IME ACM-ICPC Team Notebook

Contents

1 Template + vimrc

1.1 Template

```cpp
#include <bits/stdc++.h>
using namespace std;

#define st first
#define nd second
#define mp make_pair
#define pb push_back
#define cl(x, v) memset((x), (v), sizeof(x))
#define gcd(x,y) __gcd((x),(y))

#ifndef ONLINE_JUDGE
    #define db(x) cerr << #x << " == " << x << endl
    #define dbs(x) cerr << x << endl
    #define _ << ", " <<
#else
    #define db(x) ((void)0)
    #define dbs(x) ((void)0)
#endif

typedef long long ll;
typedef long double ld;

typedef pair<int, int> pii;
typedef pair<int, pii> piii;
typedef pair<ll, ll> pll;
typedef vector<int> vi;

const ld EPS = 1e-9, PI = acos(-1.);
const int INF = 0x3f3f3f3f, MOD = 1e9+7;
const int N = 1e5+5;

int main() {
    //freopen("in", "r", stdin);
    //freopen("out", "w", stdout);
    return 0;
}
```

1.2 vimrc

```
syntax on
set et ts=2 sw=0 sts=-1 ai nu hls cindent
nnoremap ; :
vnoremap ; :
noremap <c-j> 15gj
noremap <c-k> 15gk
nnoremap <s-k> i<CR><ESC>
inoremap ,. <esc>
vnoremap ,. <esc>
nnoremap ,. <esc>
```

2 Graphs

2.1 Articulation points and bridges

```cpp
// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;

void articulation(int u) {
  low[u] = num[u] = ++cnt;
  for (int v : adj[u]) {
    if (!num[v]) {
      par[v] = u; ch[u]++;
      articulation(v);
      if (low[v] >= num[u]) art[u] = 1;
      if (low[v] >  num[u]) // u-v bridge
        low[u] = min(low[u], low[v]);
    }
    else if (v != par[u]) low[u] = min(low[u], num[v]);
  }
}

for (int i = 0; i < n; ++i) if (!num[i])
  articulation(i), art[i] = ch[i]>1;
```

2.2 Strongly Connected Components

```cpp
// Kosaraju - SCC O(V+E)
vi adj[N], adjt[N];
int ord[N], ordn, scc[N], sccn, vis[N];

void dfs(int u) {
  vis[u] = 1;
  for (int v : adj[u]) if (!vis[v]) dfs(v);
  ord[ordn++] = u;
}

void dfst(int u) {
  vis[u] = 0;
  for (int v : adjt[u]) if (vis[v]) dfst(v);
  scc[u] = sccn;
}

// in main
sccn = ordn = 1;
for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
for (int i = n; i > 0; --i) if (vis[ord[i]]) dfst(ord[i]), sccn++;
```

2.3 MST (Kruskal)

```cpp
// Kruskal - MST O(ElogE)
vector<piii> edges;
// + Union-find

sort(edges.begin(), edges.end());
int cost = 0;
for (auto e : edges)
  if (find(e.nd.st) != find(e.nd.nd))
    unite(e.nd.st, e.nd.nd), cost += e.st;
```

2.4 MST (Prim)

```cpp
// Prim - MST O(ElogE)
vi adj[N], adjw[N];
int vis[N];

priority_queue<pii> pq;
pq.push(mp(0, 0));

while (!pq.empty()) {
```

```cpp
  int u = pq.top().nd;
  pq.pop();
  if (vis[u]) continue;
  vis[u]=1;
  for (int i = 0; i < adj[u].size(); ++i) {
    int v = adj[u][i];
    int w = adjw[u][i];
    if (!vis[v]) pq.push(mp(-w, v));
  }
}
```

2.5 Shortest Path (SPFA)

```cpp
// Shortest Path Faster Algoritm O(VE)
int dist[N], inq[N];

cl(dist,63);
queue<int> q;
q.push(0); dist[0] = 0; inq[0] = 1;

while (!q.empty()) {
  int u = q.front(); q.pop(); inq[u]=0;
  for (int i = 0; i < adj[u].size(); ++i) {
    int v = adj[u][i], w = adjw[u][i];
    if (dist[v] > dist[u] + v) {
      dist[v] = dist[u] + w;
      if (!inq[v]) q.push(v), inq[v] = 1;
    }
  }
}
```

2.6 Shortest Path (Floyd-Warshall)

```cpp
// Floyd-Warshall (APSP) O(V^3)
int adj[N][N]; // no-edge = INF

for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
```

2.7 Max Flow

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5+1, INF = 1e9;
struct edge {int v, c, f;};

int n, src, snk, h[N], ptr[N];
vector<edge> edgs;
vector<int> g[N];

void add_edge (int u, int v, int c) {
  int k = edgs.size();
  edgs.push_back({v, c, 0});
  edgs.push_back({u, 0, 0});
  g[u].push_back(k);
  g[v].push_back(k+1);
}

bool bfs() {
  memset(h, 0, sizeof h);
  queue<int> q;
  h[src] = 1;
  q.push(src);
  while(!q.empty()) {
    int u = q.front(); q.pop();
    for(int i : g[u]) {
      int v = edgs[i].v;
      if (!h[v] and edgs[i].f < edgs[i].c)
        q.push(v), h[v] = h[u] + 1;
    }
  }
  return h[snk];
```

```cpp
}

int dfs (int u, int flow) {
  if (!flow or u == snk) return flow;
  for (int &i = ptr[u]; i < g[u].size(); ++i) {
    edge &dir = edgs[g[u][i]], &rev = edgs[g[u][i]^1];
    int v = dir.v;
    if (h[v] != h[u] + 1)  continue;
    int inc = min(flow, dir.c - dir.f);
    inc = dfs(v, inc);
    if (inc) {
      dir.f += inc, rev.f -= inc;
      return inc;
    }
  }
  return 0;
}

int dinic() {
  int flow = 0;
  while (bfs()) {
    memset(ptr, 0, sizeof ptr);
    while (int inc = dfs(src, INF)) flow += inc;
  }
  return flow;
}
```

2.8   Min Cost Max Flow

```cpp
// w: weight or cost, c : capacity
struct edge {int v, f, w, c; };

int node_count, flw_lmt=INF, src, snk, flw, cst, p[N], d[N], et[N];
vector<edge> e;
vector<int> g[N];

void add_edge(int u, int v, int w, int c) {
  int k = e.size();
  node_count = max(node_count, u+1);
  node_count = max(node_count, v+1);
  g[u].push_back(k);
  g[v].push_back(k+1);
  e.push_back({ v, 0,  w, c });
  e.push_back({ u, 0, -w, 0 });
}

void clear() {
  flw_lmt = INF;
  for(int i=0; i<node_count; ++i) g[i].clear();
  e.clear();
  node_count = 0;
}

void min_cost_max_flow() {
  flw = 0, cst = 0;
  while (flw < flw_lmt) {
    memset(et, 0, node_count * sizeof et[0]);
    memset(d, 63, node_count * sizeof d[0]);
    deque<int> q;
    q.push_back(src), d[src] = 0;

    while (!q.empty()) {
      int u = q.front(); q.pop_front();
      et[u] = 2;

      for(int i : g[u]) {
        edge &dir = e[i];
        int v = dir.v;
        if (dir.f < dir.c and d[u] + dir.w < d[v]) {
          d[v] = d[u] + dir.w;
          if (et[v] == 0) q.push_back(v);
          else if (et[v] == 2) q.push_front(v);
          et[v] = 1;
          p[v] = i;
        }
      }
    }

    if (d[snk] > INF) break;

    int inc = flw_lmt - flw;
    for (int u=snk; u != src; u = e[p[u]^1].v) {
      edge &dir = e[p[u]];
      inc = min(inc, dir.c - dir.f);
    }
```

```cpp
    for (int u=snk; u != src; u = e[p[u]^1].v) {
      edge &dir = e[p[u]], &rev = e[p[u]^1];
      dir.f += inc;
      rev.f -= inc;
      cst += inc * dir.w;
    }

    if (!inc) break;
    flw += inc;
  }
}
```

2.9   Max Bipartite Cardinality Matching (Kuhn)

```cpp
// Kuhn - Maximum Cardinality Bipartite Matching (MCBM) O(VE)
// TIP: If too slow, shuffle nodes and try again.
int x, vis[N], b[N], ans;

bool match(int u) {
  if (vis[u] == x) return 0;
  vis[u] = x;
  for (int v : adj[u])
    if (!b[v] or match(b[v])) return b[v]=u;
  return 0;
}

for (int i = 1; i <= n; ++i) ++x, ans += match(i);

// Maximum Independent Set on bipartite graph
MIS + MCBM = V

// Minimum Vertex Cover on bipartite graph
MVC = MCBM
```

2.10   Lowest Common Ancestor

```cpp
// Lowest Common Ancestor <O(nlogn), O(logn)>
const int N = 1e6, M = 25;
int anc[M][N], h[N], rt;

// TODO: Calculate h[u] and set anc[0][u] = parent of node u for each u

// build (sparse table)
anc[0][rt] = rt; // set parent of the root to itself
for (int i = 1; i < M; ++i)
  for (int j = 1; j <= n; ++j)
    anc[i][j] = anc[i-1][anc[i-1][j]];

// query
int lca(int u, int v) {
  if (h[u] < h[v]) swap(u, v);
  for (int i = M-1; i >= 0; --i) if (h[u]-(1<<i) >= h[v])
    u = anc[i][u];

  if (u == v) return u;

  for (int i = M-1; i >= 0; --i) if (anc[i][u] != anc[i][v])
    u = anc[i][u], v = anc[i][v];
  return anc[0][u];
}
```

2.11   2-SAT

```cpp
// 2-SAT - O(V+E)
// For each variable x, we create two nodes in the graph: u and !u
// If the variable has index i, the index of u and !u are: 2*i and 2*i+1
int ans[N], cn, vis[N], sccn, scc[N], scch[N];
vector<int> stk, adj[N], radj[N];

void koj(int u) {
  vis[u] = 1;
  for (int v : adj[u]) if (!vis[v]) koj(v);
  stk.pb(u);
}
```

```cpp
void rkoj(int u) {
  vis[u] = 0;
  for (int v : radj[u]) if (vis[v]) rkoj(v);
  scc[u] = sccn;
}

void rsat(int);

void sat(int u) {
  if (ans[u] != -1) return;
  ans[u] = 1, rsat(u^1);
  for (int v : adj[u]) sat(v);
}

void rsat(int u) {
  if (ans[u] != -1) return;
  ans[u] = 0, sat(u^1);
  for (int v : radj[u]) rsat(v);
}

bool sat() {
  sccn = 1; // Kosaraju's
  for (int i = 2; i <= cn; i++) if (!vis[i]) koj(i);
  while (stk.size()) {
    int u = stk.back(); stk.pop_back();
    if (vis[u]) rkoj(u), scch[sccn++] = u;
  }

  // Checks if exists a valid assingment.
  for (int i = 2; i <= cn; i+=2) if (scc[i] == scc[i^1]) return 0;

  memset(ans, -1, sizeof ans);
  // To set value of x to true (or false) call sat(x) (or sat(x^1)) below.

  // Finds a valid assignment. Erase if not needed.
  for(int i = sccn - 1; i > 0; --i) sat(scch[i]);

  // Check for inconsistencies in implication graph.
  for (int u = 2; u <= cn; u++) if (ans[u] == 1)
    for (int v : adj[u]) if (!ans[v]) return 0;

  return 1;
}

void add_clause(int x, int y) {
  cn = max(cn, max(x|1, y|1));
   adj[x].pb(y),  adj[y^1].pb(x^1);
  radj[y].pb(x), radj[x^1].pb(y^1);
}

void reset() {
  memset(ans, -1, sizeof ans);
  for(int i=0; i<=cn; ++i) adj[i].clear(), radj[i].clear();
  cn = 0;
}
```

2.12 Erdos-Gallai

```cpp
// Erdos-Gallai - O(nlogn)
// check if it's possible to create a simple graph (undirected edges) from
// a sequence of vertice's degrees
bool gallai(vector<int> v) {
  vector<ll> sum;
  sum.resize(v.size());

  sort(v.begin(), v.end(), greater<int>());
  sum[0] = v[0];
  for (int i = 1; i < v.size(); i++) sum[i] = sum[i-1] + v[i];
  if (sum.back() % 2) return 0;

  for (int k = 1; k < v.size(); k++) {
    int p = lower_bound(v.begin(), v.end(), k, greater<int>()) - v.begin();
    if (p < k) p = k;
    if (sum[k-1] > 1ll*k*(p-1) + sum.back() - sum[p-1]) return 0;
  }
  return 1;
}
```

3 Mathematics

3.1 Basics

```cpp
// Greatest Common Divisor & Lowest Common Multiple
ll gcd(ll a, ll b) { return b ? gcd(b, a%b) : a; }
ll lcm(ll a, ll b) { return a*(b/gcd(a, b)); }

// Multiply caring overflow
ll mulmod(ll a, ll b, ll m = MOD) {
  ll r=0;
  for (a %= m; b; b>>=1, a=(a*2)%m) if (b&1) r=(r+a)%m;
  return r;
}

// Fast exponential
ll fexp(ll a, ll b, ll m = MOD) {
  ll r=1;
  for (a %= m; b; b>>=1, a=(a*a)%m) if (b&1) r=(r*a)%m;
  return r;
}

// Multiplicative Inverse
ll inv[N];
inv[1] = 1;
for (int i = 2; i < MOD; ++i)
  inv[i] = (MOD - (MOD/i)*inv[MOD%i]%MOD)%MOD;

// Fibonacci
// Fn = Fn-1 + Fn-2
// F0 = 0 ; F1 = 1
f[0] = 0; f[1] = 1;
for (int i = 2; i <= n; i++) f[i] = f[i-1] + f[i-2];

// Catalan
// Cn = (2n)! / ((n+1)! * n!)
// 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440
cat[n] = (2*(2*n-1)/(n+1)) * cat[n-1]

// Stirling
// S(n, 1) = S(n, n) = 1
// S(n, k) = k*S(n-1, k) + S(n-1, k-1)
```

3.2 Euler Phi

```cpp
// Euler phi (totient)
int ind = 0, pf = primes[0], ans = n;
while (1ll*pf*pf <= n) {
  if (n%pf==0) ans -= ans/pf;
  while (n%pf==0) n /= pf;
  pf = primes[++ind];
}
if (n != 1) ans -= ans/n;

// IME2014
int phi[N];
void totient() {
  for (int i = 1; i < N; ++i)  phi[i]=i;
  for (int i = 2; i < N; i+=2) phi[i]>>=1;
  for (int j = 3; j < N; j+=2) if (phi[j]==j) {
    phi[j]--;
    for (int i = 2*j; i < N; i+=j) phi[i]=phi[i]/j*(j-1);
  }
}
```

3.3 Extended Euclidean

```cpp
// Extended Euclid: gcd(a, b) = x*a + y*b
void euclid(int a, int b, int &x, int &y, int &d) {
  if (b) euclid(b, a%b, y, x, d), y -= x*(a/b);
  else x = 1, y = 0, d = a;
}
```

3.4 Miller-Rabin

```
// Miller-Rabin - Primarily Test O(k*logn*logn*logn)
bool miller(ll a, ll n) {
  if (a >= n) return 1;
  ll s = 0, d = n-1;
  while (d%2 == 0 and d) d >>= 1, s++;
  ll x = fexp(a, d, n);
  if (x == 1 or x == n-1) return 1;
  for (int r = 0; r < s; r++, x = mulmod(x,x,n)) {
    if (x == 1) return 0;
    if (x == n-1) return 1;
  }
  return 0;
}

bool isprime(ll n) {
  int base[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
  for (int i = 0; i < 12; ++i) if (!miller(base[i], n)) return 0;
  return 1;
}
```

3.5 Prime Factors

```
// Prime factors (up to 9*10^13. For greater see Pollard Rho)
vi factors;
int ind=0, pf = primes[0];
while (pf*pf <= n) {
  while (n%pf == 0) n /= pf, factors.pb(pf);
  pf = primes[++ind];
}
if (n != 1) factors.pb(n);
```

3.6 Pollard-Rho

```
// Pollard Rho - Integer factoring O(n^1/4)
// requires n to be composite (use Miller Rabin to test)
ll pollard(ll n) {
  ll x, y, d, c=-1;
  if (n%2==0) return 2;
  while (1) {
    y = x = 2;
    while (1) {
      x = mulmod(x,x,n); x = (x-c)%n;
      y = mulmod(y,y,n); y = (y-c)%n;
      y = mulmod(y,y,n); y = (y-c)%n;
      d = gcd(abs(n+y-x), n);
      if (d == n) break;
      else if (d > 1) return d;
    }
    c--;
  }
}

// Factorize number using pollar
void fator(ll n, vector<ll>& v) {
  if (isprime(n)) { v.pb(n); return; }
      ;                                          ll f = pollard(n)
  ll f = pollard(n);
  factorize(f, v); factorize(n/f, v);
}
```

3.7 Fast Fourier Transform

```
// Fast Fourier Transform - O(nlogn)

/*
// Use struct instead. Performance will be way better!
typedef complex<ld> T;
T a[N], b[N];
*/
```

```
struct T {
  ld x, y;
  T() : x(0), y(0) {}
  T(ld a, ld b=0) : x(a), y(b) {}

  T operator/=(ld k) { x/=k; y/=k; return (*this); }
  T operator*(T a) const { return T(x*a.x - y*a.y, x*a.y + y*a.x); }
  T operator+(T a) const { return T(x+a.x, y+a.y); }
  T operator-(T a) const { return T(x-a.x, y-a.y); }
} a[N], b[N];

// a: vector containing polynomial
// n: power of two greater or equal product size
/*
// Use iterative version!
void fft_recursive(T* a, int n, int s) {
  if (n == 1) return;
  T tmp[n];
  for (int i = 0; i < n/2; ++i)
    tmp[i] = a[2*i], tmp[i+n/2] = a[2*i+1];

  fft_recursive(&tmp[0], n/2, s);
  fft_recursive(&tmp[n/2], n/2, s);

  T wn = T(cos(s*2*PI/n), sin(s*2*PI/n)), w(1,0);
  for (int i = 0; i < n/2; i++, w=w*wn)
    a[i] = tmp[i] + w*tmp[i+n/2],
    a[i+n/2] = tmp[i] - w*tmp[i+n/2];
}
*/

void fft(T* a, int n, int s) {
  for (int i=0, j=0; i<n; i++) {
    if (i>j) swap(a[i], a[j]);
    for (int l=n/2; (j^=l) < l; l>>=1);
  }

  for(int i = 1; (1<<i) <= n; i++){
    int M = 1 << i;
    int K = M >> 1;
    T wn = T(cos(s*2*PI/M), sin(s*2*PI/M));
    for(int j = 0; j < n; j += M) {
      T w = T(1, 0);
      for(int l = j; l < K + j; ++l){
        T t = w*a[l + K];
        a[l + K] = a[l]-t;
        a[l] = a[l] + t;
        w = wn*w;
      }
    }
  }
}

// assert n is a power of two greater of equal product size
// n = na + nb; while (n&(n-1)) n++;
void multiply(T* a, T* b, int n) {
  fft(a,n,1);
  fft(b,n,1);
  for (int i = 0; i < n; i++) a[i] = a[i]*b[i];
  fft(a,n,-1);
  for (int i = 0; i < n; i++) a[i] /= n;
}

// Convert to integers after multiplying:
// (int)(a[i].x + 0.5);
```

3.8 Number Theoretic Transform

```
// Number Theoretic Transform - O(nlogn)

// if long long is not necessary, use int instead to improve performance
const int mod = 20*(1<<23)+1;
const int root = 3;

ll w[N];

// a: vector containing polynomial
// n: power of two greater or equal product size
void ntt(ll* a, int n, bool inv) {
  for (int i=0, j=0; i<n; i++) {
    if (i>j) swap(a[i], a[j]);
    for (int l=n/2; (j^=l) < l; l>>=1);
  }

  // TODO: Rewrite this loop using FFT version
```

```
    ll k, t, nrev;
    w[0] = 1;
    k = exp(root, (mod-1) / n, mod);
    for (int i=1;i<=n;i++) w[i] = w[i-1] * k % mod;
    for(int i=2; i<=n; i<<=1) for(int j=0; j<n; j+=i) for(int l=0; l<(i/2); l++) {
        int x = j+l, y = j+l+(i/2), z = (n/i)*l;
        t = a[y] * w[inv ? (n-z) : z] % mod;
        a[y] = (a[x] - t + mod) % mod;
        a[x] = (a[j+l] + t) % mod;
    }

    nrev = exp(n, mod-2, mod);
    if (inv) for(int i=0; i<n; ++i) a[i] = a[i] * nrev % mod;
}

// assert n is a power of two greater of equal product size
// n = na + nb; while (n&(n-1)) n++;
void multiply(ll* a, ll* b, int n) {
    ntt(a, n, 0);
    ntt(b, n, 0);
    for (int i = 0; i < n; i++) a[i] = a[i]*b[i] % mod;
    ntt(a, n, 1);
}
```

3.9 Fast Walsh-Hadamard Transform

```
// Fast Walsh-Hadamard Transform - O(nlogn)
//
// Multiply two polynomials, but instead of x^a * x^b = x^(a+b)
// we have x^a * x^b = x^(a XOR b).
//
// Also you can look at it as a multivariable cyclic convolution
void fwht(ll* a, int n, bool inv) {
    for(int l=1; 2*l <= n; l<<=1) {
        for(int i=0; i < n; i+=2*l) {
            for(int j=0; j<l; j++) {
                ll u = a[i+j], v = a[i+l+j];

                a[i+j] = (u+v) % MOD;
                a[i+l+j] = (u-v+MOD) % MOD;
                // % is kinda slow, you can use add() macro instead
                // #define add(x,y) (x+y >= MOD ? x+y-MOD : x+y)
            }
        }
    }

    if(inv) {
        for(int i=0; i<n; i++) {
            a[i] = a[i] / n;
        }
    }
}

// assert n is a power of two!
```

3.10 Chinese Remainder

```
// finds z (mod p*q) so z = x (mod p) and z = y (mod q)
ll chinese(ll x, ll p, ll y, ll q) {
    ll s, r, d, mod = p*q;
    euclid(p, q, r, s, d);
    return ((x*(s*q%mod) + y*(r*p%mod) % mod) + mod) % mod;
}
```

3.11 Primitive Root

```
// Finds a primitive root modulo p
ll root(ll p) {
    ll n = p-1;
    vector<ll> fact;

    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) {
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= exp(res, (p-1) / fact[i], p) != 1;
        if (ok)  return res;
    }

    return -1;
}
```

3.12 Gaussian Elimination (double)

```
//double A[N][M+1], X[M]

for(int j=0; j<m; j++) { //collumn to eliminate
    int l = j;
    for(int i=j+1; i<n; i++) //find largest pivot
        if(fabs(A[i][j])>fabs(A[l][j]))
            l=i;
    for(int k = 0; k < m+1; k++) { //Swap lines
        swap(A[l][k],A[j][k]);
    }
    for(int i = j+1; i < n; i++) { //eliminate column
        double t=A[i][j]/A[j][j];
        for(int k = j; k < m+1; k++)
            A[i][k]-=t*A[j][k];
    }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
    for(int j = m-1; j > i; j--)
        A[i][m] -= A[i][j]*X[j];
    X[i]=A[i][m]/A[i][i];
}
```

3.13 Gaussian Elimination (modulo prime)

```
//ll A[N][M+1], X[M]

for(int j=0; j<m; j++) { //collumn to eliminate
    int l = j;
    for(int i=j+1; i<n; i++) //find nonzero pivot
        if(A[i][j]%p)
            l=i;
    for(int k = 0; k < m+1; k++) { //Swap lines
        swap(A[l][k],A[j][k]);
    }
    for(int i = j+1; i < n; i++) { //eliminate column
        ll t=mulmod(A[i][j],inv(A[j][j],p),p);
        for(int k = j; k < m+1; k++)
            A[i][k]=(A[i][k]-mulmod(t,A[j][k],p)+p)%p;
    }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
    for(int j = m-1; j > i; j--)
        A[i][m] = (A[i][m] - mulmod(A[i][j],X[j],p)+p)%p;
    X[i] = mulmod(A[i][m],inv(A[i][i],p),p);
}
```

3.14 Gaussian Elimination (extended inverse)

```
// Gauss-Jordan Elimination with Scaled Partial Pivoting
// Extended to Calculate Inverses - O(n^3)
// To get more precision choose m[j][i] as pivot the element such that m[j][i] / mx[j] is maximized.
// mx[j] is the element with biggest absolute value of row j.

ld C[N][M]; // N = 1000, M = 2*N+1;
int row, col;

bool elim() {
    for(int i=0; i<row; ++i) {
        int p = i; // Choose the biggest pivot
```

```cpp
    for(int j=i; j<row; ++j) if (abs(C[j][i]) > abs(C[p][i])) p = j;
    for(int j=i; j<col; ++j) swap(C[i][j], C[p][j]);

    if (!C[i][i]) return 0;

    ld c = 1/C[i][i]; // Normalize pivot line
    for(int j=0; j<col; ++j) C[i][j] *= c;

    for(int k=i+1; k<col; ++k) {
      ld c = -C[k][i]; // Remove pivot variable from other lines
      for(int j=0; j<col; ++j) C[k][j] += c*C[i][j];
    }
  }

  // Make triangular system a diagonal one
  for(int i=row-1; i>=0; --i) for(int j=i-1; j>=0; --j) {
    ld c = -C[j][i];
    for(int k=i; k<col; ++k) C[j][k] += c*C[i][k];
  }

  return 1;
}

// Finds inv, the inverse of matrix m of size n x n.
// Returns true if procedure was successful.
bool inverse(int n, ld m[N][N], ld inv[N][N]) {
  for(int i=0; i<n; ++i) for(int j=0; j<n; ++j)
    C[i][j] = m[i][j], C[i][j+n] = (i == j);

  row = n, col = 2*n;
  bool ok = elim();

  for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) inv[i][j] = C[i][j+n];
  return ok;
}

// Solves linear system m*x = y, of size n x n
bool linear_system(int n, ld m[N][N], ld *x, ld *y) {
  for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) C[i][j] = m[i][j];
  for(int j=0; j<n; ++j) C[n][j] = x[j];

  row = n, col = n+1;
  bool ok = elim();

  for(int j=0; j<n; ++j) y[j] = C[n][j];
  return ok;
}
```

3.15 Golden Section Search (Ternary Search)

```cpp
double gss(double l, double r) {
    double m1 = r-(r-l)/gr, m2 = l+(r-l)/gr;
    double f1 = f(m1), f2 = f(m2);
    while(fabs(l-r)>EPS) {
        if(f1>f2) l=m1, f1=f2, m1=m2, m2=l+(r-l)/gr, f2=f(m2);
        else r=m2, f2=f1, m2=m1, m1=r-(r-l)/gr, f1=f(m1);
    }
    return l;
}
```

3.16 Josephus

```cpp
// UFMG
/* Josephus Problem - It returns the position to be, in order to not die. O(n)*/
/* With k=2, for instance, the game begins with 2 being killed and then n+2, n+4, ... */
ll josephus(ll n, ll k) {
  if(n==1) return 1;
  else return (josephus(n-1, k)+k-1)%n+1;
}

/* Another Way to compute the last position to be killed - O(d * log n) */
ll josephus(ll n, ll d) {
  ll K = 1;
  while (K <= (d - 1)*n) K = (d * K + d - 2) / (d - 1);
  return d * n + 1 - K;
}
```

3.17 Simpson Rule

```cpp
// Simpson Integration Rule
// define the function f
double f(double x) {
  // ...
}

double simpson(double a, double b, int n = 1e6) {
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i += 2) s += 4 * f(a + h*i);
    for (int i = 2; i < n; i += 2) s += 2 * f(a + h*i);
    return s*h/3;
}
```

3.18 Discrete Log (Baby-step Giant-step)

```cpp
// solve the discrete equation a^k = m mod(p).
// i.e find k such that the equation above is satisfied.
ll discrete_log(ll a, ll m, ll p) {
  unordered_map<ll, ll> babystep;

  ll b = 1, an = a;
  // set b to the ceil of sqrt(p):
  while(b*b < p) b++, an = (an * a) % p;

  // babysteps:
  ll bstep = m;
  for(ll i=0; i<=b; i++) {
    babystep[bstep] = i;
    bstep = (bstep * a) % p;
  }

  // giantsteps:
  ll gstep = an;
  for(ll i=1; i<=b; i++) {
    if(babystep.count(gstep)) {
      return (b * i - babystep[gstep]);
    }
    gstep = (gstep * an) % p;
  }

  // returns -1 if there isn't any possible value for the answer.
  return -1;
}
```

3.19 Mobius Function

```cpp
// 1 if n == 1
// 0 if exists x | n%x  == 0
// else (-1)^k, k = #(p) | p is prime and n%p == 0

int mobius(int n){
  int p = 0;
  int tmp = n;
  for(int i = 2; i*i <= n; i++)
    if(n%i == 0){
      n /= i;
      p++;
      if(n%i == 0) return 0;
    }
  return p&1 ? 1 : -1;
}
```

4 Strings

4.1 Rabin-Karp

```
// Rabin-Karp - String Matching + Hashing O(n+m)
const int B = 31;
char s[N], p[N];
int n, m; // n = strlen(s), m = strlen(p)

void rabin() {
  if (n<m) return;

  int hp = 0, hs = 0, E = 1;
  for (int i = 0; i < m; ++i)
    hp = ((hp*B)%MOD + p[i])%MOD,
    hs = ((hs*B)%MOD + s[i])%MOD,
    E = (E*B)%MOD;

  if (hs == hp) { /* matching position 0 */ }
  for (int i = m; i < n; ++i) {
    hs = ((hs*B)%MOD + s[i])%MOD;
    hs = (hs + (MOD - (s[i-m]*E)%MOD)%MOD)%MOD;
    if (hs == hp) { /* matching position i-m+1 */ }
  }
}
```

4.2 Knuth-Morris-Pratt

```
// Knuth-Morris-Pratt - String Matching O(n+m)
char s[N], p[N];
int b[N], n, m; // n = strlen(s), m = strlen(p);

void kmppre() {
  b[0] = -1;
  for (int i = 0, j = -1; i < m; b[++i] = ++j)
    while (j >= 0 and p[i] != p[j])
      j = b[j];
}

void kmp() {
  for (int i = 0, j = 0; i < n;) {
    while (j >= 0 and s[i] != p[j]) j=b[j];
    i++, j++;
    if (j == m) {
      // match position i-j
      j = b[j];
    }
  }
}
```

4.3 Z Function

```
void z_f(char *s, int *z) {
  int l=0,r=0, n=strlen(s);
  for(int i = 1; i<n; i++) {
    if(i <= r)
      z[i]=min(z[i-l],r-i+1);
    while(z[i]+i<n and s[z[i]+i]==s[z[i]])
      z[i]++;
    if(r<i+z[i]-1)
      l=i,r=i+z[i]-1;
  }
}
```

4.4 Prefix Function

```
void p_f(char *s, int *pi) {
  int n = strlen(s);
  for(int i = 2; i <= n; i++) {
    pi[i]=pi[i-1];
    while(pi[i]>0 and s[pi[i]]!=s[i])
      pi[i]=pi[pi[i]];
    if(s[pi[i]]==s[i-1])
      pi[i]++;
  }
}
```

4.5 Recursive-String Matching

```
void p_f(char *s, int *pi) {
  int n = strlen(s);
  pi[0]=pi[1]=0;
  for(int i = 2; i <= n; i++) {
    pi[i] = pi[i-1];
    while(pi[i]>0 and s[pi[i]]!=s[i])
      pi[i]=pi[pi[i]];
    if(s[pi[i]]==s[i-1])
      pi[i]++;
  }
}

int main() {
  //...
  //Initialize prefix function
  char p[N]; //Pattern
  int len = strlen(p); //Pattern size
  int pi[N]; //Prefix function
  p_f(p, pi);

  // Create KMP automaton
  int A[N][128]; //A[i][j]: from state i (size of largest suffix of text which is prefix of pattern),
      append character j -> new state A[i][j]
  for( char c : ALPHABET )
    A[0][c] = (p[0] == c);
  for( int i = 1; p[i]; i++ ) {
    for( char c : ALPHABET ) {
      if(c==p[i])
        A[i][c]=i+1; //match
      else
        A[i][c]=A[pi[i]][c]; //try second largest suffix
    }
  }

  //Create KMP "string appending" automaton
  // g_n = g_(n-1) + char(n) + g_(n-1)
  // g_0 = "", g_1 = "a", g_2 = "aba", g_3 = "abacaba", ...
  int F[M][N]; //F[i][j]: from state j (size of largest suffix of text which is prefix of pattern), append
      string g_i -> new state F[i][j]
  for(int i = 0; i < m; i++) {
    for(int j = 0; j <= len; j++) {
      if(i==0)
        F[i][j] = j; //append empty string
      else {
        int x = F[i-1][j]; //append g_(i-1)
        x = A[x][j]; //append character j
        x = F[i-1][x]; //append g_(i-1)
        F[i][j] = x;
      }
    }
  }

  //Create number of matches matrix
  int K[M][N]; //K[i][j]: from state j (size of largest suffix of text which is prefix of pattern), append
      string g_i -> K[i][j] matches
  for(int i = 0; i < m; i++) {
    for(int j = 0; j <= len; j++) {
      if(i==0)
        K[i][j] = (j==len); //append empty string
      else {
        int x = F[i-1][j]; //append g_(i-1)
        x = A[x][j]; //append character j

        K[i][j] = K[i-1][j] /*append g_(i-1)*/ + (x==len) /*append character j*/ + K[i-1][x]; /*
            append g_(i-1)*/
      }
    }
  }
  //number of matches in g_k
  int answer = K[0][k];
  //...
}
```

4.6 Aho-Corasick

```
// Aho Corasick - <O(sum(m)), O(n + #matches)>
// Multiple string matching

int p[N], f[N], nxt[N][26], ch[N];
```

```cpp
int tsz=1; // size of the trie

int cnt[N]; // used to know number of matches

// used to know which strings matches.
// S is the number of strings. Can use set instead
const int S = 2e3+5;
bitset<S> elem[N];

void init() {
  tsz=1;
  memset(f, 0, sizeof(f));
  memset(nxt, 0, sizeof(nxt));
  memset(cnt, 0, sizeof(cnt));
  for (int i = 0; i < N; i++) elem[i].reset();
}

void add(const string &s, int x) {
  int cur = 1; // the first element of the trie is the root

  for (int i=0; s[i]; ++i) {
    int j = s[i] - 'a';
    if (!nxt[cur][j]) {
      tsz++;
      p[tsz] = cur;
      ch[tsz] = j;
      nxt[cur][j] = tsz;
    }
    cur = nxt[cur][j];
  }

  cnt[cur]++; //
  elem[cur].set(x);
}

void build() {
  queue<int> q;

  for(int i=0; i<26; ++i) {
    nxt[0][i] = 1;
    if (nxt[1][i]) q.push(nxt[1][i]);
  }

  while (!q.empty()) {
    int v = q.front(); q.pop();
    int u = f[p[v]];

    while (u and !nxt[u][ch[v]]) u = f[u];
    f[v] = nxt[u][ch[v]];
    cnt[v] += cnt[f[v]];

    for (int i = 0; i < 26; ++i) if (nxt[v][i])
      q.push(nxt[v][i]);
  }
}

// Return ans to get number of matches
// Return a map (or global array) if want to know how many of each string have matched
bitset<S> match(char *s) {
  int ans = 0;    // used to know the number of matches
  bitset<S> found; // used to know which strings matches

  int x = 1;
  for (int i = 0; s[i]; ++i) {
    int t = s[i] - 'a';
    while (x and !nxt[x][t]) x = f[x];
    x = nxt[x][t];

    // match found
    ans += cnt[x];
    found |= elem[x];
  }

  return found;
}
```

4.7   Manacher

```cpp
// Manacher (Longest Palindromic String) - O(n)
int lps[2*N+5];
char s[N];

int manacher() {
  int n = strlen(s);
```

```cpp
  string p (2*n+3, '#');
  p[0] = '^';
  for (int i = 0; i < n; i++) p[2*(i+1)] = s[i];
  p[2*n+2] = '$';

  int k = 0, r = 0, m = 0;
  int l = p.length();
  for (int i = 1; i < l; i++) {
    int o = 2*k - i;
    lps[i] = (r > i) ? min(r-i, lps[o]) : 0;
    while (p[i + 1 + lps[i]] == p[i - 1 - lps[i]]) lps[i]++;
    if (i + lps[i] > r) k = i, r = i + lps[i];
    m = max(m, lps[i]);
  }
  return m;
}
```

4.8   Suffix Array

```cpp
// Suffix Array O(nlogn)
char s[N];
int n, r, ra[N], sa[N], tra[N], tsa[N], c[N]; // n = strlen(s)

void count(int k) {
  int sum = 0, x = max(300, n); cl(c, 0);
  for (int i = 0; i < n; ++i) c[i+k<n ? ra[i+k] : 0]++;
  for (int i = 0; i < x; ++i) sum += c[i], c[i] = sum-c[i];
  for (int i = 0; i < n; ++i) tsa[c[sa[i]+k<n ? ra[sa[i]+k] : 0]++] = sa[i];
  for (int i = 0; i < n; ++i) sa[i] = tsa[i];
}

void suffixarray() {
  for (int i = 0; i < n; ++i) ra[i] = s[i], sa[i] = i;
  for (int k = 1; k < n; k <<= 1) {
    count(k); count(0);
    tra[sa[0]] = r = 0;
    for (int i = 1; i < n; ++i)
      tra[sa[i]] = (ra[sa[i]] == ra[sa[i-1]] and ra[sa[i]+k] == ra[sa[i-1]+k]) ? r : ++r;
    for (int i = 0; i < n; ++i) ra[i] = tra[i];
    if (ra[sa[n-1]] == n-1) break;
  }
}

// String matching with SA O(mlogn)
int mlo, mhi;
int match(char p[], int m) {
  int lo = 0, hi = n-1;
  while (lo < hi) {
    int mid = (lo+hi)/2;
    int res = strncmp(s+sa[mid], p, m);
    if (res < 0) lo = mid+1;
    else hi = mid;
  }
  if (strncmp(s+sa[lo], p, m)) return 0; // no match
  mlo = lo;

  lo = 0, hi = n-1;
  while (lo < hi) {
    int mid = (lo+hi)/2;
    int res = strncmp(s+sa[mid], p, m);
    if (res <= 0) lo = mid+1;
    else hi = mid;
  }
  if (strncmp(s+sa[hi], p, m)) hi--;
  mhi = hi;
  return 1;
}

// Longest Common Prefix with SA O(n)
// strcat(s, "$"); n = strlen(s);
int phi[N], plcp[N], lcp[N];
void calcLCP() {
  int l = 0;
  phi[sa[0]] = -1; plcp[sa[0]] = 0;
  for (int i = 1; i < n; ++i) phi[sa[i]] = sa[i-1];
  for (int i = 0; i < n; ++i) {
    if (phi[i] == -1) continue;
    while (s[i+l] == s[phi[i]+l]) l++;
    plcp[i] = l;
    l = max(l-1, 0);
  }
  for (int i = 0; i < n; ++i) lcp[i] = plcp[sa[i]];
}
```

```
// Longest Repeated Substring O(n)
int lrs = 0;
for (int i = 0; i < n; ++i) lrs = max(lrs, lcp[i]);

// Longest Common Substring O(n)
// m = strlen(s);
// strcat(s, "$"); strcat(s, p); strcat(s, "#");
// n = strlen(s);
int lcs = 0;
for (int i = 1; i < n; ++i) if ((sa[i] < m) != (sa[i-1] < m))
  lcs = max(lcs, lcp[i]);
```

4.9   Suffix Automaton

```
// Suffix Automaton Construction - O(n)

const int N = 1e6+1, K = 26;
int sl[2*N], len[2*N], sz, last;
ll cnt[2*N];
map<int, int> adj[2*N];

void add(int c) {
  int u = sz++;
  len[u] = len[last] + 1;
  cnt[u] = 1;

  int p = last;
  while(p != -1 and !adj[p][c])
    adj[p][c] = u, p = sl[p];

  if (p == -1) sl[u] = 0;
  else {
    int q = adj[p][c];
    if (len[p] + 1 == len[q]) sl[u] = q;
    else {
      int r = sz++;
      len[r] = len[p] + 1;
      sl[r] = sl[q];
      adj[r] = adj[q];
      while(p != -1 and adj[p][c] == q)
        adj[p][c] = r, p = sl[p];
      sl[q] = sl[u] = r;
    }
  }

  last = u;
}

void clear() {
  for(int i=0; i<=sz; ++i) adj[i].clear();
  last = 0;
  sz = 1;
  sl[0] = -1;
}

void build(char *s) {
  clear();
  for(int i=0; s[i]; ++i) add(s[i]);
}

// Pattern matching - O(|p|)
bool check(char *p) {
  int u = 0, ok = 1;
  for(int i=0; p[i]; ++i) {
    u = adj[u][p[i]];
    if (!u) ok = 0;
  }
  return ok;
}

// Substring count - O(|p|)
ll d[2*N];

void substr_cnt(int u) {
  d[u] = 1;
  for(auto p : adj[u]) {
    int v = p.second;
    if (!d[v]) substr_cnt(v);
    d[u] += d[v];
  }
}

ll substr_cnt() {
  memset(d, 0, sizeof d);
  substr_cnt(0);
```

```
  return d[0] - 1;
}

// k-th Substring - O(|s|)
// Just find the k-th path in the automaton.
// Can be done with the value d calculated in previous problem.

// Smallest cyclic shift - O(|s|)
// Build the automaton for string s + s. And adapt previous dp
// to only count paths with size |s|.

// Number of occurences - O(|p|)
vector<int> t[2*N];

void occur_count(int u) {
  for(int v : t[u]) occur_count(v), cnt[u] += cnt[v];
}

void build_tree() {
  for(int i=1; i<=sz; ++i)
    t[sl[i]].push_back(i);
  occur_count(0);
}

ll occur_count(char *p) {
  // Call build tree once per automaton
  int u = 0;
  for(int i=0; p[i]; ++i) {
    u = adj[u][p[i]];
    if (!u) break;
  }
  return !u ? 0 : cnt[u];
}

// First occurence - (|p|)
// Store the first position of occurence fp.
// Add the the code to add function:
// fp[u] = len[u] - 1;
// fp[r] = fp[q];

// To answer a query, just output fp[u] - strlen(p) + 1
// where u is the state corresponding to string p

// All occurences - O(|p| + |ans|)
// All the occurences can reach the first occurence via suffix links.
// So every state that contains a occreunce is reacheable by the
// first occurence state in the suffix link tree. Just do a DFS in this
// tree, starting from the first occurence.
// OBS: cloned nodes will output same answer twice.

// Smallest substring not contained in the string - O(|s| * K)
// Just do a dynamic programming:
// d[u] = 1 // if d does not have 1 transition
// d[u] = 1 + min d[v] // otherwise

// LCS of 2 Strings - O(|s| + |t|)
// Build automaton of s and traverse the automaton wih string t
// mantaining the current state and the current lenght.
// When we have a transition: update state, increase lenght by one.
// If we don't update state by suffix link and the new lenght will
// should be reduced (if bigger) to the new state length.
// Answer will be the maximum length of the whole traversal.

// LCS of n Strings - O(n*|s|*K)
// Create a new string S = s_1 + d1 + ... + s_n + d_n,
// where d_i are delimiters that are unique (d_i != d_j).
// For each state use DP + bitmask to calculate if it can
// reach a d_i transition without going through other d_j.
// The answer will be the biggest len[u] that can reach all
// d_i's.
```

5   Data Structures

5.1   Sparse Table 2D

```
// 2D Sparse Table - <O(n^2 (log n) ^ 2), O(1)>
const int N = 1e3+1, M = 10;
int t[N][N], v[N][N], dp[M][M][N][N], lg[N], n, m;
```

```cpp
void build() {
    int k = 0;
    for(int i=1; i<N; ++i) {
        if (1<<k == i/2) k++;
        lg[i] = k;
    }

    // Set base cases
    for(int x=0; x<n; ++x) for(int y=0; y<m; ++y) dp[0][0][x][y] = v[x][y];
    for(int j=1; j<M; ++j) for(int x=0; x<n; ++x) for(int y=0; y+(1<<j)<=m; ++y)
        dp[0][j][x][y] = max(dp[0][j-1][x][y], dp[0][j-1][x][y+(1<<j-1)]);

    // Calculate sparse table values
    for(int i=1; i<M; ++i) for(int j=0; j<M; ++j)
        for(int x=0; x+(1<<i)<=n; ++x) for(int y=0; y+(1<<j)<=m; ++y)
            dp[i][j][x][y] = max(dp[i-1][j][x][y], dp[i-1][j][x+(1<<i-1)][y]);
}

int query(int x1, int x2, int y1, int y2) {
    int i = lg[x2-x1+1], j = lg[y2-y1+1];
    int m1 = max(dp[i][j][x1][y1], dp[i][j][x2-(1<<i)+1][y1]);
    int m2 = max(dp[i][j][x1][y2-(1<<j)+1], dp[i][j][x2-(1<<i)+1][y2-(1<<j)+1]);
    return max(m1, m2);
}
```

5.2   Fenwick Tree 2D

```cpp
// Fenwick Tree 2D / Binary Indexed Tree 2D
int bit[N][N];

void add(int i, int j, int v) {
    for (; i < N; i+=i&-i)
        for (int jj = j; jj < N; jj+=jj&-jj)
            bit[i][jj] += v;
}

int query(int i, int j) {
    int res = 0;
    for (; i; i-=i&-i)
        for (int jj = j; jj; jj-=jj&-jj)
            res += bit[i][jj];
    return res;
}

// Whole BIT 2D set to 1
void init() {
    cl(bit,0);
    for (int i = 1; i <= r; ++i)
        for (int j = 1; j <= c; ++j)
            add(i, j, 1);
}

// Return number of positions set
int query(int imin, int jmin, int imax, int jmax) {
    return query(imax, jmax) - query(imax, jmin-1) - query(imin-1, jmax) + query(imin-1, jmin-1);
}

// Find all positions inside rect (imin, jmin), (imax, jmax) where position is set
void proc(int imin, int jmin, int imax, int jmax, int v, int tot) {
    if (tot < 0) tot = query(imin, jmin, imax, jmax);
    if (!tot) return;

    int imid = (imin+imax)/2, jmid = (jmin+jmax)/2;
    if (imin != imax) {
        int qnt = query(imin, jmin, imid, jmax);
        if (qnt) proc(imin, jmin, imid, jmax, v, qnt);
        if (tot-qnt) proc(imid+1, jmin, imax, jmax, v, tot-qnt);
    } else if (jmin != jmax) {
        int qnt = query(imin, jmin, imax, jmid);
        if (qnt) proc(imin, jmin, imax, jmid, v, qnt);
        if (tot-qnt) proc(imin, jmid+1, imax, jmax, v, tot-qnt);
    } else {
        // single position set!
        // now process position!!!
        add(imin, jmin, -1);
    }
}
```

5.3   Range Update Point Query Fenwick Tree

```cpp
struct BIT {
    ll b[N]={};
    ll sum(int x) {
        ll r=0;
        for(x+=2;x;x-=x&-x)
            r += b[x];
        return r;
    }
    void upd(int x, ll v) {
        for(x+=2;x<N;x+=x&-x)
            b[x]+=v;
    }
};
struct BITRange {
    BIT a,b;
    ll sum(int x) {
        return a.sum(x)*x+b.sum(x);
    }
    void upd(int l, int r, ll v) {
        a.upd(l,v), a.upd(r+1,-v);
        b.upd(l,-v*(l-1)), b.upd(r+1,v*r);
    }
};
```

5.4   Segment Tree 2D

```cpp
// Segment Tree 2D - O(nlog(n)log(n)) of Memory and Runtime
const int N = 1e8+5, M = 2e5+5;
int n, k=1, st[N], lc[N], rc[N];

void addx(int x, int l, int r, int u) {
    if (x < l or r < x) return;

    st[u]++;
    if (l == r) return;

    if(!rc[u]) rc[u] = ++k, lc[u] = ++k;
    addx(x, l, (l+r)/2, lc[u]);
    addx(x, (l+r)/2+1, r, rc[u]);
}

// Adds a point (x, y) to the grid.
void add(int x, int y, int l, int r, int u) {
    if (y < l or r < y) return;

    if (!st[u]) st[u] = ++k;
    addx(x, 1, n, st[u]);

    if (l == r) return;

    if(!rc[u]) rc[u] = ++k, lc[u] = ++k;
    add(x, y, l, (l+r)/2, lc[u]);
    add(x, y, (l+r)/2+1, r, rc[u]);
}

int countx(int x, int l, int r, int u) {
    if (!u or x < l) return 0;
    if (r <= x) return st[u];

    return countx(x, l, (l+r)/2, lc[u]) +
           countx(x, (l+r)/2+1, r, rc[u]);
}

// Counts number of points dominated by (x, y)
// Should be called with l = 1, r = n and u = 1
int count(int x, int y, int l, int r, int u) {
    if (!u or y < l) return 0;
    if (r <= y) return countx(x, 1, n, st[u]);

    return count(x, y, l, (l+r)/2, lc[u]) +
           count(x, y, (l+r)/2+1, r, rc[u]);
}
```

5.5   Persistent Segment Tree

```cpp
// Persistent Segtree
// Memory: O(n logn)
// Operations: O(log n)

int li[N], ri[N]; // [li(u), ri(u)] is the interval of node u
```

```
int st[N], lc[N], rc[N]; // Value, left son and right son of node u
int stsz; // Size of segment tree

// Returns root of initial tree.
// i and j are the first and last elements of the tree.
int init(int i, int j) {
  int v = ++stsz;
  li[v] = i, ri[v] = j;

  if (i != j) {
    rc[v] = init(i, (i+j)/2);
    rc[v] = init((i+j)/2+1, j);
    st[v] = /* calculate value from rc[v] and rc[v] */;
  } else {
    st[v] = /* insert initial value here */;
  }

  return v;
}

// Gets the sum from i to j from tree with root u
int sum(int u, int i, int j) {
  if (j < li[u] or ri[u] < i) return 0;
  if (i <= li[u] and ri[u] <= j) return st[u];
  return sum(rc[u], i, j) + sum (rc[u], i, j);
}

// Copies node j into node i
void clone(int i, int j) {
  li[i] = li[j], ri[i] = ri[j];
  st[i] = st[j];
  rc[i] = rc[j], rc[i] = rc[j];
}

// Sums v to index i from the tree with root u
int update(int u, int i, int v) {
  if (i < li[u] or ri[u] < i) return u;

  clone(++stsz, u);
  u = stsz;
  rc[u] = update(rc[u], i, v);
  rc[u] = update(rc[u], i, v);

  if (li[u] == ri[u]) st[u] += v;
  else st[u] = st[rc[u]] + st[rc[u]];

  return u;
}
```

5.6 Persistent Segment Tree (Naum)

```
// Persistent Segment Tree
int n;
int rcnt;
int lc[M], rc[M], st[M];

int update(int p, int l, int r, int i, int v) {
  int rt = ++rcnt;
  if (l == r) { st[rt] = v; return rt; }

  int mid = (l+r)/2;
  if (i <= mid) lc[rt] = update(lc[p], l, mid, i, v), rc[rt] = rc[p];
  else          rc[rt] = update(rc[p], mid+1, r, i, v), lc[rt] = lc[p];
  st[rt] = st[lc[rt]] + st[rc[rt]];

  return rt;
}

int query(int p, int l, int r, int i, int j) {
  if (l > j or r < i) return 0;
  if (i <= l and r <= j) return st[p];

  return query(lc[p], l, (l+r)/2, i, j)+query(rc[p], (l+r)/2+1, r, i, j);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) {
    int a;
    scanf("%d", &a);
    r[i] = update(r[i-1], 1, n, i, 1);
  }

  return 0;
}
```

5.7 Heavy-Light Decomposition

```
// Heavy-Light Decomposition
vector<int> adj[N];
int par[N], h[N];

int chainno, chain[N], head[N], chainpos[N], chainsz[N], pos[N], arrsz;
int sc[N], sz[N];

void dfs(int u) {
  sz[u] = 1, sc[u] = 0; // nodes 1-indexed (0-ind: sc[u]=-1)
  for (int v : adj[u]) if (v != par[u]) {
    par[v] = u, h[v] = h[u]+1, dfs(v);
    sz[u]+=sz[v];
    if (sz[sc[u]] < sz[v]) sc[u] = v; // 1-indexed (0-ind: sc[u]<0 or ...)
  }
}

void hld(int u) {
  if (!head[chainno]) head[chainno] = u; // 1-indexed
  chain[u] = chainno;
  chainpos[u] = chainsz[chainno];
  chainsz[chainno]++;
  pos[u] = ++arrsz;

  if (sc[u]) hld(sc[u]);

  for (int v : adj[u]) if (v != par[u] and v != sc[u])
    chainno++, hld(v);
}

int lca(int u, int v) {
  while (chain[u] != chain[v]) {
    if (h[head[chain[u]]] < h[head[chain[v]]]) swap(u, v);
    u = par[head[chain[u]]];
  }
  if (h[u] > h[v]) swap(u, v);
  return u;
}

int query_up(int u, int v) {
  if (u == v) return 0;
  int ans = -1;
  while (1) {
    if (chain[u] == chain[v]) {
      if (u == v) break;
      ans = max(ans, query(1, 1, n, chainpos[v]+1, chainpos[u]));
      break;
    }

    ans = max(ans, query(1, 1, n, chainpos[head[chain[u]]], chainpos[u]));
    u = par[head[chain[u]]];
  }
  return ans;
}

int query(int u, int v) {
  int l = lca(u, v);
  return max(query_up(u, l), query_up(v, l));
}
```

5.8 Centroid Decomposition

```
// Centroid decomposition
vector<int> adj[N];
int forb[N], sz[N], par[N];
int n, m;
unordered_map<int, int> dist[N];

void dfs(int u, int p) {
  sz[u] = 1;
  for(int v : adj[u]) {
    if(v != p and !forb[v]) {
      dfs(v, u);
      sz[u] += sz[v];
    }
  }
}
```

```cpp
int find_cen(int u, int p, int qt) {
  for(int v : adj[u]) {
    if(v == p or forb[v]) continue;
    if(sz[v] > qt / 2) return find_cen(v, u, qt);
  }
  return u;
}

void getdist(int u, int p, int cen) {
  for(int v : adj[u]) {
    if(v != p and !forb[v]) {
      dist[cen][v] = dist[v][cen] = dist[cen][u] + 1;
      getdist(v, u, cen);
    }
  }
}

void decomp(int u, int p) {
  dfs(u, -1);

  int cen = find_cen(u, -1, sz[u]);
  forb[cen] = 1;
  par[cen] = p;
  dist[cen][cen] = 0;
  getdist(cen, -1, cen);

  for(int v : adj[cen]) if(!forb[v])
    decomp(v, cen);
}

// main
decomp(1, -1);
```

5.9   Mergesort Tree

```cpp
// Mergesort Tree - Time <O(nlognlogn), O(nlogn)> - Memory O(nlogn)
// Mergesort Tree is a segment tree that stores the sorted subarray
// on each node.
vi st[4*N];

void build(int p, int l, int r) {
  if (l == r) { st[p].pb(s[l]); return; }
  build(2*p, l, (l+r)/2);
  build(2*p+1, (l+r)/2+1, r);
  st[p].resize(r-l+1);
  merge(st[2*p].begin(), st[2*p].end(),
        st[2*p+1].begin(), st[2*p+1].end(),
        st[p].begin());
}

int query(int p, int l, int r, int i, int j, int a, int b) {
  if (j < l or i > r) return 0;
  if (i <= l and j >= r)
    return upper_bound(st[p].begin(), st[p].end(), b) -
           lower_bound(st[p].begin(), st[p].end(), a);
  return query(2*p, l, (l+r)/2, i, j, a, b) +
         query(2*p+1, (l+r)/2+1, r, i, j, a, b);
}
```

5.10   Treap

```cpp
// Treap (probabilistic BST)
// O(logn) operations (supports lazy propagation)

mt19937_64 llrand(random_device{}());

struct node {
  int val;
  int cnt, rev;
  int mn, mx, mindiff; // value-based treap only!
  ll pri;
  node* l;
  node* r;

  node() {}
  node(int x) : val(x), cnt(1), rev(0), mn(x), mx(x), mindiff(INF), pri(llrand()), l(0), r(0) {}
};

struct treap {
```

```cpp
node* root;
treap() : root(0) {}
~treap() { clear(); }

int cnt(node* t) { return t ? t->cnt : 0; }
int mn (node* t) { return t ? t->mn : INF; }
int mx (node* t) { return t ? t->mx : -INF; }
int mindiff(node* t) { return t ? t->mindiff : INF; }

void clear() { del(root); }
void del(node* t) {
  if (!t) return;
  del(t->l); del(t->r);
  delete t;
  t = 0;
}

void push(node* t) {
  if (!t or !t->rev) return;
  swap(t->l, t->r);
  if (t->l) t->l->rev ^= 1;
  if (t->r) t->r->rev ^= 1;
  t->rev = 0;
}

void update(node*& t) {
  if (!t) return;
  t->cnt = cnt(t->l) + cnt(t->r) + 1;
  t->mn = min(t->val, min(mn(t->l), mn(t->r)));
  t->mx = max(t->val, max(mx(t->l), mx(t->r)));
  t->mindiff = min(mn(t->r) - t->val, min(t->val - mx(t->l), min(mindiff(t->l), mindiff(t->r))));
}

node* merge(node* l, node* r) {
  push(l); push(r);
  node* t;
  if (!l or !r) t = l ? l : r;
  else if (l->pri > r->pri) l->r = merge(l->r, r), t = l;
  else r->l = merge(l, r->l), t = r;
  update(t);
  return t;
}

pair<node*, node*> split(node* t, int pos) {
  if (!t) return {0, 0};
  push(t);

  if (cnt(t->l) < pos) {
    auto x = split(t->r, pos-cnt(t->l)-1);
    t->r = x.st;
    update(t);
    return { t, x.nd };
  }

  auto x = split(t->l, pos);
  t->l = x.nd;
  update(t);
  return { x.st, t };
}

// Position-based treap
// used when the values are just additional data
// the positions are known when it's built, after that you
// query to get the values at specific positions
/*
void insert(int pos, int val) {
  push(root);
  node* x = new node(val);
  auto t = split(root, pos);
  root = merge(merge(t.st, x), t.nd);
}

void erase(int pos) {
  auto t1 = split(root, pos);
  auto t2 = split(t1.nd, 1);
  delete t2.st;
  root = merge(t1.st, t2.nd);
}

int get_val(int pos) { return get_val(root, pos); }
int get_val(node* t, int pos) {
  push(t);
  if (cnt(t->l) == pos) return t->val;
  if (cnt(t->l) < pos) return get_val(t->r, pos-cnt(t->l)-1);
  return get_val(t->l, pos);
}
*/
// -------------------
```

```
// Value-based treap
// used when the values needs to be ordered
int order(node* t, int val) {
  if (!t) return 0;
  push(t);
  if (t->val < val) return cnt(t->l) + 1 + order(t->r, val);
  return order(t->l, val);
}

bool has(node* t, int val) {
  if (!t) return 0;
  push(t);
  if (t->val == val) return 1;
  return has((t->val > val ? t->l : t->r), val);
}

void insert(int val) {
  if (has(root, val)) return; // avoid repeated values
  push(root);
  node* x = new node(c);
  auto t = split(root, order(root, val));
  root = merge(merge(t.st, x), t.nd);
}

void erase(int val) {
  if (!has(root, val)) return;

  auto t1 = split(root, order(root, val));
  auto t2 = split(t1.nd, 1);
  delete t2.st;
  root = merge(t1.st, t2.nd);
}

// Get the maximum difference between values
int querymax(int i, int j) {
  if (i == j) return -1;
  auto t1 = split(root, j+1);
  auto t2 = split(t1.st, i);

  int ans = mx(t2.nd) - mn(t2.nd);
  root = merge(merge(t2.st, t2.nd), t1.nd);
  return ans;
}

// Get the minimum difference between values
int querymin(int i, int j) {
  if (i == j) return -1;
  auto t2 = split(root, j+1);
  auto t1 = split(t2.st, i);

  int ans = mindiff(t1.nd);
  root = merge(merge(t1.st, t1.nd), t2.nd);
  return ans;
}
// ------------------

void reverse(int l, int r) {
  auto t2 = split(root, r+1);
  auto t1 = split(t2.st, l);
  t1.nd->rev = 1;
  root = merge(merge(t1.st, t1.nd), t2.nd);
}

void print() { print(root); printf("\n"); }
void print(node* t) {
  if (!t) return;
  push(t);
  print(t->l);
  printf("%d ", t->val);
  print(t->r);
}
};
```

6 Dynamic Programming

6.1 Longest Increasing Subsequence

```
// Longest Increasing Subsequence - O(nlogn)
//
// dp(i) = max j<i { dp(j) | a[j] < a[i] } + 1
//
```

```
int dp[N], v[N], n, lis;

memset(dp, 63, sizeof dp);
for (int i=0; i<n; ++i) {
  // increasing: lower_bound
  // non-decreasing: upper_bound
  int j = lower_bound(dp, dp+lis, v[i]) - dp;
  dp[j] = min(dp[j], v[i]);
  lis = max(lis, j+1);
}
```

6.2 Convex Hull Trick

```
// Convex Hull Trick

// In case of floating point parameters swap long long with long double
typedef long long type;
struct line { type b, m; }

// Order by increasing m if b is equal, otherwise by decreasing b
// If you need the minimum convex hull use { -b, -m }
// [](line s, line t){ return s.b == t.b ? s.m < t.m : s.b > t.b }

// nh: number of lines on convex hull
// pos: position for linear time search
// hull: lines in the convex hull
int nh, pos;
line hull[N];

bool check(line s, line t, line u) {
  return (s.b - t.b)*(u.m - s.m) < (s.b - u.b)*(t.m - s.m);
}

// Add new line to convex hull, if possible
// Must receive lines in the correct order, otherwise it won't work
void update(line s) {
  // 1. if first lines have the same b, get the one with bigger m
  // 2. if line is parallel to the one at the top, ignore
  // 3. pop lines that are worse
  // 3.1 if you can do a linear time search, use
  // 4. add new line

  if (nh == 1 and hull[nh-1].b == s.b) nh--;
  if (nh > 0  and hull[nh-1].m >= s.m) return;
  while (nh >= 2 and !check(hull[nh-2], hull[nh-1], l)) nh--, pos = min(pos, nh);
  hull[nh++] = l;
}

type eval(int id, type x) { return hull[id].b + hull[id].m * x; }

/*
// Linear search query - O(n) for all queries
// Only possible if the queries always move to the right
type query(type x) {
  while (pos+1 < nh and eval(pos, x) > eval(pos+1, x)) pos++;
  return eval(pos, x);
}
*/

// Ternary search query - O(logn) for each query
type query(type x) {
  int lo = 0, hi = nh-1;
  while (lo < hi) {
    int mid = (lo+hi)/2;
    if (eval(mid, x) < eval(mid+1, x)) hi = mid;
    else lo = mid+1;
  }
  return eval(lo, x);
}

// better use geometry line_intersect (this assumes s and t are not parallel)
ld intersect_x(line s, line t) { return (t.b - s.b)/(ld)(s.m - t.m); }
ld intersect_y(line s, line t) { return s.b + s.m * intersect_x(s, t); }
```

6.3 Divide and Conquer Optimization

```
// Divide and Conquer DP Optimization - O(k*n^2) => O(k*n*logn)
//
// dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
```

```
// Condition: A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
//
// reference (pt-br): https://algorithmmarch.wordpress.com/2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-
//     maratona/

int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void calc(int l, int r, int j, int kmin, int kmax) {
  int m = (l+r)/2;
  dp[m][j] = LINF;

  for (int k = kmin; k <= kmax; ++k) {
    ll v = dp[k][j-1] + cost(k, m);

    // store the minimum answer for d[m][j]
    // in case of maximum, use v > dp[m][j]
    if (v < dp[m][j]) a[m][j] = k, dp[m][j] = v;
  }

  if (l < r) {
    calc(l,   m, j, kmin,    a[m][k]);
    calc(m+1, r, j, a[m][k], kmax   );
  }
}

// run for every j
for (int j = 2; j <= maxj; ++j)
  calc(1, n, j, 1, n);
```

6.4 Knuth Optimization

```
// Knuth DP Optimization - O(n^3) -> O(n^2)
//
// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j-1] <= A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
//
// reference (pt-br): https://algorithmmarch.wordpress.com/2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-
//     maratona/
//

// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
int n;
int dp[N][N], a[N][N];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void knuth() {
  // calculate base cases
  memset(dp, 63, sizeof(dp));
  for (int i = 1; i <= n; i++) dp[i][i] = 0;

  // set initial a[i][j]
  for (int i = 1; i <= n; i++) a[i][i] = i;

  for (int j = 2; j <= n; ++j)
    for (int i = j; i >= 1; --i)
      for (int k = a[i][j-1]; k <= a[i+1][j]; ++k) {
        ll v = dp[i][k] + dp[k][j] + cost(i, j);

        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
}

// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
int n, maxj;
int dp[N][J], a[N][J];
```

```
// declare the cost function
int cost(int i, int j) {
  // ...
}

void knuth() {
  // calculate base cases
  memset(dp, 63, sizeof(dp));
  for (int i = 1; i <= n; i++) dp[i][1] = // ...

  // set initial a[i][j]
  for (int i = 1; i <= n; i++) a[i][0] = 0, a[n+1][i] = n;

  for (int j = 2; j <= maxj; j++)
    for (int i = n; i >= 1; i--)
      for (int k = a[i][j-1]; k <= a[i+1][j]; k++) {
        ll v = dp[k][j-1] + cost(k, i);

        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
}
```

7 Geometry

7.1 Basic

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

const int INF = 0x3f3f3f3f;

typedef long double ld;
const double EPS = 1e-9, PI = acos(-1.);

// Change long double to long long if using integers
typedef long double type;

bool ge(type x, type y) { return x + EPS > y; }
bool le(type x, type y) { return x - EPS < y; }
bool eq(type x, type y) { return ge(x, y) and le(x, y); }

struct point {
  type x, y;

  point() : x(0), y(0) {}
  point(type x, type y) : x(x), y(y) {}

  point operator -() { return point(-x, -y); }
  point operator +(point p) { return point(x+p.x, y+p.y); }
  point operator -(point p) { return point(x-p.x, y-p.y); }

  point operator *(type k) { return point(k*x, k*y); }
  point operator /(type k) { return point(x/k, y/k); }

  type operator *(point p) { return x*p.x + y*p.y; }
  type operator %(point p) { return x*p.y - y*p.x; }

  // o is the origin, p is another point
  // dir == +1 => p is clockwise from this
  // dir ==  0 => p is colinear with this
  // dir == -1 => p is counterclockwise from this
  int dir(point o, point p) {
    type x = (*this - o) % (p - o);
    return ge(x,0) - le(x,0);
  }

  bool on_seg(point p, point q) {
    if (this->dir(p, q)) return 0;
    return ge(x, min(p.x, q.x)) and le(x, max(p.x, q.x)) and
           ge(y, min(p.y, q.y)) and le(y, max(p.y, q.y));
  }

  ld abs() { return sqrt(x*x + y*y); }
  type abs2() { return x*x + y*y; }
  ld dist(point x) { return (*this - x).abs(); }
  type dist2(point x) { return (*this - x).abs2(); }

  // Project point on vector y
  point project(point y) { return y * ((*this * y) / (y * y)); }
```

```cpp
// Project point on line generated by points x and y
point project(point x, point y) { return x + (*this - x).project(y-x); }

ld dist_line(point x, point y) { return dist(project(x, y)); }

ld dist_seg(point x, point y) {
    return project(x, y).on_seg(x, y) ? dist_line(x, y) :  min(dist(x), dist(y));
}

point rotate(ld a) {
    return point(cos(a)*x-sin(a)*y, sin(a)*x+cos(a)*y);
}

point rotate(point p) { // rotate around the argument from vector p
    ld hyp = p.abs();
    ld c = p.x / hyp;
    ld s = p.y / hyp;
    return point(c*x-s*y, s*x+c*y);
}
};

int direction(point o, point p, point q) { return p.dir(o, q); }

bool segments_intersect(point p, point q, point a, point b) {
    int d1, d2, d3, d4;
    d1 = direction(p, q, a);
    d2 = direction(p, q, b);
    d3 = direction(a, b, p);
    d4 = direction(a, b, q);
    if (d1*d2 < 0 and d3*d4 < 0) return 1;
    return p.on_seg(a, b) or q.on_seg(a, b) or
            a.on_seg(p, q) or b.on_seg(p, q);
}

point lines_intersect(point p, point q, point a, point b) {
    point r = q-p, s = b-a, c(p%q, a%b);
    if (eq(r%s,0)) return point(INF, INF);
    return point(point(r.x, s.x) % c, point(r.y, s.y) % c) / (r%s);
}

// Sorting points in counterclockwise order.
// If the angle is the same, closer points to the origin come first.
point origin;
bool radial(point p, point q) {
    int dir = p.dir(origin, q);
    return dir > 0 or (!dir and p.on_seg(origin, q));
}

// Graham Scan
vector<point> convex_hull(vector<point> pts) {
    vector<point> ch(pts.size());
    point mn = pts[0];

    for(point p : pts) if (p.y < mn.y or (p.y == mn.y and p.x < p.y)) mn = p;

    origin = mn;
    sort(pts.begin(), pts.end(), radial);

    int n = 0;

    // IF: Convex hull without collinear points
    for(point p : pts) {
        while (n > 1 and ch[n-1].dir(ch[n-2], p) < 1) n--;
        ch[n++] = p;
    }

    /* ELSE IF: Convex hull with collinear points
    for(point p : pts) {
        while (n > 1 and ch[n-1].dir(ch[n-2], p) < 0) n--;
        ch[n++] = p;
    }

    for(int i=pts.size()-1; i >=1; --i)
        if (pts[i] != ch[n-1] and !pts[i].dir(pts[0], ch[n-1]))
            ch[n++] = pts[i];
    // END IF */

    ch.resize(n);
    return ch;
}

// Double of the triangle area
ld double_of_triangle_area(point p1, point p2, point p3) {
    return abs((p2-p1) % (p3-p1));
}

bool point_inside_triangle(point p, point p1, point p2, point p3) {
```

```cpp
    ld a1, a2, a3, a;
    a =  double_of_triangle_area(p1, p2, p3);
    a1 = double_of_triangle_area(p, p2, p3);
    a2 = double_of_triangle_area(p, p1, p3);
    a3 = double_of_triangle_area(p, p1, p2);
    return eq(a, a1 + a2 + a3);
}

bool point_inside_convex_poly(int l, int r, vector<point> v, point p) {
    while(l+1 != r) {
        int m = (l+r)/2;
        if (p.dir(v[0], v[m])) r = m;
        else l = m;
    }
    return point_inside_triangle(p, v[0], v[l], v[r]);
}

vector<point> circle_circle_intersection(point p1, ld r1, point p2, ld r2) {
    vector<point> ret;

    ld d = p1.dist(p2);
    if (d > r1 + r2 or d + min(r1, r2) < max(r1, r2)) return ret;

    ld x = (r1*r1 - r2*r2 + d*d) / (2*d);
    ld y = sqrt(r1*r1 - x*x);

    point v = (p2 - p1)/d;

    ret.push_back(p1 + v * x + v.rotate(PI/2) * y);
    if (y > 0)
        ret.push_back(p1 + v * x - v.rotate(PI/2) * y);

    return ret;
}
```

7.2   Closest Pair of Points

```cpp
//Time complexity: o(nlogn), using merge sort strategy

struct pnt{
    long long x, y;
    pnt operator-(pnt p){ return {x - p.x, y - p.y}; }
    long long operator!(){ return x*x+y*y; }
};

const int N = 1e5 + 5;
pnt pnts[N];
pnt tmp[N];
pnt p1, p2;
unsigned long long d = 9e18;

void closest(int l, int r){
    if(l == r) return;
    int mid = (l + r)/2;

    int midx = pnts[mid].x;
    closest(l, mid), closest(mid + 1, r);

    merge(pnts + l, pnts + mid + 1, pnts + mid + 1, pnts + r + 1, tmp + l,
          [](pnt a, pnt b){ return a.y < b.y; });

    for (int i = l; i <= r; i++) pnts[i] = tmp[i];

    vector<pnt> margin;
    for(int i = l; i <= r; i++)
        if((pnts[i].x - midx)*(pnts[i].x - midx) < d)
            margin.push_back(pnts[i]);

    for(int i = 0; i < margin.size(); i++)
        for(int j = i + 1;
            j < margin.size() and
            (margin[j].y - margin[i].y)*(margin[j].y - margin[i].y) < d;
            j++) {
            if(!(margin[i] - margin[j]) < d)
                p1 = margin[i], p2 = margin[j], d = !(p1 - p2);
        }
}
```

7.3   Nearest Neighbours

```cpp
// Closest Neighbor - O(n * log(n))
const ll N = 1e6+3, INF = 1e18;
ll n, cn[N], x[N], y[N]; // number of points, closes neighbor, x coordinates, y coordinates

ll sqr(ll i) { return i*i; }
ll dist(int i, int j) { return sqr(x[i]-x[j]) + sqr(y[i]-y[j]); }
ll dist(int i) { return i == cn[i] ? INF : dist(i, cn[i]); }

bool cpx(int i, int j) { return x[i] < x[j] or (x[i] == x[j] and y[i] < y[j]); }
bool cpy(int i, int j) { return y[i] < y[j] or (y[i] == y[j] and x[i] < x[j]); }

ll calc(int i, ll x0) {
    ll dlt = dist(i) - sqr(x[i]-x0);
    return dlt >= 0 ? ceil(sqrt(dlt)) : -1;
}

void updt(int i, int j, ll x0, ll &dlt) {
    if (dist(i) > dist(i, j)) cn[i] = j, dlt = calc(i, x0);
}

void cmp(vi &u, vi &v, ll x0) {
    for(int a=0, b=0; a<u.size(); ++a) {
        ll i = u[a], dlt = calc(i, x0);
        while(b < v.size() and y[i] > y[v[b]]) b++;
        for(int j = b-1; j >= 0     and y[i] - dlt <= y[v[j]]; j--) updt(i, v[j], x0, dlt);
        for(int j = b; j < v.size() and y[i] + dlt >= y[v[j]]; j++) updt(i, v[j], x0, dlt);
    }
}

void slv(vi &ix, vi &iy) {
    int n = ix.size();
    if (n == 1) { cn[ix[0]] = ix[0]; return; }

    int m = ix[n/2];

    vi ix1, ix2, iy1, iy2;
    for(int i=0; i<n; ++i) {
        if (cpx(ix[i], m)) ix1.push_back(ix[i]);
        else ix2.push_back(ix[i]);

        if (cpx(iy[i], m)) iy1.push_back(iy[i]);
        else iy2.push_back(iy[i]);
    }

    slv(ix1, iy1);
    slv(ix2, iy2);

    cmp(iy1, iy2, x[m]);
    cmp(iy2, iy1, x[m]);
}

void slv(int n) {
    vi ix, iy;
    ix.resize(n);
    iy.resize(n);
    for(int i=0; i<n; ++i) ix[i] = iy[i] = i;
    sort(ix.begin(), ix.end(), cpx);
    sort(iy.begin(), iy.end(), cpy);
    slv(ix, iy);
}
```

8 Miscellaneous

8.1 builtin

```cpp
__builtin_ctz(x) // trailing zeroes
__builtin_clz(x) // leading zeroes
__builtin_popcount(x) // # bits set
__builtin_ffs(x) // index(LSB) + 1 [0 if x==0]

// Add ll to the end for long long [__builtin_clzll(x)]
```

8.2 prime numbers

```
  2    3    5    7   11   13   17   19   23   29
 31   37   41   43   47   53   59   61   67   71
 73   79   83   89   97  101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
179  181  191  193  197  199  211  223  227  229
233  239  241  251  257  263  269  271  277  281
283  293  307  311  313  317  331  337  347  349
353  359  367  373  379  383  389  397  401  409
419  421  431  433  439  443  449  457  461  463
467  479  487  491  499  503  509  521  523  541
547  557  563  569  571  577  587  593  599  601
607  613  617  619  631  641  643  647  653  659
661  673  677  683  691  701  709  719  727  733
739  743  751  757  761  769  773  787  797  809
811  821  823  827  829  839  853  857  859  863
877  881  883  887  907  911  919  929  937  941
947  953  967  971  977  983  991  997 1009 1013
1019 1021 1031 1033 1039 1049 1051 1061 1063 1069
1087 1091 1093 1097 1103 1109 1117 1123 1129 1151
1153 1163 1171 1181 1187 1193 1201 1213 1217 1223
1229 1231 1237 1249 1259 1277 1279 1283 1289 1291
1297 1301 1303 1307 1319 1321 1327 1361 1367 1373
1381 1399 1409 1423 1427 1429 1433 1439 1447 1451
1453 1459 1471 1481 1483 1487 1489 1493 1499 1511
1523 1531 1543 1549 1553 1559 1567 1571 1579 1583
1597 1601 1607 1609 1613 1619 1621 1627 1637 1657
1663 1667 1669 1693 1697 1699 1709 1721 1723 1733
1741 1747 1753 1759 1777 1783 1787 1789 1801 1811
1823 1831 1847 1861 1867 1871 1873 1877 1879 1889
1901 1907 1913 1931 1933 1949 1951 1973 1979 1987

        970'997        971'483   921'281'269   999'279'733
1'000'000'009 1'000'000'021 1'000'000'409 1'005'012'527
```

8.3 Week day

```cpp
int v[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
int day(int d, int m, int y) {
    y -= m<3;
    return (y + y/4 - y/100 + y/400 + v[m-1] + d)%7;
}
```

8.4 Date

```cpp
struct Date {
    int d, m, y;
    static int mnt[], mntsum[];

    Date() : d(1), m(1), y(1) {}
    Date(int d, int m, int y) : d(d), m(m), y(y) {}
    Date(int days) : d(1), m(1), y(1) { advance(days); }

    bool bissexto() { return (y%4 == 0 and y%100) or (y%400 == 0); }

    int mdays() { return mnt[m] + (m == 2)*bissexto(); }
    int ydays() { return 365+bissexto(); }

    int msum() { return mntsum[m-1] + (m > 2)*bissexto(); }
    int ysum() { return 365*(y-1) + (y-1)/4 - (y-1)/100 + (y-1)/400; }

    int count() { return (d-1) + msum() + ysum(); }

    int day() {
        int x = y - (m<3);
        return (x + x/4 - x/100 + x/400 + mntsum[m-1] + d + 6)%7;
    }

    void advance(int days) {
        days += count();
        d = m = 1, y = 1 + days/366;
        days -= count();
        while(days >= ydays()) days -= ydays(), y++;
        while(days >= mdays()) days -= mdays(), m++;
        d += days;
    }
};

int Date::mnt[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int Date::mntsum[13] = {};
for(int i=1; i<13; ++i) Date::mntsum[i] = Date::mntsum[i-1] + Date::mnt[i];
```

8.5   Python

```python
# reopen
import sys
sys.stdout = open('out','w')
sys.stdin  = open('in' ,'r')
```

8.6   Sqrt Decomposition

```cpp
// Square Root Decomposition (Mo's Algorithm) - O(n^(3/2))
const int N = 1e5+1, SQ = 500;
int n, m, v[N];

void add(int p) { /* add value to aggregated data structure */ }
void rem(int p) { /* remove value from aggregated data structure */ }
```

```cpp
struct query { int i, l, r, ans; } qs[N];

bool c1(query a, query b) { return a.l/SQ == b.l/SQ ? a.r < b.r : a.l < b.l; }
bool c2(query a, query b) { return a.i < b.i; }

/* inside main */
int l = 0, r = -1;
sort(qs, qs+m, c1);
for (int i = 0; i < m; ++i) {
    query &q = qs[i];
    while (r < q.r) add(v[++r]);
    while (r > q.r) rem(v[r--]);
    while (l < q.l) rem(v[l++]);
    while (l > q.l) add(v[--l]);

    q.ans = /* calculate answer */;
}

sort(qs, qs+m, c2); // sort to original order
```

	A	C	N	Assunto	Descricao	Diff
A						
B						
C						
D						
E						
F						
G						
H						
I						
J						
K						
L						