UFSC Livro do time (2024-2025)

Sumário

1 Data Structures

1.1 Centroid Decomposition

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>

using namespace std;
using pii = pair<int, int>;

// 'closest_red', query and update were used for solving xenia and the tree.
struct CentroidDecomposition {
  vector<vector<int>> tree;
  vector<int> subtrees_sz, closest_red;
  vector<vector<pii>> parents;
  vector<bool> removed;

  CentroidDecomposition(vector<vector<int>> adj) : tree{adj} {
    int n = tree.size();

    subtrees_sz.resize(n);
    removed.assign(n, false);
    closest_red.assign(n, 1e9);
    parents.resize(n);

    centroid_decomposition(0, -1);
  }

  void calculate_subtree_sizes(int u, int p = -1) {
    subtrees_sz[u] = 1;
    for (auto v : tree[u]) {
      if (v == p || removed[v])
        continue;
      calculate_subtree_sizes(v, u);
      subtrees_sz[u] += subtrees_sz[v];
    }
  }

  int find_centroid(int u, int p, int n) {
    for (auto v : tree[u]) {
      if (v == p || removed[v])
        continue;
      if (subtrees_sz[v] > n / 2)
        return find_centroid(v, u, n);
    }

    return u;
  }

  void calculate_distance_to_centroid(int u, int p, int centroid, int d) {
    for (auto v : tree[u]) {
      if (v == p || removed[v])
        continue;
      calculate_distance_to_centroid(v, u, centroid, d + 1);
    }
    parents[u].push_back({centroid, d});
  }

  void centroid_decomposition(int u, int p = -1) {
    calculate_subtree_sizes(u);
    int centroid = find_centroid(u, p, subtrees_sz[u]);

    for (auto v : tree[centroid]) {
      if (removed[v])
        continue;
      calculate_distance_to_centroid(v, centroid, centroid, 1);
    }

    removed[centroid] = true;

    for (auto v : tree[centroid]) {
      if (removed[v])
        continue;
      centroid_decomposition(v, u);
    }
  }

  int query(int u) {
    int ret = closest_red[u];
    for (auto &[p, pd] : parents[u])
      ret = min(ret, pd + closest_red[p]);

    return ret;
  }

  void update(int u) {
    closest_red[u] = 0;
    for (auto &[p, pdist] : parents[u])
      closest_red[p] = min(closest_red[p], pdist);
    }
};
```

1.2 Centroid Decomposition 2

```
class CentroidDecomposition {
    vector<vector<int>> adj;
    vector<int> subtreeSz, removed, cnt;
    int maxDepth;

    // Sets sizes of subtrees of the tree rooted at 'u' bounded by removed vertices.
    void setSubtreeSizes(int u, int p = -1) {
        subtreeSz[u] = 1;
        for (int v : adj[u]) {
            if (v == p || removed[v]) continue;
            setSubtreeSizes(v, u);
            subtreeSz[u] += subtreeSz[v];
        }
    }

    // Returns the centroid of the tree rooted at 'root' bounded by removed vertices.
    int getCentroid(int u, int p, int root) {
        for (int v : adj[u]) {
            if (v == p || removed[v]) continue;
            if (subtreeSz[v] * 2 > subtreeSz[root])
                return getCentroid(v, u, root);
        }
        return u;
    }

    // Recursively counts paths of length exactly k (and sets up cnt for other subtrees of the centroid).
    void count(int u, int p, bool querying, int depth=1) {
        if (depth > k) return;
        if (querying) totalPathsK += cnt[k-depth];
        else cnt[depth]++;
        maxDepth = max(maxDepth, depth);
        for (int v : adj[u]) {
            if (v == p || removed[v]) continue;
            count(v, u, querying, depth+1);
        }
    }

    // This is where you iterate over the subtree with centroid 'centroid', and this is the place
    // where you edit your code and make it do whatever you need it to in O(sz) time.
    void build(int u) {
        setSubtreeSizes(u);
        int centroid = getCentroid(u, u, u);
        removed[centroid] = 1;

        maxDepth = 0;
        for (int v : adj[centroid]) {
            if (removed[v]) continue;
            count(v, centroid, true);
            count(v, centroid, false);
        }
        fill(cnt.begin()+1, cnt.begin()+maxDepth+1, 0);

        for (int v : adj[centroid])
            if (!removed[v])
                build(v);
    }

public:
    int k;
    ll totalPathsK;
    CentroidDecomposition(vector<vector<int>> &adj, int k, int root=0) : adj(adj), k(k) {
        int n = adj.size();
        totalPathsK = 0;
        subtreeSz.resize(n);
        cnt.assign(n, 0);
        cnt[0] = 1;
        removed.assign(n, 0);
        build(root);
    }
};

/*
Remember that, for any path u->v, there is one (and only one) common ancestroid on the path: namely,
the LCA(u, v) on the centroid tree. That is why, for distance stuff, you simply update all of the
ancestroids: that works because you ensure the LCA will be updated.

void setDist(int u, int p, int ancestroid, int dist=1) {
    ancestroids[u].push_back({dist, ancestroid});
    for (int v : adj[u]) {
        if (v == p || removed[v]) continue;
        setDist(v, u, ancestroid, dist+1);
    }
}
```

```
void paint(int u) {
    closestRed[u] = {0, u};
    for (auto[dist, ancestroid] : ancestroids[u])
        closestRed[ancestroid] = min(closestRed[ancestroid], {dist, u});
}

int query(int u) {
    ii ans = closestRed[u];
    for (auto[dist, ancestroid] : ancestroids[u])
        ans = min(ans, {dist+closestRed[ancestroid].first, closestRed[ancestroid].second});
    return ans.second;
}

And within build():
for (int v : adj[centroid])
    if (!removed[v])
        setDist(v, centroid, centroid);
*/
```

1.3 Fenwick Tree

```
#include <bits/stdc++.h>

using namespace std;

// 1-indexed FenwickTree
struct FenwickTree {
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  FenwickTree(vector<int> &vec) {
    ft.assign(vec.size() + 1, 0);
    for (int i = 0; i < vec.size(); ++i)
      update(i + 1, vec[i]);
  }

  inline int ls_one(int x) { return x & (-x); }

  int query(int r) {
    int sum = 0;
    while (r) {
      sum += ft[r];
      r -= ls_one(r);
    }
    return sum;
  }

  int query(int l, int r) { return query(r) - query(l - 1); }

  void update(int i, int v) {
    while (i < ft.size()) {
      ft[i] += v;
      i += ls_one(i);
    }
  }

  // Finds smallest index i on FenwickTree such that query(1, i) >= rank.
  // I.e: smallest i for [1, i] >= k
  int select(long long k) { // O(log^2 m)
    int lo = 1, hi = ft.size() - 1;
    for (int i = 0; i < 30; ++i) {
      int mid = (lo + hi) / 2;
      (query(1, mid) < k) ? lo = mid : hi = mid;
    }
    return hi;
  }

  vector<int> ft;
};

// Range Update - Point query
struct RUPQ {
  FenwickTree ft;
  RUPQ(int m) : ft{m} {}

  void range_update(int ui, int uj, int v) {
    ft.update(ui, v);
    ft.update(uj + 1, -v);
  }

  int point_query(int i) { return ft.query(i); }
};

// Range Update - Range Query
struct RURQ {
  RUPQ rupq;
  FenwickTree ft;

  RURQ(int m) : ft{m}, rupq(m) {}

  void range_update(int ui, int uj, int v) {
    rupq.range_update(ui, uj, v);
```

```
        ft.update(ui, v * (ui - 1));
        ft.update(uj + 1, -v * uj);
    }

    int query(int j) { return rupq.point_query(j) * j - ft.query(j); }

    int query(int i, int j) { return query(j) - query(i - 1); }
};
```

1.4 Heavy Light Decomposition

```
// Bring your favorite LazySegmentTree implementation here.
// The one used here should allow 'setting' and 'unsetting' a range.

class HLD {
    vector<int> sz, h, pos, p, at;
    LazySegmentTree st;
    int timer = 0;

    void dfsSz(int v, int parent, vector<vector<int>> &adj) {
        sz[v] = 1;
        p[v] = parent;
        for (int &u : adj[v]) { // pass by &reference is important!!
            if (u == parent) continue;
            dfsSz(u, v, adj);
            sz[v] += sz[u];
            // The first neighbour in the adj will be the heavy path
            if (sz[u] > sz[adj[v][0]] || adj[v][0] == parent)
                swap(u, adj[v][0]);
        }
    }

    void dfsHLD(int v, int parent, vector<vector<int>> &adj) {
        pos[v] = timer;
        at[timer++] = v;
        for (int u : adj[v]) {
            if (u == parent) continue;
            // if u is the heavy path
            if (u == adj[v][0]) h[u] = h[v]; // continue heavy path from above
            else h[u] = u; // starts its own heavy path
            dfsHLD(u, v, adj);
        }
    }

public:
    HLD(vector<vector<int>> &adj) {
        int n = adj.size();
        sz.resize(n);
        h.resize(n);
        pos.resize(n);
        p.resize(n);
        at.resize(n);
        dfsSz(0, -1, adj);
        dfsHLD(0, -1, adj);
        p[0] = -1; // DANGEROUS BUT NEEDED
            // gotta be careful with indexing if you wanna use a custom vector
        vector<int> emptyVector(n, 0);
        st = LazySegmentTree(emptyVector);
    }

    // Now, the subtree of v is on the range [pos[v], pos[v]+sz[v]-1] on the segment tree.
    // The path from v to the head of its heavy chain (h[v]) is [pos[h[v]], pos[v]] (because the
    // heavy path was the first to be visited on the dfs). This gives extreme querying power
    // when building a segment tree from it.

    int querySubtree(int u) {
        return st.query(pos[u], pos[u] + sz[u] - 1);
    }

    void updateSubtree(int u, int delta) {
        st.update(pos[u], pos[u] + sz[u] - 1, delta);
    }

    int queryPath(int a, int b) {
        if (pos[a] < pos[b]) swap(a, b);
            if (h[a] == h[b])
            return st.query(pos[b], pos[a]);
        return st.query(pos[h[a]], pos[a]) + queryPath(p[h[a]], b);
    }

    void updatePath(int a, int b, int delta) {
        if (pos[a] < pos[b]) swap(a, b);
            if (h[a] == h[b]) {
            st.update(pos[b], pos[a], delta);
            return;
        }
            st.update(pos[h[a]], pos[a], delta);
        updatePath(p[h[a]], b, delta);
    }
    void turnOnPathAbove(int u) {
        if (u == -1) return;
        // if the current heavy path is full, keep going. Else, binary search.
        int shouldBeOff = pos[u] - pos[h[u]] + 1;
        if (querySubtree(h[u]) + shouldBeOff >= sz[h[u]]) {
            st.update(pos[h[u]], pos[u], 1);
            turnOnPathAbove(p[h[u]]);
            return;
        }

        int low = pos[h[u]];
        int high = pos[u] + 1;

        while (low < high) {
            int mid = (low + high) / 2;
            int setOn = st.query(mid, mid+sz[at[mid]]-1);
            int shouldBeOff = pos[u] - mid + 1;
            if (setOn + shouldBeOff == sz[at[mid]])
                high = mid;
            else low = mid+1;
        }

        st.update(low, pos[u], 1);
    }

    void turnOnOperation(int u) {
        if (st.query(pos[u], pos[u])) return;
        updateSubtree(u, 1);
        turnOnPathAbove(p[u]);
    }

    void turnOffOperation(int u) {
        updateSubtree(u, -1);
        updatePath(0, u, -1); // DANGEROUS: updating u twice (I think)
    }
};
```

1.5 Li Chao Tree

```
struct Function {
    // define here your function (which may be a line). Any pair of functions must
    // intersect at most once.
    ii dish;
    int id;
    Function(ii dish, int id) : dish(dish), id(id) {}
    double eval(int x) const {
        return 1.0*(customers[x].value.first * dish.first + customers[x].value.second * dish.second) / (
            customers[x].value.first + customers[x].value.second);
    }
    bool isBetterThan(Function const &o, int x, bool isMin) const {
        // caution when you have a tiebreaker (on this case id)
        //if (fabs(this->eval(x)-o.eval(x)) < 1e-9)
        //    return this->id < o.id;
        // regular template code here
        if (isMin) return this->eval(x) < o.eval(x);
        else return this->eval(x) > o.eval(x);
    }
};

struct LiChaoTree {
    vector<Function> t;
    int n, isMin;

    Function best(Function const &a, Function const &b, int x) {
        return (a.isBetterThan(b, x, isMin)) ? a : b;
    }

    // n is the size. Allows indexing up to n-1
    LiChaoTree(int n, bool isMin) : n(n), isMin(isMin) {
        #warning Change Function init so it defaults to the opposite of your LiChaoTree
        if (isMin) t.assign(4*n, Function({-INF, -INF}, INF));
        else t.assign(4*n, Function({0, 0}, INF));
    }

    void addLine(Function nw) { addLine(nw, 1, 0, n-1); }
    void addLine(Function nw, int v, int l, int r) {
        int m = (l + r) / 2;
        bool lef = nw.isBetterThan(t[v], l, isMin);
        bool mid = nw.isBetterThan(t[v], m, isMin);
        if (mid) swap(t[v], nw);
        if (r == l) return;
        if (lef != mid) addLine(nw, 2 * v, l, m);
        else addLine(nw, 2 * v + 1, m+1, r);
    }

    Function get(int x) { return get(x, 1, 0, n-1); }
    Function get(int x, int v, int l, int r) {
        int m = (l + r) / 2;
        if (r == l) return t[v];
        if (x < m) return best(t[v], get(x, 2 * v, l, m), x);
        return best(t[v], get(x, 2 * v + 1, m+1, r), x);
    }
};
```

1.6 Li Chao Tree Sparse

```
struct Function {
    // define here your function (which may be a line). Any pair of functions must
    // intersect at most once.
    ii dish;
    int id;
    Function() {}
    Function(ii dish, int id) : dish(dish), id(id) {}
    double eval(int x) const {
        return 1.0*(customers[x].value.first * dish.first + customers[x].value.second * dish.second) / (
            customers[x].value.first + customers[x].value.second);
    }
    bool isBetterThan(Function const &o, int x, bool isMin) const {
        // caution when you have a tiebreaker (on this case id)
        //if (fabs(this->eval(x)-o.eval(x)) < 1e-9)
        //    return this->id < o.id;
        // regular template code here
        if (isMin) return this->eval(x) < o.eval(x);
        else return this->eval(x) > o.eval(x);
    }
};

struct LiChaoTree {
    struct Vertex {
        Function f;
        int l, r, isMin;
        Vertex *leftChild = nullptr;
        Vertex *rightChild = nullptr;

        Vertex(int l, int r, int isMin) : l(l), r(r), isMin(isMin) {
            #warning Change Function init so it defaults to the opposite of your LiChaoTree
            if (isMin) f = Function({-INF, -INF}, INF);
            else f = Function({0, 0}, INF);
        }

        void extend() {
            if (!leftChild && l != r) {
                int mid = (l+r)/2;
                leftChild = new Vertex(l, mid, isMin);
                rightChild = new Vertex(mid+1, r, isMin);
                leftChild->f = rightChild->f = f;
            }
        }

        void addLine(Function nw) {
            extend();
            int m = (l+r) / 2;
            bool lef = nw.isBetterThan(f, l, isMin);
            bool mid = nw.isBetterThan(f, m, isMin);
            if (mid) swap(f, nw);
            if (r == l) return;
            if (lef != mid) leftChild->addLine(nw);
            else rightChild->addLine(nw);
        }

        Function best(Function const &a, Function const &b, int x) {
            return (a.isBetterThan(b, x, isMin)) ? a : b;
        }

        Function get(int x) {
            extend();
            int m = (l + r) / 2;
            if (r == l) return f;
            if (x < m) return best(f, leftChild->get(x), x);
            return best(f, rightChild->get(x), x);
        }
    };

    int n;
    Vertex *root;
    LiChaoTree(int n, bool isMin) : n(n) {
        root = new Vertex(0, n-1, isMin);
    }
    void addLine(Function nw) { root->addLine(nw); }
    Function get(int x) { return root->get(x); }
};
```

1.7 Ordered Set

```
// C++ program to demonstrate the
// ordered set in GNU C++
#include <iostream>
using namespace std;

// Header files, namespaces,
// macros as defined above
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

#define ordered_set                                                    \
  tree<int, null_type, less<int>, rb_tree_tag,                         \
       tree_order_statistics_node_update>

// Driver program to test above functions
int main() {
  // Ordered set declared with name o_set
  ordered_set o_set;

  // insert function to insert in
  // ordered set same as SET STL
  o_set.insert(5);
  o_set.insert(1);
  o_set.insert(2);

  // Finding the second smallest element
  // in the set using * because
  //  find_by_order returns an iterator
  cout << *(o_set.find_by_order(1)) << endl;

  // Finding the number of elements
  // strictly less than k=4
  cout << o_set.order_of_key(4) << endl;

  // Finding the count of elements less
  // than or equal to 4 i.e. strictly less
  // than 5 if integers are present
  cout << o_set.order_of_key(5) << endl;

  // Deleting 2 from the set if it exists
  if (o_set.find(2) != o_set.end())
    o_set.erase(o_set.find(2));

  // Now after deleting 2 from the set
  // Finding the second smallest element in the set
  cout << *(o_set.find_by_order(1)) << endl;

  // Finding the number of
  // elements strictly less than k=4
  cout << o_set.order_of_key(4) << endl;

  return 0;
}
```

1.8 Segment Tree

```
#include <bits/stdc++.h>

#ifdef DEBUG
#define PRINT(s) std::cout << s << '\n';
#endif

using namespace std;
using ll = long long;

#include <bits/stdc++.h>

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

struct SegmentTree {

  inline int left(int p) { return p * 2; }

  inline int right(int p) { return p * 2 + 1; }

  SegmentTree(vector<int> &vec) {
    n = vec.size();
    int sz = 4 * n;

    tree.assign(sz, 1e9);
    lazy.assign(sz, -1);

    build(1, 0, n - 1, vec);
  }

  inline int merge(int a, int b) { return min(a, b); }

  void build(int p, int l, int r, vector<int> &vec) {
    if (l == r) {
      tree[p] = vec[l];
      return;
    }

    int m = (l + r) / 2;
    build(left(p), l, m, vec);
    build(right(p), m + 1, r, vec);

    tree[p] = merge(tree[left(p)], tree[right(p)]);
  }

  void propagate(int p, int l, int r) {
    if (lazy[p] == -1)
      return;

    tree[p] = lazy[p];

    if (l != r)
      lazy[left(p)] = lazy[right(p)] = lazy[p];

    lazy[p] = -1;
  }

  int query(int i, int j) { return query(1, 0, n - 1, i, j); }

  void update(int i, int j, int v) { update(1, 0, n - 1, i, j, v); }

  vector<int> lazy, tree;
  int n;

private:
  int query(int p, int l, int r, int i, int j) {
    propagate(p, l, r);
    if (i > j) // valor impossível. merge() deve ignorá-lo
      return 1e9;
    if (l >= i && r <= j)
      return tree[p];

    int m = (l + r) / 2;
    return merge(query(left(p), l, m, i, min(m, j)),
                 query(right(p), m + 1, r, max(i, m + 1), j));
  }

  void update(int p, int l, int r, int i, int j, int v) {
    propagate(p, l, r);
    if (i > j)
      return;
    if (l >= i && r <= j) {
      tree[p] = v;
      lazy[p] = v;
      return;
    }

    int m = (l + r) / 2;
    update(left(p), l, m, i, min(j, m), v);
    update(right(p), m + 1, r, max(i, m + 1), j, v);
    tree[p] = merge(tree[left(p)], tree[right(p)]);
  }
};
```

1.9 Segment Tree Lazy

```
class LazySegmentTree {
    struct Segment {
        int min, count, l, r, lazy;
    };

    ii NEUTRAL = {INF, 1};
    vector<Segment> t;

    void build(vector<int> &a, int v, int tl, int tr) {
        t[v] = {INF, 1, tl, tr, 0};
        if (tl == tr)
            t[v].min = a[tl];
        else {
            int mid = (tl + tr) / 2;
            build(a, 2*v, tl, mid);
            build(a, 2*v+1, mid+1, tr);
            merge(v);
        }
    }

    void merge(int v) {
        if (t[2*v].min < t[2*v+1].min) {
            t[v].min = t[2*v].min;
            t[v].count = t[2*v].count;
        }
        else if (t[2*v].min > t[2*v+1].min) {
            t[v].min = t[2*v+1].min;
            t[v].count = t[2*v+1].count;
        }
        else {
            t[v].min = t[2*v].min;
            t[v].count = t[2*v].count + t[2*v+1].count;
        }
    }

    ii merge(ii &a, ii &b) {
        if (a.first < b.first)
            return {a.first, a.second};
        else if (a.first > b.first)
            return {b.first, b.second};
        return {a.first, a.second + b.second};
    }

    void push(int v) {
        if (t[v].lazy) {
            t[v].min += t[v].lazy; // t[v].sum += delta * (t[v].r - t[v].l + 1)
            if (t[v].l != t[v].r) { // not leaf
                t[2*v].lazy += t[v].lazy;
                t[2*v+1].lazy += t[v].lazy;
            }
            t[v].lazy = 0;
        }
    }

    ii query(int v, int l, int r) {
        push(v);
        if (t[v].l > r || t[v].r < l)
            return NEUTRAL;
        if (t[v].l >= l && t[v].r <= r)
            return {t[v].min, t[v].count};

        ii left = query(2*v, l, r);
        ii right = query(2*v+1, l, r);
        return merge(left, right);
    }

    void update(int v, int l, int r, int delta) {
        push(v);
        if (t[v].l > r || t[v].r < l)
            return;
        if (t[v].l >= l && t[v].r <= r) {
            t[v].lazy += delta;
            push(v);
            return;
        }
        update(2*v, l, r, delta);
        update(2*v+1, l, r, delta);
        merge(v);
    }

public:
    LazySegmentTree(vector<int> &a) {
        t.resize(4*a.size());
        build(a, 1, 0, a.size()-1);
    }

    ii query(int l, int r) {
        return query(1, l, r);
    }

    void update(int l, int r, int delta) {
        update(1, l, r, delta);
    }
};
```

1.10 Segment Tree Point Update

```
struct SegmentTree {

  inline int left(int p) { return p * 2; }

  inline int right(int p) { return p * 2 + 1; }

  SegmentTree(vector<ll> &vec) {
    n = vec.size();
    int sz = 4 * n;

    tree.assign(sz, {1e18, 1e18});

    build(1, 0, n - 1, vec);
  }

  inline pll merge(const pll &a, const pll &b) {
    return {a.first + b.first, min(a.second, b.second + a.first)};
  }

  void build(int p, int l, int r, vector<ll> &vec) {
    if (l == r) {
      tree[p] = {vec[l], vec[l]};
      return;
    }

    int m = (l + r) / 2;
    build(left(p), l, m, vec);
    build(right(p), m + 1, r, vec);

    tree[p] = merge(tree[left(p)], tree[right(p)]);
  }

  pll query(int i, int j) { return query(1, 0, n - 1, i, j); }

  void update(int i, ll v) { update(1, 0, n - 1, i, v); }

  vector<pair<ll, ll>> tree;
  int n;

private:
  pll query(int p, int l, int r, int i, int j) {
    if (i > j) // valor impossível. merge() deve ignorá-lo
      return {0, 1e18};
    if (l >= i && r <= j)
      return tree[p];

    int m = (l + r) / 2;
    return merge(query(left(p), l, m, i, min(m, j)),
                 query(right(p), m + 1, r, max(i, m + 1), j));
  }

  void update(int p, int l, int r, int i, ll v) {
    if (l == r) {
      tree[p] = {v, v};
      return;
    }

    int m = (l + r) / 2;
    if (i <= m)
      update(left(p), l, m, i, v);
    else
      update(right(p), m + 1, r, i, v);
    tree[p] = merge(tree[left(p)], tree[right(p)]);
  }
};
```

1.11 Segment Tree Sparse

```
// build:
// Vertex* st = new Vertex(0, desiredLastIndex);
struct Vertex {
    int l, r;
    int lazy = 0;
    int mini = 0;
    Vertex *leftChild = nullptr;
    Vertex *rightChild = nullptr;
    const static int NEUTRAL = INF;

    Vertex(int l, int r) : l(l), r(r) {}

    void destroyTree() {
        if (leftChild) {
            leftChild->destroyTree();
            rightChild->destroyTree();
        }
        delete this;
    }

    void merge() {
        if (leftChild)
            mini = min(leftChild->mini, rightChild->mini);
    }

    int merge(int a, int b) {
        return min(a, b);
    }

    void extend() {
        if (!leftChild && l != r) {
            int mid = (l+r)/2;
            leftChild = new Vertex(l, mid);
            rightChild = new Vertex(mid+1, r);
        }
    }

    void push() {
        if (lazy) {
            mini += lazy;
            if (l != r) { // not leaf
                leftChild->lazy += lazy;
                rightChild->lazy += lazy;
            }
```

```
                lazy = 0;
            }
        }

        int queryMin(int ql, int qr) {
            extend();
            push();
            if (l > qr || r < ql)
                return NEUTRAL;
            if (l >= ql && r <= qr)
                return mini;
            return merge(leftChild->queryMin(ql, qr), rightChild->queryMin(ql, qr));
        }

        void incrementRange(int ql, int qr, int delta) {
            extend();
            push();
            if (l > qr || r < ql) return;
            if (l >= ql && r <= qr) {
                lazy += delta;
                push();
                return;
            }
            leftChild->incrementRange(ql, qr, delta);
            rightChild->incrementRange(ql, qr, delta);
            merge();
        }
    };
```

1.12 Sparse Table

```
struct SparseTable {
    vector<vector<int>> st;
    SparseTable(vector<int> &a) {
        int n = a.size();
        int k = log2(n);
        if ((1ll << k) < n) k++;
        st.assign(k+1, vector<int>(n, 0));
        copy(all(a), st[0].begin());

        for (int i = 1; i <= k; i++)
            for (int j = 0; j + (1 << i) <= n; j++)
                st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
    }

    int query(int l, int r) {
        int i = log2(r - l + 1);
        return min(st[i][l], st[i][r - (1 << i) + 1]);
    }
};
```

1.13 Suffix Array

```
#include <bits/stdc++.h>
#include <vector>

using namespace std;
using vi = vector<int>;
using ii = pair<int, int>;

class SuffixArray {

private:
  vi RA;                                 // rank array
  void countingSort(int k) {             // O(n)
    int maxi = max(300, n);              // up to 255 ASCII chars
    vi c(maxi, 0);                       // clear frequency table
    for (int i = 0; i < n; ++i)          // count the frequency
      ++c[i + k < n ? RA[i + k] : 0]; // of each integer rank
    for (int i = 0, sum = 0; i < maxi; ++i) {
      int t = c[i];
      c[i] = sum;
      sum += t;
    }
    vi tempSA(n);
    for (int i = 0; i < n; ++i) // sort SA
      tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
    swap(SA, tempSA); // update SA
  }
  void constructSA() { // can go up to 400K chars
    SA.resize(n);
    iota(SA.begin(), SA.end(), 0); // the initial SA
    RA.resize(n);
    for (int i = 0; i < n; ++i)
      RA[i] = T[i];                      // initial rankings
    for (int k = 1; k < n; k <<= 1) { // repeat log_2 n times
```

```
      // this is actually radix sort
      countingSort(k); // sort by 2nd item
      countingSort(0); // stable-sort by 1st item
      vi tempRA(n);
      int r = 0;
      tempRA[SA[0]] = r;           // re-ranking process
      for (int i = 1; i < n; ++i) // compare adj suffixes
        tempRA[SA[i]] = // same pair => same rank r; otherwise, increase r
            ((RA[SA[i]] == RA[SA[i - 1]]) &&
             (RA[SA[i] + k] == RA[SA[i - 1] + k]))
                ? r
                : ++r;
      swap(RA, tempRA); // update RA
      if (RA[SA[n - 1]] == n - 1)
        break; // nice optimization
    }
  }

public:
  const char *T; // the input string
  const int n;   // the length of T
  vi SA;         // Suffix Array
  SuffixArray(const char *initialT, const int _n) : T(initialT), n(_n) {
    constructSA(); // O(n log n)
  }
};
```

1.14 Union Find

```
// Constructor receives an integer n that is the amount of sets to be initially created. Zero indexed.6
struct UnionFind {
    vector<int> parent, size;

    UnionFind(int n) {
        parent.reserve(n);
        size.assign(n, 1);

        for (int i = 0; i < n; i++)
            parent.push_back(i);
    }

    int find(int v) {
        if (v == parent[v])
            return v;
        return parent[v] = find(parent[v]);
    }

    void unionSets(int a, int b) {
        a = find(a);
        b = find(b);
        if (a != b) {
            if (size[a] < size[b])
                swap(a, b);
            parent[b] = a; // subordinate b to a (smaller to bigger)
            size[a] += size[b];
        }
    }
};

// Allows you to undo set unions - O(logn) union, O(1) per rollback.
// Store the current state (i.e., history.size(), and, later on, do rollback() while history.size() != state).
struct RollbackUnionFind {
    vector<int> parent, size;
    int components;
    stack<pair<int*, int>> history;

    RollbackUnionFind(int n=0) {
        parent.resize(n);
        size.assign(n, 1);
        iota(all(parent), 0);
        components = n;
    }

    int find(int v) {
        if (v == parent[v])
            return v;
        return find(parent[v]);
    }

    void change(int &x, int newVal) {
        history.push({&x, x}); // x's current state
        x = newVal;
    }

    void rollback() {
        auto[ptr, val] = history.top(); history.pop();
        *ptr = val;
    }

    void unionSets(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) return;
        if (size[a] < size[b]) swap(a, b);
        change(parent[b], a);
        change(size[a], size[a]+size[b]);
        change(components, components-1);
    }
};
```

2 Dynamic Programming

2.1 Box Stacking

```
/*
Returns the tallest tower that can be made by stacking any amount of boxes (even repeating some)
as long as the base of a box above another is strictly smaller in both dimensions.
*/

struct Box {
    int h, w, d;
    bool operator< (Box& o) const {
        return d*w > o.d*o.w; // sort by decreasing area
    }

    bool fitsUnder(Box& o) const {
        return w > o.w && d > o.d;
    }
};

int maxStackHeight(vector<Box>& boxes) {
    int n = boxes.size();
    vector<Box> rot(3*n);

    int index = 0;
    for (int i = 0; i < n; i++) {
        rot[3*i] = {
            boxes[i].h,
            max(boxes[i].d, boxes[i].w),
            min(boxes[i].d, boxes[i].w)
        };

        rot[3*i+1] = {
            boxes[i].w,
            max(boxes[i].h, boxes[i].d),
            min(boxes[i].h, boxes[i].d)
        };

        rot[3*i+2] = {
            boxes[i].d,
            max(boxes[i].h, boxes[i].w),
            min(boxes[i].h, boxes[i].w)
        };
    }

    n = 3*n;

    sort(all(rot));

    vector<int> dp(n, 0); // maximum stack height with ith box on top
    for (int i = 0; i < n; i++ )
        dp[i] = rot[i].h;

    for (int i = 1; i < n; i++)
        for (int j = 0; j < i; j++)
            if (rot[j].fitsUnder(rot[i]))
                dp[i] = max(dp[i], dp[j] + rot[i].h);

    return *max_element(all(dp));
}
```

2.2 Coin Change

```
'''
Returns minimum amount of coins from the "coins" list
such that their sum is equal to "val".
Every element on the "coins" list can be used an unlimited
amount of times.
If no sum of coins is equal to "val", returns -1.
'''
def coin_change(coins, val):
    dp = [float("inf")] * (val + 1)
    dp[0] = 0

    for amount in range(1, val + 1):
        for coin in coins:
            if amount - coin >= 0:
                temp = 1 + dp[amount - coin]
                if temp < dp[amount]:
                    dp[amount] = temp
    return dp[val] if dp[val] != float("inf") else -1
```

2.3 Edit Distance

```
// memset this dude to -1
int dp[2001][2001];
string a, b;
int min(int a, int b, int c) { return min(a, min(b, c)); }
int editDistance(int i, int j) {
    if (i < 0) return j+1;
    if (j < 0) return i+1;
    if (dp[i][j] != -1) return dp[i][j];

    if (a[i] == b[j]) return dp[i][j] = editDistance(i-1, j-1);
    return dp[i][j] = 1 + min(
        editDistance(i, j-1), // insert b[j] onto a
        editDistance(i-1, j), // remove a[i]
        editDistance(i-1, j-1) // a[i] = b[j]
    );
}
```

2.4 Knapsack 1 0

```
#include <bits/stdc++.h>

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int n, x;
  cin >> n >> x;

  vector<int> price(n), pages(n);
  for (auto &p : price)
    cin >> p;
  for (auto &p : pages)
    cin >> p;

  vector<int> dp(x + 1, 0);
  for (int i = 0; i < n; ++i)
    for (int j = x; j >= price[i]; --j)
      dp[j] = max(dp[j], dp[j - price[i]] + pages[i]);
  cout << dp[x] << '\n';
}
```

2.5 Longest Common Subsequence

```
// dont forget to memset this to -1
int dp[10001][10001];
int lcs(string &a, string &b, int i, int j) {
    if (i < 0 || j < 0) return 0;
    if (dp[i][j] != -1) return dp[i][j];
    if (a[i] == b[j]) return dp[i][j] = 1 + lcs(a, b, i-1, j-1);
    return dp[i][j] = max(
        lcs(a, b, i, j-1),
        lcs(a, b, i-1, j)
    );
}

string lcsRecover() {
    int i = a.size() - 1;
    int j = b.size() - 1;
    lcs(i, j);
    string res = "";

    while (i >= 0 && j >= 0) {
        if (a[i] == b[j]) {
            res += a[i];
            i--; j--;
        }
        else if (i > 0 && dp[i][j] == dp[i-1][j]) i--;
        else j--;
    }

    reverse(res.begin(), res.end());
    return res;
}
```

2.6 Longest Increasing Subsequence

```
// O(n log n)
// vector d[l] stores the smallest element that a size "l" increasing subsequence ends with

// Alternative solution: compress the values of 'a' and then build a segment tree that queries
// and updates the lis that ends with value 'i'. Build it progressively from left to right.
// Then, the lis of any number a[i] is prefixMax(a[i]-1)+1 or simply 1.
// Useful when you have to extend the lis idea (like lis count of an array).
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[l-1] < a[i] && a[i] < d[l])
            d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++)
        if (d[l] < INF)
            ans = l;
    return ans;
}
```

2.7 Max 2D Range Sum

```
/*
 *
 * Given N x N matrix, find submatrix with greatest sum.
 * Complexity: O(n^3)
 */

int main() {
  int n;
  cin >> n;

  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
      cin >> A[i][j];
      if (j > 0)
        A[i][j] += A[i][j - 1];
    }

  int maxSubRect =
      -127 * 100 * 100; // -127 -> smallest value. 100 * 100 -> N x N
  for (int l = 0; l < n; ++l)
    for (int r = l; r < n; ++r) {
      int subRect = 0;
      for (int row = 0; row < n; ++row) {
        // Max 1D Range Sum on columns of this row
        if (l)
          subRect += A[row][r] - A[row][l - 1];
        else
          subRect += A[row][r];

        // Kadane's algorithm on rows
        if (subRect < 0)
          subRect = 0;
        maxSubRect = max(maxSubRect, subRect);
      }
    }
}
```

2.8 Travelling Salesman Problem

```
/*
To start:
memset(dp, -1, sizeof(dp));
int ans = tsp(0, 1);
Time complexity: O(2^n n^2)
*/

const int MAXN = 20;
int dp[MAXN+1][1048756]; // 2^20
int d[MAXN+1][MAXN+1];
int n; // vertex count
int tsp(int pos, int mask) {
    if (mask == (1 << n) - 1) return d[pos][0]; // everyone visited, return to root
    if (dp[pos][mask] != -1) return dp[pos][mask];
    int best = INF;
    for (int next = 0; next < n; next++)
        if ((mask & (1 << next)) == 0)
            best = min(best, d[pos][next] + tsp(next, mask | (1 << next)));
    return dp[pos][mask] = best;
}
```

3 Geometry

3.1 Circle Intersection

```
struct Circle {
  int x, y, r;
};

inline bool is_inside(Circle &a, Circle &b) {
  double d = sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));

  return d <= a.r + b.r || d <= a.r - b.r || d <= b.r - a.r;
}
```

3.2 Circuncenter

```
#include <bits/stdc++.h>

using namespace std;

struct Point {
  double x, y;
  Point(double x, double y) : x{x}, y{y} {}
};

inline double distance(const Point &p1, const Point &p2) {
  double x = p1.x - p2.x, y = p1.y - p2.y;
  return sqrt(x * x + y * y);
}

// Returns point equidistant to all vertices of the triangle
Point circuncenter(Point &A, Point &B, Point &C) {

  // LINE AB
  // ax + by = c
  double a = B.y - A.y, b = A.x - B.x;
  double c = a * A.x + b * A.y;

  // LINE BC
  double e = C.y - B.y, f = B.x - C.x;
  double g = e * B.x + f * B.y;

  // convert AB to perpendicular bisector
  c = -b * (A.x + B.x) / 2 + a * (A.y + B.y) / 2;
  b = exchange(a, -b);

  // convert BC to perpendicular bisector
  g = -f * (B.x + C.x) / 2 + e * (B.y + C.y) / 2;
  f = exchange(e, -f);

  double determinant = a * f - e * b;
  if (determinant == 0)
    return Point(1e9, 1e9);
  return Point((f * c - b * g) / determinant, (a * g - e * c) / determinant);
}
```

3.3 Convex Hull Trick

```
using ftype = long double;
using Point = complex<ftype>;
#define x real
#define y imag

// On CHT, dot will measure how aligned vectors are. dot(perpendiculars) = 0, dot(90 < theta < 270) is
// negative (because they don't even align). While dot(perpendicular of edge, new point) is negative,
// it means that there is a cw turn (so remove the previous point).
ftype dot(Point a, Point b) {
    // a.x * b.x + a.y * b.y
    return (conj(a) * b).x();
}

ftype cross(Point a, Point b) {
    // a.x * b.y - a.y * b.x, the same as orientation of geometry primitives
    // computes orientation of (0, 0) -> a -> b, the same as considering 'a'
    // and 'b' as vectors
```

```
        return (conj(a) * b).y();
    }

    struct CHT {
        const bool isMin;
        vector<Point> hull, vecs;
        CHT(vector<ii> lines, bool isMin) : isMin(isMin) {
            // having more than one 'k' (lines[i].first) is useless. So, if you want a min CHT, just
            // keep the ones with min 'b'. Else, keep the ones with max 'b'.
            if (isMin) sort(all(lines));
            else sort(rall(lines));

            lines.erase(
                unique(all(lines), [](ii a, ii b) {return a.first == b.first;}),
                lines.end()
            );

            int m = isMin ? 1 : -1;
            for (auto[k, b] : lines)
                addLine(k*m, b*m);
        }

        void addLine(ftype k, ftype b) {
            Point nw = {k, b};
            // nw - hull.back() is the vector from the last hull point to the new point
            while (!vecs.empty() && dot(vecs.back(), nw - hull.back()) < 0) {
                hull.pop_back();
                vecs.pop_back();
            }
            // Why 1il * lastVector?
            // last     = (a + bi)
            // 1il*last = (-b + ai)
            // and it is well known that this transformation rotates 90 deg ccw
            if (!hull.empty())
                vecs.push_back(1il * (nw - hull.back()));
            hull.push_back(nw);
        }

        ftype query(ftype x) {
            Point query = {x, 1};
            // find the first vec that is directed ccw from the query. Conveniently, its left
            // point is the desired line. Also conveniently, the index of that point is the same
            // as the index of the found vec.
            auto it = lower_bound(vecs.begin(), vecs.end(), query, [](Point a, Point b) {
                // dont forget this is like the operator<. This will return the first vec
                // such that NOT cross(a, query) > 0
                return cross(a, b) > 0;
            });
            ftype val = dot(query, hull[it - vecs.begin()]);
            if (isMin) return val;
            return -val;
        }
    };
```

3.4 Geometry

```
    const long double PI = 4*atanl(1.0);
    const long double EPS = 1e-9;

    template <typename T>
    struct Geometry {
        struct Point {
            T x, y;

            bool operator==(Point const &o) const {
                return x == o.x && y == o.y;
            }

            bool operator<(Point const &o) const {
                if (x == o.x) return y < o.y;
                return x < o.x;
            }

            Point operator+(Point const &o) const {
                return {x+o.x, y+o.y};
            }

            Point operator-(Point const &o) const {
                return {x-o.x, y-o.y};
            }

            Point operator*(T k) const {
                return {x*k, y*k};
            }
        };

        static long double dist(Point &a, Point &b) {
            return sqrtl((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
        }
```

```
static T dist2(Point &a, Point &b) {
    return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);
}

static int orientation(Point &a, Point &b, Point &c) {
    T o = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (o > 0) return 1; // ccw
    else if (o < 0) return -1; // cw
    return 0;
}

// Returns whether p lies over segment (a, b)
static bool liesOver(Point &p, Point &a, Point &b) {
    return (
        orientation(p, a, b) == 0
        && p.x >= min(a.x, b.x)
        && p.x <= max(a.x, b.x)
        && p.y >= min(a.y, b.y)
        && p.y <= max(a.y, b.y)
    );
}

// dot product of vectors a1->a2 and b1->b2
static T dot(Point &a1, Point &a2, Point &b1, Point &b2) {
    return (a2.x-a1.x)*(b2.x-b1.x) + (a2.y-a1.y)*(b2.y-b1.y);
}

// dot product of vectors a and b
static T dot(Point &a, Point &b) {
    return a.x * b.x + a.y * b.y;
}

// cross product of vectors a and b
static T cross(Point const &a, Point const &b) {
    return a.x * b.y - a.y * b.x;
}

static double angle(Point &a, Point &o, Point &b) { // Angle aob in rad
    return acos(dot(o, a, o, b) / sqrt(1.0 * dot(o, a, o, a) * dot(o, b, o, b)));
}

static bool perpendicular(Point &a1, Point &a2, Point &b1, Point &b2) {
    return dot(a1, a2, b1, b2) == 0;
}

struct Segment {
    Point p1, p2;
    Segment() {}
    Segment(Point p1, Point p2) : p1(p1), p2(p2) {}
    Segment(T x1, T y1, T x2, T y2) {
        p1 = {x1, y1};
        p2 = {x2, y2};
    }
};

static bool overlaps(Segment &a, Point &p) {
    // supposing this segment is collinear with p
    return (
        p.x >= min(a.p1.x, a.p2.x) && p.x <= max(a.p1.x, a.p2.x)
        && p.y >= min(a.p1.y, a.p2.y) && p.y <= max(a.p1.y, a.p2.y)
    );
}

static bool intersects(Segment &a, Segment &b) {
    int o1 = orientation(a.p1, a.p2, b.p1);
    int o2 = orientation(a.p1, a.p2, b.p2);
    int o3 = orientation(b.p1, b.p2, a.p1);
    int o4 = orientation(b.p1, b.p2, a.p2);

    if (o1 != o2 && o3 != o4)
        return true;

    if (o1 == 0 && overlaps(a, b.p1)) return true;
    if (o2 == 0 && overlaps(a, b.p2)) return true;
    if (o3 == 0 && overlaps(b, a.p1)) return true;
    if (o4 == 0 && overlaps(b, a.p2)) return true;
    return false;
}

#warning twicePolygonArea: Polygon should NOT wrap itself.
static T twicePolygonArea(vector<Point> &p) {
    int n = p.size();
    T area = 0;
    for (int i = 0; i < n-1; i++)
        area += p[i].x * p[i+1].y - p[i+1].x * p[i].y;
    area += p[n-1].x * p[0].y - p[0].x * p[n-1].y;
    return abs(area);
}

static inline double toRadians(double deg) {
    return deg * PI / 180;
}
```

```
static inline double toDegrees(double rad) {
    return rad * 180 / PI;
}

static double circleSectorArea(double r, double theta) {
    return r*r*theta/2;
}

static double triangleArea(double a, double b, double c) {
    double s = (a + b + c) / 2.0;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

// Returns intersection between segment p-q and line A-B.
static Point lineIntersectSeg(Point p, Point q, Point A, Point B) {
    double a = B.y-A.y, b = A.x-B.x, c = B.x*A.y - A.x*B.y;
    double u = fabs(a*p.x + b*p.y + c);
    double v = fabs(a*q.x + b*q.y + c);
    return {(p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v)};
}

// Returns the polygon at the left of the line formed by the line A-B.
#warning cutPolygon: Polygon should wrap (poly[0] == poly.back()).
static vector<Point> cutPolygon(Point A, Point B, vector<Point> &Q) {
    vector<Point> P;
    for (int i = 0; i < Q.size(); ++i) {
        int o1 = orientation(A, B, Q[i]);
        int o2 = 0;
        if (i != Q.size()-1)
            o2 = orientation(A, B, Q[i+1]);
        if (o1 > -EPS)
            P.push_back(Q[i]);
        if (o1*o2 < -EPS) // crosses line AB
            P.push_back(lineIntersectSeg(Q[i], Q[i+1], A, B));
    }
    if (!P.empty() && !(P.back() == P.front()))
        P.push_back(P.front());
    return P;
}

static bool ccw(Point &a, Point &b, Point &c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

static void convex_hull(vector<Point> &a, bool include_collinear = false) {
    if (!a.size()) return;
    Point p0 = *min_element(a.begin(), a.end());

    sort(a.begin(), a.end(), [&p0](Point &a, Point &b) {
        int o = orientation(p0, a, b);
        if (o == 0) return dist2(p0, a) < dist2(p0, b);
        return o > 0;
    });

    if (include_collinear) {
        int i = a.size()-1;
        while (i >= 0 && orientation(p0, a[i], a.back()) == 0) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<Point> st;
    for (int i = 0; i < a.size(); i++) {
        while (st.size() >= 2 && !ccw(st[st.size()-2], st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    if (!include_collinear && st.size() == 2 && st[0] == st[1])
        st.pop_back();

    a = st;
}

static void convex_hull_with_repetitions(vector<Point> &a, bool include_collinear = false) {
    if (!include_collinear) {
        convex_hull(a);
        return;
    }

    map<Point, int> freq;
    vector<Point> noReps;
    for (Point &p : a) {
        freq[p]++;
        if (freq[p] == 1)
            noReps.push_back(p);
    }
    convex_hull(noReps, true);
    a.clear();
    for (Point &p : noReps)
        while (freq[p]--)
            a.push_back(p);
    }

    static const int BOUNDARY = -1;
    static const int OUTSIDE = 0;
    static const int INSIDE = 1;

    static int inPolygon(Point &p, vector<Point> &poly) {
        if (poly.size() < 3) return false;
        poly.push_back(poly[0]);

        double sum = 0;
        for (int i = 0; i < poly.size()-1; i++) {
            if (liesOver(p, poly[i], poly[i+1])) {
                poly.pop_back();
                return BOUNDARY;
            }
            if (orientation(p, poly[i], poly[i+1]) > 0)
                sum += angle(poly[i], p, poly[i+1]);
            else sum -= angle(poly[i], p, poly[i+1]);
        }

        poly.pop_back();
        if ((fabs(sum)) > PI) return INSIDE;
        return OUTSIDE;
    }

    #warning Line: only tested linePointDistance
    struct Line {
        // ax + by + c = 0
        // b == 0 means vertical
        double a, b, c;

        Line() {}
        Line(double slope, double intercept) {
            a = -slope;
            b = 1;
            c = -intercept;
        }
        Line(Point &p1, Point &p2) {
            if (p1.x == p2.x) { // vertical
                a = 1;
                b = 0;
                c = -p1.x;
            }
            else {
                a = -1.0 * (p1.y - p2.y) / (p1.x - p2.x);
                b = 1;
                c = -1.0 * (a * p1.x) - p1.y;
            }
        }

        bool operator== (Line const &o) {
            return a == o.a && b == o.b && c == o.c;
        }
    };

    static bool isVertical(Line &l) {
        return l.b == 0;
    }

    static double getSlope(Line &l) {
        assert(!isVertical(l));
        return -l.a;
    }

    static double getIntercept(Line &l) {
        assert(!isVertical(l));
        return -l.c;
    }

    static Line getPerpendicularAtPoint(Line &l, Point &p) {
        double pslope = -1 / getSlope(l);
        double pintercept = p.y - pslope * p.x;
        return Line(pslope, pintercept);
    }

    static long double linePointDistance(Line &l, Point &p) {
        return abs(l.a * p.x + l.b * p.y + l.c) / sqrtl(l.a * l.a + l.b * l.b);
    }
};

using GI = Geometry<int>;
using GF = Geometry<long double>;
using IPoint = GI::Point;
using FPoint = GF::Point;

/*
Emergency definitions using complex<double>:
Vector addition:              a + b
Scalar multiplication:        r * a
Dot product:                  (conj(a) * b).x
Cross product:                (conj(a) * b).y
Squared distance:             norm(a - b)
```

```
Euclidean distance:           abs(a - b)
Angle of elevation:           arg(b - a)
Slope of line (a, b):         tan(arg(b - a))
Polar to cartesian:           polar(r, theta)
Cartesian to polar:           point(abs(p), arg(p))
Rotation about the origin:    a * polar(1.0, theta)
Rotation about pivot p:       (a-p) * polar(1.0, theta) + p
Angle ABC:                    abs(remainder(arg(a-b) - arg(c-b), 2.0 * M_PI))
Project p onto vector v:      v * dot(p, v) / norm(v);
Project p onto line (a, b):   a + (b - a) * dot(p - a, b - a) / norm(b - a)
Reflect p across line (a, b): a + conj((p - a) / (b - a)) * (b - a)
*/
```

3.5 Halfplane Intersection

```
const long double BOX_INF = 1e9;

#warning Halfplane keeps its LEFT side
struct Halfplane {
    // Line that goes through 'p' and with direction vector 'pq'
    FPoint p, pq;
    long double angle;

    Halfplane() {}
    Halfplane(FPoint &a, FPoint &b) : p(a), pq(b - a) {
        angle = atan2l(pq.y, pq.x);
    }

    // returns whether 'r' is outside this halfplane
    bool out(FPoint const &r) {
        return GF::cross(pq, r - p) < -EPS;
    }

    bool operator<(Halfplane const &o) const {
        return angle < o.angle;
    }
};

FPoint inter(Halfplane const &s, Halfplane const &t) {
    long double alpha = GF::cross((t.p - s.p), t.pq) / GF::cross(s.pq, t.pq);
    return s.p + (s.pq * alpha);
}

vector<FPoint> halfplaneIntersection(vector<Halfplane>& H) {
    FPoint box[4] = {  // Bounding box in CCW order
        {BOX_INF, BOX_INF},
        {-BOX_INF, BOX_INF},
        {-BOX_INF, -BOX_INF},
        {BOX_INF, -BOX_INF}
    };

    for (int i = 0; i < 4; i++) {
        Halfplane aux(box[i], box[(i+1)%4]);
        H.push_back(aux);
    };

    sort(all(H)); // sorted by atan2 angle, that is, just below x axis on the left and then full ccw turnaround

    deque<Halfplane> dq;
    int len = 0;
    for (int i = 0; i < H.size(); i++) {
        while (len > 1 && H[i].out(inter(dq[len-1], dq[len-2]))) {
            dq.pop_back();
            --len;
        }
        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(GF::cross(H[i].pq, dq[len-1].pq)) < EPS) {
            // Opposite parallel half-planes that ended up checked against each other.
            if (GF::dot(H[i].pq, dq[len-1].pq) < 0.0)
                return vector<FPoint>();

            // Same direction half-plane: keep only the leftmost half-plane.
            if (H[i].out(dq[len-1].p)) {
                dq.pop_back();
                --len;
            }
            else continue;
        }
        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<FPoint>();

    // Reconstruct the convex polygon from the remaining half-planes.
    vector<FPoint> 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;
}
```

3.6 Min Enclosing Circle

```
struct Point {
  double x, y;
};

class MinEnclosingCircle {
  double distance(const Point &a, const Point &b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
  }

  bool isInsideCircle(const Point &p, const Point &c, double r) {
    return distance(p, c) <= r + 1e-9;
  }

  Point circleCenter(const Point &a, const Point &b, const Point &c) {
    double ax = a.x, ay = a.y;
    double bx = b.x, by = b.y;
    double cx = c.x, cy = c.y;

    double d = 2 * (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by));
    double ux =
        ((ax * ax + ay * ay) * (by - cy) + (bx * bx + by * by) * (cy - ay) +
         (cx * cx + cy * cy) * (ay - by)) /
        d;
    double uy =
        ((ax * ax + ay * ay) * (cx - bx) + (bx * bx + by * by) * (ax - cx) +
         (cx * cx + cy * cy) * (bx - ax)) /
        d;

    return {ux, uy};
  }

  pair<Point, double> circleFromThreePoints(const Point &a, const Point &b,
                                            const Point &c) {
    Point center = circleCenter(a, b, c);
    double radius = distance(center, a);
    return {center, radius};
  }

  pair<Point, double> circleFromTwoPoints(const Point &a, const Point &b) {
    Point center = {(a.x + b.x) / 2, (a.y + b.y) / 2};
    double radius = distance(a, b) / 2;
    return {center, radius};
  }

public:
  pair<Point, double> mec(vector<Point> &points, vector<Point> boundary = {}) {
    if (points.empty() || boundary.size() == 3) {
      if (boundary.size() == 0) {
        return {{0, 0}, 0}; // No points
      }
      if (boundary.size() == 1) {
        return {boundary[0], 0}; // Single point
      }
      if (boundary.size() == 2) {
        return circleFromTwoPoints(boundary[0], boundary[1]);
      }
      return circleFromThreePoints(boundary[0], boundary[1], boundary[2]);
    }

    // Randomly pick a point
    srand(time(0));
    random_shuffle(points.begin(), points.end());

    Point p = points.back();
    points.pop_back();

    auto circle = mec(points, boundary);

    if (isInsideCircle(p, circle.first, circle.second)) {
      points.push_back(p);
      return circle;
    }

    boundary.push_back(p);
    auto result = mec(points, boundary);
    boundary.pop_back();
    points.push_back(p);
    return result;
  }
};
```

3.7 Tetrahedron Volume

```
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
using pdd = pair<double, double>;
using ll = long long;
using ull = unsigned long long;

template <typename T> struct Point {
  T x, y, z;
  Point() : x{0}, y{0}, z{0} {}
  Point(T x, T y, T z) : x{x}, y{y}, z{z} {}

  T dot(Point<T> &other) { return x * other.x + y * other.y + z * other.z; }

  T cross(Point<T> &other) {
    return x * other.x + x * other.y + x * other.z + y * other.x + y * other.y +
           y * other.z + z * other.x + z * other.y + z * other.z;
  }

  friend istream &operator>>(istream &is, Point<T> &p) {
    is >> p.x >> p.y >> p.z;
    return is;
  }

  Point<T> operator-(Point<T> &other) {
    return Point<T>(x - other.x, y - other.y, z - other.z);
  }
};

double volume(Point<double> &p1, Point<double> &p2, Point<double> &p3,
              Point<double> &p4) {
  auto pa = p1 - p4, pb = p2 - p4, pc = p3 - p4;

  double determinant = pa.x * (pb.y * pc.z - pc.y * pb.z) -
                       pb.x * (pa.y * pc.z - pc.y * pa.z) +
                       pc.x * (pa.y * pb.z - pb.y * pa.z);
  return abs(determinant) / 6.0;
}

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int t;
  cin >> t;

  while (t--) {
    Point<double> p1, p2, p3, p4;
    cin >> p1 >> p2 >> p3 >> p4;
    cout << fixed << setprecision(6) << volume(p1, p2, p3, p4) << '\n';
  }
}
```

4 Graphs

4.1 Bipartite

```
/*
 * Basically, checks if the graph can be colored with 2 colors.
 * If it's possible, the graph is bipartite.
 */

vector<vector<int>> AL;
bool bipartite() {
  int s = 0;
  queue<int> q;
  q.push(s);
  vi color(n, INF);
  color[s] = 0;
  bool isBipartite = true;              // add a Boolean flag
  while (!q.empty() && isBipartite) { // as with original BFS
    int u = q.front();
    q.pop();
```

```
        for (auto &v : AL[u]) {
          if (color[v] == INF) {
            color[v] = 1 - color[u]; // just record two colors
            q.push(v);
          } else if (color[v] == color[u]) { // u & v have same color
            isBipartite = false;              // a coloring conflict :(
            break;                            // optional speedup
          }
        }
      }
      return isBipartite;
    }
```

4.2 Bridges

```
int n;
vector<vector<int>> adj;
vector<bool> visited;
vector<int> tin, low;
vector<pair<int, int>> bridges;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) 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])
                bridges.push_back({v, to});
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    bridges.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

4.3 Cutpoints

```
int n;
vector<vector<int>> adj;
vector<bool> visited, is_cutpoint;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) 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] && p!=-1)
                is_cutpoint[v] = true;
            ++children;
        }
    }
    if(p == -1 && children > 1)
        is_cutpoint[v] = true;
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    is_cutpoint.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

4.4 Dijkstra

```
// {length, vertex}
vector<vector<ii>> adj;

vector<int> dijkstra(int s) {
    int n = adj.size();
    vector<int> d(n+1, INF);
    d[s] = 0;

    priority_queue<ii, vector<ii>, greater<ii>> pq;
    pq.push({0, s});

    while (!pq.empty()) {
        auto [dist, v] = pq.top(); pq.pop();

        if (dist != d[v])
            continue;

        for (auto [len, next] : adj[v]) {
            int newDist = dist + len;
            if (newDist < d[next]) {
                d[next] = newDist;
                pq.push({newDist, next});
            }
        }
    }
    return d;
}
```

4.5 Dinic

```
// Dinic's algrotihm for solving max flow.
// Copy-pasted from Halim's book.
// O(V^2E), while edmonds-karp is O(VE^2)
//
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef tuple<int, ll, ll> edge;
typedef vector<int> vi;
typedef pair<int, int> ii;
const ll INF = 1e18; // large enough
class max_flow {
private:
  int V;
  vector<edge> EL;
  vector<vi> AL;
  vi d, last;
  vector<ii> p;
  bool BFS(int s, int t) { // find augmenting path
    d.assign(V, -1);
    d[s] = 0;
    queue<int> q({s});
    p.assign(V, {-1, -1}); // record BFS sp tree
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      if (u == t)
        break;                                      // stop as sink t reached
      for (auto &idx : AL[u]) {                     // explore neighbors of u
        auto &[v, cap, flow] = EL[idx];             // stored in EL[idx]
        if ((cap - flow > 0) && (d[v] == -1))       // positive residual edge
          d[v] = d[u] + 1, q.push(v), p[v] = {u, idx}; // 3 lines in one!
      }
    }
    return d[t] != -1; // has an augmenting path
  }
  ll send_one_flow(int s, int t, ll f = INF) { // send one flow from s->t
    if (s == t)
      return f; // bottleneck edge f found
    auto &[u, idx] = p[t];
    auto &cap = get<1>(EL[idx]), &flow = get<2>(EL[idx]);
    ll pushed = send_one_flow(s, u, min(f, cap - flow));
    flow += pushed;
    auto &rflow = get<2>(EL[idx ^ 1]); // back edge
    rflow -= pushed;                   // back flow
    return pushed;
  }
  ll DFS(int u, int t, ll f = INF) { // traverse from s->t
```

```
      if ((u == t) || (f == 0))
        return f;
      for (int &i = last[u]; i < (int)AL[u].size(); ++i) { // from last edge
        auto &[v, cap, flow] = EL[AL[u][i]];
        if (d[v] != d[u] + 1)
          continue; // not part of layer graph
        if (ll pushed = DFS(v, t, min(f, cap - flow))) {
          flow += pushed;
          auto &rflow = get<2>(EL[AL[u][i] ^ 1]); // back edge
          rflow -= pushed;
          return pushed;
        }
      }
      return 0;
    }

  public:
    max_flow(int initialV) : V(initialV) {
      EL.clear();
      AL.assign(V, vi());
    }
    // if you are adding a bidirectional edge u<->v with weight w into your
    // flow graph, set directed = false (default value is directed = true)
    void add_edge(int u, int v, ll w, bool directed = true) {
      if (u == v)
        return;                                // safeguard: no self loop
      EL.emplace_back(v, w, 0);                // u->v, cap w, flow 0
      AL[u].push_back(EL.size() - 1);          // remember this index
      EL.emplace_back(u, directed ? 0 : w, 0); // back edge
      AL[v].push_back(EL.size() - 1);          // remember this index
    }
    ll edmonds_karp(int s, int t) {
      ll mf = 0;                     // mf stands for max_flow
      while (BFS(s, t)) {            // an O(V*E^2) algorithm
        ll f = send_one_flow(s, t); // find and send 1 flow f
        if (f == 0)
          break; // if f == 0, stop
        mf += f; // if f > 0, add to mf
      }
      return mf;
    }
    ll dinic(int s, int t) {
      ll mf = 0;                  // mf stands for max_flow
      while (BFS(s, t)) {         // an O(V^2*E) algorithm
        last.assign(V, 0);        // important speedup
        while (ll f = DFS(s, t)) // exhaust blocking flow
          mf += f;
      }
      return mf;
    }
  };
```

4.6 Eulerian Graph

```
  /**
   * Eulerian path --> path that traverses each edge in the graph exactly once.
   * Eulerian tour --> eulerian path that starts and finishes at the same vertex,
   * Eulerian graph --> graph with eulerian tour.
   * an undirected graph is eulerian if it's connected and all vertices have an
   * even degree. a directed graph is eulerian if it's connected and all vertices
   * have in_degree = out_degree.
   */

  // hierholzer finds an eulerian path if it exists.
  // can't say I understand everything here but whatever. I'm way too short of
  // time for that
  int N;
  vector<vi> AL; // Directed graph
  vi hierholzer(int s) {
    vi ans, idx(N, 0), st;
    st.push_back(s);
    while (!st.empty()) {
      int u = st.back();
      if (idx[u] < (int)AL[u].size()) { // still has neighbor
        st.push_back(AL[u][idx[u]]);
        ++idx[u];
      } else {
        ans.push_back(u);
        st.pop_back();
      }
    }
    reverse(ans.begin(), ans.end());
    return ans;
  }
```

4.7 Floyd Warshall

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

4.8 Ford Fulkerson

```
class FordFulkerson {
    bool __dfs(int u, int t) {
        visited[u] = 1;
        if (u == t) return true;
        for (int next : adj[u]) {
            if (!visited[next] && cap[u][next]) {
                if (__dfs(next, t)) {
                    p[next] = u;
                    return true;
                }
            }
        }
        return false;
    }

    bool dfs(int u, int t) {
        visited.assign(n, 0);
        p.assign(n, -1);
        return __dfs(u, t);
    }

public:
    vector<vector<int>> adj, cap;
    vector<int> p, visited;
    int n;

    FordFulkerson(vector<vector<int>> &adj, vector<vector<int>> &cap) : adj(adj), cap(cap) {
        n = adj.size();
    }

    int maxflow(int s, int t) {
        int maxflow = 0;
        while (dfs(s, t)) {
            int increment = INF;
            for (int u = t; p[u] >= 0; u = p[u])
                increment = min(increment, cap[p[u]][u]);

            for (int u = t; p[u] >= 0; u = p[u]) {
                cap[p[u]][u] -= increment;
                cap[u][p[u]] += increment;
            }
            maxflow += increment;
        }
        return maxflow;
    }
};

/*
Tip about matching: do not try to save memory by only putting edges when going, because
some bad stuff can happen when you start splitting vertices (for vertex capacity).
*/
```

4.9 Hungarian

```
int A[251][251];

// Matrix A is n*m - O(n^2m). Returns the index chosen for each line (1
// indexed).
#warning A is 1 based
vector<int> hungarian(int n, int m) {
  vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
  for (int i = 1; i <= n; ++i) {
    p[0] = i;
    int j0 = 0;
    vector<int> minv(m + 1, INF);
    vector<bool> used(m + 1, false);
    do {
      used[j0] = true;
      int i0 = p[j0], delta = INF, j1;
      for (int j = 1; j <= m; ++j)
        if (!used[j]) {
          int cur = A[i0][j] - u[i0] - v[j];
          if (cur < minv[j])
```

```
            minv[j] = cur, way[j] = j0;
          if (minv[j] < delta)
            delta = minv[j], j1 = j;
        }
      for (int j = 0; j <= m; ++j)
        if (used[j])
          u[p[j]] += delta, v[j] -= delta;
        else
          minv[j] -= delta;
      j0 = j1;
    } while (p[j0] != 0);
    do {
      int j1 = way[j0];
      p[j0] = p[j1];
      j0 = j1;
    } while (j0);
  }
  vector<int> ans(n + 1);
  for (int j = 1; j <= m; ++j)
    ans[p[j]] = j;
  return ans;
  // int cost = -v[0];
}
```

4.10 Kahn

```
/*
  Kahn's algorithm
  O(V+E), aka linear
  vector<int> topological_order will have the topological order after calling kahn()
  if, after kahn's queue, any indegree is not 0, there is a loop (so kahn returns false)
  if there is no loop, kahn returns true
*/

vector<vector<int>> adj;
vector<int> topological_order;

bool kahn() {
    int n = adj.size();
    vector<int> indegree(n, 0);
    topological_order.clear();
    topological_order.reserve(n);

    for (int i = 0; i < n; i++)
        for (int next : adj[i])
            indegree[next]++;

    queue<int> q;
    for (int i = 0; i < n; i++)
        if (!indegree[i])
            q.push(i);

    while (!q.empty()) {
        int u = q.front(); q.pop();
        topological_order.push_back(u);

        for (int next : adj[u]) {
            indegree[next]--;
            if (!indegree[next])
                q.push(next);
        }
    }

    // if any node has indeg > 0, then there is a cycle
    for (int i = 0; i < n; i++)
        if (indegree[i])
            return false; // cyclic
    return true; // acyclic
}
```

4.11 Kosaraju

```
vector<bool> visited;

void dfs(int v, vector<vector<int>> const& adj, vector<int> &output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

void strongly_connected_components(vector<vector<int>> const& adj,
                                  vector<vector<int>> &components,
                                  vector<vector<int>> &adj_cond) {
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // will be sorted by tout

    visited.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end()); // now reverse tout order

    vector<int> roots(n, 0);

    for (auto v : order)
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            sort(component.begin(), component.end());
            components.push_back(component);
            int root = component.front();
            for (auto u : component)
                roots[u] = root;
        }

    // optional: condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back(roots[u]);
}
```

4.12 Kruskal

```
/*
  MST (Kruskal algorithm)
  Utilizes UnionFind and Edge structures. Edge only accepts integer edge weights.
  For maximum spanning tree, just turn the edge weights negative.
  Also works for minimum edge product.
*/

struct UnionFind {
    vector<int> parent, size;

    UnionFind(int n) {
        parent.reserve(n);
        size.assign(n, 1);

        for (int i = 0; i < n; i++)
            parent.push_back(i);
    }

    int find(int v) {
        if (v == parent[v])
            return v;
        return parent[v] = find(parent[v]);
    }

    void unionSets(int a, int b) {
        a = find(a);
        b = find(b);
        if (a != b) {
            if (size[a] < size[b])
                swap(a, b);
            parent[b] = a; // subordinate b to a (smaller to bigger)
            size[a] += size[b];
        }
    }
};

struct Edge {
    int u, v, weight;

    Edge(int _u, int _v, int _weight) : u(_u), v(_v), weight(_weight) {}

    bool operator<(Edge &other) const {
        return weight < other.weight;
    }
};

struct Kruskal {
    int n;
    vector<Edge> edges;
    Kruskal(int n, vector<Edge> &edges) : n(n), edges(edges) {}

    vector<Edge> mst() {
        sort(edges.begin(), edges.end());
        UnionFind uf = UnionFind(n);
        vector<Edge> mst;
        mst.reserve(n-1);

        for (Edge e : edges) {
            if (uf.find(e.u) != uf.find(e.v)) {
                mst.push_back(e);
                uf.unionSets(e.u, e.v);
            }
            if (mst.size() == n-1)
                break;
        }
        return mst;
    }
};
```

4.13 Kuhn

```
/*
Time complexity: O(VE)
It is very improvable through heuristics. For example, run a loop and match simple edges (i.e., some
simple edge that matches a dude on the left to a dude on the right). Keep track of what left vertices
were used. Then, run Kuhn on the unvisited vertices. If you really feel like it, you can even random
shuffle the heuristics order.

Konig's theorem: the cardinality of a minimum vertex cover of a bipartite graph is the cardinality of
its maximum cardinality bipartite matching.
No proof here.
An algorithm to get the vertex cover itself given a matching is the following:
For every edge in the matching that goes from L->R, if the vertex on R is an ending of ANY alternating
path, then, add it to the vertex cover. Else, add the vertex on L to the vertex cover. Do this for every
edge on the matching and its done.
Remembering that an alternating path is a path that alternates between unmatched and matched edges.
*/

// REGULAR KUHN
// adjacency of LEFT vertices
vector<vector<int>> adj;
// match[i] is the idx of the left vertex matched to the ith right vertex (match.size() == RIGHT)
vector<int> match;
vector<bool> used;

bool tryKuhn(int v) {
    if (used[v])
        return false;
    used[v] = true;
    for (int to : adj[v]) {
        if (match[to] == -1 || tryKuhn(match[to])) {
            match[to] = v;
            return true;
        }
    }
    return false;
}

int getMatching() {
    int matchingSize = 0;
    for (int v = 0; v < adj.size(); ++v) {
        used.assign(adj.size(), false);
        matchingSize += tryKuhn(v);
    }
    return matchingSize;
}

signed main() {
    adj.assign(LEFT, {});
    match.assign(RIGHT, -1);
    int m = getMatching();
}


// HEURISTICS VERSION (everything else is equal)
int getMatching() {
    vector<bool> used1(adj.size(), false);
    int matchingSize = 0;
    for (int v = 0; v < adj.size(); ++v) {
        for (int to : adj[v]) {
            if (match[to] == -1) {
                match[to] = v;
                used1[v] = true;
                matchingSize++;
                break;
            }
        }
    }
```

```
        for (int v = 0; v < adj.size(); ++v) {
            if (used1[v])
                continue;
            used.assign(adj.size(), false);
            matchingSize += tryKuhn(v);
        }
        return matchingSize;
    }
```

4.14 Lca Max Edge

```
    struct LCA {
        int n, l;
        vector<vector<int>> tree;
        int timer;
        vector<int> tin, tout;
        vector<vector<int>> up, upMaxEdge;
        map<ii, int> weights;

        LCA (vector<vector<int>> &_tree, map<ii, int> &_weights, int root) : tree(_tree), weights(_weights) {
            n = tree.size();
            tin.resize(n);
            tout.resize(n);
            timer = 0;
            l = ceil(log2(n));
            up.assign(n, vector<int>(l + 1));
            upMaxEdge.assign(n, vector<int>(l + 1));
            weights[{root, root}] = 0;
            dfs(root, root);
        }

        void dfs(int v, int p) {
            tin[v] = ++timer;
            up[v][0] = p;
            int weightVP = 0;
            if (weights.count({v, p}))
                weightVP = weights[{v, p}];
            else
                weightVP = weights[{p, v}];
            upMaxEdge[v][0] = weightVP;

            for (int i = 1; i <= l; ++i) {
                up[v][i] = up[up[v][i-1]][i-1]; // 2^i up from v is the same as 2^(i-1) up from (2^(i-1) up from v [
                    its parent]))
                upMaxEdge[v][i] = max(upMaxEdge[v][i-1], upMaxEdge[up[v][i-1]][i-1]); // max from first half and
                    second half
            }

            for (int u : tree[v])
                if (u != p)
                    dfs(u, v);
            tout[v] = ++timer;
        }

        bool is_ancestor(int u, int v) {
            return tin[u] <= tin[v] && tout[u] >= tout[v];
        }

        int lca(int u, int v) {
            if (is_ancestor(u, v))
                return u;
            if (is_ancestor(v, u))
                return v;
            for (int i = l; i >= 0; --i)
                if (!is_ancestor(up[u][i], v))
                    u = up[u][i];
            return up[u][0];
        }

        int maxEdgeFromUToLca(int u, int lca) {
            if (u == lca) // if u is a direct parent of v, then the max edge is 0
                return 0;

            // paths are built like follows: if the dude being checked (up[u][i]) is
            // not ancestor of the lca, then u is lifted up. That is the moment where
            // you apply the max(...).
            // After the loop ends, u is not the lca, but up[u][0] is (its parent).
            // That's why you still have to do one more max at the end.
            int maxEdge = 0;
            for (int i = l; i >= 0; --i) {
                if (!is_ancestor(up[u][i], lca)) {
                    maxEdge = max(maxEdge, upMaxEdge[u][i]);
                    u = up[u][i];
                }
            }
            maxEdge = max(maxEdge, upMaxEdge[u][0]);
            return maxEdge;
        }
        int lcaMaxEdge(int u, int v) {
            int ancestor = lca(u, v);
            return max(maxEdgeFromUToLca(u, ancestor), maxEdgeFromUToLca(v, ancestor));
        }
    };
```

4.15 Max Flow

```
#include <bits/stdc++.h>
using namespace std;
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int> &parent) {
  fill(parent.begin(), parent.end(), -1);
  parent[s] = -2;
  queue<pair<int, int>> q;
  q.push({s, 1e9});

  while (!q.empty()) {
    int cur = q.front().first;
    int flow = q.front().second;
    q.pop();

    for (int next : adj[cur]) {
      if (parent[next] == -1 && capacity[cur][next]) {
        parent[next] = cur;
        int new_flow = min(flow, capacity[cur][next]);
        if (next == t)
          return new_flow;
        q.push({next, new_flow});
      }
    }
  }

  return 0;
}

int maxflow(int s, int t) {
  int flow = 0;
  vector<int> parent(n);
  int new_flow;

  while (new_flow = bfs(s, t, parent)) {
    flow += new_flow;
    int cur = t;
    while (cur != s) {
      int prev = parent[cur];
      capacity[prev][cur] -= new_flow;
      capacity[cur][prev] += new_flow;
      cur = prev;
    }
  }

  return flow;
}
```

4.16 Min Cost Max Flow

```
/*
Time complexity: O(FVE),
although it is expectable to be MUCH better than that.
O(F) is the very rare Ford-Fulkerson worst case.
O(VE) is the also very rare SPFA worst case.
*/
struct MinCostMaxFlow {
    bool spfa(int s, int t) {
        int n = adj.size();
        d.assign(n, INF);
        vector<bool> inqueue(n, false);
        queue<int> q;
        p.assign(n, -1);

        d[s] = 0;
        q.push(s);
        inqueue[s] = true;
        while (!q.empty()) {
            int v = q.front(); q.pop();
            inqueue[v] = false;

            for (int to : adj[v]) {
                int len = cost[v][to];
                if (d[v] != INF && d[v] + len < d[to] && cap[v][to] > 0) {
                    d[to] = d[v] + len;
                    p[to] = v;
                    if (!inqueue[to]) {
                        q.push(to);
                        inqueue[to] = true;
                    }
                }
            }
        }

        return d[t] != INF;
    }

    vector<vector<int>> adj, cap, cost;
    vector<int> p, d;
    int n;

    MinCostMaxFlow(vector<vector<int>> &adj, vector<vector<int>> &cap, vector<vector<int>> &cost) : adj(adj),
        cap(cap), cost(cost) {
        n = adj.size();
    }

    // {flow, cost}
    ii maxflow(int s, int t) {
        int maxflow = 0;
        int cost = 0;

        while (spfa(s, t)) {
            int increment = INF;
            for (int u = t; p[u] >= 0; u = p[u])
                increment = min(increment, cap[p[u]][u]);

            for (int u = t; p[u] >= 0; u = p[u]) {
                cap[p[u]][u] -= increment;
                cap[u][p[u]] += increment;
            }
            cost += increment * d[t];
            maxflow += increment;
        }
        return {maxflow, cost};
    }
};
```

4.17 Prim

```
#include <bits/stdc++.h>

#ifdef DEBUG
#define PRINT(s) std::cout << s << '\n';
#endif

using namespace std;
using pii = pair<int, int>;
using ull = unsigned long long;
using ll = long long;

// O(E log V)
// For finding minimum spanning trees
ull prim(vector<vector<pii>> &adj, vector<bool> &visited, int og,
         int &num_visited) {
  priority_queue<pii, vector<pii>, greater<pii>> pq;

  ull cost = 0;
  // vector<bool> visited(adj.size(), false);

  visited[og] = true;
  int n = adj.size();

  for (auto &[w, v] : adj[og])
    if (!visited[v])
      pq.push({w, v});

  while (!pq.empty() && num_visited != n - 1) {
    auto [w, u] = pq.top();
    pq.pop();

    if (visited[u])
      continue;

    visited[u] = true;
    num_visited++;
    cost += w;

    for (auto &[wv, v] : adj[u])
      if (!visited[v])
        pq.push({wv, v});
  }

  return cost;
}
```

4.18 Spfa

```
const int INF = 1e9;
// {vertex, distance}
vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {
    int n = adj.size();
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front(); q.pop();
        inqueue[v] = false;

        for (auto [to, len] : adj[v]) {
            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                    cnt[to]++;
                    if (cnt[to] > n)
                        return false;  // negative cycle
                }
            }
        }
    }
    return true;
}
```

5 Linear Sorting

5.1 Radix Sort

```
#include <bits/stdc++.h>

using namespace std;
using ll = long long;
using ull = unsigned long long;

// If numbers are too large, MAYBE increase base.
// Once I had to use 2^15 to get AC
// Some numbers are using ll. In practice, this is only needed for very large
// bases.

// int base = 32768;
const int base = 512; // IDK a good value

int get_digit(int a, int divisor) { return a / divisor % base; }

bool cmp(int a, int b, int divisor) {
  return get_digit(a, divisor) < get_digit(b, divisor);
}

void counting_sort(vector<int> &vec, vector<int> &output, int divisor) {
  int l = INT32_MAX, r = 0;

  vector<int> aux(vec.begin(), vec.end());
  for (auto &v : aux) {
    v = get_digit(v, divisor);
    l = min(l, v);
    r = max(r, v);
  }

  vector<int> f(r - l + 1);

  for (auto &v : aux)
    ++f[v - l];

  for (int i = 1; i < f.size(); ++i)
    f[i] = f[i - 1] + f[i];

  for (ll i = vec.size() - 1; i >= 0; --i) {
    int d = aux[i];
    output[f[d - l] - 1] = vec[i];
    f[d - l]--;
  }
}

void radix_sort(vector<int> &vec) {
  auto [il, ir] = minmax_element(vec.begin(), vec.end());
  int l = *il, r = *ir;
```

```
    int num_digits = 1;
    int tmp = r;
    while (tmp >= base) {
      num_digits++;
      tmp = tmp / base;
    }

    vector<int> a(vec.begin(), vec.end()), b(vec.begin(), vec.end());

    auto *output = &a;
    auto *aux = &b;

    int divisor = 1;
    for (int i = 0; i < num_digits; ++i) {
      swap(aux, output);
      counting_sort(*aux, *output, divisor);
      divisor *= base;
    }

    for (int i = 0; i < vec.size(); ++i)
      vec[i] = (*output)[i];
  }
```

6 Math

6.1 Comb

```
// precomputes factorials and inverse factorials
// O(n log mod) needed for the inverse factorials

tuple<int, int, int> extendedGcd(int a, int b) {
    if (b == 0) return make_tuple(a, 1, 0);
    auto[q, w, e] = extendedGcd(b, a%b);
    return make_tuple(q, e, w-e*(a/b));
}

int multiplicativeInverse(int n, int mod) {
    auto[g, x, y] = extendedGcd(n, mod);
    return (x % mod + mod) % mod;
}

int mod;
vector<int> fac, ifac;
int comb(int n, int k) {
    return fac[n] * ifac[k] % mod * ifac[n-k] % mod;
}

int main() {
    const int MAXK = 5000 + 10;
    fac.resize(MAXK);
    ifac.resize(MAXK);
    fac[0] = ifac[0] = 1;
    for (int i = 1; i < MAXK; i++) {
        fac[i] = fac[i-1] * i % mod;
        ifac[i] = multiplicativeInverse(fac[i], mod);
    }
}
```

6.2 Divisor Count

```
#include <bits/stdc++.h>

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int ans = 0;
  int limit = t - 1;
  for (int i = 1; i * i <= limit; ++i) {
    if (limit % i)
      continue;

    ans++;
    int b = limit / i;
    if (b != i)
      ans++;
  }
}
```

6.3 Extended Euclides

```
// ax + by = gcd(a, b)
// returns [gcd, x, y]
tuple<int, int, int> extendedGcd(int a, int b) {
    if (b == 0) return make_tuple(a, 1, 0);
    auto[q, w, e] = extendedGcd(b, a%b);
    return make_tuple(q, e, w-e*(a/b));
}

int multiplicativeInverse(int n, int mod) {
    // (n)x + (mod)y = 1 (aka their difference is 1)
    n = (n % mod + mod) % mod;
    auto[g, x, y] = extendedGcd(n, mod);
    return (x % mod + mod) % mod;
}
```

6.4 Fast Exponentiation

```
long long binpow(long long a, long long b) {
  long long res = 1;
  while (b > 0) {
    if (b & 1)
      res = res * a;
    a = a * a;
    b >>= 1;
  }
  return res;
}
```

6.5 Fft

```
const double PI = 4*atan(1);
using cd = complex<double>;

void fft(vector<cd> &a) {
    int n = a.size();
    if (n == 1) return; // because a_0*z^0 = a_0 (evaluating at any point returns a_0)
    vector<cd> a0(n/2), a1(n/2);
    for (int i = 0; i < n/2; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0); fft(a1);
    double alpha = 2*PI/n;
    cd omega(cos(alpha), sin(alpha));
    cd omega_i = 1;

    for (int i = 0; i < n/2; i++) {
        a[i] = a0[i] + omega_i * a1[i];
        a[i+n/2] = a0[i] - omega_i * a1[i]; // -omega_i*a1[i] == (-1)*omega_i*a1[i] == omega_(n/2)*omega_i*a1[i]
             == omega_(i+n/2)*a1[i]
        omega_i *= omega;
    }
}

void ifft(vector<cd> &a) { // DFT(DFT(c0, c1, c2, ..., c(n-1))) = (nc0, nc(n-1), nc(n-2), ..., nc1)
    int n = a.size();
    fft(a);
    reverse(a.begin()+1, a.end());
    for (cd &i : a)
        i /= n;
}

vector<int> multiply(vector<int> &a, vector<int> &b) {
    vector<cd> fa(all(a)), fb(all(b));
    int n = 1;
    while (n < a.size() + b.size())
        n *= 2;
    fa.resize(n); fb.resize(n);
    fft(fa); fft(fb);

    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];

    ifft(fa);
    vector<int> res(n);
    for (int i = 0; i < n; i++)
        res[i] = round(fa[i].real());
    return res;
}

/*
About changing summing conditions.
Let's take a look at what a regular FFT condition looks like:
ans[k] = sum(i+j=k)       A[i] * B[j]
       = sum(i=0 until k) A[i] * B[k-i]
```

```
If you can write your desired sum condition as
sum(i=0 until r) A[ai] * B[bi]
such that ai+bi=r, then, you run FFT on A and B and your desired value ans[k] is on fft[r].
The deal is that it may happen that the indices of A and B do not sum to 'r'. In that case,
you will build a new vector (say, "A transformed" AT), such that its index, when summed to
B's index, equals 'r'.

Example below for the condition i-j=k:
ans[k] = sum(i-j=k) A[i] * B[j]
       = sum(i=k until n-1) A[i] * B[i-k]
       = sum(i=0 until n-1-k) A[i+k] * B[i]
We have that r = n-1-k. We need the sum of A's index and B's index to be r. Let's transform A.
It should hold for AT's new index 'f' that:
f+i = n-1-k
f = n-1-k-i
So, let's build a vector AT such that its index f = n-1-k-i points to A[i+k]. Therefore:
AT[n-1-k-i] = A[i+k]
AT[n-1-(i+k)] = A[i+k]
So, given A's index 'q', then:
AT[n-1-q] = A[q]

And, after we do the convolution, we have that
ans[k] = fft[n-1-k]
*/

// sum(i-j=k) A[i] * B[j]
vector<int> subtractionConvolution(vector<int> &a, vector<int> &b) {
    vector<int> at = a;
    reverse(all(at));
    vector<int> c = multiply(at, b);
    c.resize(a.size());
    reverse(all(c));
    return c;
}
```

6.6 Gauss

```
/**
  *
  *
  Gauss method for solving linear systems.
  vector<vector<double>> a --> matrix of the system.
  vector<double> ans --> solution.
  */

const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity or a big number

int gauss(vector<vector<double>> a, vector<double> &ans) {
  int n = (int)a.size();
  int m = (int)a[0].size() - 1;

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

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

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

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

6.7 Gcd Convolution

```
/*
O(nlogn)
GCD convolution:
d[k] = sum(every i,j such that gcd(i,j)=k) a[i]*b[j]

Idea:
We wanna compute the vector d[k] = sum(every i,j such that gcd(i, j)=k) a[i] * b[j]
Start by computing this arbitrary sum S[g]:
S[g] = sum of every d[k] such that g | k
= sum of a[i] * b[j] such that gcd(i, j) = k and g | k
= sum of a[i] * b[j] such that g | gcd(i, j) (1)
I claim it also equals this:
(sum of b[i] for every i such that g|i) * (sum of c[i] for every i such that g|i) (2)
Proof:
Every product of (1) has the property that g|i and g|j, so, they appear on (2). But also every pair
of (2) has the property of g | gcd(i, j). Therefore, they appear on (1), and (1) == (2).

Both of the parts of the product can be done in nlogn because of harmonic series.
Now, how to use S to build d? Well, since S[g] is the sum of d[k] for k multiple of g, then, I can just
use it as d[g] and subtract values of d[k] (k > g) already computed (in decreasing dp harmonic style).
*/

vector<int> gcdConvolution(vector<int> &a, vector<int> &b) {
    int n = min(a.size(), b.size());
    vector<int> S(n, 0);

    for (int i = 1; i < n; i++) {
        int as = 0;
        int bs = 0;
        for (int j = i; j < n; j += i) {
            as += a[j];
            bs += b[j];
        }
        S[i] = as * bs;
    }

    // here S is d itself
    for (int g = n-1; g >= 1; g--)
        for (int j = 2*g; j < n; j += g)
            S[g] -= S[j];
    return S;
}
```

6.8 Matrix

```
struct Matrix {
    int l, c;
    vector<vector<ll>> m;
    Matrix(int l, int c) : l(l), c(c) {
        m.assign(l, vector<ll>(c, 0));
    }

    Matrix operator* (Matrix &o) {
        Matrix res(l, o.c);
        for (int i = 0; i < l; i++)
            for (int j = 0; j < o.c; j++)
                for (int k = 0; k < c; k++)
                    res[i][j] = (res[i][j] + m[i][k] * o[k][j]) % MOD;
        return res;
    }

    vector<ll>& operator[](int idx) {
        return m[idx];
    }

    Matrix pow(ll e) {
        if (e == 0) return identity(l);
        if (e == 1) return *this;
        Matrix squared = (*this) * (*this);
        if (e % 2 == 0) return squared.pow(e/2);
        return squared.pow(e/2) * (*this);
    }

    Matrix identity(int sz) {
        Matrix mm(sz, sz);
        for (int i = 0; i < sz; i++)
            mm[i][i] = 1;
        return mm;
    }
};
```

6.9 Maximal Number Of Divisors

```
/*
Maximal number of divisors for any n-digit number.

4 -> 4 divisors for a 1-digit number
12 -> 12 divisors for a 2-digit number
...

4, 12, 32, 64, 128, 240, 448, 768, 1344, 2304, 4032, 6720, 10752, 17280, 26880,
41472, 64512, 103680, 161280, 245760, 368640, 552960, 860160, 1290240, 1966080,
2764800, 4128768, 6193152, 8957952, 13271040, 19660800, 28311552, 41287680,
59719680, 88473600, 127401984, 181665792, 264241152, 382205952, 530841600
*/
```

6.10 Mex

```
struct MexSet {
    map<int, int> inSet;
    set<int> notInSet;

    // n should be the maximum possible mex (aka set size)
    MexSet(int n) {
        for (int i = 0; i <= n; i++)
            notInSet.insert(i);
    }

    void insert(int elem) {
        inSet[elem]++;
        if (notInSet.count(elem))
            notInSet.erase(elem);
    }

    void erase(int elem) {
        int count = --inSet[elem];
        if (!count)
            notInSet.insert(elem);
    }

    int mex() {
        assert(notInSet.size() > 0);
        return *notInSet.begin();
    }
};
```

6.11 Ntt

```
const int MOD = 998244353;

inline int add(int a, int b) {
    a += b;
    if (a >= MOD) a -= MOD;
    return a;
}

inline int mul(int a, int b) {
    return a * b % MOD;
}

int pwr(int a, int b) {
    int r = 1;
    while (b) {
        if (b & 1) r = mul(r, a);
        a = mul(a, a);
        b >>= 1;
    }
    return r;
}

int inv(int x) {
    return pwr(x, MOD-2);
}

void ntt(vector<int> &a, bool rev) {
    int n = a.size();
    vector<int> b = a;
    int g = 1;
    while (pwr(g, MOD / 2) == 1)
        g++;
    if (rev)
        g = inv(g);
    for (int step = n / 2; step; step /= 2) {
        int w = pwr(g, MOD / (n / step));
        int wn = 1;
        for (int i = 0; i < n / 2; i += step) {
            for (int j = 0; j < step; j++) {
                int u = a[2 * i + j];
                int v = mul(wn, a[2 * i + j + step]);
                b[i + j] = add(u, v);
                b[i + n / 2 + j] = add(u, MOD - v);
            }
            wn = mul(wn, w);
        }
        swap(a, b);
    }
    if (rev) {
        int n1 = inv(n);
        for (int &x : a)
            x = mul(x, n1);
    }
}

vector<int> multiply(vector<int> a, vector<int> b) {
    int n = 1;
    while (n < a.size() + b.size())
        n *= 2;
    a.resize(n); b.resize(n);
    ntt(a, false); ntt(b, false);

    for (int i = 0; i < n; i++)
        a[i] = mul(a[i], b[i]);

    ntt(a, true);
    return a;
}

// discards everything after lastIndex
vector<int> pwr(vector<int> a, int b, int lastIndex) {
    vector<int> r(a.size(), 0); r[0] = 1; // neutral polynomial
    while (b) {
        if (b & 1) {
            r = multiply(r, a);
            r.resize(lastIndex+1);
        }
        a = multiply(a, a);
        a.resize(lastIndex+1);
        b >>= 1;
    }
    return r;
}
```

6.12 Phi

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}

void phiSieve(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++)
        if (phi[i] == i)
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
}
```

6.13 Probabilistic Prime Testing

```
class Main {
public
  static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (sc.hasNext()) {
      int N = sc.nextInt();
      System.out.printf("%d is ", N);
      BigInteger BN = BigInteger.valueOf(N);
      String R = new StringBuffer(BN.toString()).reverse().toString();
      int RN = Integer.parseInt(R);
      BigInteger BRN = BigInteger.valueOf(RN);
      if (!BN.isProbablePrime(10)) // certainty 10 is enough
        System.out.println("not prime.");
      else if ((N != RN) && BRN.isProbablePrime(10))
        System.out.println("emirp.");
      else
        System.out.println("prime.");
    }
  }
}
```

6.14 Range Xor

```
// xor from [0, n] (the same as [1, n])
int prefixXor(int n) {
    int mod = n % 4;
    if (mod == 0) return n;
    if (mod == 1) return 1;
    if (mod == 2) return n+1;
    return 0;
}

int rangeXor(int l, int r) {
    return prefixXor(l-1) ^ prefixXor(r);
}
```

6.15 Segmented Sieve

```
#include <bits/stdc++.h>
using namespace std;
// Same complexity as traditional sieve, but better constants because of cache.
// Also, memory complexity is O(sqrt(n) + S)
vector<int> segmented_sieve(int n) {

  int nsqrt = sqrt(n);

  // Block size
  const int S = max(nsqrt, (int)1e5);

  vector<int> primes, result;
  vector<char> is_prime(nsqrt + 2, true);

  for (int i = 2; i <= nsqrt; ++i) {
    if (is_prime[i]) {
      primes.push_back(i);
      for (int j = i * i; j <= nsqrt; j += i)
        is_prime[j] = false;
    }
  }

  vector<char> block(S);
  for (int k = 0; k * S <= n; ++k) {
    fill(block.begin(), block.end(), true);

    int start = k * S;
    for (auto &p : primes) {
      int start_idx = (start + p - 1) / p; // same as ceil(start/p)
      int j = max(start_idx, p) * p - start;
      for (; j < S; j += p)
        block[j] = false;
    }

    if (k == 0)
      block[0] = block[1] = false;
    for (int i = 0; i < S && start + i <= n; ++i)
      if (block[i])
        result.push_back(start + i);
  }
  return result;
}
```

6.16 Sieve Factors Divisors

```
vector<bool> sieve(int n) {
    vector<bool> isPrime(n+1, true);
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i * i <= n; i++)
        if (isPrime[i])
            for (int j = i * i; j <= n; j += i)
                isPrime[j] = false;
    return isPrime;
}

vector<int> factors(int n) {
    vector<int> f;
    for (int i = 2; i*i <= n; i++) {
        while (n % i == 0) {
            f.push_back(i);
            n /= i;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}

vector<vector<int>> divsSieve(int n) {
    vector<vector<int>> divisors(n+1);
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j += i)
            divisors[j].push_back(i);
    return divisors;
}
```

6.17 Sum Of Divisors

```
long long SumOfDivisors(long long num) {
  long long total = 1;

  for (int i = 2; (long long)i * i <= num; i++) {
    if (num % i == 0) {
      int e = 0;
      do {
        e++;
        num /= i;
      } while (num % i == 0);

      long long sum = 0, pow = 1;
      do {
        sum += pow;
        pow *= i;
      } while (e-- > 0);
      total *= sum;
    }
  }
  if (num > 1) {
    total *= (1 + num);
  }
  return total;
}
```

7 String Algorithms

7.1 Eertree

```
/*
An Eertree is a graph where each vertex represents a unique palindromic substring.
Every palindrome is made by adding a letter to the front and the back of a palindrome. The base cases
are the empty palindrome (of length 0) and the imaginary palindrome (of length -1). That is convenient
because it allows us to dinamically (online) extend an eertree only by keeping track of its already
existing palindromes.

Claim: every new unique palindromic substring gotten by appending a new character to the string
necessarily is a suffix of the string.
Proof: if it doesn't include the last character, it existed before it. So it must include the last (and
is therefore a suffix).

Claim: appending a letter to a string increases the amount of unique palindromic substrings by at most one.
Proof: if the letter is new, clearly, the only new palindrome is itself.
Else, let's assume 2 new palindromes were created. Then, there must be a suffix of the form:
cyczc
Where "c" is the new character added, with both "ycz" and "z" being palindromes, with "y" and "z" possibly
empty and with the 2 new palindromes being "cyczc" and "czc".

Let s' be the reverse of a string s. Then, because "ycz" is a palindrome, we have that:
cyczc
= cz'cy'c
But since z is a palindrome, z' = z, so
= czcy'c
And we have that "czc" already appeared before appending "c" (absurd).

So, our data structure starts with 2 vertices, the "real" length 0 palindromic substring, and the "imag"
length -1 palindromic substring. Every vertex has a "link" pointing to its longest proper palindromic
suffix and a vector of labeled edges indicating the palindromes that can be made by adding letters on
the start and end of the current string.

We will keep track of the longest palindromic suffix of the current string (from now on called "last",
and initialized as "real"), and will progressively add characters of the string we wanna build the tree of.
There is a function "getLink()" that returns the largest suffix that is also palindromic when appended the
new character.
It is clear that getLink(last) is the only CANDIDATE for being a new vertex, because any other suffix is
contained on getLink(last), and, therefore, falls into the absurd proven before (it is never new). So
we set last = getLink(last) (and from now on I'll refer to the new value as "last").

After we found such vertex, last will become c + last + c (that is, to[last][c]). Luckily this is valid
```

```
for len[last] = -1, too. If vertex c + last + c didn't exist, we simply create it.

Creating a vertex consists of finding its link and plugging the last's "to" to it. Its link is clearly
the longest palindromic substring that is NOT to[last][c], so you simply try to find it from last's link
(v = getLink(link[last])) and then append "c" to that. It is clear that operation is valid if len[v] is
>= 0 (because to[v][c] is a palindromic suffix contained in to[last][c], and, therefore, is already
initialized).
But what if len[v] = -1?
If len[last] = -1, then, last == imag, and if we are creating a vertex, it means that
to[last][c] = to[imag][c] is uninitialized. If we set the uninitialized value as REAL, then, everything
will work as intended.
If len[last] >= 0 (and len[v] = -1), it means to[v][c] is a suffix of to[last][c], and, because of our
theorem, it is already initialized. Therefore, it is safe to grab to[v][c].
*/

const int MAXN = 1e6;
const int E = 26; // alphabet

struct Eertree {
    const static int REAL = 0;
    const static int IMAG = 1;

    int len[MAXN+2], start[MAXN+2], link[MAXN+2], cnt[MAXN+2], to[MAXN+2][E];
    //map<int, int> to[MAXN+2];
    int s[MAXN+2]; // the actual string is s[2:n-1]
    int n = 2, sz = 2, last = REAL, totalPalindromes = 0;
    int lps, lpsIdx; // longest palindromic substring

    Eertree() {
        memset(len, 0, sizeof(len));
        memset(start, 0, sizeof(start));
        memset(link, 0, sizeof(link));
        memset(cnt, 0, sizeof(cnt));
        memset(to, 0, sizeof(to)); // REMOVE THIS WHEN USING std::map
        len[IMAG] = -1;
        link[IMAG] = link[REAL] = IMAG;
        s[IMAG] = s[REAL] = -1; // value that should not be on alphabet
    }

    int getLink(int v) {
        while (s[n - len[v] - 2] != s[n - 1])
            v = link[v];
        return v;
    }

    void addLetter(int c) {
        s[n++] = c;
        last = getLink(last);

        //if (!to[last].count(c)) {
        if (!to[last][c]) {
            len[sz] = len[last] + 2;
            //if (len[last] == -1 && len[getLink(link[last])] == -1)
            //    to[last][c] = REAL;
            link[sz] = to[getLink(link[last])][c];
            start[sz] = n-2-len[sz];
            to[last][c] = sz++;
        }
        last = to[last][c];
        cnt[last]++;
    }

    void setupData() {
        totalPalindromes = lps = lpsIdx = 0;
        // this iterates over every vertex
        for (int i = sz-1; i >= 2; i--) {
            totalPalindromes += cnt[i];
            cnt[link[i]] += cnt[i];
            if (len[i] > lps) {
                lps = len[i];
                lpsIdx = start[i];
            }
        }
    }

    inline int palindromeCount() { return totalPalindromes; }
    inline int uniquePalindromeCount() { return sz - 2; }
    inline ii longestPalindromicSubstring() { return {lpsIdx, lps}; }

    /*
    You can dinamically keep track of palindromic substring count by doing something like this:

    // v should be to[last][c], that is, "last" after adding a letter.
    int increment = 0;
    while (v != REAL) {
        increment++;
        v = link[v];
    }
    totalPalindromes += increment;

    While also keeping a rollback to undo operations.
    */
};
```

7.2 Kmp

```
/*
Some useful KMP properties:
k = i + 1 - pi[i] gives you the size of the minimum string that, when concatenated several
times with itself, plus a prefix of itself, gives n. If you want to build the original string
only by concatenating another string several times (i.e., without a prefix of itself), then
k must divide n.
if (n % k == 0) ans = k;
else ans = n;

For pattern matching, build the string pat+"#"+text and then whenever pi[i] = pat.size(),
it means that a pattern matched i - (pat.size()-1) positions before.
*/

vector<int> buildLps(string &s) {
    int n = s.length();
    vector<int> lps(n);
    for (int i = 1; i < n; i++) {
        int j = lps[i-1];
        while (j > 0 && s[i] != s[j])
            j = lps[j-1];
        if (s[i] == s[j])
            j++;
        lps[i] = j;
    }
    return lps;
}
```

7.3 String Hash

```
long long compute_hash(string const &s) {
  const int p = 31; // should be roughly the size of input alphabet. For lower
                    // and upper, use 53.
  const int m = 1e9 + 9;
  long long hash_value = 0;
  long long p_pow = 1;
  for (char c : s) {
    hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
    p_pow = (p_pow * p) % m;
  }
  return hash_value;
}
// precomputing the powers of p might give a performance boost
// 10^6 comparisons gives a collision chance of about 1e-3.
// to reduce that chance, hash the string s with 2 functions,
// each one with different p and m
```

7.4 Suffix Array

```
vector<int> sortCyclicShifts(string &s) {
    int n = s.size();
    const int alphabet = 256;
    vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
```

```
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
        if (c[p[n-1]] == n-1) break;
    }
    return p;
}

vector<int> getSuffixArray(string &s) {
    s += '$';
    vector<int> ans = sortCyclicShifts(s);
    s.pop_back();
    ans.erase(ans.begin());
    return ans;
}

/*
Kasai lcp construction algorithm. lcp[i] is the longest common prefix of the suffix at sa[i] and its next suffix
    .
Claim: given the indices i and j, the lcp of the suffixes at positions i and j of sa is
lcp(i, j) = min(lcp[i], lcp[i+1], ..., lcp[j-1]) (no proof here)

Algorithm idea: let i and j be some suffix (other than the last) and its next suffix. It is clear that the lcp
    of the suffixes (i+1) and (j+1) is exactly lcp[rank[i]]-1 (given that lcp[rank[i]] > 0 and that the suffixes
     i+1 and j+1 exist).
But since the (i+1)th suffix comes before the (j+1)th suffix, and given the claim above, then, the lcp of all
    the suffixes between the position of the (i+1)th suffix and the (j+1)th suffix is at least lcp[rank[i]]-1,
    that is:
lcp[rank[i+1]] and lcp[rank[i+1]+1] and lcp[rank[i+1]+2] and ... and lcp[rank[j+1]-1] all are >= lcp[rank[i]]-1
In particular,
lcp[rank[i+1]] >= lcp[rank[i]]-1

The algorithm below iterates through the suffixes in regular string order and keeps a variable "k" that
    indicates what would be the lcp of the previous suffix in STRING ORDER minus 1.
Then, we bruteforce the answer for the ith suffix but starting the comparison from the kth character.

Edge cases:
 - rank[i] == n-1: its lcp shouldn't be calculated, and setting k to 0 is neutral (complexity is still O(n) even
      though k = 0 means bruteforcing).
 - lcp[rank[i]-1] == 0: even though the inequality still holds, we set k to 0 instead of lcp[rank[i]-1]-1 so our
      search space never becomes negative.
*/

vector<int> getLcp(string &s, vector<int> &sa) {
    int n = s.size();
    int k = 0;
    vector<int> lcp(n-1, 0), rank(n, 0);

    for (int i = 0; i < n; i++)
        rank[sa[i]] = i;

    for (int i = 0; i < n; i++, k?k--:0) {
        if (rank[i] == n-1) {
            k = 0;
            continue;
        }
        int j = sa[rank[i]+1];
        while (i+k<n && j+k<n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
    }
    return lcp;
}

/*
Different substrings:
Let's count the total amount of substrings and then subtract the substrings we counted repeatedly. The total
    amount of substrings is:
n + (n-1) + (n-2) + ... + 1 (amount of prefixes starting at each letter)
= n*(n+1)/2

Now, let's go through each suffix in SA order and check how many prefixes of the current suffix were already
    counted before.
Suppose the current suffix differs from the previous suffix at the ith character of the string. This means that
    the first 'i' prefixes of the current suffix were already counted (because they match the previous suffix).
But what about the remaining prefixes that are not the first 'i' ones? Are they unique?
They are, because the prefix of length 'i+1' of the current suffix is strictly greater than any prefix of length
     'i+1' counted before (this is because we are iterating in the suffix array order, and, therefore, in
    increasing lexicographical order). That argument applies to prefixes of length 'i+2', 'i+3', and so on.
    Therefore, the only repeated substrings of the current suffix are the first 'i' ones. With that we have that
     the answer is:
n*(n+1)/2 - (sum of all i for every suffix other than the first)

And how can we calculate 'i'? Well, since 'i' is the first index that mismatches, it means that the first 'i'
```

```
    characters matched. That number is, by definition, the lcp of the current suffix. Therefore, the answer is:
n*(n+1)/2 - accumulate(all(lcp), 0ll);
*/

int differentSubstringCount(vector<int> &lcp) {
    int n = lcp.size()+1;
    return n*(n+1)/2 - accumulate(all(lcp), 0ll);
}

/*
Longest Repeated Substring:
The suffix array indirectly stores data about substring similarity. That is because the most similar substrings
are in consecutive positions of the suffix array, and, because the lcp array is about consecutive suffixes, then
     it
says a lot about substring similarity in general.
Say the lcp at some position is 'i'. That means two suffixes share 'i' characters, i.e., that substring of
    length 'i' happens at least twice.
That is why the longest repeated substring is the one that is pointed by the maximum lcp.
*/

pair<int, int> longestRepeatedSubstring(vector<int> &sa, vector<int> &lcp) {
    int lrsIndexAtLcp = max_element(all(lcp)) - lcp.begin();
    // (lrsIndexAtLcp)th suffix in SA order has the maximum lcp value
    int len = lcp[lrsIndexAtLcp];
    int idx = sa[lrsIndexAtLcp];
    return {idx, len};
}

// Indices are from SA and not regular string indices. For regular string indices, input rank[i] and rank[j]
    instead.
int longestCommonPrefix(int i, int j, SparseTable &st) {
    if (i > j) swap(i, j);
    return st.query(i, j-1);
}
```

7.5 Trie

```
struct Trie {
    vector<map<char, int>> next;
    vector<int> count;
    vector<int> mostFrequentContinuationCount;
    vector<int> mostFrequentContinuationId;
    vector<int> indexOnOrderedVector;

    Trie() { addNode(); } // you can reserve the vectors here if you need to

    int addNode() {
        next.push_back({});
        count.push_back(0);
        mostFrequentContinuationCount.push_back(-1);
        mostFrequentContinuationId.push_back(-1);
        indexOnOrderedVector.push_back(0);
        return next.size()-1;
    }

    void addWord(string const &s, int idx) {
        int cur = 0;
        for (char c : s) {
            if (!next[cur].count(c))
                next[cur][c] = addNode();
            cur = next[cur][c];
        }
        count[cur]++;
        indexOnOrderedVector[cur] = idx;
    }

    int getId(string const &s) {
        int cur = 0;
        for (char c : s) {
            if (!next[cur].count(c)) return -1;
            cur = next[cur][c];
        }
        return cur;
    }

    int getCount(string const &s) {
        return count[getId(s)];
    }

    // update 'mostFrequentContinuation' for every prefix of s
    void relax(string const &s) {
        int id = getId(s);
        int currentCount = count[id];
        int cur = 0;
        for (char c : s) {
            cur = next[cur][c];
            if (mostFrequentContinuationCount[cur] < currentCount) {
                mostFrequentContinuationCount[cur] = currentCount;
                mostFrequentContinuationId[cur] = id;
            }
        }
    }
};
```

7.6 Z Function

```
// z[i] is the length of the longest common prefix of s[0] and s[i]
// s[0] is undefined (you can set it to n manually).

vector<int> zFunction(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;
}
```

8 Utils

8.1 Binary Search

```
'''
Returns index of target, in ascending iterables.
For descending iterables, swap "<" with ">".
If number doesn't exist, returns -1.
'''
def binary_search(iterable, target, down_idx=0, top_idx=None):
    if top_idx is None:
        top_idx = len(iterable) - 1

    while down_idx <= top_idx:
        cur = (down_idx + top_idx) // 2
        if iterable[cur] == target:
            return cur
        elif iterable[cur] < target: #swap here
            down_idx = cur + 1
        else:
            top_idx = cur - 1
    return -1


'''
Returns index of smallest number in iterable bigger than or equal
to target, in ascending iterables.
Swapping ">=" with "<=" returns index of biggest number in iterable
smaller than or equal to target, in descending iterables.
If number doesn't exist, returns -1.
'''
def binary_search(iterable, target, down_idx=0, top_idx=None):
    if top_idx is None:
        top_idx = len(iterable) - 1
    res = -1

    while down_idx <= top_idx:
        cur = (down_idx + top_idx) // 2
        if iterable[cur] >= target: #swap here
            res = cur
            top_idx = cur - 1
        else:
            down_idx = cur + 1
    return res


'''
Returns index of smallest number in iterable bigger than or equal
to target, in descending iterables.
Swapping ">=" with "<=" returns index of biggest number in iterable
smaller than or equal to target, in ascending iterables.
If number doesn't exist, returns -1.
'''
def binary_search(iterable, target, down_idx=0, top_idx=None):
    if top_idx is None:
        top_idx = len(iterable) - 1
    res = -1
    while down_idx <= top_idx:
        cur = (down_idx + top_idx) // 2
        if iterable[cur] >= target: #swap here
            res = cur
            down_idx = cur + 1
        else:
            top_idx = cur - 1
    return res
```

8.2 Binary Search For Smallest Possible Value

```
using namespace std;

bool valid(ull time, ull goal, vull &machines) {
  ull sum = 0;
  for (auto &m : machines)
    sum += time / m;

  return sum >= goal;
}

int main() {
  fast_io();

  ull n, t;
  cin >> n >> t;

  vull machines;
  while (n--) {
    ull tmp;
    cin >> tmp;
    machines.push_back(tmp);
  }

  ull boundary = t * (*max_element(machines.begin(), machines.end())) + 1;
  DEBUG(boundary);
  ull k = 0;
  for (ull b = boundary / 2; b >= 1; b /= 2) {
    DEBUG(valid(k + b, t, machines));
    while (!valid(k + b, t, machines))
      k += b;
  }

  cout << k + 1 << '\n';
}
```

8.3 Bitwise Operations

```
// Usefull bitwise functions
//

// __builtin_popcount(x): Counts the number of one's(set bits) in an
// integer(long/long long).
// __builtin_clz(x): Counts the leading number of zeros of the integer(long/long
// long)
// __builtin_ctz(x): Counts the trailing number of zeros of the
// integer(long/long long)
// __builtin_ffs(x): Index of the least significant bit + 1
// __lg(x): Index of most significant bit.
// Generate all subsets of size N
for (int mask = 0; mask < (1 << n); ++mask)
  for (int i = 0; i < n; ++i)
    if ((mask & (1 << i)))
      // inside subset.
      else
        // outside subset

        // Generate all combinations C(n, k). IDK how it works, but it does.
        int n,
        k;
int mask = (1 << k) - 1, r, c;
while (mask <= (1 << n) - (1 << (n - k))) {
  // code here
  c = mask & -mask, r = mask + c, mask = r | (((r ^ mask) >> 2) / c);
}
```

8.4 Fast Io

```
import sys

input = lambda: sys.stdin.readline().removesuffix('\n')
print = lambda s="", end="\n": sys.stdout.write(str(s)+end)
```

8.5 Inversion Counting

```cpp
#include <bits/stdc++.h>

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

struct FenwickTree {
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  FenwickTree(vector<int> &vec) {
    ft.assign(vec.size() + 1, 0);
    for (int i = 0; i < vec.size(); ++i)
      update(i + 1, vec[i]);
  }

  inline int ls_one(int x) { return x & (-x); }

  int query(int r) {
    int sum = 0;
    while (r) {
      sum += ft[r];
      r -= ls_one(r);
    }
    return sum;
  }

  int query(int l, int r) { return query(r) - query(l - 1); }

  void update(int i, int v) {
    while (i < ft.size()) {
      ft[i] += v;
      i += ls_one(i);
    }
  }

  // Finds smallest index i on FenwickTree such that query(1, i) >= rank.
  // I.e: smallest i for [1, i] >= k
  int select(long long k) { // O(log^2 m)
    int lo = 1, hi = ft.size() - 1;
    for (int i = 0; i < 30; ++i) {
      int mid = (lo + hi) / 2;
      (query(1, mid) < k) ? lo = mid : hi = mid;
    }
    return hi;
  }

  vector<int> ft;
};

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int n;
  while (cin >> n && n) {
    vector<int> seq(n);
    for (auto &v : seq)
      cin >> v;

    FenwickTree ft(n);

    int inv = 0;
    for (int i = 0; i < n; ++i) {
      inv += ft.query(seq[i] + 1, n);
      ft.update(seq[i], 1);
    }
    cout << (inv % 2 == 0 ? "Carlos\n" : "Marcelo\n");
  }
}
```

8.6 Inversion Counting

```python
"""
Conta inversões presentes em um array

3,2,4,5

3 e 2 representam uma inversão (2 está a frente de 3, mas é menor que ele)
"""


from fenwick_tree import FenwickTree
```

```
def normalize(iteratable) -> dict[any, int]:
    """
    Cria um dicionário mapeando cada elemento do array para um número no intervalo [1, len(iteratable)]
    """
    sorted_iteratable = sorted(iteratable)

    mp = {}
    num = 1

    for val in sorted_iteratable:
        if val not in mp:
            mp[val] = num
            num += 1
    return mp

def inversion_count(iteratable) -> int:
    """
    conta a quantidade total de inversões encontradas no array.
    """

    """
    A fenwick tree conta a frequência de elementos encontrados no array até o momento, permitindo
    a realização de queries para saber quantos valores já apareceram em um determinado intervalo de números.
    por exemplo:
        suponha que um loop itere sobre os valores 5 4 3 2 1.

        na terceira iteração do for loop (quando o valor for 3),
        a árvore indicará que foram encontrados dois valores no intervalo [3:5]

    isso permite encontrar inversões de forma rápida, uma vez que tudo que precisamos fazer para encontrar todas
    as inversões de um número n é descobrir quantos números maiores que ele aparecem antes dele.
    i.e: bast fazer uma query a fenwick tree no intervalo [n:len(iteratable)].
    """

    ft = FenwickTree(len(iteratable))
    mp = normalize(iteratable)
    inv = 0
    for val in iteratable:
        inv += ft.query(mp[val], len(iteratable))
        ft.update(mp[val], 1)
    return inv
```

8.7 Maximum Subarray Sum

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
  sum = max(array[k], sum + array[k]);
  best = max(best, sum);
}
cout << best << "\n";
```

8.8 Random

```
const int minRand = 1;
const int maxRand = 1e12;

random_device rd;
mt19937 gen(rd());
uniform_int_distribution<int> distribution(minRand, maxRand);

int someRand = distribution(gen);

// shuffling a vector
random_device rd;
mt19937 gen(rd());
shuffle(all(vec), gen);
```

8.9 Random Number Generation

```
#include <random>

using namespace std;

const int L = 1;
const int R = 1e9;

default_random_engine gen;

uniform_int_distribution<int> distribution(L, R);

int num() { return distribution(gen); }
```

8.10 Ternary Search

```
double ternary_search(double l, double r) {
  double eps = 1e-9; // set the error limit here
  while (r - l > eps) {
    double m1 = l + (r - l) / 3;
    double m2 = r - (r - l) / 3;
    double f1 = f(m1); // evaluates the function at m1
    double f2 = f(m2); // evaluates the function at m2
    if (f1 < f2)       // swap "<" for ">" for min instead of max
      l = m1;
    else
      r = m2;
  }
  return f(l); // return the maximum of f(x) in [l, r]
}
```

9 Zolutions/Classical Problems

9.1 Josephus Find Last Survivor

```
#include <bits/stdc++.h>

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

int backtrack(int n, int k) {
  return n == 1 ? 0 : (backtrack(n-1, k) + k) % n;
}

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int t, k;
  cin >> t >> k;

  cout << backtrack(t, k) << endl;

}
```

9.2 Meet In The Middle

```
/*
 *
 * Find all combinations C(m, n) where the sum of all elements equals t.
 * For a sufficiently small m, we can use Meet in the Middle.
 *
 * Usually, if you wanted to do this naively, you'd have to test each subset of
 * size N of the given vector, which would give you a complexity of O(2^m)
 *
 * However, if we divide the vector in two, find all subsets of size <= N and
 * them merge both results, we can achieve a complexity of O(2^(m/2))
 */

#include <bits/stdc++.h>

#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

int dp[2][31][10'000 * 30 + 2];

void gen_subsets(vector<int> &labels, int begin, int end, int n, int idx) {
  int limit = 1 << (end - begin);

  for (int mask = 0; mask < limit; ++mask) {
    int cur_sum = 0;
    int cnt = 0;
    for (int j = 0; j < end - begin; ++j)
      if ((mask & (1 << j)))
        cur_sum += labels[begin + j], cnt++;

    if (cnt <= n) {
      dp[idx][cnt][cur_sum]++;
    }
  }
}

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);
```

```
    int game;
    cin >> game;

    for (int g = 1; g <= game; ++g) {
      int m;
      cin >> m;

      vector<int> labels(m);
      for (auto &l : labels)
        cin >> l;

      int n, t;
      cin >> n >> t;

      if (!n && !t) {
        cout << "Game " << g << " -- 1 : 0\n";
        continue;
      } else if (!n) {
        cout << "Game " << g << " -- 0 : 1\n";
        continue;
      }

      int sum = accumulate(all(labels), 0);
      memset(dp, 0, sizeof(dp));

      int mid = labels.size() / 2;
      gen_subsets(labels, 0, mid, n, 0);
      gen_subsets(labels, mid, m, n, 1);

      unsigned int win = 0;
      for (int i = 0; i <= n; ++i)
        for (int j = 0; j <= t; ++j)
          if (dp[0][i][j] && dp[1][n - i][t - j])
            win += (dp[0][i][j] * dp[1][n - i][t - j]);

      double all_games = 1;
      for (int i = 1; i <= n; ++i)
        all_games = all_games * (m - n + i) / i;
      unsigned int all_g = (unsigned int)(all_games + 0.01);

      cout << "Game " << g << " -- " << win << " : " << all_g - win << '\n';
    }
  }
```

10 Zolutions/Cses/Introductory Problems

10.1 Chessboard-And-Queens

```
#include <algorithm>
#include <array>
#include <iostream>
#include <vector>

typedef std::vector<int> vi;
typedef std::vector<ll> vll;
typedef std::vector<ull> vull;

#define sz(x) (int(x.size()))
#define fast_io()                                                    \
  std::ios::sync_with_stdio(0);                                      \
  std::cin.tie(0);

/*
 * Your task is to place eight queens on a chessboard so that no two queens are
 * attacking each other. As an additional challenge, each square is either free
 * or reserved, and you can only place queens on the free squares. However, the
 * reserved squares do not prevent queens from attacking each other. How many
 * possible ways are there to place the queens?
 */

std::array<std::array<char, 8>, 8> arr;
// std::array<std::array<bool, 8>, 8> cols;
std::array<bool, 8> col;
std::array<bool, 15> diag, diag2;

int n = 0;

void solve(int k) {
  if (k == arr.size()) {
    n++;
    return;
  } else {
    // diag --> i+j
    // diag2 --> j-i+arr.size()-1
    int i = k;

    for (int j = 0; j < sz(arr); ++j) {
      if (arr[i][j] == '*' || col[j] || diag[i + j] ||
          diag2[j - i + sz(arr) - 1])
```

```
        continue;

      col[j] = diag[i + j] = diag2[j - i + sz(arr) - 1] = true;
      solve(k + 1);
      col[j] = diag[i + j] = diag2[j - i + sz(arr) - 1] = false;
    }
  }
}

int main() {
  fast_io();

  for (int i = 0; i < 8; ++i)
    for (int j = 0; j < 8; ++j)
      std::cin >> arr[i][j];

  solve(0);
  std::cout << n;
}
```

10.2 Coin-Piles

```
#include <cmath>
#include <iostream>
#define ll long long
#define ull unsigned long long

int main() {
  /*
  2x + y = a
  2y + x = b

   y = a - 2x --> a - 2*(2a - b)/3
   x = b + 4x - 2a --> x = (2a - b )/3


  */

  std::ios::sync_with_stdio(false);

  int t;
  std::cin >> t;

  while (t--) {
    ll a, b;
    std::cin >> a >> b;

    ll max = (a > b) ? a : b;
    ll min = (a > b) ? b : a;

    // ll x = (2*max-min)/3;
    // ll y = max - 2*(2*max-min)/3;

    if ((2 * max - min) % 3 == 0 && 2 * (2 * max - min) / 3 <= max)
      std::cout << "YES\n";
    else
      std::cout << "NO\n";
  }
}
```

10.3 Digit-Queries

```
#include <algorithm>
#include <array>
#include <cmath>
#include <iostream>
#include <tuple>
#include <vector>

typedef long long ll;
typedef uint64_t ull;
typedef std::vector<int> vi;
typedef std::vector<ll> vll;
typedef std::vector<ull> vull;

#define sz(x) (int(x.size()))
#define fast_io()                                                    \
  std::ios::sync_with_stdio(0);                                      \
  std::cin.tie(0);

std::tuple<ull, ull, ull> find_min_pos(ull n) {
  ull pos = 0, num = 1, nd = 1;
  while (true) {
    pos += (9ll * nd * (ll)pow(10, nd - 1));
    if (pos > n)
      break;
    nd++;
  }
```

```
    num = pow(10, nd - 1);
    return std::make_tuple(num, nd, pos - (9ll * nd * (ll)pow(10, nd - 1)));
  }

  int main() {
    fast_io();
    int q;
    std::cin >> q;

    while (q--) {
      ull k;
      std::cin >> k;
      k--;
      auto [num, nd, pos] = find_min_pos(k);

      ull diff = k - pos;
      ull mult = diff / nd; // integer division, it isnt redundant
      ull left_pos = pos + mult * nd;

      num = num + mult;

      std::string val = std::to_string(num);
      std::cout << val[k - left_pos] << '\n';
    }
  }
```

10.4 Tower-Of-Hanoi

```
  #include <cmath>
  #include <iostream>

  #define ll long long
  #define ull uint64_t

  /*
  The Tower of Hanoi game consists of three stacks (left, middle and right) and n
  round disks of different sizes. Initially, the left stack has all the disks, in
  increasing order of size from top to bottom. The goal is to move all the disks
  to the right stack using the middle stack. On each move you can move the
  uppermost disk from a stack to another stack. In addition, it is not allowed to
  place a larger disk on a smaller disk. Your task is to find a solution that
  minimizes the number of moves.
  */

  void print_move(int start, int end) {
    std::cout << start << ' ' << end << '\n';
  }

  void mv(int n, int start, int end) {
    if (n == 1) {
      print_move(start, end);
      return;
    }

    int mid = 6 - start - end;
    mv(n - 1, start, mid);
    mv(1, start, end);
    mv(n - 1, mid, end);
  }

  int main() {
    std::ios::sync_with_stdio(0);

    int n;
    std::cin >> n;
    std::cout << (ll)pow(2, n) - 1 << '\n';
    mv(n, 1, 3);
  }
```

10.5 Trailing-Zeros

```
  #include <cmath>
  #include <iostream>
  #define ll long long
  #define ull unsigned long long
  /*
   * Your task is to calculate the number of trailing zeros in the factorial n!.
   * For example, 20!=2432902008176640000 and it has 4 trailing zeros.
   */

  int main() {

    std::ios::sync_with_stdio(false);

    int n;
    std::cin >> n;
    int ans = 0;
    int num = 1;

    while (num < n) {
      num *= 5;
      ans += n / num;
    }

    std::cout << ans;
  }
```

11 Zolutions/Cses/Sorting And Searching

11.1 Array-Division

```
  /*
  *
  * You are given an array containing n positive integers.
  Your task is to divide the array into k subarrays so that the maximum sum in a
  subarray is as small as possible.
  */

  bool find_sub_arrays(vector<int> &vec, ll max_sum, int x) {
    ll curr = 0;
    int n = 1;

    int i = 0;
    while (i < vec.size()) {
      if (vec[i] > max_sum)
        return false;

      if (curr + vec[i] <= max_sum) {
        curr += vec[i];
      } else {
        n++;
        curr = vec[i];
      }
      ++i;
    }

    return n <= x;
  }

  int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    int n, x;
    cin >> n >> x;
    vector<int> vec(n);

    for (auto &v : vec)
      cin >> v;

    ll l = *max_element(vec.begin(), vec.end());
    ll r = accumulate(vec.begin(), vec.end(), 0ll);

    ll ans = 0;
    while (l < r + 1) {
      ll mid = (l + r) / 2;

      bool possible = find_sub_arrays(vec, mid, x);

      if (possible)
        ans = mid;
      // cout << "l=" << l << ",r=" << r << ",mid=" << mid << ",possible=" <<
      // possible << '\n';
      if (possible)
        r = mid - 1;
      else
        l = mid + 1;
    }

    cout << ans << '\n';
  }
```

11.2 Factory-Machines

```
  using namespace std;

  /*
   *
   * vull --> vector<unsigned long long>
   *
  *A factory has n machines which can be used to make products. Your goal is to
  make a total of t products. *For each machine, you know the number of seconds it
```

```
needs to make a single product. The machines can work simultaneously, and you
can freely decide their schedule. *What is the shortest time needed to make t
products?
*
*The first input line has two integers n and t: the number of machines and
products. The next line has n integers k_1,k_2,\dots,k_n: the time needed to
make a product using each machine.
*/

bool valid(ull time, ull goal, vull &machines) {
  ull sum = 0;
  for (auto &m : machines)
    sum += time / m;

  return sum >= goal;
}

int main() {
  fast_io();

  ull n, t;
  cin >> n >> t;

  vull machines;
  while (n--) {
    ull tmp;
    cin >> tmp;
    machines.push_back(tmp);
  }

  ull boundary = t * (*max_element(machines.begin(), machines.end())) + 1;
  DEBUG(boundary);
  ull k = 0;
  for (ull b = boundary / 2; b >= 1; b /= 2) {
    DEBUG(valid(k + b, t, machines));
    while (!valid(k + b, t, machines))
      k += b;
  }

  cout << k + 1 << '\n';
}
```

11.3 Josephus-Problem-Ii

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

#define sz(x) (int(x.size()))
#define fast_io()                                                    \
  {                                                                  \
    ios::sync_with_stdio(0);                                         \
    cin.tie(NULL);                                                   \
  }

typedef long long ll;
typedef unsigned long long ull;
typedef std::vector<int> vi;
typedef std::vector<ll> vll;
typedef std::vector<ull> vull;
typedef std::pair<int, int> pi;
typedef std::pair<ll, ll> pll;
typedef std::pair<ull, ull> pull;
typedef std::vector<pi> vii;
using namespace std;
using namespace __gnu_pbds;
/*
Consider a game where there are n children (numbered 1,2,\dots,n) in a circle.
During the game, repeatedly k children are skipped and one child is removed from
the circle. In which order will the children be removed?
*/
int main() {
  // fast_io();

  int n, k;
  cin >> n >> k;

  tree<int, null_type, less<int>, rb_tree_tag,
       tree_order_statistics_node_update>
      tr;
  for (int i = 0; i < n; ++i)
    tr.insert(i + 1);

  auto it = tr.find_by_order(k % tr.size());
  int i = k % tr.size();

  while (true) {
    cout << *it << ' ';
    it = tr.erase(it);
    if (tr.empty())
      break;
    int num = k % tr.size();
    if (i + num >= tr.size()) {
      it = tr.begin();
      num = num - (tr.size() - i);
    } else {
      num += i;
    }

    it = tr.find_by_order(num);
    i = (i + (k % tr.size())) % tr.size();
  }
}
```

11.4 Josephus-Problem

```
#define sz(x) (int(x.size()))
#define fast_io()                                                            \
  {                                                                          \
    ios::sync_with_stdio(0);                                                 \
    cin.tie(NULL);                                                           \
  }

typedef long long ll;
typedef unsigned long long ull;
typedef std::vector<int> vi;
typedef std::vector<ll> vll;
typedef std::vector<ull> vull;
typedef std::pair<int, int> pi;
typedef std::pair<ll, ll> pll;
typedef std::pair<ull, ull> pull;
typedef std::vector<pi> vii;
using namespace std;

constexpr const int sz = 2e5 + 5;

/*
*
Consider a game where there are n children (numbered 1,2,\dots,n) in a circle.
During the game, every other child is removed from the circle until there are no
children left. In which order will the children be removed?
*/

int main() {
  fast_io();

  int n;
  int arr[2][sz];

  cin >> n;

  for (int i = 0; i < n; ++i)
    arr[0][i] = i + 1;

  arr[0][n] = 0;

  int count = 0;
  int curr = 0, other = 1;
  bool even = n % 2 == 0;

  while (count != n) {
    int i = even ? 0 : 1;
    int round = even ? 0 : 1;
    int i2 = 0;
    while (arr[curr][i] != 0) {
      if (round) {
        round = 0;
        count++;
        cout << arr[curr][i] << ' ';
      }

      else {
        round = 1;
        arr[other][i2] = arr[curr][i];
        i2++;
      }

      i++;
    }

    if (!even) {
      cout << arr[curr][0] << ' ';
      count++;
    }

    arr[other][i2] = 0;
    // cout << "\nFOOO\n";
    // for ( int i3 = 0; i3 < i2; ++i3 )
    //     cout << arr[other][i3] << ' ';
    // cout << "\nFOOO\n";
    curr = (curr == 1) ? 0 : 1;
    other = (other == 1) ? 0 : 1;

    even = i2 % 2 == 0;
  }
}
```

11.5 Maximum-Subarray-Sum-Ii

```
/*
 *
 *Given an array of n integers, your task is to find the maximum sum of values
 *in a contiguous subarray with length between a and b.
 */
int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  ll n, a, b;
  cin >> n >> a >> b;

  vector<ll> prefix(n);
  cin >> prefix[0];

  for (ll i = 1; i < n; ++i) {
    ll tmp;
    cin >> tmp;
    prefix[i] = prefix[i - 1] + tmp;
  }

  multiset<ll> window;
  unordered_map<ll, queue<ll>> pref_to_idx;

  ll ans = INT64_MIN;
  ll last_insert = 0;
  for (ll i = a - 1; i < n; ++i) {
    if (i - last_insert > b) {
      window.erase(window.lower_bound(prefix[last_insert++]));
    }

    if (!window.size() || (i < b && *window.begin() == prefix[0]))
      ans = max(ans, prefix[i]);
    else {
      ans = max(ans, prefix[i] - *window.begin());
    }

    if (a == 1)
      ans = max(ans, (i == 0 ? prefix[0] : prefix[i] - prefix[i - 1]));

    window.insert(prefix[i - a + 1]);
  }

  cout << ans << '\n';
}
```

11.6 Movie-Festival

```
#include <algorithm>
#include <bits/stdc++.h>
#include <chrono>
#include <random>
#include <set>
#include <vector>
#define sz(x) (int(x.size()))
#define fast_io()                                                               \
  {                                                                             \
    ios::sync_with_stdio(0);                                                    \
    cin.tie(NULL);                                                              \
  }

typedef long long ll;
typedef uint64_t ull;
typedef std::vector<int> vi;
typedef std::vector<ll> vll;
typedef std::vector<ull> vull;
typedef std::pair<int, int> pi;
typedef std::pair<ll, ll> pll;
typedef std::pair<ull, ull> pull;
typedef std::vector<pi> vii;
using namespace std;

/*
 *In a movie festival n movies will be shown. You know the starting and ending
 *time of each movie. What is the maximum number of movies you can watch
 *entirely?
 */
```

```
int main() {
  fast_io();

  int n;
  cin >> n;

  vii vec;
  while (n--) {
    int b, e;
    cin >> b >> e;

    vec.emplace_back(b, e);
  }

  sort(vec.begin(), vec.end(),
       [](const pi &a, const pi &b) { return a.second < b.second; });

  int curr_ed = 0;
  int ans = 0;

  for (auto &p : vec) {
    if (curr_ed <= p.first) {
      ans++;
      curr_ed = p.second;
    }
  }
  cout << ans << '\n';
}
```

11.7 Nearest-Smaller-Values

```
/*
  *
  *
Given an array of n integers, your task is to find for each array position the
nearest position to its left having a smaller value.
  */
using namespace std;

int main() {
  int n;
  scanf("%d ", &n);
  stack<pii> st;

  for (int i = 0; i < n; ++i) {
    int num;
    scanf("%d ", &num);

    while (!st.empty() && st.top().first >= num)
      st.pop();

    if (st.empty())
      printf("%d ", 0);
    else
      printf("%d ", st.top().second);

    st.push({num, i + 1});
  }
}
```

11.8 Nested-Ranges-Check

```
#define sz(x) (int(x.size()))
#define fast_io()                                                        \
  {                                                                      \
    ios::sync_with_stdio(0);                                             \
    cin.tie(NULL);                                                       \
  }
#define mult_vec(name, T, n, m)                                          \
  std::vector<std::vector<T>> name(n, std::vector<T>(m))

using namespace std;

/*
 *
 *Given n ranges, your task is to determine for each range if it contains
 *some other range and if some other range contains it. Range [a,b] contains
 *range [c,d] if a \le c and d \le b.

The first input line has an integer n: the number of ranges.
After this, there are n lines that describe the ranges. Each line has two
integers x and y: the range is [x,y]. You may assume that no range appears more
than once in the input.

 */

struct Fenwick {
  Fenwick(int n) : vec{vi(n + 1, 0)} {}

  void add(int idx, int val) {
    while (idx < vec.size()) {
      vec[idx] += val;
      idx += idx & -idx;
    }
  }

  int query(int l, int r) { return query(r) - query(l - 1); }

  int query(int r) {
    int sum = 0;
    int i = r;
    while (i > 0) {
      sum += vec[i];
      i -= i & -i;
    }

    return sum;
  }

  vi vec;
};

struct Range {
  Range(int l, int r, int i) : left{l}, right{r}, i{i} {}

  int left;
  int right;
  int i;
};

unordered_map<int, int> normalize(vector<Range> vec) {
  unordered_map<int, int> mp;
  int count = 1;

  sort(vec.begin(), vec.end(), [](Range &a, Range &b) {
    return (a.right == b.right) ? a.left < b.left : a.right < b.right;
  });

  for (auto &n : vec) {
    if (!mp.contains(n.right)) {
      mp[n.right] = count;
      count++;
    }
  }

  return mp;
}

vpi count_inv(vector<Range> &vec) {
  Fenwick fwl(vec.size()), fwr(vec.size());
  vpi ret(vec.size());

  auto norm = normalize(vec);

  for (auto &n : vec) {
    ret[n.i].second += fwl.query(norm[n.right], vec.size());
    fwl.add(norm[n.right], 1);
  }
  for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
    auto &n = *it;
    ret[n.i].first += fwr.query(1, norm[n.right]);
    fwr.add(norm[n.right], 1);
  }

  return ret;
}

int main() {
  fast_io();

  int n;
  cin >> n;

  vector<Range> vec;

  for (int i = 0; i < n; ++i) {
    int l, r;
    cin >> l >> r;
    vec.emplace_back(l, r, i);
  }

  sort(vec.begin(), vec.end(), [](Range &a, Range &b) {
    return (a.left == b.left) ? a.right > b.right : a.left < b.left;
  });

  auto v = count_inv(vec);

  for (auto &p : v)
    cout << (p.first > 0) << ' ';
  cout << '\n';

  for (auto &p : v)
    cout << (p.second > 0) << ' ';
}
```

11.9 Nested-Ranges-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. Range [a,b] contains range
 *[c,d] if a \le c and d \le b.
 *
 *
The first input line has an integer n: the number of ranges.
After this, there are n lines that describe the ranges. Each line has two
integers x and y: the range is [x,y]. You may assume that no range appears more
than once in the input.
 */

struct Fenwick {
  Fenwick(int n) : vec{vi(n + 1, 0)} {}

  void add(int idx, int val) {
    while (idx < vec.size()) {
      vec[idx] += val;
      idx += idx & -idx;
    }
  }

  int query(int l, int r) { return query(r) - query(l - 1); }

  int query(int r) {
    int sum = 0;
    int i = r;
    while (i > 0) {
      sum += vec[i];
      i -= i & -i;
    }

    return sum;
  }

  vi vec;
};

struct Range {
  Range(int l, int r, int i) : left{l}, right{r}, i{i} {}

  int left;
  int right;
  int i;
};

unordered_map<int, int> normalize(vector<Range> vec) {
  unordered_map<int, int> mp;
  int count = 1;

  sort(vec.begin(), vec.end(), [](Range &a, Range &b) {
    return (a.right == b.right) ? a.left < b.left : a.right < b.right;
  });

  for (auto &n : vec) {
    if (!mp.contains(n.right)) {
      mp[n.right] = count;
      count++;
    }
  }

  return mp;
}

vpi count_inv(vector<Range> &vec) {
  Fenwick fwl(vec.size()), fwr(vec.size());
  vpi ret(vec.size());

  auto norm = normalize(vec);

  for (auto &n : vec) {
    ret[n.i].second += fwl.query(norm[n.right], vec.size());
    fwl.add(norm[n.right], 1);
  }
  for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
    auto &n = *it;
    ret[n.i].first += fwr.query(1, norm[n.right]);
    fwr.add(norm[n.right], 1);
  }

  return ret;
}

int main() {
  fast_io();

  int n;
  cin >> n;

  vector<Range> vec;
```

```
    for (int i = 0; i < n; ++i) {
      int l, r;
      cin >> l >> r;
      vec.emplace_back(l, r, i);
    }

    sort(vec.begin(), vec.end(), [](Range &a, Range &b) {
      return (a.left == b.left) ? a.right > b.right : a.left < b.left;
    });

    auto v = count_inv(vec);

    for (auto &p : v)
      cout << (p.first) << ' ';
    cout << '\n';

    for (auto &p : v)
      cout << (p.second) << ' ';
  }
```

11.10 Sliding-Window-Cost

```
  /*
    *You are given an array of n integers. Your task is to calculate for each
  window of k elements, from left to right, the minimum total cost of making all
  elements equal. You can increase or decrease each element with cost x where x is
  the difference between the new and the original value. The total cost is the sum
  of such costs.
    */

  using ll = long long;

  void debug(ll i, multiset<ll> &lo, multiset<ll> &hi) {
    cout << "I: " << i << '\n';
    cout << "LO: ";
    for (auto &v : lo)
      cout << v << ' ';
    cout << "\nHI: ";
    for (auto &v : hi)
      cout << v << ' ';

    cout << "\nMEDIAN: " << *lo.rbegin() << "\n\n";
  }

  int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    ll n, k;
    cin >> n >> k;

    vector<ll> vec(n);
    for (auto &v : vec)
      cin >> v;

    multiset<ll> lo, hi;
    ll l = 0, r = 0;
    ll median_left = k % 2 == 0 ? k / 2 - 1 : k / 2;
    ll median_right = k / 2;
    for (ll i = 0; i < k; ++i) {
      lo.insert(vec[i]);
      l += vec[i];
    }

    for (ll i = 0; i < k / 2; ++i) {
      ll tmp = *lo.rbegin();
      l -= tmp;
      r += tmp;
      lo.erase(prev(lo.end()));
      hi.insert(tmp);
    }

    ll median = *lo.rbegin();
    l -= median;
    if (k == 1)
      cout << "0 ";
    else
      cout << abs(r - median * median_right) + abs(median * median_left - l)
           << ' ';
    l += median;

    for (ll i = k; i < n; ++i) {
      ll tmp = vec[i];

      if (k == 1) {
        cout << 0 << ' ';
        continue;
      }
      if (tmp > median) {
        hi.insert(tmp);
        r += tmp;
      } else {
        lo.insert(tmp);
        l += tmp;
      }

      ll to_remove = vec[i - k];
      if (to_remove > median) {
        hi.erase(hi.find(to_remove));
        r -= to_remove;
      } else {
        lo.erase(lo.find(to_remove));
        l -= to_remove;
      }

      if (hi.size() > lo.size()) {
        ll tmp = *hi.begin();
        hi.erase(hi.find(*hi.begin()));
        lo.insert(tmp);
        r -= tmp;
        l += tmp;
      } else if (lo.size() > hi.size() + 1) {
        ll tmp = *lo.rbegin();
        lo.erase(lo.find(*prev(lo.end())));
        hi.insert(tmp);
        l -= tmp;
        r += tmp;
      }

      median = *lo.rbegin();
      l -= median;
      cout << abs(r - median * median_right) + abs(median * median_left - l)
           << ' ';
      l += median;
    }
    cout << '\n';
  }
```

11.11 Sliding-Window-Median

```
/**
  *
  *
  *You are given an array of n integers. Your task is to calculate the median of
each window of k elements, from left to right.
The median is the middle element
when the elements are sorted. If the number of elements is even, there are two
possible medians and we assume that the median is the smaller of them.
  */

using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less_equal<int>, rb_tree_tag,
             tree_order_statistics_node_update>
    ordered_tree;

int get_median(ordered_tree &t, int k) {
  return t.size() % 2 == 0 ? *t.find_by_order(k / 2 - 1)
                           : *t.find_by_order(k / 2);
}

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  int n, k;
  cin >> n >> k;

  vector<int> vec(n);
  for (auto &v : vec)
    cin >> v;

  ordered_tree window;
  queue<int> to_remove;

  for (int i = 0; i < k; ++i) {
    window.insert(vec[i]);
    to_remove.push(vec[i]);
  }

  cout << get_median(window, k) << ' ';
  window.erase(window.upper_bound(to_remove.front()));
  to_remove.pop();

  for (int i = k; i < n; ++i) {
    window.insert(vec[i]);
    to_remove.push(vec[i]);
    cout << get_median(window, k) << ' ';
    window.erase(window.upper_bound(to_remove.front()));
    to_remove.pop();
  }

  cout << '\n';
}
```

11.12 Subarray-Sums-Ii

```
/*
 *Given an array of n integers, your task is to count the number of subarrays
 *having sum x.
 */
int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  ll n, x;
  cin >> n >> x;
  vector<ll> vec(n);
  for (auto &val : vec)
    cin >> val;

  map<ll, ll> mp_sum;
  ll curr = 0;

  ull ans = 0;
  int i = 0;
  for (auto &v : vec) {
    if (v + curr == x)
      ans++;
    if (mp_sum.count(v + curr - x))
      ans += mp_sum[v + curr - x];
    mp_sum[v + curr]++;
    curr += v;
  }

  cout << ans << '\n';
}
```

12 Zolutions/Interesting Problems

12.1 Bustour

```
/**
Given an HQ, some hotels H and an attraction A, find shortest path that follows this:

HQ -> First len(H)/2 hotels -> remaining len(H) - len(H)/2 hotels -> A -> First len(H)/2 hotels -> remaining len
    (H) - len(H)/2 hotels -> HQ

input: N -> len(H) + 2 (includes HQ and A)
       M -> number of edges.

You can pass in front of a hotel without visiting it, so do a floyd-warshall to get all shortest paths from U to
     V.

Use TSP dp solution.
 */


#include <algorithm>
#include <bits/stdc++.h>

#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;


int N, M;
int d[21][21];

int dp_first_half[1 << 18][18], dp_second_half[1 << 18][18];

void tsp() {
  int limit = 1 << (N-2);
  for (int mask = 0; mask < limit; ++mask) {
    for (int i = 0; i < N - 2; ++i) {
      if (!(mask & (1 << i)))
        continue;
      for (int j = 0; j < N - 2; ++j) {
        if (!(mask & (1 << j)) || i == j)
          continue;

        dp_first_half[mask][j] = min(
          dp_first_half[mask][j],
          dp_first_half[mask ^ (1 << j)][i] + d[i+1][j+1]
        );

        dp_second_half[mask][j] = min(
          dp_second_half[mask][j],
          dp_second_half[mask ^ (1 << j)][i] + d[i+1][j+1]
        );
      }
    }
  }
}

int solve() {
  if (N == 3) {
    return d[0][1] + d[1][2] + d[2][1] + d[1][0];
  }
  tsp();

  int n = N - 2;
  int k = n / 2;

  int mask = (1 << k) - 1, r, c;
  int ans = 1e9;
  while (mask <= (1 << n) - (1 << (n - k))) {
    int first_half = 1e9;
    int not_mask = (~mask) & ((1 << (N - 2)) - 1);
    for (int i = 0; i < N - 2; ++i) {
      if (!(mask & (1 << i)))
        continue;

      for (int j = 0; j < N - 2; ++j) {
        if (!(mask & (1 << j)))
          continue;

        ans = min(
          ans,
          dp_first_half[mask][i] + dp_second_half[not_mask | (1 << i)][i] + dp_second_half[mask][j] +
              dp_first_half[not_mask | (1 << j)][j]
        );
      }
    }

    c = mask & -mask, r = mask + c, mask = r | (((r ^ mask) >> 2) / c);
  }
  return ans;
}

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int t = 1;
  while (cin >> N >> M) {
    memset(d, 63, sizeof(d));
    memset(dp_first_half, 63, sizeof(dp_first_half));
    memset(dp_second_half, 63, sizeof(dp_second_half));


    for (int i = 0; i < N; ++i)
      d[i][i] = 0;
    while (M--) {
      int u, v, w;
      cin >> u >> v >> w;

      d[u][v] = d[v][u] = w;
    }


    for (int k = 0; k < N; ++k)
      for (int u = 0; u < N; ++u)
        for (int v = 0; v < N; ++v)
          d[u][v] = min(d[u][v], d[u][k] + d[k][v]);
    for (int i = 0; i < N-2; ++i)
      dp_first_half[1 << i][i] = d[0][i+1], dp_second_half[0][i] = dp_second_half[1 << i][
          i] = d[i+1][N-1];

    cout << "Case " << t++ << ": " << solve() << '\n';
  }
}
```

12.2 Intercept

```
  /**
  Find cutpoints/bridges in a DAG created by dijkstra's algorithm.
```

```
As the DAG always points from S to T, we can just treated it as a undirected graph and use the standard
    algorithm.
*/

#include <bits/stdc++.h>

#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

#define int long long

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

void dfs(int u, vector<vector<int>>& dag, vector<char>& visited) {
  visited[u] = true;
  for (auto& v : dag[u]) {
    if (visited[v])
      continue;
    dfs(v, dag, visited);
  }
}

void find_cutpoints(int u, int p, vector<char>& tvisited, vector<vector<int>>& adj, vector<int>& tin, vector<int
    >& low, vector<char>& visited, vector<bool>& cutpoint,  int& cnt) {
  tin[u] = low[u] = cnt++;
  visited[u] = true;

  for (auto& v : adj[u]) {
    if (v == p || !tvisited[v])
      continue;
    if (visited[v])
      low[u] = min(low[u], tin[v]);
    else {
      find_cutpoints(v, u, tvisited,adj, tin, low, visited, cutpoint, cnt);

      if (low[v] >= tin[u])
        cutpoint[u] = true;
      low[u] = min(low[u], low[v]);
    }
  }
}

tuple<vector<vector<int>>, vector<vector<int>>, vector<int>, vector<int>> dijkstra(vector<vector<pii>>& adj, int
     s, int t) {
  vector<int> d(adj.size(), 1e18);


  using iii = tuple<int, int, int>;
  vector<vector<int>> tdag(adj.size());
  vector<vector<int>> dag(adj.size());
  vector<int> tdegree(adj.size(), 0);
  priority_queue<iii, vector<iii>, greater<iii>> pq;

  pq.push({0, -1, s});
  d[s] = 0;
  vector<char> visited(adj.size(), 0);
  visited[s] = true;

  while (!pq.empty()) {
    auto[dd, parent, u] = pq.top(); pq.pop();

    if (dd != d[u])
      continue;

    if (parent != -1) {
      dag[parent].push_back(u);
      dag[u].push_back(parent);
      tdag[u].push_back(parent);
      tdegree[parent]++;
    }

    if (parent != -1 && visited[u])
      continue;
    visited[u] = true;

    for (auto&[w, v] : adj[u]) {
      int nd = d[u] + w;
      if (nd > d[v])
        continue;
      d[v] = nd, pq.push({nd, u, v});
    }
  }

  return {dag, tdag, tdegree, d};
}

signed main() {
  ios::sync_with_stdio(0);
```

```
    cin.tie(0);

    int n, m;
    cin >> n >> m;

    vector<vector<pii>> adj(n);
    while (m--) {
      int u, v, w;
      cin >> u >> v >> w;
      adj[u].push_back({w, v});
    }

    int s, t;
    cin >> s >> t;
    auto result = dijkstra(adj, s, t);

    auto&[dag, tdag, tdegree, d] = result;

    vector<char> tvisited(n, 0);
    dfs(t, tdag, tvisited);

    vector<int> tin(dag.size()), low(dag.size());
    vector<bool> cutpoint(dag.size(), 0);

    vector<char> visited(dag.size(), 0);
    int cnt = 0;
    find_cutpoints(s, -1, tvisited, dag, tin, low, visited, cutpoint, cnt);

    vector<int> ans;
    ans.push_back(s);
    if (s != t)
      ans.push_back(t);
    for (int i = 0; i < n; ++i)
      if (cutpoint[i] && i != s && i != t)
        ans.push_back(i);
    sort(all(ans));
    for (int i = 0; i < ans.size(); ++i)
      cout << ans[i] << (i == ans.size() - 1 ? "\n" : " ");
  }
```

12.3 Lagrange Interpolation

```
/*
About Lagrange interpolation:
You are given k+1 points. You want the coefficients of the kth degree polynomial
that goes through them. You can do that for arbitrary (x, y) in O(k^2) time.
This is explained as follows: Suppose the said polynomial is P(x) = l_1*f(x1) +
l_2*f(x2) + ... + l_k+1*f(x_(k+1)) If you compute P(x1), you'd like the
following to happen: l_1 = 1 l_2 = 0 l_3 = 0
...
l_(k+1) = 0
In general, for P(xi), you want l_i to be 1 if x's index is i and 0 if its not.
Those l_i's are actually polynomials, and their shape is the following:
l_i = (x-x1) * (x-x2) * (x-x3) * ... * (x-x_(i-1)) * (x-x_(i+1)) * ... *
(x-x_(k+1)) DIVIDED BY (x_i-x1) * (x_i-x2) * (xi-x3) * ... * (xi-x_(i-1)) *
(xi-x_(i+1)) * ... * (xi-x_(k+1)) The numerator has everyone but (x-xi), and the
denominator has everyone but (xi-xi). It is pretty clear that if you throw xi
onto l_i it will be 1, and 0 if you throw any of the other (k+1-1) points.

So, it means that P is already the polynomial you were searching for. Evaluating
it takes O(k^2), unless there is any property of the numbers you are working
with such that you can transition from l_i to l_(i+1) in sublinear time...
*/

#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define detachingFromC                                                    \
  ios::sync_with_stdio(0);                                                \
  cin.tie(0);
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

const int MOD = 1e9 + 7;

/*
1) Compute k+2 points to define the k+1th degree polynomial (the sum itself)
2) Compute the polynomial l1(n) in O(k) time
3) Transition from l_i to l_(i+1) in O(1) time for all i and evaluate the
polynomial
*/

tuple<int, int, int> extendedGcd(int a, int b) {
  if (b == 0)
```

```
    return make_tuple(a, 1, 0);
  auto [q, w, e] = extendedGcd(b, a % b);
  return make_tuple(q, e, w - e * (a / b));
}

int multiplicativeInverse(int n, int mod) {
  // (n)x + (mod)y = 1 (aka their difference is 1)
  n = (n % mod + mod) % mod;
  auto [g, x, y] = extendedGcd(n, mod);
  return (x % mod + mod) % mod;
}

int fastPow(int base, int e) {
  if (e == 0)
    return 1;
  if (e == 1)
    return base;
  int h = fastPow(base, e / 2);
  if (e % 2 == 1)
    return h * h % MOD * base % MOD;
  return h * h % MOD;
}

signed main() {
  detachingFromC;

  int n, k;
  cin >> n >> k;

  // 1)
  vector<int> y(k + 3, 0);
  y[1] = 1;
  for (int xi = 2; xi <= k + 2; xi++)
    y[xi] = (y[xi - 1] + fastPow(xi, k)) % MOD;

  if (n <= k + 2) {
    cout << y[n] << endl;
    return 0;
  }

  // 2)
  int num = 1;
  int den = 1;
  for (int xi = 2; xi <= k + 2; xi++) {
    num = num * (n - xi) % MOD;
    den = (den * (1 - xi) % MOD + MOD) % MOD;
  }

  // 3)
  int ans = num * multiplicativeInverse(den, MOD) % MOD * y[1] % MOD;
  for (int xi = 2; xi <= k + 2; xi++) {
    // num: take out factor (x-x_i) and bring (x-x_(i-1))
    num = num * multiplicativeInverse(n - xi, MOD) % MOD;
    num = num * (n - (xi - 1)) % MOD;
    // den: take out factor (x_(i-1) - x_last) and bring (xi - x1)
    den = den * multiplicativeInverse(xi - 1 - (k + 2), MOD) %
          MOD; // k+2 == x_last
    den = den * (xi - 1) % MOD;
    // increment answer by num/den * yi
    ans += num * multiplicativeInverse(den, MOD) % MOD * y[xi] % MOD;
    ans %= MOD;
  }

  cout << ans << endl;
  return 0;
}
```

12.4 Longest Common Subsequence Given Two Permutations

```
#include <bits/stdc++.h>

#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

struct FenwickTree {
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  FenwickTree(vector<int> &vec) {
    ft.assign(vec.size() + 1, 0);
    for (int i = 0; i < vec.size(); ++i)
      update(i + 1, vec[i]);
  }

  inline int ls_one(int x) { return x & (-x); }
  int query(int r) {
    int ret = 0;
    while (r) {
      ret = max(ret, ft[r]);
      r -= ls_one(r);
    }
    return ret;
  }
  void update(int i, int v) {
    while (i < ft.size()) {
      ft[i] = max(ft[i], v);
      i += ls_one(i);
    }
  }

  vector<int> ft;
};

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int n;
  cin >> n;

  vector<int> va(n), vb(n);
  vector<pii> pairs(n);
  for (int i = 0; i < n; ++i)
    cin >> va[i], pairs[va[i]].first = i;
  for (int i = 0; i < n; ++i)
    cin >> vb[i], pairs[vb[i]].second = i;

  int ans = 0;
  FenwickTree ft(n);
  for (int i = 0; i < n; ++i) {
    int num = va[i];
    int ib = pairs[num].second;

    int cur =  ft.query(ib+1);
    ft.update(ib+1, cur + 1);

    ans = max(cur + 1, ans);
  }

  cout << ans << '\n';
}
```

12.5 Purplerain

```
/*
 * Given a string composed of R's and B's, find subarray where the difference
 * between the number of R's and the number of B's is maximized.
 *
 * How to solve:
 *   - Create two arrays: one where every R has value of 1 and every B has value
 * of -1 and another where the opposite occurs.
 *   - Use Kadane's algorithm to find greates subarray on each array and then
 * print the best result.
 */

#include <bits/stdc++.h>

#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

void kadane(int &ans, int &ans_l, int &ans_r, vector<int> &vec) {
  int cur = 0, cur_l = 0;

  for (int i = 0; i < vec.size(); ++i) {
    int num = vec[i];

    if (cur < 0)
      cur = num, cur_l = i;
    else
      cur += num;

    if (cur > ans ||
        (cur == ans && (cur_l < ans_l || (cur_l == ans_l && i < ans_r)))) {
      ans = cur;
      ans_l = cur_l;
      ans_r = i;
    }
  }
}
```

```
int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int ans = 0, ans_l = 0, ans_r = 0;

  string s;
  cin >> s;

  vector<int> vec(s.size());
  for (int i = 0; i < vec.size(); ++i)
    vec[i] = s[i] == 'R' ? 1 : -1;
  kadane(ans, ans_l, ans_r, vec);
  for (int i = 0; i < vec.size(); ++i)
    vec[i] = s[i] != 'R' ? 1 : -1;
  kadane(ans, ans_l, ans_r, vec);

  cout << ans_l + 1 << ' ' << ans_r + 1 << '\n';
}
```

13 Zolutions/Latam 2024

13.1 Beating The Record

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
A guy wants to beat a world record of a game. He has 2 strategies per level, each one having a chance of
succeeding and failing, and a time to go through the level on each circumstance. The most interesting thing
is that he can restart the whole thing whenever he wants. Compute the expected play time until the record
is beaten (circular dp dependency of the first level with itself because of restarting).

Idea:
Say a state is determined by its level 'i' and the 'tt' seconds that you have been playing on this run.
Then, you can compute the expected remaining time of gaming simply with states at level 'i+1' and
time tt1 > tt, except for the possibility of restarting. But since restarting depends on the levels
themselves, you cannot solve for the expected total time. The deal is that this starting expected time is
binary searchable, because:

1) The expected time you guessed must be equal to the expected time you get by computing it from
its children.
2) If you guessed too low, then, it won't be worth it trying and every level will just restart. Then, the
computed expected will be roughly the guess itself (it should be equal, but since we compute it S steps
further, then, the computed value will be probably the guess + S).
3) If you guessed too high, then, the walk itself will be better than the expected, which doesn't make sense.
In that case, you continue the search below (high = mid).

Therefore, the higher the guess, the smaller the computed expected, and you need to equalize them.
Time complexity: O(B2^n), where B = binary search steps.
"Improvable" to O(B*(MAXT)*n) if n was larger (because of dp).

Details about expected computation:
In any level, you can choose between strat1 and strat2, and, for each strat, you can either continue or
restart, depending on whether it failed or not. That's why you will always min(restart, smt) everywhere
(because you can restart anywhere).
*/

using ftype = long double;
int N, S, T;
ftype restart;
ftype G[4][2];
ftype B[4][2];
ftype P[4][2];

ftype expected(int i, int tt) {
    if (tt >= T) return restart;
    if (i == N) return 0;

    ftype ans = restart;
    for (int strat = 0; strat < 2; strat++) {
        ftype good =    P[i][strat] * min(restart, G[i][strat] + expected(i+1, tt+G[i][strat]));
        ftype bad = (1-P[i][strat]) * min(restart, B[i][strat] + expected(i+1, tt+B[i][strat]));
        ans = min(ans, good+bad);
    }

    return ans;
}
```

```cpp
signed main() {
    cin.tie(0)->sync_with_stdio(0);

    cin >> N >> T >> S;

    for (int i = 0; i < N; i++) {
        for (int s = 0; s < 2; s++) {
            cin >> P[i][s] >> G[i][s] >> B[i][s];
            P[i][s] /= 100;
        }
    }

    ftype low = S;
    ftype high = 1e18;

    for (int i = 0; i < 150; i++) {
        ftype mid = (low + high) / 2;
        restart = mid;
        ftype computed = expected(0, S) + S;
        if (mid <= computed) low = mid;
        else high = mid;
    }

    cout << setprecision(20) << fixed << low << '\n';

    return 0;
}
```

13.2 Fair Distribution

```cpp
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

/*
Exercise:
You are given n=1e5 blueprints of buildings. Those buildings have a single ground floor of height 'G'
and at least 1 (or millions, up to you) residential floors of height 'R'. You wanna know whether it is
possible to distribute those blueprints onto two nonempty sets such that it is possible to build all
buildings of both sets and that the sum of the heights of each set of buildings ends up equal.

Idea:
Say you partitioned the blueprints onto the nonempty sets A and B.
This means you need the following to hold:

sum(i E A) Gi + sum(i E A) Ri * ki = sum(i E B) Gi + sum(i E B) Ri * ji
sum(i E A) Ri * ki - sum(i E B) Ri * ji = sum(i E B) Gi - sum(i E A) Gi
sum(i E A) Ri * ki + sum(i E B) Ri * (-ji) = sum(i E B) Gi - sum(i E A) Gi

Notice that the above expression is a linear combination of all Ri. This means that an answer
exists iff there is a linear combination of Ri that ends up summing any possible difference of
the G of the sets, given that there are 2 nonempty partitions of G with that difference.
Therefore, from "extended Bezout", there is an answer iff
gcd(all Ri) | some difference of G
Also, it is clear that, if some possible difference is different than the sum of all G, there
is always 2 nonempty sets that give that difference. But if the difference is EQUAL to the sum
of all G, then, you need at least a Gi = 0 and any other G, that is:
Gi = 0 for some Gi
n >= 2

But a linear combination may include negative coefficients. What about that?
Suppose you want the coefficient Ri to be increased on the left. Then, you grab any Rj on the
right, and sum to both sides Ri*Rj*k, where k is any integer > 1. What will end up happening
is that you increased left's Ri coefficient by (Rj*k), while right's Rj coefficient by (Ri*k).
Therefore, you can increase any coefficient arbitrarily.

So now, the solution is to iterate over all possible differences of G, do the checks above, and
see if gcd(all Ri) divides the current difference.

How to get those differences efficiently? The DP is like subset sum, but we know that the sum of Gi
is O(n), that is, there are at most O(sqrt(n)) different Gi.
With that in mind, we can make a map of frequencies of Gi, and, instead of putting them all in the
knapsack vector, we will COMPRESS them.

Say there are 14 items with weight 'w'. You can write them as:
[w, 2w, 4w, 7w]
and every sum that was possible with the 14 individual items is possible now with these powers
of 2. Therefore, if our frequency map had O(sqrt(n)) entries, each one with frequency O(n), then,
we can reduce that to O(sqrt(n) * log(n)) entries. Then, the other dimension of the DP is their sum
O(n), leading to a O(n * sqrt(n) * log(n)) solution.
*/
```

```
signed main() {
    cin.tie(0)->sync_with_stdio(0);

    // ----------- READING INPUT ----------- //
    int n;
    cin >> n;

    int accGcd;
    int sumG = 0;
    bool nullG = false;
    map<int, int> freq;
    for (int i = 0; i < n; i++) {
        int g, r;
        cin >> g >> r;
        if (i == 0) accGcd = r;
        else accGcd = gcd(accGcd, r);
        freq[g]++;
        sumG += g;
        if (!g)
            nullG = true;
    }

    // ----------- COMPUTING COMPRESSED KNAPSACK VALUES ----------- //
    vector<int> cost;
    for (auto[weight, count] : freq) {
        int rem = count;
        for (int i = 1; i <= rem; i *= 2) {
            cost.push_back(weight*i);
            rem -= i;
        }
        if (rem) cost.push_back(weight*rem);
    }

    // ----------- RUNNING DP ----------- //
    // can achieve weight 'w' using first 'i' indices
    vector<vector<int>> dp(sumG+1, vector<int>(cost.size(), 0));
    for (int i = 0; i < cost.size(); i++)
        dp[0][i] = 1;
    dp[cost[0]][0] = 1;

    for (int w = 1; w <= sumG; w++) {
        for (int i = 1; i < cost.size(); i++) {
            if (dp[w][i-1])
                dp[w][i] = 1;
            else if (cost[i] <= w)
                dp[w][i] = dp[w-cost[i]][i-1];
        }
    }

    for (int w = 1; w < sumG; w++) {
        if (dp[w][cost.size()-1]) {
            int A = w;
            int B = sumG - A;
            int D = abs(A-B);
            if (D % accGcd == 0) {
                cout << "Y\n";
                return 0;
            }
        }
    }

    // last case: difference == sumG
    if (sumG % accGcd == 0 && nullG && n >= 2)
        cout << "Y\n";
    else
        cout << "N\n";
    return 0;
}
```

13.3 Greek Casino

```
#include <bits/stdc++.h>

using namespace std;

//#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;
using ftype = double;

/*
Exercise:
There are n slots (from 1 to n). You start at slot 1. Say you are currently at slot 'x'. A random integer
```

```
'y' from 1 to n is chosen. If z=lcm(x, y) > n, then, the game ends. Else, you receive 1 coin and you go
to slot z. Print the expected number of coins given the probabilities of each number being selected.

Idea:
Let's build a Markov Chain of the input. To compute the edges, I'll use the fact that a
vertex 'i' only goes to a multiple of 'i'. That bounds the amount of edges by O(nlogn).
For each edge (i, m*i), I wanna see what 'j's randomly selected would lead me to go through it.
That is, which j have the following property:
lcm(i, j) = m*i
i*j/gcd(i, j) = m*i
j = m * gcd(i, j)

I don't know how to solve that mathematically, but I do know how to bruteforce it.
Iterate through every divisor 'g' of 'i' and consider it as a gcd of something. Compute
j = m*g
and see if that makes the equation true. If it does, we found a working 'j'.

Total complexity: O(nlog^3n)
*/

vector<vector<int>> divsSieve(int n) {
    vector<vector<int>> divisors(n+1);
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j += i)
            divisors[j].push_back(i);
    return divisors;
}

signed main() {
    int n;
    scanf("%d", &n);

    vector<int> w(n+1);
    int wSum = 0;
    for (int i = 1; i <= n; i++) {
        scanf("%d", &w[i]);
        wSum += w[i];
    }

    vector<vector<pair<int, int>>> adj(n+1);
    vector<vector<int>> divisors = divsSieve(n);

    for (int i = 1; i <= n; i++) {                      // O(n)
        for (int m = 2; i*m <= n; m++) {                // O(logn average) because of harmonic series
            int likelihood = 0;
            for (long long g : divisors[i]) {           // O(logn average) because of sieve
                if (g*m > n) break;
                if (gcd(i, g*m) == g)                   // O(logn)
                    likelihood += w[g*m];
            }
            if (likelihood)
                adj[i].push_back({likelihood, i*m});
        }

        // now self edge
        int selfLikelihood = 0;
        for (int g : divisors[i])
            selfLikelihood += w[g];
        adj[i].push_back({selfLikelihood, i});
    }

    vector<ftype> expected(n+1, 0);
    for (int i = n; i >= 1; i--)
        for (auto[w, next] : adj[i]) {
            ftype s = ftype(w) / wSum;
            if (next == i)
                expected[i] = (expected[i] + s) / (1-s);
            else
                expected[i] += s * (expected[next]+1);
        }
    printf("%.20f", expected[1]);
    return 0;
}
```

14 Zolutions/Nacional 2022

14.1 Lazy Printing

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()
```

```
/*
Exercise:
You wanna compose a string 's' as a sum of several strings 'm' of length <= D
such that s = m1^j1 + p(m1) + m2^j2 + p(m2) + ... + mk^jk + p(mk)
where p(r) is a (possibly empty) prefix of the string r.

Idea: suppose you have a string s and you want to build it by concatenating some
other string p several times + a prefix of p. The minimum length p can have is
given by KMP's n - lps[n-1] for a string of length n.

With that in mind, we will greedily build the lps list while the value
k = i + 1 - lps[i]
is within the acceptable length (k <= d). The moment k exceeds that value,
we increment the answer and start a new kmp from scratch.
*/

signed main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    string s;
    int d;
    cin >> s >> d;
    int n = s.size();

    vector<int> lps = {0};
    int ops = 1;
    int sz = n;
    int i = 0;
    int offset = 0;
    while (++i < sz) {
        int j = lps[i-1];
        while (j > 0 && s[offset+i] != s[offset+j])
            j = lps[j-1];
        if (s[offset+i] == s[offset+j])
            j++;
        lps.push_back(j);

        int k = i + 1 - lps[i];
        if (k > d) {
            // Starting new kmp here (removing the first i letters of the string, i.e.,
            // everyone before the current position)
            ops++;
            lps = {0};
            sz -= i;
            offset += i;
            i = 0;
        }
    }

    cout << ops << '\n';

    return 0;
}
```

15 Zolutions/Nacional 2023

15.1 Keen On Order

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
Given 'k' and a sequence 'a' of size 'n', determine whether every permutation of the numbers
from 1 to 'k' appears as a subsequence of 'a'.

Idea:
If k >= 25, there is always a permutation that does not exist. The following algorithm always works:
 - Take the number whose first appearance is as far ahead as possible. It is at least the 25th element.
 - Now, take another number whose first appearence AFTER the previous number is as far ahead as possible.
   It is necessarily at least 24 numbers after the previous one.
 - Continue that idea.
That algorithm always works because for such subsequence to exist, it would have size
25 + 24 + ... + 1 = 25(1+25)/2 = 325 > n = 300
How to solve for k <= 24?
Keep a DP of bitmasks. DP[mask] returns the largest index that any permutation of the numbers on the
mask can end at. The recursion is trivial: all the permutations of this mask end in some element of
```

```
the mask. Therefore, iterate through every element, and compute what would be the maximum length with
it as last element. That length is exactly the first appearance of 'i' after dp[mask without i].
To recover the permutation itself, we will keep a vector 'p' that tells what element I should append
to get maximum permutation ending.
*/

int dp[1<<24];
int p[1<<24];
int nex[300][300];

int maxSubsequenceEnding(int mask) {
    if (dp[mask] != -1)
        return dp[mask];

    if (__builtin_popcountll(mask) == 1) {
        int i = 0;
        while (!(mask & (1<<i)))
            i++;
        p[mask] = i;
        return dp[mask] = nex[i][0]; // first appearance of i
    }

    for (int i = 0; i < 24; i++) {
        if (!(mask & (1<<i))) continue;
        int next = maxSubsequenceEnding(mask ^ (1<<i));
        if (next == INF || nex[i][next] == INF) { // this is a losing subsequence
            p[mask] = i;
            return dp[mask] = INF;
        }
        if (nex[i][next] > dp[mask]) {
            dp[mask] = nex[i][next];
            p[mask] = i; // appending i leads to the rightmost ending subsequence
        }
    }

    return dp[mask];
}

vector<int> ans;
void buildAns(int mask) {
    if (mask == 0) return;
    int next = p[mask];
    buildAns(mask ^ (1<<next));
    ans.push_back(next);
}

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int n, k;
    cin >> n >> k;

    vector<int> a(n);
    for (int &i : a) {
        cin >> i;
        i--;
    }

    if (k >= 25) {
        set<int> notUsed, notPrinted;
        for (int i = 0; i < k; i++) {
            notUsed.insert(i);
            notPrinted.insert(i);
        }

        for (int i = 0; i < n; i++) {
            if (notUsed.count(a[i])) {
                notUsed.erase(a[i]);
                if (!notUsed.size()) {
                    cout << a[i]+1 << " ";
                    notPrinted.erase(a[i]);
                    for (int i : notPrinted)
                        notUsed.insert(i);
                }
            }
        }

        for (int i : notPrinted)
            cout << i+1 << " ";
        cout << '\n';
        return 0;
    }

    for (int i = 0; i < 300; i++)
        for (int j = 0; j < 300; j++)
            nex[i][j] = INF;
    for (int i = n-1; i >= 0; i--)
        for (int j = i; j >= 0; j--)
            nex[a[i]][j] = i;

    // out of all the subsequences of this mask, which one ends the furthest away?
    memset(dp, -1, sizeof(dp));
    memset(p, -1, sizeof(p));

    int fullMask = (1<<k)-1;
    if (maxSubsequenceEnding(fullMask) != INF) cout << "*\n";
    else {
        buildAns(fullMask);
        for (int i : ans)
            cout << i+1 << " ";
        cout << '\n';
    }

    return 0;
}
```

16 Zolutions/Nacional 2024

16.1 Evereth Expedition

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

/*
Exercise:
You are given a permutation vector with numbers [1, N], some possibly missing (a 0 is placed when some
number is missing). You wanna compute any way to fill those gaps such that the final vector ends up as
a mountain permutation (strictly increasing then strictly decreasing), or tell that it is impossible.

Idea:
Either end of the mountain necessarily is 1. After we place the 1, we recursively build the mountain
for the remaining positions [2, N]. That will be the general idea of the algorithm.
There are a few cases where there is only one way to place the 1, namely:
1) The whole array only has zeroes:
    place the 1 on the left and build recursively (this will end up as an increasing sequence).
2) The 1 is already placed on the mountain:
    if its in one of the ends, alright, continue building the mountain in the opposite direction.
    else, abort (no solution).
3) There is some end that is non-zero:
    you will place the 1 in the end that is zeroed. If no end is zeroed, abort.
4) Both ends are zero:
    grab the leftmost non-zero index 'i' such that i >= l.
    grab the rightmost non-zero index 'j' such that j <= r
    let x = a[i], y = a[j]
    if x < y, then, we know that the mountain peak CANNOT be in a position <= i, because that would imply
    the height decreasing to x but then increasing again to y. If the peak is not before i, then, the segment
    [l, i] is always increasing. Therefore, there is only one position for the 1, namely, at a[l].
    Symmetric argument applies to x > y.
    What if x == y (i == j)? Well, I claim that placing the 1 on the smallest contiguous sequence of zeroes
    is optimal.
    "Proof":
    You need to make 'i' become a border as soon as possible. If you take too long, you will reach the
    case "2)", but without it being on the border - abort. Reaching it early is no problem, because
    every number smaller than it that was not placed will always be placeable on the other side (because
    the other side will be always free).

Therefore, the algorithm iterates from 1 to N and builds the mountain progressively. If something fails
in the middle, abort. Else, you end up with a valid mountain.
*/

vector<int> a, pref, nonZeroRight, nonZeroLeft, placed;

bool buildMountain(int n) {
    auto isAllZeroes = [&](int l, int r) {
        int sum = pref[r];
        if (l != 0) sum -= pref[l-1];
        return sum == 0;
    };

    int l = 0;
    int r = n-1;
    for (int cur = 1; cur <= n; cur++) {
        if (isAllZeroes(l, r)) a[l++] = cur;
        else if (placed[cur]) {
            if (a[l] == cur) l++;
            else if (a[r] == cur) r--;
            else return false;
        }
        else if (a[l] != 0 || a[r] != 0) {
            if (a[l] == 0) a[l++] = cur;
```

```
                else if (a[r] == 0) a[r--] = cur;
                else return false;
            }
            else { // both are zero
                int x = a[nonZeroRight[l]];
                int y = a[nonZeroLeft[r]];
                if (x < y) a[l++] = cur;
                else if (x > y) a[r--] = cur;
                else { // x == y
                    int dx = nonZeroRight[l] - l;
                    int dy = r - nonZeroLeft[r];

                    if (dx < dy) a[l++] = cur;
                    else a[r--] = cur;
                }
            }
        }
        return true;
    }

    signed main() {
        cin.tie(0)->sync_with_stdio(0);

        int n;
        cin >> n;

        a.resize(n);
        for (int &i : a) cin >> i;

        pref.resize(n);
        pref[0] = a[0];
        for (int i = 1; i < n; i++)
            pref[i] = pref[i-1]+a[i];

        nonZeroLeft.assign(n, 0);
        nonZeroRight.assign(n, 0);

        nonZeroLeft[0] = 0;
        for (int i = 1; i < n; i++)
            if (a[i] != 0) nonZeroLeft[i] = i;
            else nonZeroLeft[i] = nonZeroLeft[i-1];

        nonZeroRight[n-1] = n-1;
        for (int i = n-2; i >= 0; i--)
            if (a[i] != 0) nonZeroRight[i] = i;
            else nonZeroRight[i] = nonZeroRight[i+1];

        placed.assign(n+1, 0);
        for (int i : a)
            if (i)
                placed[i] = 1;

        if (!buildMountain(n)) cout << "*\n";
        else {
            for (int i : a)
                cout << i << " ";
            cout << '\n';
        }

        return 0;
    }
```

17 Zolutions/Other

17.1 Burnside

```
    signed main() {
        /*
        Given a crown with k slots for gems, where each slot can be filled in m ways, find the amount
        of different crowns that can be made (considering rotations give the same crown).
        Then, sum it all for all k in [1, n].
        This algorithm is slow, but is didactic for the Burnside's Lemma.

        Burnside's Lemma:
        (1/|G|) * sum(for g : G) |X^g|

        |G| is the amount of operations: in this case, k cyclic shifts.
        X^g is the set of elements that are equivalent under the action of g (possibly several times).
        Therefore, |X^g| is the amount of sets of elements that are equivalent under g (applied several times).

        In this case, we have k rotations, denoted by p_j (j is the amount of positions moved to the right).

        p_0 -> by not rotating anything, all the arrangements are unique: m^k arrangements
        p_1 -> only all the gems equal are equivalent: m arrangements
        ...
        p_j -> the amount of visited gems is k / gcd(k, j). Therefore, the pattern size is gcd(k, j). Since we can
            choose
        each of the gems of the pattern in m ways, the total ways to choose the pattern (and therefore the
            equivalent crowns with
        j rotations) is m^gcd(k, j)
        */

        int n, m;
        cin >> n >> m;
        int ans = 0;

        for (int k = 1; k <= n; k++) {
            int prelim = 0;
            for (int j = 0; j < k; j++) {
                int g = gcd(k, j);
                prelim = (prelim + fastPow(m, g)) % MOD;
            }

            // now dividing by k
            prelim = prelim * multiplicativeInverse(k, MOD) % MOD;
            ans = (ans + prelim) % MOD;
        }

        cout << ans << endl;
        return 0;
    }
```

17.2 Chopping Down Trees

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

/*
Exercise:
You will chop down M trees that are on a field with integer coordinates on [1, N]. You don't know their
coordinates yet. There are neighbour's trees on 0 and N+1. All of the trees have height H. Chopping down a
tree at position x will knock down trees up to x+H (inclusive), in the direction the tree at x was falling.
Given that the distributions of the trees on the field are equaly likely, compute the probability that you
can chop down all of your trees without knocking down your neighbour's trees.

Idea:
There are M trees. This means that there are M+1 space intervals in between your trees and neighbour's trees.
The big idea is that the job can be completed iff one of those spaces is > h. Ignoring other combinatorial
parts, we have to count in how many ways we can distribute N+1 units of space in M+1 intervals such that
each interval has size <= h (we are computing the complement). This can be done with FFT + binary power.
Build a vector of size h+1 where every element is 1 except h[0]. This represents that there is 1 way to
distribute 'i' units of space in 1 interval (except 0 units of space, because the trees must be spaced by at
least 1 unit). Now, if we compute h^k, we will know in how many ways we can distribute 'i' units of space
in 'k' intervals. The recursive idea is: we are given h^(k-1), the distributions of 'i' units of space in
'k-1' intervals. Now, to compute h^k we would, for each position 'i', iterate through every size 's' (s <= i)
the rightmost interval could have and append it to the right of 'k-1' intervals with sum i-s. That is
the definition of FFT (h[i] = sum p[s]*p[i-s] for all s, where p is the previous h vector, h^(k-1)).
So, we build 'h', compute h^(m+1) and use h[n+1] to do the rest of the exercise (which I won't explain here).
*/

const int MOD = 998244353;

inline int add(int a, int b) {
    a += b;
    if (a >= MOD) a -= MOD;
    return a;
}

inline int mul(int a, int b) {
    return a * b % MOD;
}

int pwr(int a, int b){
    int r = 1;
    while (b) {
        if (b & 1) r = mul(r, a);
        a = mul(a, a);
        b >>= 1;
    }
    return r;
}

int inv(int x) {
    return pwr(x, MOD-2);
}

void ntt(vector<int> &a, bool rev) {
    int n = a.size();
    vector<int> b = a;
    int g = 1;
    while (pwr(g, MOD / 2) == 1)
        g++;
    if (rev)
        g = inv(g);
    for (int step = n / 2; step; step /= 2) {
        int w = pwr(g, MOD / (n / step));
        int wn = 1;
        for (int i = 0; i < n / 2; i += step) {
            for (int j = 0; j < step; j++) {
                int u = a[2 * i + j];
                int v = mul(wn, a[2 * i + j + step]);
                b[i + j] = add(u, v);
                b[i + n / 2 + j] = add(u, MOD - v);
            }
            wn = mul(wn, w);
        }
        swap(a, b);
    }
    if (rev) {
        int n1 = inv(n);
        for (int &x : a)
            x = mul(x, n1);
    }
}

vector<int> multiply(vector<int> a, vector<int> b) {
    int n = 1;
    while (n < a.size() + b.size())
        n *= 2;
    a.resize(n); b.resize(n);
    ntt(a, false); ntt(b, false);

    for (int i = 0; i < n; i++)
        a[i] = mul(a[i], b[i]);

    ntt(a, true);
    return a;
}

// discards everything after lastIndex
vector<int> pwr(vector<int> a, int b, int lastIndex) {
    vector<int> r(a.size(), 0); r[0] = 1; // neutral polynomial
    while (b) {
        if (b & 1) {
            r = multiply(r, a);
            r.resize(lastIndex+1);
        }
        a = multiply(a, a);
        a.resize(lastIndex+1);
        b >>= 1;
    }
    return r;
}

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int tt;
    cin >> tt;
    while (tt--) {
        int n, m, h;
        cin >> n >> m >> h;

        vector<int> poly(h+1, 1);
        poly[0] = 0;
        poly = pwr(poly, m+1, n+1);

        int nf = 1;
        for (int i = 2; i <= n; i++)
            nf = mul(nf, i);
        int mf = 1;
        for (int i = 2; i <= m; i++)
            mf = mul(mf, i);
        int nmf = 1;
        for (int i = 2; i <= n-m; i++)
            nmf = mul(nmf, i);
        int combNM = mul(nf, mul(inv(mf), inv(nmf)));
        int icombNM = inv(combNM);
        int ans = mul(add(combNM, MOD - poly[n+1]), icombNM);
        cout << ans << '\n';
    }

    return 0;
}
```

17.3 Different Triangles

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

//#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;
const int MOD = 1e9+7;

/*
Exercise:
Count how many triples (a, b, c) with sum <= n there are such that
a <= b <= c
and (a, b, c) forms a triangle (that is, a+b > c).

Idea:
Build a function count(n, k) that returns in how many ways you can distribute at most 'n'
pellets into 'k' piles, in increasing amount between piles (it works recursively by
incrementing all k piles with 1 pellet at once).
Then, if you set the answer as count(n, 3), you have all of the triangles, but you also
counted wrongly the ones where the following did not hold:
a+b > c
That is, where the following happened:
a+b <= c
To compute the incorrect sums, we can iterate through every possible value of c [0, n]
and then count in how many ways I can distribute the remaining n-c pellets in 2 piles
'a' and 'b' such that their sum is at most c. That is exactly count(min(n-c, c), 2).
*/

inline int add(int a, int b) {
    a += b;
    if (a >= MOD) a -= MOD;
    return a;
}

const int MAXN = 1e6;
int dp[MAXN+1][4];
int count(int n, int k) {
    if (n < 0) return 0;
    if (k == 0 || n == 0) return 1;
    if (dp[n][k] != -1) return dp[n][k];
    return dp[n][k] = add(
        count(n-k, k),
        count(n, k-1)
    );
}

signed main() {
    cin.tie(0)->sync_with_stdio(0);
    memset(dp, -1, sizeof(dp));

    int n;
    cin >> n;

    int ans = count(n, 3);
    for (int c = 0; c <= n; c++)
        ans = add(ans, MOD-count(min(n-c, c), 2));
    cout << ans << '\n';

    return 0;
}
```

17.4 Knights In The Board

```cpp
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

/*
Exercise:
You are given a n*n = 25*25 chess board with some knights on it. Print the smallest amount of knights
you have to remove so no knight attacks another.

Idea:
If you build the collision graph, you will end up with a bipartite graph. Now, what you want is to
remove the smallest amount of vertices such that the graph ends up with no edges. That definition
is the same as the min vertex cover problem (smallest set of vertices such that every edge has at
least one endpoint on that set). By some theorem, the size of that set is exactly the maximum flow.
So simply build the graph and throw maxflow.
*/

vector<ii> cor = {
    {-1, -2},
    {1, -2},
    {2, -1},
    {2, 1},
    {1, 2},
    {-1, 2},
    {-2, 1},
    {-2, -1}
};

class FordFulkerson {
    bool __dfs(int u, int t) {
        visited[u] = 1;
        if (u == t) return true;
        for (int next : adj[u]) {
            if (!visited[next] && cap[u][next]) {
                if (__dfs(next, t)) {
                    p[next] = u;
                    return true;
                }
            }
        }
        return false;
    }

    bool dfs(int u, int t) {
        visited.assign(n, 0);
        p.assign(n, -1);
        return __dfs(u, t);
    }

public:
    vector<vector<int>> adj, cap;
    vector<int> p, visited;
    int n;

    FordFulkerson(vector<vector<int>> &adj, vector<vector<int>> &cap) : adj(adj), cap(cap) {
        n = adj.size();
    }

    int maxflow(int s, int t) {
        int maxflow = 0;
        while (dfs(s, t)) {
            int increment = INF;
            for (int u = t; p[u] >= 0; u = p[u])
                increment = min(increment, cap[p[u]][u]);

            for (int u = t; p[u] >= 0; u = p[u]) {
                cap[p[u]][u] -= increment;
                cap[u][p[u]] += increment;
            }
            maxflow += increment;
        }
        return maxflow;
    }
};

const int MAXN = 25;
int id[MAXN][MAXN];
ii coord[MAXN*MAXN];

signed main() {
    cin.tie(0)->sync_with_stdio(0);
    memset(id, -1, sizeof(id));

    int n, k;
    cin >> n >> k;

    const int SOURCE = n*n;
    const int SINK = n*n+1;
    const int V = n*n+2;
    vector<vector<int>> adj(V), cap(V, vector<int>(V));

    for (int i = 0; i < k; i++) {
        int r, c;
        cin >> r >> c;
        r--; c--;
        id[r][c] = i;
        coord[i] = {r, c};
    }

    for (int i = 0; i < k; i++) {
        auto[r, c] = coord[i];

        if ((r+c)&1) { // right side
            adj[i].push_back(SINK);
            adj[SINK].push_back(i);
            cap[i][SINK] = 1;
```

```
                continue;
            }

            // left side
            adj[i].push_back(SOURCE);
            adj[SOURCE].push_back(i);
            cap[SOURCE][i] = 1;

            for (auto[x, y] : cor) {
                int xx = r+x;
                int yy = c+y;
                if (xx >= 0 && xx < n && yy >= 0 && yy < n && id[xx][yy] != -1) {
                    adj[i].push_back(id[xx][yy]);
                    adj[id[xx][yy]].push_back(i);
                    cap[i][id[xx][yy]] = 1;
                }
            }
        }

    FordFulkerson ff(adj, cap);
    cout << ff.maxflow(SOURCE, SINK) << '\n';

    return 0;
}
```

17.5 Lagrange Interpolation

```
/*
About Lagrange interpolation:
You are given k+1 points. You want the coefficients of the kth degree polynomial
that goes through them. You can do that for arbitrary (x, y) in O(k^2) time. This
is explained as follows:
Suppose the said polynomial is
P(x) = l_1*f(x1) + l_2*f(x2) + ... + l_k+1*f(x_(k+1))
If you compute P(x1), you'd like the following to happen:
l_1 = 1
l_2 = 0
l_3 = 0
...
l_(k+1) = 0
In general, for P(xi), you want l_i to be 1 if x's index is i and 0 if its not.
Those l_i's are actually polynomials, and their shape is the following:
l_i = (x-x1) * (x-x2) * (x-x3) * ... * (x-x_(i-1)) * (x-x_(i+1)) * ... * (x-x_(k+1))
DIVIDED BY
(x_i-x1) * (x_i-x2) * (xi-x3) * ... * (xi-x_(i-1)) * (xi-x_(i+1)) * ... * (xi-x_(k+1))
The numerator has everyone but (x-xi), and the denominator has everyone but (xi-xi).
It is pretty clear that if you throw xi onto l_i it will be 1, and 0 if you throw
any of the other (k+1-1) points.

So, it means that P is already the polynomial you were searching for. Evaluating it
takes O(k^2), unless there is any property of the numbers you are working with such
that you can transition from l_i to l_(i+1) in sublinear time...
*/

#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define detachingFromC ios::sync_with_stdio(0);cin.tie(0);
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

const int MOD = 1e9+7;

/*
1) Compute k+2 points to define the k+1th degree polynomial (the sum itself)
2) Compute the polynomial l1(n) in O(k) time
3) Transition from l_i to l_(i+1) in O(1) time for all i and evaluate the polynomial
*/

tuple<int, int, int> extendedGcd(int a, int b) {
    if (b == 0) return make_tuple(a, 1, 0);
    auto[q, w, e] = extendedGcd(b, a%b);
    return make_tuple(q, e, w-e*(a/b));
}

int multiplicativeInverse(int n, int mod) {
    // (n)x + (mod)y = 1 (aka their difference is 1)
    n = (n % mod + mod) % mod;
    auto[g, x, y] = extendedGcd(n, mod);
    return (x % mod + mod) % mod;
}

int fastPow(int base, int e) {
    if (e == 0) return 1;
    if (e == 1) return base;
    int h = fastPow(base, e/2);
    if (e % 2 == 1) return h*h % MOD * base % MOD;
    return h*h % MOD;
}

signed main() {
    detachingFromC;

    int n, k;
    cin >> n >> k;

    // 1)
    vector<int> y(k+3, 0);
    y[1] = 1;
    for (int xi = 2; xi <= k+2; xi++)
        y[xi] = (y[xi-1] + fastPow(xi, k)) % MOD;

    if (n <= k+2) {
        cout << y[n] << endl;
        return 0;
    }

    // 2)
    int num = 1;
    int den = 1;
    for (int xi = 2; xi <= k+2; xi++) {
        num = num * (n - xi) % MOD;
        den = (den * (1 - xi) % MOD + MOD) % MOD;
    }

    // 3)
    int ans = num * multiplicativeInverse(den, MOD) % MOD * y[1] % MOD;
    for (int xi = 2; xi <= k+2; xi++) {
        // num: take out factor (x-x_i) and bring (x-x_(i-1))
        num = num * multiplicativeInverse(n - xi, MOD) % MOD;
        num = num * (n - (xi-1)) % MOD;
        // den: take out factor (x_(i-1) - x_last) and bring (xi - x1)
        den = den * multiplicativeInverse(xi-1 - (k+2), MOD) % MOD; // k+2 == x_last
        den = den * (xi-1) % MOD;
        // increment answer by num/den * yi
        ans += num * multiplicativeInverse(den, MOD) % MOD * y[xi] % MOD;
        ans %= MOD;
    }

    cout << ans << endl;
    return 0;
}
```

17.6 Mos Algorithm

```
/*
Mo's Algorithm: used in some very particular range queries that do NOT
have updates.

Idea: sort the queries and try to use the result of some for the next.
It is done like this: you will separate the queries by their left indices
on buckets of size sqrt(vector_length) and then sort them in decreasing right.

Then, when you process them:
    for a given bucket, initialize l = 0 and r as the right of the first query.
    Then, for the next queries on this bucket, r will only go left and l
    will jump between any position of the sqrt bucket.
*/

struct Query {
    int l, r, id;
};

int BUCKET_SIZE = ceil(sqrt(n));
vector<vector<Query>> buckets(BUCKET_SIZE);

for (int i = 0; i < m; i++) {
    int l, r;
    cin >> l >> r;
    l--; r--;
    buckets[l/BUCKET_SIZE].push_back({l, r, i});
}

auto cmpReverseRight = [&](Query &q1, Query &q2) {
    return q1.r > q2.r;
};

vector<int> counter(100000+5, 0);
int l = 0;
int r = -1;
int ans = 0;
vector<int> answers(m);

for (vector<Query> &v : buckets) {
    // set r to first query (increase range)
    if (!v.size()) continue;
    sort(all(v), cmpReverseRight);
    while (r < v[0].r) { // this one grabs v[0].r (inclusive)
        r++;
        if (counter[a[r]] == a[r]) ans--;
        counter[a[r]]++;
        if (counter[a[r]] == a[r]) ans++;
    }

    for (Query &q : v) {
        // move l right, if needed (decrease range)
        while (l < q.l) {
            if (counter[a[l]] == a[l]) ans--;
            counter[a[l]]--;
            if (counter[a[l]] == a[l]) ans++;
            l++;
        }

        // move l left, if needed (increase range)
        while (l > q.l) {
            l--;
            if (counter[a[l]] == a[l]) ans--;
            counter[a[l]]++;
            if (counter[a[l]] == a[l]) ans++;
        }

        // move r left, if needed (decrease range)
        while (r > q.r) {
            if (counter[a[r]] == a[r]) ans--;
            counter[a[r]]--;
            if (counter[a[r]] == a[r]) ans++;
            r--;
        }

        answers[q.id] = ans;
    }
}
```

17.7 Offline Dynamic Connectivity

```
/*
Offline dynamic connectivity. O(q log q log n)
You wanna know how many connected components a graph has and insert/delete edges efficiently. What you
will do is the following: keep something similar to a "segment tree" of the queries. Then, read the
queries and keep track of the timespan of an edge. Update the segment tree and say "edge (u,v) lasted
between the lth and rth query", that is, push back the edge (u, v) on each of the logn segments that
range is composed of. Then, do a dfs from the root to each of the 'q' leaves of the segment tree. On
the way down, add the edges to your graph, and when coming back, remove them. When you reach the leaf,
you have exactly all the edges the query should have in that moment. Print the amount of connected
components and you are done.

On this exercise specifically, you wanna know how many edges you need to insert onto A until it contains B.
Say the result graph is a graph made by adding onto A every edge (x,y) such that (x,y) was on B but (x,y) were
not connected on A. The answer would be the difference between the amount of connected components on A and on
that resulting graph. But since adding the edges (x,y) that did not hold that property is neutral, then, that
resulting graph has the same connected components as the graph AUB (adding everyone from B onto A). So now, the
answer becomes the difference in connected components of A and AUB, which is easier to compute.
We have a list of edges that should be on graphs A and B for each query on the segment tree. You simply add A's
edges onto A and AUB, and B's edges to AUB. Print their difference in components and that's it.
*/

#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

struct RollbackUnionFind {
    vector<int> parent, size;
    int components;
    stack<pair<int*, int>> history;

    RollbackUnionFind(int n=0) {
        parent.resize(n);
        size.assign(n, 1);
        iota(all(parent), 0);
        components = n;
    }

    int find(int v) {
        if (v == parent[v])
            return v;
        return find(parent[v]);
    }

    void change(int &x, int newVal) {
        history.push({&x, x}); // x's current state
        x = newVal;
    }

    void rollback() {
        auto[ptr, val] = history.top(); history.pop();
        *ptr = val;
    }

    void unionSets(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) return;
        if (size[a] < size[b]) swap(a, b);
        change(parent[b], a);
        change(size[a], size[a]+size[b]);
        change(components, components-1);
    }
};

struct Edge {
    char c;
    int x, y;
    Edge(char c, int x, int y) : c(c), x(x), y(y) {}
};

class SegmentTree {
    struct Segment {
        int l, r;
        vector<Edge> edg;
    };

    vector<Segment> t;
    RollbackUnionFind A, AUB;

    void build(int v, int tl, int tr) {
        t[v] = {tl, tr, {}};
        if (tl != tr) {
            int mid = (tl + tr) / 2;
            build(2*v, tl, mid);
            build(2*v+1, mid+1, tr);
        }
    }

    void addEdge(int v, int l, int r, Edge &e) {
        if (t[v].l > r || t[v].r < l)
            return;
        if (t[v].l >= l && t[v].r <= r) {
            t[v].edg.push_back(e);
            return;
        }
        addEdge(2*v, l, r, e);
        addEdge(2*v+1, l, r, e);
    }

    void dfs(int v) {
        int Astate = A.history.size();
        int AUBstate = AUB.history.size();
        for (Edge &e : t[v].edg) {
            AUB.unionSets(e.x, e.y);
            if (e.c == 'A')
                A.unionSets(e.x, e.y);
        }
        if (t[v].l == t[v].r) {
            // at leaf. Compute answer.
            ans[t[v].l] = A.components - AUB.components;
        }
        else {
            dfs(2*v); dfs(2*v+1);
        }

        while (A.history.size() != Astate) A.rollback();
        while (AUB.history.size() != AUBstate) AUB.rollback();
    }

public:
    vector<int> ans;
    SegmentTree(int sz, int v) {
        t.resize(4*sz);
        ans.resize(sz);
        build(1, 0, sz-1);
        A = RollbackUnionFind(v);
        AUB = RollbackUnionFind(v);
    }

    void addEdge(int l, int r, Edge e) {
        addEdge(1, l, r, e);
    }
    void dfs() {
        dfs(1);
    }
};

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int n, q;
    cin >> n >> q;

    vector<map<ii, int>> pos(2);
    SegmentTree st(q, n);

    for (int cur = 0; cur < q; cur++) {
        char c;
        int x, y;
        cin >> c >> x >> y;
        x--; y--;
        if (x > y) swap(x, y);
        int id = c-'A';
        if (pos[id].count({x, y})) {
            st.addEdge(pos[id][{x, y}], cur-1, {c, x, y}); // cur-1 because its last appearance is on the
                previous query
            pos[id].erase({x, y});
        }
        else {
            pos[id][{x, y}] = cur;
        }
    }

    // add every remaining edge to q-1
    for (int i = 0; i < 2; i++)
        for (auto[edg, idx] : pos[i])
            st.addEdge(idx, q-1, {char(i+'A'), edg.first, edg.second});

    st.dfs();
    for (int i : st.ans)
        cout << i << '\n';

    return 0;
}
```

18 Zolutions/Regional 2019

18.1 Denouncing Mafia

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

/*
Exercise:
Given a tree, you can spend an operation to remove a vertex and everyone above him. Remove the maximum
amount of vertices with 'k' operations.

Idea:
If you have a forest and a single query, you know that the best thing you can do is remove the tallest
path (i.e., the path from the vertex with most vertices below it, and everyone below it). I won't prove it,
but that is the optimal greedy strategy (even for several picks).
Therefore, what you will do is grab the vertices from tallest to shortest, and, if it wasn't picked before,
mark is as "visited" and pick everyone below him.
*/

const int BOSS = 0;
const int NO_CHILD = -1;
const int MAXN = 1e5;

vector<vector<int>> adj;
priority_queue<ii> pq;
int depth[MAXN];
int son[MAXN];
int visited[MAXN];

void dfs(int u) {
    // depth[u] starts as 0, I'll increment it later
    for (int next : adj[u]) {
        dfs(next);
        if (depth[next] >= depth[u]) {
                depth[u] = depth[next];
                son[u] = next;
            }
        }
        depth[u]++;
        pq.push({depth[u], u});
}

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int n, k;
    cin >> n >> k;
    adj.assign(n, {});

    for (int i = 1; i < n; i++) {
        int p;
        cin >> p;
        p--;
        // only putting edges downwards
        adj[p].push_back(i);
    }

    memset(son, NO_CHILD, sizeof(son));
    dfs(BOSS);

    int ans = 0;
    while (k) {
        if (!pq.size()) break;
        auto[depth, node] = pq.top(); pq.pop();
        if (!visited[node]) {
            ans += depth;
            k--;
            while (node != NO_CHILD) {
                visited[node] = 1;
                node = son[node];
            }
        }
    }

    cout << ans << '\n';

    return 0;
}
```

18.2 Keep Calm And Sell Balloons

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define detachingFromC ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

/*
Exercise:
You are given a 2*n matrix. You start on some cell and move orthogonally/diagonally without repeating
cells. You wanna know in how many different ways you can traverse the matrix.

Idea:
Let 'back(n)' be the amount of ways you can walk from the corner of a 2*n block to the
other side, but coming back. Clearly, back(n) = 2^n.

Let 'fwd(n)' be the amount of ways you can walk from the corner of a 2*n block to the
other side, but not necessarily coming back. Then:
fwd(n) = back(n) + 2*fwd(n-1) + 4*fwd(n-2)
The first term (back(n)) is simply coming back.
The second term (2*fwd(n-1)) is filling the first 2 tiles and going forward.
The third term (4*fwd(n-2)) is grabbing the first and second column, going back and then forward.
The base cases are:
fwd(0) = 1
fwd(1) = 2

Then, the answer is the following sum s(n):
int ans = 2*fwd(n); // Needed for the 2 corners (so n must be >= 2)
for (int col = 2; col <= n-1; col++) {
    ans += 2 * back(col-1) * fwd(n-col);
    ans += 2 * back(n-col) * fwd(col-1);
}

That is because you are summing, for each column other than the corners, the 2 ways to start on it,
times (the ways to go back left then forward right plus back right and forward left).
That can be further simplified to:
ans += back(col) * fwd(n-col);
ans += back(n-col+1) * fwd(col-1);
```

```
And, because each sum appears twice on the summing process:
ans += 2 * back(col) * fwd(n-col);

And, therefore:
ans += back(col+1) * fwd(n-col);

So, we have, for n >= 2, the following sum:
for (int col = 2; col <= n-1; col++)
    ans += back(col+1) * fwd(n-col);

We want to write it recursively, because that sum is O(n). Take a look at that sum for some given n:
s(n) = back(3) * fwd(n-2)
     + back(4) * fwd(n-3)
     + ...
     + back(n) * fwd(1)
And the sum s(n+1) we want is:
s(n+1) = back(3) * fwd(n-1)
       + back(4) * fwd(n-2)
       + ...
       + back(n) * fwd(2)
       + back(n+1) * fwd(1)
It is kinda clear that:
s(n+1) = 2*s(n) + back(3) * fwd(n-1)
s(n)   = 2*s(n-1) + 8*fwd(n-2)

And that gives us all of our recurrences:
back(n) = 2*back(n-1)                     | back(0) = 1
fwd(n)  = back(n) + 2*fwd(n-1) + 4*fwd(n-2) | fwd(0) = 1, fwd(1) = 2
s(n)    = 2*s(n-1) + 8*fwd(n-2)             | s(2) = 0

The answer for the exercise is 2*fwd(n) + s(n).
Let's build the matrix for the fast power. It should have s(n+1), so the previous phase
must have s(n) and fwd(n-1).
s(n)      | s(n+1)
fwd(n-1)  |

Since the previous has fwd(n-1), this one should have fwd(n), and fwd(n) demands back(n),
fwd(n-1) and fwd(n-2):
s(n)      | s(n+1)
fwd(n-1)  | fwd(n)
back(n)   |
fwd(n-2)  |

Since the previous has back(n), this one must have back(n+1) (no dependencies). It also must
have fwd(n-2 + 1) = fwd(n-1), also without dependencies. Therefore:
s(n)      | s(n+1)
fwd(n-1)  | fwd(n)
back(n)   | back(n+1)
fwd(n-2)  | fwd(n-1)

And our square matrix is:
            s(n)
            fwd(n-1)
            back(n)
            fwd(n-2)
===================
2 8 0 0     s(n+1)
0 2 1 4     fwd(n)
0 0 2 0     back(n+1)
0 1 0 0     fwd(n-1)

Since n-1 is the smallest argument in there, and it must be >= 0, we will hardcode our
base matrix with:
s(2)    = 0
fwd(1)  = 2
back(2) = 4
fwd(0)  = 1

And that's all.
*/

const bool DEBUG = true;

const int MOD = 1e9 + 7;

struct Matrix {
    int l, c;
    vector<vector<ll>> m;
    Matrix(int l, int c) : l(l), c(c) {
        m.assign(l, vector<ll>(c, 0));
    }

    Matrix operator* (Matrix &o) {
        Matrix res(l, o.c);
        for (int i = 0; i < l; i++)
            for (int j = 0; j < o.c; j++)
                for (int k = 0; k < c; k++)
                    res[i][j] = (res[i][j] + m[i][k] * o[k][j]) % MOD;
        return res;
    }

    vector<ll>& operator[](int idx) {
        return m[idx];
    }

    Matrix pow(ll e) {
        if (e == 0) return identity(l);
        if (e == 1) return *this;
        Matrix squared = (*this) * (*this);
        if (e % 2 == 0) return squared.pow(e/2);
        return squared.pow(e/2) * (*this);
    }

    Matrix identity(int sz) {
        Matrix mm(sz, sz);
        for (int i = 0; i < sz; i++)
            mm[i][i] = 1;
        return mm;
    }
};

signed main() {
    detachingFromC;

    int n;
    cin >> n;

    if (n == 1) {
        cout << 2 << endl;
        return 0;
    }

    Matrix adv(4, 4);
    adv[0][0] = 2;
    adv[0][1] = 8;
    adv[1][1] = 2;
    adv[1][2] = 1;
    adv[1][3] = 4;
    adv[2][2] = 2;
    adv[3][1] = 1;

    // This one has s(2) precomputed
    Matrix base(4, 1);
    base[0][0] = 0;
    base[1][0] = 2;
    base[2][0] = 4;
    base[3][0] = 1;

    // Still gotta advance n-2 more times
    Matrix result = adv.pow(n-2) * base;
    int res = result[0][0]; // s(n)
    // Advance once again to get fwd(n)
    result = adv * result;
    int fwd = result[1][0];
    cout << (2*fwd%MOD + res%MOD) % MOD << endl;

    return 0;
}
```

19 Zolutions/Regional 2023

19.1 K Pra Mais K Pra Menos

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
You are given the coefficients of the polynomials t(x) and p(x). Compute t(x+k) + p(x-k).

Idea:
Let's count how many times the term x^t appears on the whole expansion of the polynomial at (x+k).
Remembering binomial expansion:
(x+k)^n = (n choose 0)*x^n*k^0 + (n choose 1)*x^(n-1)*k^1 + ... + (n choose n)*x^0*k^n

The term x^t will appear on the expansion of x^i as its jth term iff i-j=t.
So, our desired vector becomes
c[t] = sum(i-j=t) (i choose j) * k^j * a[j];
     = sum(i-j=t) i! * k^j * a[j] / (j! * t!)
     = 1/t! * sum(i-j=t) i! * k^j * a[i] / j!        // ALWAYS TAKE OUT CONSTANTS OF THE SUM, WHENEVER POSSIBLE
     = 1/t! * sum(i-j=t) A[i] * B[j]
```

```
where
A[i] = i! * a[i]
B[i] = k^i * / i!
And that can be trivially computed by reversing A (btw, reversing B is not the same).
Now, given the frequency of x^t on the expansion, we do that for the two polynomials and simply
sum the frequencies of each x^i for every i.
*/

const int MOD = 998244353;

inline int add(int a, int b){
    a += b;
    if (a >= MOD) a -= MOD;
    return a;
}

inline int mul(int a, int b){
    return a * b % MOD;
}

int pwr(int a, int b){
    int r = 1;
    while (b) {
        if (b & 1) r = mul(r, a);
        a = mul(a, a);
        b >>= 1;
    }
    return r;
}

int inv(int x) {
    return pwr(x, MOD-2);
}

void ntt(vector<int> &a, bool rev){
    int n = a.size();
    vector<int> b = a;
    int g = 1;
    while (pwr(g, MOD / 2) == 1)
        g++;
    if (rev)
        g = inv(g);
    for (int step = n / 2; step; step /= 2) {
        int w = pwr(g, MOD / (n / step));
        int wn = 1;
        for (int i = 0; i < n / 2; i += step) {
            for (int j = 0; j < step; j++) {
                int u = a[2 * i + j];
                int v = mul(wn, a[2 * i + j + step]);
                b[i + j] = add(u, v);
                b[i + n / 2 + j] = add(u, MOD - v);
            }
            wn = mul(wn, w);
        }
        swap(a, b);
    }
    if(rev) {
        int n1 = inv(n);
        for (int &x : a)
            x = mul(x, n1);
    }
}

vector<int> multiply(vector<int> &a, vector<int> &b) {
    vector<int> fa(all(a)), fb(all(b));
    int n = 1;
    while (n < a.size() + b.size())
        n *= 2;
    fa.resize(n); fb.resize(n);
    ntt(fa, false); ntt(fb, false);

    for (int i = 0; i < n; i++)
        fa[i] = mul(fa[i], fb[i]);

    ntt(fa, true);
    return fa;
}

vector<int> subtractionConvolution(vector<int> &a, vector<int> &b) {
    vector<int> at = a;
    reverse(all(at));
    vector<int> c = multiply(at, b);
    c.resize(a.size());
    reverse(all(c));
    return c;
}

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int n, k;
    cin >> n >> k;
    vector<int> t(n+1), p(n+1);
    for (int &i : t) cin >> i;
    for (int &i : p) cin >> i;
    if (k < 0) k += MOD;

    // Precomputing everything
    vector<int> fac(n+1), ifac(n+1), A(n+1), B(n+1), kpow(n+1);
    fac[0] = ifac[0] = 1;
    kpow[0] = 1;
    for (int i = 1; i <= n; i++) {
        fac[i] = mul(fac[i-1], i);
        ifac[i] = inv(fac[i]);
        kpow[i] = mul(kpow[i-1], k);
    }

    //A[i] = i! * a[i]
    //B[i] = k^i * / i!
    for (int i = 0; i <= n; i++) {
        A[i] = mul(fac[i], t[i]);
        B[i] = mul(kpow[i], ifac[i]);
    }

    vector<int> c1 = subtractionConvolution(A, B);
    for (int t = 0; t <= n; t++)
        c1[t] = mul(c1[t], ifac[t]); // multiply by the dude out of the sum (1/t!)

    // Now doing the same but for 'p'. Don't forget k swapped signs.
    k = add(0, MOD-k); // k = -k
    for (int i = 1; i <= n; i++)
        kpow[i] = mul(kpow[i-1], k);

    //A[i] = i! * a[i], but for p
    //B[i] = k^i * / i! with the new k
    for (int i = 0; i <= n; i++) {
        A[i] = mul(fac[i], p[i]);
        B[i] = mul(kpow[i], ifac[i]);
    }

    // Good old fft
    vector<int> c2 = subtractionConvolution(A, B);
    for (int t = 0; t <= n; t++)
        c2[t] = mul(c2[t], ifac[t]);

    // now we have the frequency of each dude. Print their sum
    for (int i = 0; i <= n; i++)
        cout << add(c1[i], c2[i]) << " \n"[i==n];
    return 0;
}
```

20 Zolutions/Regional 2024

20.1 Bacon Number

```
#include <bits/stdc++.h>

using namespace std;

//#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = true;

/*
Exercise:
Given a bipartite graph of 1e2 movies and 1e6 actors, print, for each of the 1e4 queries, a path of
ANY length that connects two given actors.

Idea:
Because the queries only ask for a path and not the shortest path, you can compact the graph into
any spanning (i.e., dfs) tree. Now, the big deal is, to find the path, you won't do a regular dfs
(because you wouldn't know which way to go, and would visit the whole 1e6 sized tree). Instead,
you root it and then compute LCA of that tree. Then, you will always know where to go: simply go
from cur to up[cur][0] until you reach the common ancestor.
Now, about the bounds of the answer. Since the graph is a bipartite graph (of actors and movies),
we know that, in at most 2*n steps, we visited all of the movies. Therefore, if two actors belong
to a same "movie tree", then, in at most 2*n steps, you will reach their lowest common ancestor.
That bounds the answer to at most 2*(2*n) = 4*n vertices per query.
*/

vector<vector<int>> tree;
void dfs(int u, vector<int> &vis, vector<vector<int>> &adj, int component) {
    vis[u] = component;
    for (int next : adj[u])
        if (!vis[next]) {
            tree[u].push_back(next);
            tree[next].push_back(u);
            dfs(next, vis, adj, component);
        }
}

struct LCA {
    int n, l;
    vector<vector<int>> &tree;
    int timer;
    vector<int> tin, tout;
    vector<vector<int>> up;

    LCA(vector<vector<int>> &tree, int root) : tree(tree) {
        n = tree.size();
        tin.assign(n, -1);
        tout.resize(n);
        l = ceil(log2(n));
        up.assign(n, vector<int>(l + 1));
        for (int i = 0; i < n; i++)
            if (tin[i] == -1) {
                timer = 0;
                dfs(i, i);
            }
    }

    void dfs(int v, int p) {
        tin[v] = ++timer;
        up[v][0] = p;

        for (int i = 1; i <= l; ++i)
            up[v][i] = up[up[v][i-1]][i-1];

        for (int u : tree[v])
            if (u != p)
                dfs(u, v);
        tout[v] = ++timer;
    }

    bool is_ancestor(int u, int v) {
        return tin[u] <= tin[v] && tout[u] >= tout[v];
    }

    int lca(int u, int v) {
        if (is_ancestor(u, v))
            return u;
        if (is_ancestor(v, u))
            return v;
        for (int i = l; i >= 0; --i)
            if (!is_ancestor(up[u][i], v))
                u = up[u][i];
        return up[u][0];
    }
};

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int n, m;
    cin >> n >> m;

    vector<vector<int>> adj(n+m);
    tree.assign(n+m, {});
    for (int i = 0; i < n; i++) {
        int actors;
        cin >> actors;
        while (actors--) {
            int a;
            cin >> a; a--;
            adj[i+m].push_back(a);
            adj[a].push_back(i+m);
        }
    }

    vector<int> visited(n+m, 0);
    int component = 1;
    for (int i = 0; i < n+m; i++)
        if (!visited[i])
            dfs(i, visited, adj, component++);

    LCA lca(tree, 0);

    int q;
    cin >> q;

    while (q--) {
        int x, y;
        cin >> x >> y;
        x--; y--;

        if (visited[x] != visited[y]) {
            cout << "-1\n";
            continue;
        }
        int ancestor = lca.lca(x, y);
        vector<int> fst, scd;
        int cur = x;
        while (cur != ancestor) {
            int i = cur;
            if (i >= m)
                i -= m;
            fst.push_back(i);
            cur = lca.up[cur][0];
        }
        if (ancestor >= m) fst.push_back(ancestor-m);
        else fst.push_back(ancestor);
        cur = y;
        while (cur != ancestor) {
            int i = cur;
            if (i >= m)
                i -= m;
            scd.push_back(i);
            cur = lca.up[cur][0];
        }
        reverse(all(scd));

        cout << (fst.size() + scd.size())/2 + 1 << '\n';
        for (int i : fst)
            cout << i+1 << " ";
        for (int i : scd)
            cout << i+1 << " ";
        cout << '\n';
    }

    return 0;
}
```

20.2 Couple Of Bipbop

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
Given a sequence of dance moves, determine the expected lcp of (i, j), given that (i, j) are randomly
and uniformly chosen.

Idea:
Build the suffix array and the lcp array of the input. Now, we have that solving the original problem is
almost the same as getting the expected range min of the lcp array (since the range min is the lcp of two
indices), except that we can't represent equal (i, j) on lcp array. Therefore, we must get the expected range
min PLUS the expected lcp of a pair of indices (i, i) (that is, j == i). The expected lcp of equal indices
is obviously 1+2+...+n, so we just need to focus on the expected range min.

That can be done easily:
For any given range, you can find its minimum, and you know that every path that goes through it will
have that minimum. Therefore, you sum that amount of paths and then continue the process with the left
and right ranges (that do not include this min index). That is O(n * c), where 'c' is the cost of range
min query (log(n) with segtree).
*/

class SegmentTree {
    struct Segment {
        int min, pos, l, r;
    };

    ii NEUTRAL = {INF, INF};
    vector<Segment> t;

    void build(vector<int> &a, int v, int tl, int tr) {
        t[v] = {a[tl], tl, tl, tr};
        if (tl != tr) {
            int mid = (tl + tr) / 2;
            build(a, 2*v, tl, mid);
            build(a, 2*v+1, mid+1, tr);
            merge(v);
        }
    }

    void merge(int v) {
        if (t[2*v].min <= t[2*v+1].min) {
            t[v].min = t[2*v].min;
            t[v].pos = t[2*v].pos;
            }
            else {
                t[v].min = t[2*v+1].min;
                t[v].pos = t[2*v+1].pos;
            }
        }

        ii merge(ii &a, ii &b) {
            return min(a, b);
        }

        ii query(int v, int l, int r) {
            if (t[v].l > r || t[v].r < l)
                return NEUTRAL;
            if (t[v].l >= l && t[v].r <= r)
                return {t[v].min, t[v].pos};

            ii left = query(2*v, l, r);
            ii right = query(2*v+1, l, r);
            return merge(left, right);
        }

    public:
        SegmentTree(vector<int> &a) {
            t.resize(4*a.size());
            build(a, 1, 0, a.size()-1);
        }

        ii query(int l, int r) {
            return query(1, l, r);
        }
    };

    const int alphabet = 1e5+1;
    vector<int> sortCyclicShifts(vector<int> &s) {
        int n = s.size();
        vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
        for (int i = 0; i < n; i++)
            cnt[s[i]]++;
        for (int i = 1; i < alphabet; i++)
            cnt[i] += cnt[i-1];
        for (int i = 0; i < n; i++)
            p[--cnt[s[i]]] = i;
        c[p[0]] = 0;
        int classes = 1;
        for (int i = 1; i < n; i++) {
            if (s[p[i]] != s[p[i-1]])
                classes++;
            c[p[i]] = classes - 1;
        }
        vector<int> pn(n), cn(n);
        for (int h = 0; (1 << h) < n; ++h) {
            for (int i = 0; i < n; i++) {
                pn[i] = p[i] - (1 << h);
                if (pn[i] < 0)
                    pn[i] += n;
            }
            fill(cnt.begin(), cnt.begin() + classes, 0);
            for (int i = 0; i < n; i++)
                cnt[c[pn[i]]]++;
            for (int i = 1; i < classes; i++)
                cnt[i] += cnt[i-1];
            for (int i = n-1; i >= 0; i--)
                p[--cnt[c[pn[i]]]] = pn[i];
            cn[p[0]] = 0;
            classes = 1;
            for (int i = 1; i < n; i++) {
                pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
                pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};
                if (cur != prev)
                    ++classes;
                cn[p[i]] = classes - 1;
            }
            c.swap(cn);
            if (c[p[n-1]] == n-1) break;
        }
        return p;
    }

    vector<int> getSuffixArray(vector<int> &s) {
        s.push_back(0);
        vector<int> ans = sortCyclicShifts(s);
        s.pop_back();
        ans.erase(ans.begin());
        return ans;
    }

    vector<int> getLcp(vector<int> &s, vector<int> const &sa) {
        int n = s.size();
        int k = 0;
        vector<int> lcp(n-1, 0), rank(n, 0);
        for (int i = 0; i < n; i++)
            rank[sa[i]] = i;

        for (int i = 0; i < n; i++, k?k--:0) {
            if (rank[i] == n-1) {
                k = 0;
                continue;
            }
            int j = sa[rank[i]+1];
            while (i+k<n && j+k<n && s[i+k] == s[j+k])
                k++;
            lcp[rank[i]] = k;
        }
        return lcp;
    }

    int e = 0;

    void getExpected(int l, int r, SegmentTree &st) {
        if (l > r) return;
        auto[mini, idx] = st.query(l, r);
        // Every path that goes through idx is dominated by mini.
        // A path can start on [l, idx] and end on [idx, r].
        int amount = (idx - l + 1) * (r - idx + 1);
        e += 2 * mini * amount; // Times 2 because both i,j and j,i are valid indices with this range min
        getExpected(l, idx-1, st);
        getExpected(idx+1, r, st);
    }

    signed main() {
        cin.tie(0)->sync_with_stdio(0);

        int n;
        cin >> n;

        vector<int> v(n);
        for (int &i : v)
            cin >> i;

        if (n == 1) {
            cout << "1/1\n";
            return 0;
        }

        vector<int> lcp = getLcp(v, getSuffixArray(v));
        SegmentTree st(lcp);
        getExpected(0, lcp.size()-1, st);

        // Sum of all paths that start on equal indices (n + n-1 + n-2 + ... + 1)
        e += n*(n+1)/2;

        int d = n*n; // all possible pairs of indices
        int g = gcd(e, d);
        e /= g;
        d /= g;
        cout << e << '/' << d << '\n';

        return 0;
    }
```

21 Zolutions/Swerc 2023

21.1 Flag Performance

```
#include <bits/stdc++.h>

using namespace std;

//#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
n guys hold colored flags. An operation swaps the flags a pair of dudes hold. In how many ways can
you make each one hold its own flag in exactly K operations?
n <= 30, k <= 50

Idea:
Consider a set of cycles made by who owns what flag. For example, person 1 owns flag 3, person
3 owns flag 4, person 4 owns flag 1, and so on. That will decompose an arrangement of people-flags
into several cycles with sum of sizes equal to n. The deal is that swapping 2 flags of the same cycle
splits it, and 2 flags on different cycles merge them.
```

```
Let's do a dp. The amount of states is the amount of integer partitions p(n) and remaining movements.
p(n) for n = 30 is 5604, and the maximum amount of movements is k = 50. For each state, compute the n^2
movements you can do (one for each swapped pair) and count in how many ways you can reach the ending
state (all of the cycles with size 1). This gives a time complexity of:
p(n) * k * n^2
= 5604 * 50 * 30 * 30
= 252180000
= 2e8
Alright, but there is one last problem: our state is a vector, and we need to ensure O(1) dp lookup (else
we bring in another O(n) to the loop because of dp checking). The only way to do that is hashing the vector
and updating it while it goes down the recursion.
*/

const int MOD = 1e9+7;

int mul(int a, int b) {
    return 1ll*a*b%MOD;
}

inline int add(int a, int b) {
    a += b;
    a -= (a>=MOD)*MOD;
    return a;
}

const int MAXN = 30;
int ppowers[MAXN+1];
const int p = 31;

inline int hashVector(vector<int> &vec) {
    long long v = 0;
    int idx = 0;
    for (int i : vec)
        v = add(v, mul(i+1, ppowers[idx++]));
    return v;
}

inline int addToHash(int hash, int sz, vector<int> &szFreq) {
    szFreq[sz]++;
    return add(hash, ppowers[sz]);
}

inline int removeFromHash(int hash, int sz, vector<int> &szFreq) {
    szFreq[sz]--;
    return add(hash, MOD-ppowers[sz]);
}

const int MAXK = 50;
map<int, int> dp[MAXK+1];
int n;

int dfs(int u, vector<int> &v, vector<vector<int>> &adj) {
    v[u] = 1;
    for (int next : adj[u])
        if (!v[next])
            return dfs(next, v, adj) + 1;
    return 1;
}

int countWays(vector<int> &szFreq, int k, int v) {
    if (dp[k].count(v)) return dp[k][v];
    if (k == 0) return dp[k][v] = (szFreq[1] == n);

    vector<int> groupSizes;

    for (int i = 1; i <= n; i++) {
        int f = szFreq[i];
        while (f--)
            groupSizes.push_back(i);
    }

    int ans = 0;
    for (int i = 0; i < groupSizes.size(); i++) {
        int szi = groupSizes[i];

        //  - Compute all the answers of swaps within the same group here.
        // every swap with itself will give cycles with size E [1, groupSizes[i]-1], so we iterate
        // over them and count how many times each size appears.
        // swapping 2 indices with distance d will give one group of size 'd' and another with the
        // remaining quantity 's-d'.
        // Now, about how many times sz1 appears: an index 'i' has another index j = i+d after it
        // iff 'i' is not within the last 'd' elements. Therefore, out of all the 's' indices, only
        // s-d have distance d. Therefore, frequency of sz1 is groupSizes[i] - sz1 == sz2.

        for (int sz1 = 1; sz1 < groupSizes[i]; sz1++) {
            // splitting szi into (sz1, sz2)
            int sz2 = szi - sz1;
            v = removeFromHash(v, szi, szFreq);
            v = addToHash(v, sz1, szFreq);
            v = addToHash(v, sz2, szFreq);
            ans = add(ans, mul(countWays(szFreq, k-1, v), sz2)); // here is the multiplication by sz2
            v = addToHash(v, szi, szFreq);
            v = removeFromHash(v, sz1, szFreq);
```

```
                v = removeFromHash(v, sz2, szFreq);
            }

            //  - Compute all the answers of swaps with other groups here.
            // any pair of indices (i, j) we swap (i from current group, j from other) will end up in the
            // same cycle sizes. Therefore, there are szi*szj pairs that lead to that.

            for (int j = i+1; j < groupSizes.size(); j++) {
                // merging (szi, szj) onto newSize
                int szj = groupSizes[j];
                int newSize = szi+szj;
                v = addToHash(v, newSize, szFreq);
                v = removeFromHash(v, szi, szFreq);
                v = removeFromHash(v, szj, szFreq);
                int w = countWays(szFreq, k-1, v);
                v = removeFromHash(v, newSize, szFreq);
                v = addToHash(v, szi, szFreq);
                v = addToHash(v, szj, szFreq);
                ans = add(ans, mul(w, szi*szj));
            }
        }

        return dp[k][v] = ans;
}

signed main() {
    int k, t;
    scanf("%d %d %d", &n, &k, &t);

    ppowers[0] = 1;
    for (int i = 1; i <= MAXN; i++)
        ppowers[i] = mul(ppowers[i-1], p);

    vector<int> a(n);
    while (t--) {
        for (int i = 0; i < n; i++) {
            scanf("%d", &a[i]);
            a[i]--;
        }
        vector<vector<int>> adj(n);
        for (int i = 0; i < n; i++)
            adj[i].push_back(a[i]);

        vector<int> szFreq(n+1, 0);
        vector<int> visited(n, 0);
        for (int i = 0; i < n; i++)
            if (!visited[i])
                szFreq[dfs(i, visited, adj)]++;
        printf("%d\n", countWays(szFreq, k, hashVector(szFreq)));
    }

    return 0;
}
```

22 Zolutions/Swerc 2024

22.1 Recovering The Tablet

```
#include <bits/stdc++.h>

using namespace std;

//#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
You are given a 16x16 board with white and black cells. White cells have values in [1, 9]. Black cells
may have "vertical" or "horizontal" constraints: a vertical constraint says that the sum of the contiguous
white cells below the black cell must be the sum specified by the constraint. Same for horizontal towards
the right.
You wanna compute the board that differs the less from the given board but that is also a valid solution.
Board difference is the sum of the abs difference of their white cells.
Important: every white cell belongs to exactly 1 horizontal and 1 vertical constraint.

Idea:
Let's create a flow network, where the flow will go through iff there is an arrangement of white cells that
satisfies everything.
First of all, we will reweight the white vertices and the constraints. The exercise wants at least 1 unit
per white cell, so, we simply decrease all of them by 1 (updating constraints) and reduce our problem to
regular max flow.
Each white cell is represented by its in and out vertices. We (initially) put capacity 8 on its edge
```

```
(because the exercise says each white vertex can have a number in [1, 9] (after reweight, in [0, 8])).
Then, we plug each V constraint to the in of the white cells it reaches. We do the same for the H
constraints and the out of each white cell.
We will plug the source to the V constraints, edges with capacity equal to the constraint themselves. Same
for horizontal and sink.
It is clear that if we can make the sum of the V constraints go through it (and given that sum(V) == sum(H)),
then, there is a solution.
Now, to minimize cost: instead of a single white edge of capacity 8, we will put 2 edges, one with capacity
equal to whatever is on the board already (and cost -1 per unit of flow), and another with 8-that and cost
1 per unit of flow.
The solution is assuming you pay for all of the values of the target cells + the min cost flow. The idea is
that whatever flow goes through an edge, but less than the target, will discount its cost. Whatever goes above
it, will increase it again.
This also means you can use this technique to solve any problem where the total flow cost per edge is a
piecewise linear function (concave up).

Complexity: O(F*V*E)
F = 2048 (at most 8 per white board cell, i.e. 8*16*16)
V = 1538 (1 + V + 4*W + H + 1) = (2 + 2*16^2 + 4*16^2)
E = 1792 (V + W + 2*W + W + W + H) = (V + H + 5*W) = (2*16^2 + 5*16^2)
total = 5.644.484.608 = 56e8
But it is the real worst case of ford fulkerson AND spfa...

We will need 2 edges between w(j)in and w(j)out. Therefore, I will make two dummy vertices for each
w(j), called w(j)(a) and w(j)(b), each one holding an edge.

| LAYER 1 | LAYER 2 | LAYER 3 | LAYER 4 | LAYER 5 | LAYER 6 | LAYER 7 |
|         |    v1   |  w1in   |  w1a/b  |  w1out  |    h1   |         |
|         |    v2   |  w2in   |  w2a/b  |  w2out  |    h1   |         |
|         |    v3   |  w3in   |  w3a/b  |  w3out  |    h1   |         |
|    s    |    v4   |  w4in   |  w4a/b  |  w4out  |    h1   |    t    |
|         |    v5   |  w5in   |  w5a/b  |  w5out  |    h1   |         |
|         |    v6   |  w6in   |  w6a/b  |  w6out  |    h1   |         |
|         |    v7   |  w7in   |  w7a/b  |  w7out  |    h1   |         |
|    1    |    V    |    W    |   2*W   |    W    |    H    |    1    |
*/

// O(FVE)
struct MinCostMaxFlow {
    bool spfa(int s, int t) {
        int n = adj.size();
        d.assign(n, INF);
        vector<bool> inqueue(n, false);
        queue<int> q;
        p.assign(n, -1);

        d[s] = 0;
        q.push(s);
        inqueue[s] = true;
        while (!q.empty()) {
            int v = q.front(); q.pop();
            inqueue[v] = false;

            for (int to : adj[v]) {
                int len = cost[v][to];
                if (d[v] != INF && d[v] + len < d[to] && cap[v][to] > 0) {
                    d[to] = d[v] + len;
                    p[to] = v;
                    if (!inqueue[to]) {
                        q.push(to);
                        inqueue[to] = true;
                    }
                }
            }
        }

        return d[t] != INF;
    }

    vector<vector<int>> adj, cap, cost;
    vector<int> p, d;
    int n;

    MinCostMaxFlow(vector<vector<int>> &adj, vector<vector<int>> &cap, vector<vector<int>> &cost) : adj(adj),
        cap(cap), cost(cost) {
        n = adj.size();
    }

    // {flow, cost}
    ii maxflow(int s, int t) {
        int maxflow = 0;
        int cost = 0;

        while (spfa(s, t)) {
            int increment = INF;
            for (int u = t; p[u] >= 0; u = p[u])
                increment = min(increment, cap[p[u]][u]);

            for (int u = t; p[u] >= 0; u = p[u]) {
                cap[p[u]][u] -= increment;
                cap[u][p[u]] += increment;
                }
                cost += increment * d[t];
                maxflow += increment;
            }
            return {maxflow, cost};
        }
};

signed main() {
    cin.tie(0)->sync_with_stdio(0);

    int m, n, s;
    cin >> m >> n >> s;

    vector<vector<int>> grid(m, vector<int>(n, 0));
    vector<vector<int>> whiteIdx(m, vector<int>(n, -1));
    vector<ii> whitePosition;
    int whiteCellsSum = 0;

    for (int i = 0; i < m; i++) {
        string line;
        cin >> line;
        for (int j = 0; j < n; j++) {
            grid[i][j] = line[j]-'0';
            if (grid[i][j] != 0) {
                whiteIdx[i][j] = whitePosition.size();
                whitePosition.push_back({i, j});
                grid[i][j]--; // applying -1 white cell correction
                whiteCellsSum += grid[i][j];
            }
        }
    }

    struct Constraint { int i, j, sum; };
    vector<Constraint> v, h;
    while (s--) {
        char c;
        int i, j, sum;
        cin >> c >> i >> j >> sum;
        i--; j--;
        if (c == 'V') v.push_back({i, j, sum});
        else h.push_back({i, j, sum});
    }

    const int W = whitePosition.size();
    const int V = v.size();
    const int H = h.size();
    const int SIZE = 1 + V + W + 2*W + W + H + 1;

    vector<vector<int>> adj(SIZE);
    vector<vector<int>> cap(SIZE, vector<int>(SIZE, 0));
    vector<vector<int>> cost(SIZE, vector<int>(SIZE, 0));

    const int SOURCE = SIZE-2;
    const int SINK = SIZE-1;
    // LAYER 2:   [0, V)
    // LAYER 3:   [V, V+W)
    // LAYER 4 a: [V+W, V+2*W)
    // LAYER 4 b: [V+2*W, V+3*W)
    // LAYER 5:   [V+3*W, V+4*W)
    // LAYER 6:   [V+4*W, V+4*W+H)
    auto vid =    [&](int u) { return u; };
    auto winid =  [&](int u) { return u+V; };
    auto waid =   [&](int u) { return u+V+W; };
    auto wbid =   [&](int u) { return u+V+2*W; };
    auto woutid = [&](int u) { return u+V+3*W; };
    auto hid =    [&](int u) { return u+V+4*W; };

    auto addEdge = [&](int u, int v, int w, int c) {
        adj[u].push_back(v);
        adj[v].push_back(u);
        cost[u][v] = c;
        cost[v][u] = -c;
        cap[u][v] = w;
    };

    int vsum = 0;
    for (int i = 0; i < v.size(); i++) {
        // LAYER 2->3, vertical to win
        auto[r, c, sum] = v[i];
        int connected = 0;
        for (int j = r+1; j < m && whiteIdx[j][c] != -1; j++) {
            int idx = whiteIdx[j][c];
            addEdge(vid(i), winid(idx), sum, 0);
            connected++;
        }

        // LAYER 1->2 source to vertical
        int vEdgeCap = v[i].sum - connected;
        addEdge(SOURCE, vid(i), vEdgeCap, 0);
        vsum += vEdgeCap;
    }

    for (int i = 0; i < W; i++) {
        // LAYER 3->4, win to wa/b
        addEdge(winid(i), waid(i), INF, 0);
        addEdge(winid(i), wbid(i), INF, 0);

        // LAYER 4->5, wa/b to wout
        auto[x, y] = whitePosition[i];
        int cheapCapacity = grid[x][y];
        int expensiveCapacity = 8-cheapCapacity;
        addEdge(waid(i), woutid(i), cheapCapacity, -1);
        addEdge(wbid(i), woutid(i), expensiveCapacity, 1);
    }

    int hsum = 0;
    for (int i = 0; i < h.size(); i++) {
        // LAYER 5->6, wout to h
        auto[r, c, sum] = h[i];
        int connected = 0;
        for (int j = c+1; j < n && whiteIdx[r][j] != -1; j++) {
            int idx = whiteIdx[r][j];
            addEdge(woutid(idx), hid(i), INF, 0);
            connected++;
        }

        // LAYER 6->7, h to SINK
        int hEdgeCap = h[i].sum - connected;
        addEdge(hid(i), SINK, hEdgeCap, 0);
        hsum += hEdgeCap;
    }

    MinCostMaxFlow mf(adj, cap, cost);
    auto[flow, flowcost] = mf.maxflow(SOURCE, SINK);
    if (vsum == hsum && vsum == flow)
        cout << flowcost + whiteCellsSum << '\n';
    else
        cout << "IMPOSSIBLE\n";

    return 0;
}
```

22.2 Yaxchilan Maze

```
#include <bits/stdc++.h>

using namespace std;

#define int long long
const int INF = 0x3F3F3F3F;
using ii = pair<int, int>;
using ll = long long;
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

const bool DEBUG = 0;

/*
Exercise:
There are A archaeologists on caves from 1 to A. There are N caves. The last E caves are endings.
Some caves have wasp traps triggered only when their component has size > K. Corridors appear and
disappear from time 0 to T-1, with duration M. Compute the earliest time each archaeologist can
exit.

Idea:
Keep the connectivity, presence of traps, wasps and masks of possible archaeologist positions with
the usual offline dynamic connectivity. Do not rollback wasps and arch mask, because even tho the
components separe, the arch and wasps go to both.
The most interesting and unusual thing here is that whenever you undo a set union, you must
propagate to the subordinated the stats of the parent. Since the parent is always updated, you
simply keep another history stack of merges and copy the parent values to the subordinated.
*/

struct RollbackUnionFind {
    vector<int> parent, size, hasWasps, hasTraps, arch, ans;
    int components, k;
    stack<pair<int*, int>> history;
    stack<ii> unionsToPropagate;

    RollbackUnionFind() {}
    RollbackUnionFind(int n, int e, int a, int k, vector<int> &traps) : k(k) {
        parent.resize(n);
        iota(all(parent), 0);
        size.assign(n, 1);
        hasWasps.assign(n, 0);
        components = n;
        ans.assign(a, -1);

        arch.assign(n, 0);
        for (int i = 0; i < a; i++)
```

```
                arch[i] ^= 1ll<<i;

            hasTraps.assign(n, 0);
            for (int i : traps) {
                hasTraps[i] = 1;
                if (k <= 1)
                    hasWasps[i] = 1;
            }
        }

        int find(int v) {
            if (v == parent[v])
                return v;
            return find(parent[v]);
        }

        void change(int &x, int newVal) {
            history.push({&x, x}); // x's current state
            x = newVal;
        }

        void rollback() {
            auto[ptr, val] = history.top(); history.pop();
            *ptr = val;
        }

        void unionSets(int a, int b) {
            a = find(a);
            b = find(b);
            if (a == b) return;
            if (size[a] < size[b]) swap(a, b);
            change(parent[b], a);
            change(size[a], size[a]+size[b]);
            change(components, components-1);

            if (hasTraps[b])
                change(hasTraps[a], 1);
            if (hasTraps[a] && size[a] >= k)
                hasWasps[a] = 1; // notice this one doesn't go to rollbacks
            if (hasWasps[b])
                hasWasps[a] = 1;

            // also doesn't rollback
            arch[a] |= arch[b];
            unionsToPropagate.push({b, a});
        }
    };

    struct Edge {
        int x, y;
    };

    struct SegmentTree {
        struct Segment {
            int l, r;
            vector<Edge> edg;
        };

        vector<Segment> t;
        RollbackUnionFind uf;
        int e, n, a;

        void build(int v, int tl, int tr) {
            t[v] = {tl, tr, {}};
            if (tl != tr) {
                int mid = (tl + tr) / 2;
                build(2*v, tl, mid);
                build(2*v+1, mid+1, tr);
            }
        }

        void addEdge(int v, int l, int r, Edge const &e) {
            if (t[v].l > r || t[v].r < l)
                return;
            if (t[v].l >= l && t[v].r <= r) {
                t[v].edg.push_back(e);
                return;
            }
            addEdge(2*v, l, r, e);
            addEdge(2*v+1, l, r, e);
        }

        void dfs(int v) {
            int state = uf.history.size();
            int unionsState = uf.unionsToPropagate.size();

            for (Edge &e : t[v].edg)
                uf.unionSets(e.x, e.y);

            if (t[v].l == t[v].r) {
                // at leaf. Compute answer.
                for (int i = n-e; i < n; i++) {
                    int parent = uf.find(i);
                    if (!uf.hasWasps[parent]) {
                        int mask = uf.arch[parent];
                        for (int j = 0; j < a; j++)
                            if (mask & (1ll<<j))
                                if (uf.ans[j] == -1)
                                    uf.ans[j] = t[v].l;
                    }
                }
            }
            else {
                dfs(2*v); dfs(2*v+1);
            }

            // now manually update archeologist and wasps
            while (uf.unionsToPropagate.size() != unionsState) {
                auto[b, a] = uf.unionsToPropagate.top(); uf.unionsToPropagate.pop();
                uf.arch[b] = uf.arch[a];
                uf.hasWasps[b] = uf.hasWasps[a];
            }
            while (uf.history.size() != state) uf.rollback();
        }

        SegmentTree(int sz, int n, int e, int a, int k, vector<int> &traps) : e(e), n(n), a(a) {
            t.resize(4*sz);
            build(1, 0, sz-1);
            uf = RollbackUnionFind(n, e, a, k, traps);
        }

        void addEdge(int l, int r, Edge e) {
            addEdge(1, l, r, e);
        }

        void dfs() {
            dfs(1);
        }
    };

    signed main() {
        cin.tie(0)->sync_with_stdio(0);

        int a, n, m, e, t, b, k;
        cin >> a >> n >> m >> e >> t >> b;

        vector<int> traps(b);
        for (int &i : traps) cin >> i;
        cin >> k;
        k++; // at least k other chambers: "component of size k+1"

        SegmentTree st(t, n, e, a, k, traps);
        for (int time = 0; time < t; time++) {
            int u, v;
            cin >> u >> v;
            if (u == v) continue; // I won't connect the dude to itself
            int start = time;
            int end = min(t-1, time+m-1);
            st.addEdge(start, end, {u, v});
        }

        st.dfs();
        for (int i : st.uf.ans)
            if (i == -1) cout << "IMPOSSIBLE\n";
            else cout << i << '\n';

        return 0;
    }
```