Contents

Base **1**

Adhocs **1**

Data Structure **3**

Graph **26**

Trees **36**

Strings **40**

Base

template

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int long long
void solve() {}
signed main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
// add_definitions(-D Clion) # add it to CMakeLists.txt
#ifdef Clion
  freopen("Input.txt", "r", stdin);
  freopen("Output.txt", "w", stdout);
#endif
  int t;
  cin >> t;
  while (t--) solve();
  return 0;
}
```

Adhocs

01.Kadane

```cpp
int kadane(const vector<int>& nums) {
  int maxSoFar = INT_MIN;
  int maxEndingHere = 0;
  for (int num : nums) {
    maxEndingHere += num;
    if (maxSoFar < maxEndingHere) maxSoFar = maxEndingHere;
    if (maxEndingHere < 0) maxEndingHere = 0;
  }
  return maxSoFar;
}
```

02.2D Kadane

```cpp
int kadane(vector<int> &arr, int &start, int &end) {
  int maxSum = INT_MIN, currSum = 0, tempStart = 0;
  start = end = 0;
  for (int i = 0; i < arr.size(); ++i) {
    currSum += arr[i];
    if (currSum < arr[i]) {
      currSum = arr[i];
      tempStart = i;
    }
    if (currSum > maxSum) {
      maxSum = currSum;
      start = tempStart;
      end = i;
    }
  }
  return maxSum;
}
int maxRectSum(vector<vector<int>> &mat) {
  int rows = mat.size(), cols = mat[0].size(), maxSum = INT_MIN;
  int finalTop = 0, finalBottom = 0, finalLeft = 0, finalRight = 0;
  vector<int> temp(rows);

  for (int left = 0; left < cols; ++left) {
    fill(temp.begin(), temp.end(), 0);  // reset temp for each new left
    for (int right = left; right < cols; ++right) {
      for (int row = 0; row < rows; ++row) temp[row] += mat[row][right];
      int startRow, endRow;
      int sum = kadane(temp, startRow, endRow);
      if (sum > maxSum) {
        maxSum = sum;
        finalTop = startRow;
        finalBottom = endRow;
        finalLeft = left;
        finalRight = right;
      }
    }
  }
  cout << "(Top, Left)     : (" << finalTop << ", " << finalLeft << ")\n";
  cout << "(Bottom, Right) : (" << finalBottom << ", " << finalRight <<
  ↪   ")\n";
  cout << "Maximum Sum     : " << maxSum << endl;
  return maxSum;
}
```

```
// vector<vector<int>> mat = {{1, 2, -1, -4, -20}, {-8, -3, 4, 2, 1}, {3, 8, 10, 1, 3}};
// maxRectSum(mat);
```

03.2D PrefixSum

```
void PrefixSum_2D(int n, int m, vector<vector<int>>& Prefix) {
  for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) { Prefix[i][j] += Prefix[i][j - 1]; }
  }
  for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) { Prefix[j][i] += Prefix[j - 1][i]; }
  }
}
int getSum(int i, int j, int x, int y) {
  return Prefix[x][y] - Prefix[i - 1][y] - Prefix[x][j - 1] + Prefix[i - 1][j - 1];
}
```

04.nextSmaller

```
vector<int> nextSmaller(vector<int> &a) {
  int n = a.size();
  vector<int> result(n, n), st;
  for (int i = 0; i < n; ++i) {
    while (!st.empty() && a[st.back()] > a[i]) result[st.back()] = i, st.pop_back();
    st.push_back(i);
  }
  return result;
}
```

05.minimum swaps

```
// minimum adjacent swaps to convert a to b
int cost(vector<int> &a, vector<int> &b) {
  int n = a.size();
  map<int, deque<int>> pos;
  ordered_set<int> st;
  int res = 0;
  for (int i = 0; i < n; ++i) {
    pos[a[i]].push_back(i);
    st.insert(i);
  }
  for (int i = 0; i < n; ++i) {
    int idx = pos[b[i]].front();
    pos[b[i]].pop_front();
    res += st.order_of_key(idx);
    st.erase(idx);
  }
  return res;
}
```

06.previousSmaller

```
vector<int> previousSmaller(vector<int> &a) {
  int n = a.size();
  vector<int> result(n, -1), st;
  for (int i = 0; i < n; ++i) {
    while (!st.empty() && a[st.back()] >= a[i]) st.pop_back();
    if (!st.empty()) result[i] = st.back();
    st.push_back(i);
  }
  return result;
}
```

07.2D PartialSum

```
void PartialSum_in2D(int x1, int y1, int x2, int y2, vector<vector<int>>& Prefix) {
  if (x1 > x2) swap(x1, x2);
  if (y1 > y2) swap(y1, y2);
  Prefix[x1][y1]++;
  Prefix[x2 + 1][y1]--;
  Prefix[x1][y2 + 1]--;
  Prefix[x2 + 1][y2 + 1]++;
}
```

08.nextGreater

```
vector<int> nextGreater(vector<int>& a) {
  int n = a.size();
  vector<int> result(n, n), st;
  for (int i = 0; i < n; ++i) {
    while (!st.empty() && a[st.back()] < a[i]) result[st.back()] = i,
     ↪  st.pop_back();
    st.push_back(i);
  }
  return result;
}
```

09.previousGreater

```
vector<int> previousGreater(vector<int>& a) {
  int n = a.size();
  vector<int> result(n, -1), st;
  for (int i = 0; i < n; ++i) {
    while (!st.empty() && a[st.back()] <= a[i]) st.pop_back();
    if (!st.empty()) result[i] = st.back();
    st.push_back(i);
  }
  return result;
}
```

Data Structure

2D SparseTable

```
const int N = 1005;
const int LOG = 10;
int n, m;
int matrix[N][N];
int sp[N][N][LOG][LOG];
// Preprocesses the 2D sparse table
void buildSparseTable() {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j) sp[i][j][0][0] = matrix[i + 1][j + 1];
  for (int x = 0; (1 << x) <= n; ++x) {
    for (int y = 0; (1 << y) <= m; ++y) {
      if (x + y > 0) {
        for (int i = 0; i + (1 << x) - 1 < n; ++i) {
          for (int j = 0; j + (1 << y) - 1 < m; ++j) {
            if (x == 0)
              sp[i][j][x][y] =
                  max(sp[i][j][x][y - 1], sp[i][j + (1 << (y - 1))][x][y -
                  ↪  1]);
            else
              sp[i][j][x][y] =
                  max(sp[i][j][x - 1][y], sp[i + (1 << (x - 1))][j][x -
                  ↪  1][y]);
          }
        }
      }
    }
  }
}
// Answers RMQ from (x1, y1) to (x2, y2)
int query(int x1, int y1, int x2, int y2) {
  x1--, y1--, x2--, y2--;
  int kx = __lg(x2 - x1 + 1);
  int ky = __lg(y2 - y1 + 1);
  int min1 = max(sp[x1][y1][kx][ky], sp[x2 - (1 << kx) + 1][y1][kx][ky]);
  int min2 = max(sp[x1][y2 - (1 << ky) + 1][kx][ky],
                 sp[x2 - (1 << kx) + 1][y2 - (1 << ky) + 1][kx][ky]);
  return max(min1, min2);
}
```

3D MO

```
const int N = 1e5 + 1, B = 4500, C = N / B + 1;
struct DistinctCounter {
  int cnt[N * 2];  // Frequency of each element
  int distinct;    // Number of distinct elements
  DistinctCounter() {
    memset(cnt, 0, sizeof cnt);
    distinct = 0;
  }
  void add(int x) {
    // If x was not present, increment distinct count
    if (cnt[x] == 0) distinct++;
    cnt[x]++;
  }
  void del(int x) {
    cnt[x]--;
```

```cpp
    // If x is no longer present, decrement distinct count
    if (cnt[x] == 0) distinct--;
  }
  int get() {
    // Return the number of distinct elements
    return distinct;
  }
} t[C * (C + 1) / 2 + 10], ds;
int st[C], en[C], BC = 0;
int a[N], I[N];
int query(int l, int r) {
  int L = l / B, R = r / B;
  // Adjust R if the range doesn't fully cover the last block
  if (r != en[R]) R--;
  // Adjust L if the range doesn't fully cover the first block
  if (l != st[L]) L++;
  if (R < L) {
    // If the range is within a single block, handle it directly
    for (int i = l; i <= r; i++) ds.add(a[i]);
    int ans = ds.get();
    for (int i = l; i <= r; i++) ds.del(a[i]);
    return ans;
  }
  // Otherwise, use the precomputed distinct counts for the fully covered
  ↪  blocks
  int id = I[L * BC + R];
  // Add elements from the left partial block
  for (int i = l; i < st[L]; i++) t[id].add(a[i]);
  // Add elements from the right partial block
  for (int i = en[R] + 1; i <= r; i++) t[id].add(a[i]);
  int ans = t[id].get();
  // Remove elements from the left partial block
  for (int i = l; i < st[L]; i++) t[id].del(a[i]);
  // Remove elements from the right partial block
  for (int i = en[R] + 1; i <= r; i++) t[id].del(a[i]);
  return ans;
}
inline void upd(int id, int pos, int val) {
  t[id].del(a[pos]);  // Remove the old value
  t[id].add(val);     // Add the new value
}
map<int, int> mp;
int nxt = 0;
int get(int x) {  // Coordinate compression
```

```cpp
  return mp.count(x) ? mp[x] : mp[x] = ++nxt;
}
int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, q;
  cin >> n >> q;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
    a[i] = get(a[i]);  // Compress coordinates
  }
  for (int i = 0; i < n; i++) {
    if (i % B == 0) st[i / B] = i, BC++;              // Start of each block
    if (i % B == B - 1 || i == n - 1) en[i / B] = i;  // End of each block
  }
  int nw = 0;
  for (int i = 0; i < BC; i++) {
    for (int j = i; j < BC; j++) {
      int id = nw;
      I[i * BC + j] = nw++;  // Map block range to ID
      // Precompute distinct counts for block ranges
      for (int p = st[i]; p <= en[j]; p++) t[id].add(a[p]);
    }
  }
  while (q--) {
    int ty;
    cin >> ty;
    if (ty == 2) {
      int l, r;
      cin >> l >> r;
      --l;
      --r;
      cout << n - query(l, r) << '\n';  // Handle query
    } else {
      int pos, val;
      cin >> pos >> val;
      --pos;
      val = get(val);  // Compress the new value
      for (int i = 0; i < BC; i++) {
        for (int j = i; j < BC; j++) {  // Update all relevant block ranges
          if (st[i] <= pos && pos <= en[j]) upd(I[i * BC + j], pos, val);
        }
      }
      a[pos] = val;  // Update the array
    }
  }
  return 0;
}
```

Binary Trie

```
// The Kth smallest subarray of length >= 2
// the value of a subarray is the minimum a[i] ^ a[j]
struct Trie_B {
  private:
  struct Node {
    int child[2]{};
    int cnt = 0, isEnd = 0;
    int &operator[](int x) { return child[x]; }
  };
  vector<Node> node;
  public:
  Trie_B() { node.emplace_back(); }
  int newNode() {
    node.emplace_back();
    return node.size() - 1;
  }
  int sz(int x) { return node[x].cnt; }
  int M = 30;
  void update(int x, int op) {  // op -> 1 add || op -> -1 erase
    int cur = 0;
    for (int i = M - 1; i >= 0; --i) {
      int c = x >> i & 1;
      if (node[cur][c] == 0) node[cur][c] = newNode();
      cur = node[cur][c];
      node[cur].cnt += op;
    }
    node[cur].isEnd += op;
  }
  int min_xor(int x) {  // min xor with x
    int cur = 0, res = 0;
    for (int i = M - 1; i >= 0; --i) {
      int cx = x >> i & 1;
      if (sz(node[cur][cx])) {
        cur = node[cur][cx];
      } else {
        res += 1LL << i;
        cur = node[cur][!cx];
      }
    }
    return res;
  }
  int max_xor(int x) {
    int cur = 0, res = 0;
    for (int i = M - 1; i >= 0; --i) {
      int cx = (x >> i) & 1;
      if (sz(node[cur][!cx])) {
        res |= (1LL << i);
        cur = node[cur][!cx];
      } else {
        cur = node[cur][cx];
      }
    }
    return res;
  }
  int count(int x, int k) {  // number of x ^ a[i] >= k
    int cur = 0, res = 0;
    for (int i = M - 1; i >= 0; --i) {
      int cx = x >> i & 1;
      int ck = k >> i & 1;
      if (!ck) res += sz(node[cur][!cx]);
      cur = node[cur][ck ^ cx];
      if (sz(cur) == 0) break;
    }
    return res + node[cur].cnt;
  }
};
signed main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; ++i) cin >> a[i];
  auto can = [&](int x) {
    int l = 0, r = 0, cnt_pairs = 0, res = 0;
    Trie_B trie;
    while (r < n) {
      cnt_pairs += trie.count(a[r], x);
      trie.update(a[r++], 1);
      while (cnt_pairs > 0) {
        trie.update(a[l], -1);
        cnt_pairs -= trie.count(a[l++], x);
      }
      res += l;
    }
    return res >= k;
  };
  int l = 0, r = 4e9, mid, ans = 0;
  while (l <= r) {
    mid = (l + r) / 2;
    if (can(mid)) {
      ans = mid;
      r = mid - 1;
    } else {
      l = mid + 1;
    }
  }
  cout << ans << '\n';
  return 0;
}
```

BIT

```
struct BIT {
    int n;  vector< ll > bit1, bit2;
    BIT(int size) : n(size) {
        bit1.assign(n + 2, 0);
        bit2.assign(n + 2, 0);
    }
    void add(vector< ll > &bit, int idx, ll val) {
        idx++;
        while (idx <= n) {
            bit[idx] += val;
            idx += idx & -idx;
        }
    }
    ll query_bit(const vector< ll > &bit, int idx) const {
        idx++;
        ll res = 0;
        while (idx > 0) {
            res += bit[idx];
            idx -= idx & -idx;
        }
        return res;
    }
    void add(int i, ll val) { update(i, i, val); }
    void update(int l, int r, ll val) {
        add(bit1, l, val);
        add(bit1, r + 1, -val);
        add(bit2, l, val * (l - 1));
        add(bit2, r + 1, -val * r);
    }
    ll prefix_sum(int i) const {
        return query_bit(bit1, i) * i - query_bit(bit2, i);
    }
    ll range_query(int l, int r) const {
        return prefix_sum(r) - prefix_sum(l - 1);
    }
    ll point_query(int i) const { return range_query(i, i); }
};
```

BIT-Simple

```
struct BIT {
    int n;
    vector< int > t;
    BIT(int n): n(n), t(n + 1) {}
    void add(int i, int x) {
        for (; i < n; i |= i + 1) t[i] += x;
    }
    int get(int i) {
        int res = 0;
        for (; i >= 0; i = (i & i + 1) - 1) res += t[i];
        return res;
    }
    int get(int l, int r) { return get(r) - get(l - 1); }
};
```

DSU

```
/*
 * You are given an empty undirected graph on n vertices.
 * Each vertex is colored in one of two colors 0 or 1
 * such that each edge connects the vertices with different colors.
 * There are two types of queries:
 * 1. You are given two vertices x and y from different connected
 * components: add an edge (x,y) to the graph, and change the colors
 * to satisfy the condition.
 * 2. You are given two vertices x and y from one connected
 * component: answer whether they are of the same color.
 * */
struct DSU {
  vector<int> parent, size, depth;
  int component;
  DSU(int n) {
    parent.assign(n + 1, {});
    size.assign(n + 1, 1);
    depth.assign(n + 1, 0);
    component = n;
    iota(parent.begin(), parent.end(), 0);
  }
  pair<int, int> find(int x) {
    if (x == parent[x]) return {x, 0};
    auto v = find(parent[x]);
    parent[x] = v.first;   // Path compression
    depth[x] += v.second;  // update depth
    return {v.first, depth[x]};
  }
  void union_set(int a, int b) {
    int rootA = find(a).first;
    int rootB = find(b).first;
    if (rootA == rootB) return;
    if (size[rootA] < size[rootB]) swap(rootA, rootB);
    parent[rootB] = rootA;
    depth[rootB] = (depth[a] + depth[b] + 1) % 2;
    size[rootA] += size[rootB];
  }
  bool isBip(int a, int b) { return find(a).second % 2 == find(b).second %
  ↪  2; }
  bool isConnected(int a, int b) { return find(a).first == find(b).first; }
};
signed main() {
  int n, m, shift = 0;
  cin >> n >> m;
  DSU dsu(n);
  for (int i = 0, t, a, b; i < m; ++i) {
    cin >> t >> a >> b;
    int x = (a + shift) % n;
    int y = (b + shift) % n;
    if (t == 0) {
      dsu.union_set(x, y);
    } else {
      bool ok = dsu.isBip(x, y);
      if (ok) {
        shift = (shift + 1) % n;
        cout << "YES\n";
      } else {
        cout << "NO\n";
      }
    }
  }
}
```

DSU RollBack

```
/*
 *You are given a graph with n vertices and m undirected edges.
 * Write a program that processes k queries of the form (li,ri):
 * the answer for the i-th query is the number of connected components
 * if we remove all the edges from the graph except edges with indices from
 ↪  li
 *to ri inclusive. The queries should be answered independently. In other
 ↪  words,
 *to answer the i-th query, you should consider a graph that has n vertices
 ↪  and
 *ri-li+1 edges.
 * */
struct DSU_RollBack {
  vector<int> parent, size, checkPoint;
  vector<pair<int, int>> update;
  int component;
  DSU_RollBack(int n) {
    parent.assign(n + 1, {});
    size.assign(n + 1, 1);
    component = n;
    iota(parent.begin(), parent.end(), 0);
  }
  int find(int x) {
    while (x != parent[x]) x = parent[x];
    return x;
  }
  void snapShot() { checkPoint.emplace_back(update.size()); }
  void union_sets(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
    component--;
    update.emplace_back(a, b);
  }
  void RollBack() {
    while (checkPoint.back() != update.size()) {
      auto [a, b] = update.back();
      update.pop_back();
      parent[b] = b;
      size[a] -= size[b];
      component++;
    }
    checkPoint.pop_back();
  }
};
signed main() {
  int n, m;
  cin >> n >> m;
  vector<array<int, 2>> edges(m);
  for (int i = 0; i < m; ++i) cin >> edges[i][0] >> edges[i][1];
  int B = (int)sqrt(m) + 1;
  vector<array<int, 3>> query[B];  // r , l , i;
  int k;
  cin >> k;
  vector<int> ans(k);
  for (int i = 0, l, r; i < k; ++i) {
    cin >> l >> r, --l, --r;
    if (r - l + 1 <= B) {
      DSU_RollBack ds(n);
      for (int j = l; j <= r; ++j) { ds.union_sets(edges[j][0],
      ↪  edges[j][1]); }
      ans[i] = ds.component;
      continue;
    }
    query[l / B].push_back({r, l, i});
  }
  for (int i = 0; i < B; ++i) {
    if (query[i].empty()) continue;
    sort(query[i].begin(), query[i].end());
    int r = (i + 1) * B - 1;
    DSU_RollBack ds(n);
    ds.union_sets(edges[r][0], edges[r][1]);
    for (auto [rq, lq, iq] : query[i]) {
      int l = (i + 1) * B - 1;
      while (rq > r) {
        ++r;
        ds.union_sets(edges[r][0], edges[r][1]);
      }
      ds.snapShot();
      while (lq < l) {
        --l;
        ds.union_sets(edges[l][0], edges[l][1]);
      }
      ans[iq] = ds.component;
      ds.RollBack();
    }
  }
  for (auto i : ans) { cout << i << '\n'; }
  return 0;
}
```

Dynamic Connectivity

```
/*
 * You are given an empty undirected graph with n vertices. You have to
 ↪  answer
 * the queries of three types:
 * 1. "+ u v" - add an undirected edge u-v to the graph.
 * 2. "- u v" - remove an undirected edge u-v from the graph.
 * 3. "?" - calculate the number of connected components in the graph
 * */
struct DSU {
  vector<int> parent, size, checkPoint;
  vector<pair<int, int>> update;
  int component;
  DSU(int n) {
    parent.assign(n + 1, {});
    size.assign(n + 1, 1);
    component = n;
    iota(parent.begin(), parent.end(), 0);
  }
  int find(int x) {
    if (x == parent[x]) return x;
    return find(parent[x]);
  }
  void union_sets(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
    component--;
    update.emplace_back(a, b);
  }
  void snapshot() { checkPoint.push_back(update.size()); }
  void RollBack() {
    while (checkPoint.back() != update.size()) {
      auto [a, b] = update.back();
      update.pop_back();
      parent[b] = b;
      size[a] -= size[b];
      component++;
    }
    checkPoint.pop_back();
  }
};
const int N = 3e5 + 5;
vector<pair<int, int>> tree[N * 4];
#define mid ((l + r) / 2)
#define LF (2 * node + 1)
#define RT (2 * node + 2)
void update(int node, int l, int r, int Lx, int Rx, pair<int, int> x) {
  if (r < Lx or Rx < l) return;
  if (Lx <= l and r <= Rx) {
    tree[node].push_back(x);
    return;
  }
  update(LF, l, mid, Lx, Rx, x);
  update(RT, mid + 1, r, Lx, Rx, x);
}
DSU ds(N);
void dfs(int node, int l, int r) {
  ds.snapshot();
  for (auto [a, b] : tree[node]) { ds.union_sets(a, b); }
  if (l == r) {
    cout << ds.component << '\n';
  } else {
    dfs(LF, l, mid);
    dfs(RT, mid + 1, r);
  }
  ds.RollBack();
}
signed main() {
  int n, m;
  cin >> n >> m;
  if (m == 0) return 0;
  map<pair<int, int>, int> query;  // a , b , time
  vector<array<int, 4>> add;       // a , b , start , end
  ds = DSU(n);
  char t;
  int a, b, Q = 0;
  for (int i = 0; i < m; ++i) {
    cin >> t;
    if (t == '?') {
      Q++;
    } else if (t == '-') {
      cin >> a >> b;
      if (a > b) swap(a, b);
      int start = query[{a, b}];
      if (start < Q) add.push_back({a, b, start, Q - 1});
      query.erase({a, b});
    } else {
      cin >> a >> b;
      if (a > b) swap(a, b);
      query[{a, b}] = Q;
    }
  }
  for (auto &[x, start] : query) { update(0, 0, Q - 1, start, Q - 1, x); }
  for (auto &[u, v, start, end] : add) { update(0, 0, Q - 1, start, end,
  ↪ {u, v}); }
  dfs(0, 0, Q - 1);
  return 0;
}
```

dynamic merge sort tree

```
/*
 * You have an array a of size N and two types of query to process.
 * 1. change the value at position x to v.
 * 2. count number of element less than k from l to r
 * */
#include <ext/pb_ds/assoc_container.hpp>
```

```
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
template <class T>
using ordered_set =
    tree<T, null_type, less_equal<T>, rb_tree_tag,
    ↪  tree_order_statistics_node_update>;
// order_of_key(k): Gives the count of elements smaller than k. - O(log n)
// find_by_order(k): Returns the iterator for the kth element (use k = 0 for
↪  the
// first element). - O(log n)0
//  9
struct Node {
  ordered_set<int> v;
  Node(){};
  Node(int x) { v.insert(x); };
  void erase(int x) { v.erase(v.find_by_order(v.order_of_key(x))); }
};
struct MergeTree {
#define LNodeIDX (2 * node + 1)
#define RNodeIDX (2 * node + 2)
#define mid ((l + r) / 2)
  vector<Node> Tree;
  int SegSize = 1;
  Node defVal = 0;

  Node Merge(Node &a, Node &b) {
    Node res;
    for (auto i : a.v) res.v.insert(i);
    for (auto i : b.v) res.v.insert(i);
    return res;
  }

  void build(int l, int r, int node, vector<int> &a) {
    if (l == r) {
      if (l < a.size()) Tree[node] = a[l];
      return;
    }
    build(l, mid, LNodeIDX, a);
    build(mid + 1, r, RNodeIDX, a);
    Tree[node] = Merge(Tree[LNodeIDX], Tree[RNodeIDX]);
  }
```

```
  void update(int l, int r, int node, int idx, int oldV,
              int newV) {  // O(log^2(n))
    Tree[node].erase(oldV);
    Tree[node].v.insert(newV);
    if (l == r) return;
    if (idx <= mid) update(l, mid, LNodeIDX, idx, oldV, newV);
    else update(mid + 1, r, RNodeIDX, idx, oldV, newV);
  }

  int query(int l, int r, int node, int Lx, int Rx, int v) {  // O(log^2(n))
    if (Lx > r or Rx < l) return 0;
    if (Lx <= l and r <= Rx) return (int)Tree[node].v.order_of_key(v + 1);
    int L = query(l, mid, LNodeIDX, Lx, Rx, v);
    int R = query(mid + 1, r, RNodeIDX, Lx, Rx, v);
    return L + R;
  }
  public:

  MergeTree(vector<int> &a) {
    int n = (int)a.size();
    while (SegSize < n) SegSize *= 2;
    Tree.assign(2 * SegSize, defVal);
    build(0, SegSize - 1, 0, a);
  }

  void update(int idx, int oldV, int newV) { update(0, SegSize - 1, 0, idx,
  ↪  oldV, newV); }

  int query(int l, int r, int v) { return query(0, SegSize - 1, 0, l, r,
  ↪  v); }
};
```

```cpp
signed main() {
  int n, q;
  cin >> n >> q;
  vector<int> a(n);
  for (int i = 0; i < n; ++i) cin >> a[i];
  MergeTree tree(a);
  while (q--) {
    char t;
    int l, r;
    cin >> t >> l >> r, --l;
    if (t == '!') {
      tree.update(l, a[l], r);
      a[l] = r;
    } else {
      int v;
      cin >> v;
      --r;
      cout << tree.query(l, r, v) << '\n';
    }
  }
  return 0;
}
```

Hash Treap

```cpp
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
const int N = 2e5 + 5, inf = 1e18, M = 1;
const int mod[M] = {1000000009}, B[M] = {351};
int power[M][N];
struct Node {
  char val;
  int size = 1, priority = rng();
  array<int, M> hash{}, rev_hash{};
  bool reverse = false, replace_flag = false;
  char replace_val = 0;
  Node *left = nullptr, *right = nullptr;
  Node(char _val) : val(_val) {
    int x = _val;
    for (int i = 0; i < M; ++i) hash[i] = rev_hash[i] = x;
  }
};
using TreapNode = Node *;
struct Treap {
TreapNode root = nullptr;
~Treap() { clear(root); }
void clear(TreapNode t) {
  if (!t) return;
  clear(t->left);
  clear(t->right);
  delete t;
}
int size(TreapNode t) { return t ? t->size : 0; }
array<int, M> get_hash(TreapNode t) { return t ? t->hash : array<int, M>{}; }
array<int, M> get_rev_hash(TreapNode t) { return t ? t->rev_hash : array<int, M>{}; }
void apply_replace(TreapNode t) {
  if (!t || !t->replace_flag) return;
  propagate(t->left);
  propagate(t->right);
  int c = t->replace_val;
  for (int i = 0; i < M; ++i) {
    int mod_val = mod[i];
    t->hash[i] = t->rev_hash[i] =
        (c * (power[i][t->size] - 1 + mod_val) % mod_val * ((mod_val + 1) / 2)) %
        mod_val;
  }
  t->val = t->replace_val;
  if (t->left) {
    t->left->replace_flag = true;
    t->left->replace_val = t->replace_val;
  }
  if (t->right) {
    t->right->replace_flag = true;
    t->right->replace_val = t->replace_val;
  }
  t->replace_flag = false;
}
void propagate(TreapNode t) {
  if (!t) return;
  apply_replace(t);
  if (t->reverse) {
    swap(t->left, t->right);
    swap(t->hash, t->rev_hash);
    if (t->left) t->left->reverse ^= true;
    if (t->right) t->right->reverse ^= true;
    t->reverse = false;
  }
}
void update(TreapNode t) {
  if (!t) return;
  propagate(t->left);
  propagate(t->right);
  int lsz = size(t->left), rsz = size(t->right);
  t->size = lsz + rsz + 1;
  for (int i = 0; i < M; ++i) {
    int mod_val = mod[i];
    int x = t->val;
    int lh = t->left ? t->left->hash[i] : 0;
    int rh = t->right ? t->right->hash[i] : 0;
    int lr = t->left ? t->left->rev_hash[i] : 0;
    int rr = t->right ? t->right->rev_hash[i] : 0;
    t->hash[i] =
        (lh + x * power[i][lsz] % mod_val + rh * power[i][lsz + 1] %
        ↪   mod_val) % mod_val;
    t->rev_hash[i] =
        (rr + x * power[i][rsz] % mod_val + lr * power[i][rsz + 1] %
        ↪   mod_val) % mod_val;
  }
}
void split(TreapNode t, TreapNode &l, TreapNode &r, int k, int add = 0) {
  if (!t) return void(l = r = nullptr);
  propagate(t);
  int idx = add + size(t->left);
  if (idx <= k) split(t->right, t->right, r, k, idx + 1), l = t;
  else split(t->left, l, t->left, k, add), r = t;
  update(t);
}
void merge(TreapNode &t, TreapNode l, TreapNode r) {
  propagate(l);
  propagate(r);
  if (!l || !r) t = l ? l : r;
  else if (l->priority > r->priority) merge(l->right, l->right, r), t = l;
  else merge(r->left, l, r->left), t = r;
  update(t);
}
void insert(int pos, char c) {
  TreapNode L, R;
  TreapNode mid = new Node(c);
  split(root, L, R, pos - 1);
  merge(L, L, mid);
  merge(root, L, R);
}
void delete_at(int pos) {
  if (pos < 0 || pos >= size(root)) return;
  TreapNode L, Mid, R;
  split(root, L, R, pos - 1);  // L = [0..pos-1], R = [pos..end]
  split(R, Mid, R, 0);         // Mid = [pos], R = [pos+1..end]
  delete Mid;
  merge(root, L, R);
}
void cyclic_shift(int l, int r, int k, bool left_shift = false) {
  if (l > r || l < 0 || r >= size(root)) return;
  k %= (r - l + 1);
  if (k == 0) return;
  TreapNode L, R, Mid, F, S;
  split(root, L, R, l - 1);
  split(R, Mid, R, r - l);
  if (!left_shift) {
    split(Mid, F, S, (r - l + 1) - k);
    merge(Mid, S, F);
  } else {
    split(Mid, F, S, k);
    merge(Mid, S, F);
  }
  merge(R, Mid, R);
  merge(root, L, R);
}
void cut_insert(int i, int j, int k) {
  TreapNode L, Mid, R, T;
  split(root, L, R, i - 1);
  split(R, Mid, R, j - i);
  merge(T, L, R);
  split(T, L, R, k - 1);
  merge(L, L, Mid);
  merge(root, L, R);
}
void reverse(int l, int r) {
  TreapNode L, R, Mid;
  split(root, L, R, l - 1);
  split(R, Mid, R, r - l);
  if (Mid) Mid->reverse ^= true;
  merge(R, Mid, R);
```

```
    merge(root, L, R);
  }
  array<int, M> getHash(int l, int r) {
    if (l > r || l < 0 || r >= size(root)) return array<int, M>{};
    TreapNode L, Mid, R;
    split(root, L, R, l - 1);  // L = [0..l-1], R = [l..end]
    split(R, Mid, R, r - l);   // Mid = [l..r], R = [r+1..end]
    propagate(Mid);
    array<int, M> res = get_hash(Mid);
    merge(R, Mid, R);
    merge(root, L, R);
    return res;
  }
  array<int, M> getReverseHash(int l, int r) {
    if (l > r || l < 0 || r >= size(root)) return array<int, M>{};
    TreapNode L, Mid, R;
    split(root, L, R, l - 1);
    split(R, Mid, R, r - l);
    propagate(Mid);
    array<int, M> res = get_rev_hash(Mid);
    merge(R, Mid, R);
    merge(root, L, R);
    return res;
  }
  void print(TreapNode t) {
    if (!t) return;
    propagate(t);
    print(t->left);
    cout << t->val;
    print(t->right);
  }
  bool is_palindrome(int l, int r) {
    TreapNode L, R, Mid;
    split(root, L, R, l - 1);
    split(R, Mid, R, r - l);
    propagate(Mid);
    bool ok = true;
    for (int i = 0; i < M; ++i) ok &= (Mid->hash[i] == Mid->rev_hash[i]);
    merge(R, Mid, R);
    merge(root, L, R);
    return ok;
  }
};
void precompute() {
  for (int i = 0; i < M; ++i) {
    power[i][0] = 1;
    for (int j = 1; j < N; ++j) power[i][j] = power[i][j - 1] * B[i] %
    ↪  mod[i];
  }
}
vector<int> linear_sieve(int n) {
  vector<int> lp(n + 1);
  vector<int> pr;
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = i;
      pr.push_back(i);
    }
    for (int j = 0; i * pr[j] <= n; ++j) {
      lp[i * pr[j]] = pr[j];
      if (pr[j] == lp[i]) { break; }
    }
  }
  return lp;
}
void solve() {
  int n, q;
  string s;
  cin >> n >> q >> s;
  Treap treap;
  for (int i = 0; i < n; ++i) treap.insert(i, s[i]);
  vector<vector<int>> divs(n + 1);
  for (int i = 1; i <= n; ++i)
    for (int j = i; j <= n; j += i) divs[j].emplace_back(i);
  auto lp = linear_sieve(s.size());
  auto minPeriod = [&](int l, int r) -> long long {
    if ((l == r) || treap.getHash(l, r - 1) == treap.getHash(l + 1, r))
    ↪  return 1;
    int len = (r - l + 1), ans = len;
    while (len > 1) {
      if (treap.getHash(l, r - ans / lp[len]) == treap.getHash(l + ans /
      ↪  lp[len], r))
        ans /= lp[len];
      len /= lp[len];
    }
    return ans;
  };
  set<array<int, M>> st;
  while (q--) {
    int l, r;
    cin >> l >> r, --l, --r;
    int minP = minPeriod(l, r);
    auto hash = treap.getHash(l, l + minP - 1);
    if (st.count(hash)) continue;
    st.insert(hash);
    treap.reverse(l, r);
  }
  treap.print(treap.root);
}
//  precompute();
```

LazySegmentTree

```
/*
 * Your task is to maintain an array of n values and efficiently process the
 * following types of queries:
 * 1. Increase each value in range [a,b] by x.
 * 2. Set each value in range [a,b] to x.
 * 3. Calculate the sum of values in range [a,b].
 * */
struct Node {
  int v = 0, lazyAdd = 0, lazyAssign = 0;
  int isLazyAdd = 0, isLazyAssign = 0;
  Node(){};
  Node(int x) : v(x){};
  void add(int val, int l, int r) {
    v += val * (r - l + 1);
    lazyAdd += val;
    isLazyAdd = 1;
  }
  void assign(int val, int l, int r) {
    v = val * (r - l + 1);
    lazyAssign = val;
    isLazyAssign = 1;
    lazyAdd = 0;
    isLazyAdd = 0;
  }
};
struct LazySegmentTree {
#define LNodeIDX (2 * node + 1)
#define RNodeIDX (2 * node + 2)
#define mid ((l + r) / 2)
  private:
  vector<Node> Tree;
  int SegSize = 1;
  Node defVal = 0;
  Node merge(Node &a, Node &b) {
    Node res;
    res.v = a.v + b.v;
    return res;
  }
  void build(int l, int r, int node, vector<int> &a) {
    if (l == r) {
      if (l < a.size()) Tree[node] = Node(a[l]);
      return;
    }
    build(l, mid, LNodeIDX, a);
    build(mid + 1, r, RNodeIDX, a);
    Tree[node] = merge(Tree[LNodeIDX], Tree[RNodeIDX]);
  }
  void propagate(int node, int l, int r) {
    if (l == r) return;
    if (Tree[node].isLazyAssign) {
      Tree[LNodeIDX].assign(Tree[node].lazyAssign, l, mid);
      Tree[RNodeIDX].assign(Tree[node].lazyAssign, mid + 1, r);
    }
    if (Tree[node].isLazyAdd) {
      Tree[LNodeIDX].add(Tree[node].lazyAdd, l, mid);
      Tree[RNodeIDX].add(Tree[node].lazyAdd, mid + 1, r);
    }
    Tree[node].isLazyAdd = Tree[node].isLazyAssign = 0;
    Tree[node].lazyAdd = Tree[node].lazyAssign = 0;
  }
  void update(int l, int r, int node, int Lx, int Rx, int val, int t) {
    propagate(node, l, r);
    if (Lx > r or Rx < l) return;
    if (Lx <= l and r <= Rx) {
      if (t == 1) {
        Tree[node].isLazyAdd = 1;
        Tree[node].add(val, l, r);
      } else {
        Tree[node].isLazyAssign = 1;
        Tree[node].assign(val, l, r);
      }
      return;
    }
    update(l, mid, LNodeIDX, Lx, Rx, val, t);
    update(mid + 1, r, RNodeIDX, Lx, Rx, val, t);
    Tree[node] = merge(Tree[LNodeIDX], Tree[RNodeIDX]);
  }
  Node query(int l, int r, int node, int Lx, int Rx) {
    propagate(node, l, r);
    if (Lx > r or Rx < l) return defVal;
    if (Lx <= l and r <= Rx) return Tree[node];
    Node L = query(l, mid, LNodeIDX, Lx, Rx);
    Node R = query(mid + 1, r, RNodeIDX, Lx, Rx);
    return merge(L, R);
  }
  public:
  LazySegmentTree(int n) {
    while (SegSize < n) SegSize *= 2;
    Tree.assign(2 * SegSize, defVal);
  }
  void build(vector<int> &a) { build(0, SegSize - 1, 0, a); }
  void update(int l, int r, int val, int t) { update(0, SegSize - 1, 0, l,
  ↪  r, val, t); }
  Node query(int l, int r) { return query(0, SegSize - 1, 0, l, r); }
};
```

MO

```
struct MO {
  vector<int> v, frq;
  int B = 0, n, ans = 0;
  MO(vector<int> &a) {
    v = a;
    n = (int)a.size();
    B = sqrt(n) + 1;
    frq.assign(n + 1, {});
  }
  void add(int idx) {}
  void erase(int idx) {}
  vector<int> done(vector<array<int, 3>> &query) {
    sort(query.begin(), query.end(), [&](array<int, 3> a, array<int, 3> b) {
      return make_pair(a[0] / B, a[1]) < make_pair(b[0] / B, b[1]);
    });
    vector<int> ret(query.size());
    int l = query[0][0], r = l;
    add(l);
    for (const auto [lq, rq, idx] : query) {
      while (lq < l) --l, add(l);
      while (rq > r) ++r, add(r);
      while (lq > l) erase(l), ++l;
      while (rq < r) erase(r), --r;
      ret[idx] = ans;
    }
    return ret;
  }
};
// Mo optimizations
int block = 0;

struct Query {
  int l, r, idx;
  Query(int L, int R, int i) { l = L, r = R, idx = i; }
  inline pair<int, int> toPair() const {
    return make_pair(l / block, ((l / block) & 1) ? -r : +r);
  }
};

inline int64_t hilbertOrder(int x, int y, int pow, int rotate) {
  if (pow == 0) { return 0; }
  int hpow = 1 << (pow - 1);
  int seg = (x < hpow) ? ((y < hpow) ? 0 : 3) : ((y < hpow) ? 1 : 2);
  seg = (seg + rotate) & 3;
  const int rotateDelta[4] = {3, 0, 0, 1};
  int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
  int nrot = (rotate + rotateDelta[seg]) & 3;
  int64_t subSquareSize = int64_t(1) << (2 * pow - 2);
  int64_t ans = seg * subSquareSize;
  int64_t add = hilbertOrder(nx, ny, pow - 1, nrot);
  ans += (seg == 1 || seg == 2) ? add : (subSquareSize - add - 1);
  return ans;
}
struct Query {
  int l, r, idx;
  int64_t ord;  // Hilbert order value
  Query(int ll, int rr, int iidx) {
    l = ll, r = rr, idx = iidx;
    ord = hilbertOrder(l, r, 21, 0);
  }
};
bool operator<(const Query &a, const Query &b) { return a.ord < b.ord; }
```

MO RollBack

```
/*
 * You have an array a of size n and q queries , for each query
 * calculate the maximum distance between two same numbers from l to r
 * */
struct MO_RollBack {
  vector<int> v, L, R;
  int n = 0, B = 0, ans = 0;
  MO_RollBack(vector<int> &a) {
    v = a;
    n = (int)a.size();
    B = (int)sqrt(n) + 2;
    L.assign(n + 1, -1);
    R.assign(n + 1, -1);
  }
  void add(int idx) {
    int x = v[idx];
    if (L[x] + R[x] == -2) {
      L[x] = R[x] = idx;
    } else if (R[x] < idx) {
      R[x] = idx;
      ans = max(ans, R[x] - L[x]);
    } else {
      L[x] = idx;
      ans = max(ans, R[x] - L[x]);
    }
  }
  int get() { return ans; }
  void Process(vector<array<int, 3>> query[], vector<int> &res) {
    for (int i = 0; i < B; ++i) {  // answer every block
      if (query[i].empty()) continue;
      sort(query[i].begin(), query[i].end());
      L.assign(n + 1, -1);
      R.assign(n + 1, -1);
      ans = 0;
      int l = (i + 1) * B - 1, r = l;
      add(r);
      for (const auto [rq, lq, idx] : query[i]) {
        while (r < rq) {
          ++r;
          add(r);
        }
        int tmp_ans = ans;              // save current answer
        vector<array<int, 3>> updates;  // save all changes to rollback
        while (lq < l) {
          --l;
          updates.push_back({v[l], L[v[l]], R[v[l]]});
          add(l);
        }
        res[idx] = get();
        ans = tmp_ans;  // return to the last results
        reverse(updates.begin(),
                updates.end());  // start from the last change to the first
                ↪  change
        for (auto [x, lst_L, lst_R] : updates) {
          L[x] = lst_L;
          R[x] = lst_R;
        }
        l = (i + 1) * B - 1;
      }
    }
  }
};
int main() {
  int n, m, q;
  cin >> n >> m >> q;
  vector<int> a(n);
  for (int i = 0; i < n; ++i) { cin >> a[i]; }
  int B = sqrt(n) + 2;
  MO_RollBack mo(a);
  vector<int> res(q);
  vector<array<int, 3>> query[B];
  for (int i = 0; i < q; ++i) {
    int l, r;
    cin >> l >> r;
    --l, --r;
    if (r - l + 1 < B) {
      for (int j = l; j <= r; ++j) {
        mo.L[a[j]] = -1;
        mo.R[a[j]] = -1;
      }
      for (int j = l; j <= r; ++j) mo.add(j);
      res[i] = mo.get();
      mo.ans = 0;
      continue;
    }
    query[l / B].push_back({r, l, i});
  }
  mo.Process(query, res);
  for (auto i : res) cout << i << '\n';
  return 0;
}
```

OrderedSet

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <class T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag,
    ↪  tree_order_statistics_node_update>;
// order_of_key(k): Gives the count of elements smaller than k. - O(log n)
// find_by_order(k): Returns the iterator for the kth element (use k = 0 for
↪  the
// first element). - O(log n)O
```

powerful merge sort tree

```
struct Node {
  int v;
  Node(){};
  Node(int x) { v = x; };
};
struct SegmentTree {
#define LNodeIDX (2 * node + 1)
#define RNodeIDX (2 * node + 2)
#define mid ((l + r) / 2)
  vector<Node> Tree;
  int SegSize = 1, size = 0;
  Node defVal = 0;
  Node Merge(Node &a, Node &b) {
    Node res;
    res.v = a.v + b.v;
    return res;
  }
  void update(int l, int r, int node, int idx, int val) {
    if (l == r) {
      Tree[node].v = val;
      return;
    }
    if (idx <= mid) {
      update(l, mid, LNodeIDX, idx, val);
    } else {
      update(mid + 1, r, RNodeIDX, idx, val);
    }
    Tree[node] = Merge(Tree[LNodeIDX], Tree[RNodeIDX]);
  }
  Node query(int l, int r, int node, int Lx, int Rx) {
    if (Lx > r or Rx < l) { return defVal; }
    if (Lx <= l and r <= Rx) { return Tree[node]; }
    Node L = query(l, mid, LNodeIDX, Lx, Rx);
    Node R = query(mid + 1, r, RNodeIDX, Lx, Rx);
    return Merge(L, R);
  }
  public:
  SegmentTree(int n) {
    while (SegSize < n) SegSize *= 2;
    size = n;
    Tree.assign(2 * SegSize, defVal);
  }
  void update(int idx, int val) { update(0, SegSize - 1, 0, idx, val); }
  Node query(int l, int r) { return query(0, SegSize - 1, 0, l, r); }
};
struct MergeTree {
#define LNodeIDX (2 * node + 1)
#define RNodeIDX (2 * node + 2)
#define mid ((l + r) / 2)
  vector<SegmentTree> Tree;
  int SegSize = 1;
  Node defVal = 0;
  SegmentTree Merge(SegmentTree &a, SegmentTree &b) {
    SegmentTree res(a.size + b.size);
    int i = 0, j = 0, idx = 0;
    while (i < a.size and j < b.size) {
      int x = a.query(i, i).v, y = b.query(j, j).v;
      if (x <= y) res.update(idx++, x), ++i;
      else res.update(idx++, y), ++j;
    }
    while (i < a.size) res.update(idx++, a.query(i, i).v), ++i;
    while (j < b.size) res.update(idx++, b.query(j, j).v), ++j;
    return res;
  }
  void build(int l, int r, int node, vector<int> &a) {
```

```
  if (l == r) {
    if (l < a.size()) {
      Tree[node] = SegmentTree(1);
      Tree[node].update(0, a[l]);
    }
    return;
  }
  build(l, mid, LNodeIDX, a);
  build(mid + 1, r, RNodeIDX, a);
  Tree[node] = SegmentTree(Tree[LNodeIDX].size + Tree[RNodeIDX].size);
  Tree[node] = Merge(Tree[LNodeIDX], Tree[RNodeIDX]);
}
int getSum(int node, int k) {
  int l = 0, r = Tree[node].size - 1, md, ans = -1;
  while (l <= r) {
    md = (l + r) / 2;
    if (Tree[node].query(md, md).v <= k) {
      ans = md;
      l = md + 1;
    } else r = md - 1;
  }
  int res = 0;
  if (ans != -1) { res = Tree[node].query(0, ans).v; }
  return res;
}
int query(int l, int r, int node, int Lx, int Rx, int k) {
  if (Lx > r or Rx < l) return 0;
  if (Lx <= l and r <= Rx) {
    int x = getSum(node, k);
    return x;
  }
  int L = query(l, mid, LNodeIDX, Lx, Rx, k);
  int R = query(mid + 1, r, RNodeIDX, Lx, Rx, k);
  return L + R;
}
public:
MergeTree(vector<int> &a) {
  int n = (int)a.size();
  while (SegSize < n) SegSize *= 2;
  Tree.assign(2 * SegSize, SegmentTree(0));
  build(0, SegSize - 1, 0, a);
}
int query(int l, int r, int k) { return query(0, SegSize - 1, 0, l, r,
↪  k); }
};
void solve() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i) { cin >> a[i]; }
  MergeTree tree(a);
  int q, B = 0;
  cin >> q;
  while (q--) {
    int l, r, k;
    cin >> l >> r >> k;
    l ^= B, r ^= B, k ^= B;
    --l, --r;
    B = tree.query(l, r, k);
    cout << B << '\n';
  }
}
signed main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
#ifndef ONLINE_JUDGE
  freopen("Input.txt", "r", stdin);
  freopen("Output.txt", "w", stdout);
#endif
  int test = 1;
  //    cin >> test;
  for (int i = 1; i <= test; ++i) { solve(); }
  return 0;
}
```

SegmentTree

```
/*
 * There is an array consisting of n integers. Some values of the array will
 ↪   be
 * updated, and after each update, your task is to report the maximum
 ↪   subarray
 * sum in the array.
 * */
struct Node {
  int Suffix = 0, Prefix = 0, maxV = 0, Sum = 0;
  Node(){};
  Node(int v) {
    Sum = v;
    Suffix = Prefix = maxV = max(v, 0LL);
  };
};
struct SegmentTree {
#define LNodeIDX (2 * node + 1)
#define RNodeIDX (2 * node + 2)
#define mid ((l + r) / 2)
  private:
  vector<Node> Seg;
  int SegSize = 1;
  Node difVal;
  Node Merge(Node a, Node b) {
    Node res;
    res.Sum = a.Sum + b.Sum;
    res.Prefix = max(a.Prefix, a.Sum + b.Prefix);
    res.Suffix = max(b.Suffix, b.Sum + a.Suffix);
    res.maxV = max({res.Sum, a.maxV, b.maxV, a.Suffix + b.Prefix});
    return res;
  }
  void build(int l, int r, int node, vector<int> &a) {
    if (l == r) {
      if (r < a.size()) Seg[node] = Node(a[r]);
      return;
    }
    build(l, mid, LNodeIDX, a);
    build(mid + 1, r, RNodeIDX, a);
    Seg[node] = Merge(Seg[LNodeIDX], Seg[RNodeIDX]);
  }
  void update(int l, int r, int node, int idx, int Value) {
    if (l == r) {
      Seg[node] = Node(Value);
      return;
    }
    if (mid >= idx) update(l, mid, LNodeIDX, idx, Value);
    else update(mid + 1, r, RNodeIDX, idx, Value);
    Seg[node] = Merge(Seg[LNodeIDX], Seg[RNodeIDX]);
  }
  Node query(int l, int r, int node, int Lx, int Rx) {
    if (l > Rx or r < Lx) return difVal;
    if (l >= Lx and r <= Rx) return Seg[node];
    Node Left = query(l, mid, LNodeIDX, Lx, Rx);
    Node Right = query(mid + 1, r, RNodeIDX, Lx, Rx);
    return Merge(Left, Right);
  }
  public:
  SegmentTree(vector<int> &a) {
    int n = (int)a.size();
    SegSize = 1;
    while (SegSize < n) SegSize *= 2;
    Seg.assign(2 * SegSize, difVal);
    build(0, SegSize - 1, 0, a);
  }
  void update(int idx, int Value) { update(0, SegSize - 1, 0, idx, Value); }
  Node query(int l, int r) { return query(0, SegSize - 1, 0, l, r); }
#undef LNodeIDX
#undef RNodeIDX
#undef mid
};
```

SegmentTree2D

```
const int mod = 1e9 + 7;
struct SegmentTree2D {
  int rows, cols;
  vector<vector<int>> tree;
  SegmentTree2D(int n, int m) : rows(n), cols(m) {
    tree.assign(4 * rows, vector<int>(4 * cols, 0));
  }
  void build(vector<vector<int>> &data) { buildX(1, 0, rows - 1, data); }
  void buildX(int vx, int lx, int rx, const vector<vector<int>> &data) {
    if (lx != rx) {
      int mx = (lx + rx) / 2;
      buildX(vx * 2, lx, mx, data);
      buildX(vx * 2 + 1, mx + 1, rx, data);
    }
    buildY(vx, lx, rx, 1, 0, cols - 1, data);
  }
  void buildY(int vx, int lx, int rx, int vy, int ly, int ry,
              const vector<vector<int>> &data) {
    if (ly == ry) {
      if (lx == rx) tree[vx][vy] = data[lx][ly];
      else tree[vx][vy] = merge(tree[vx * 2][vy], tree[vx * 2 + 1][vy]);
    } else {
      int my = (ly + ry) / 2;
      buildY(vx, lx, rx, vy * 2, ly, my, data);
      buildY(vx, lx, rx, vy * 2 + 1, my + 1, ry, data);
    tree[vx][vy] = merge(tree[vx][vy * 2], tree[vx][vy * 2 + 1]);
  }
}
void update(int x, int y, int newValue) { updateX(1, 0, rows - 1, x, y,
↪   newValue); }
int query(int x1, int y1, int x2, int y2) {
  return queryX(1, 0, rows - 1, x1, x2, y1, y2);
}
private:
int merge(int a, int b) { return gcd(a, b); }
void updateY(int vx, int lx, int rx, int vy, int ly, int ry, int x, int y,
             int newValue) {
  if (ly == ry) {
    if (lx == rx) tree[vx][vy] = newValue;
    else tree[vx][vy] = merge(tree[vx * 2][vy], tree[vx * 2 + 1][vy]);
  } else {
    int my = (ly + ry) / 2;
    if (y <= my) updateY(vx, lx, rx, vy * 2, ly, my, x, y, newValue);
    else updateY(vx, lx, rx, vy * 2 + 1, my + 1, ry, x, y, newValue);
    tree[vx][vy] = merge(tree[vx][vy * 2], tree[vx][vy * 2 + 1]);
  }
}
void updateX(int vx, int lx, int rx, int x, int y, int newValue) {
  if (lx != rx) {
    int mx = (lx + rx) / 2;
    if (x <= mx) updateX(vx * 2, lx, mx, x, y, newValue);
    else updateX(vx * 2 + 1, mx + 1, rx, x, y, newValue);
  }
  updateY(vx, lx, rx, 1, 0, cols - 1, x, y, newValue);
}
int queryY(int vx, int vy, int tly, int try_, int ly, int ry) {
  if (ly > ry) return 0;
  if (ly == tly && try_ == ry) return tree[vx][vy];
  int tmy = (tly + try_) / 2;
  return merge(queryY(vx, vy * 2, tly, tmy, ly, min(ry, tmy)),
               queryY(vx, vy * 2 + 1, tmy + 1, try_, max(ly, tmy + 1),
               ↪   ry));
}
int queryX(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
  if (lx > rx) return 0;
  if (lx == tlx && trx == rx) return queryY(vx, 1, 0, cols - 1, ly, ry);
  int tmx = (tlx + trx) / 2;
  return merge(queryX(vx * 2, tlx, tmx, lx, min(rx, tmx), ly, ry),
               queryX(vx * 2 + 1, tmx + 1, trx, max(lx, tmx + 1), rx, ly,
               ↪   ry));
  }
};
void solve() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> grid(n, vector<int>(m));
  for (int i = 0; i < n; ++i)
    for (int j = 0, x; j < m; ++j) cin >> grid[i][j];
  SegmentTree2D seg(n, m);
  seg.build(grid);
  int answer = 1;
  int q;
  cin >> q;
  while (q--) {
    int x, y, k;
    cin >> x >> y >> k;
    --x, --y;
    int g = seg.query(x, y, x + k - 1, y + k - 1);
    answer = 1LL * answer * g % mod;
  }
  cout << answer << '\n';
}
```

Simple DSU

```
struct DSU {
  vector<int> parent, size;
  DSU(int n) {
    parent.assign(n + 1, {});
    size.assign(n + 1, 1);
    iota(parent.begin(), parent.end(), 0);
  }
  int find(int x) {
    if (x == parent[x]) return x;
    return parent[x] = find(parent[x]);
  }
  bool union_set(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
    return true;
  }
  bool isConnected(int a, int b) { return find(a) == find(b); }
};
```

SparseTable

```
template <typename T, class CMP = function<T(const T &, const T &)>>
class SparseTable {
  public:
  int n;
  vector<vector<T>> sp;
  CMP func;
  SparseTable(const vector<T> &a, const CMP &f) : func(f) {
    n = a.size();
    int max_log = 32 - __builtin_clz(n);
    sp.resize(max_log);
    sp[0] = a;
    for (int j = 1; j < max_log; ++j) {
      sp[j].resize(n - (1 << j) + 1);
      for (int i = 0; i + (1 << j) <= n; ++i) {
        sp[j][i] = func(sp[j - 1][i], sp[j - 1][i + (1 << (j - 1))]);
      }
    }
  }
  T get(int l, int r) const {
    int lg = __lg(r - l + 1);
    return func(sp[lg][l], sp[lg][r - (1 << lg) + 1]);
  }
};
/*
  SparseTable sp(a, [](int a, int b) {
    return max(a, b);
  });
 */
```

static merge sort tree

```
/*
 * You have an array a of size n and q queries
 * for each query calculate number of elements grater than k from l to r
 * */
struct Node {
  vector<int> v;
  Node(){};
  Node(int x) { v.push_back(x); };
};
struct MergeTree {
#define LNodeIDX (2 * node + 1)
#define RNodeIDX (2 * node + 2)
#define mid ((l + r) / 2)
#define all(v) v.begin(), v.end()
  vector<Node> Tree;
  int SegSize = 1;
  Node defVal = 0;
  Node Merge(Node &a, Node &b) {
    Node res;
    res.v.assign(a.v.size() + b.v.size(), {});
    merge(all(a.v), all(b.v), res.v.begin());
    return res;
  }
  void build(int l, int r, int node, vector<int> &a) {
    if (l == r) {
      if (l < a.size()) Tree[node] = a[l];
      return;
    }
    build(l, mid, LNodeIDX, a);
    build(mid + 1, r, RNodeIDX, a);
    Tree[node] = Merge(Tree[LNodeIDX], Tree[RNodeIDX]);
  }
  int query(int l, int r, int node, int Lx, int Rx, int k) {
    if (Lx > r or Rx < l) return 0;
    if (Lx <= l and r <= Rx) {
      int g =
          upper_bound(Tree[node].v.begin(), Tree[node].v.end(), k) -
          ↪  Tree[node].v.begin();
      return Tree[node].v.size() - g;
    }
    int L = query(l, mid, LNodeIDX, Lx, Rx, k);
    int R = query(mid + 1, r, RNodeIDX, Lx, Rx, k);
    return L + R;
  }
  public:
  MergeTree(vector<int> &a) {
    int n = (int)a.size();
    while (SegSize < n) SegSize *= 2;
    Tree.assign(2 * SegSize, defVal);
    build(0, SegSize - 1, 0, a);
  }
  int query(int l, int r, int k) { return query(0, SegSize - 1, 0, l, r,
   ↪  k); }
};
```

Treap

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
// Beware!!!here Treap is 0-indexed
struct Node {
  int val = 0, size = 0, priority = 0, lazy = 0, max_val = 0;
  int min_val = 0, replace = 0, sum = 0;
  bool replace_flag = false, reverse = false;
  Node *left = nullptr, *right = nullptr, *parent = nullptr;
  Node() = default;
  explicit Node(int _val) {
    val = sum = max_val = min_val = _val;
    size = 1;
    priority = rnd();
    left = right = parent = nullptr;
  }
};
typedef Node *TreapNode;
struct Treap {
  TreapNode root{};
  map<int, TreapNode> position;  // positions of all the values
  bool exist = false;
  void clear() {
    delete_nodes(root);
    root = nullptr;
    position.clear();
    exist = false;
  }
  Treap() { clear(); }
  static int size(TreapNode t) { return t ? t->size : 0; }
  static void update_size(TreapNode &t) {
    if (t) t->size = size(t->left) + size(t->right) + 1;
  }
  static void update_parent(TreapNode &t) {
    if (!t) return;
    if (t->left) t->left->parent = t;
    if (t->right) t->right->parent = t;
  }
static void update_sum(TreapNode &t) {
  if (!t or !t->lazy) return;
  t->sum += t->lazy * size(t);
  t->val += t->lazy;
  t->max_val += t->lazy;
  t->min_val += t->lazy;
  if (t->left) t->left->lazy += t->lazy;
  if (t->right) t->right->lazy += t->lazy;
  t->lazy = 0;
}
// replace update
static void apply_replace(TreapNode &t) {
  if (!t or !t->replace_flag) return;
  t->val = t->max_val = t->min_val = t->replace;
  t->sum = t->val * size(t);
  if (t->left) {
    t->left->replace = t->replace;
    t->left->replace_flag = true;
  }
  if (t->right) {
    t->right->replace = t->replace;
    t->right->replace_flag = true;
  }
  t->replace_flag = false;
  t->replace = 0;
}
// reverse update
static void apply_reverse(TreapNode &t) {
  if (!t or !t->reverse) return;
  t->reverse = false;
  swap(t->left, t->right);
  if (t->left) t->left->reverse ^= true;
  if (t->right) t->right->reverse ^= true;
}
// reset the value of current node assuming it now
// represents a single element of the array
static void reset(TreapNode &t) {
  if (!t) return;
  t->sum = t->val;
  t->max_val = t->val;
  t->min_val = t->val;
}
// perform all operations
void operation(TreapNode &t) {
  if (!t) return;
  apply_reverse(t);
  apply_replace(t);
  update_sum(t);
  recalculate(t);
}
// split node t in l and r by key k
void split(TreapNode t, TreapNode &l, TreapNode &r, int k, int add = 0) {
  if (t == nullptr) {
    l = nullptr;
    r = nullptr;
    return;
  }
  operation(t);
  int idx = add + size(t->left);
  if (t->left) t->left->parent = nullptr;
  if (t->right) t->right->parent = nullptr;
  if (idx <= k) split(t->right, t->right, r, k, idx + 1), l = t;
  else split(t->left, l, t->left, k, add), r = t;
  update_parent(t);
  update_size(t);
  operation(t);
}
// merge node l with r in t
void merge(TreapNode &t, TreapNode l, TreapNode r) {
  operation(l);
  operation(r);
  if (!l) {
    t = r;
    return;
  }
  if (!r) {
    t = l;
    return;
  }
  if (l->priority > r->priority) merge(l->right, l->right, r), t = l;
  else merge(r->left, l, r->left), t = r;
  update_parent(t);
  update_size(t);
  operation(t);
}
void recalculate(TreapNode &t) {
  if (!t) return;
  t->size = 1 + size(t->left) + size(t->right);
```

```
  int lsum = t->left ? t->left->sum : 0;
  int rsum = t->right ? t->right->sum : 0;
  int lsize = size(t->left);
  if (lsize % 2 == 0) t->sum = lsum + t->val - rsum;
  else t->sum = lsum - t->val + rsum;
}
// returns index of node curr
int get_pos(TreapNode curr, TreapNode son = nullptr) {
  bool exists = true;
  if (curr == nullptr) {
    exists = false;
    return 0;
  }
  if (!son) {
    if (curr == root) return size(curr->left);
    else return size(curr->left) + get_pos(curr->parent, curr);
  }
  if (curr == root) {
    if (son == curr->left) return 0;
    else return size(curr->left) + 1;
  }
  if (curr->left == son) return get_pos(curr->parent, curr);
  else return get_pos(curr->parent, curr) + size(curr->left) + 1;
}
void delete_nodes(TreapNode t) {
  if (!t) return;
  delete_nodes(t->left);
  delete_nodes(t->right);
  position.erase(t->val);
  delete t;
}
// insert val in position a[pos]
void insert(int pos, int val) {
  if (root == nullptr) {
    auto to_add = new Node(val);
    root = to_add;
    position[val] = root;
    return;
  }
  TreapNode l, r, mid;
  mid = new Node(val);
  position[val] = mid;
  split(root, l, r, pos - 1);
```

```
    merge(l, l, mid);
    merge(root, l, r);
  }
  // erase from qL to qR indexes
  void erase(int qL, int qR) {
    TreapNode l, r, mid;
    split(root, l, r, qL - 1);
    split(r, mid, r, qR - qL);
    delete_nodes(mid);
    merge(root, l, r);
  }
  // returns answer for corresponding types of query [sum, max, min]
  int query(int qL, int qR) {
    TreapNode l, r, mid;
    split(root, l, r, qL - 1);
    split(r, mid, r, qR - qL);
    recalculate(root);
    int answer = mid->sum;
    merge(r, mid, r);
    merge(root, l, r);
    recalculate(root);
    return answer;
  }
  // add val in all the values from a[qL] to a[qR] positions
  void update(int qL, int qR, int val) {
    TreapNode l, r, mid;
    split(root, l, r, qL - 1);
    split(r, mid, r, qR - qL);
    mid->lazy += val;
    merge(r, mid, r);
    merge(root, l, r);
  }
  // reverse all the values from qL to qR
  void reverse(int qL, int qR) {
    TreapNode l, r, mid;
    split(root, l, r, qL - 1);
    split(r, mid, r, qR - qL);
    mid->reverse ^= 1;
    merge(r, mid, r);
    merge(root, l, r);
  }
  // replace all the values from a[qL] to a[qR] by v
  void replace(int qL, int qR, int v) {
    TreapNode l, r, mid;
    split(root, l, r, qL - 1);
    split(r, mid, r, qR - qL);
    mid->replace_flag = true;
    mid->replace = v;
    merge(r, mid, r);
    merge(root, l, r);
  }
  // it will cyclic right shift the array k times
  void cyclic_shift(int qL, int qR, int k, bool left_shift = false) {
    if (qL == qR) return;
    k %= (qR - qL + 1);
    TreapNode l, r, mid, fh, sh;
    split(root, l, r, qL - 1);
    split(r, mid, r, qR - qL);
    if (!left_shift) split(mid, fh, sh, (qR - qL + 1) - k - 1);
    else split(mid, fh, sh, k - 1);
    merge(mid, sh, fh);
    merge(r, mid, r);
    merge(root, l, r);
  }
  // returns index of the value
  int get_pos(int value) {
    if (position.find(value) == position.end()) return -1;
    int x = get_pos(position[value]);
    return x;
  }
  //  access index in the array
  int get_val(int pos) { return query(pos, pos); }
  int size() { return size(root); }
  bool find(int val) { return position.count(val) > 0; }
  void print(TreapNode t) {
    if (!t) return;
    propagate(t);
    print(t->left);
    cout << t->val;
    print(t->right);
  }
};
```

XOR Basis

```
const int lg = 64, mod = 1e9 + 7, inf = 1e9 + 7;
struct Basis {
    int size = 0, n = 0;
    int basis[lg];

    Basis() {
        size = n = 0;
        for (int i = lg - 1; i >= 0; --i) basis[i] = 0;
    }

    bool insert(int x) {
        n++;
        for (int i = lg - 1; i >= 0; --i) {
            if (((x >> i) & 1) == 0) continue;
            if (not basis[i]) {
                basis[i] = x, ++size;
                return true;
            }
            x = (x ^ basis[i]);
        }
        return false;
    }

    bool merge(Basis &w) {
        bool repeat = false;
        for (int i = 0; i < lg; ++i)
            if (w.basis[i] > 0 and not insert(w.basis[i])) repeat = true;
        return repeat;
    }

    bool can(int x) { // if n > size then you can get x = 0
        for (int i = lg - 1; i >= 0; --i)
            if (basis[i] and (x & (1LL << i))) x = (x ^ basis[i]);
        return x == 0;
    }

    int count_xors(int x) { // NOTE: Add exponentiation template.
        return (can(x) ? (exp(2, n - size) + mod - 1) % mod : 0);
    }
    int kth(int k) {
        int x = 0;
        for (int i = lg - 1, c = size; i >= 0; --i) {
            if (not basis[i]) continue;
            --c;
            if (x & (1LL << i)) {
                if ((1LL << c) >= k) x = (x ^ basis[i]);
                else k = k - (1LL << c);
            } else if (k > (1LL << c)) {
                x = (x ^ basis[i]), k = k - (1LL << c);
            }
        }
        return x;
    }

    int get_max() {
        int ans = 0;
        for (int i = lg - 1; i >= 0; --i) {
            if (basis[i] && not(ans & (1LL << i))) ans = (ans ^ basis[i]);
        }
        return ans;
    }

    void AND(int x) {
        vector< int > upd;
        for (int i = lg - 1; i >= 0; --i) {
            basis[i] = (basis[i] & x);
            if (basis[i]) upd.push_back(basis[i]);
            basis[i] = 0;
        }
        for (int &val: upd) insert(val);
    }

    void OR(int x) {
        vector< int > upd;
        for (int i = lg - 1; i >= 0; --i) {
            basis[i] = (basis[i] | x);
            if (basis[i]) upd.push_back(basis[i]);
            basis[i] = 0;
        }
        for (int &val: upd) insert(val);
    }
};
```

Graph

ArticulationPoints

```
class Graph {
  int V;
  vector<vector<int>> adj;
  vector<bool> visited;
  vector<int> tin, low;
  set<int> articulation_points;
  int timer;
  void dfs(int v, int parent) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children = 0;
    for (int to : adj[v]) {
      if (to == parent) continue;
      if (visited[to]) {
        low[v] = min(low[v], tin[to]);
      } else {
        dfs(to, v);
        low[v] = min(low[v], low[to]);
        if (low[to] >= tin[v] && parent != -1)
        ↪  articulation_points.insert(v);
        ++children;
      }
    }
    if (parent == -1 && children > 1) articulation_points.insert(v);
  }
  public:
  Graph(int V) {
    this->V = V;
    adj.resize(V);
    visited.assign(V, false);
    tin.resize(V, -1);
    low.resize(V, -1);
    timer = 0;
  }
  void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  void findArticulationPoints() {
    for (int i = 0; i < V; ++i) {
      if (!visited[i]) dfs(i, -1);
    }
    cout << "Articulation Points:\n";
    for (int point : articulation_points) cout << point << " ";
    cout << endl;
  }
};
int main() {
  int V = 5;
  Graph g(V);
  g.addEdge(0, 1), g.addEdge(0, 2), g.findArticulationPoints();
  return 0;
}
```

bellmanford

```
const ll oo = 1e18;
struct edge {
  ll x, y, cost;
};
vector<edge> edges;
vector<ll> dis;
vector<bool> in_negative_cycle;
vector<ll> v;
void bellman_ford(ll u) {
  dis.assign(n + 1, oo);
  in_negative_cycle.assign(n + 1, false);
  dis[u] = 0;
  for (int i = 0; i < n - 1; i++) {
    for (auto& e : edges) {
      if (dis[e.x] != oo && dis[e.x] + e.cost < dis[e.y]) {
        dis[e.y] = dis[e.x] + e.cost;
      }
    }
  }
  for (auto& e : edges) {
    if (dis[e.x] != oo && dis[e.x] + e.cost < dis[e.y]) {
    ↪  in_negative_cycle[e.y] = true; }
  }
  queue<ll> q;
  for (int i = 1; i <= n; i++) {
    if (in_negative_cycle[i]) q.push(i);
  }
  while (!q.empty()) {
    ll x = q.front();
    q.pop();
    for (auto e : edges) {
      if (e.x == x && !in_negative_cycle[e.y]) {
        in_negative_cycle[e.y] = true;
        q.push(e.y);
      }
    }
  }
}
```

bfs 0-1

```
struct edge {
  ll to, cost;
};
vector<ll> dis;
vector<vector<edge>> adj;
void BFS_0_1(ll n) {
  dis[n] = 0;
  deque<ll> d;
  d.push_back(n);
  while (!d.empty()) {
    ll u = d.front();
    d.pop_front();
    for (auto& x : adj[u]) {
      if (dis[u] + x.cost < dis[x.to]) {
        dis[x.to] = dis[u] + x.cost;
        if (x.cost == 1) d.push_back(x.to);
        else d.push_front(x.to);
      }
    }
  }
}
```

Dijkstra

```
vector<int> Dijkstra(int start, vector<vector<pair<int, int>>> &adj) {
  int n = (int)adj.size();
  vector<int> dist(n, inf);
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;  // dist , node
  pq.push({0, start});
  dist[start] = 0;
  while (!pq.empty()) {
    auto [W, u] = pq.top();
    pq.pop();
    if (dist[u] < W) continue;
    for (auto [node, Weight] : adj[u]) {
      int newW = W + Weight;
      if (newW < dist[node]) {
        dist[node] = newW;
        pq.push({newW, node});
      }
    }
  }
  return dist;
}
```

Dinic

```
// O(V^2 * E)
// O(E * Sqrt(V)) in maximum matching problem (Unit Networks)
static const int oo = 2e15;
struct Edge {
    int u, v, flow = 0, cap = 0; // keep the order
    Edge(int u, int v) : u(u), v(v) {}
    Edge(int u, int v, int c) : u(u), v(v), cap(c) {}
    int rem() { return cap - flow; }
};
struct Dinic {
    int n, s, t, id = 1, flow = 0;
    vector<Edge> edges;
    vector<vector<int>> adj;
    vector<int> lvl, ptr;
    Dinic(int n, int src, int sink) : n(n), s(src), t(sink) {
        adj.assign(n + 1, {});
        ptr.assign(n + 1, {});
    }
    void addEdge(int u, int v, int w = oo, int undir = 0) {
        adj[u].push_back(edges.size());
        edges.push_back(Edge(u, v, w));
        adj[v].push_back(edges.size());
        edges.push_back(Edge(v, u, w * undir));
    }
    void run() {
        while (bfs()) {
            ptr.assign(n + 1, {});
            while (int f = dfs(s)) flow += f;
        }
    }
    bool bfs() {
        lvl.assign(n + 1, -1);
        queue<int> q;
        q.push(s), lvl[s] = 0;
        while (!q.empty()) {
            auto u = q.front();
            q.pop();
            for (auto &i: adj[u]) {
                auto &[_, v, f, c] = edges[i];
                if (~lvl[v] || f == c) continue;
                lvl[v] = lvl[u] + 1;
                q.push(v);
            }
        }
        return lvl[t] != -1;
    }
    int dfs(int u, int currFlow = oo) {
        if (u == t) return currFlow;
        if (!currFlow) return 0;
        for (; ptr[u] < adj[u].size(); ++ptr[u]) {
            int i = adj[u][ptr[u]];
            auto [_, v, f, c] = edges[i];
            if (f == c || (lvl[v] != lvl[u] + 1)) continue;
            int bottleNeck = dfs(v, min(currFlow, c - f));
            if (!bottleNeck) continue;
            edges[i].flow += bottleNeck;
            edges[i ^ 1].flow -= bottleNeck;
            return bottleNeck;
        }
        return 0;
    }
};
void solve() {
    int n, m;
    cin >> n >> m;
    vector<Edge> adj;
    Dinic go(n, 1, n);
    for (int i = 0, u, v, w; i < m; ++i) {
        cin >> u >> v >> w;
        go.addEdge(u, v, w);
    }
    go.run();
    cout << go.flow << '\n';
}
```

EdmondKarp

```
// O(V^2 * E)
// O(E * Sqrt(V)) in maximum matching problem (Unit Networks)
static const int oo = 2e15;

struct Edge {
    int u, v, flow = 0, cap = 0; // keep the order
    Edge(int u, int v) : u(u), v(v) {}
    Edge(int u, int v, int c) : u(u), v(v), cap(c) {}
    int rem() { return cap - flow; }
};
struct Dinic {
    int n, s, t, id = 1, flow = 0;
    vector<Edge> edges;
    vector<vector<int>> adj;
    vector<int> lvl, ptr;

    Dinic(int n, int src, int sink) : n(n), s(src), t(sink) {
        adj.assign(n + 1, {});
        ptr.assign(n + 1, {});
    }

    void addEdge(int u, int v, int w = oo, int undir = 0) {
        adj[u].push_back(edges.size());
        edges.push_back(Edge(u, v, w));
        adj[v].push_back(edges.size());
        edges.push_back(Edge(v, u, w * undir));
    }

    void run() {
        while (bfs()) {
            ptr.assign(n + 1, {});
            while (int f = dfs(s)) flow += f;
        }
    }
    bool bfs() {
        lvl.assign(n + 1, -1);
        queue<int> q;
        q.push(s), lvl[s] = 0;
        while (!q.empty()) {
            auto u = q.front();
            q.pop();
            for (auto &i: adj[u]) {
                auto &[_, v, f, c] = edges[i];
                if (~lvl[v] || f == c) continue;
                lvl[v] = lvl[u] + 1;
                q.push(v);
            }
        }
        return lvl[t] != -1;
    }

    int dfs(int u, int currFlow = oo) {
        if (u == t) return currFlow;
        if (!currFlow) return 0;
        for (; ptr[u] < adj[u].size(); ++ptr[u]) {
            int i = adj[u][ptr[u]];
            auto [_, v, f, c] = edges[i];
            if (f == c || (lvl[v] != lvl[u] + 1)) continue;
            int bottleNeck = dfs(v, min(currFlow, c - f));
            if (!bottleNeck) continue;
            edges[i].flow += bottleNeck;
            edges[i ^ 1].flow -= bottleNeck;
            return bottleNeck;
        }
        return 0;
    }
};
void solve() {
    int n, m;
    cin >> n >> m;
    vector<Edge> adj;
    Dinic go(n, 1, n);
    for (int i = 0, u, v, w; i < m; ++i) {
        cin >> u >> v >> w;
        go.addEdge(u, v, w);
    }
    go.run();
    cout << go.flow << '\n';
}
```

floyd warshall

```
vector<vector<ll>> dis;
void floyd_warshall() {
  for (int i = 1; i <= n; i++) dis[i][i] = 0;
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        if (dis[i][k] < oo && dis[k][j] < oo)
          dis[i][j] = dis[j][i] = min(dis[i][j], dis[i][k] + dis[k][j]);
      }
    }
  }
}
```

Ford-Fulkerson

```
// O(E * MxFlow)
static const int oo = 2e15;

struct Edge {
    int u, v, flow = 0, cap = 0; // keep the order
    Edge(int u, int v): u(u), v(v) {}
    Edge(int u, int v, int c): u(u), v(v), cap(c) {}
    int rem() { return cap - flow; }
};
struct Ford {
    int n, s, t, id = 1;
    vector<Edge> edges;
    vector<vector<int>> adj;
    vector<int> vis;
    Ford(int n, int s, int t): n(n), s(s), t(t) {
        adj.assign(n + 1, {});
        vis.assign(n + 1, {});
    }
    // undir = 1 --> the edge is undirected

    void addEdge(int u, int v, int w = oo, int undir = 0) {
        adj[u].push_back(edges.size());
        edges.push_back(Edge(u, v, w));
        adj[v].push_back(edges.size());
        edges.push_back(Edge(v, u, w * undir));
    }
    int flow() {
        int res = 0, flow;
        while((flow = dfs(s))) res += flow, ++id;
        return res;
    }

    int dfs(int u, int flow = oo) {
        if(u == t) return flow;
        vis[u] = id;
        for(auto &i: adj[u]) {
            if(vis[edges[i].v] != id && edges[i].rem()) {
                int bottleNeck = dfs(edges[i].v, min(flow, edges[i].rem()));
                edges[i].flow += bottleNeck;
                edges[i ^ 1].flow -= bottleNeck;
                if(bottleNeck) return bottleNeck;
            }
        }
        return 0;
    }
};
void solve() {
    int n, m;
    cin >> n >> m;
    vector<Edge> adj;
    Ford go(n, 1, n);
    for (int i = 0, u, v, w; i < m; ++i) {
        cin >> u >> v >> w;
        go.addEdge(u, v, w);
    }
    cout << go.flow() << '\n';
}
```

Hopcroft Karp

```
struct HK {
  int n, m;
  vector<vector<int>> g;
  vector<int> l, r, d, p;
  int ans;
  HK(int n, int m) : n(n), m(m), g(n), l(n, -1), r(m, -1), ans(0) {}
  void add_edge(int u, int v) { g[u].push_back(v); }
  int match() {
    while (true) {
      queue<int> q;
      d.assign(n, -1);
      for (int i = 0; i < n; i++)
        if (l[i] == -1) q.push(i), d[i] = 0;
      while (!q.empty()) {
        int x = q.front();
        q.pop();
        for (int y : g[x])
          if (r[y] != -1 && d[r[y]] == -1) d[r[y]] = d[x] + 1, q.push(r[y]);
      }
      bool match = false;
      for (int i = 0; i < n; i++)
        if (l[i] == -1 && dfs(i)) ++ans, match = true;
      if (!match) break;
    }
    return ans;
  }
  bool dfs(int x) {
    for (int y : g[x])
      if (r[y] == -1 || (d[r[y]] == d[x] + 1 && dfs(r[y])))
        return l[x] = y, r[y] = x, d[x] = -1, true;
    return d[x] = -1, false;
  }
};
```

IsBipartite

```
int isBipartite(int n, vector<vector<int>>& graph) {
  int cnt = 0;
  vector<int> color(n + 1, -1);
  for (int start = 1; start <= n; ++start) {
    if (color[start] == -1) {
      cnt++;
      queue<int> q;
      q.push(start);
      color[start] = 0;
      while (!q.empty()) {
        int node = q.front();
        q.pop();
        for (int neighbor : graph[node]) {
          if (color[neighbor] == -1) {
            color[neighbor] = 1 - color[node];
            q.push(neighbor);
          } else if (color[neighbor] == color[node]) {
            return 0;
          }
        }
      }
    }
  }
  return cnt;
}
//// AbdelSame3 Code
// clockwise
int dx[8] = {-1, -1, 0, 1, 1, 1, 0, -1};
int dy[8] = {0, 1, 1, 1, 0, -1, -1, -1};
int dx[] = {0, 0, -1, 1, -1, 1, -1, 1};
int dy[] = {-1, 1, 0, 0, -1, -1, 1, 1};
vector<ll> co;
bool dfs_bipartite(ll n, ll c, ll p) {
  co[n] = c;
  ll ans = true;
  for (auto& x : adj[n]) {
    if (x == p) continue;
    if (co[x] == c) return false;
    if (co[x] == -1) ans &= dfs_bipartite(x, !c, n);
  }
  return ans;
}
```

Kruskal

```
struct DSU {
  vector<int> parent, size;
  int component;
  DSU(int n) {
    parent.assign(n + 1, {});
    size.assign(n + 1, 1);
    component = n;
    iota(parent.begin(), parent.end(), 0);
  }
  int find(int x) {
    if (x == parent[x]) return x;
    return parent[x] = find(parent[x]);
  }
  void union_sets(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
    component--;
  }
  bool isConnected(int a, int b) { return find(a) == find(b); }
};
int Kruskal(int n, vector<array<int, 3>> &edges) {
  sort(edges.begin(), edges.end());
  DSU ds(n);
  int sum = 0;
  for (auto &edge : edges) {
    if (!ds.isConnected(edge[1], edge[2])) {
      sum += edge[0];
      ds.union_sets(edge[1], edge[2]);
    }
  }
  return sum;
}
```

MCMF

```
static const int oo = 2e15;
struct Edge {
    int u, v, flow = 0, cap = 0, cost; // keep the order
    Edge(int u, int v, int c, int cost) : u(u), v(v), cap(c), cost(cost) {}
    int rem() { return cap - flow; }
};
struct MCMF {
    int n, s, t, cost = 0, flow = 0;
    vector<Edge> edges;
    vector<vector<int>> adj;
    vector<int> from;
    MCMF(int n, int s, int t) : n(n), s(s), t(t) {
        adj.assign(n + 1, {});
    }
    void addEdge(int u, int v, int w = oo, int cost = 0, int undir = 0) {
        adj[u].push_back(edges.size());
        edges.push_back(Edge(u, v, w, cost));
        adj[v].push_back(edges.size());
        edges.push_back(Edge(v, u, w * undir, -cost));
    }
    void run() {
        while (bfs()) {
            int u = t, addflow = oo;
            while (u != s) {
                Edge &e = edges[from[u]];
                addflow = min(addflow, e.rem());
                u = e.u;
            }
            u = t;
            while (u != s) {
                int i = from[u];
                edges[i].flow += addflow;
                edges[i ^ 1].flow -= addflow;
                cost += edges[i].cost * addflow;
                u = edges[i].u;
            }
            flow += addflow;
        }
    }
    bool bfs() {
        from.assign(n + 1, -1);
        vector<int> d(n + 1, oo), state(n + 1, 2);
        deque<int> q;
        state[s] = 1, d[s] = 0;
        q.clear();
        q.push_back(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop_front();
            state[u] = 0;
            for (auto &i: adj[u]) {
                auto &[_, v, f, c, cost] = edges[i];
                if (f >= c || d[v] <= d[u] + cost) continue;
                d[v] = d[u] + cost;
                from[v] = i;
                if (state[v] == 1) continue;
                if (!state[v] || (!q.empty() && d[q.front()] > d[v]))
                    q.push_front(v);
                else q.push_back(v);
                state[v] = 1;
            }
        }
        return ~from[t];
    }
};
void solve() {
    int n, m;
    cin >> n >> m;
    vector<Edge> adj;
    MCMF go(n, 1, n);
    for (int i = 0, u, v, w; i < m; ++i) {
        cin >> u >> v >> w;
        go.addEdge(u, v, w);
    }
    go.run();
    cout << go.flow << '\n';
}
```

OnlineBridges

```
struct OnlineBridges {
  private:
  vector<int> parent, twoECC, dsu_cc, size, lastVisit;
  int bridgeCount = 0;
  int lcaIteration = 0;
  int find2ECC(int v) {
    if (v == -1) return -1;
    return twoECC[v] == v ? v : twoECC[v] = find2ECC(twoECC[v]);
  }
  int findCC(int v) {
    v = find2ECC(v);
    return dsu_cc[v] == v ? v : dsu_cc[v] = findCC(dsu_cc[v]);
  }
  void makeRoot(int v) {
    int root = v;
    int child = -1;
    while (v != -1) {
      int p = find2ECC(parent[v]);
      parent[v] = child;
      dsu_cc[v] = root;
      child = v;
      v = p;
    }
    size[root] = size[child];
  }
  void mergePath(int a, int b) {
    ++lcaIteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
      if (a != -1) {
        a = find2ECC(a);
        path_a.push_back(a);
        if (lastVisit[a] == lcaIteration) {
          lca = a;
          break;
        }
        lastVisit[a] = lcaIteration;
        a = parent[a];
      }
      if (b != -1) {
        b = find2ECC(b);
        path_b.push_back(b);
        if (lastVisit[b] == lcaIteration) {
          lca = b;
          break;
        }
        lastVisit[b] = lcaIteration;
        b = parent[b];
      }
    }
    for (int v : path_a) {
      twoECC[v] = lca;
      if (v == lca) break;
      --bridgeCount;
    }
    for (int v : path_b) {
      twoECC[v] = lca;
      if (v == lca) break;
      --bridgeCount;
    }
  }
  public:
  OnlineBridges(int n) {
    parent.assign(n + 1, -1);
    twoECC.resize(n + 1);
    dsu_cc.resize(n + 1);
    size.assign(n + 1, 1);
    lastVisit.assign(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
      twoECC[i] = i;
      dsu_cc[i] = i;
    }
  }
  void addEdge(int a, int b) {
    a = find2ECC(a);
    b = find2ECC(b);
    if (a == b) return;
    int ca = findCC(a);
    int cb = findCC(b);
    if (ca != cb) {
      ++bridgeCount;
      if (size[ca] > size[cb]) {
        swap(a, b);
        swap(ca, cb);
      }
      makeRoot(a);
      parent[a] = dsu_cc[a] = b;
      size[cb] += size[a];
    } else {
      mergePath(a, b);
    }
  }
  int getBridgeCount() { return bridgeCount; }
  bool sameConnectedComponent(int u, int v) { return findCC(u) ==
  ↪  findCC(v); }
  bool inSimpleCycle(int u, int v) { return find2ECC(u) == find2ECC(v); }
};
void solve() {
  int n, q;
  cin >> n >> q;
  OnlineBridges adj(n + 1);
  while (q--) {
    int t, u, v;
    cin >> t >> u >> v;
    if (t == 1) {
      adj.addEdge(u, v);
    } else {
      cout << (adj.inSimpleCycle(u, v) ? "YES\n" : "NO\n");
    }
  }
}
```

Prim

```
int Prim(int start, vector<vector<pair<int, int>>> &adj) {
  int n = (int)adj.size();
  vector<bool> inMST(n);
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
  int sum = 0;
  pq.emplace(0, start);
  while (!pq.empty()) {
    auto [w, u] = pq.top();
    pq.pop();
    if (inMST[u]) continue;
    inMST[u] = true;
    sum += w;
    for (auto [weight, v] : adj[u]) {
      if (inMST[v]) continue;
      pq.emplace(weight, v);
    }
  }
  return sum;
}
int Prim(int start, vector<vector<pair<int, int>>> &adj) {
  int n = (int)adj.size();
  vector<bool> inMST(n);
  priority_queue<array<int, 3>, vector<array<int, 3>>, greater<>> pq;
  vector<vector<pair<int, int>>> MST(n + 1);
  int sum = 0;
  pq.push({0, start, 0});
  while (!pq.empty()) {
    auto [w, u, p] = pq.top();
    pq.pop();
    if (inMST[u]) continue;
    if (p) {
      MST[u].emplace_back(p, w);
      MST[p].emplace_back(u, w);
    }
    inMST[u] = true;
    sum += w;
    for (auto [v, weight] : adj[u]) {
      if (inMST[v]) continue;
      pq.push({weight, v, u});
    }
  }
  adj = MST;
  return sum;
}
```

SCC

```
struct SCC {
    int n, tim, countSCC;
    vector< bool > stacked;
    vector< int > in, low, id, st;
    vector< vector< int > > adj, comp;
    vector< set< int > > DAG;
    void init(int _n, vector< vector< int > > &graph) {
        n = _n, tim = 1, countSCC = 0;
        adj = graph, in.assign(n + 1, 0);
        low.assign(n + 1, 0), id.assign(n + 1, 0);
        stacked.assign(n + 1, false), st.clear(), comp.clear(), DAG.clear();
        for (int v = 1; v <= n; ++v) if (not in[v]) dfs(v);
        makeDAG();
    }
    void dfs(int v) {
        in[v] = low[v] = tim++;
        st.push_back(v), stacked[v] = true;
        for (int u: adj[v]) {
            if (not in[u]) dfs(u), low[v] = min(low[v], low[u]);
            else if (stacked[u]) low[v] = min(low[v], in[u]);
        }
        if (low[v] == in[v]) {
            comp.push_back({});
            int u = -1;
            while (u != v) {
                u = st.back();
                st.pop_back(), stacked[u] = false;
                id[u] = countSCC, comp.back().push_back(u);
            }
            ++countSCC;
        }
    }
    void makeDAG() {
        DAG.assign(countSCC, {});
        for (int v = 1; v <= n; ++v)
            for (int u: adj[v])
                if (id[u] != id[v])
                    DAG[id[v]].insert(id[u]);
    }
    vector< int > topo() {
        vector< int > indeg(countSCC);
        for (int i = 0; i < countSCC; ++i)
```

```
            for (int to: DAG[i]) ++indeg[to];
        queue< int > q;
        vector< int > ans;
        for (int i = 0; i < countSCC; ++i) {
            if (indeg[i] == 0) ans.push_back(i), q.push(i);
        }
        while (not q.empty()) {
            int v = q.front();
            q.pop();
            for (int u: DAG[v]) {
                --indeg[u];
                if (indeg[u] == 0) ans.push_back(u), q.push(u);
            }
        }
        return ans;
    }
};
```

SPFA

```
const ll oo = 1e18;
struct edge {
  ll to, cost;
};
vector<vector<edge>> adj;
vector<ll> dis;
vector<ll> cnt;
vector<bool> in;
bool SPFA(ll u) {
  dis.assign(n + 1, oo);
  cnt.assign(n + 1, 0);
  in.assign(n + 1, false);
  queue<ll> q;
  q.push(u);
  dis[u] = 0;
  in[u] = true;
  while (!q.empty()) {
    ll x = q.front();
    q.pop();
    in[x] = false;
    for (auto& e : adj[x]) {
      if (dis[e.to] > dis[x] + e.cost) {
        dis[e.to] = dis[x] + e.cost;
        if (!in[e.to]) {
          q.push(e.to);
          in[e.to] = true;
          cnt[e.to]++;
          if (cnt[e.to] > n) return false;
        }
      }
    }
  }
  return true;
}
```

Tarjan

```
void IS_BRIDGE(int v, int to) {}
int n, m, x;
vector<vector<int>> adj;
vector<bool> visited;
vector<int> tin, low;
int timer;
void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
  bool parent_skipped = false;
  for (int to : adj[v]) {
    if (to == p && !parent_skipped) {
      parent_skipped = true;
      continue;
    }
    if (visited[to]) {
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      if (low[to] > tin[v]) IS_BRIDGE(v, to);
    }
  }
}
void find_bridges() {
  timer = 0;
  visited.assign(n + 1, false);
  tin.assign(n + 1, -1);
  low.assign(n + 1, -1);
  for (int i = 1; i <= n; ++i) {
    if (!visited[i]) dfs(i);
  }
}
```

Trees

LCA

```
struct LCA {
  vector<vector<int>> ancestor, minE;
  vector<int> level;
  int LG;
  LCA(vector<vector<pair<int, int>>> &adj) {
    int n = (int)adj.size();
    LG = __lg(n) + 1;
    ancestor.assign(LG, vector<int>(n));
    minE.assign(LG, vector<int>(n));
    level.assign(n, {});
    build(1, 0, adj);
    for (int i = 1; i < LG; ++i) {
      for (int u = 1; u < n; ++u) {
        ancestor[i][u] = ancestor[i - 1][ancestor[i - 1][u]];
        minE[i][u] = min(minE[i - 1][u], minE[i - 1][ancestor[i - 1][u]]);
      }
    }
  }
  void build(int u, int p, vector<vector<pair<int, int>>> &adj) {
    for (auto [v, w] : adj[u]) {
      if (v == p) continue;
      level[v] = level[u] + 1;
      ancestor[0][v] = u;
      minE[0][v] = w;
      build(v, u, adj);
    }
  }
  int KthAnc(int u, int k) {
    for (int i = 0; k; ++i, k >>= 1) {
      if (k & 1) u = ancestor[i][u];
    }
    return u;
  }
  int getLCA(int u, int v) {
    if (level[u] > level[v]) swap(u, v);
    int k = level[v] - level[u];
    v = KthAnc(v, k);
    if (v == u) return v;
    for (int i = LG - 1; ~i; --i) {
      if (ancestor[i][v] != ancestor[i][u]) {
        v = ancestor[i][v];
        u = ancestor[i][u];
      }
    }
    return ancestor[0][u];
  }
  int getDist(int u, int v) {
    int lca = getLCA(u, v);
    return level[u] + level[v] - 2 * level[lca];
  }
  int query(int u, int v) {
    if (level[u] > level[v]) swap(u, v);
    int k = level[v] - level[u];
    int res = inf;
    for (int i = 0; k; ++i, k >>= 1) {
      if (k & 1) {
        res = min(res, minE[i][v]);
        v = ancestor[i][v];
      }
    }
    if (v == u) return res;
    for (int i = LG - 1; ~i; --i) {
      if (ancestor[i][v] != ancestor[i][u]) {
        res = min({res, minE[i][v], minE[i][u]});
        v = ancestor[i][v];
        u = ancestor[i][u];
      }
    }
    res = min({res, minE[0][v], minE[0][u]});
    return res;
  }
};
```

MO on Tree

```
// count number of distinct colors in path
const int N = 2e5 + 5, B = 360, LG = 25;
```

```
int in[N], out[N], tree[2 * N], depth[N],
    timer = 1;  // don't forget to reset timer in test cases
vector<vector<int>> adj, ancestor;
int KthAnc(int u, int k) {
  for (int i = 0; k; ++i, k >>= 1) {
    if (k & 1) u = ancestor[i][u];
  }
  return u;
}
int LCA(int u, int v) {
  if (depth[u] > depth[v]) swap(u, v);
  int k = depth[v] - depth[u];
  v = KthAnc(v, k);
  if (v == u) return v;
  for (int i = LG - 1; ~i; --i) {
    if (ancestor[i][v] != ancestor[i][u]) {
      v = ancestor[i][v];
      u = ancestor[i][u];
    }
  }
  return ancestor[0][u];
}
void tour(int u, int p) {
  tree[timer] = u;
  in[u] = timer++;
  for (int i = 1; i < LG; ++i) { ancestor[i][u] = ancestor[i -
  ↪ 1][ancestor[i - 1][u]]; }
  for (auto v : adj[u]) {
    if (v == p) continue;
    depth[v] = depth[u] + 1;
    ancestor[0][v] = u;
    tour(v, u);
  }
  tree[timer] = u;
  out[u] = timer++;
}
struct Query {
  int l, r, lca, idx;
  Query(int u, int v, int i) {
    idx = i;
    if (in[u] > in[v]) swap(u, v);
    int lc = LCA(u, v);
    if (lc == u) {
      l = in[u], r = in[v];
      lca = -1;
    } else {
      l = out[u], r = in[v];
      lca = lc;
    }
  }
  inline pair<int, int> toPair() const {
    return make_pair(l / B, ((l / B) & 1) ? -r : +r);
  }
};
struct MO_onTree {
  vector<int> v, frq, vis;
  int n, ans = 0;
  MO_onTree(vector<int> &a) {
    v = a;
    n = (int)a.size();
    frq.assign(n + 1, 0);
    vis.assign(n + 1, 0);
  }
  void add(int u) {
    u = v[u];
    if (++frq[u] == 1) ans++;
  }
  void erase(int u) {
    u = v[u];
    if (--frq[u] == 0) ans--;
  }
  void go(int idx) {
    int u = tree[idx];
    vis[u] ^= 1;
    if (vis[u]) add(u);
    else erase(u);
  }
  vector<int> Process(vector<Query> &query) {
    sort(query.begin(), query.end(),
         [&](Query &a, Query &b) { return a.toPair() < b.toPair(); });
    vector<int> ret(query.size());
    int l = query[0].l, r = l;
    go(l);
    for (const auto &[lq, rq, lca, idx] : query) {
      while (lq < l) --l, go(l);
      while (rq > r) ++r, go(r);
      while (lq > l) go(l), ++l;
      while (rq < r) go(r), --r;
      if (~lca) add(lca);
      ret[idx] = ans;
      if (~lca) erase(lca);
    }
    return ret;
  }
};
void solve() {
  int n, q;
  cin >> n >> q;
  vector<int> c(n + 1);
  adj.assign(n + 1, {});
  ancestor.assign(LG, vector<int>(n + 1));
  map<int, int> mp;
  for (int i = 1; i <= n; ++i) cin >> c[i], mp[c[i]];
  int id = 1;
  for (auto &[_, v] : mp) v = id++;
  for (int i = 1; i <= n; ++i) c[i] = mp[c[i]];
  for (int i = 1, a, b; i < n; ++i) {
    cin >> a >> b;
    adj[a].emplace_back(b);
    adj[b].emplace_back(a);
  }
  timer = 1;
  tour(1, 0);
  vector<Query> query;
  for (int i = 0, a, b; i < q; ++i) {
    cin >> a >> b;
    query.emplace_back(a, b, i);
  }
  MO_onTree mo(c);
  auto res = mo.Process(query);
  for (auto i : res) cout << i << '\n';
}
```

rooted Tree Isomorphism

```
const int mod = 2131131137, N = 1e5 + 5, B = 219;
int in[2][N], out[2][N], pw[N], timer = 1;
vector<vector<int>> adj[2];
void tour(int u, int p, int t) {
  in[t][u] = timer++;
  for (auto v : adj[t][u]) {
    if (v == p) continue;
    tour(v, u, t);
  }
  out[t][u] = timer;
}
int dfs(int u, int p, int t) {
  int hash = '(';
  vector<pair<int, int>> res;  // hash , number of nodes
  for (auto v : adj[t][u]) {
    if (v == p) continue;
    int h = dfs(v, u, t);
    res.emplace_back(h, out[t][v] - in[t][v]);
  }
  sort(res.begin(), res.end());
  for (auto [h, cnt] : res) { hash = (hash * pw[cnt] % mod + h) % mod; }
  hash = (hash * B + ')') % mod;
  return hash;
}
void solve() {
  int n;
  cin >> n;
  auto get = [&](int t) {
    adj[t].assign(n + 1, {});
    for (int i = 1, a, b; i < n; ++i) {
      cin >> a >> b;
      adj[t][a].emplace_back(b);
      adj[t][b].emplace_back(a);
    }
  };
  get(0), get(1);
  tour(1, 0, 0);
  timer = 1;
  tour(1, 0, 1);
  timer = 1;
  int hash1 = dfs(1, 0, 0);
  int hash2 = dfs(1, 0, 1);
  cout << (hash1 == hash2 ? "YES" : "NO") << '\n';
}
// pw[0] = 1;
// for (int i = 1; i < N; ++i) { pw[i] = pw[i - 1] * B % mod; }
```

Sack

```
// check if vertices in subtree u of level h can form a palindrome string
const int N = 5e5 + 5;
int in[N], out[N], tree[N], level[N], timer = 1;
vector<vector<int>> adj;
void tour(int u, int p) {
  tree[timer] = u;
  in[u] = timer++;
  level[u] = level[p] + 1;
  for (auto v : adj[u]) {
    if (v == p) continue;
    tour(v, u);
  }
  out[u] = timer;
}
int frq[N][26], answer[N];
vector<pair<int, int>> query[N];
string s;
void add(int u) { frq[level[u]][s[u]]++; }
void erase(int u) { frq[level[u]][s[u]]--; }
int get(int h) {
  int res = 0;
  for (int i = 0; i < 26; ++i) res += frq[h][i] & 1;
  return res < 2;
}
void dfs(int u, int p, int keep) {
  int bgChild = -1, maxS = -1;
  for (auto v : adj[u]) {
    if (v == p or out[v] - in[v] <= maxS) continue;
    maxS = out[v] - in[v];
    bgChild = v;
  }
  for (auto v : adj[u]) {
    if (v == p or v == bgChild) continue;
    dfs(v, u, 0);
  }
  if (~bgChild) dfs(bgChild, u, 1);
  for (auto v : adj[u]) {
    if (v == p or v == bgChild) continue;
    for (int i = in[v]; i < out[v]; ++i) add(tree[i]);
  }
  add(u);
  for (auto [h, i] : query[u]) answer[i] = get(h);
  if (!keep) {
    for (int i = in[u]; i < out[u]; ++i) erase(tree[i]);
  }
}
void solve() {
  int n, q;
  cin >> n >> q;
  adj.assign(n + 1, {});
  for (int i = 2, a; i <= n; ++i) {
    cin >> a;
    adj[a].emplace_back(i);
    adj[i].emplace_back(a);
  }
  cin >> s;
  s = '#' + s;
  for (auto &i : s) i -= 'a';
  for (int i = 0, u, h; i < q; ++i) {
    cin >> u >> h;
    query[u].emplace_back(h, i);
  }
  tour(1, 0);
  dfs(1, 0, 0);
  for (int i = 0; i < q; ++i) cout << (answer[i] ? "Yes" : "No") << '\n';
}
```

Tree Isomorphism

```
const int mod = 2131131137, N = 1e5 + 5, B = 219;
int in[2][N], out[2][N], pw[N], timer = 1, n;
vector<vector<int>> adj[2];
void tour(int u, int p, int t) {
  in[t][u] = timer++;
  for (auto v : adj[t][u]) {
    if (v == p) continue;
    tour(v, u, t);
  }
  out[t][u] = timer;
}
vector<int> getCentroid(int u, int p, int t) {
  vector<int> result;
  bool isCentroid = true;
  for (auto v : adj[t][u]) {
    if (v == p) continue;
    auto answer = getCentroid(v, u, t);
    for (auto c : answer) result.emplace_back(c);
    if (out[t][v] - in[t][v] > n / 2) isCentroid = false;
  }
  if (n - (out[t][u] - in[t][u]) > n / 2) isCentroid = false;
  if (isCentroid) result.emplace_back(u);
  return result;
}
int dfs(int u, int p, int t) {
  int hash = '(';
  vector<pair<int, int>> res;  // hash , number of nodes
  for (auto v : adj[t][u]) {
    if (v == p) continue;
    int h = dfs(v, u, t);
    res.emplace_back(h, out[t][v] - in[t][v]);
  }
  sort(res.begin(), res.end());
  for (auto [h, cnt] : res) { hash = (hash * pw[cnt] % mod + h) % mod; }
  hash = (hash * B + ')') % mod;
  return hash;
}
void solve() {
  cin >> n;
  auto get = [&](int t) {
    adj[t].assign(n + 1, {});
    for (int i = 1, a, b; i < n; ++i) {
      cin >> a >> b;
      adj[t][a].emplace_back(b);
      adj[t][b].emplace_back(a);
    }
  };
  get(0), get(1);
  tour(1, 0, 0);
  timer = 1;
  tour(1, 0, 1);
  auto Centroid1 = getCentroid(1, 0, 0);
  auto Centroid2 = getCentroid(1, 0, 1);
  for (auto i : Centroid1) {
    for (auto j : Centroid2) {
      tour(i, 0, 0);
      timer = 1;
      tour(j, 0, 1);
      if (dfs(i, 0, 0) == dfs(j, 0, 1)) {
        cout << "YES\n";
        return;
      }
    }
  }
  cout << "NO\n";
}
// pw[0] = 1;
// for (int i = 1; i < N; ++i) { pw[i] = pw[i - 1] * B % mod; }
```

Strings

2D Hashing

```
// Find all occurrence of a 2d word in a 2d word.
const int N = 2005, M = 2005;
int baseX, baseY, mod, powX[N], powY[M];
struct Hashing {
  vector<vector<int>> hash;
  int n, m;

  void build() {
    if (mod) return;
    mt19937 rng(chrono::system_clock::now().time_since_epoch().count());
    auto rnd = [&](int a, int b) { return a + rng() % (b - a + 1); };
    auto check = [&](int x) {
      for (int i = 2; i * i <= x; ++i)
        if (x % i == 0) return true;
      return false;
    };
    baseX = rnd(130, 500);
    baseY = rnd(130, 500);
    mod = rnd(1e9, 2e9);
    while (check(mod)) mod--;
    powX[0] = powY[0] = 1;
    for (int i = 0; i <= 2000; i++) powX[i + 1] = 1LL * powX[i] * baseX %
     ↪  mod;
    for (int i = 0; i <= 2000; i++) powY[i + 1] = 1LL * powY[i] * baseY %
     ↪  mod;
  }
  Hashing(vector<string>& s) {
    build();
    n = (int)s.size(), m = (int)s[0].size();
    hash.assign(n + 1, vector<int>(m + 1, 0));
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) { hash[i + 1][j + 1] = s[i][j]; }
    }
    for (int i = 0; i <= n; i++) {
      for (int j = 0; j < m; j++) {
        hash[i][j + 1] = (hash[i][j + 1] + 1LL * hash[i][j] * baseY % mod)
        ↪  % mod;
      }
    }
    for (int i = 0; i < n; i++) {
      for (int j = 0; j <= m; j++) {
        hash[i + 1][j] = (hash[i + 1][j] + 1LL * hash[i][j] * baseX % mod)
        ↪  % mod;
      }
    }
  }

  int get_hash(int x1, int y1, int x2, int y2) {  // 1-indexed
    assert(1 <= x1 && x1 <= x2 && x2 <= n);
    assert(1 <= y1 && y1 <= y2 && y2 <= m);
    x1--;
    y1--;
    int dx = x2 - x1, dy = y2 - y1;
    return (1LL * (hash[x2][y2] - 1LL * hash[x2][y1] * powY[dy] % mod +
    ↪  mod) % mod -
            1LL * (hash[x1][y2] - 1LL * hash[x1][y1] * powY[dy] % mod +
            ↪  mod) % mod *
                powX[dx] % mod +
            mod) %
           mod;
  }

  int get_hash() { return get_hash(1, 1, n, m); }
};
```

```
void PartialSum_in2D(int x1, int y1, int x2, int y2, vector<vector<int>>&
↪  Prefix) {
  if (x1 > x2) swap(x1, x2);
  if (y1 > y2) swap(y1, y2);
  Prefix[x1][y1]++;
  Prefix[x2 + 1][y1]--;
  Prefix[x1][y2 + 1]--;
  Prefix[x2 + 1][y2 + 1]++;
}

void PrefixSum_2D(int n, int m, vector<vector<int>>& Prefix) {
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) Prefix[i][j] += Prefix[i][j - 1];
  for (int i = 1; i <= m; ++i)
    for (int j = 1; j <= n; ++j) Prefix[j][i] += Prefix[j - 1][i];
}
```

```
signed main() {
  int n, m;
  cin >> n >> m;
  vector<string> word(n);
  for (int i = 0; i < n; ++i) cin >> word[i];
  int X, Y;
  cin >> X >> Y;
  vector<string> grid(X);
  for (int i = 0; i < X; ++i) cin >> grid[i];
  Hashing hash(word);
  int pattern = hash.get_hash();
  hash = Hashing(grid);
  vector<vector<int>> res(X + 2, vector<int>(Y + 2));
  for (int i = 1; i + n - 1 <= X; ++i)
    for (int j = 1; j + m - 1 <= Y; ++j)
      if (pattern == hash.get_hash(i, j, i + n - 1, j + m - 1))
        PartialSum_in2D(i, j, i + n - 1, j + m - 1, res);
  PrefixSum_2D(X, Y, res);
  for (int i = 1; i <= X; ++i) {
    for (int j = 1; j <= Y; ++j) cout << (res[i][j] ? grid[i - 1][j - 1] :
    ↪  '.');
    cout << '\n';
  }
  return 0;
}
```

evaluate expression

```
bool delim(char c) { return c == ' '; }
bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }
int priority(char op) {
  if (op == '+' || op == '-') return 1;
  if (op == '*' || op == '/') return 2;
  return -1;
}
void process_op(stack<int> &st, char op) {
  int r = st.top();
  st.pop();
  int l = st.top();
  st.pop();
  switch (op) {
    case '+':
      st.push(l + r);
      break;
    case '-':
      st.push(l - r);
      break;
    case '*':
      st.push(l * r);
      break;
    case '/':
      st.push(l / r);
      break;
  }
}
int evaluate(string &s) {
  stack<int> st;
  stack<char> op;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim(s[i])) continue;
    if (s[i] == '(') {
      op.push('(');
    } else if (s[i] == ')') {
      while (op.top() != '(') {
        process_op(st, op.top());
        op.pop();
      }
      op.pop();
    } else if (is_op(s[i])) {
      char cur_op = s[i];
      while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
        process_op(st, op.top());
        op.pop();
      }
      op.push(cur_op);
    } else {
      int number = 0;
      while (i < (int)s.size() && isalnum(s[i])) number = number * 10 +
      ↪  s[i++] - '0';
      --i;
      st.push(number);
    }
  }
  while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
  }
  return st.top();
}
```

Hashing

```
/*
 * Given a string, your task is to determine the longest palindromic
 ↪ substring
 * of the string. For example, the longest palindrome in aybabtu is bab.
 * */
const int M = 2, N = 1e6 + 5;
int B[M], mods[M], pw[M][N];
struct Hashing {
  vector<array<int, M>> prefix, suffix;
  void build() {
    if (B[0]) return;
    mt19937 rng(chrono::system_clock::now().time_since_epoch().count());
    auto rnd = [&](int a, int b) { return a + rng() % (b - a + 1); };
    auto check = [&](int x) {
      for (int i = 2; i * i <= x; ++i)
        if (x % i == 0) return true;
      return false;
    };
    for (int i = 0; i < M; ++i) {
      B[i] = rnd(100, 500);
      mods[i] = rnd(1e9, 2e9);
      while (check(mods[i])) mods[i]--;
      pw[i][0] = 1;
      for (int j = 1; j < N; ++j) pw[i][j] = pw[i][j - 1] * B[i] % mods[i];
    }
  }
  Hashing(string &s, bool withSuffix = false) {
    build();
    InitPrefix(s);
    if (withSuffix) InitSuffix(s);
  }
  void InitPrefix(string &s) {
    int n = (int)s.size();
    prefix.assign(n, {});
    array<int, M> hash{};
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < M; ++j) hash[j] = (hash[j] * B[j] + s[i]) %
       ↪ mods[j];
      prefix[i] = hash;
    }
  }
  void InitSuffix(string &s) {
    int n = (int)s.size();
    suffix.assign(n, {});
    array<int, M> hash{};
    for (int i = n - 1; i >= 0; --i) {
      for (int j = 0; j < M; ++j) hash[j] = (hash[j] * B[j] + s[i]) %
       ↪ mods[j];
      suffix[i] = hash;
    }
  }
  array<int, M> get_substr_Hash(int l, int r) {
    if (l == 0) return prefix[r];
    array<int, M> res = prefix[r];
    for (int i = 0; i < M; ++i) {
      res[i] -= prefix[l - 1][i] * pw[i][r - l + 1] % mods[i];
      if (res[i] < 0) res[i] += mods[i];
    }
    return res;
  }
  array<int, M> get_reverse_Hash(int l, int r) {
    if (r + 1 == suffix.size()) return suffix[l];
    array<int, M> res = suffix[l];
    for (int i = 0; i < M; ++i) {
      res[i] -= suffix[r + 1][i] * pw[i][r - l + 1] % mods[i];
      if (res[i] < 0) res[i] += mods[i];
    }
    return res;
  }
  bool isPal(int l, int r) { return get_substr_Hash(l, r) ==
   ↪ get_reverse_Hash(l, r); }
};
signed main() {
  string s;
  cin >> s;
  Hashing hash(s, true);
  auto can = [&](int c, int L, bool even) {
    if (c - L < 0 or c + L + even >= s.size()) return false;
    return hash.isPal(c - L, c + L + even);
  };
  int ans = 1;
  for (int center = 0; center < s.size() - 1; ++center) {
    if (s[center] == s[center + 1]) {
      int l = 0, r = s.size() + 5, mid = 0, res = 0;
      while (l <= r) {
        mid = (l + r) / 2;
        if (can(center, mid, true)) {
          res = 2 * mid + 2;
          l = mid + 1;
        } else {
          r = mid - 1;
        }
      }
      ans = max(res, ans);
    }
    int l = 0, r = s.size() + 5, mid = 0, res = 0;
    while (l <= r) {
      mid = (l + r) / 2;
      if (can(center, mid, false)) {
        res = 2 * mid + 1;
        l = mid + 1;
      } else {
        r = mid - 1;
      }
    }
    ans = max(res, ans);
  }
  for (int i = 0; i + ans <= s.size(); ++i) {
    if (hash.isPal(i, i + ans - 1)) {
      cout << s.substr(i, ans);
      return 0;
    }
  }
}
```

KMP

```
struct KMP {
  vector<int> pi;
  string pattern;
  int n = 0;
  KMP(string &pat) {
    pattern = pat;
    n = (int)pat.size();
    pi.assign(n + 1, {});
    for (int i = 1; i < n; ++i) {  // O(n)
      int k = pi[i - 1];
    while (k > 0 and pat[k] != pat[i]) k = pi[k - 1];
    k += pat[i] == pat[k];
    pi[i] = k;
  }
}
// find all occurrence of pattern in string s
vector<int> find(string &s) {
  int k = 0;
  vector<int> ret;
  for (int i = 0; i < s.size(); ++i) {
    while (k and pattern[k] != s[i]) k = pi[k - 1];
    k += s[i] == pattern[k];
    if (k == pattern.size()) ret.push_back(i - k + 1);
  }
  return ret;
}
// count the number of appearances of each prefix
vector<int> count() {
  vector<int> ans(n + 1);
  for (int i = 0; i < n; i++) ans[pi[i]]++;
  for (int i = n - 1; i > 0; i--) ans[pi[i - 1]] += ans[i];
  for (int i = 0; i < n; i++) ans[i]++;
  return ans;
}
// find all periods of a string
// A period of a string is a prefix that can be used to
// generate the whole string by repeating the prefix
// if we want a full period the n % (n - k) == 0
vector<int> period() {
  vector<int> ans;
  int k = pi[n - 1];
  while (k) {
    ans.push_back(n - k);
    k = pi[k - 1];
  }
  ans.push_back(n);
  return ans;
}
vector<vector<int>> compute_automaton(string s) {
  s += '#';
  vector<vector<int>> aut(s.size(), vector<int>(26));
  for (int i = 0; i < s.size(); ++i) {
    for (int c = 0; c < 26; ++c) {
      if (i and c + 'a' != s[i]) aut[i][c] = aut[pi[i - 1]][c];
```

```
        else aut[i][c] = i + (c + 'a' == s[i]);
      }
    }
    return aut;
  }
};
vector<int> computePrefix(string &s) {
  vector<int> longestPrefix(s.size());
  for (int i = 1; i < s.size(); ++i) {
    int k = longestPrefix[i - 1];
    while (k and s[k] != s[i]) k = longestPrefix[k - 1];
    k += s[k] == s[i];
    longestPrefix[i] = k;
  }
  return longestPrefix;
}
```

manacher

```
// Extend to Palindrome -> append character to make the string palindrome
vector<int> manacher_odd(string s) {
  int n = s.size();
  s = "$" + s + "^";
  vector<int> p(n + 2);
  int l = 0, r = 1;
  for (int i = 1; i <= n; i++) {
    p[i] = min(r - i, p[l + (r - i)]);
    while (s[i - p[i]] == s[i + p[i]]) { p[i]++; }
    if (i + p[i] > r) { l = i - p[i], r = i + p[i]; }
  }
  return vector<int>(begin(p) + 1, end(p) - 1);
}
vector<int> manacher(string s) {
  string t;
  for (auto c : s) { t += string("#") + c; }
  auto res = manacher_odd(t + "#");
  return res;
}
signed main() {
  string s;
  while (cin >> s) {
    auto x = manacher(s);
    int maxIDX = s.size();
    for (int i = 1; i < x.size() - 1; ++i) {
      x[i]--;
      int start = (i - x[i]) / 2;
      if (start + x[i] == s.size()) { maxIDX = min(maxIDX, start); }
    }
    cout << s;
    --maxIDX;
    while (~maxIDX) cout << s[maxIDX--];
    cout << '\n';
  }
}
```

minmum Period Query

```
vector<int> linear_sieve(int n) {
  vector<int> lp(n + 1);
  vector<int> pr;
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = i;
      pr.push_back(i);
    }
    for (int j = 0; i * pr[j] <= n; ++j) {
      lp[i * pr[j]] = pr[j];
      if (pr[j] == lp[i]) { break; }
    }
  }
  return lp;
}
struct HashString {
  const int A = 31;
  const int B = 991831889;
  vector<int> pows, sums;
  HashString(string s) {
    int n = s.size();
    pows.resize(n + 1);
    pows[0] = 1;
    sums.resize(n + 1);
    sums[0] = 0;
    for (int i = 1; i <= n; i++) {
      pows[i] = pows[i - 1] * A % B;
      sums[i] = (sums[i - 1] * A + s[i - 1]) % B;
    }
  }
  int hash(int a, int b) {
    int h = sums[b + 1] - sums[a] * pows[b - a + 1];
    return (h % B + B) % B;
  }
};
void solve() {
  int n;
  string s;
  cin >> n >> s;
  HashString Hash(s);
  auto lp = linear_sieve(s.size());
  auto minPeriod = [&](int l, int r) -> long long {
    if ((l == r) || Hash.hash(l, r - 1) == Hash.hash(l + 1, r)) return 1;
    int len = (r - l + 1), ans = len;
    while (len > 1) {
      if (Hash.hash(l, r - ans / lp[len]) == Hash.hash(l + ans / lp[len],
      ↪  r))
        ans /= lp[len];
      len /= lp[len];
    }
    return ans;
  };
  int q;
  cin >> q;
  while (q--) {
    int l, r;
    cin >> l >> r;
    cout << minPeriod(l - 1, r - 1) << '\n';
  }
}
```

Palindrome InRange

```
template <typename T>
struct WaveletTree {
  int lo, hi;
  WaveletTree *l = nullptr, *r = nullptr;
  vector<int> b;  // prefix-counts of "go-left"
  vector<ll> c;   // prefix-sums of values
  // Build from [first, last), assuming all values in range [x, y]
  template <typename It>
  WaveletTree(It first, It last, T x, T y) : lo(x), hi(y) {
    int n = distance(first, last);
    if (n <= 0) return;
    T mid = T((lo + hi) >> 1);
    b.reserve(n + 1);
    c.reserve(n + 1);
    b.push_back(0);
    c.push_back(0);
    for (It it = first; it != last; ++it) {
      b.push_back(b.back() + (*it <= mid));
      c.push_back(c.back() + *it);
    }
    if (lo == hi) return;
    auto pivot = stable_partition(first, last, [mid](T v) { return v <=
    ↪  mid; });
    l = new WaveletTree(first, pivot, lo, mid);
    r = new WaveletTree(pivot, last, mid + 1, hi);
  }
  // k-th smallest in [L, R], 1-based k
  T kth(int L, int R, int k) const {
    if (L > R) return T(0);
    if (lo == hi) return lo;
    int inLeft = b[R] - b[L - 1];
    int lb = b[L - 1], rb = b[R];
    if (k <= inLeft) return l->kth(lb + 1, rb, k);
    else return r->kth(L - lb, R - rb, k - inLeft);
  }
  // count <= k in [L, R]
  int LTE(int L, int R, T k) const {
    if (L > R || k < lo) return 0;
    if (hi <= k) return R - L + 1;
    int lb = b[L - 1], rb = b[R];
    return l->LTE(lb + 1, rb, k) + r->LTE(L - lb, R - rb, k);
  }
  // sum of all values <= k in [L, R]
  ll sum(int L, int R, T k) const {
    if (L > R || k < lo) return 0;
    if (hi <= k) return c[R] - c[L - 1];
    int lb = b[L - 1], rb = b[R];
    return l->sum(lb + 1, rb, k) + r->sum(L - lb, R - rb, k);
  }
  ~WaveletTree() {
    delete l;
    delete r;
  }
};
struct PalWavelet {
  int n;
```

```cpp
vector<int> d1, d2;
vector<ll> A, B, C, D;
WaveletTree<ll> *oddl = nullptr, *oddr = nullptr, *evenl = nullptr,
↪  *evenr = nullptr;
static constexpr int MAXV = 5000000;
PalWavelet(const string &s) {
  n = s.size();
  manacher(s, d1, d2);
  A.resize(n + 1);
  B.resize(n + 1);
  C.resize(n + 1);
  D.resize(n + 1);
  for (int i = 1; i <= n; i++) {
    A[i] = ll(d1[i - 1]) - i;
    B[i] = ll(d1[i - 1]) + i;
    C[i] = ll(d2[i - 1]) - i;
    D[i] = ll(d2[i - 1]) + i;
  }
  // build four wavelet trees over 1..n
  oddl = new WaveletTree<ll>(A.begin() + 1, A.end(), -MAXV, MAXV);
  oddr = new WaveletTree<ll>(B.begin() + 1, B.end(), -MAXV, MAXV);
  evenl = new WaveletTree<ll>(C.begin() + 1, C.end(), -MAXV, MAXV);
  evenr = new WaveletTree<ll>(D.begin() + 1, D.end(), -MAXV, MAXV);
}
~PalWavelet() {
  delete oddl;
  delete oddr;
  delete evenl;
  delete evenr;
}
// answer number of palindromic substrings sums in [l, r]
ll query(int l, int r) const { return odd(l, r) + even(l, r); }
private:
// Manacher algorithm
static void manacher(const string &s, vector<int> &d1, vector<int> &d2) {
  int n = s.size();
  d1.assign(n, 0);
  for (int i = 0, l = 0, rr = -1; i < n; i++) {
    int k = i > rr ? 1 : min(d1[l + rr - i], rr - i + 1);
    while (i - k >= 0 && i + k < n && s[i - k] == s[i + k]) k++;
    d1[i] = k--;
    if (i + k > rr) {
      l = i - k;
      rr = i + k;
    }
  }
  d2.assign(n, 0);
  for (int i = 0, l = 0, rr = -1; i < n; i++) {
    int k = i > rr ? 0 : min(d2[l + rr - i + 1], rr - i + 1);
    while (i - k - 1 >= 0 && i + k < n && s[i - k - 1] == s[i + k]) k++;
    d2[i] = k--;
    if (i + k > rr) {
      l = i - k - 1;
      rr = i + k;
    }
  }
}
static inline ll get_sum(int l, int r) {
  // sum of 1..r minus sum of 1..(l-1)
  return ll(r) * (r + 1) / 2 - ll(l - 1) * l / 2;
}
ll odd(int l, int r) const {
  int m = (l + r) >> 1;
  // left half [l..m]
  ll c = 1 - l;
  int less = oddl->LTE(l, m, c);
  ll ansL = get_sum(l, m) + oddl->sum(l, m, c) + ll(m - l + 1 - less) * c;
  // right half [m+1..r]
  c = 1 + r;
  less = oddr->LTE(m + 1, r, c);
  ll ansR = -get_sum(m + 1, r) + oddr->sum(m + 1, r, c) + ll(r - m -
  ↪  less) * c;
  return ansL + ansR;
}
ll even(int l, int r) const {
  int m = (l + r) >> 1;
  // left half [l..m]
  ll c = -l;
  int less = evenl->LTE(l, m, c);
  ll ansL = get_sum(l, m) + evenl->sum(l, m, c) + ll(m - l + 1 - less) *
  ↪  c;
  // right half [m+1..r]
  c = 1 + r;
  less = evenr->LTE(m + 1, r, c);
  ll ansR = -get_sum(m + 1, r) + evenr->sum(m + 1, r, c) + ll(r - m -
  ↪  less) * c;
  return ansL + ansR;
  }
};
signed main() {
  string s;
  cin >> s;
  PalWavelet pw(s);
  int q;
  cin >> q;
  while (q--) {
    int l, r;
    cin >> l >> r;
    cout << pw.query(l, r) << "\n";
  }
  return 0;
}
```

PalindromicTree

```
struct Node {
  int len;
  int suffLink;
  map<char, int> next;
  int numOccur;
  int firstPos;
  Node(int l, int link) : len(l), suffLink(link), numOccur(0), firstPos(-1)
   ↪  {}
};
class PalindromicTree {
  public:
  vector<Node> tree;
  string s;
  int suff;
  PalindromicTree(const string &str) : s(str) {
    tree.emplace_back(-1, 0);
    tree.emplace_back(0, 0);
    tree[0].suffLink = 0;
    suff = 1;
    build();
  }
  void build() {
    for (int i = 0; i < s.length(); ++i) addChar(i);
  }
  void addChar(int pos) {
    int cur = suff;
    char ch = s[pos];
    while (true) {
      int curlen = tree[cur].len;
      if (pos - curlen - 1 >= 0 && s[pos - curlen - 1] == ch) break;
      cur = tree[cur].suffLink;
    }
    if (tree[cur].next.count(ch)) {
      suff = tree[cur].next[ch];
      tree[suff].numOccur++;
      return;
    }
    int newNode = tree.size();
    tree.emplace_back(tree[cur].len + 2, 0);
    tree[newNode].firstPos = pos;
    tree[cur].next[ch] = newNode;
    if (tree[newNode].len == 1) {
      tree[newNode].suffLink = 1;
    } else {
      int temp = tree[cur].suffLink;
      while (true) {
        int templen = tree[temp].len;
        if (pos - templen - 1 >= 0 && s[pos - templen - 1] == ch) break;
        temp = tree[temp].suffLink;
      }
      tree[newNode].suffLink = tree[temp].next[ch];
    }
    tree[newNode].numOccur = 1;
    suff = newNode;
  }
  int countDistinctPalindromes() { return (int)tree.size() - 2; }
  vector<int> minPalindromePartitions(const string &s) {
    int n = s.size();
    vector<int> dp(n, INT_MAX);
    dp[0] = 0;
    for (int i = 0; i < n; ++i) {
      addChar(i);
      int cur = suff;
      while (cur > 1) {  // nodes 0 and 1 are roots
        int len = tree[cur].len;
        int start = i - len + 1;
        int prev = start - 1;
        dp[i] = min(dp[i], (prev >= 0 ? dp[prev] : 0) + 1);
        cur = tree[cur].suffLink;
      }
    }
    return dp;
  }
  void countOccurrences() {
    vector<int> order(tree.size());
    for (int i = 0; i < order.size(); ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&](int a, int b) { return tree[a].len > tree[b].len; });
    for (int i : order) {
      int link = tree[i].suffLink;
      if (i != link) tree[link].numOccur += tree[i].numOccur;
    }
  }
  void printAllPalindromes() {
    for (int i = 2; i < tree.size(); ++i) {
      int start = tree[i].firstPos - tree[i].len + 1;
      cout << s.substr(start, tree[i].len) << " -> " << tree[i].numOccur << '\n';
    }
  }
  vector<string> getPalindromes() {
    vector<string> result;
    for (int i = 2; i < tree.size(); ++i) {
      int start = tree[i].firstPos - tree[i].len + 1;
      result.push_back(s.substr(start, tree[i].len));
    }
    return result;
  }
};
```

simple hash

```
struct HashString {
  const int A = 31;
  const int B = 991831889;
  vector<int> pows, sums;
  HashString(string s) {
    int n = s.size();
    pows.resize(n + 1);
    pows[0] = 1;
    sums.resize(n + 1);
    sums[0] = 0;
    for (int i = 1; i <= n; i++) {
      pows[i] = pows[i - 1] * A % B;
      sums[i] = (sums[i - 1] * A + s[i - 1]) % B;
    }
  }
  int hash(int a, int b) {
    int h = sums[b + 1] - sums[a] * pows[b - a + 1];
    return (h % B + B) % B;
  }
};
```

SuffixArray

```
struct SuffixArray {
  // suff is the suffix array with the empty suffix being suff[0]
  // lcp[i] holds the lcp between sa[i], sa[i - 1]
  int n;
  vector<int> suff, lcp, pos, lg;
  vector<array<int, 21>> table;
  SuffixArray(string &s, int lim = 256) {
    n = s.size() + 1;
    int k = 0, a, b;
    vector<int> c(s.begin(), s.end() + 1), tmp(n), frq(max(n, lim));
    c.back() = 0;
    suff = lcp = pos = tmp, iota(suff.begin(), suff.end(), 0);
    for (int j = 0, p = 0; p < n; j = max(1ll, j * 2), lim = p) {
      p = j, iota(tmp.begin(), tmp.end(), n - j);
      for (int i = 0; i < n; i++)
        if (suff[i] >= j) tmp[p++] = suff[i] - j;
      fill(frq.begin(), frq.end(), 0);
      for (int i = 0; i < n; i++) frq[c[i]]++;
      for (int i = 1; i < lim; i++) frq[i] += frq[i - 1];
      for (int i = n; i--;) suff[--frq[c[tmp[i]]]] = tmp[i];
      swap(c, tmp), p = 1, c[suff[0]] = 0;
      for (int i = 1; i < n; i++) {
        a = suff[i - 1], b = suff[i];
        c[b] = tmp[a] == tmp[b] && tmp[a + j] == tmp[b + j] ? p - 1 : p++;
      }
    }
    for (int i = 1; i < n; i++) pos[suff[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[pos[i++]] = k)
      for (k &&k--, j = suff[pos[i] - 1]; s[i + k] == s[j + k]; k++) {}
  }
  void preLcp() {
    lg.resize(n + 5);
    table.resize(n + 5);
    for (int i = 2; i < n + 5; ++i) lg[i] = lg[i / 2] + 1;
    for (int i = 0; i < n; ++i) table[i][0] = lcp[i];
    for (int j = 1; j <= lg[n]; ++j)
      for (int i = 0; i <= n - (1 << j); ++i)
        table[i][j] = min(table[i][j - 1], table[i + (1 << (j - 1))][j -
        ↪  1]);
  }
  // pass the pos of the suffixes
  int queryLcp(int i, int j) {
    if (i == j) return n - suff[i] - 1;
    if (i > j) swap(i, j);
    i++;
    int len = lg[j - i + 1];
    return min(table[i][len], table[j - (1 << len) + 1][len]);
  }
};
```

Trie

```
struct Trie {
  private:
  struct Node {
    map<char, int> child;
    int cnt = 0, isEnd = 0;
    int &operator[](char x) { return child[x]; }
  };
  vector<Node> tree;
  public:
  Trie() { tree.emplace_back(); }
  int newNode() {
    tree.emplace_back();
    return (int)tree.size() - 1;
  }
  int sz(int x) { return tree[x].cnt; }
  void update(string &s, int op) {  // op -> 1 add || op -> -1 erase
    int cur = 0;
    for (auto &c : s) {
      if (tree[cur][c] == 0) tree[cur][c] = newNode();
      cur = tree[cur][c];
      tree[cur].cnt += op;
    }
    tree[cur].isEnd += op;
  }
  int count_prefix(string &s) {  // count strings that share a prefix s
    int cur = 0;
    for (auto &c : s) {
      if (tree[cur][c] == 0) return 0;
      cur = tree[cur][c];
    }
    return tree[cur].cnt;
  }
  int LCP(string &s) {  // longest common prefix
    int cur = 0, res = 0;
    for (auto &i : s) {
      if (sz(tree[cur][i]) == 0) { return res; }
      cur = tree[cur][i];
      res++;
    }
    return res;
  }
  string min_string(string &s) {  // return the smallest string have a
  ↪  prefix s
    string t = s;
    int cur = 0;
    for (auto &c : s) {
      if (tree[cur][c] == 0) return "-1";
      cur = tree[cur][c];
    }
    while (tree[cur].isEnd == 0) {
      for (char c = 'a'; c <= 'z'; ++c) {
        if (tree[cur][c] != 0) {
          t += c;
          cur = tree[cur][c];
          break;
        }
      }
    }
    return t;
  }
};
signed main() {
  int n;
  cin >> n;
  vector<string> s(n);
  Trie trie;
  for (int i = 0; i < n; ++i) {
    cin >> s[i];
    trie.update(s[i], 1);
  }
  for (int i = 0; i < n; ++i) {
    trie.update(s[i], -1);
    cout << trie.LCP(s[i]) << '\n';
    trie.update(s[i], 1);
  }
}
```

z function

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

Counting

Burnside

```
/*
 * Your task is to count the number of different necklaces that consist of n
 * pearls and each pearl has m possible colors. Two necklaces are considered
 ↪  to
 * be different if it is not possible to rotate one of them so that they
 ↪  look
 * the same.
 * */
const int mod = 1e9 + 7;
int exp(int Base, int Power) {
  int Result = 1;
  while (Power) {
    if (Power & 1) Result = (Result * Base) % mod;
    Base = (Base * Base) % mod, Power >>= 1;
  }
  return Result;
}
signed main() {
  int n, k;
  cin >> n >> k;
  int res = 0;
  // count number of groups for each rotation
  for (int i = 1; i <= n; ++i) res = (res + exp(k, gcd(i, n))) % mod;
  res = res * exp(n, mod - 2) % mod;
  cout << res;
}
```

Combinatorics

```
const int mod = 1e9 + 7, N = 2e6 + 6;
struct Combinatorics {
  vector<int> fact, inv;

  Combinatorics(int n) {
    fact.assign(n + 1, 1);
    inv.assign(n + 1, 1);
    for (int i = 1; i <= n; ++i) fact[i] = (fact[i - 1] * i) % mod;
    // fact[i + 1] = fact[i] * (i + 1);
    // invfact[i]  = invfact[i + 1] * (i + i);
    inv[n] = modInv(fact[n]);
    for (int i = n - 1; i >= 0; --i) inv[i] = (inv[i + 1] * (i + 1)) % mod;
  }

  int modInv(int n) { return exp(n, mod - 2); }
  int exp(int base, int pow) {
    int res = 1;
    while (pow) {
      if (pow & 1) res = (res * base) % mod;
      base = (base * base) % mod, pow >>= 1;
    }
    return res;
  }
  int nCr(int n, int r) {
    if (r < 0 or n < r) return 0;
    return fact[n] * inv[n - r] % mod * inv[r] % mod;
  }

  int nPr(int n, int r) {
    if (r < 0 or n < r) return 0;
    return fact[n] * inv[n - r] % mod;
  }
  int starsABars(int n, int m) { return nCr(n + m - 1, m); }
  // 1, 1, 2, 5, 14, 42, 132 , 429
  int Catalan(int n) { return modInv(n + 1) * nCr(2 * n, n) % mod; }
  int invCatalan(int n) { return nCr(2 * n, n - 1); }
} comb(N);

//  nCr is odd if r is a submask of n
int nCr_Parity(int n, int r) {
  if (n < r) return 0;
  return (n & r) == r;
}

int C(int n, int k) {
  double res = 1;
  for (int i = 1; i <= k; ++i) res = res * (n - k + i) / i;
  return (int)(res + 0.01);
}

vector<vector<int>> Pascal(int n) {
  vector<vector<int>> C(n + 1, vector<int>(n + 1));
  C[0][0] = 1;
  for (int i = 1; i <= n; ++i) {
    C[i][0] = C[i][i] = 1;
    for (int k = 1; k < i; ++k) C[i][k] = C[i - 1][k - 1] + C[i - 1][k];
  }
  return C;
}
```

count number of pair has gcd x

```
// count number of pairs has gcd equal to x
const int N = 1e6 + 6;
int cnt[N], dp[N];
void solve() {
  int n, mx = 0;
  cin >> n;
  for (int i = 0, t; i < n; ++i) {
    cin >> t;
    mx = max(mx, t);
    cnt[t]++;
  }
  for (int i = mx; i; --i) {
    int mul = 0;
    for (int j = i; j <= mx; j += i) { mul += cnt[j]; }
    dp[i] = mul * (mul - 1) / 2;
    for (int j = i + i; j <= mx; j += i) dp[i] -= dp[j];
  }
  cout << dp[1];
}
```

Derangement

```
//  int how many ways you can get a permutation of size n such that pi != i
vector<int> Derangement(int n) {
  vector<int> D(n + 1, 0);
  D[2] = 1;
  for (int i = 3; i <= n; ++i) { D[i] = (i - 1) * (D[i - 1] + D[i - 2]) %
  ↪  mod; }
  return D;
}
```

Divide and Conq

SmallToLarge

```
/*
 * For each pair (i, j) such that i < j. Let M be the maximum number in the
 * array between i and j i.e.M = max(A_i , A_i+1, ..., A_j-1, A_j ). If the
 ↪  xor
 * of A_i and A_j is greater than M, add M to the sum
 * */
struct Trie_B {
  private:
  struct Node {
    int child[2]{};
    int cnt = 0, isEnd = 0;
    int &operator[](int x) { return child[x]; }
  };
  vector<Node> node;
  public:
  Trie_B() { node.emplace_back(); }
  int newNode() {
    node.emplace_back();
    return node.size() - 1;
  }
  int sz(int x) { return node[x].cnt; }
  int M = 30;
  void update(int x, int op) {  // op -> 1 add || op -> -1 erase
    int cur = 0;
    for (int i = M - 1; i >= 0; --i) {
      int c = x >> i & 1;
      if (node[cur][c] == 0) node[cur][c] = newNode();
      cur = node[cur][c];
      node[cur].cnt += op;
    }
    node[cur].isEnd += op;
  }

  int count(int x, int k) {  // number of x ^ a[i] >= k
    int cur = 0, res = 0;
    for (int i = M - 1; i >= 0; --i) {
      int cx = x >> i & 1;
      int ck = k >> i & 1;
      if (!ck) res += sz(node[cur][!cx]);
      cur = node[cur][ck ^ cx];
      if (sz(cur) == 0) break;
    }
    return res;
  }
};
// #include "sparse_table.h"
void solve() {
  int n;
  cin >> n;
  vector<pair<int, int>> a(n + 1);
  for (int i = 1; i <= n; ++i) cin >> a[i].first, a[i].second = i;
  SparseTable sp(a, [&](pair<int, int> x, pair<int, int> y) { return max(x,
  ↪  y); });
  int answer = 0;
  function<Trie_B(int, int)> go = [&](int l, int r) -> Trie_B {
    if (l > r) { return {}; }
    auto [M, idx] = sp.get(l, r);
    Trie_B trieL = go(l, idx - 1);
    Trie_B trieR = go(idx + 1, r);
    int Lsize = idx - l, Rsize = r - idx;
    if (Lsize > Rsize) {
      trieL.update(M, 1);
      for (int i = idx; i <= r; ++i) { answer += trieL.count(a[i].first, M)
      ↪  * M; }
    } else {
      trieR.update(M, 1);
      for (int i = l; i <= idx; ++i) { answer += trieR.count(a[i].first, M)
      ↪  * M; }
    }
    if (Rsize >= Lsize) {
      for (int i = l; i < idx; ++i) trieR.update(a[i].first, 1);
      return trieR;
    } else {
      for (int i = idx + 1; i <= r; ++i) trieL.update(a[i].first, 1);
      return trieL;
    }
  };
  go(1, n);
  cout << answer << '\n';
}
```

SmallToLarge 2

```
/*
 * For each k=1,...,N, solve the following problem: A has N - k+1
 ↪  (contiguous)
 * subarrays of length k. Take the maximum of each of them, and output the
 ↪  sum
 * of these maxima.
 * */
// #include "sparse_table.h"
void solve() {
  int n;
  cin >> n;
  vector<pair<int, int>> a(n + 1);
  for (int i = 1; i <= n; ++i) cin >> a[i].first, a[i].second = i;
  SparseTable sp(a, [&](pair<int, int> x, pair<int, int> y) { return max(x,
  ↪  y); });
  vector<int> answer(n + 2);
  function<int(int, int)> go = [&](int l, int r) -> int {
    if (l > r) return 0;
    auto [M, idx] = sp.get(l, r);
    int L = go(l, idx - 1);
    int R = go(idx + 1, r);
    if (L > R) swap(L, R);
    for (int i = 0; i <= L; ++i) {
      answer[i + 1] += M;
      answer[i + R + 2] -= M;
    }
    return L + R + 1;
  };
  go(1, n);
  for (int i = 1; i <= n; ++i) answer[i] += answer[i - 1];
  for (int i = 1; i <= n; ++i) cout << answer[i] << '\n';
}
```

DP

LCS

```
const int N = 1e5 + 5;
int n, m, mem[N][N];
string s, t;
vector< vector< int > > nxt;
vector< vector< bool > > vis;

int dp(int idx, int sz) {
    if (!sz) return -1;
    if (idx < 0) return n;
    int &res = mem[idx][sz];
    if (vis[idx][sz]) return res;
    vis[idx][sz] = true;
    int leave = dp(idx - 1, sz);
    int take = dp(idx - 1, sz - 1);
    take = nxt[take + 1][t[idx] - 'a'];
    return res = min(take, leave);
}

void solve() {
    // min(n * n , m * m)
    // dp[idx][sz] min IDX in s when match string of size sz
    cin >> s >> t;
    n = s.size();
    m = t.size();
    vis.assign(m + 1, vector< bool >(m + 1, false));
    nxt.assign(n + 2, vector< int >(26, n));
    for (int i = n - 1; i >= 0; --i) {
        nxt[i] = nxt[i + 1];
        nxt[i][s[i] - 'a'] = i;
    }
    for (int i = m; i >= 0; --i) {
        if (dp(m - 1, i) < n) {
            cout << i << '\n';
            return;
        }
    }
}
```

LIS

```
//  s -> input array , LIS-> answer of lis ending at position i
void LIS(const vector<int> &S, vector<int> &LIS) {
  vector<int> L(S.size());
  int lisCount = 0;
  for (size_t i = 0; i < S.size(); ++i) {
    // if you need equal values change it to upper_bound
    int pos = lower_bound(L.begin(), L.begin() + lisCount, S[i]) -
    ↪   L.begin();
    L[pos] = S[i];
    if (pos == lisCount) ++lisCount;
    LIS[i] = pos + 1;
  }
}
```

Matrix Power

```
static const int mod = 1e9 + 7, M = 26;
typedef vector<vector<int>> matrix;
```

```
matrix mul(matrix a, matrix b) {
  matrix res = matrix(a.size(), vector<int>(b[0].size()));
  for (int i = 0; i < a.size(); i++) {
    for (int j = 0; j < b[0].size(); j++) {
      res[i][j] = 0;
      for (int k = 0; k < a.size(); k++) {
        res[i][j] = (res[i][j] + a[i][k] * b[k][j]) % mod;
      }
    }
  }
  return res;
}

matrix mat_power(matrix a, int p) {
  matrix res = matrix(a.size(), vector<int>(a.size(), 0));
  for (int i = 0; i < a.size(); i++) res[i][i] = 1;
  while (p) {
    if (p & 1) res = mul(res, a);
    a = mul(a, a);
    p /= 2;
  }
  return res;
}
```

Geometry

1-basics

```
typedef ld T;
typedef complex<T> pt;
#define x real()
#define y imag()
bool operator==(pt a, pt b) { return fabs(a.x - b.x) < EPS && fabs(a.y -
↪  b.y) < EPS; }
bool operator!=(pt a, pt b) { return !(a == b); }
int sgn(T val) { return (T(0) < val) - (val < T(0)); }
T sq(pt p) { return p.x * p.x + p.y * p.y; }
ld abs(pt p) { return sqrt(sq(p)); }
pt perp(pt p) { return {-p.y, p.x}; }
T dot(pt v, pt w) { return v.x * w.x + v.y * w.y; }
T cross(pt v, pt w) { return v.x * w.y - v.y * w.x; }
bool isPerp(pt v, pt w) { return dot(v, w) == 0; }
```

2-transformations

```
// scale point p by a factor around c
pt scale(pt c, T factor, pt p) { return c + (p - c) * factor; }
// To rotate point p by a certain angle \phi  around center c
pt rot(pt p, pt c, ld a) {
  pt v = p - c;
  return {c.x + v.x * cos(a) - v.y * sin(a), c.y + v.x * sin(a) + v.y *
  ↪  cos(a)};
}
// point p has image fp, point q has image fq then what is image of point r
pt linearTransfo(pt p, pt q, pt r, pt fp, pt fq) {
  pt pq = q - p, num{cross(pq, fq - fp), dot(pq, fq - fp)};
  return fp + pt{cross(r - p, num), dot(r - p, num)} / sq(pq);
}
```

3-angles

```
//(AB X AC) --> relative to AB: if(C right) ret neg else if (C left) pos
T orient(pt a, pt b, pt c) { return cross(b - a, c - a); }
// check p in between angle(bac) counter clockwise
bool inAngle(pt a, pt b, pt c, pt p) {
  T abp = orient(a, b, p), acp = orient(a, c, p), abc = orient(a, b, c);
  if (abc < 0) swap(abp, acp);
  return (abp >= 0 && acp <= 0) ^ (abc < 0);
}
// Get angle between V, W
ld angle(pt v, pt w) { return acos(clamp(dot(v, w) / abs(v) / abs(w),
↪  (T)-1.0, (T)1.0)); }
// calc BAC angle
ld orientedAngle(pt a, pt b, pt c) {
  if (orient(a, b, c) >= 0) return angle(b - a, c - a);
  else return 2 * M_PI - angle(b - a, c - a);
}
// amplitude travelled around point A, from P to Q
ld angleTravelled(pt a, pt p, pt q) {
  double ampli = angle(p - a, q - a);
  if (orient(a, p, q) > 0) return ampli;
  else return -ampli;
}
bool half(pt p) { return p.y > 0 || (p.y == 0 && p.x < 0); }
```

4-lines

```
struct line {
  pt v;
  T c;
  // From direction vector v and offset c
  line(pt v, T c) : v(v), c(c) {}
  // From equation ax+by=c
  line(T a, T b, T _c) {
    v = {b, -a};
    c = _c;
  }
  // From points P and Q
  line(pt p, pt q) { v = q - p, c = cross(v, p); }
  // - these work with T = int
  T side(pt p) { return cross(v, p) - c; }
  double dist(pt p) { return abs(side(p)) / abs(v); }
  double sqDist(pt p) { return side(p) * side(p) / (T)sq(v); }
  line perpThrough(pt p) { return {p, p + perp(v)}; }
  bool cmpProj(pt p, pt q) { return dot(v, p) < dot(v, q); }
  line translate(pt t) { return {v, c + cross(v, t)}; }
  // - these require T = double
  line shiftLeft(double dist) { return {v, c + dist * abs(v)}; }
  pt proj(pt p) { return p - perp(v) * side(p) / sq(v); }
  pt refl(pt p) { return p - perp(v) * (T)2.0 * side(p) / sq(v); }
};
// Two lines Intersection
bool inter(line l1, line l2, pt &out) {
  T d = cross(l1.v, l2.v);
  if (fabs(d) <= EPS) return false;
  out = (l2.v * l1.c - l1.v * l2.c) / d;  // requires floating-point
  ↪  coordinates
  return true;
}
// Bisector of Two lines (interior da hatl)
line bisector(line l1, line l2, bool interior) {
  assert(cross(l1.v, l2.v) != 0);  // l1 and l2 cannot be parallel!
  T sign = interior ? 1 : -1;
  return {l2.v / (T)abs(l2.v) + l1.v / (T)abs(l1.v) * sign,
          l2.c / abs(l2.v) + l1.c / abs(l1.v) * sign};
}
tuple<ll, ll, ll> normalize(pt p1, pt p2) {
  ll a = p2.y - p1.y;
  ll b = p1.x - p2.x;
  ll c = (p2.x * p1.y) - (p1.x * p2.y);
  ll gc = gcd(gcd(abs(a), abs(b)), abs(c));
  a /= gc;
  b /= gc;
  c /= gc;
  if (a < 0 || (a == 0 && b < 0)) {
    a *= -1;
    b *= -1;
    c *= -1;
  }
  return {a, b, c};
}
bool in_line(ll a, ll b, ll c, pt p) { return a * p.x + b * p.y + c == 0; }
```

5-segments

```
bool inDisk(pt a, pt b, pt p) { return dot(a - p, b - p) <= EPS; }
bool onSegment(pt a, pt b, pt p) {
  return fabsl(orient(a, b, p)) <= EPS && inDisk(a, b, p);
}
bool properInter(pt a, pt b, pt c, pt d, pt &out) {
  T oa = orient(c, d, a), ob = orient(c, d, b), oc = orient(a, b, c),
    od = orient(a, b, d);
  // Proper intersection exists iff opposite signs
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0) {
    out = (a * ob - b * oa) / (ob - oa);
    return true;
  }
  return false;
}
set<pair<ld, ld>> inters(pt a, pt b, pt c, pt d) {
  set<pair<ld, ld>> s;
  pt out;
  if (a == c || a == d) { s.insert(make_pair(a.x, a.y)); }
  if (b == c || b == d) { s.insert(make_pair(b.x, b.y)); }
  if (s.size()) return s;
  if (properInter(a, b, c, d, out)) return {make_pair(out.x, out.y)};
  if (onSegment(c, d, a)) s.insert(make_pair(a.x, a.y));
  if (onSegment(c, d, b)) s.insert(make_pair(b.x, b.y));
  if (onSegment(a, b, c)) s.insert(make_pair(c.x, c.y));
  if (onSegment(a, b, d)) s.insert(make_pair(d.x, d.y));
  return s;
}
```

```
ld segPoint(pt a, pt b, pt p) {
  if (a != b) {
    line l(a, b);
    if (l.cmpProj(a, p) && l.cmpProj(p, b))  // if closest to projection
      return l.dist(p);                      // output distance to line
  }
  return min(abs(p - a), abs(p - b));  // otherwise distance to A or B
}
ld segSeg(pt a, pt b, pt c, pt d) {
  pt dummy;
  if (properInter(a, b, c, d, dummy)) return 0;
  return min(
      {segPoint(a, b, c), segPoint(a, b, d), segPoint(c, d, a), segPoint(c,
      ↪  d, b)});
}
```

6-polygons

```
bool isConvex(vector<pt> p) {
  bool hasPos = false, hasNeg = false;
  for (int i = 0, n = p.size(); i < n; i++) {
    int o = orient(p[i], p[(i + 1) % n], p[(i + 2) % n]);
    if (o > 0) hasPos = true;
    if (o < 0) hasNeg = true;
  }
  return !(hasPos && hasNeg);
}
ld areaTriangle(pt a, pt b, pt c) { return abs(cross(b - a, c - a)) / 2.0; }
ld areaPolygon(vector<pt> p) {
  ld area = 0.0;
  for (int i = 0, n = p.size(); i < n; i++) {
    area += cross(p[i], p[(i + 1) % n]);  // wrap back to 0 if i == n - 1
  }
  return abs(area) / 2.0;
}
// true if P at least as high as A
bool above(pt a, pt p) { return p.y >= a.y; }
// check if [PQ] crosses ray from A
bool crossesRay(pt a, pt p, pt q) {
  return (above(a, q) - above(a, p)) * orient(a, p, q) > 0;
}
// if strict, returns false when A is on the boundary
bool inPolygon(vector<pt> p, pt a, bool strict = true) {
  int numCrossings = 0;
  for (int i = 0, n = p.size(); i < n; i++) {
    if (onSegment(p[i], p[(i + 1) % n], a)) return !strict;
    numCrossings += crossesRay(a, p[i], p[(i + 1) % n]);
  }
  return numCrossings & 1;  // inside if odd number of crossings
}
```

7-circles

```
pt circumCenter(pt a, pt b, pt c) {
  b -= a;
  c -= a;
  pt o = a + prep(b * sq(c) - c * sq(b)) / (2 * cross(b, c));
  return o;
}
int circleLine(pt o, double r, line l, pair<pt, pt> &out) {
  double h2 = r * r - l.sqDist(o);
  if (h2 >= 0) {                               // the line touches the circle
    pt p = l.proj(o);                          // point P
    pt h = l.v * (T)(sqrt(h2) / abs(l.v));     // vector parallel to l, of
    ↪  length h
    out = {p - h, p + h};
  }
  return 1 + sgn(h2);
}
int circleCircle(pt o1, T r1, pt o2, T r2, pair<pt, pt> &out) {
  pt d = o2 - o1;
  T d2 = sq(d);
  if (d2 == 0) {
    assert(r1 != r2);
    return 0;
  }  // concentric circles
  T pd = (d2 + r1 * r1 - r2 * r2) / 2;  // = |O_1P| * d
  T h2 = r1 * r1 - pd * pd / d2;        // = h^2
  if (h2 >= 0) {
    pt p = o1 + d * pd / d2, h = perp(d) * sqrt(h2 / d2);
    out = {p - h, p + h};
  }
  return 1 + sgn(h2);
}
int tangents(pt o1, T r1, pt o2, T r2, bool inner, vector<pair<pt, pt>>
↪  &out) {
  if (inner) r2 = -r2;
  pt d = o2 - o1;
  T dr = r1 - r2, d2 = sq(d), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0) {
    assert(h2 != 0);
    return 0;
  }
  for (T sign : {-1, 1}) {
    pt v = (d * dr + perp(d) * sqrt(h2) * sign) / d2;
    out.push_back({o1 + v * r1, o2 + v * r2});
  }
  return 1 + (h2 > 0);
}
ld dist(array<ld, 2> a, array<ld, 2> b) {
  return sqrt((a[0] - b[0]) * (a[0] - b[0]) + (a[1] - b[1]) * (a[1] -
  ↪   b[1]));
}
```

9.inConvex

```
bool inConvex(vector<pt>& v, pt p) {
    if (v.size() < 3) return 0;
    ll n = v.size();
    ll l = 1, r = n - 2, ans = 1;
    while (l <= r) {
        ll mid = l + (r - l) / 2;
        if (sgn(orient(v[0], v[mid], p)) > 0) {
            l = mid + 1;
            ans = mid;
        }
        else r = mid - 1;
    }
    return inPolygon({v[0], v[ans], v[ans + 1]}, p, 0);
}
```

CHT

```
struct Line {
    int m, b;
    mutable function<const Line *()> succ;
    bool operator<(const Line &other) const {
        return m < other.m;
    }
    bool operator<(const int &x) const {
        const Line *s = succ();
        if (not s) return false;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : multiset<Line, less<>> {
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return false;
            return y->m == z->m and y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end())
            return y->m == x->m and y->b <= x->b;
        return (long double) (x->b - y->b) * (z->m - y->m) >= (long double)
        ↪   (y->b - z->b) * (y->m - x->m);
    }
    void insert_line(int m, int b) {
        // for minimum
        m *= -1, b *= -1;
        auto y = insert({m, b});
        y->succ = [=] {
            return next(y) == end() ? 0 : &*next(y);
        };
        if (bad(y)) {
            erase(y);
            return;
        }
        while (next(y) != end() and bad(next(y)))
            erase(next(y));
        while (y != begin() and bad(prev(y)))
            erase(prev(y));
    }

    int eval(int x) {
        auto l = *lower_bound(x);
        // for minimum
        return -(l.m * x + l.b);
//        return l.m * x + l.b;
    }
};
```

```
void solve() {
    int n, x;
    cin >> n >> x;
    vector<int> s(n + 1), f(n + 1);
    for (int i = 1; i <= n; ++i)cin >> s[i];
    for (int i = 1; i <= n; ++i)cin >> f[i];
    vector<int> dp(n + 1);
    HullDynamic hul;
    hul.insert_line(x, 0);
    for (int i = 1; i <= n; ++i) {
        dp[i] = hul.eval(s[i]);
        hul.insert_line(f[i], dp[i]);
    }
    cout << dp[n];
}
```

geo addone

```
    // nearest 2 points
    sort(all(v), [&](pt& a, pt& b)->bool {
        return a.x < b.x;
    });
    set<array<ll, 2>> s;
    ll ans = 8e18 + 5, j = 0, bestsq = sqrt(ans);
    for (int i = 0; i < n; i++) {
        while (j < n && v[i].x - v[j].x > bestsq) {
            s.erase({v[j].y, v[j].x});
            j++;
        }
        auto st = s.lower_bound({v[i].y - bestsq, -oo});
        auto en = s.lower_bound({v[i].y + bestsq, -oo});
        if (en != s.end()) en++;
        while (st != en) {
            pt cur = {(*st)[1], (*st)[0]};
            ans = min(ans, sq(v[i] - cur));
            bestsq = min(bestsq, abs(v[i] - cur));
            st++;
        }
        s.insert({v[i].y, v[i].x});
    }
    // picks theoerm
    // area = inside + (boundry / 2) - 1
    ll laticeAB(pt a, pt b){
        return gcd(abs(a.x - b.x), abs(a.y - b.y));
    }
    ll boundry(vector<pt>& v) {
        ll n = v.size(), ans = 0;
        for (int i = 0; i < n; i++) {
            ans += laticeAB(v[i], v[(i + 1) % n]);
        }
        return ans;
    }
```

geo addone2

```
ld angle(pt v) { return atan2(v.y, v.x); }
pt rot(pt p, ld a) { return p * polar((ld) 1, a); }
int compare(ld a, ld b) {
    if (fabsl(a - b) < EPS) return 0;
    return (a < b) ? -1 : 1;
}
```

geo addone3

```
bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = sgn(orient(a, b, c));
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return sgn(orient(a, b, c)) == 0; }
void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = sgn(orient(p0, a, b));
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                   < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }
    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i], include_collinear))
            st.pop_back();
        if(st.empty() || a[i] != st.back())
            st.push_back(a[i]);
    }
    if (include_collinear == false && st.size() == 2 && st[0] == st[1])
        st.pop_back();
    a = st;
}
//////////////////////////////////////////////////////////////////////////////////////////////
void reorder_polygon(vector<pt> & P){
    size_t pos = 0;
    for(size_t i = 1; i < P.size(); i++){
        if(P[i].y < P[pos].y || (P[i].y == P[pos].y && P[i].x < P[pos].x))
            pos = i;
    }
    rotate(P.begin(), P.begin() + pos, P.end());
}
//p must be counter clockwise
vector<pt> minkowski(vector<pt> P, vector<pt> Q){
    // the first vertex must be the lowest
    reorder_polygon(P);
    reorder_polygon(Q);
    // we must ensure cyclic indexing
    P.push_back(P[0]);
    P.push_back(P[1]);
    Q.push_back(Q[0]);
    Q.push_back(Q[1]);
    // main part
    vector<pt> result;
    size_t i = 0, j = 0;
    while(i < P.size() - 2 || j < Q.size() - 2){
        result.push_back(P[i] + Q[j]);
        auto cross = (P[i + 1] - P[i]).cross(Q[j + 1] - Q[j]);
        if(cross >= 0 && i < P.size() - 2)
            ++i;
        if(cross <= 0 && j < Q.size() - 2)
            ++j;
    }
    return result;
}
```

```
//////////////////////////////////////////////////////////////////////////////////////////////
// Basic half-plane struct.
struct Halfplane {
    // 'p' is a passing point of the line and 'pq' is the direction vector of the line.
    pt p, pq;
    long double angle;
    Halfplane() {}
    Halfplane(const pt& a, const pt& b) : p(a), pq(b - a) {
        angle = atan2l(pq.y, pq.x);
    }
    // Check if point 'r' is outside this half-plane.
    // Every half-plane allows the region to the LEFT of its line.
    bool out(const pt& r) {
        return cross(pq, r - p) < -EPS;
    }
    // Comparator for sorting.
    bool operator < (const Halfplane& e) const {
        return angle < e.angle;
    }
    // Intersection point of the lines of two half-planes. It is assumed they're never parallel.
    friend pt inter(const Halfplane& s, const Halfplane& t) {
        long double alpha = cross((t.p - s.p), t.pq) / cross(s.pq, t.pq);
        return s.p + (s.pq * alpha);
    }
};
vector<pt> hp_intersect(vector<Halfplane>& H) {
    const int inf = 1e9;
    pt box[4] = {  // Bounding box in CCW order
            pt(inf, inf),
            pt(-inf, inf),
            pt(-inf, -inf),
            pt(inf, -inf)
    };
    for(int i = 0; i<4; i++) { // Add bounding box half-planes.
        Halfplane aux(box[i], box[(i+1) % 4]);
        H.push_back(aux);
    }
    // Sort by angle and start algorithm
    sort(H.begin(), H.end());
    deque<Halfplane> dq;
    int len = 0;
    for(int i = 0; i < int(H.size()); i++) {
        // Remove from the back of the deque while last half-plane is
        ↪   redundant
        while (len > 1 && H[i].out(inter(dq[len-1], dq[len-2]))) {
            dq.pop_back();
            --len;
        }
        // Remove from the front of the deque while first half-plane is
        ↪   redundant
        while (len > 1 && H[i].out(inter(dq[0], dq[1]))) {
            dq.pop_front();
            --len;
        }
        // Special case check: Parallel half-planes
        if (len > 0 && fabsl(cross(H[i].pq, dq[len-1].pq)) < EPS) {
            // Opposite parallel half-planes that ended up checked against
            ↪   each other.
            if (dot(H[i].pq, dq[len-1].pq) < 0.0)
                return vector<pt>();
            // Same direction half-plane: keep only the leftmost half-plane.
            if (H[i].out(dq[len-1].p)) {
                dq.pop_back();
                --len;
            }
            else continue;
        }
        // Add new half-plane
        dq.push_back(H[i]);
        ++len;
    }
    // Final cleanup: Check half-planes at the front against the back and
    ↪   vice-versa
    while (len > 2 && dq[0].out(inter(dq[len-1], dq[len-2]))) {
        dq.pop_back();
        --len;
    }
    while (len > 2 && dq[len-1].out(inter(dq[0], dq[1]))) {
        dq.pop_front();
        --len;
    }
    // Report empty intersection if necessary
    if (len < 3) return vector<pt>();
    // Reconstruct the convex polygon from the remaining half-planes.
    vector<pt> ret(len);
```

```
    for(int i = 0; i+1 < len; i++) {
        ret[i] = inter(dq[i], dq[i+1]);
    }
    ret.back() = inter(dq[len-1], dq[0]);
    return ret;
}
///////////////////////////////////////////////////////////////////////////
↪   /////////////
vector<pair<int, int>> all_anti_podal(int n, vector<pt> &p) {
    vector<pair<int, int>> result;
    auto nx = [&](int i){return (i+1)%n;};
    auto pv = [&](int i){return (i-1+n)%n;};
    // parallel edges should't be visited twice
    vector<bool> vis(n, false);
    for (int p1 = 0, p2 = 0; p1 < n; ++p1) {
        // the edge that we are going to consider in this iteration
        // the datatype is Point, but it acts as a vector
        pt base = p[nx(p1)] - p[p1];
        // the last condition makes sure that the cross products don't have
        ↪   the same sign
        while (p2 == p1 || p2 == nx(p1) || sgn(cross(base, p[nx(p2)] -
        ↪   p[p2])) == sgn(cross(base, p[p2] - p[pv(p2)]))) {
            p2 = nx(p2);
        }
        if (vis[p1]) continue;
        vis[p1] = true;
        result.push_back({p1, p2});
        result.push_back({nx(p1), p2});
        // if both edges from p1 and p2 are parallel to each other
        if (sgn(cross(base, p[nx(p2)] - p[p2])) == 0) {
            result.push_back({p1, nx(p2)});
            result.push_back({nx(p1), nx(p2)});
            vis[p2] = true;
        }
    }
    return result;
}

// maximum distance from a convex polygon to another convex polygon
double maximum_dist_from_polygon_to_polygon(vector<PT> &u, vector<PT> &v){
↪   //O(n)
    int n = (int)u.size(), m = (int)v.size();
    double ans = 0;
    if (n < 3 || m < 3) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) ans = max(ans, dist2(u[i], v[j]));
        }
        return sqrt(ans);
    }
    if (u[0].x > v[0].x) swap(n, m), swap(u, v);
    int i = 0, j = 0, step = n + m + 10;
    while (j + 1 < m && v[j].x < v[j + 1].x) j++ ;
    while (step--) {
        if (cross(u[(i + 1)%n] - u[i], v[(j + 1)%m] - v[j]) >= 0) j = (j +
         ↪  1) % m;
        else i = (i + 1) % n;
        ans = max(ans, dist2(u[i], v[j]));
    }
    return sqrt(ans);
}
```

Triangle SharedArea

```
const double EPS = 1e-9;
struct Point {
  double x, y;
  bool operator==(const Point &p) const {
    return fabs(x - p.x) < EPS && fabs(y - p.y) < EPS;
  }
};
double cross(Point a, Point b) { return a.x * b.y - a.y * b.x; }
Point operator-(Point a, Point b) { return {a.x - b.x, a.y - b.y}; }
// Shoelace formula for polygon area
double polygon_area(const vector<Point> &pts) {
  double area = 0;
  int n = pts.size();
  for (int i = 0; i < n; ++i) area += cross(pts[i], pts[(i + 1) % n]);
  return fabs(area) / 2;
}
// Check if point p is inside triangle abc
bool point_in_triangle(Point p, Point a, Point b, Point c) {
  double A = fabs(cross(b - a, c - a));
  double A1 = fabs(cross(a - p, b - p));
  double A2 = fabs(cross(b - p, c - p));
  double A3 = fabs(cross(c - p, a - p));
  return fabs(A - (A1 + A2 + A3)) < EPS;
}
// Segment-segment intersection
bool seg_intersect(Point a, Point b, Point c, Point d, Point &out) {
  Point r = b - a, s = d - c;
  double denom = cross(r, s);
  if (fabs(denom) < EPS) return false;  // Parallel or colinear
  double t = cross(c - a, s) / denom;
  double u = cross(c - a, r) / denom;
  if (t >= -EPS && t <= 1 + EPS && u >= -EPS && u <= 1 + EPS) {
    out = {a.x + t * r.x, a.y + t * r.y};
    return true;
  }
  return false;
}
// Remove duplicate points
void unique_points(vector<Point> &pts) {
  sort(pts.begin(), pts.end(), [](Point a, Point b) {
    return a.x < b.x - EPS || (fabs(a.x - b.x) < EPS && a.y < b.y - EPS);
  });
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
}
// Sort points around centroid (for convex polygon area)
void sort_ccw(vector<Point> &pts) {
  Point center{0, 0};
  for (auto &p : pts) {
    center.x += p.x;
    center.y += p.y;
  }
  center.x /= pts.size();
  center.y /= pts.size();
  sort(pts.begin(), pts.end(), [&](Point a, Point b) {
    double angleA = atan2(a.y - center.y, a.x - center.x);
    double angleB = atan2(b.y - center.y, b.x - center.x);
    return angleA < angleB;
  });
}
// Main function: compute shared area of two triangles
double shared_triangle_area(Point A, Point B, Point C, Point D, Point E,
 ↪  Point F) {
  vector<Point> poly;
  // Add all intersection points between triangle edges
  Point out;
  vector<pair<Point, Point>> edges1 = {{A, B}, {B, C}, {C, A}};
  vector<pair<Point, Point>> edges2 = {{D, E}, {E, F}, {F, D}};
  for (auto &[p1, p2] : edges1) {
    for (auto &[q1, q2] : edges2) {
      if (seg_intersect(p1, p2, q1, q2, out)) poly.push_back(out);
    }
  }
  // Add vertices of triangle 1 inside triangle 2
  for (Point p : {A, B, C})
    if (point_in_triangle(p, D, E, F)) poly.push_back(p);
  // Add vertices of triangle 2 inside triangle 1
  for (Point p : {D, E, F})
    if (point_in_triangle(p, A, B, C)) poly.push_back(p);
  unique_points(poly);
  if (poly.size() < 3) return 0.0;  // No overlapping area
  sort_ccw(poly);
  return polygon_area(poly);
}
signed main() {
  double a, b, c;
  cin >> a >> b >> c;
  Point A = {0, 0}, B = {0, c}, C = {c, 0};
  Point D = {0, 0}, E = {a, 0}, F = {0, b};
  double valid = shared_triangle_area(A, B, C, D, E, F);
  cout << valid << '\n';
}
```

Number Theory

Euler Totient

```
/*
 * some facts
 * 1. \phi (p^k) = p^k - p^(k-1) where p is prime
 * 2. \phi (ab) = \phi (a) \phi (b) where a and b are coprime
 * form 1 and 2 -> 3
 * 3. \phi (n)= p1^(k1-1)*(p1 - 1) * p2^(k2-1)*(p2 - 1) * p3^(k3-1)*(p3 - 1)
 * ......
 * 4. \sum (d|n) \phi (d) = n
 */
vector<int> compute_phi(int n) {
  // \sum (d|n) \phi (d) = n
  // using the fact that sum of phi[divisors of n] = n
  vector<int> phi(n + 1);
  for (int i = 1; i <= n; i++) {
    phi[i] += i;
    for (int j = 2 * i; j <= n; j += i) { phi[j] -= phi[i]; }
  }
  return phi;
}
vector<int> linear_phi(int n) {
  vector<int> lp(n + 1);
  vector<int> pr;
  vector<int> phi(n + 1);
  phi[1] = 1;
  for (int i = 2; i < n; ++i) {
    if (!lp[i]) {
      pr.push_back(i);
      phi[i] = i - 1;  // i is pr
    }
    for (int j = 0; j < pr.size() && i * pr[j] <= n; ++j) {
      lp[i * pr[j]] = true;
      if (i % pr[j] == 0) {
        phi[i * pr[j]] = phi[i] * pr[j];  // pr[j] divides i
        break;
      } else {
        phi[i * pr[j]] = phi[i] * phi[pr[j]];  // pr[j] does not divide i
      }
    }
  }
  return phi;
}
int phi(int n) {
  vector<pair<int, int>> divisors = PrimeFact(n);
  // pairs {prime number, exponent}
  int ans = 1;
  for (auto [prime, exp] : divisors) {
    int power = 1;
    for (int i = 1; i < exp; i++) { power *= prime; }
    ans *= (power * prime - power);  // (p^exp - p^{exp-1})
  }
  return ans;
}
int phi(int n) {
  int result = n == 1 ? 0 : n;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      while (n % i == 0) n /= i;
      result -= result / i;
    }
  }
  if (n > 1) result -= result / n;
  return result;
}
```

Extended Euclidean

```
/*
 * http://e-maxx.ru/algo/diofant_2_equation&usg=ALkJrhhAzF9yCVA7pOjdWhVRIFd
 ↪  PsBlzmA
 */
int extended_gcd(int a, int b, int &x, int &y) {
  if (a == 0) {
    x = 0;
    y = 1;
    return b;
  }
  int x1, y1;
  int g = extended_gcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return g;
}
bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
  g = extended_gcd(abs(a), abs(b), x0, y0);
  if (c % g != 0) return false;
  x0 *= c / g;
  y0 *= c / g;
  if (a < 0) x0 *= -1;
  if (b < 0) y0 *= -1;
  return true;
}
void shift_solution(int &x, int &y, int a, int b, int cnt) {
  x += cnt * b;
  y -= cnt * a;
  // ax + by = c
  // x += k * b / g
  // y -= k * a / g
}
int find_all_solutions(int a, int b, int c, int minx, int maxx, int miny,
↪  int maxy) {
  int x, y, g;
  if (!find_any_solution(a, b, c, x, y, g)) return 0;
  a /= g;
  b /= g;
  int sign_a = a > 0 ? +1 : -1;
  int sign_b = b > 0 ? +1 : -1;
  shift_solution(x, y, a, b, (minx - x) / b);
  if (x < minx) shift_solution(x, y, a, b, sign_b);
  if (x > maxx) return 0;
  int lx1 = x;
  shift_solution(x, y, a, b, (maxx - x) / b);
  if (x > maxx) shift_solution(x, y, a, b, -sign_b);
  int rx1 = x;
  shift_solution(x, y, a, b, -(miny - y) / a);
  if (y < miny) shift_solution(x, y, a, b, -sign_a);
  if (y > maxy) return 0;
  int lx2 = x;
  shift_solution(x, y, a, b, -(maxy - y) / a);
  if (y > maxy) shift_solution(x, y, a, b, sign_a);
  int rx2 = x;
  if (lx2 > rx2) swap(lx2, rx2);
  int lx = max(lx1, lx2);
  int rx = min(rx1, rx2);
  //    for (int curx = lx; curx <= rx; curx += abs(b)) {
  //        int cury = (c - a * curx) / b;
  //        if (cury >= miny && cury <= maxy)
  //            solutions.emplace_back(curx, cury);
  //    }
  return (rx - lx) / abs(b) + 1;
}
```

Factorization

```
vector<int> Factorize(int n) {
  vector<int> a;
  for (int i = 1; i * i <= n; ++i) {
    if (n % i == 0) {
      a.push_back(i);
      if (i * i != n) a.push_back(n / i);
    }
  }
  sort(a.begin(), a.end());
  return a;
}
```

isPrime

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

linear sieve

```
vector<int> linear_sieve(int n) {
  vector<int> lp(n + 1);
  vector<int> pr;
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = i;
      pr.push_back(i);
    }
    for (int j = 0; i * pr[j] <= n; ++j) {
      lp[i * pr[j]] = pr[j];
      if (pr[j] == lp[i]) break;
    }
  }
  return lp;
}
```

Pollard

```
using ul = uint64_t;
using db = long double;
class Pollard {
  public:
  map<ul, int> cnt_primes;
  vector<ul> primes, divisors;
  ul modMul(ul a, ul b, const ul mod) {
    long long ret = a * b - mod * (ul)((db)a * b / mod);
    return ret + ((ret < 0) - (ret >= (long long)mod)) * mod;
  }
  ul modPow(ul a, ul b, const ul mod) {
    if (b == 0) return 1;
    ul res = modPow(a, b / 2, mod);
    res = modMul(res, res, mod);
    return b & 1 ? modMul(res, a, mod) : res;
  }
  bool rabin_miller(ul n) {  // not ll!
    if (n < 2 || n % 6 % 4 != 1) return n - 2 < 2;
    ul A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
       s = __builtin_ctzll(n - 1), d = n >> s;
    for (auto a : A) {  // ^ count trailing zeroes
      ul p = modPow(a, d, n), i = s;
      while (p != 1 && p != n - 1 && a % n && i--) p = modMul(p, p, n);
      if (p != n - 1 && i != s) return false;
    }
    return true;
  }
  ul pollard(ul n) {  // return some nontrivial factor of n
    auto f = [n, this](ul x) { return modMul(x, x, n) + 1; };
    ul x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 || __gcd(prd, n) == 1) {  /// speedup: don't take gcd every it
      if (x == y) x = ++i, y = f(x);
      if ((q = modMul(prd, max(x, y) - min(x, y), n))) prd = q;
      x = f(x), y = f(f(y));
    }
    return __gcd(prd, n);
  }
  void factor_rec(ul n, map<ul, int> &cnt) {
    if (n == 1) return;
    if (rabin_miller(n)) {
      ++cnt[n];
      return;
    }
    ul u = pollard(n);
    factor_rec(u, cnt), factor_rec(n / u, cnt);
  }
  void calcDivisorsRec(ul cur, int i) {
    if (i >= primes.size()) {
      divisors.push_back(cur);
      return;
    }
    int r = cnt_primes[primes[i]];
    for (int j = 0; j <= r; j++) {
      calcDivisorsRec(cur, i + 1);
      cur = cur * primes[i];
    }
  }
  void calcDivisors(ul x) {
    cnt_primes.clear();
    primes.clear();
    divisors.clear();
    factor_rec(x, cnt_primes);
    for (auto &u : cnt_primes) { primes.push_back(u.first); }
    calcDivisorsRec(1, 0);
  }
} pollard;
```

Prime Factorization

```
vector<pair<int, int>> PrimeFact(int n) {
  vector<pair<int, int>> a;
  for (int i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      int cnt = 0;
      while (n % i == 0) {
        cnt++;
        n /= i;
      }
      a.emplace_back(i, cnt);
    }
  }
  if (n > 1) a.emplace_back(n, 1);
  return a;
}
```

primitive roots

```
int exp(int Base, int Power, int n) {
  int Result = 1;
  while (Power) {
    if (Power & 1) Result = (Result * Base) % n;
    Base = (Base * Base) % n, Power >>= 1;
  }
  return Result;
}
int phi(int n) {
  int result = n;
  for (int p = 2; p * p <= n; ++p) {
    if (n % p == 0) {
      while (n % p == 0) n /= p;
      result -= result / p;
    }
  }
  if (n > 1) result -= result / n;
  return result;
}
vector<int> PrimeFact(int n) {
  vector<int> a;
  for (int i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      int cnt = 0;
      while (n % i == 0) cnt++, n /= i;
      a.emplace_back(i);
    }
  }
  if (n > 1) a.emplace_back(n);
  return a;
}
bool is_primitive_root(int g, int n, int phi_n, const vector<int>& factors)
↪  {
  for (auto factor : factors) {
    if (exp(g, phi_n / factor, n) == 1) return false;
  }
  return true;
}
vector<int> get_primitive_roots(int n) {
  // any prime number has phi(n - 1) primitive root
  vector<int> roots;
  int phi_n = phi(n);
  vector<int> factors = PrimeFact(phi_n);
  for (int g = 2; g < n; ++g) {
    if (gcd(g, n) == 1 && is_primitive_root(g, n, phi_n, factors)) {
    ↪  roots.push_back(g); }
  }
  return roots;
}
void solve() {
  auto x = get_primitive_roots(100057);
  for (auto i : x) cout << i << ' ';
}
```

ProductOfDivs

```
int ProdOfDivs_FromPrimes(vector<pair<int, int>> primes) {
  int res = 1, d = 1;
  for (auto [pr, cnt] : primes) {
    int v = exp(pr, cnt * (cnt + 1) / 2);
    res = exp(res, cnt + 1) * exp(v, d) % mod;
    d = d * (cnt + 1) % (mod - 1);
  }
  return res;
}
```

Sieve

```
vector<int> Sieve(int n) {
  vector<int> sieve(n + 1, -1);
  for (int i = 2; i * i <= n; ++i) {
    if (sieve[i] == -1)
      for (int j = i; j <= n; j += i) sieve[j] = i;
  }
  return sieve;
}
```

Math

Big int Multiply

```
string mul_two_big_int(const string &s1, const string &s2) {
  int n = s1.size(), m = s2.size();
  vector<int> poly1(n), poly2(m);
  for (int i = 0; i < n; ++i) { poly1[n - i - 1] = s1[i] - '0'; }
  for (int i = 0; i < m; ++i) { poly2[m - i - 1] = s2[i] - '0'; }
  vector<int> ans = multiply(poly1, poly2);  // using FFT
  int k = ans.size();
  for (int i = 0; i < k - 1; ++i) {
    ans[i + 1] += ans[i] / 10;
    ans[i] = ans[i] % 10;
  }
  string final = to_string(ans[k - 1]);
  for (int i = k - 2; i >= 0; --i) { final += (char)(ans[i] + '0'); }
  for (int i = 0; i < k; ++i) {
    if (final[i] != '0') return final.substr(i);
  }
  return "0";
}
string power_of_big_int(string s, int p) {
  string ans = "1";
  while (p) {
    if (p & 1) ans = mul_two_big_int(ans, s);
    s = mul_two_big_int(s, s);
    p >>= 1;
  }
  return ans;
}
```

Fast String Matching

```
using cd = complex<double>;
// If you get a wrong answer you can change the eps lower of higher till you
// pass
const double PI = acos(-1), eps = 5e-4;
void fft(vector<cd>& a, bool invert) {
  int n = a.size();
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1) j ^= bit;
    j ^= bit;
    if (i < j) swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI / len * (invert ? -1 : 1);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
        cd u = a[i + j], v = a[i + j + len / 2] * w;
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
        w *= wlen;
      }
    }
  }
  if (invert) {
    for (cd& x : a) x /= n;
  }
}
vector<cd> multiply(vector<cd> const& a, vector<cd> const& b) {
  vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while (n < (int)a.size() + (int)b.size()) n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++) fa[i] *= fb[i];
  fft(fa, true);
  return fa;
}
void solve() {
  string s, patt;
  cin >> s >> patt;
  int n = (int)s.length(), m = (int)patt.length();
  vector<cd> poly1(n), poly2(m);
  for (int i = 0; i < n; ++i) {
    double angle = 2 * PI * (s[i] - 'a') / 26;
    poly1[i] = cd(cos(angle), sin(angle));
  }
  for (int i = 0; i < m; ++i) {
    if (patt[m - i - 1] == '*') poly2[i] = cd(0, 0);  // Wild Card
    else {
      double angle = 2 * PI * (patt[m - i - 1] - 'a') / 26;
      poly2[i] = cd(cos(angle), -sin(angle));
    }
  }
  vector<cd> ans = multiply(poly1, poly2);
  int wild_cnt = (int)count(patt.begin(), patt.end(), '*');
  int tot = 0;
  vector<int> pos;
  for (int i = 0; i < n; ++i) {
    if (fabs(ans[m - 1 + i].real() - (m - wild_cnt)) < eps &&
        fabs(ans[m - 1 + i].imag()) < eps) {
      ++tot;
      pos.push_back(i);
    }
  }
  cout << tot << "\n";
  for (auto& p : pos) cout << p << " ";
  cout << "\n";
}
```

FFT

```
const int limit = 1e6 + 1;
using cd = complex<double>;
const double PI = acos(-1);
void fft(vector<cd> &a, bool invert) {
  int n = a.size();
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1) j ^= bit;
    j ^= bit;
    if (i < j) swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI / len * (invert ? -1 : 1);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
        cd u = a[i + j], v = a[i + j + len / 2] * w;
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
        w *= wlen;
      }
    }
  }
  if (invert) {
    for (cd &x : a) x /= n;
  }
}
vector<int> multiply(vector<int> const &a, vector<int> const &b) {
  vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while (n < (int)(a.size() + b.size())) n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++) fa[i] *= fb[i];
  fft(fa, true);
  vector<int> result(n);
  for (int i = 0; i < n; i++) result[i] = llround(fa[i].real());
  result.resize(min((int)(a.size() + b.size()), limit));
  return result;
}
vector<int> poly_pow(vector<int> poly, int p, int limit = 1e9) {
  vector<int> ans{1};
  while (p) {
    if (p & 1) ans = conv(ans, poly);
    poly = conv(poly, poly);
    ans.resize(limit + 1);
    poly.resize(limit + 1);
    p >>= 1;
  }
  return ans;
}
vector<int> power(vector<int> &a, int b) {
  vector<int> res = {1};
  while (b) {
    if (b & 1) res = multiply(res, a);
    a = multiply(a, a), b >>= 1;
  }
  return res;
}
vector<int> multiply_moreThanOne(vector<vector<int>> polys) {
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
  for (int i = 0; i < polys.size(); ++i) pq.emplace(polys[i].size(), i);
  while (pq.size() > 1) {
    int a = pq.top().second;
    pq.pop();
    int b = pq.top().second;
    pq.pop();
    auto res = multiply(polys[a], polys[b]);
    polys[a] = res;
    pq.emplace(res.size(), a);
  }
  return polys[pq.top().second];
}
```

FFT MOD

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C> &a) {
  int n = a.size(), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n);
    rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    for (int i = k; i < 2 * k; ++i) rt[i] = R[i] = i & 1 ? R[i / 2] * x :
    ↪  R[i / 2];
  }
  vector<int> rev(n);
  for (int i = 0; i < n; ++i) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2) {
    for (int i = 0; i < n; i += 2 * k) {
      for (int j = 0; j < k; ++j) {
        // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rointed)  ///
        // include-line
        auto x = (double *)&rt[j + k],
             y = (double *)&a[i + j + k];  /// exclude-line
        C z(x[0] * y[0] - x[1] * y[1],
            x[0] * y[1] + x[1] * y[0]);  /// exclude-line
        a[i + j + k] = a[i + j] - z;
        a[i + j] += z;
      }
    }
  }
}
template <int M>
vector<int> multiply(const vector<int> &a, const vector<int> &b) {
  if (a.empty() || b.empty()) return {};
  vector<int> res(a.size() + b.size() - 1);
  int B = 32 - __builtin_clz(res.size()), n = 1 << B, cut = (int)(sqrt(M));
  vector<C> L(n), R(n), outs(n), outl(n);
  for (int i = 0; i < a.size(); ++i) L[i] = C((int)a[i] / cut, (int)a[i] %
  ↪  cut);
  for (int i = 0; i < b.size(); ++i) R[i] = C((int)b[i] / cut, (int)b[i] %
  ↪  cut);
  fft(L), fft(R);
  for (int i = 0; i < n; ++i) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  for (int i = 0; i < res.size(); ++i) {
    int av = (int)(real(outl[i]) + .5), cv = (int)(imag(outs[i]) + .5);
    int bv = (int)(imag(outl[i]) + .5) + (int)(real(outs[i]) + .5);
    res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
  }
  return res;
}
```

findFastMinSumSolution

```
bool findFastMinSumSolution(int a, int b, int c, int &minX, int &minY) {
    int x0, y0, g; //func:find_any_solution() <- Extended Euclidean.cpp
    if (!find_any_solution(a, b, c, x0, y0, g)) return false;
    int dx = b / g;
    int dy = a / g;
    // Bounds for k to keep x and y non-negative
    int k_min = (int) ceil((double) (-x0) / dx);
    int k_max = (int) floor((double) (y0) / dy);
    if (k_min > k_max) return false;
    // delta tells how x+y changes with k
    int delta = dx - dy, k_opt;
    if (delta > 0) k_opt = k_min;
    else if (delta < 0) k_opt = k_max;
    else k_opt = k_min; // any k in range gives same sum
    minX = x0 + k_opt * dx;
    minY = y0 - k_opt * dy;
    return true;
}
```

FWHT

```
// size of arrays must be power of 2
void fwht(vector<__int128> &a, int inv, int f) {
  int sz = a.size();
  for (int len = 1; 2 * len <= sz; len <<= 1) {
    for (int i = 0; i < sz; i += 2 * len) {
      for (int j = 0; j < len; j++) {
        __int128 x = a[i + j];
        __int128 y = a[i + j + len];
        if (f == 0) {
          if (!inv) a[i + j] = y, a[i + j + len] = x + y;
          else a[i + j] = y - x, a[i + j + len] = x;
        } else if (f == 1) {
          if (!inv) a[i + j + len] = x + y;
          else a[i + j + len] = y - x;
        } else {
          a[i + j] = x + y;
          a[i + j + len] = x - y;
        }
      }
    }
  }
}
vector<__int128> mul(vector<__int128> a, vector<__int128> b,
                     int f) {  // 0:AND, 1:OR, 2:XOR
  int sz = a.size();
  fwht(a, 0, f);
  fwht(b, 0, f);
  vector<__int128> c(sz);
  for (int i = 0; i < sz; ++i) { c[i] = a[i] * b[i]; }
  fwht(c, 1, f);
  if (f) {
    for (int i = 0; i < sz; ++i) { c[i] = c[i] / sz; }
  }
  return c;
}
```

NTT

```
const int mod = (119 << 23) + 1, root = 62;  // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 << 21
// and 483 << 21 (same root). The last two are > 10^9.
int modpow(int b, int e) {
  int ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    if (e & 1) ans = ans * b % mod;
  return ans;
}
// Primitive Root of the mod of form 2^a * b + 1
int generator() {
  vector<int> fact;
  int phi = mod - 1, n = phi;
  for (int i = 2; i * i <= n; ++i)
    if (n % i == 0) {
      fact.push_back(i);
      while (n % i == 0) n /= i;
    }
  if (n > 1) fact.push_back(n);
  for (int res = 2; res <= mod; ++res) {
    bool ok = true;
    for (size_t i = 0; i < fact.size() && ok; ++i) ok &= modpow(res, phi /
    ↪  fact[i]) != 1;
    if (ok) return res;
  }
  return -1;
```

```
}
int modpow(int b, int e, int m) {
  int ans = 1;
  for (; e; b = (int)b * b % m, e /= 2)
    if (e & 1) ans = (int)ans * b % m;
  return ans;
}
void ntt(vector<int> &a) {
  int n = (int)a.size(), L = 31 - __builtin_clz(n);
  vector<int> rt(2, 1);  // erase the static if you want to use two moduli;
  for (int k = 2, s = 2; k < n;
       k *= 2, s++) {  // erase the static if you want to use two moduli;
    rt.resize(n);
    int z[] = {1, modpow(root, mod >> s, mod)};
    for (int i = k; i < 2 * k; ++i) rt[i] = (int)rt[i / 2] * z[i & 1] % mod;
  }
  vector<int> rev(n);
  for (int i = 0; i < n; ++i) { rev[i] = (rev[i / 2] | (i & 1) << L) / 2; }
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2) {
    for (int i = 0; i < n; i += 2 * k) {
      for (int j = 0; j < k; ++j) {
        int z = (int)rt[j + k] * a[i + j + k] % mod, &ai = a[i + j];
        a[i + j + k] = ai - z + (z > ai ? mod : 0);
        ai += (ai + z >= mod ? z - mod : z);
      }
    }
  }
}
vector<int> multiply(const vector<int> &a, const vector<int> &b) {
  if (a.empty() || b.empty()) return {};
  int s = (int)a.size() + (int)b.size() - 1, B = 32 - __builtin_clz(s), n =
  ↪  1 << B;
  int inv = modpow(n, mod - 2, mod);
  vector<int> L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  for (int i = 0; i < n; ++i) out[-i & (n - 1)] = (int)L[i] * R[i] % mod *
  ↪  inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
vector<int> shift_poly(vector<int> &p, int k) {  // using NTT
  k = (k % mod + mod) % mod;
  int n = p.size();
  vector<int> fact(n, 1), inv(n, 1);
  for (int i = 1; i < n; ++i) {
    fact[i] = fact[i - 1] * i % mod;
    inv[i] = modpow(fact[i], mod - 2);
  }
  vector<int> p1(n), p2(n);
  for (int i = 0; i < n; i++) p1[i] = fact[i] * p[i] % mod;
  int curr = 1;
  for (int i = 0; i < n; i++) {
    p2[n - i - 1] = inv[i] * curr % mod;
    curr = curr * k % mod;
  }
  vector<int> res = multiply(p1, p2);
  vector<int> ans;
  for (int i = n - 1; i < res.size(); i++) ans.push_back(res[i] * inv[i -
  ↪  (n - 1)] % mod);
  return ans;
}
int CRT(int a, int m1, int b, int m2) {
  __int128 m = m1 * m2;
  int ans =
      a * m2 % m * modpow(m2, m1 - 2, m1) % m + m1 * b % m * modpow(m1, m2
      ↪  - 2, m2) % m;
  return ans % m;
}
int mod, root, desired_mod = 1000000007;
const int mod1 = 167772161;
const int mod2 = 469762049;
const int mod3 = 754974721;
const int root1 = 3;
const int root2 = 3;
const int root3 = 11;
int CRT(int a, int b, int c, int m1, int m2, int m3) {
  __int128 M = (__int128)m1 * m2 * m3;
  int M1 = (int)m2 * m3;
  int M2 = (int)m1 * m3;
  int M3 = (int)m2 * m1;
  int M_1 = modpow(M1 % m1, m1 - 2, m1);
  int M_2 = modpow(M2 % m2, m2 - 2, m2);
  int M_3 = modpow(M3 % m3, m3 - 2, m3);
  __int128 ans = (__int128)a * M1 * M_1;
  ans += (__int128)b * M2 * M_2;
  ans += (__int128)c * M3 * M_3;
  return (ans % M) % desired_mod;
}
```

slow String Matching

```
using cd = complex<double>;
const double PI = acos(-1);
void fft(vector<cd>& a, bool invert) {
  int n = a.size();
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1) j ^= bit;
    j ^= bit;
    if (i < j) swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI / len * (invert ? -1 : 1);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
        cd u = a[i + j], v = a[i + j + len / 2] * w;
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
        w *= wlen;
      }
    }
  }
  if (invert) {
    for (cd& x : a) x /= n;
  }
}
vector<int> multiply(vector<int> const& a, vector<int> const& b) {
  vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while (n < (int)a.size() + (int)b.size()) n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++) fa[i] *= fb[i];
  fft(fa, true);
  vector<int> result(n);
  for (int i = 0; i < n; i++) result[i] = round(fa[i].real());
  return result;
}
void solve() {
  string s, t;
  cin >> s >> t;
  int n = s.size(), m = t.size();
  string T = "ACTG";
  vector<int> match(n + 1);
  for (auto c : T) {
    vector<int> a(n + 5), b(m + 5);
    for (int i = 0; i < n; ++i) a[i] = s[i] == c;
    for (int i = 0; i < m; ++i) b[m - i] = t[i] == c;
    auto res = multiply(a, b);
    for (int i = 0; i <= n - m; ++i) match[i] += res[i + m];
  }
  int answer = 1e9;
  for (int i = 0; i < n; ++i) answer = min(answer, m - match[i]);
  cout << answer;
}
```

solveLinearEquations

```
const int inf = 1e18;
pair<int, int> solveLinearEquations(double a1, double b1, double c1, double
↪ a2, double b2,
                                    double c2) {
  double determinant = a1 * b2 - a2 * b1;
  pair<int, int> result{};
  if (fabs(determinant) < 1e-10) {
    if (fabs(a1 * c2 - a2 * c1) < 1e-10 && fabs(b1 * c2 - b2 * c1) < 1e-10)
      result = {inf, inf};       // infinite number of solutions
    else result = {-inf, -inf};  // no solution
  } else {
    double x = (b2 * c1 - b1 * c2) / determinant;
    double y = (a1 * c2 - a2 * c1) / determinant;
    result = {x, y};
  }
  return result;
}
pair<int, int> solveLinearEquations(int a1, int b1, int c1, int a2, int b2,
↪  int c2) {
  __int128 determinant = (__int128)a1 * b2 - (__int128)a2 * b1;
  __int128 x = ((__int128)b2 * c1 - (__int128)b1 * c2) / determinant;
  __int128 y = ((__int128)a1 * c2 - (__int128)a2 * c1) / determinant;
  return {x, y};
}
pair<int, int> solveLinearEquations(int a1, int b1, int c1, int a2, int b2,
↪  int c2) {
  __int128 determinant = (__int128)a1 * b2 - (__int128)a2 * b1;
  if (determinant == 0) return {0, 0};  // No unique solution
  __int128 x = ((__int128)b2 * c1 - (__int128)b1 * c2);
  __int128 y = ((__int128)a1 * c2 - (__int128)a2 * c1);
  if (x % determinant != 0 || y % determinant != 0) return {-1, -1};  // No
  ↪  value solution
  x /= determinant;
  y /= determinant;
  return {(int)x, (int)y};
}
```

solveQuadratic

```
#define double long double  // ************
const double EPS = 1e-9;
pair<complex<double>, complex<double>> solveQuadratic(double a, double b,
↪  double c) {
  complex<double> discriminant = b * b - 4.0 * a * c;
  complex<double> sqrt_discriminant = sqrt(discriminant);
  complex<double> root1 = (-b + sqrt_discriminant) / (2.0 * a);
  complex<double> root2 = (-b - sqrt_discriminant) / (2.0 * a);
  return {root1, root2};
}
```

General

01.Count Number of MOD

```
int count_number_of_mod(int l, int r, int x, int mod) {
  // count number of y such that y%n = x , for y from l to r
  if (x >= mod) return 0;
  return (r - x + mod) / mod - (l - 1 - x + mod) / mod;
}
```

02.Calc Ones n Range

```
ll calc_ones(ll a, ll bit) {
  ++bit;
  ll ones = a / (1ll << bit) * (1ll << (bit - 1));
  if (a % (1ll << bit) >= (1ll << (bit - 1)))
    ones += a % (1ll << bit) - (1ll << (bit - 1)) + 1;
  return ones;
}
// calc how many bit number (bit) appear in range (1, a)
// if you want range [l, r]  = calc_ones(r, bit_number) - calc(l - 1,
// bit_number) now you can get xor , or  , and easily  you have the number
↪  of
// this bit appear in or it's enough for this bit to appear at least once to
// consider it in your answer in and it should appear (r - l + 1) , in xor
// should appear odd times
```

03.Custom Hash

```
struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM = rng();
    return splitmix64(x + FIXED_RANDOM);
  }
};
template <typename T>
using safe_set = unordered_set<T, custom_hash>;
template <typename T, typename V>
using safe_map = unordered_map<T, V, custom_hash>;
struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
  size_t operator()(uint64_t x) const { // single integers
    static const uint64_t FIXED_RANDOM = rng();
    return splitmix64(x + FIXED_RANDOM);
  }
  template <size_t N>
  size_t operator()(const std::array<int, N> &arr) const {
    static const uint64_t FIXED_RANDOM = rng();
    uint64_t hash = FIXED_RANDOM;
    for (int x : arr) {
      hash ^=
          splitmix64(static_cast<uint64_t>(x) + 0x9e3779b9 + (hash << 6) +
          ↪  (hash >> 2));
    }
    return hash;
  }
};
```

04.Fib

```
pair<int, int> fib(int n) {
  if (n == 0) return {0, 1};
  auto p = fib(n >> 1);
  int c = p.first * ((2 * p.second % mod - p.first + 4 * mod) % mod) % mod;
  int d = (p.first * p.first % mod + p.second * p.second % mod) % mod;
  if (n & 1) return {d, (c + d) % mod};
  return {c, d};
}
```

05.Rnd

```
mt19937 rng(chrono::system_clock::now().time_since_epoch().count());
int rnd(int a, int b) {
  if (a > b) return 0;
  return a + rng() % (b - a + 1);
}
```

06.Nested Range Count

```
// Given n ranges, your task is to count for each range how many other
↪  ranges it contains and how many other ranges contain it.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <class T> using ordered_set =
    tree<T, null_type, less_equal<T>, rb_tree_tag,
    ↪  tree_order_statistics_node_update>;
// order_of_key(k): Gives the count of elements smaller than k. - O(log n)
// find_by_order(k): Returns the iterator for the kth element (use k = 0 for
↪  the first element). - O(log n)O
struct Range {
  int l, r, idx;
  bool operator<(Range &other) {
    if (other.l == l) return r > other.r;
    return l < other.l;
  }
};
void solve() {
  int n; cin >> n;
  vector<Range> ranges(n);
  for (int i = 0; i < n; ++i) {
    cin >> ranges[i].l >> ranges[i].r;
    ranges[i].idx = i;
  }
  sort(ranges.begin(), ranges.end());
  vector<int> contains(n), contained(n);
  ordered_set<int> maxR, minR;
  for (int i = 0; i < n; ++i) {
    contains[ranges[i].idx] = maxR.size() - maxR.order_of_key(ranges[i].r);
    maxR.insert(ranges[i].r);
  }
  for (int i = n - 1; ~i; --i) {
    contained[ranges[i].idx] = minR.order_of_key(ranges[i].r + 1);
    minR.insert(ranges[i].r);
  }
  for (auto i : contained) cout << i << ' '; cout << '\n';
  for (auto i : contains) cout << i << ' ';
}
```

07.Matrix Rotation

```
vector<vector<int>> rotateClockWise(vector<vector<int>> &g) {
  int n = g.size(), m = g[0].size();
  vector res(m, vector<int>(n));
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j) res[i][j] = g[n - j - 1][i];
  return res;
}
```

```
vector<vector<int>> rotateCounterClockwise(vector<vector<int>> &g) {
  if (g.empty() || g[0].empty()) return {};  // Handle empty matrix
  int n = g.size(), m = g[0].size();
  vector<vector<int>> res(m, vector<int>(n));
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j) res[i][j] = g[j][m - i - 1];
  return res;
}
```

08.Knight Move

```
int dx[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
int dy[8] = {-1, -2, -2, -1, 1, 2, 2, 1};
```

09.Pragma

```
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```

10.Negative Indexed Array

```
#define arrRange(t, a, mn, mx) t a##_[mx - mn + 1], *a = (a##_) - mn;
```

11.Submask of mask

```
for (int b = (b - mask) & mask; b; b = (b - mask) & mask)
  res = max(res, go(mask ^ b) + cost[b]);
```

12. builtin functions

```
// count the number of one's
__builtin_popcount() // __builtin_popcountll()
// returns 1 if the number has odd one's parity
__builtin_parity()
// count the leading zeros
__builtin_clz()
// count the trailing zeros
__builtin_ctz()
```

13.ternarySearch

```
const double EPS = 1e-7;
// f(x) = -(x - 3)^2 + 10 → maximum at x = 3
double f(double x) { return -(x - 3) * (x - 3) + 10; }
```

```
double ternarySearchMax(double left, double right, double eps = 1e-6) {
    while (right - left > eps) {
        double mid1 = left + (right - left) / 3;
        double mid2 = right - (right - left) / 3;
        if (f(mid1) < f(mid2)) // reverse for minimum
            left = mid1;
        else right = mid2;
    }
    return (left + right) / 2; // Approximate maximum point
}
double f(double x, double y) {
    return (x - 2) * (x - 2) + (y + 3) * (y + 3); // Minimum at x=2, y=-3
}
double ternarySearchY(double x, double y_left, double y_right) {
    while (y_right - y_left > EPS) {
        double y1 = y_left + (y_right - y_left) / 3.0;
        double y2 = y_right - (y_right - y_left) / 3.0;
        if (f(x, y1) < f(x, y2)) // reverse for maximum
            y_right = y2;
        else y_left = y1;
    }
    return (y_left + y_right) / 2.0;
}

pair<pair<double, double>, double> ternarySearchX(double x_left, double
↪   x_right, double y_left, double y_right) {
    while (x_right - x_left > EPS) {
        double x1 = x_left + (x_right - x_left) / 3.0;
        double x2 = x_right - (x_right - x_left) / 3.0;
        double y1 = ternarySearchY(x1, y_left, y_right);
        double y2 = ternarySearchY(x2, y_left, y_right);
        if (f(x1, y1) < f(x2, y2)) // reverse for maximum
            x_right = x2;
        else x_left = x1;
    }
    double x_opt = (x_left + x_right) / 2.0;
    double y_opt = ternarySearchY(x_opt, y_left, y_right);
    double val = f(x_opt, y_opt);
    return {{x_opt, y_opt}, val};
}
```

Some Math Equations

1. Composite numbers

Highly Composite Numbers		
Digits	**Number**	**Divisors**
1	6	4
2	60	12
3	840	32
4	7560	64
5	83160	128
6	720720	240
7	8648640	448
8	73513440	768
9	735134400	1344
10	6983776800	2304
11	97772875200	4032
12	963761198400	6720
13	9316358251200	10752
14	97821761637600	17280
15	866421317361600	26880
16	8086598962041600	41472
17	74801040398884800	64512
18	897612484786617600	103680

Binomial Identities

1. **Symmetry Rule**

$$\binom{n}{k} = \binom{n}{n-k}$$

2. **Factoring In**

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

3. **Sum Over k**

$$\sum_{k=0}^{n}\binom{n}{k} = 2^n$$

4. **Sum Over n**

$$\sum_{m=0}^{n}\binom{m}{k} = \binom{n+1}{k+1}$$

5. **Sum Over n and k**

$$\sum_{k=0}^{m}\binom{n+k}{k} = \binom{n+m+1}{m}$$

6. **Sum of the Squares**

$$\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$$

7. **Weighted Sum**

$$1\cdot\binom{n}{1} + 2\cdot\binom{n}{2} + \cdots + n\cdot\binom{n}{n} = n\cdot 2^{n-1}$$

8. **Connection with Fibonacci Numbers**

$$\binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k} + \cdots + \binom{0}{n} = F_{n+1}$$

Combinatorics

Binomial Coefficient Identities

1. **Fibonacci Binomial Identity**

$$\sum_{0\le k\le n}\binom{n-k}{k} = \mathrm{Fib}_{n+1}$$

2. **Symmetry Rule**

$$\binom{n}{k} = \binom{n}{n-k}$$

3. **Pascal's Recurrence**

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}$$

4. **Absorption Identity**

$$k\binom{n}{k} = n\binom{n-1}{k-1}$$

5. **Factoring In**

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

6. **Sum of Binomial Coefficients**

$$\sum_{i=0}^{n}\binom{n}{i} = 2^n$$

7. **Sum of Even Binomial Coefficients**

$$\sum_{i\geq 0}\binom{n}{2i} = 2^{n-1}$$

8. **Sum of Odd Binomial Coefficients**

$$\sum_{i\geq 0}\binom{n}{2i+1} = 2^{n-1}$$

9. **Alternating Sum Identity**

$$\sum_{i=0}^{k}(-1)^i\binom{n}{i} = (-1)^k\binom{n-1}{k}$$

10. **Hockey-Stick Identity**

$$\sum_{i=0}^{k}\binom{n+i}{i} = \sum_{i=0}^{k}\binom{n+i}{n} = \binom{n+k+1}{k}$$

11. **Weighted Linear Sum**

$$\sum_{i=1}^{n} i\binom{n}{i} = n2^{n-1}$$

12. **Weighted Quadratic Sum**

$$\sum_{i=1}^{n} i^2\binom{n}{i} = (n+n^2)2^{n-2}$$

13. **Vandermonde Convolution**

$$\sum_{k=0}^{r}\binom{m}{k}\binom{n}{r-k} = \binom{m+n}{r}$$

14. **Upward Hockey-Stick Identity**

$$\sum_{i=r}^{n}\binom{i}{r} = \binom{n+1}{r+1}$$

15. **Sum of Squared Binomial Coefficients**

$$\sum_{i=0}^{k}\binom{k}{i}^2 = \binom{2k}{k}$$

16. **Chu-Vandermonde Special Case**

$$\sum_{i=1}^{n}\binom{n}{i}\binom{n-1}{i-1} = \binom{2n-1}{n-1}$$

17. **Double Subset Selection**

$$\sum_{k=q}^{n}\binom{n}{k}\binom{k}{q} = 2^{n-q}\binom{n}{q}$$

18. **Generalized Binomial Theorem**

$$\sum_{i=0}^{n} k^i\binom{n}{i} = (k+1)^n$$

19. **Half Binomial Sum**

$$\sum_{i=0}^{n}\binom{2n}{i} = 2^{2n-1} + \frac{1}{2}\binom{2n}{n}$$

20. **Squared Central Binomial Sum**

$$\sum_{i=0}^{n}\binom{2n}{i}^2 = \frac{1}{2}\left(\binom{4n}{2n} + \binom{2n}{n}^2\right)$$

Common Summation Formulas

1. **Sum of First n Integers**

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

2. **Sum of Squares**

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

3. **Sum of Cubes**

$$\sum_{k=1}^{n} k^3 = \left[\frac{n(n+1)}{2}\right]^2$$

4. **Sum of Fourth Powers**

$$\sum_{k=1}^{n} k^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

5. **Sum of Fifth Powers**

$$\sum_{k=1}^{n} k^5 = \frac{n(n+1)(2n+1)(3n^2+3n-1)(4n^3+6n^2-1)}{30}$$

6. **Sum of Odd Numbers**

- Sum of odd numbers $\leq n$:

$$\sum_{\substack{k=1 \\ k \text{ odd}}}^{n} k = \left\lfloor \frac{n+1}{2} \right\rfloor^2$$

7. **Sum of Even Numbers**

- Sum of even numbers $\leq n$:

$$\sum_{\substack{k=1 \\ k \text{ even}}}^{n} k = \left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right)$$

8. **Finite Geometric Series**

$$\sum_{k=0}^{n} a^k = \frac{1-a^{n+1}}{1-a} \quad (a \neq 1)$$

9. **Weighted Geometric Series (Linear)**

$$\sum_{k=0}^{n} k a^k = \frac{a[1-(n+1)a^n + na^{n+1}]}{(1-a)^2}$$

10. **Weighted Geometric Series (Quadratic)**

$$\sum_{k=0}^{n} k^2 a^k$$

$$= \frac{a[(1+a)-(n+1)^2 a^n + (2n^2+2n-1)a^{n+1} - n^2 a^{n+2}]}{(1-a)^3}$$

11. **Binomial Theorem**

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

12. **Alternating Binomial Sum**

$$\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$$

13. **Weighted Binomial Sum**

$$\sum_{k=0}^{n} k \binom{n}{k} = n \cdot 2^{n-1}$$

14. **Harmonic Series**

$$\sum_{k=1}^{n} \frac{1}{k} = H_n \approx \ln n + \gamma$$

15. **Sum of Reciprocal Squares**

$$\sum_{k=1}^{n} \frac{1}{k^2} = \frac{\pi^2}{6}$$

16. **Fibonacci Series Sum**

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1$$

17. **Telescoping Series**

$$\sum_{k=1}^{n} \left(\frac{1}{k} - \frac{1}{k+1} \right) = 1 - \frac{1}{n+1}$$

18. **Sine Taylor Series**

$$\sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} = \sin x$$

19. **Cosine Taylor Series**

$$\sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} = \cos x$$

20. **Exponential Taylor Series**

$$\sum_{k=0}^{\infty} \frac{x^k}{k!} = e^x$$

21. **Geometric Series (Base 2)**

$$\sum_{k=1}^{n} 2^k = 2^{n+1} - 2$$

22. **Finite Geometric Series (General)**

$$\sum_{k=1}^{n} x^k = \frac{x(x^n - 1)}{x - 1}, \quad x \neq 1$$

$$\sum_{k=1}^{n} x^k = \frac{1 - x^{n+1}}{1 - x}, \quad x \neq 1$$

23. **Weighted Geometric Series (Base 2)**

$$\sum_{k=1}^{n} 2^k \cdot k = 2^{n+1} \cdot n - 2$$

24. **Exponential Series with Constant Exponent**

$$\sum_{k=1}^{n} 2^{km} = \frac{2^m(2^{mn} - 1)}{2^m - 1}, \quad m \text{ constant}$$

Euler Totient Function Properties

1. **Multiplicative Property**

   - If $\gcd(m, n) = 1$, then $\phi(m \cdot n) = \phi(m) \cdot \phi(n)$

2. **Closed Form Formula**

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

3. **Prime Power Case**

   - For prime p and $k \geq 1$:

$$\phi(p^k) = p^{k-1}(p - 1) = p^k \left(1 - \frac{1}{p}\right)$$

4. **Jordan Totient Generalization**

$$J_k(n) = n^k \prod_{p|n} \left(1 - \frac{1}{p^k}\right)$$

   - $J_1(n) = \phi(n)$ counts $(k+1)$-tuples coprime with n

5. **Sum of Jordan Totients**

$$\sum_{d|n} J_k(d) = n^k$$

6. **Divisor Sum**

$$\sum_{d|n} \phi(d) = n$$

7. **Möbius Inversion Formulas**

$$\phi(n) = \sum_{d|n} \mu(d) \frac{n}{d} = n \sum_{d|n} \frac{\mu(d)}{d}$$

$$\phi(n) = \sum_{d|n} d \cdot \mu\left(\frac{n}{d}\right)$$

8. **Divisibility Property**

$$a \mid b \Rightarrow \phi(a) \mid \phi(b)$$

9. **Exponent Divisibility**

$$n \mid \phi(a^n - 1) \quad \text{for } a, n > 1$$

10. **General Product Formula**

$$\phi(mn) = \phi(m)\phi(n)\frac{d}{\phi(d)} \quad \text{where } d = \gcd(m, n)$$

11. **Special Cases**

   - For even numbers:

$$\phi(2m) = \begin{cases} 2\phi(m) & \text{if } m \text{ is even} \\ \phi(m) & \text{if } m \text{ is odd} \end{cases}$$

- Power formula:

$$\phi(n^m) = n^{m-1}\phi(n)$$

12. **LCM-GCD Relationship**

$$\phi(\text{lcm}(m,n)) \cdot \phi(\text{gcd}(m,n)) = \phi(m) \cdot \phi(n)$$

13. **Parity Property**

- $\phi(n)$ is even for $n \geq 3$
- If n has r distinct odd prime factors, then $2^r \mid \phi(n)$

14. **Reciprocal Sum**

$$\sum_{d|n} \frac{\mu^2(d)}{\phi(d)} = \frac{n}{\phi(n)}$$

15. **Sum of Coprimes**

$$\sum_{\substack{1\leq k\leq n \\ \text{gcd}(k,n)=1}} k = \frac{1}{2}n\phi(n) \quad \text{for } n > 1$$

16. **Radical Property**

$$\frac{\phi(n)}{n} = \frac{\phi(\text{rad}(n))}{\text{rad}(n)}$$

- where $\text{rad}(n) = \prod_{p|n} p$

17. **Bounds**

- Lower bound: $\phi(m) \geq \log_2 m$
- Iterated totient bound: $\phi(\phi(m)) \leq \frac{m}{2}$

18. **Exponent Reduction**

- For $x \geq \log_2 m$:

$$n^x \mod m = n^{\phi(m)+x \mod \phi(m)} \mod m$$

19. **GCD Sum**

$$\sum_{\substack{1\leq k\leq n \\ \text{gcd}(k,n)=1}} \text{gcd}(k-1,n) = \phi(n)d(n)$$

- Also holds for $\text{gcd}(a \cdot k - 1, n)$ when $\text{gcd}(a,n) = 1$

20. **Non-uniqueness**

- For every n there exists $m \neq n$ with $\phi(m) = \phi(n)$

21. **Weighted Sum**

$$\sum_{i=1}^{n} \phi(i) \left\lfloor \frac{n}{i} \right\rfloor = \frac{n(n+1)}{2}$$

22. **Odd-indexed Sum**

$$\sum_{\substack{i=1 \\ i \text{ odd}}}^{n} \phi(i) \left\lfloor \frac{n}{i} \right\rfloor = \sum_{k\geq 1} \left[\frac{n}{2^k}\right]^2$$

- where $[\cdot]$ denotes rounding

23. **Double Sum**

$$\sum_{i=1}^{n}\sum_{j=1}^{n} ij[\text{gcd}(i,j)=1] = \sum_{i=1}^{n} \phi(i)i^2$$

24. **Average Value**

- The average of coprimes of n less than n is $\dfrac{n}{2}$

Fibonacci

1. **Definition**

- $F_0 = 0,\ F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

2. **Combinatorial Formula**

$$F_n = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-k-1}{k}$$

3. **Binet's Formula**

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

4. **Sum of First n Fibonacci Numbers**

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1$$

5. **Sum of Odd-indexed Fibonacci Numbers**

$$\sum_{i=0}^{n-1} F_{2i+1} = F_{2n}$$

6. **Sum of Even-indexed Fibonacci Numbers**

$$\sum_{i=1}^{n} F_{2i} = F_{2n+1} - 1$$

7. **Sum of Squares**

$$\sum_{i=1}^{n} F_i^2 = F_n F_{n+1}$$

8. **Cassini's Identity**

$$F_m F_{n+1} - F_{m-1} F_n = (-1)^n F_{m-n}$$

$$F_{2n} = F_{n+1}^2 - F_{n-1}^2 = F_n (F_{n+1} + F_{n-1})$$

9. **Addition Formulas**

$$F_m F_n + F_{m-1} F_{n-1} = F_{m+n-1}$$

$$F_m F_{n+1} + F_{m-1} F_n = F_{m+n}$$

10. **Fibonacci Test**

- A natural number n is Fibonacci iff $5n^2 + 4$ or $5n^2 - 4$ is a perfect square

11. **Divisibility Property**

- Every k-th Fibonacci number is a multiple of F_k

12. **GCD Property**

$$\text{gcd}(F_m, F_n) = F_{\text{gcd}(m,n)}$$

13. **Coprimality**

- Any three consecutive Fibonacci numbers are pairwise coprime

14. **Periodicity Modulo n**

- Fibonacci sequence modulo n is periodic with Pisano period $\leq 6n$

GCD and LCM Properties

1. **Basic GCD Properties**

$$\text{gcd}(a, 0) = a$$

$$\text{gcd}(a, b) = \text{gcd}(b, a \mod b)$$

- Every common divisor of a and b divides $\text{gcd}(a, b)$

2. **Linear Combination Property**

$$\text{gcd}(a + m \cdot b, b) = \text{gcd}(a, b) \quad \text{for any integer } m$$

3. **Multiplicative Property**

- If $\text{gcd}(a_1, a_2) = 1$, then $\text{gcd}(a_1 a_2, b) = \text{gcd}(a_1, b) \cdot \text{gcd}(a_2, b)$

4. **GCD-LCM Product Identity**

$$\text{gcd}(a, b) \cdot \text{lcm}(a, b) = |a \cdot b|$$

5. **GCD-LCM Distributive Laws**

$$\text{gcd}(a, \text{lcm}(b, c)) = \text{lcm}(\text{gcd}(a, b), \text{gcd}(a, c))$$

$$\text{lcm}(a, \text{gcd}(b, c)) = \text{gcd}(\text{lcm}(a, b), \text{lcm}(a, c))$$

6. **Exponent GCD Property**

$$\text{gcd}(n^a - 1, n^b - 1) = n^{\text{gcd}(a,b)} - 1$$

7. **Totient Sum Representation**

$$\text{gcd}(a, b) = \sum_{\substack{k|a \\ k|b}} \phi(k)$$

8. **Counting GCD Values**

$$\sum_{i=1}^{n} [\text{gcd}(i, n) = k] = \phi\left(\frac{n}{k}\right)$$

9. **Sum of GCDs**

$$\sum_{k=1}^{n} \text{gcd}(k, n) = \sum_{d|n} d \cdot \phi\left(\frac{n}{d}\right)$$

10. **Exponential GCD Sum**

$$\sum_{k=1}^{n} x^{\text{gcd}(k,n)} = \sum_{d|n} x^d \cdot \phi\left(\frac{n}{d}\right)$$

11. **Reciprocal GCD Sum**

$$\sum_{k=1}^{n} \frac{1}{\text{gcd}(k, n)} = \frac{1}{n} \sum_{d|n} d \cdot \phi(d)$$

12. **Weighted Reciprocal GCD Sum**

$$\sum_{k=1}^{n} \frac{k}{\text{gcd}(k,n)} = \frac{1}{2}\sum_{d|n} \phi(d)$$

13. **Modified GCD Sum**

$$\sum_{k=1}^{n} \frac{n}{\text{gcd}(k,n)} = 2\sum_{k=1}^{n} \frac{k}{\text{gcd}(k,n)} - 1 \quad (n > 1)$$

14. **Coprime Pairs Count**

$$\sum_{i=1}^{n}\sum_{j=1}^{n}[\text{gcd}(i,j) = 1] = \sum_{d=1}^{n} \mu(d) \left\lfloor \frac{n}{d} \right\rfloor^2$$

15. **Sum of GCD Pairs**

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\text{gcd}(i,j) = \sum_{d=1}^{n} \phi(d) \left\lfloor \frac{n}{d} \right\rfloor^2$$

16. **Coprime Pairs Product Sum**

$$\sum_{i=1}^{n}\sum_{j=1}^{n} i \cdot j[\text{gcd}(i,j) = 1] = \sum_{i=1}^{n} \phi(i)i^2$$

17. **LCM Pairs Sum**

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\text{lcm}(i,j) = \sum_{l=1}^{n} \left(\frac{(1 + \lfloor n/l \rfloor) \cdot \lfloor n/l \rfloor}{2} \right)^2 \sum_{d|l} \mu(d)ld$$

18. **Multiple GCD-LCM Relationship**

$$\text{gcd}(\text{lcm}(a,b), \text{lcm}(b,c), \text{lcm}(a,c))$$
$$= \text{lcm}(\text{gcd}(a,b), \text{gcd}(b,c), \text{gcd}(a,c))$$

19. **Array GCD Property**

$$\text{gcd}(A_L, A_{L+1}, \ldots, A_R) = \text{gcd}(A_L, A_{L+1} - A_L, \ldots, A_R - A_{R-1})$$

20. **LCM Sum Formula**

- $\text{SUM} = \sum_{k=1}^{n} \text{lcm}(k,n)$

$$\text{SUM} = \frac{n}{2}\left(\sum_{d|n} \phi(d) \cdot d + 1 \right)$$

Geometric Series

1. **Standard Geometric Series**

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r}, \quad |r| < 1$$

2. **Geometric Series with First Term a**

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}, \quad |r| < 1$$

3. **Alternating Geometric Series**

$$\sum_{k=0}^{\infty} (-1)^k r^k = \frac{1}{1+r}, \quad |r| < 1$$

4. **Series Starting at $k = 1$**

$$\sum_{k=1}^{\infty} r^k = \frac{r}{1-r}, \quad |r| < 1$$

5. **Weighted Geometric Series (Linear)**

$$\sum_{k=1}^{\infty} kr^k = \frac{r}{(1-r)^2}, \quad |r| < 1$$

6. **Weighted Geometric Series (Quadratic)**

$$\sum_{k=1}^{\infty} k^2 r^k = \frac{r(1+r)}{(1-r)^3}, \quad |r| < 1$$

7. **Weighted Geometric Series (Cubic)**

$$\sum_{k=1}^{\infty} k^3 r^k = \frac{r(1+4r+r^2)}{(1-r)^4}, \quad |r| < 1$$