Codebook- Team Far_Behind
IIT Delhi, India

Ayush Ranjan, Naman Jain, Manish Tanwar

Contents

1 Syntax

1.1 Template

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
```

```
using namespace __gnu_pbds;
using namespace std;
template<class T> ostream& operator<<(ostream &os, ↩
   vector<T> V) {
 os << "[ "; for(auto v : V) os << v << " "; return ↩
    os << "]";}
template<class L, class R> ostream& operator<<(↩
   ostream &os, pair<L,R> P) {
  return os << "(" << P.first << "," << P.second << "↩
     )";}
#define TRACE
#ifdef TRACE
#define trace(...) __f(#__VA_ARGS__, __VA_ARGS__)
template <typename Arg1>
void __f(const char* name, Arg1&& arg1){
  cout << name << " : " << arg1 << std::endl;
}
template <typename Arg1, typename... Args>
void __f(const char* names, Arg1&& arg1, Args&&... ↩
   args){
  const char* comma = strchr(names + 1, ',');cout.↩
     write(names, comma - names) << " : " << arg1<<" ↩
     | ";__f(comma+1, args...);
}
#else
#define trace(...) 1
#endif
#define ll long long
#define ld long double
#define vll vector<ll>
#define pll pair<ll,ll>
#define vpll vector<pll>
#define I insert
#define pb push_back
#define F first
#define S second
#define all(x) x.begin(),x.end()
#define endl "\n"
// const ll MAX=1e6+5;
// int mod=1e9+7;
inline int mul(int a,int b){return (a*1ll*b)%mod;}
inline int add(int a,int b){a+=b;if(a>=mod)a-=mod;↩
   return a;}
inline int sub(int a,int b){a-=b;if(a<0)a+=mod;return↩
    a;}
inline int power(int a,int b){int rt=1;while(b>0){if(↩
   b&1)rt=mul(rt,a);a=mul(a,a);b>>=1;}return rt;}
inline int inv(int a){return power(a,mod-2);}
inline void modadd(int &a,int b){a+=b;if(a>=mod)a-=↩
   mod;}
int main(){
  ios_base::sync_with_stdio(false);cin.tie(0);cout.↩
     tie(0);cout<<setprecision(25);
}
// clock
clock_t clk = clock();
clk = clock() - clk;
((ld)clk)/CLOCKS_PER_SEC
// fastio
```

```
inline ll read() {
    ll n = 0; char c = getchar_unlocked();
    while (!('0' <= c && c <= '9')) c = ↩
       getchar_unlocked();
    while ('0' <= c && c <= '9')
        n = n * 10 + c - '0', c = getchar_unlocked();
    return n;
}
inline void write(ll a){
    register char c; char snum[20]; ll i=0;
    do{
        snum[i++]=a%10+48;
        a=a/10;
    }
    while(a!=0); i--;
    while(i>=0)
        putchar_unlocked(snum[i--]);
    putchar_unlocked('\n');
}
using getline, use cin.ignore()
// gp_hash_table
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table;  //cc_hash_table can ↩
   also be used
//custom hash function
const int RANDOM = chrono::high_resolution_clock::now↩
   ().time_since_epoch().count();
struct chash {
    int operator()(int x) { return hash<int>{}(x ^ ↩
       RANDOM); }
};
gp_hash_table<int, int, chash> table;
//custom hash function for pair
struct chash {
    int operator()(pair<int,int> x) const { return x.↩
       first* 31 + x.second; }
};
// random
mt19937 rng(chrono::steady_clock::now().↩
   time_since_epoch().count());
uniform_int_distribution<int> uid(l,r);
int x=uid(rng);
//mt19937_64 rng(chrono::steady_clock::now().↩
   time_since_epoch().count());
 // - for 64 bit unsigned numbers
vector<int> per(N);
for (int i = 0; i < N; i++)
    per[i] = i;
shuffle(per.begin(), per.end(), rng);
// string splitting
// this splitting is better than custom function(w.r.↩
   t time)
string line = "Ge";
vector <string> tokens;
stringstream check1(line);
string ele;
```

```
// Tokenizing w.r.t. space ' '
while(getline(check1, ele, ' '))
tokens.push_back(ele);
```

1.2 C++ Sublime Build

```
{
  "cmd": ["bash", "-c", "g++ -std=c++11 -O3 '${file}' -o '${file_path}/${file_base_name}' && gnome-terminal -- bash -c '\"${file_path}/${file_base_name}\" < input.txt >output.txt' "],
  "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
  "working_dir": "${file_path}",
  "selector": "source.c++, source.cpp",
}
```

2 Data Structures

2.1 Fenwick

```
/*All indices are 1 indexed*/
/*Range update and point query: maintain BIT of prefix sum of updates
  to add val in range [a,b] add val at a and -val at b
  value[a]=BITsum(a)+arr[a] where arr is constant*/
/***Range update and range query: maintain 2 BITs B1 and B2
**to add val in [a,b] add val at a and -val at b+1 in B1. Add val*(a-1) at a and -val*b at b+1
**sum[1,b]=B1sum(1,b)*b-B2sum(1,b)
**sum[a,b]=sum[1,b]-sum[1,a-1]*/
ll n;
ll fen[MAX_N];
void update(ll p,ll val){
  for(ll i = p;i <= n;i += i & -i)
    fen[i] += val;
}
ll sum(ll p){
  ll ans = 0;
  for(ll i = p;i;i -= i & -i)
    ans += fen[i];
  return ans;
}
```

2.2 2D-BIT

```
//point updates and range sum in a ractangle
//all indices are 1 indices. to increment value of cell (i,j) by val call update(x,y,val)
//to find sum of rectangle [a,b]-[c,d] find sum of rectangles [1,1]-[c,d],[1,1]-[c,b],
//[1,1][a,d] and [1,1]-[a,b] and use inclusion exclusion
ll bit[MAX][MAX];
void update(ll x , ll y, ll val)
{
  while( x < MAX )
  {
    ll y1 = y;
    while( y1 < MAX )
      bit[x][y1] += val , y1 += ( y1 & -y1 );
    x += (x & -x);
  }
}
ll sum(ll x , ll y)
{
  ll ans = 0;
  while( x > 0 )
  {
    ll y1 = y;
    while( y1 > 0 )
      ans += bit[x][y1] , y1 -= ( y1 & -y1 );
    x -= (x & -x);
  }
  return ans;
}
```

2.3 Segment Tree

```
/*
Sum segment tree
All arrays are 0 indexed. How to use:
to build segtree for arr[n] build(0,n-1,1)
to increment all values in [x,y] by val: upd(0,n-1,1,x,y,val)
call ppgt before every recursive call
to get sum of range [x,y]: sum(0,n-1,1,x,y)
for an array of size N use segment tree of size 4*N
*/
#define ll long long
const ll N=1e5+10;
ll arr[N],st[N<<2], lazy[N<<2];
void ppgt(ll l, ll r,ll id)
{
  if(l == r) return;
  ll m = l + r >> 1;
  lazy[id << 1] += lazy[id]; lazy[id << 1 | 1] += lazy[id];
  st[id << 1] += (m - l + 1) * lazy[id];
  st[id << 1 | 1] += (r - m) * lazy[id];
```

```
  lazy[id] = 0;
}
void build(ll l,ll r,ll id)
{
  if(l==r) { st[id] = arr[l]; return; }
  build ( l, l+r >>1 , id<< 1); build( (l + r >> 1) + 1, r , id<< 1 | 1);
  st[id] = st[ id << 1] + st[id << 1 | 1];
}
void upd(ll l,ll r,ll id,ll x,ll y,ll val)
{
  if (l > y || r < x ) return;
  ppgt(l, r, id);
  if (l >= x && r <= y ) { lazy[id] += val; st[id] += (r - l + 1)*val; return;}
  upd(l,l + r >> 1,id << 1, x, y, val);upd((l + r >> 1) + 1,r ,id << 1 | 1,x, y, val);
  st[id] = st[id << 1] + st[ id << 1 | 1];
}
ll sum(ll l,ll r,ll id,ll x,ll y)
{
  if (l > y || r < x ) return 0;
  ppgt(l, r, id);
  if (l >= x && r <= y ) return st[id];
  return sum(l, l + r >> 1,id << 1, x, y) + sum((l + r >> 1 ) + 1,r ,id << 1 | 1,x, y);
}
```

2.4 Persistent Segment Tree

```
/*Persistent Segment Tree for sum with point updates and range sum
Usage: See sample main for kth largest number in a range
**id of first node is 0. call build(0,n-1) first. afterwards call upd(0,n-1,previous id,i,val) to add
val in ith number. it returns root of new segment tree after modification
**sum(0,n-1,id of root,l,r) gives sum of values whose index is between l and r in
tree rooted at id
**size of st,lchild and rchild should be at least N*2+Q*logN
*/
const ll N=1e5+10;
ll arr[N],st[20*N];
ll lchild[20*N],rchild[20*N];
ll ids[N];
ll cnt=0;
void build(ll l,ll r)
{
  if(l==r) { lchild[cnt] = rchild[cnt] = -1; st[cnt] = arr[l]; ++cnt; return; }
  ll id = cnt++;
  lchild[id] = cnt;
  build ( l, l+r >>1);
  rchild[id] = cnt; build( (l + r >> 1) + 1, r);
  st[id] = st[lchild[id]] + st[rchild[id]];
}
ll upd(ll l,ll r,ll id,ll x,ll val)
{
  if(l == r) {lchild[cnt] = rchild[cnt] = -1; st[cnt] = st[id] + val; ++cnt; return cnt-1;}
  ll myid = cnt++; ll mid = l + r >>1;
  if(x <= mid)
    rchild[myid] = rchild[id],lchild[myid] = upd(l, mid, lchild[id], x, val);
  else
    lchild[myid] = lchild[id],rchild[myid] = upd(mid+1, r, rchild[id], x, val);
  st[myid] = st[lchild[myid]] + st[rchild[myid]]; return myid;}
ll sum(ll l,ll r,ll id,ll x,ll y)
{
  if (l > y || r < x ) return 0;
  if (l >= x && r <= y ) return st[id];
  return sum(l, l + r >> 1,lchild[id], x, y) + sum((l + r >> 1 ) + 1,r ,rchild[id],x, y);
}
ll gkth(ll l,ll r,ll id1,ll id2,ll k)
{
  if(l==r) return l;
  ll mid = l+r>>1;
  ll a = st[lchild[id2]] - (id1 >= 0 ? st[lchild[id1]] : 0);
  if(a >= k)
    return gkth(l, mid ,(id1>=0?lchild[id1]:-1), lchild[id2], k);
  else
    return gkth(mid+1, r,(id1>=0?rchild[id1]:-1), rchild[id2], k-a);
}
int main()
{
  ll n,m;cin>>n>>m;vector<ll> finalid(n);vpll v;
  for(ll i=0;i<n;i++)cin>>arr[i],v.pb({arr[i],i});sort(all(v));
  for(ll i=0;i<n;i++)finalid[v[i].second]=i;memset(arr,0,sizeof(ll)*N);
  arr[finalid[0]]++;build(0,n-1);
  for(ll i=1;i<n;i++) ids[i]=upd(0,n-1,ids[i-1],finalid[i],1);
  while(m--){
    ll i,j,k;cin>>i>>j>>k;
    --i;--j;
    ll ans=gkth(0,n-1,(i==0?-1:ids[i-1]),ids[j],k);
    cout<<v[ans].F<<endl;}
}
```

2.5 DP Optimization

```
/*You have an array of size L.You need to split it into G intervals,
minimizing the cost. (G<=L otherwise we can just split in 1-intervals).
There is a cost function C[i,j] of taking an interval.The cost function
satisfies : C[a,b]+C[c,d]<=C[a,d]+ C[c,b] for all a<=c<=b<=d.
This is the quadrangle inequality and intuitively you can think that
the cost function increases at a rate which is more than linear
at all intervals (may not be strictly true). So , if the cost function
satisfies this inequality, the following property holds :
F(g,l) : min cost of spliting first l elements into g intervals
Basic recurrence : F(g,l) = min(F(g-1,k)+C(k+1,l)) over all valid k.
P(g,l) : lowest position k s.t. it minimizes F(g,l).
P(g,0)<=P(g,1)<=P(g,2).....<=P(g,l-1)<=P(g,l). (DivConqOpti,O(G.L.log(L)))
Also, P(0,l)<=P(1,l)<=P(2,l)....<=P(G-1,l)<=P(G,l).
This with previous inequality leads to Knuth Opti, complexity O(L.L).
For div&conq, we calculate P(g,l) for each g 1 by 1. In each g,
we calculate for mid-l and solve recursively using the obtained
upper and lower bounds.For knuth, we use P(g,l-1)<=P(g,l)<=P(g+1,l),
and fill our table in increasing l and decreasing g.
In opt. BST type problems, use  bk[i][j-1]<= bk[i][j]<=bk[i+1][j] . */
// Code for Divide and Conquer Opti O(G.L.log(L)): -
ll C[8111];
ll sums[8111];
ll F[811][8111];   // optimal value
int P[811][8111];  // optimal position.
// note first val. in arrays is for no. of groups
ll cost(int i, int j) { // cost function
    return i > j ? 0 : (sums[j] - sums[i-1]) * (j - i + 1);
}
// fill(g,l1,l2,p1,p2) calculates all P[g][l] and F[g][l]
// for l1 <= l <= l2,with the knowledge that p1 <= P[g][l] <= p2
void fill(int g, int l1, int l2, int p1, int p2) {
    if (l1 > l2) return;
    int lm = (l1 + l2) >> 1;
    ll nv=INF,nv1=-1;
    for (int k = p1; k <= min(lm-1,p2); k++) {
        ll new_cost = F[g-1][k] + cost[k+1][lm];
        if (nv > new_cost) {
            nv = new_cost;
            nv1 = k;
        }
    }
    P[g][lm]=nv1; F[g][lm]=nv;
    fill(g, l1, lm-1, p1, P[g][lm]);
    fill(g, lm+1, l2, P[g][lm], p2);
}
int main() { // example call
    for(i=0;i<=n;i++)F[0][i]=INF;
    for(i=0;i<=k;i++)F[i][0]=0;
    F[0][0]=0;
    for(i=1;i<=k;i++)fill(i,1,n,0,n);
}
// Code for Knuth Optimization O(L.L) :-
ll dp[8002][802];
int a[8002],s[8002][802];
ll sum[8002];
// index strats from 1
ll run(int n,int m) {
    memset(dp,0xff,sizeof(dp));
    dp[0][0] = 0;
    for (int i = 1; i <= n; ++i) {
        sum[i] = sum[i - 1] + a[i];
        int maxj = min(i, m), mk;
        ll mn = INF;
        for (int k = 0; k < i; ++k) {
            if (dp[k][maxj - 1] >= 0) {
                ll tmp = dp[k][maxj - 1] + (sum[i] - sum[k]) * (i - k); //k + 1..i
                if (tmp < mn) {
                    mn = tmp;
                    mk = k;
                }
            }
        }
        dp[i][maxj] = mn;
    s[i][maxj] = mk;
        for (int j = maxj - 1; j >= 1; --j) {
            ll mn = INF;
            int mk;
            for (int k = s[i - 1][j]; k <= s[i][j + 1]; ++k) {
                if (dp[k][j - 1] >= 0) {
                    ll tmp = dp[k][j - 1] + (sum[i] - sum[k]) * (i - k);
                    if (tmp < mn) {
                        mn = tmp;
                        mk = k;
                    }
                }
            }
            dp[i][j] = mn;
            s[i][j] = mk;
        }
    }
```

```
    return dp[n][m];
}
// call -> run(n, min(n,m))
```

3 Flows and Matching

3.1 General Matching

```
/*Given any directed graph, finds maximal matching
  Vertices-0-indexed, O(n^3) per call to edmonds*/
vll adj[MAXN]; int p[MAXN], base[MAXN], match[MAXN];
int lca(int n, int u, int v){
  vector<bool> used(n);
  for (;;) {
    u = base[u]; used[u] = true;
    if (match[u] == -1) break; u = p[match[u]];}
  for (;;) {
    v = base[v]; if (used[v]) return v;
    v = p[match[v]];}}
void mark_path(vector<bool> &blo, int u, int b, int ↩
   child){
  for (; base[u] != b; u = p[match[u]]){
    blo[base[u]] = true; blo[base[match[u]]] = true;
    p[u] = child; child = match[u];}}
int find_path(int n, int root) {
  vector<bool> used(n);
  for (int i = 0; i < n; ++i)
    p[i] = -1, base[i] = i;
  used[root] = true;
  queue<int> q; q.push(root);
  while(!q.empty()) {
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = adj[u][j];
      if(base[u]==base[v] || match[u]==v)continue;
      if(v==root || (match[v]!= -1 && p[match[v]]!= ↩
         -1)){
        int curr_base = lca(n, u, v);
        vector<bool> blossom(n);
        mark_path(blossom, u, curr_base, v);
        mark_path(blossom, v, curr_base, u);
        for(int i = 0; i < n; i++){
          if(blossom[base[i]]){
            base[i] = curr_base;
            if(!used[i]) used[i] = true, q.push(i);
          }
        }
      }else if (p[v] == -1){
        p[v] = u;
        if (match[v] == -1) return v;
        v=match[v]; used[v]=true; q.push(v);
      }
    }
  }
  return -1;}
int edmonds(int n){
  for(int i=0;i<n;i++) match[i] = -1;
  for(int i = 0; i < n; i++){
    if (match[i] == -1) {
      int u, pu, ppu;
      for (u = find_path(n, i); u != -1; u = ppu) {
        pu = p[u]; ppu = match[pu];
        match[u] = pu; match[pu] = u;
      }
    }
  }
  int matches = 0;
  for (int i = 0; i < n; i++)
    if (match[i] != -1) matches++;
  return matches/2;
}
u--; v--; adj[u].pb(v); adj[v].pb(u);
cout << edmonds(n) * 2 << endl;
for (int i = 0; i < n; i++) {
    if (match[i] != -1 && i < match[i]) {
        cout << i + 1 << " " << match[i] + 1 << endl;
    }
}
```

3.2 Global Mincut

```
/*finds min weighted cut in undirected graph in
O(n^3), Adj Matrix, 0-indexed vertices
output-(min cut value, nodes in half of min cut)*/
typedef vector<int> VI;
typedef vector<VI> VVI;
const int INF = 1000000000;
pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;
  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
  if (!added[j] && (last == -1 || w[j] > w[last])) ↩
     last = j;
      if (i == phase-1) {
  for (int j = 0; j < N; j++) weights[prev][j] += ↩
     weights[last][j];
  for (int j = 0; j < N; j++) weights[j][prev] = ↩
     weights[prev][j];
  used[last] = true;
```

```
  cut.push_back(last);
  if (best_weight == -1 || w[last] < best_weight) {
    best_cut = cut;
    best_weight = w[last];
  }
        } else {
  for (int j = 0; j < N; j++)
    w[j] += weights[last][j];
  added[last] = true;
        }
      }
  }
  return make_pair(best_weight, best_cut);
}
VVI weights(n, VI(n));
pair<int, VI> res = GetMinCut(weights);
```

3.3 Hopcroft Matching

```
// O(m * \sqrt{n})
struct graph {
  int L, R; // 0-indexed vertices
  vector<vector<int>> adj;
  graph(int L, int R) : L(L), R(R), adj(L+R) { }
  void add_edge(int u, int v) {
    adj[u].pb(v+L);
    adj[v+L].pb(u);
  }
  int maximum_matching() {
    vector<int> level(L), mate(L+R, -1);
    function<bool(void)> levelize = [&]() { // BFS
      queue<int> Q;
      for (int u = 0; u < L; ++u) {
        level[u] = -1;
        if (mate[u] < 0)
          level[u] = 0, Q.push(u);
      }
      while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int w: adj[u]) {
          int v = mate[w];
          if (v < 0) return true;
          if (level[v] < 0) {
            level[v] = level[u] + 1; Q.push(v);
          }
        }
      }
      return false;
    };
    function<bool(int)> augment = [&](int u) { // DFS
      for (int w: adj[u]) {
        int v = mate[w];
        if (v < 0 || (level[v] > level[u] && augment(↩
           v))) {
          mate[u] = w;
          mate[w] = u;
          return true;
        }
      }
      return false;
    };
    int match = 0;
    while (levelize())
      for (int u = 0; u < L; ++u)
        if (mate[u] < 0 && augment(u))
          ++match;
    return match;
  }
};
int main() {
  int L, R, m;
  scanf("%d %d %d", &L, &R, &m);
  graph g(L, R);
  for (int i = 0; i < m; ++i) {
    int u, v;
    scanf("%d %d", &u, &v); u--;v--;
    g.add_edge(u, v);
  }
  printf("%d\n", g.maximum_matching());
}
```

3.4 Dinic

```
/*Time: O(m*n^2) and for any unit capacity network O(↩
   m * n^1/2)
TIme: O(min(fm,mn^2)) (f: flow routed)
(so for bipartite matching as well)
In practice it is pretty fast for any bipartite ↩
   network
I/O:     n -> vertice; DinicFlow net(n);
         for(z : edges) net.addEdge(z.F,z.S,cap);
         max flow = maxFlow(s,t);
e=(u,v), e.flow represents the effective flow from u ↩
   to v
(i.e f(u->v) - f(v->u)), vertices are 1-indexed
*** use int if possible(ll could be slow in dinic) ↩
   *** */
struct edge {ll x, y, cap, flow;};
struct DinicFlow {
    // *** change inf accordingly *****
    const ll inf = (1e18);
    vector <edge> e;
    vector <ll> cur, d;
    vector < vector <ll> > adj;
    ll n, source, sink;
    DinicFlow() {}
    DinicFlow(ll v) {
        n = v;
        cur = vector <ll> (n + 1);
        d = vector <ll> (n + 1);
        adj = vector < vector <ll> > (n + 1);
```

```
    }
    void addEdge(ll from, ll to, ll cap) {
        edge e1 = {from, to, cap, 0};
        edge e2 = {to, from, 0, 0};
        adj[from].push_back(e.size()); e.push_back(e1);
        adj[to].push_back(e.size()); e.push_back(e2);
    }
    ll bfs() {
        queue <ll> q;
        for(ll i = 0; i <= n; ++i) d[i] = -1;
        q.push(source); d[source] = 0;
        while(!q.empty() and d[sink] < 0) {
            ll x = q.front(); q.pop();
            for(ll i = 0; i < (ll)adj[x].size(); ++i)
            {
                ll id = adj[x][i], y = e[id].y;
                if(d[y] < 0 and e[id].flow < e[id].cap) {
                    q.push(y); d[y] = d[x] + 1;
                }
            }
        }
        return d[sink] >= 0;
    }
    ll dfs(ll x, ll flow) {
        if(!flow) return 0;
        if(x == sink) return flow;
        for(;cur[x] < (ll)adj[x].size(); ++cur[x]) {
            ll id = adj[x][cur[x]], y = e[id].y;
            if(d[y] != d[x] + 1) continue;
            ll pushed = dfs(y, min(flow, e[id].cap - e[id].flow));
            if(pushed) {
                e[id].flow += pushed;
                e[id ^ 1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
    ll maxFlow(ll src, ll snk) {
        this->source = src; this->sink = snk;
        ll flow = 0;
        while(bfs()) {
            for(ll i = 0; i <= n; ++i) cur[i] = 0;
            while(ll pushed = dfs(source, inf)) {
                flow += pushed;
            }
        }
        return flow;
    }
};
```

3.5 Edmond Karp

```
// running time - O(n*m^2)
// The matrix capacity stores the capacity for every pair of vertices. adj
// is the adjacency list of the undirected graph, since we have also to use
// the reversed of directed edges when we are looking for augmenting paths.
// The function maxflow will return the value of the maximal flow. During
// the algorithm the matrix capacity will actually store the residual capacity
// of the network. The value of the flow in each edge will actually no stored,
// but it is easy to extent the implementation - by using an additional matrix
// - to also store the flow and return it.
const ll N = 3e3;
ll n; // number of vertices
ll capacity[N][N]; // adj matrix for capacity
vll adj[N]; // adj list of the corresponding undirected graph(***imp***)
// E = {1-2,2->3,3->2}, adj list should be => {1->2,2->1,2->3,3->2}
// *** vertices are 0-indexed ***
ll INF = (1e18);
ll bfs(ll s, ll t, vector<ll>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<ll, ll>> q;
    q.push({s, INF});
    while (!q.empty()) {
        ll cur = q.front().first;
        ll flow = q.front().second;
        q.pop();
        for (ll next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                ll new_flow = min(flow, capacity[cur][next]);
                if (next == t)
                    return new_flow;
                q.push({next, new_flow});
            }
        }
    }
    return 0;
}
ll maxflow(ll s, ll t) {
    ll flow = 0; ll new_flow;
    vector<ll> parent(n);
    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        ll cur = t;
```

```
        while (cur != s) {
            ll prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}
```

3.6 Ford Fulkerson

```
// running time - O(f*m) (f -> flow routed)
const ll N = 3e3;
ll n; // number of vertices
ll capacity[N][N]; // adj matrix for capacity
vll adj[N]; // adj list of the corresponding ↩
    undirected graph(***imp***)
// E = {1-2,2->3,3->2}, adj list should be => ↩
    {1->2,2->1,2->3,3->2}
// *** vertices are 0-indexed ***
ll INF = (1e18);
ll snk, cnt; // cnt for vis, no need to initialize ↩
    vis
vector<ll> par, vis;
ll dfs(ll u,ll curr_flow){
  vis[u] = cnt; if(u == snk) return curr_flow;
  if(adj[u].size() == 0) return 0;
  for(ll j=0;j<5;j++){ // random for good ↩
      augmentation(**sometimes take time**)
        ll a = rand()%(adj[u].size());
        ll v = adj[u][a];
        if(vis[v] == cnt || capacity[u][v] == 0) ↩
            continue;
        par[v] = u;
        ll f = dfs(v,min(curr_flow, capacity[u][v]));↩
            if(vis[snk] == cnt) return f;
    }
    for(auto v : adj[u]){
      if(vis[v] == cnt || capacity[u][v] == 0) ↩
          continue;
      par[v] = u;
      ll f = dfs(v,min(curr_flow, capacity[u][v])); ↩
          if(vis[snk] == cnt) return f;
    }
    return 0;
}
ll maxflow(ll s, ll t) {
    snk = t; ll flow = 0; cnt++;
    par = vll(n,-1); vis = vll(n,0);
    while(ll new_flow = dfs(s,INF)){
      flow += new_flow; cnt++;
      ll cur = t;
      while(cur != s){
        ll prev = par[cur];
        capacity[prev][cur] -= new_flow;
        capacity[cur][prev] += new_flow;
        cur = prev;
      }
    }
    return flow;
}
```

3.7 Push Relabel

```
// Adjacency list implementation of FIFO push relabel↩
    maximum flow
// with the gap relabeling heuristic.  This ↩
    implementation is
// significantly faster than straight Ford-Fulkerson.↩
     It solves
// random problems with 10000 vertices and 1000000 ↩
    edges in a few
// seconds, though it is possible to construct test ↩
    cases that
// achieve the worst-case.
// Time: O(V^3)
// I/O:- addEdge(),src,snk ** vertices are 0-indexed ↩
    **
//      - To obtain the actual flow values, look at ↩
    all edges with
//         capacity > 0 (zero capacity edges are ↩
    residual edges).
struct edge {
  ll from, to, cap, flow, index;
  edge(ll from, ll to, ll cap, ll flow, ll index) :
    from(from), to(to), cap(cap), flow(flow), index(↩
        index) {}
};
struct PushRelabel {
  ll n;
  vector<vector<edge> > G;
  vector<ll> excess;
  vector<ll> dist, active, count;
  queue<ll> Q;
  PushRelabel(ll n) : n(n), G(n), excess(n), dist(n),↩
      active(n), count(2*n) {}
  void addEdge(ll from, ll to, ll cap) {
    G[from].push_back(edge(from, to, cap, 0, G[to].↩
        size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(edge(to, from, 0, 0, (ll)G[from].↩
        size() - 1));
  }
  void enqueue(ll v) {
    if (!active[v] && excess[v] > 0) { active[v] = ↩
        true; Q.push(v); }
  }
  void push(edge &e) {
    ll amt = min(excess[e.from], e.cap - e.flow);
```

```
    if (dist[e.from] <= dist[e.to] || amt == 0) ↩
       return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    enqueue(e.to);
  }
  void gap(ll k) {
    for (ll v = 0; v < n; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--; dist[v] = max(dist[v], n+1);
      count[dist[v]]++; enqueue(v);
    }
  }
  void relabel(ll v) {
    count[dist[v]]--;
    dist[v] = 2*n;
    for (ll i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
    dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    enqueue(v);
  }
  void discharge(ll v) {
    for (ll i = 0; excess[v] > 0 && i < G[v].size(); ↩
       i++) push(G[v][i]);
    if (excess[v] > 0) {
      if (count[dist[v]] == 1)  gap(dist[v]);
      else relabel(v);
    }
  }
  ll getMaxFlow(ll s, ll t) {
    count[0] = n-1;
    count[n] = 1;
    dist[s] = n;
    active[s] = active[t] = true;
    for (ll i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      push(G[s][i]);
    }
    while (!Q.empty()) {
      ll v = Q.front(); Q.pop();
      active[v] = false; discharge(v);
    }
    ll totflow = 0;
    for (ll i = 0; i < G[s].size(); i++) totflow += G↩
       [s][i].flow;
    return totflow;
  }
};
```

3.8 MCMF

```
// MCMF Theory:
// 1.  If a network with negative costs had no ↩
   negative cycle it is possible to transform it into↩
    one with nonnegative
//      costs. Using Cij_new(pi) = Cij_old + pi(i) - ↩
   pi(j), where pi(x) is shortest path from s to x in↩
    network with an
//      added vertex s. The objective value remains ↩
   the same (z_new = z + constant). z(x) = sum(cij*↩
   xij)
//      (x->flow, c->cost, u->cap, r->residual cap).
// 2.  Residual Network: cji = -cij, rij = uij-xij, ↩
   rji = xij.
// 3.  Note: If edge (i,j),(j,i) both are there then ↩
   residual graph will have four edges b/w i,j (pairs↩
    of parellel edges).
// 4.  let x* be a feasible soln, its optimal iff ↩
   residual network Gx* contains no negative cost ↩
   cycle.
// 5.  Cycle Cancelling algo => Complexity O(n*m^2*U*↩
   C) (C->max abs value of cost, U->max cap) (m*U*C ↩
   iterations).
// 6.  Succesive shortest path algo => Complexity O(n↩
   ^3 * B) / O(nmBlogn)(using heap in Dijkstra)(B -> ↩
   largest supply node).
//Works for negative costs, but does not work for ↩
   negative cycles
//Complexity: O(min(E^2 *V log V, E logV * flow))
// to use -> graph G(n), G.add_edge(u,v,cap,cost), G.↩
   min_cost_max_flow(s,t)
// ********* INF is used in both flow_type and ↩
   cost_type so change accordingly
const ll INF = 99999999;
// vertices are 0-indexed
struct graph {
  typedef ll flow_type; // **** flow type ****
  typedef ll cost_type; // **** cost type ****
  struct edge {
    int src, dst;
    flow_type capacity, flow;
    cost_type cost;
    size_t rev;
  };
  vector<edge> edges;
  void add_edge(int src, int dst, flow_type cap, ↩
     cost_type cost) {
    adj[src].push_back({src, dst, cap, 0, cost, adj[↩
       dst].size()});
    adj[dst].push_back({dst, src, 0, 0, -cost, adj[↩
       src].size()-1});
  }
  int n;
  vector<vector<edge>> adj;
  graph(int n) : n(n), adj(n) { }
  pair<flow_type, cost_type> min_cost_max_flow(int s,↩
      int t) {
    flow_type flow = 0;
```

```
    cost_type cost = 0;
    for (int u = 0; u < n; ++u) // initialize
      for (auto &e: adj[u]) e.flow = 0;
    vector<cost_type> p(n, 0);
    auto rcost = [&](edge e) { return e.cost + p[e.src] - p[e.dst]; };
    for (int iter = 0; ; ++iter) {
      vector<int> prev(n, -1); prev[s] = 0;
      vector<cost_type> dist(n, INF); dist[s] = 0;
      if (iter == 0) { // use Bellman-Ford to remove negative cost edges
        vector<int> count(n); count[s] = 1;
        queue<int> que;
        for (que.push(s); !que.empty(); ) {
          int u = que.front(); que.pop();
          count[u] = -count[u];
          for (auto &e: adj[u]) {
            if (e.capacity > e.flow && dist[e.dst] > dist[e.src] + rcost(e)) {
              dist[e.dst] = dist[e.src] + rcost(e);
              prev[e.dst] = e.rev;
              if (count[e.dst] <= 0) {
                count[e.dst] = -count[e.dst] + 1;
                que.push(e.dst);
              }
            }
          }
        }
        for(int i=0;i<n;i++) p[i] = dist[i]; // added it
        continue;
      } else { // use Dijkstra
        typedef pair<cost_type, int> node;
        priority_queue<node, vector<node>, greater<node>> que;
        que.push({0, s});
        while (!que.empty()) {
          node a = que.top(); que.pop();
          if (a.S == t) break;
          if (dist[a.S] > a.F) continue;
          for (auto e: adj[a.S]) {
            if (e.capacity > e.flow && dist[e.dst] > a.F + rcost(e)) {
              dist[e.dst] = dist[e.src] + rcost(e);
              prev[e.dst] = e.rev;
              que.push({dist[e.dst], e.dst});
            }
          }
        }
      }
      if (prev[t] == -1) break;
      for (int u = 0; u < n; ++u)
        if (dist[u] < dist[t]) p[u] += dist[u] - dist[t];
      function<flow_type(int,flow_type)> augment = [&](int u, flow_type cur) {
        if (u == s) return cur;
        edge &r = adj[u][prev[u]], &e = adj[r.dst][r.rev];
        flow_type f = augment(e.src, min(e.capacity - e.flow, cur));
        e.flow += f; r.flow -= f;
        return f;
      };
      flow_type f = augment(t, INF);
      flow += f;
      cost += f * (p[t] - p[s]);
    }
    return {flow, cost};
  }
};
```

3.9 MinCost 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.
typedef ll cost_type;
typedef vector<cost_type> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
cost_type MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());
  // construct dual feasible solution
  VD u(n);
  VD 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]);
  }
  // construct primal solution satisfying ↩
     complementary slackness
  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 (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
      //**** change this comparision if double cost ↩
         ****
        Lmate[i] = j;
        Rmate[j] = i;
        mated++;
        break;
      }
    }
  }
  VD dist(n); VI dad(n); VI seen(n);
  // repeat until primal solution is feasible
  while (mated < n) {
    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;
    // initialize Dijkstra
    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) {
      // find closest
      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;
      // termination condition
      if (Rmate[j] == -1) break;
      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        const cost_type new_dist = dist[j] + cost[i][↩
           k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }
    // update dual variables
    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];
    // augment along path
    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++;
  }
  cost_type value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];
  return value;
}
```

4 Geometry

4.1 Geometry

```
//small non recursive functions should me made inline
//do not read input in double format if they are ↩
   integer points
#define ld double
#define PI acos(-1)
//atan2(y,x) slope of line (0,0)->(x,y) in radian (-↩
   PI,PI]
// to convert to degree multiply by 180/PI
ld INF = 1e100;
ld EPS = 1e-9;
inline bool eq(ld a,ld b) {return fabs(a-b)<EPS;}
inline bool lt(ld a,ld b) {return a+EPS<b;}
inline bool gt(ld a,ld b) {return a>b+EPS;}
inline bool le(ld a,ld b) {return lt(a,b)||eq(a,b);}
inline bool ge(ld a,ld b) {return gt(a,b)||eq(a,b);}
struct pt {
  ld x, y;
  pt() {}
  pt(ld x, ld y) : x(x), y(y) {}
  pt(const pt &p) : x(p.x), y(p.y)    {}
  pt operator + (const pt &p)  const { return pt(x+p.↩
     x, y+p.y); }
  pt operator - (const pt &p)  const { return pt(x-p.↩
     x, y-p.y); }
```

```cpp
  pt operator * (ld c)      const { return pt(x*c,    y*c  ); }
  pt operator / (ld c)      const { return pt(x/c,    y/c  ); }
  bool operator < (const pt &p) const{ return lt(y,p.y)||(eq(y,p.y)&&lt(x,p.x));}
  bool operator > (const pt &p) const{ return p<pt(x,y);}
  bool operator <= (const pt &p) const{ return !(pt(x,y)>p);}
  bool operator >= (const pt &p) const{ return !(pt(x,y)<p);}
  bool operator == (const pt &p) const{ return (pt(x,y)<=p)&&(pt(x,y)>=p);}
};
ld dot(pt p,pt q) {return p.x*q.x+p.y*q.y;}
ld dist2(pt p, pt q) {return dot(p-q,p-q);}
ld dist(pt p,pt q) {return sqrt(dist2(p,q));}
ld norm2(pt p) {return dot(p,p);}
ld norm(pt p) {return sqrt(norm2(p));}
ld cross(pt p, pt q) { return p.x*q.y-p.y*q.x;}
ostream &operator<<(ostream &os, const pt &p) {
  return os << "(" << p.x << "," << p.y << ")";}
istream& operator >> (istream &is, pt &p){
  return is >> p.x >> p.y;}
//returns 0 if a,b,c are collinear,1 if a->b->c is cw and -1 if ccw
int orient(pt a,pt b,pt c)
{
  pt p=b-a,q=c-b;double cr=cross(p,q);
  if(eq(cr,0))return 0;if(lt(cr,0))return 1;return -1;}
// rotate a point CCW or CW around the origin
pt RotateCCW90(pt p)   { return pt(-p.y,p.x); }
pt RotateCW90(pt p)    { return pt(p.y,-p.x); }
pt RotateCCW(pt p, ld t) { //rotate by angle t degree ccw
  return pt(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t)); }
// project point c onto line (not segment) through a and b assuming a != b
pt ProjectPointLine(pt a, pt b, pt c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);}
// project point c onto line segment through a and b (closest point on line segment)
pt ProjectPointSegment(pt a, pt b, pt c) {
  ld r = dot(b-a,b-a); if (eq(r,0)) return a;//a and b are same
  r = dot(c-a, b-a)/r;if (lt(r,0)) return a;//c on left of a
  if (gt(r,1)) return b; return a + (b-a)*r;}
// compute distance from c to segment between a and b
ld DistancePointSegment(pt a, pt b, pt c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));}
// compute distance from c to line between a and b
ld DistancePointLine(pt a, pt b, pt c) {
  return sqrt(dist2(c, ProjectPointLine(a, b, c)));}
// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(pt a, pt b, pt c, pt d) {
  return eq(cross(b-a, c-d),0); }
bool LinesCollinear(pt a, pt b, pt c, pt d) {
  return LinesParallel(a, b, c, d) && eq(cross(a-b, a-c),0) && eq(cross(c-d, c-a),0);}
// determine if line segment from a to b intersects with line segment from c to d
bool SegmentsIntersect(pt a, pt b, pt c, pt d) {
  if (LinesCollinear(a, b, c, d)) {
    //a->b and c->d are collinear and have one point common
    if(eq(dist2(a,c),0)||eq(dist2(a,d),0)||eq(dist2(b,c),0)||eq(dist2(b,d),0)) return true;
    if(gt(dot(c-a,c-b),0)&&gt(dot(d-a,d-b),0)&&gt(dot(c-b,d-b),0)) return false;
    return true;}
  if(gt(cross(d-a,b-a)*cross(c-a,b-a),0)) return false;//c,d on same side of a,b
  if(gt(cross(a-c,d-c)*cross(b-c,d-c),0)) return false;//a,b on same side of c,d
  return true;}
// compute intersection of line passing through a and b
// with line passing through c and d,assuming that **unique** intersection exists;
//*for segment intersection,check if segments intersect first
pt ComputeLineIntersection(pt a,pt b,pt c,pt d){
  b=b-a;d=c-d;c=c-a;//lines must not be collinear
  assert(gt(dot(b, b),0)&&gt(dot(d, d),0));
  return a + b*cross(c, d)/cross(b, d);}
//returns true if point a,b,c are collinear and b lies between a and c
bool between(pt a,pt b,pt c){
  if(!eq(cross(b-a,c-b),0))return 0;//not collinear
  return le(dot(b-a,b-c),0);
}
//compute intersection of line segment a-b and c-d
pt ComputeSegmentIntersection(pt a,pt b,pt c,pt d){
  if(!SegmentsIntersect(a,b,c,d))return {INF,INF};//don't intersect
  //if collinear then infinite intersection points, this returns any one
  if(LinesCollinear(a,b,c,d)){if(between(a,c,b))return c;if(between(c,a,d))return a;return b;}
  return ComputeLineIntersection(a,b,c,d);
}
// compute center of circle given three points - *a,b,c shouldn't be collinear
pt ComputeCircleCenter(pt a,pt b,pt c){
  b=(a+b)/2;c=(a+c)/2;
```

```
  return ComputeLineIntersection(b,b+RotateCW90(a-b),↩
     c,c+RotateCW90(a-c));}
//point in polygon using winding number -> returns 0 ↩
   if point is outside
//winding number>0 if point is inside and equal to 0 ↩
   if outside
//draw a ray to the right and add 1 if side goes from↩
    up to down and -1 otherwise
bool PointInPolygon(const vector<pt> &p,pt q){
  int n=p.size(),windingNumber=0;
  for(int i=0;i<n;++i){
    if(eq(dist2(q,p[i]),0)) return 1;//q is a vertex
    int j=(i+1)%n;
    if(eq(p[i].y,q.y)&&eq(p[j].y,q.y)) {//i,i+1 ↩
       vertex is vertical
      if(le(min(p[i].x,p[j].x),q.x)&&le(q.x,max(p[i].↩
         x, p[j].x))) return 1;}//q lies on boundary
    else {
      bool below=lt(p[i].y,q.y);
      if(below!=lt(p[j].y,q.y)) {
        auto orientation=orient(q,p[j],p[i]);
        if(orientation==0) return 1;//q lies on ↩
           boundary i->j
        if(below==(orientation>0)) windingNumber+=↩
           below?1:-1;}}}
  return windingNumber==0?0:1;
}
// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<pt> &p,pt q) {
  for (int i = 0; i < p.size(); i++)
    if (eq(dist2(ProjectPointSegment(p[i],p[(i+1)%p.↩
       size()],q),q),0)) return true;
  return false;}
// Compute area or centroid of any polygon (↩
   coordinates must be listed in cw/ccw
//fashion.The centroid is often known as center of ↩
   gravity/mass
ld ComputeSignedArea(const vector<pt> &p) {
  ld ans=0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    ans+=cross(p[i],p[j]);
  } return ans / 2.0;}
ld ComputeArea(const vector<pt> &p) {
  return fabs(ComputeSignedArea(p));
}
// compute intersection of line through points a and ↩
   b with
// circle centered at c with radius r > 0
vector<pt> CircleLineIntersection(pt a, pt b, pt c, ↩
   ld r) {
  vector<pt> ret;
  b = b-a;a = a-c;
  ld A = dot(b, b),B = dot(a, b),C = dot(a, a) - r*r,↩
     D = B*B - A*C;
  if (lt(D,0)) return ret;  //don't intersect
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (gt(D,0)) ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;}
// compute intersection of circle centered at a with ↩
   radius r
// with circle centered at b with radius R
vector<pt> CircleCircleIntersection(pt a, pt b, ld r,↩
   ld R) {
  vector<pt> ret;
  ld d = sqrt(dist2(a, b)),d1=dist2(a,b);
  pt inf(INF,INF);
  if(eq(d1,0)&&eq(r,R)){ret.pb(inf);return ret;}//↩
     circles are same return (INF,INF)
  if(gt(d,r+R) || lt(d+min(r, R),max(r, R)) ) return ↩
     ret;
  ld x = (d*d-R*R+r*r)/(2*d),y = sqrt(r*r-x*x);
  pt v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (gt(y,0)) ret.push_back(a+v*x - RotateCCW90(v)*y↩
     );
  return ret;}
//compute centroid of simple polygon by dividing it ↩
   into disjoint triangles
//and taking weighted mean of their centroids (Jerome↩
   )
pt ComputeCentroid(const vector<pt> &p) {
  pt c(0,0),inf(INF,INF);
  ld scale = 6.0 * ComputeSignedArea(p);
  if(p.empty())return inf;//empty vector
  if(eq(scale,0))return inf;//all points on straight ↩
     line
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*cross(p[i],p[j]);}
  return c / scale;}
// tests whether or not a given polygon (in CW or CCW↩
    order) is simple
bool IsSimple(const vector<pt> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;}}
  return true;}
/*point in convex polygon
****bottom left point must be at index 0 and top is ↩
   the index of upper right vertex
****if not call make_hull once*/
bool pointinConvexPolygon(vector<pt> poly,pt point, ↩
   int top) {
  if (point < poly[0] || point > poly[top]) return 0;↩
     //0 for outside and 1 for on/inside
```

```
  auto orientation = orient(point, poly[top], poly↩
     [0]);
  if (orientation == 0) {
    if (point == poly[0] || point == poly[top]) ↩
       return 1;
    return top == 1 || top + 1 == poly.size() ? 1 : ↩
       1;//checks if point lies on boundary when
    //bottom and top points are adjacent
  } else if (orientation < 0) {
    auto itRight = lower_bound(poly.begin() + 1, poly↩
       .begin() + top, point);
    return orient(itRight[0], point, itRight[-1])<=0;
    } else {
    auto itLeft = upper_bound(poly.rbegin(), poly.↩
       rend() - top-1, point);
    return (orient(itLeft == poly.rbegin() ? poly[0] ↩
       : itLeft[-1], point, itLeft[0]))<=0;
    }
}
/*maximum distance between two points in convexy ↩
   polygon using rotating calipers
make sure that polygon is convex. if not call ↩
   make_hull first
*/
ld maxDist2(vector<pt> poly) {
  int n = poly.size();
  ld res=0;
  for (int i = 0, j = n < 2 ? 0 : 1; i < j; ++i)
    for (;; j = j+1 %n) {
        res = max(res, dist2(poly[i], poly[j]));
      if (gt(cross(poly[j+1 % n] - poly[j],poly[i+1] ↩
         - poly[i]),0)) break;
  }
  return res;
}
//Line polygon intersection: check if given line ↩
   intersects any side of polygon
//if yes then line intersects. If no, then check if ↩
   its midpoint is inside polygon
//if midpoint is inside then line is inside else ↩
   outside
// compute distance between point (x,y,z) and plane ↩
   ax+by+cz=d
ld DistancePointPlane(ld x,ld y,ld z,ld a,ld b,ld c,↩
   ld d)
{ return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);}
```

4.2 Convex Hull

```
pt firstpoint;
//for sorting points in ccw(counter clockwise) ↩
   direction w.r.t firstpoint (leftmost and ↩
   bottommost)
bool compare(pt x,pt y){
  ll o=orient(firstpoint,x,y);
  if(o==0)return lt(x.x+x.y,y.x+y.y);
  return o<0;
}
/*takes as input a vector of points containing input ↩
   points and an empty vector for making hull
the points forming convex hull are pushed in vector ↩
   hull
returns hull containing minimum number of points in ↩
   ccw order
****remove EPS for making integer hull
*/
void make_hull(vector<pt>& poi,vector<pt>& hull)
{
  pair<ld,ld> bl={INF,INF};
  ll n=poi.size();ll ind;
  for(ll i=0;i<n;i++){
    pair<ld,ld> pp={poi[i].y,poi[i].x};
    if(pp<bl){
      ind=i;bl={poi[i].y,poi[i].x};
    }
  }
  swap(bl.F,bl.S);firstpoint=pt(bl.F,bl.S);
  vector<pt> cons;
  for(ll i=0;i<n;i++){
    if(i==ind)continue;cons.pb(poi[i]);
  }
  sort(cons.begin(),cons.end(),compare);
  hull.pb(firstpoint);ll m;
  for(auto z:cons){
    if(hull.size()<=1){hull.pb(z);continue;}
    pt pr,ppr;bool fl=true;
    while((m=hull.size())>=2){
      pr=hull[m-1];ppr=hull[m-2];
      ll ch=orient(ppr,pr,z);
      if(ch==-1){break;}
      else if(ch==1){hull.pop_back();continue;}
      else {
        ld d1,d2;
        d1=dist2(ppr,pr);d2=dist2(ppr,z);
        if(gt(d1,d2)){fl=false;break;}else {hull.↩
           pop_back();}
      }
    }
    if(fl){hull.push_back(z);}
  }
  return;
}
```

4.3 Convex Hull Trick

```
/*
maintains upper convex hull of lines ax+b and gives ↩
   minimum value at a given x
```

```
to add line ax+b: sameoldcht.addline(a,b), to get min
    value at x: sameoldcht.getbest(x)
to get maximum value at x add -ax-b as lines instead
   of ax+b and use -sameoldcht.getbest(x)
*/
const int N = 1e5 + 5;
int n;
int a[N];
int b[N];
long long dp[N];
struct line{
    long long a , b;
    double xleft;
    bool type;
    line(long long _a , long long _b){
        a = _a;
        b = _b;
        type = 0;
    }
    bool operator < (const line &other) const{
        if(other.type){
            return xleft < other.xleft;
        }
        return a > other.a;
    }
};
double meet(line x , line y){
    return 1.0 * (y.b - x.b) / (x.a - y.a);
}
struct cht{
    set < line > hull;
    cht(){
        hull.clear();
    }
    typedef set < line > :: iterator ite;
    bool hasleft(ite node){
        return node != hull.begin();
    }
    bool hasright(ite node){
        return node != prev(hull.end());
    }
    void updateborder(ite node){
        if(hasright(node)){
            line temp = *next(node);
            hull.erase(temp);
            temp.xleft = meet(*node , temp);
            hull.insert(temp);
        }
        if(hasleft(node)){
            line temp = *node;
            temp.xleft = meet(*prev(node) , temp);
            hull.erase(node);
            hull.insert(temp);
        }
        else{
            line temp = *node;
            hull.erase(node);
            temp.xleft = -1e18;
            hull.insert(temp);
        }
    }
    bool useless(line left , line middle , line right
       ){
        double x = meet(left , right);
        double y = x * middle.a + middle.b;
        double ly = left.a * x + left.b;
        return y > ly;
    }
    bool useless(ite node){
        if(hasleft(node) && hasright(node)){
            return useless(*prev(node) , *node , *
               next(node));
        }
        return 0;
    }
    void addline(long long a , long long b){
        line temp = line(a , b);
        auto it = hull.lower_bound(temp);
        if(it != hull.end() && it -> a == a){
            if(it -> b > b){
                hull.erase(it);
            }
            else{
                return;
            }
        }
        hull.insert(temp);
        it = hull.find(temp);
        if(useless(it)){
            hull.erase(it);
            return;
        }
        while(hasleft(it) && useless(prev(it))){
            hull.erase(prev(it));
        }
        while(hasright(it) && useless(next(it))){
            hull.erase(next(it));
        }
        updateborder(it);
    }
    long long getbest(long long x){
        if(hull.empty()){
            return 1e18;
        }
        line query(0 , 0);
        query.xleft = x;
        query.type = 1;
        auto it = hull.lower_bound(query);
        it = prev(it);
        return it -> a * x + it -> b;
    }
};
cht sameoldcht;
int main()
{
    scanf("%d" , &n);
```

```
    for(int i = 1 ; i <= n ; ++i){
        scanf("%d" , a + i);
    }
    for(int i = 1 ; i <= n ; ++i){
        scanf("%d" , b + i);
    }
    sameoldcht.addline(b[1] , 0);
    for(int i = 2 ; i <= n ; ++i){
        dp[i] = sameoldcht.getbest(a[i]);
        sameoldcht.addline(b[i] , dp[i]);
    }
    printf("%lld\n" , dp[n]);
}
```

5 Trees

5.1 BlockCut Tree

```
// code credits - http://codeforces.com/contest/487/↩
   submission/15921824
// Take care it is 0 indexed -_-
struct BiconnectedComponents {
    struct Edge {
        int from, to;
    };
    struct To {
        int to; int edge;
    };
    vector<Edge> edges;
    vector<vector<To> > g;
    vector<int> low, ord, depth;
    vector<bool> isArtic;
    vector<int> edgeColor;
    vector<int> edgeStack;
    int colors;
    int dfsCounter;
    void init(int n) {
        edges.clear();
        g.assign(n, vector<To>());
    }
    void addEdge(int u, int v) {
        if(u > v) swap(u, v);
        Edge e = { u, v };
        int ei = edges.size();
        edges.push_back(e);
        To tu = { v, ei }, tv = { u, ei };
        g[u].push_back(tu);
        g[v].push_back(tv);
    }
    void run() {
        int n = g.size(), m = edges.size();
        low.assign(n, -2);
        ord.assign(n, -1);
        depth.assign(n, -2);
        isArtic.assign(n, false);
        edgeColor.assign(m, -1);
        edgeStack.clear();
        colors = 0;
        for(int i = 0; i < n; ++ i) if(ord[i] == -1) ↩
           {
            dfsCounter = 0;
            dfs(i);
        }
    }
private:
    void dfs(int i) {
        low[i] = ord[i] = dfsCounter ++;
        for(int j = 0; j < (int)g[i].size(); ++ j) {
            int to = g[i][j].to, ei = g[i][j].edge;
            if(ord[to] == -1) {
                depth[to] = depth[i] + 1;
                edgeStack.push_back(ei);
                dfs(to);
                low[i] = min(low[i], low[to]);
                if(low[to] >= ord[i]) {
                    if(ord[i] != 0 || j >= 1)
                        isArtic[i] = true;
                    while(!edgeStack.empty()) {
                        int fi = edgeStack.back(); ↩
                           edgeStack.pop_back();
                        edgeColor[fi] = colors;
                        if(fi == ei) break;
                    }
                    ++ colors;
                }
            }else if(depth[to] < depth[i] - 1) {
                low[i] = min(low[i], ord[to]);
                edgeStack.push_back(ei);
            }
        }
    }
};
```

5.2 Bridges Online

```
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) {  // 2-edge connected comp.
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
}
int find_cc(int v) {  // connected comp.
    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;
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            path_b.push_back(b);
            b = find_2ecc(b);
            if (last_visit[b] == lca_iteration)
                lca = b;
            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);
    }
}
```

5.3 HLD

```
/*
v is adjacency matrix of tree. clear v[i],hdc[i]=0,i=-1 before every run
clear ord and curc=0
*/
const ll MAX=250005;
vll v[MAX],ord;
ll par[MAX],noc[MAX],hdc[MAX],curc,posinch[MAX],len[MAX],ti=-1;
ll sta[MAX],en[MAX],subs[MAX],level[MAX];
ll st[4*MAX],lazy[4*MAX];
ll n;
void dfs(ll x){
    subs[x]=1;
    for(auto z:v[x]){
        if(z!=par[x]){par[z]=x;level[z]=level[x]+1;
      dfs(z);subs[x]+=subs[z];
        }}}
void makehld(ll x){
    if(hdc[curc]==0){hdc[curc]=x;len[curc]=0;}
    noc[x]=curc;posinch[x]=++len[curc];
    ll a,b,c;a=b=0;ord.pb(x);sta[x]=++ti;
    for(auto z:v[x]){   if(z==par[x])continue;
        if(subs[z]>b){b=subs[z];a=z;}
    }
    if(a!=0)makehld(a);
```

```
    for(auto z:v[x]){if(z==par[x]||z==a)continue;curc↩
      ++;makehld(z);}
    en[x]=ti;
}
inline void upd(ll x,ll y)//to update on path from a ↩
  to b
{
  ll a,b,c,d;
  while(x!=y){
    a=hdc[noc[x]],b=hdc[noc[y]];
    if(a==b){
      if(level[x]>level[y])swap(x,y);c=sta[x],d=sta[y↩
        ];
      //lca=a;
      update(1,0,n-1,c+1,d);return;}
    if(level[a]>level[b])swap(a,b),swap(x,y);
    update(1,0,n-1,sta[b],sta[y]);y=par[b];}}//update↩
      on segment tree
int main() {
    //v is adjacency matrix
    for(i=1;i<=n;i++) v[i].clear(),hdc[i]=0,ti=-1;
    ord.clear(),curc=0;
    level[1]=0;par[1]=0;curc=1;dfs(1);makehld(1);
    cin>>m;
    while(m--){cin>>a>>b;upd(a,b);ll ans=sumq(1,0,n↩
      -1,0,n-1);}
}
```

5.4 LCA

```
const int N = int(1e5)+10;
const int LOGN = 20;
set<int> g[N];
int level[N];
int DP[LOGN][N];
int n,m;
/*----------- Pre-Processing ------------*/
/* Code Cridits : Tanuj Khattar codeforces submission↩
   */
void dfs0(int u)
{
  for(auto it=g[u].begin();it!=g[u].end();it++)
    if(*it!=DP[0][u])
    {
      DP[0][*it]=u;
      level[*it]=level[u]+1;
      dfs0(*it);
    }
}
void preprocess()
{
  level[0]=0;
  DP[0][0]=0;
  dfs0(0);
  for(int i=1;i<LOGN;i++)
    for(int j=0;j<n;j++)
      DP[i][j] = DP[i-1][DP[i-1][j]];
}
int lca(int a,int b)
{
  if(level[a]>level[b])swap(a,b);
  int d = level[b]-level[a];
  for(int i=0;i<LOGN;i++)
    if(d&(1<<i))
      b=DP[i][b];
  if(a==b)return a;
  for(int i=LOGN-1;i>=0;i--)
    if(DP[i][a]!=DP[i][b])
      a=DP[i][a],b=DP[i][b];
  return DP[0][a];
}
int dist(int u,int v)
{
  return level[u] + level[v] - 2*level[lca(u,v)];
}
```

5.5 Centroid Decompostion

```
/*
nx:maximum number of nodes
adj:adjacency list of tree,adj1: adjacency list of ↩
   centroid tree
par:parents of nodes in centroid tree,timstmp: ↩
   timestamps of nodes when they became centroids (↩
   helpful in comparing which of the two nodes became↩
    centroid first)
ssize,vis:utility arrays for storing subtree size and↩
    visit times in dfs
tim: utility for doing dfs (for deciding which nodes ↩
   to visit)
cntrorder: centroids stored in order in which they ↩
   were formed
dist[nx]: vector of vectors with dist[i][0][j]=number↩
    of nodes at distance of k in subtree of i in ↩
   centroid tree and dist[i][j][k]=number of nodes at↩
    distance k in jth child of i in centroid tree ↩
   ***(use adj while doing dfs instead of adj1)***
dfs: find subtree sizes visiting nodes starting from ↩
   root without visiting already formed centroids
dfs1: root- starting node, n- subtree size remaining ↩
   after removing centroids -> returns centroid in ↩
   subtree of root
preprocess: stores all values in dist array
*/
const int nx=1e5;
vector<int> adj[nx],adj1[nx]; //adj is adjacency list↩
    of tree and adj1 is adjacency list for centroid ↩
   tree
int par[nx],timstmp[nx],ssize[nx],vis[nx];//par is ↩
   parent of each node in centroid tree,ssize is ↩
   subtree size of each node in centroid tree,vis and↩
    timstmp are auxillary arrays for visit times in ↩
```

```
   dfs- timstmp contains nonzero values only for ↩
   centroids
int tim=1;
vector<int> cntrorder;//contains list of centroids ↩
   generated (in order)
vector<vector<int> > dist[nx];
int dfs(int root)
{
  vis[root]=tim;
  int t=0;
  for(auto i:adj[root])
  {
    if(!timstmp[i]&&vis[i]<tim)
      t+=dfs(i);
  }
  ssize[root]=t+1;return t+1;
}
int dfs1(int root,int n)
{
  vis[root]=tim;pair<int,int> mxc={0,-1};bool poss=↩
     true;
  for(auto i:adj[root])
  {
    if(!timstmp[i]&&vis[i]<tim)
      poss&=(ssize[i]<=n/2),mxc=max(mxc,{ssize[i],i})↩
         ;
  }
  if(poss&&(n-ssize[root])<=n/2)return root;
  return dfs1(mxc.second,n);
}
int findc(int root)
{
  dfs(root);
  int n=ssize[root];tim++;
  return dfs1(root,n);
}
void cntrdecom(int root,int p)
{
  int cntr=findc(root);
  cntrorder.push_back(cntr);
  timstmp[cntr]=tim++;
  par[cntr]=p;
  if(p>=0)adj1[p].push_back(cntr);
  for(auto i:adj[cntr])
    if(!timstmp[i])
      cntrdecom(i,cntr);
}
void dfs2(int root,int nod,int j,int dst)
{
  if(dist[root][j].size()==dst)dist[root][j].↩
     push_back(0);
  vis[nod]=tim;
  dist[root][j][dst]+=1;
  for(auto i:adj[nod])
  {
    if((timstmp[i]<=timstmp[root])||(vis[i]==vis[nod↩
       ]))continue;
    vis[i]=tim;dfs2(root,i,j,dst+1);
  }
}
void preprocess()
{
  for(int i=0;i<cntrorder.size();i++)
  {
    int root=cntrorder[i];
    vector<int> temp;
    dist[root].push_back(temp);
    temp.push_back(0);
    ++tim;
    dfs2(root,root,0,0);
    int cnt=0;
    for(int j=0;j<adj[root].size();j++)
    {
      int nod=adj[root][j];
      if(timstmp[nod]<timstmp[root])
        continue;
      dist[root].push_back(temp);
      ++tim;
      dfs2(root,nod,++cnt,1);
    }
  }
}
```

6 Maths

6.1 Chinese Remainder Theorem

```
/*solves system of equations x=rem[i]%mods[i] for any↩
    mod (need not be coprime)
intput:vector of remainders and moduli
output: pair of answer(x%lcm of modulo) and lcm of ↩
   all the modulo (returns -1 if it is inconsistent)↩
   */
ll GCD(ll a, ll b) { return (b == 0) ? a : GCD(b, a %↩
    b); }
inline ll LCM(ll a, ll b) { return a / GCD(a, b) * b;↩
    }
inline ll normalize(ll x, ll mod) { x %= mod; if (x <↩
    0) x += mod; return x; }
struct GCD_type { ll x, y, d; };
GCD_type ex_GCD(ll a, ll 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> &rem,vector<ll> &mods)
{
    ll n=rem.size();
    ll ans=rem[0];
    ll lcm=mods[0];
```

```
    for(ll i=1;i<n;i++)
    {
        auto pom=ex_GCD(lcm,mods[i]);
        ll x1=pom.x;
        ll d=pom.d;
        if((rem[i]-ans)%d!=0)return {-1,0};
        ans=normalize(ans+x1*(rem[i]-ans)/d%(mods[i]/d)*lcm,lcm*mods[i]/d);
        lcm=LCM(lcm,mods[i]); // you can save time by replacing above lcm * n[i] /d by lcm = lcm * n[i] / d
    }
    return {ans,lcm};
}
```

6.2 Discrete Log

```
// Discrete Log , Baby-Step Giant-Step , e-maxx
// The idea is to make two functions,
// f1(p) , f2(q) and find p,q s.t.
// f1(p) = f2(q) by storing all possible values of f1,
// and checking for q. In this case a^(x) = b (mod m) is
// solved by substituting x by p.n-q , where
// is choosen optimally , usually sqrt(m).
// returns a soln. for a^(x) = b (mod m)
// for given a,b,m . -1 if no. soln.
// complexity : O(sqrt(m).log(m))
// use unordered_map to remove log factor.
// IMP : works only if a,m are co-prime. But can be modified.
int solve (int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;
    int an = 1;
    for (int i=0; i<n; ++i)
        an = (an * a) % m;
    map<int,int> vals;
    for (int i=1, cur=an; i<=n; ++i) {
        if (!vals.count(cur))
            vals[cur] = i;
        cur = (cur * an) % m;
    }
    for (int i=0, cur=b; i<=n; ++i) {
        if (vals.count(cur)) {
            int ans = vals[cur] * n - i;
            if (ans < m) return ans;
        }
        cur = (cur * a) % m;
    }
    return -1;
}
```

6.3 NTT

```
/*
Kevin's different Code: https://s3.amazonaws.com/codechef_shared/download/Solutions/JUNE15/tester/MOREFB.cpp
****There is no problem that FFT can solve while this NTT cannot
  Case1: If the answer would be small choose a small enough NTT prime modulus
  Case2: If the answer is large(> ~1e9) FFT would not work anyway due to precision issues
  In Case2 use NTT. If max_answer_size=n*(largest_coefficient^2)
  So use two or three modulus to solve it
****Compute a*b%mod if a%mod*b%mod would result in overflow in O(log(a)) time:
  ll mulmod(ll a, ll b, ll mod) {
      ll res = 0;
      while (a != 0) {
          if (a & 1) res = (res + b) % m;
          a >>= 1;
          b = (b << 1) % m;
      }
      return res;
  }
Fastest NTT (can also do polynomial multiplication if max coefficients are upto 1e18 using 2 modulus and CRT)
How to use:
P=A*B
Polynomial1 = A[0]+A[1]*x^1+A[2]*x^2+..+A[n-1]*x^n-1
Polynomial2 = B[0]+B[1]*x^1+B[2]*x^2+..+B[n-1]*x^n-1
P=multiply(A,B)
A and B are not passed by reference because they are changed in multiply function
For CRT after obtaining answer modulo two primes p1 and p2:
x = a1 mod p1, x = a2 mod p2 => x=((a1*(m2^-1)%m1)*m2+(a2*(m1^-1)%m2)*m1)%m1m2
*** Before each call to multiply:
  set base=1,roots={0,1},rev={0,1},max_base=x (such that if mod=c*(2^k)+1 then x<=k and 2^x is greater than equal to nearest power of 2 of 2*n)
  root=primitive_root^((mod-1)/(2^max_base))
  For P=A*A use square function
Some useful modulo and examples
mod1=463470593 = 1768*2^18+1 primitive root = 3 => max_base=18,root=3^1768
mod2=469762049 = 1792*2^18+1 primitive root = 3 => max_base=18,root=3^1792
(mod1^-1)%mod2=313174774 (mod2^-1)%mod1=154490124
Some prime modulus and primitive root
    635437057   11
      639631361   6
      645922817   3
      648019969   17
      666894337   5
      683671553   3
      710934529   17
      715128833   3
      740294657   3
```

```
      754974721    11
      786432001    7
      799014913    13
      824180737    5
      880803841    26
      897581057    3
      899678209    7
      918552577    5
      924844033    5
      935329793    3
      943718401    7
      950009857    7
      962592769    7
      975175681    17
      985661441    3
      998244353    3
*/
//x = a1 mod m1, x = a2 mod m2, invm2m1 = (m2^-1)%m1, invm1m2 = (m1^-1)%m2, gives x%m1*m2
#define chinese(a1,m1,invm2m1,a2,m2,invm1m2) ((a1 *1ll* invm2m1 % m1 * 1ll*m2 + a2 *1ll* invm1m2 % m2 * 1ll*m1) % (m1 *1ll* m2))
int mod;//reset mod everytime with required modulus
inline int mul(int a,int b){return (a*1ll*b)%mod;}
inline int add(int a,int b){a+=b;if(a>=mod)a-=mod;return a;}
inline int sub(int a,int b){a-=b;if(a<0)a+=mod;return a;}
inline int power(int a,int b){int rt=1;while(b>0){if(b&1)rt=mul(rt,a);a=mul(a,a);b>>=1;}return rt;}
inline int inv(int a){return power(a,mod-2);}
inline void modadd(int &a,int &b){a+=b;if(a>=mod)a-=mod;}
int base = 1;
vector<int> roots = {0, 1};
vector<int> rev = {0, 1};
int max_base=18;  //x such that 2^x|(mod-1) and 2^x>max answer size(=2*n)
int root=202376916;   //primitive root^((mod-1)/(2^max_base))
void ensure_base(int nbase) {
  if (nbase <= base) {
    return;
  }
  assert(nbase <= max_base);
  rev.resize(1 << nbase);
  for (int i = 0; i < (1 << nbase); i++) {
    rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
  }
  roots.resize(1 << nbase);
  while (base < nbase) {
    int z = power(root, 1 << (max_base - 1 - base));
    for (int i = 1 << (base - 1); i < (1 << base); i++) {
      roots[i << 1] = roots[i];
      roots[(i << 1) + 1] = mul(roots[i], z);
    }
    base++;
  }
}
void fft(vector<int> &a) {
  int n = (int) a.size();
  assert((n & (n - 1)) == 0);
  int zeros = __builtin_ctz(n);
  ensure_base(zeros);
  int shift = base - zeros;
  for (int i = 0; i < n; i++) {
    if (i < (rev[i] >> shift)) {
      swap(a[i], a[rev[i] >> shift]);
    }
  }
  for (int k = 1; k < n; k <<= 1) {
    for (int i = 0; i < n; i += 2 * k) {
      for (int j = 0; j < k; j++) {
        int x = a[i + j];
        int y = mul(a[i + j + k], roots[j + k]);
        a[i + j] = x + y - mod;
        if (a[i + j] < 0) a[i + j] += mod;
        a[i + j + k] = x - y + mod;
        if (a[i + j + k] >= mod) a[i + j + k] -= mod;
      }
    }
  }
}
vector<int> multiply(vector<int> a, vector<int> b, int eq = 0) {
  int need = (int) (a.size() + b.size() - 1);
  int nbase = 0;
  while ((1 << nbase) < need) nbase++;
  ensure_base(nbase);
  int sz = 1 << nbase;
  a.resize(sz);
  b.resize(sz);
  fft(a);
  if (eq) b = a; else fft(b);
  int inv_sz = inv(sz);
  for (int i = 0; i < sz; i++) {
    a[i] = mul(mul(a[i], b[i]), inv_sz);
  }
  reverse(a.begin() + 1, a.end());
  fft(a);
  a.resize(need);
  return a;
}
vector<int> square(vector<int> a) {
  return multiply(a, a, 1);
}
```

6.4 Online FFT

```
//f[i]=sum over j from 0 to i-1 f[j]*g[i-1-j]
//handle f[0] and g[0] separately
const int nx=131072;int f[nx],g[nx];
void onlinefft(int a,int b,int c,int d)
{
  vector<int> v1,v2;
```

```
  v1.pb(f+a,f+b+1);v2.pb(g+c,g+d+1); vector<int> res=↩
     multiply(v1,v2);
  for(int i=0;i<res.size();i++)
    if(a+c+i+1<nx) f[a+c+i+1]=add(f[a+c+i+1],res[i]);
}
void precal()
{
  g[0]=1;
  for(int i=1;i<nx;i++)
    g[i]=power(i,i-1);
  f[1]=1;
  for(int i=1;i<=100000;i++)
  {
    f[i+1]=add(f[i+1],g[i]);f[i+1]=add(f[i+1],f[i]);
    f[i+2]=add(f[i+2],mul(f[i],g[1]));f[i+3]=add(f[i↩
       +3],mul(f[i],g[2]));
    for(int j=2;i%j==0&&j<nx;j=j*2) onlinefft(i-j,i↩
       -1,j+1,2*j);
  }
}
```

6.5 Langrange Interpolation

```
/* Input :
Degree of polynomial: k
Polynomial values at x=0,1,2,3,...,k
Output :
Polynomial value at x
Complexity: O(degree of polynomial)
Works only if the points are equally spaced
*/
ll lagrange(vll& v , int k, ll x,int mod){
    if(x <= k)
        return v[x];
    ll inn = 1;
    ll den = 1;
    for(int i = 1;i<=k;i++)
    {
        inn = (inn*(x - i))%mod;
        den = (den*(mod - i))%mod;
    }
    inn = (inn*inv(den % mod))%mod;
    ll ret = 0;
    for(int i = 0;i<=k;i++){
        ret = (ret + v[i]*inn)%mod;
        ll md1 = mod - ((x-i)*(k-i))%mod;
        ll md2 = ((i+1)*(x-i-1))%mod;
        if(i!=k)
            inn = (((inn*md1)%mod)*inv(md2 % mod))%↩
               mod;
    }
    return ret;
}
```

6.6 Matrix Struct

```
struct matrix{
    ld B[N][N], n;
    matrix(){n = N; memset(B,0,sizeof B);}
    matrix(int _n){
        n = _n; memset(B, 0, sizeof B);
    }
    void iden(){
      for(int i = 0; i < n; i++)
        B[i][i] = 1;
    }
    void operator += (matrix M){
        for(int i = 0; i < n; i++)
          for(int j = 0; j < n; j++)
            B[i][j] = add(B[i][j], M.B[i][j]);
    }
    void operator -= (matrix M){}
    void operator *= (ld b){}
    matrix operator - (matrix M){}
    matrix operator + (matrix M){
        matrix ret = (*this);
        ret += M; return ret;
    }
    matrix operator * (matrix M){
        matrix ret = matrix(n); memset(ret.B, 0, ↩
           sizeof ret.B);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n;j++)
                for(int k = 0; k < n; k++){
                    ret.B[i][j] = add(ret.B[i][j], ↩
                       mul(B[i][k], M.B[k][j]));
                }
        return ret;
    }
    matrix operator *= (matrix M){ *this = ((*this) *↩
       M);}
    matrix operator * (int b){
        matrix ret = (*this); ret *= b; return ret;
    }
    vector<double> multiply(const vector<double> & v)↩
       const{
      vector<double> ret(n);
      for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++){
          ret[i] += B[i][j] * v[j];
        }
      return ret;
    }
};
```

6.7 nCr(Non Prime Modulo)

```
// sandwich, jtnydv25 video
// calculates nCr, for small
// non-prime modulo, and (very) big n,r.
ll phimod;
vll pr,prn;vll fact;
ll power(ll a,ll x,ll mod){
  ll ans=1;
  while(x){
    if((1LL)&(x))ans=(ans*a)%mod;
    a=(a*a)%mod;x>>=1LL;
  }
  return ans;
}
// prime factorization of x.
// pr-> prime ; prn -> it's exponent
void getprime(ll x){
    pr.clear();prn.clear();
    ll i,j,k;
    for(i=2;(i*i)<=x;i++){
      k=0;while((x%i)==0){k++;x/=i;}
      if(k>0){pr.pb(i);prn.pb(k);}
    }
    if(x!=1){pr.pb(x);prn.pb(1);}
    return;
}
// factorials are calculated ignoring
//  multiples of p.
void primeproc(ll p,ll pe){   // p , p^e
    ll i,d;
    fact.clear();fact.pb(1);d=1;
    for(i=1;i<pe;i++){
      if(i%p){fact.pb((fact[i-1]*i)%pe);}
      else {fact.pb(fact[i-1]);}
    }
    return;
}
// again note this has ignored multiples of p
ll Bigfact(ll n,ll mod){
  ll a,b,c,d,i,j,k;
  a=n/mod;a%=phimod;a=power(fact[mod-1],a,mod);
  b=n%mod;a=(a*fact[b])%mod;
  return a;
}
// Chinese Remainder Thm.
vll crtval,crtmod;
ll crt(vll &val,vll &mod){
  ll a,b,c,d,i,j,k;b=1;
  for(ll z:mod)b*=z;
  ll ans=0;
  for(i=0;i<mod.size();i++){
    a=mod[i];c=b/a;
    d=power(c,(((a/pr[i])*(pr[i]-1))-1),a);
    c=(c*d)%b;c=(c*val[i])%b;ans=(ans+c)%b;
  }
  return ans;
}
// calculate for prime powers and
// take crt. For each prime power,
// first ignore multiples of p,
// and then do recursively, calculating
// the powers of p separately.
ll Bigncr(ll n,ll r,ll mod){
  ll a,b,c,d,i,j,k;ll p,pe;
  getprime(mod);ll Fnum=1;ll Fden;
  crtval.clear();crtmod.clear();
  for(i=0;i<pr.size();i++){
    Fnum=1;Fden=1;
    p=pr[i];pe=power(p,prn[i],1e17);
    primeproc(p,pe);
    a=1;d=0;
    phimod=(pe*(p-1LL))/p;
    ll n1=n,r1=r,nr=n-r;
    while(n1){
      Fnum=(Fnum*(Bigfact(n1,pe)))%pe;
      Fden=(Fden*(Bigfact(r1,pe)))%pe;
      Fden=(Fden*(Bigfact(nr,pe)))%pe;
      d+=n1-(r1+nr);
      n1/=p;r1/=p;nr/=p;
    }
    Fnum=(Fnum*(power(Fden,(phimod-1LL),pe)))%pe;
    if(d>=prn[i])Fnum=0;
    else Fnum=(Fnum*(power(p,d,pe)))%pe;
    crtmod.pb(pe);crtval.pb(Fnum);
  }
  // you can just iterate instead of crt
  // for(i=0;i<mod;i++){
  //  bool cg=true;
  //  for(j=0;j<crtmod.size();j++){
  //    if(i%crtmod[j]!=crtval[j])cg=false;
  //  }
  //  if(cg)return i;
  // }
  return crt(crtval,crtmod);
}
```

6.8 Primitive Root Generator

```
/*To find generator of U(p),we check for all
  g in [1,p]. But only for powers of the
  form phi(p)/p_j, where p_j is a prime factor of
  phi(p). Note that p is not prime here.
  Existence , if one of these :
  1. p = 1,2,4
  2. p = q^k , where q -> odd prime.
  3. p = 2.(q^k) , where q-> odd prime
  Note that a.g^(phi(p)) = 1 (mod p)
            b.there are phi(phi(p)) generators if ↩
               exists.
*/
// Finds "a" generator of U(p),
```

```
// multiplicative group of integers mod p.
// here calc_phi returns the toitent function for p
// Complexity : O(Ans.log(phi(p)).log(p)) + time for factorizing phi(p).
// By some theorem, Ans = O((log(p))^6). Should be fast generally.
int generator (int p) {
    vector<int> fact;
    int phi = calc_phi(p),  n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            fact.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        fact.push_back (n);
    for (int res=2; res<=p; ++res) {
        if(gcd(res,p)!=1)continue;
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= powmod (res, phi / fact[i], p) != 1;
        if (ok)  return res;
    }
    return -1;
}
```

7 Strings

7.1 Hashing Theory

```
If order not imp. and count/frequency imp. use this as hash fn:-
( (a_1+h)^k + (a_2+h)^k + (a_3+h)^k + (a_4+h)^k ) % p. Select : h,k,p
Alternate:
((x)^(a_1)+(x)^(a_2)+...+(x)^(a_k))%mod where x and mod are fixed and a_1...a_k is an unordered set
```

7.2 Manacher

```
// Same idea as Z_algo, Time : O(n)
// [l,r] represents : boundaries of rightmost detected subpalindrom(with max r)
// takes string s and returns a vector of lengths of odd length palindrom
// centered around that char(e.g abac for 'b' returns 2(not 3))
vll manacher_odd(string s){
    ll n = s.length(); vll d1(n);
    for(ll i = 0, l = 0, r = -1;i<n;i++){
        d1[i] = 1;
        if(i <= r){
            d1[i] = min(r-i+1,d1[l+r-i]); // use prev val
        }
        while(i+d1[i] < n && i-d1[i] >= 0 && s[i+d1[i]] == s[i-d1[i]]) d1[i]++; // trivial matching
        if(r < i+d1[i]-1) l=i-d1[i]+1, r=i+d1[i]-1; // update r
    }
    return d1;
}
// takes string s and returns vector of lengths of even length ...
// (it's centered around the right middle char, bb is centered around the later 'b')
vll manacher_even(string s){
    ll n = s.length(); vll d2(n);
    for(ll i = 0, l = 0, r = -1;i<n;i++){
        d2[i] = 0;
        if(i <= r){
            d2[i] = min(r-i+1,d2[l+r+1-i]);
        }
        while(i+d2[i] < n && i-d2[i]-1 >= 0 && s[i+d2[i]] == s[i-d2[i]-1]) d2[i]++;
        if(d2[i] > 0 && r < i+d2[i]-1) l=i-d2[i], r=i+d2[i]-1;
    }
    return d2;
}
// Other mtd : To do both things in one pass, add special char e.g string "abc" => "$a$b$c$"
```

7.3 Suffix Array

```
//code credits - https://cp-algorithms.com/string/suffix-array.html
/*Theory :-
Sorted array of suffixes = sorted array of cyclic shifts of string+$.
We consider a prefix of len. 2^k of the cyclic, in the kth iteration.
And find the sorted order, using values for (k-1)th iteration and
kind of radix sort. Could be thought as some kind of binary lifting.
String of len. 2^k -> combination of 2 strings of len. 2^(k-1), whose
order we know. Just radix sort on pair for next iteration.
Time :- O(nlog(n) + alphabet)
Applications :-
Finding the smallest cyclic shift;Finding a substring in a string;
```

```
Comparing two substrings of a string;Longest common ↩
   prefix of two substrings;
Number of different substrings.
*/
// return list of indices(permutation of indices ↩
   which are in sorted order)
vector<ll> sort_cyclic_shifts(string const& s) {
    ll n = s.size();
    const ll alphabet = 256;
    //********* change the alphabet size accordingly ↩
       and indexing ******************
        vector<ll> p(n), c(n), cnt(max(alphabet, n), ↩
           0);
    // p -> sorted order of 1-len prefix of each ↩
       cyclic shift index.
    // c -> class of a index
    // pn -> same as p for kth iteration . ||ly cn.
    for (ll i = 0; i < n; i++)
        cnt[s[i]]++;
    for (ll i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (ll i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    ll classes = 1;
    for (ll i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
        vector<ll> pn(n), cn(n);
    for (ll h = 0; (1 << h) < n; ++h) {
        for (ll i = 0; i < n; i++) {  // sorting w.r.↩
           t second part.
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (ll i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (ll i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (ll i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];   // sorting ↩
               w.r.t first (more significant) part.
        cn[p[0]] = 0;
        classes = 1;
        for (ll i = 1; i < n; i++) {  // determining ↩
           new classes in sorted array.
            pair<ll, ll> cur = {c[p[i]], c[(p[i] + (1↩
                << h)) % n]};
            pair<ll, ll> 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<ll> suffix_array_construction(string s) {
    s += "$";
    vector<ll> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
// For comparing two substring of length l starting ↩
   at i,j.
// k - 2^k > l/2. check the first 2^k part, if equal,
// check last 2^k part. c[k] is the c in kth iter of ↩
   S.A construction.
int compare(int i, int j, int l, int k) {
    pair<int, int> a = {c[k][i], c[k][(i+l-(1 << k))%↩
       n]};
    pair<int, int> b = {c[k][j], c[k][(j+l-(1 << k))%↩
       n]};
    return a == b ? 0 : a < b ? -1 : 1;
}
/*
Kasai's Algo for LCP construction :
Longest Common Prefix for consecutive suffixes in ↩
   suffix array.
lcp[i]=length of lcp of ith and (i+1)th suffix in the↩
    susffix array.
1. Consider suffixes in decreasing order of length.
2. Let p = s[i....n]. It will be somewhere in the S.A↩
   .We determine its lcp = k.
3. Then lcp of q=s[(i+1)....n] will be atleast k-1. ↩
   Why?
4. Remove the first char of p and its successor in ↩
   the S.A. These are suffixes with lcp k-1.
5. But note that these 2 may not be consecutive in S.↩
   A. But however lcp of strings in
   b/w have to be also atleast k-1.
*/
vector<ll> lcp_construction(string const& s, vector<↩
   ll> const& p) {
    ll n = s.size();
    vector<ll> rank(n, 0);
    for (ll i = 0; i < n; i++)
        rank[p[i]] = i;
    ll k = 0;
    vector<ll> lcp(n-1, 0);
    for (ll i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        ll 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;
}
```

7.4 Trie

```
const ll AS = 26; // alphabet size
ll go[MAX][AS];
ll cnt[MAX];ll cn=0;
// cn -> index of next new node
// convert all strings to vll
ll newNode() {
  for(ll i=0;i<AS;i++)
    go[cn][i]=-1;
  return cn++;
}
// call newNode once *******
// before adding anything **
void addTrie(vll &x) {
  ll v = 0;
  cnt[v]++;
  for(ll i=0;i<x.size();i++){
    ll y=x[i];
    if(go[v][y]==-1)
      go[v][y]=newNode();
    v=go[v][y];
    cnt[v]++;
  }
}
// returns count of substrings with prefix x
ll getcount(vll &x){
  ll v=0;
  for(i=0;i<x.size();i++){
    ll y=x[i];
    if(go[v][y]==-1)
      go[v][y]=newNode();
    v=go[v][y];
  }
  return cnt[v];
}
```

7.5 Z-algorithm

```
// [l,r] -> indices of the rightmost segment match
// (the detected segment that ends rightmost(with max r))
// 2 cases -> 1st. i <= r : z[i] is atleast min(r-i+1,z[i-l]), then match trivially
// 2nd. o.w compute z[i] with trivial matching
// update l,r
// Time : O(n)(asy. behavior), Proof : each iteration of inner while loop make r pointer advance to right,
// Applications:   1) Search substring(text t, pattern p) s = p + '$' + t.
// 3) String compression(s = t+t+...+t, then find |t|)
// 2) Number of distinct substrings (in O(n^2))
// (useful when appending or deleting characters online from the end or beginning)
vector<ll> z_function(string s) {
    ll n = (ll) s.length();
    vector<ll> z(n);
    for (ll i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]); // use previous z val
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) // trivial matching
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1; // update rightmost segment matched
    }
    return z;
}
```

7.6 Aho Corasick

```
const int K = 26;
// remember to set K
struct Vertex {
    int next[K]; bool leaf = false;
    int p = -1;char pch;
    int link = -1;
    int go[K];
    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};
vector<Vertex> aho(1);
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (aho[v].next[c] == -1) {
            aho[v].next[c] = aho.size();
            aho.emplace_back(v, ch);
        }
        v = aho[v].next[c];
    }
    aho[v].leaf = true;
```

```
}
int go(int v, char ch);
int get_link(int v) {
    if (aho[v].link == -1) {
        if (v == 0 || aho[v].p == 0)
            aho[v].link = 0;
        else
            aho[v].link = go(get_link(aho[v].p), aho[←
               v].pch);
    }
    return aho[v].link;
}
int go(int v, char ch) {
    int c = ch - 'a';
    if (aho[v].go[c] == -1) {
        if (aho[v].next[c] != -1)
            aho[v].go[c] = aho[v].next[c];
        else
            aho[v].go[c] = v == 0 ? 0 : go(get_link(v←
               ), ch);
    }
    return aho[v].go[c];
}
```

7.7 KMP

```
/*Time:O(n)(j increases n times(& j>=0) only so asy. ←
   O(n))
pi[i] = length of longset prefix of s ending at i
applications: search substring, # of different ←
   substrings(O(n^2)),
3) String compression(s = t+t+...+t, then find |t|, k←
   =n-pi[n-1],if k|n)
4) Building Automaton(Gray Code Example)*/
vector<ll> prefix_function(string s) {
    ll n = (ll)s.length();
    vector<ll> pi(n);
    for (ll i = 1; i < n; i++) {
        ll 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;
}
// searching s in t, returns all occurences(indices)
vector<ll> search(string s,string t){
    vll pi = prefix_function(s);
    ll m = s.length(); vll ans; ll j = 0;
    for(ll i=0;i<t.length();i++){
        while(j > 0 && t[i] != s[j])
            j = pi[j-1];
        if(t[i] == s[j]) j++;
        if(j == m) ans.pb(i-m+1);
    }
    return ans; // if ans empty then no occurence
}
```

7.8 Palindrome Tree

```
const ll MAX=1e5+15;
ll par[MAX];  // stores index of parent node
ll suli[MAX]; // stores index of suffix link
ll len[MAX];  // stores length of largest pallindrome←
   ending at that node
ll child[MAX][30];  // stores the children of the ←
   node
/*←
   ------------------------------------------------------
index 0 - root "-1"
index 1 - root  "0"
therefore node of s[i] is i+2
initialize all child[i][j] to -1
----------------------------------------------------------
   */
void eer_tree(string s){
  ll a,b,c,d,i,j,k,e,f;
  suli[1]=0;suli[0]=0;len[1]=0;len[0]=-1;
  ll n=s.length();
  for(i=0;i<n+10;i++)
    for(j=0;j<30;j++)child[i][j]=-1;
  ll cur=1;d=1;
  for(i=0;i<s.size();i++){
    ++d;
    while(true){
      a=i-1-len[cur];
      if(a>=0){
        if(s[a]==s[i]){
          if(child[cur][(ll)(s[i]-'a')]==-1){
            par[d]=cur;child[cur][(ll)(s[i]-'a')]=d;
            len[d]=len[cur]+2;cur=d;
          }
          else{
            par[d]=cur;len[d]=len[cur]+2;
            cur=child[cur][(ll)(s[i]-'a')];
          }
          break;
        }
      }
      if(cur==0)break;
      cur=suli[cur];
    }
    if(cur!=d)continue;
    if(len[d]==1)suli[d]=1;
    else{
      c=suli[par[d]];
      while(child[c][(ll)(s[i]-'a')]==-1){
        if(c==0)break;
        c=suli[c];
      }
```

```
      suli[d]=child[c][(ll)(s[i]-'a')];
    }
  }
}
```

7.9 Suffix Array

```
//code credits - https://cp-algorithms.com/string/suffix-array.html
/*Theory :-
Sorted array of suffixes = sorted array of cyclic shifts of string+$.
We consider a prefix of len. 2^k of the cyclic, in the kth iteration.
And find the sorted order, using values for (k-1)th iteration and
kind of radix sort. Could be thought as some kind of binary lifting.
String of len. 2^k -> combination of 2 strings of len. 2^(k-1), whose
order we know. Just radix sort on pair for next iteration.
Time :- O(nlog(n) + alphabet)
Applications :-
Finding the smallest cyclic shift;Finding a substring in a string;
Comparing two substrings of a string;Longest common prefix of two substrings;
Number of different substrings.
*/
// return list of indices(permutation of indices which are in sorted order)
vector<ll> sort_cyclic_shifts(string const& s) {
    ll n = s.size();
    const ll alphabet = 256;
    //********* change the alphabet size accordingly and indexing ******************
        vector<ll> p(n), c(n), cnt(max(alphabet, n), 0);
    // p -> sorted order of 1-len prefix of each cyclic shift index.
    // c -> class of a index
    // pn -> same as p for kth iteration . ||ly cn.
    for (ll i = 0; i < n; i++)
        cnt[s[i]]++;
    for (ll i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (ll i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    ll classes = 1;
    for (ll i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
        vector<ll> pn(n), cn(n);
    for (ll h = 0; (1 << h) < n; ++h) {
        for (ll i = 0; i < n; i++) {  // sorting w.r.t second part.
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (ll i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (ll i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (ll i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];   // sorting w.r.t first (more significant) part.
        cn[p[0]] = 0;
        classes = 1;
        for (ll i = 1; i < n; i++) {  // determining new classes in sorted array.
            pair<ll, ll> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
            pair<ll, ll> 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<ll> suffix_array_construction(string s) {
    s += "$";
    vector<ll> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
// For comparing two substring of length l starting at i,j.
// k - 2^k > l/2. check the first 2^k part, if equal,
// check last 2^k part. c[k] is the c in kth iter of S.A construction.
int compare(int i, int j, int l, int k) {
    pair<int, int> a = {c[k][i], c[k][(i+l-(1 << k))%n]};
    pair<int, int> b = {c[k][j], c[k][(j+l-(1 << k))%n]};
    return a == b ? 0 : a < b ? -1 : 1;
}
/*
Kasai's Algo for LCP construction :
Longest Common Prefix for consecutive suffixes in suffix array.
lcp[i]=length of lcp of ith and (i+1)th suffix in the susffix array.
```

```
1. Consider suffixes in decreasing order of length.
2. Let p = s[i....n]. It will be somewhere in the S.A.We determine its lcp = k.
3. Then lcp of q=s[(i+1)....n] will be atleast k-1. Why?
4. Remove the first char of p and its successor in the S.A. These are suffixes with lcp k-1.
5. But note that these 2 may not be consecutive in S.A. But however lcp of strings in
 b/w have to be also atleast k-1.
*/
vector<ll> lcp_construction(string const& s, vector<ll> const& p) {
    ll n = s.size();
    vector<ll> rank(n, 0);
    for (ll i = 0; i < n; i++)
        rank[p[i]] = i;
    ll k = 0;
    vector<ll> lcp(n-1, 0);
    for (ll i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        ll 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;
}
```

7.10 Suffix Tree

```
const int N=1000000, // set it more than 2*(len. of string)
string str; // input string for which the suffix tree is being built
int chi[N][26],
lef[N], // left...
rig[N], // ...and right boundaries of the substring of a which correspond to incoming edge
par[N], // parent of the node
sfli[N],  // suffix link
tv,tp,la,
ts; // the number of nodes
void ukkadd(int c) {
  suff:;
  if (rig[tv]<tp){
    if (chi[tv][c]==-1){chi[tv][c]=ts;lef[ts]=la;
    par[ts++]=tv;tv=sfli[tv];tp=rig[tv]+1;goto suff;}
    tv=chi[tv][c];tp=lef[tv];
  }
  if (tp==-1 || c==str[tp]-'a')tp++;
  else {
    lef[ts]=lef[tv]; rig[ts]=tp-1; par[ts]=par[tv];
    chi[ts][str[tp]-'a']=tv; chi[ts][c]=ts+1;
    lef[ts+1]=la; par[ts+1]=ts; lef[tv]=tp; par[tv]=ts;
    chi[par[ts]][str[lef[ts]]-'a']=ts;  ts+=2;
    tv=sfli[par[ts-2]]; tp=lef[ts-2];
    while (tp <= rig[ts-2]) {
      tv=chi[tv][str[tp]-'a']; tp+=rig[tv]-lef[tv]+1;}
    if (tp == rig[ts-2]+1) sfli[ts-2]=tv; else sfli[ts-2]=ts;
    tp=rig[tv]-(tp-rig[ts-2])+2;goto suff;
  }
}
void build() {
  ts=2; tv=0; tp=0;
  ll ss = str.size();ss*=2;ss+=15;
  fill(rig,rig+ss,(int)str.size()-1);
  // initialize data for the root of the tree
  sfli[0]=1; lef[0]=-1; rig[0]=-1;
  lef[1]=-1; rig[1]=-1; for(ll i=0;i<ss;i++)
  fill (chi[i], chi[i]+27,-1);
  fill(chi[1],chi[1]+26,0);
  // add the text to the tree, letter by letter
  for (la=0; la<(int)str.size(); ++la)
  ukkadd (str[la]-'a');
}
```