.

Base Julios GOD

Contents

1   Data structures

1.1   Segment tree

```
1  #define M ((l + r) >> 1)
2  #define op(x, y) (x + y)
3  int st[4*N], values[N];
4  void build(int l, int r, int i) {
5      if(l == r) {
6          st[i] = values[l];
7          return;
8      }
9      build(l, M, 2*i+1);
10     build(M+1, r, 2*i+2);
11     st[i] = op(st[2*i+1], st[2*i+2]);  // #!
12 }
13 // Point update - range query
14 void update(int l, int r, int idx, int x, int i) {
15     if(l == r) {
16         st[i] += x;
17         return;
18     }
19     if(idx <= M) update(l, M, idx, x, 2*i+1);
20     else update(M+1, r, idx, x, 2*i+2);
21     st[i] = op(st[2*i+1], st[2*i+2]);
22 }
23 int query(int l, int r, int a, int b, int i) {
24     if(a > r || b < l) return 0;        // change for min/max
25     if(a <= l && r <= b) return st[i];
26     return op(query(l, M, a, b, 2*i+1), query(M+1, r, a, b, 2*i+2));
27 }
28 // l = 0, r = n-1, i = 0
```

1.2   Segment tree - Range update, point query

```
1  // Range update - point query
2  // same as classic segment tree, but comment this line #!
3  void update(int l, int r, int a, int b, int x, int i) {
4      if (a > r || b < l) return;
5      if (a <= l && r <= b) {
6          st[i] += x;
7          return;
8      }
9      update(l, M, a, b, x, 2*i+1);
10     update(M+1, r, a, b, x, 2*i+2);
11 }
12 int query(int l, int r, int idx, int i) {
13     if(idx > r || idx < l) return 0;        // change for min/max
14     if(idx <= l && r <= idx) return st[i];
15     return op(op(query(l, M, idx, 2*i+1), query(M+1, r, idx, 2*i+2)), st[i]);
16 }
```

1.3   Segment tree - Lazy propagation

```
1  #define M ((l + r) >> 1)
2  #define op(x, y) (x + y)
3  ll st[4*N], lazy[4*N], arr[N];
4  void build(int l, int r, int i) {
5      lazy[i] = 0;
6      if (l == r) {st[i] = arr[l]; return;}
7      build(l, M, 2*i+1);
8      build(M+1, r, 2*i+2);
9      st[i] = op(st[2*i+1], st[2*i+2]);
10 }
11 void push(int l, int r, int i) {
12     if (!lazy[i]) return;
13     st[i] += (r-l+1) * lazy[i];   // change for min/max
14     if (l != r) {
15         lazy[2*i+1] += lazy[i];
16         lazy[2*i+2] += lazy[i];
17     }
18     lazy[i] = 0;   // change for multiplication
19 }
20 void update(int l, int r, int a, int b, ll x, int i) {
21     push(l, r, i);
22     if (a > r || b < l) return;
23     if (a <= l && r <= b) {
24         lazy[i] += x;
25         push(l, r, i);
26         return;
27     }
28     update(l, M, a, b, x, 2*i+1);
29     update(M+1, r, a, b, x, 2*i+2);
30     st[i] = op(st[2*i+1], st[2*i+2]);
31 }
```

```
32  ll query(int l, int r, int a, int b, int i) {
33      if (a > r || b < l) return 0;    // change for min/max
34      push(l, r, i);
35      if (a <= l && r <= b) return st[i];
36      return op(query(l, M, a, b, 2*i+1), query(M+1, r, a, b, 2*i+2));
37  }  // i=0, l=0, r=n-1, x=value, a,b=range query
```

1.4 Segment tree - Persistence

```
1   #define M ((l + r) >> 1)
2   struct Node{
3       Node *left, *right;
4       ll val;
5       Node(ll x) : left(NULL), right(NULL), val(x) {}        // Add value
6       Node(Node *l, Node *r) : left(l), right(r), val(0) {    // Update
            values
7           if(l) val += l->val;
8           if(r) val += r->val;
9       }
10      Node(Node *root) : left(root->left), right(root->right), val(root->
            val) {}    // Make copy
11  };
12  Node *build(int l, int r, vector<int> &values) {
13      if(l == r) return new Node(values[l]);
14      return new Node(build(l, M, values), build(M+1, r, values));
15  }
16  Node *update(Node *node, int l, int r, int idx, ll k) {
17      if(l == r) return new Node(k);
18      if(idx <= M) return new Node(update(node->left, l, M, idx, k), node
            ->right);
19      return new Node(node->left, update(node->right, M+1, r, idx, k));
20  }
21  ll query(Node *node, int l, int r, int a, int b) {
22      if(a > r || b < l) return 0;
23      if(a <= l && r <= b) return node->val;
24      return query(node->left, l, M, a, b) + query(node->right, M+1, r, a,
            b);
25  }
26  Node *roots[N];
27  // 0 based indexing
28  // roots[copy++] = build(0, n-1);
29  // roots[copy++] = new Node(roots[--idx]);
30  // roots[--copy] = update(roots[--copy], 0, n-1, --idx, x);
31  // query(roots[--copy], 0, n-1, --a, --b)
```

1.5 Fenwick tree

```
1   struct FT{
2       int n;
3       vector<int> ft;
4       FT(int _n) : n(_n), ft(_n+1) {}
5       void add(int idx, int k) {
6           for(; idx<=n; idx+=idx&-idx)
7               ft[idx] += k;
8       }
9       int query(int idx) {
10          int sum = 0;
11          for(; idx>0; idx-=idx&-idx)
12              sum += ft[idx];
13          return sum;
14      }
15      int query(int l, int r) {
16          return query(r) - query(l-1);
17      }
18      int lower_bound(int k) { // LOG = log2(n) + 1
19          int sum = 0, idx = 0;
20          for(int i=LOG-1; i>=0; i--) {
21              if(idx + (1 << i) <= n && sum + ft[idx + (1 << i)] < k) {
22                  sum += ft[idx + (1 << i)];
23                  idx += 1 << i;
24              }
25          }
26          return idx + 1;
27      }
28  }; // 1-based indexing
```

1.6 DSU

```
1   struct DSU{
2       int n;
3       vector<int> parent, rank;
4       DSU(int _n) : n(_n), parent(_n), rank(_n) {
5           for(int i=0; i<n; i++) {
6               parent[i] = i;
7               rank[i] = 0;
8           }
9       }
```

```
10      int find_set(int v) {
11          if (v == parent[v]) return v;
12          return parent[v] = find_set(parent[v]);
13      }
14      void union_sets(int a, int b) {
15          a = find_set(a), b = find_set(b);
16          if (a != b) {
17              if (rank[a] < rank[b]) swap(a, b);
18              parent[b] = a;
19              if (rank[a] == rank[b]) rank[a]++;
20          }
21      }
22      int components() {
23          int cnt = 0;
24          for(int i=0; i<n; i++) {
25              if(find_set(i) == i) cnt++;
26          }
27          return cnt;
28      }
29  };
```

1.7   SQRT decomposition

```
1  struct SQ{
2      int n, b;
3      vector<int> values, blocks;
4      SQ(int _n) : n(_n), values(_n) {
5          b = (int)sqrt(n) + 1;
6          blocks = vector<int> (b);
7      }
8      // Basic update / query
9      void operation(int l, int r, int k) {
10          int bl = l/b, br = r/b;
11          // operation lies in same block
12          if(bl == br) {
13              for(int i=l; i<=r; i++) {
14                  blocks[bl] -= values[i];
15                  values[i] += k;
16                  blocks[bl] += values[i];
17              }
18          }
19          // operation on different blocks
20          else {
```

```
21              for(int i=l; i<(bl+1)*b; i++) {
22                  blocks[bl] -= values[i];
23                  values[i] += k;
24                  blocks[bl] += values[i];
25              }
26              for(int i=br*b; i<=r; i++) {
27                  blocks[br] -= values[i];
28                  values[i] += k;
29                  blocks[br] += values[i];
30              }
31              for(int i=(bl+1)*b; i<br; i++) {
32                  blocks[i] += k * b;
33              }
34          }
35      }
36  };
```

1.8   Trie

```
1  struct Node{
2      vector<Node*> ocu;
3      bool flag;
4      Node() : ocu(26), flag(false) {}
5  };
6  struct Trie{
7      Node *root;
8      Trie() : root(new Node()) {}
9      void insert(string word) {
10          Node *curr = root;
11          for(auto &it : word) {
12              if(!curr -> ocu[it - 'a'])
13                  curr -> ocu[it - 'a'] = new Node();
14              curr = curr -> ocu[it - 'a'];
15          }
16          curr -> flag = true;
17      }
18      bool search(string word) {
19          Node *curr = root;
20          for(auto &it : word) {
21              if(!curr -> ocu[it - 'a']) return false;
22              curr = curr -> ocu[it - 'a'];
23          }
24          return curr -> flag;
```

```
25          }
26  };
```

1.9   Trie XOR

```
1   struct Node{
2       vector<Node*> ocu;
3       Node() : ocu(2) {}
4   };
5   struct Trie {
6       Node *root;
7       Trie() : root(new Node()) { insert(0); }
8
9       void insert(long long x) {
10          Node *curr = root;
11          for(int mask=63; mask>=0; mask--) {
12              bool currBit = (x >> mask) & 1;
13              if(!curr -> ocu[currBit])
14                  curr -> ocu[currBit] = new Node();
15              curr = curr -> ocu[currBit];
16          }
17      }
18      long long query(long long prefix) {
19          Node *curr = root;
20          long long res = 0;
21          for(int mask=63; mask>=0; mask--) {
22              bool currBit = (prefix >> mask) & 1;
23              if(curr -> ocu[currBit ^ 1]) {
24                  res |= (1LL << mask);
25                  curr = curr -> ocu[currBit ^ 1];
26              }
27              else curr = curr -> ocu[currBit];
28          }
29          return res;
30      }
31  };
```

1.10   Map custom hash

```
1   #include <ext/pb_ds/assoc_container.hpp>
2   using namespace __gnu_pbds;
3
4   struct custom_hash {
5       static uint64_t splitmix64(uint64_t x) {
```

```
6           x += 0x9e3779b97f4a7c15;
7           x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
8           x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
9           return x ^ (x >> 31);
10      }
11      size_t operator()(uint64_t x) const {
12          static const uint64_t FIXED_RANDOM = chrono::steady_clock::now()
                    .time_since_epoch().count();
13          return splitmix64(x + FIXED_RANDOM);
14      }
15  };
16  gp_hash_table<int, int, custom_hash> freq;
```

1.11   Ordered set

```
1   #include<ext/pb_ds/assoc_container.hpp>
2   #include<ext/pb_ds/tree_policy.hpp>
3   using namespace __gnu_pbds;
4   typedef tree<int,null_type,less<int>,rb_tree_tag,
        tree_order_statistics_node_update> ordered_set;
5   // find_by_order(i) -> iterator to ith element
6   // order_of_key(k) -> position (int) of lower_bound of k
```

1.12   Merge Sort Tree

```
1   #define M ((l + r) >> 1)
2   struct ST{
3       int n;
4       vector<vector<int>> st;
5       ST(int _n) : n(_n), st(4*_n) {}
6
7       void build(int l, int r, int i, vector<int> &values) {
8           if(l == r) {
9               st[i].pb(values[l]);
10              return;
11          }
12          build(l, M, 2*i+1, values);
13          build(M+1, r, 2*i+2, values);
14          st[i].resize(st[2*i+1].size() + st[2*i+2].size());
15          merge(st[2*i+1].begin(), st[2*i+1].end(), st[2*i+2].begin(), st
                    [2*i+2].end(), st[i].begin());
16      }
17      int query(int l, int r, int a, int b, int k, int i) {
18          if(a > r || b < l) return 0;
```

```
19        if(a <= l && r <= b) return st[i].end() - upper_bound(st[i].
              begin(), st[i].end(), k);
20        return query(l, M, a, b, k, 2*i+1) + query(M+1, r, a, b, k, 2*i
              +2);
21    }
22 };
```

1.13 Wavelet Tree

```
1  // indexed in 1
2  // from pointer to first element and to to end
3  // x and y The minimun element and y the max element
4  // If you need only one function or more erase the others
5  // If you need tu construct other function you only required to
       understand the limit, this
6  // are the same
7  struct wavelet_tree{
8    int lo, hi;
9    wavelet_tree *l, *r;
10   vector<int> b;
11   wavelet_tree(int *from, int *to, int x, int y){
12     lo = x, hi = y;
13     if(lo == hi or from >= to) return;
14     int mid = (lo+hi)/2;
15     auto f = [mid](int x){ return x <= mid;};
16     b.reserve(to-from+1);
17     b.pb(0);
18     for(auto it = from; it != to; it++)
19       b.push_back(b.back() + f(*it));
20     auto pivot = stable_partition(from, to, f);
21     l = new wavelet_tree(from, pivot, lo, mid);
22     r = new wavelet_tree(pivot, to, mid+1, hi);
23   }
24   //kth smallest element in [l, r]
25   int kth(int l, int r, int k){
26     if(l > r) return 0;
27     if(lo == hi) return lo;
28     int inLeft = b[r] - b[l-1];
29     int lb = b[l-1];
30     int rb = b[r];
31     if(k <= inLeft) return this->l->kth(lb+1, rb , k);
32     return this->r->kth(l-lb, r-rb, k-inLeft);
33   }
```

```
34   //count of nos in [l, r] Less than or equal to k
35   int LTE(int l, int r, int k) {
36     if(l > r or k < lo) return 0;
37     if(hi <= k) return r - l + 1;
38     int lb = b[l-1], rb = b[r];
39     return this->l->LTE(lb+1, rb, k) + this->r->LTE(l-lb, r-rb, k);
40   }
41   //count of nos in [l, r] equal to k
42   int count(int l, int r, int k) {
43     if(l > r or k < lo or k > hi) return 0;
44     if(lo == hi) return r - l + 1;
45     int lb = b[l-1], rb = b[r], mid = (lo+hi)/2;
46     if(k <= mid) return this->l->count(lb+1, rb, k);
47     return this->r->count(l-lb, r-rb, k);
48   }
49 };
```

1.14 Monotonic stack

```
1  int main() {
2      ios::sync_with_stdio(0); cin.tie(0);
3      int n; cin >> n;
4      vector<int> v(n), l(n), r(n);
5      for(auto &it : v) cin >> it;
6
7      stack<pair<int, int>> st;
8      for(int i=n-1; i>=0; i--) {
9          while(!st.empty() && v[i] < st.top().first) {
10             l[st.top().second] = i;
11             st.pop();
12         }
13         st.push({v[i], i});
14     }
15
16     while(!st.empty()) l[st.top().second] = -1, st.pop();
17
18     rep(i,0,n) {
19         while(!st.empty() && v[i] <= st.top().first) {
20             r[st.top().second] = i;
21             st.pop();
22         }
23         st.push({v[i], i});
24     }
```

```
25
26      while(!st.empty()) r[st.top().second] = n, st.pop();
27
28      vector<int> res(n);
29      rep(i,0,n) {
30          int curr = r[i] - l[i] - 1;
31          res[curr-1] = max(res[curr-1], v[i]);
32      }
33      for(int i=n-2; i>=0; i--) {
34          res[i] = max(res[i], res[i+1]);
35      }
36
37      for(auto &it : res) cout << it << "␣";
38      cout << endl;
39
40      return 0;
41  }
```

1.15   Matrix operations

```
1   struct Matrix {
2     ll a[N][N];
3     Matrix() {memset(a,0,sizeof(a));}
4     Matrix operator *(Matrix other) {  // Product of a matrix
5       Matrix product=Matrix();
6           rep(i,0,N) rep(j,0,N) rep(k,0,N) {
7               product.a[i][k]+=a[i][j]*other.a[j][k];
8               product.a[i][k]%=MOD;
9           }
10      return product;
11    }
12  };
13  Matrix expo_power(Matrix a, ll n) {  // Matrix exponentiation
14    Matrix res=Matrix();
15      rep(i,0,N) res.a[i][i]=1;  // Matriz identidad
16    while(n){  // binpow
17          if(n & 1) res = res * a;
18          n >>= 1;
19          a = a * a;
20    }
21    return res;
22  } // Ej. Matrix M=Matrix();  M.a[0][0]=1;  M=M*M;   Matrix res=
        expo_power(M,k);
```

1.16   Heavy Light Decomposition

```
1   int sz[N], in[N], out[N], timer, head[N], parent[N];
2   void dfs_sz(int u = 0, int p = -1) {
3       sz[u] = 1;
4       parent[u] = p;
5       for(auto &v : adj[u]) {
6           if(v == p) continue;
7           dfs_sz(v, u);
8           sz[u] += sz[v];
9           if(adj[u][0] == p || sz[v] > sz[adj[u][0]]) {
10              swap(v, adj[u][0]);
11          }
12      }
13  }
14  void dfs_hld(int u = 0, int p = -1) {
15      in[u] = timer++;
16      for(auto &v : adj[u]) {
17          if(v == p) continue;
18          head[v] = (adj[u][0] == v ? head[u] : v);
19          dfs_hld(v, u);
20      }
21      out[u] = timer;
22  }
23  ll query_path(int u, int v) {
24      ll res = 0;
25      while(head[u] != head[v]) {
26          if(in[u] < in[v]) swap(u, v);
27          res += query(0, timer-1, in[head[u]], in[u], 0);
28          u = parent[head[u]];
29      }
30      if(in[u] > in[v]) swap(u, v);
31      res += query(0, timer-1, in[u]+1, in[v], 0);
32      // in[u] (node)  / in[u]+1 (edge)
33      return res;
34  }
```

1.17   Convex Hull Trick

```
1   typedef ll tc;
2   struct Line{tc m,h;};
3   struct CHT { // for minimum (for maximum just change the sign of lines)
4     vector<Line> c;
```

```
5    int pos=0;
6    tc in(Line a, Line b){
7      tc x=b.h-a.h,y=a.m-b.m;
8      return x/y+(x%y?!((x>0)^(y>0)):0); // ==ceil(x/y)
9    }
10   void add(tc m, tc h){ // m's should be non increasing
11     Line l=(Line){m,h};
12     if(c.size()&&m==c.back().m){
13       l.h=min(h,c.back().h);c.pop_back();if(pos)pos--;
14     }
15     while(c.size()>1&&in(c.back(),l)<=in(c[c.size()-2],c.back())){
16       c.pop_back();if(pos)pos--;
17     }
18     c.pb(l);
19   }
20   inline bool fbin(tc x, int m){return in(c[m],c[m+1])>x;}
21   tc query(tc x){
22     // O(log n) query:
23     int s=0,e=c.size();
24     while(e-s>1){int m=(s+e)/2;
25       if(fbin(x,m-1))e=m;
26       else s=m;
27     }
28     return c[s].m*x+c[s].h;
29     // O(1) query (for ordered x's):
30     while(pos>0&&fbin(x,pos-1))pos--;
31     while(pos<c.size()-1&&!fbin(x,pos))pos++;
32     return c[pos].m*x+c[pos].h;
33   }
34 }; // y=mx+b -> cht.add(m, b)   |   cht.query(x)
```

1.18   Dynamic Convex Hull Trick

```
1  using ll = long long;
2  struct Line {
3    mutable ll m, b, p;
4    bool operator<(const Line& o) const { return m < o.m; }
5    bool operator<(ll x) const { return p < x; }
6  };
7  struct LineContainer : multiset<Line, less<>> {
8    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
9    static const ll inf = LLONG_MAX;
10   ll div(ll a, ll b) { // floored division
```

```
11     return a / b - ((a ^ b) < 0 && a % b); }
12   bool isect(iterator x, iterator y) {
13     if (y == end()) return x->p = inf, 0;
14     if (x->m == y->m) x->p = x->b > y->b ? inf : -inf;
15     else x->p = div(y->b - x->b, x->m - y->m);
16     return x->p >= y->p;
17   }
18   void add(ll m, ll b) {
19     auto z = insert({m, b, 0}), y = z++, x = y;
20     while (isect(y, z)) z = erase(z);
21     if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
22     while ((y = x) != begin() && (--x)->p >= y->p)
23       isect(x, erase(y));
24   }
25   ll query(ll x) {
26     assert(!empty());
27     auto l = *lower_bound(x);
28     return l.m * x + l.b;
29   }
30 }; // y=mx+b -> cht.add(m, b)   |   cht.query(x)
```

2   Graphs

2.1   Euler tour

```
1  vector<int> in(n+1), out(n+1), euler(n+1);
2  int timer = 0;
3  auto dfs = [&](auto self, int u, int p = -1) -> void {
4      in[u] = ++timer;
5      euler[timer] = u;
6      for(auto &v : adj[u]) {
7          if(v == p) continue;
8          self(self, v, u);
9      }
10     out[u] = timer;
11 };
12 dfs(dfs, 0);
13 queries[i] = {in[u], out[u], k, i};
14 int x = values[euler[idx]];
```

2.2   Find bridges

```
1  int in[N], low[N], timer;
```

```
2   /* Articulation bridges */
3   void dfs(int u, int p = -1) {
4       in[u] = low[u] = ++timer;
5       for(auto &v : adj[u]) {
6           if(v == p) continue;
7           if(!in[v]) {
8               dfs(v, u);
9               low[u] = min(low[u], low[v]);
10              if(low[v] > in[u]) IS_BRIDGE(U, V);
11          }
12          else low[u] = min(low[u], in[v]);
13      }
14  }
15
16  /* Bridge tree */
17  // Use dfs to find bridges, but change
18  if(low[v] <= in[u]) dsu.union_sets(u, v);
19
20  // Iterate over the edges
21  for(auto [u, v] : edges) {
22      u = dsu.find_set(u);
23      v = dsu.find_set(v);
24      if(u != v) {
25          bridgeTree[u].pb(v);
26          bridgeTree[v].pb(u);
27      }
28  }
29
30  /* Articulation points */
31  // Use same code as finding bridges
32  if(low[v] >= in[u]) IS_ARTICULATION_POINT(U, V);
```

2.3   Dijkstra

```
1   void dijkstra(int start, int n) {
2       priority_queue<pair<ll, int>, vector<pair<ll,int>>, greater<pair<ll,
            int>>> pq;
3       for(int i=0; i<n; i++) {
4           dist[i] = 1e14;
5       }
6
7       dist[start] = 0;
8       pq.push({0, start});
```

```
9
10      while(!pq.empty()) {
11          auto [d, u] = pq.top();
12          pq.pop();
13
14          if(d > dist[u]) continue;
15
16          for(auto &[v, w] : adj[u]) {
17              if(dist[u] + w < dist[v]) {
18                  dist[v] = dist[u] + w;
19                  pq.push({dist[v], v});
20              }
21          }
22      }
23  }
```

2.4   Binary Lifting LCA

```
1   int depth[N], up[N][LOG];
2   void dfs1(int u, int p) {
3       up[u][0] = p;
4       rep(i,1,LOG) {
5           if(up[u][i-1] != -1)
6               up[u][i] = up[up[u][i-1]][i-1];
7           else
8               up[u][i] = -1;
9       }
10      for(auto &v : adj[u]) {
11          if(v == p) continue;
12          depth[v] = depth[u] + 1;
13          dfs1(v, u);
14      }
15  }
16  int lift(int u, int k) {
17      for(int i=LOG-1; i>=0; i--) {
18          if(k & (1 << i)) {
19              u = up[u][i];
20              if(u == -1) return 0;
21          }
22      }
23      return u;
24  }
25  int lca(int u, int v) {
```

```
26      if(depth[u] < depth[v]) swap(u, v);
27      u = lift(u, depth[u] - depth[v]);
28      if(u == v) return u;
29
30      for(int i=LOG-1; i>=0; i--) {
31          if(up[u][i] != -1 && up[u][i] != up[v][i]) {
32              u = up[u][i];
33              v = up[v][i];
34          }
35      }
36      return up[u][0] == -1 ? 0 : up[u][0];
37  }
```

2.5   Centroid decomposition

```
1   void dfs(int u, int p) {
2       sz[u] = 1;
3       for(auto &v : adj[u]) {
4           if(v == p || centroid[v]) continue;
5           dfs(v, u);
6           sz[u] += sz[v];
7       }
8   }
9   int getCentroid(int u, int p, int n) {
10      for(auto &v : adj[u]) {
11          if(v == p || centroid[v]) continue;
12          if(2 * sz[v] > n) return getCentroid(v, u, n);
13      }
14      return u;
15  }
16  void getCount(int u, int p, int depth) {
17      if(depth > k) return;
18      res += 0ll + cnt[k - depth];
19      for(auto &v : adj[u]) {
20          if(v == p || centroid[v]) continue;
21          getCount(v, u, depth+1);
22      }
23  }
24  void getPaths(int u, int p, int depth, int x) {
25      if(depth > k) return;
26      cnt[depth] += x;
27      for(auto &v : adj[u]) {
28          if(v == p || centroid[v]) continue;
```

```
29          getPaths(v, u, depth+1, x);
30      }
31  }
32  void decompose(int u, int p) {
33      dfs(u, p);
34      int c = getCentroid(u, p, sz[u]);
35      centroid[c] = 1;
36      cnt[0] = 1;
37
38      for(auto &v : adj[c]) {
39          if(centroid[v]) continue;
40          getCount(v, c, 1);
41          getPaths(v, c, 1, 1);
42      }
43
44      getPaths(c, p, 0, -1);
45
46      for(auto &v : adj[c]) {
47          if(v == c || centroid[v]) continue;
48          decompose(v, c);
49      }
50  }
```

2.6   Edmonds karp

```
1   const int N = 1e3+5;
2   vector<int> adj[N];
3   int parent[N], cap[N][N];
4
5   void add_edge(int u, int v, int c) {
6       adj[u].pb(v);
7       adj[v].pb(u);
8       cap[u][v] += c;
9   }
10  int bfs(int s, int t) {
11      memset(parent, -1, sizeof(parent));
12      parent[s] = -2;
13      queue<pair<int, int>> q;
14      q.push({s, (int)1e9});
15
16      while(!q.empty()) {
17          auto [u, f] = q.front();
18          q.pop();
```

```
19
20          for(auto &v : adj[u]) {
21              if(parent[v] != -1 || !cap[u][v]) continue;
22
23              parent[v] = u;
24              int flow = min(cap[u][v], f);
25              if(v == t) return flow;
26
27              q.push({v, flow});
28          }
29      }
30      return 0;
31  }
32  int maxFlow(int s, int t) {
33      int mx = 0, flow;
34      while(flow = bfs(s, t)) {
35          mx += flow;
36          int curr = t;
37          while(curr != s) {
38              int prev = parent[curr];
39              cap[prev][curr] -= flow;
40              cap[curr][prev] += flow;
41              curr = prev;
42          }
43      }
44      return mx;
45  }
```

2.7   Dinics

```
1   struct Dinic {
2       int nodes, src, dst;
3       vector<int> dist, q, work;
4       struct edge{int to, rev; ll f, cap;};
5       vector<vector<edge>> g;
6       Dinic(int x) : nodes(x), g(x), dist(x), q(x), work(x){}
7       void add_edge(int s, int t, ll cap) {
8           g[s].pb((edge) {t, SZ(g[t]), 0, cap});
9           g[t].pb((edge) {s, SZ(g[s])-1, 0, 0});
10      }
11      bool dinic_bfs() {
12          fill(all(dist), -1); dist[src]=0;
13          int qt=0; q[qt++]=src;
14          for(int qh=0; qh<qt; qh++) {
15              int u=q[qh];
16              rep(i, 0, SZ(g[u])) {
17                  edge &e = g[u][i]; int v=g[u][i].to;
18                  if(dist[v]<0&&e.f<e.cap) dist[v]=dist[u]+1, q[qt++]=v;
19              }
20          }
21          return dist[dst]>=0;
22      }
23      ll dinic_dfs(int u, ll f) {
24          if(u==dst) return f;
25          for(int &i=work[u]; i<SZ(g[u]); i++) {
26              edge &e=g[u][i];
27              if(e.cap <= e.f) continue;
28              int v=e.to;
29              if(dist[v]==dist[u]+1) {
30                  ll df=dinic_dfs(v, min(f, e.cap-e.f));
31                  if(df > 0) {e.f+=df; g[v][e.rev].f-=df; return df;}
32              }
33          }
34          return 0;
35      }
36      ll max_flow(int _src, int _dst) {
37          src=_src, dst=_dst;
38          ll result=0;
39          while(dinic_bfs()) {
40              fill(all(work), 0);
41              while(ll delta=dinic_dfs(src, 1e12)) result+=delta;
42          }
43          return result;
44      }
45  };
46
47  int main() {io
48      int n,m,sl,el,s,t,u,v;
49      cin>>n>>sl>>el;
50      s=n+n, t=s+1;
51      Dinic nf(n+n+2);
52      rep(i,0,sl){
53          cin>>u; u--;
54          nf.add_edge(s,u,1);
55      }
56      rep(i,0,el){
```

```
57      cin>>u; u--;
58      nf.add_edge(u+n,t,1);
59    }
60    rep(i,0,n) nf.add_edge(i,i+n,1);
61    cin>>m;
62    rep(i,0,m){
63      int u,v;
64      cin>>u>>v;
65      u--,v--;
66      nf.add_edge(u+n,v,1);
67    }
68   cout<<nf.max_flow(s,t)<<endl;
69    return 0;
70 }
```

2.8   Bipartite check

```
1  int color[N];
2  bool f = 1;
3  // memset(color, -1, sizeof(color));
4  bool dfs(int u = 0, int c = 0) {
5      color[u] = c;
6      for(auto &v : adj[u]) {
7          if(color[v] == -1 && !dfs(v, 1-c)) return false;
8          else if(color[v] == c) return false;
9      }
10 }
```

2.9   Has cycle?

```
1  int color[N];
2  // directed graph
3  bool dfs(int u, int p = -1) {
4      color[u] = 1;
5      for(auto &v : adj[u]) {
6          if(!color[v]) if(!dfs(v, u)) return false;
7          if(color[v] == 1) return false;
8      }
9      color[u] = 2;
10     return true;
11 }
12
13 // undirected graph
14 bool dfs(int u, int p = -1) {
```

```
15      color[u] = 1;
16      parent[u] = p;
17      for(auto &v : adj[u]) {
18          if(v == p) continue;
19          if(!color[v]) {
20              if(dfs(v, u)) return true;
21          }
22          else {
23              t = u, s = v;
24              return true;
25          }
26      }
27      return false;
28 }
```

2.10   Kuhn - matching

```
1  vector<int> adj[N]; // [0,n)->[0,m)
2  int n, m, mat[M];
3  bool vis[N];
4  int match(int x){
5    if(vis[x]) return 0;
6    vis[x] = 1;
7    for(auto y : adj[x]) if(mat[y]<0 || match(mat[y])) {mat[y]=x; return
        1;}
8    return 0;
9  }
10 vector<pair<int,int>> max_matching(){
11   vector<pair<int, int>> r;
12   memset(mat, -1, sizeof(mat));
13   for(int i=0; i<n; i++) memset(vis, false, sizeof(vis)), match(i);
14   for(int i=0; i<m; i++) if(mat[i] >= 0) r.pb({mat[i], i});
15   return r;
16 }
```

2.11   Hopcroft Karp - matching

```
1  vector<int> g[MAXN]; // [0,n)->[0,m)
2  int n,m;
3  int mt[MAXN],mt2[MAXN],ds[MAXN];
4  bool bfs(){
5    queue<int> q;
6    memset(ds,-1,sizeof(ds));
7    rep(i,0,n)if(mt2[i]<0)ds[i]=0,q.push(i);
```

```
8     bool r=false;
9     while(!q.empty()){
10      int x=q.front();q.pop();
11      for(int y:g[x]){
12        if(mt[y]>=0&&ds[mt[y]]<0)ds[mt[y]]=ds[x]+1,q.push(mt[y]);
13        else if(mt[y]<0)r=true;
14      }
15    }
16    return r;
17  }
18  bool dfs(int x){
19    for(int y:g[x])if(mt[y]<0||ds[mt[y]]==ds[x]+1&&dfs(mt[y])){
20      mt[y]=x;mt2[x]=y;
21      return true;
22    }
23    ds[x]=1<<30;
24    return false;
25  }
26  int mm(){
27    int r=0;
28    memset(mt,-1,sizeof(mt));memset(mt2,-1,sizeof(mt2));
29    while(bfs()){
30      rep(i,0,n)if(mt2[i]<0)r+=dfs(i);
31    }
32    return r;
33  }
```

2.12   Min cost - max flow

```
1   typedef ll tf;
2   typedef ll tc;
3   const tf INFFLOW=1e9;
4   const tc INFCOST=1e9;
5   struct MCF{
6     int n;
7     vector<tc> prio, pot; vector<tf> curflow; vector<int> prevedge,
          prevnode;
8     priority_queue<pair<tc, int>, vector<pair<tc, int>>, greater<pair<tc,
          int>>> q;
9     struct edge{int to, rev; tf f, cap; tc cost;};
10    vector<vector<edge>> g;
11    MCF(int n):n(n),prio(n),curflow(n),prevedge(n),prevnode(n),pot(n),g(n)
          {}
```

```
12    void add_edge(int s, int t, tf cap, tc cost) {
13      g[s].pb((edge){t,SZ(g[t]),0,cap,cost});
14      g[t].pb((edge){s,SZ(g[s])-1,0,0,-cost});
15    }
16    pair<tf,tc> get_flow(int s, int t) {
17      tf flow=0; tc flowcost=0;
18      while(1){
19        q.push({0, s});
20        fill(ALL(prio),INFCOST);
21        prio[s]=0; curflow[s]=INFFLOW;
22        while(!q.empty()) {
23          auto cur=q.top();
24          tc d=cur.fst;
25          int u=cur.snd;
26          q.pop();
27          if(d!=prio[u]) continue;
28          for(int i=0; i<SZ(g[u]); ++i) {
29            edge &e=g[u][i];
30            int v=e.to;
31            if(e.cap<=e.f) continue;
32            tc nprio=prio[u]+e.cost+pot[u]-pot[v];
33            if(prio[v]>nprio) {
34              prio[v]=nprio;
35              q.push({nprio, v});
36              prevnode[v]=u; prevedge[v]=i;
37              curflow[v]=min(curflow[u], e.cap-e.f);
38            }
39          }
40        }
41        if(prio[t]==INFCOST) break;
42        rep(i,0,n) pot[i]+=prio[i];
43        tf df=min(curflow[t], INFFLOW-flow);
44        flow+=df;
45        for(int v=t; v!=s; v=prevnode[v]) {
46          edge &e=g[prevnode[v]][prevedge[v]];
47          e.f+=df; g[v][e.rev].f-=df;
48          flowcost+=df*e.cost;
49        }
50      }
51      return {flow,flowcost};
52    }
53  };
```

2.13   2sat

```cpp
struct two_sat {
    int n;
    vector<vector<int>> g, gr;
    vector<int> comp, topological_order, answer;
    vector<bool> vis;

    two_sat(int _n) {
        n = _n;
        g.assign(2 * n, vector<int>());
        gr.assign(2 * n, vector<int>());
        comp.resize(2 * n);
        vis.resize(2 * n);
        answer.resize(2 * n);
    }

    void add_edge(int u, int v) {
        g[u].pb(v);
        gr[v].pb(u);
    }

    void dfs(int u) {
        vis[u] = true;
        each(v, g[u]) if (!vis[v]) dfs(v);
        topological_order.push_back(u);
    }

    void scc(int u, int id) {
        vis[u] = true, comp[u] = id;
        each(v, gr[u]) if (!vis[v]) scc(v, id);
    }

    bool satisfiable() {
        fill(vis.begin(), vis.end(), false);
        for (int i = 0; i < 2 * n; i++) if (!vis[i]) dfs(i);
        fill(vis.begin(), vis.end(), false);
        reverse(topological_order.begin(), topological_order.end());
        int id = 0;
        for(const auto &v : topological_order) if (!vis[v]) scc(v, id++)
            ;
        for(int i=0; i<n; i++) {
            if (comp[i] == comp[i + n]) return false;
            answer[i] = (comp[i] > comp[i + n] ? 1 : 0);
        }
        return true;
    }

    // Conditions
    void add_clause(int a, int b, string op, int c) {
        if(op=="=") {
            if(c==0) add_nor(a, b);
            else if(c==1) add_01_10(a, b);
            else add_and(a, b);
        }
        else if(op=="!=") {
            if(c==0) add_or(a, b);
            else if(c==1) add_same(a, b);
            else add_nand(a, b);
        }
        else if(op=="<") {
            if(c==0) {
                cout<<"No"<<endl;
                exit(0);
            }
            else if(c==1) add_nor(a, b);
            else add_nand(a, b);
        }
        else if(op==">") {
            if(c==0) add_or(a, b);
            else if(c==1) add_and(a, b);
            else {
                cout<<"No"<<endl;
                exit(0);
            }
        }
        else if(op=="<=") {
            if(c==0) add_nor(a, b);
            else if(c==1) add_nand(a, b);
            else return;
        }
        else {
            if(c==0) return;
            else if(c==1) add_or(a, b);
            else add_and(a, b);
        }
```

```
84        }
85
86        void add_nor(int a, int b) {
87            add_edge(a, a+n);
88            add_edge(b, b+n);
89        }
90        void add_01_10(int a, int b){
91            add_or(a, b);
92            add_nand(a, b);
93        }
94        void add_and(int a, int b) {
95            add_edge(a+n, a);
96            add_edge(b+n, b);
97        }
98        void add_or(int a, int b){
99            add_edge(a+n, b);
100           add_edge(b+n, a);
101       }
102       void add_same(int a, int b) {
103           add_edge(a, b);
104           add_edge(b+n, a+n);
105           add_edge(a+n, b+n);
106           add_edge(b, a);
107       }
108       void add_nand(int a, int b) {
109           add_edge(a, b+n);
110           add_edge(b, a+n);
111       }
112   };
113
114   int main() { io
115       int n, m; cin>>n>>m;
116       two_sat ts(n);
117       rep(i,0,m) {
118           int a, b, c; string op;
119           cin>>a>>b>>op>>c;
120           ts.add_clause(a, b, op, c);
121       }
122       cout<< (ts.satisfiable() ? "Yes" : "No")<<endl;
123       return 0;
124   }
```

3   Math

3.1   Binpow

```
1   ll binpow(ll a, ll b, ll m) {
2       a %= m;
3       ll res = 1;
4       while (b > 0) {
5           if (b & 1) res = res * a % m;
6           a = a * a % m;
7           b >>= 1;
8       }
9       return res;
10  }
```

3.2   Modular inverse

```
1   tuple<int, int, int> extendedGCD(int mod, int a) {
2       if (!a) return {mod, 1, 0};
3       auto[r, x, y] = extendedGCD(a, mod % a);
4       return {r, y, x - mod / a * y};
5   }
6   int modInverse(int mod, int a) {
7       auto[r, x, y] = extendedGCD(mod, a);
8       if(r != 1) {
9           return -1;
10      }
11      return ((y < 0) ? y + mod : y);
12  }
```

3.3   Linear Sieve

```
1   const int N = 1e7+5;  // N: range to get primes
2   int p[N];
3   vector<int> pr;
4   void linearSieve() {
5       for(int i=2; i<N; i++) {
6           if(!p[i]) {
7               p[i] = i;
8               pr.push_back(i);
9           }
10          for(int j=0; i*pr[j] < N; j++) {
11              p[i*pr[j]] = pr[j];
12              if(pr[j] == p[i])
```

```
13              break;
14          }
15      }
16 }
```

3.4   Factorials

```
1  void pre() {
2      fact[0] = 1;
3      for(int i=1; i<N; i++) {
4          fact[i] = 1LL * fact[i-1] * i % MOD;
5      }
6
7      ifact[N-1] = binpow(fact[N-1], MOD-2, MOD);
8      for(int i=N-2; i>=0; i--) {
9          ifact[i] = 1LL * ifact[i+1] * (i+1) % MOD;
10     }
11 }
```

3.5   Prime factorization

```
1  set<int> factors;
2  for(int i=2; i*i<=n; i++) {
3      while(!(n % i)) {
4          factors.insert(i);
5          n /= p;
6      }
7  }
8  if(n > 1) factors.insert(n);
```

3.6   Divisors

```
1  vector<int> div;
2  for(int i=1; i*i<=n; i++) {
3      if(!(n % i)) {
4          div.pb(d);
5          if(d != n/d) div.pb(n / d);
6      }
7  }
8  sort(all(div));
```

3.7   Identities

$$C_n = \frac{2(2n-1)}{n+1} C_{n-1}$$
$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$
$$\sigma(n) = O(\log(\log(n))) \text{ (number of divisors of } n)$$
$$F_{2n+1} = F_n^2 + F_{n+1}^2$$
$$F_{2n} = F_{n+1}^2 - F_{n-1}^2$$
$$\sum_{i=1}^{n} F_i = F_{n+2} - 1$$
$$F_{n+i}F_{n+j} - F_n F_{n+i+j} = (-1)^n F_i F_j$$
$$\text{(Fermat's little theorem) } a^p \, mod(p) = a \quad a^{p-1} \, mod(p) = 1 \quad a^{p-2} \, mod(p) = a^{-1}$$
$$\text{(Möbius Inv. Formula) Let } g(n) = \sum_{d|n} f(d), \text{ then } f(n) = \sum d \mid ng(d)\mu\left(\frac{n}{d}\right).$$

4   Strings

4.1   Rolling Hash

```
1  #define rep(i,a,b) for(int i=a;i<b;i++)
2  using ll = long long;
3
4  const int base = 251, MOD = 1e9+7, N = 2e5+5;
5  ll p[N], hashes[N];
6  vector<int> rolling_hash(string text, string pattern) {
7      int n = text.length(), m = pattern.length();
8      p[0] = 1;
9      rep(i,1,n+1) p[i] = p[i-1] * base % MOD;
10
11     rep(i,1,n+1) hashes[i] = (hashes[i-1] + (text[i-1]-'a'+1) * p[i-1] %
            MOD) % MOD;
12
13     ll patternHash = 0;
14     rep(i,1,m+1) patternHash = (patternHash + (pattern[i-1]-'a'+1) * p[i
            -1] % MOD) % MOD;
15
16     vector<int> ocu;
17     rep(i,0,n-m+1) {
18         ll curr = (hashes[i+m] - hashes[i] + MOD) % MOD;
19         if(curr == patternHash * p[i] % MOD) ocu.push_back(i);
20     }
21     return ocu;
22 }
23
24 // 0. Define 2 prime numbers: base and mod
25 // 1. Precompute powers of the base
26 // 2. Compute prefix hashes for the text
27 // 3. Compute hash of the pattern
```

```
28    // 4. Sliding window of pattern size over the text to find matches
```

4.2   LCP using hashing + bs

```
1    int getLCP(int i, int j, int len) {
2        int l = 1, r = len;
3        while(l <= r) {
4            int m = (l+r) >> 1;
5            ll left = (hashes[i+m] - hashes[i] + MOD) % MOD;
6            ll right = (hashes[j+m] - hashes[j] + MOD) % MOD;
7            left = left * p[j-i] % MOD;
8            if(left == right) l = m + 1;
9            else r = m - 1;
10        }
11        return r;
12    }
13    // Compute prefix hashes for the text
14    // LCP: getLCP(idx, idx+len, n)
15    // length of most repeated substring: LCP+len
```

5   Geometry

5.1   Convex Hull

```
1    const double eps = 1e-9;
2    bool leq(double a, double b){return b-a >= -eps;}
3    bool le(double a, double b){return b-a > eps;}
4    bool eq(double a, double b){return fabs(a-b) <= eps;}
5
6    struct point{
7        double x, y;
8        int idx = -1;
9        point(): x(0), y(0){}
10        point(double x, double y): x(x), y(y){}
11        point operator-(const point &p) const{return point(x - p.x, y - p.y)
              ;}
12        point operator*(const int &k) const{return point(x * k, y * k);}
13        bool operator<(const point &p) const{return le(x, p.x) || (eq(x, p.x
              )&& le(y, p.y));}
14        bool operator==(const point &p) const{return eq(x, p.x) && eq(y, p.y
              );}
15        double cross(const point &p) const{return x * p.y - y * p.x;}
16    };
```

```
17
18    istream &operator>>(istream &is, point &p){return is >> p.x >> p.y;}
19    ostream &operator<<(ostream &os, const point &p){return os << "(" << p.x
              << ",␣" << p.y << ")";}
20
21    vector<point> convexHull(vector<point> P) {
22        sort(P.begin(), P.end());
23        vector<point> L, U;
24        for (int i = 0; i < P.size(); i++) {
25            while (L.size() >= 2 && le((L[L.size() - 2] - P[i]).cross(L[L.
                  size() - 1] - P[i]), 0)) {
26                L.pop_back();
27            }
28            L.push_back(P[i]);
29        }
30        for (int i = P.size() - 1; i >= 0; i--) {
31            while (U.size() >= 2 && le((U[U.size() - 2] - P[i]).cross(U[U.
                  size() - 1] - P[i]), 0)) {
32                U.pop_back();
33            }
34            U.push_back(P[i]);
35        }
36        L.pop_back();
37        U.pop_back();
38        L.insert(L.end(), U.begin(), U.end());
39        return L;
40    }
41
42    bool pointInConvexHull(const vector<point> &poly, point p) {
43        int n = poly.size();
44        if(n < 3) return false;
45        rep(i,0,n) {
46            point a = poly[i], b = poly[(i+1) % n];
47            double cp = (b - a).cross(p - a);
48            if(!le(0, cp)) return false;
49        }
50        return true;
51    }
52
53    int main() {
54        ios::sync_with_stdio(0); cin.tie(0);
55        int n;
56        double h;
```

```
57        cin >> n >> h;
58
59        vector<point> islands(n);
60        vector<double> time(n), heights(n);
61        rep(i,0,n) {
62            cin >> islands[i] >> heights[i];
63            islands[i].idx = i;
64            time[i] = max(0.0, h - heights[i]);
65        }
66
67        double l = 0.0, r = *max_element(all(time));
68        point c;
69        cin >> c;
70        auto valid = [&](double t) {
71            vector<point> aux;
72            rep(i,0,n) {
73                if(t >= time[i]) aux.pb(islands[i]);
74            }
75            if(aux.size() < 3) return false;
76            aux = convexHull(aux);
77            return pointInConvexHull(aux, c);
78        };
79
80        if(!valid(r)) {
81            cout << -1 << endl;
82            return 0;
83        }
84
85        for(int it=0; it<100; it++) {
86            double m = (l + r) / 2.0;
87            if(valid(m)) {
88                r = m;
89            }
90            else {
91                l = m;
92            }
93        }
94        cout << fixed << setprecision(12) << r << endl;
95
96        return 0;
97 }
```

6   Techniques

6.1   MO's algorithm

```
1  const int N = 3e5+5, B = 550;
2  int values[N], freq[N], res[N], ans, has[B];
3  void add(int idx) {
4      has[values[idx]/B]++;
5      freq[values[idx]]++;
6  }
7  void remove(int idx) {
8      has[values[idx]/B]--;
9      freq[values[idx]]--;
10 }
11 int getAns(int k) {
12     rep(i,0,B) if(has[i] > k) rep(j,i*B,(i+1)*B)
13         if(freq[j] > k) return j;
14     return -1;
15 }
16 void mos(vector<array<int, 4>> &queries) {
17     sort(all(queries), [](array<int, 4> &a, array<int, 4> &b) {
18         if(a[0]/B != b[0]/B) return a[0] < b[0];
19         return (a[0]/B & 1 ? a[1] < b[1] : a[1] > b[1]);
20     });
21
22     int l = 0, r = -1;
23     for(auto &[ql, qr, k, idx] : queries) {
24         while(l > ql) add(--l);
25         while(r < qr) add(++r);
26         while(l < ql) remove(l++);
27         while(r > qr) remove(r--);
28         res[idx] = getAns(k);
29     }
30 }
```

6.2   Parallel binary search

```
1  void parallel_bs() {
2      vector<array<int, 2>> queries(n+1);
3      rep(i,1,n+1) queries[i] = {1, q};
4
5      bool f = 1;
6      while(f) {
```

```
 7          f = 0;
 8          vector<vector<int>> mids(m+2);
 9          for(auto &it : queries) {
10              if(it.l <= it.r) {
11                  int mid = (it.l + it.r) >> 1;
12                  mids[mid].pb(it.idx);
13                  f = 1;
14              }
15          }
16
17          obj.reset();
18
19          rep(mid,1,q+1) {
20              auto &[l, r, k] = values[mid];
21              obj.update(l, r, k);
22
23              for(auto &idx : mids[mid]) {
24                  Queries &q = queries[idx];
25                  if(ok) q.r = mid - 1;
26                  else q.l = mid + 1;
27              }
28          }
29      }
30 }
```

6.3   Split objects into light and heavy

```
 1 vector<pair<int, int>> adj[N];
 2 bitset<N> heavy;
 3 const int B = 2050;
 4 vector<pair<int, int>> heavyVertices[N];
 5 int values[N], deg[N];
 6
 7 void solve() {
 8     ll res = 0;
 9     auto dfs = [&](auto self, int u, int p = -1) -> void {
10         for(auto &[v, x] : adj[u]) {
11             if(v == p) continue;
12             if(values[u] != values[v]) res += 1ll * x;
13             deg[u]++, deg[v]++;
14             self(self, v, u);
15         }
16     };
```

```
17     dfs(dfs, 0);
18
19     rep(i,0,n) if(deg[i] >= B) heavy[i] = 1;
20
21     rep(u,0,n) {
22         if(heavy[u]) {
23             for(auto &[v, x] : adj[u]) {
24                 colors[u][values[v]] += 1ll * x;
25                 if(heavy[v])
26                     heavyVertices[u].pb({v, x});
27                     heavyVertices[v].pb({u, x});
28             }
29         }
30     }
31
32     while(q--) {
33         int u, c;
34         cin >> u >> c;
35         u--;
36
37         if(heavy[u]) {
38             res += 1ll * colors[u][values[u]];
39             res -= 1ll * colors[u][c];
40             for(auto &[v, x] : heavyVertices[u]) {
41                 colors[v][values[u]] -= 1ll * x;
42                 colors[v][c] += 1ll * x;
43             }
44             values[u] = c;
45         }
46         else {
47             for(auto &[v, x] : adj[u]) {
48                 if(values[u] != values[v]) {
49                     if(c == values[v]) res -= 1ll * x;
50                 }
51                 else {
52                     if(c != values[v]) {
53                         res += 1ll * x;
54                     }
55                 }
56                 if(heavy[v]) {
57                     colors[v][values[u]] -= 1ll * x;
58                     colors[v][c] += 1ll * x;
59                 }
```

```
60          }
61          values[u] = c;
62        }
63
64      cout << res << endl;
65    }
66 }
```

6.4   Meet in the middle

```
1  int main() {
2      ios::sync_with_stdio(0); cout.tie(0); cin.tie(0);
3      int n; cin >> n;
4      vector<long long> a(n), b(n);
5      for(auto &it : a) cin >> it;
6      for(auto &it : b) cin >> it;
7
8      auto subsets = [](vector<long long> &a, vector<long long> &b, bool f
           ){
9          int n = a.size();
10         vector<long long> aux;
11         for(int mask=0; mask<(1<<n); mask++) {
12             long long sum = 0;
13             for(int j=0; j<n; j++) {
14                 if((1 << j) & mask) sum += a[j];
15                 else sum -= b[j];
16             }
17             if(f) sum = abs(sum);
18             aux.push_back(sum);
19         }
20         return aux;
21     };
22
23     if(n <= 20) {
24         vector<long long> res = subsets(a, b, 1);
25         sort(res.begin(), res.end());
26         for(auto &it : res) cout << it << " ";
27         cout<< endl;
28         return 0;
29     }
30
31     vector<long long> leftA(a.begin(), a.begin()+n/2),
32                       leftB(b.begin(), b.begin()+n/2),
33                       rightA(a.begin()+n/2, a.end()),
34                       rightB(b.begin()+n/2, b.end());
35
36     vector<long long> left = subsets(leftA, leftB, 0),
37                       right = subsets(rightA, rightB, 0);
38     sort(right.begin(), right.end());
39
40     priority_queue<tuple<long long, int, int, int>, vector<tuple<long
           long, int, int, int>>, greater<tuple<long long, int, int, int>>>
            pq;
41     for(int i=0; i<left.size(); i++) {
42         int idx = lower_bound(right.begin(), right.end(), -left[i]) -
               right.begin();
43         long long best = LONG_LONG_MAX, curr;
44         int idxBest;
45         for(int j=-1; j<=1; j++) {
46             if(idx + j >= 0 && idx + j < right.size()) {
47                 curr = abs(left[i] + right[idx+j]);
48                 if(curr < best) {
49                     best = curr;
50                     idxBest = idx+j;
51                 }
52             }
53         }
54         pq.push({best, idxBest, i, 0});
55     }
56
57     int req = 1 << 20, cnt = 0;
58     while(!pq.empty() && cnt < req) {
59         auto [sum, idx, i, event] = pq.top();
60         pq.pop();
61         cout << sum << " ";
62         cnt++;
63         if(event <= 0 && idx-1 >= 0) {
64             long long lsum = abs(left[i] + right[idx-1]);
65             pq.push({lsum, idx-1, i, -1});
66         }
67         if(event >=0 && idx+1 < right.size()) {
68             long long rsum = abs(left[i] + right[idx+1]);
69             pq.push({rsum, idx+1, i, 1});
70         }
71     }
72     cout << endl;
```

```
73      return 0;
74
75  }
```

6.5   Dijkstra on ST

```
1   vector<pair<int, int>> adj[2*4*N];
2   ll dist[2*4*N];
3   int mp[N], vis[2*4*N];
4   void build(int l, int r, int i) {
5       if (l == r) {
6           mp[l] = i;
7           adj[i].pb({i+4*N, 0});
8           adj[i+4*N].pb({i, 0});
9           return;
10      }
11      int m = (l+r)>>1;
12      build(l, m, 2*i+1);
13      build(m+1, r, 2*i+2);
14      adj[i].pb({2*i+1, 0});
15      adj[i].pb({2*i+2, 0});
16      adj[2*i+1+4*N].pb({i+4*N, 0});
17      adj[2*i+2+4*N].pb({i+4*N, 0});
18  }
19  void add(int l, int r, int u, int a, int b, int w, int op, int i) {
20      if (a > r || b < l) return;
21      if (a <= l && r <= b) {
22          if(op == 2) adj[mp[u]].pb({i, w});
23          else adj[i+4*N].pb({mp[u]+4*N, w});
24          return;
25      }
26      int m = l+r>>1;
27      add(l, m, u, a, b, w, op, 2*i+1);
28      add(m+1, r, u, a, b, w, op, 2*i+2);
29  }
```

6.6   Venice set

```
1   struct VeniceSet {
2       multiset<int> st;
3       int global = 0;
4
5       void add(int x) {
6           st.insert(x + global);
```

```
7       }
8       void remove(int x) {
9           st.erase(st.find(x + global));
10      }
11      void updateAll(int x) {
12          global += x;
13      }
14      int min(int x) {
15          return *st.begin() - global;
16      }
17  };
```

7   Basic techniques

7.1   Difference array

```
1   // If we only need to get the final result, we can increase l and r+1
2   // to simulate range update and use a prefix sum to get the answer.
3   diff[l] += k;
4   diff[r+1] -= k;
5
6   // If we have initial values, we need to subtract the previous values:
7   diff[i] -= values[i - 1];
```

7.2   Prefix Sum 2D

```
1   // Compute prefix sum 2D
2   rep(i,1,n+1) rep(j,1,m+1)
3       prefix[i][j] = prefix[i-1][j] + prefix[i][j-1]
4                                     - prefix[i-1][j-1]
5                                     + matrix[i-1][j-1]
6
7   // Query prefix sum 2D
8   query = prefix[i][j] - prefix[i-1][j] - prefix[i][j-1] + prefix[i-1][j
        -1]
9
10  // Specific parts
11  query = prefix[r2][c2] - prefix[r1-1][c2] - prefix[r2][c1-1] + prefix[r1
        -1][c1-1];
```

7.3   Random number generator

```
1   // Secure random seed
2   mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
```

```
3
4  // Generate values using distribution
5  uniform_int_distribution<int> dist(0, N);        // [0, N]
6  randval = dist(rng);
```

7.4   Coordinate compression

```
1  // Push to an array all possible values
2  coord.pb(x);
3  sort(all(coord));
4  coord.resize(unique(all(coord)) - coord.begin());
5
6  auto get = [&](int x) = {
7      return lower_bound(all(coord), x) - coord.begin();
8  };
```

7.5   Digit DP

```
1  int dp[11][2][2];
2  auto calc = [&](auto self, int idx, bool f1, bool f2) -> int {
3      if(idx >= n) return 0;
4
5      int &x = dp[idx][f1][f2];
6      if(x != -1) return x;
7
8      int num1 = a[idx] - '0', num2 = b[idx] - '0';
9      int l = (f1 ? 0 : num1);
10     int r = (f2 ? 9 : num2);
11     int mn = 1e9+5;
12
13     rep(i,l,r+1) {
14         mn = min(mn, self(self, idx+1, f1 | (i > num1), f2 | (i < num2))
                   + (num1==i) + (num2==i));
15     }
16     return x = mn;
17 };
18 memset(dp, -1, sizeof(dp));
19 cout << calc(calc, 0, 0, 0) << endl;
```

7.6   Intersection [L1, R1] and [L2, R2]

```
1  We need to check whether there is any number common to both ranges:
2  max(L1, L2) <= min(R1, R2)
```