structure: example-(int, pair<int,
     int>)
#define ordered_set tree<T, null_type, less<T>, rb_tree_tag,
      tree_order_statistics_node_update>
/* how to use:
*X.find_by_order(1)
X.order_of_key(400)
 * }
 * */
```

57 z function

```
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;
}

// * An element Z[i] of Z array stores length of the longest
      substring
// * starting from str[i] which is also a prefix of str[0..n
     -1]. The first
// * Z[0] is meaningless as complete string is always prefix
      of itself.
```