Codebook

Wesley Leung

September 2018

Contents

1 Data Structures

1.1 General

1.1.1 Indexed Priority Queue

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Indexed Priority Queue supporting changing the key of an index
// Comparator convention is same as priority_queue in STL
// Time Complexity:
//   constructor: O(N)
//   empty, size, top, keyOf: O(1)
//   insert, remove, pop, changeKey: O(log N)
// Memory Complexity: O(N)
template <class Key, class Comparator = less<Key>> struct IndexedPQ {
    Comparator cmp; int maxN, N; vector<int> pq, qp; vector<Key> keys;
    void exch(int i, int j) { swap(pq[i], pq[j]); qp[pq[i]] = i; qp[pq[j]] = j; }
    void swim(int k) { while (k > 1 && cmp(keys[pq[k / 2]], keys[pq[k]])) { exch(k, k / 2); k /= 2; } }
    void sink(int k) {
        while (2 * k <= N) {
            int j = 2 * k;
            if (j < N && cmp(keys[pq[j]], keys[pq[j + 1]])) j++;
            if (!cmp(keys[pq[k]], keys[pq[j]])) break;
            exch(k, j); k = j;
        }
    }
    IndexedPQ(int maxN) : maxN(maxN), N(0), pq(maxN + 1), qp(maxN, -1), keys(maxN) { assert(maxN >= 0); }
    bool empty() { return N == 0; }
    bool containsIndex(int i) { assert(0 <= i && i < maxN); return qp[i] != -1; }
    int size() { return N; }
    void push(int i, Key key) {
        assert(0 <= i && i < maxN);
        if (containsIndex(i)) throw invalid_argument("index is already in the priority queue");
        N++; qp[i] = N; pq[N] = i; keys[i] = key; swim(N);
    }
    pair<int, Key> top() {
        if (N == 0) throw no_such_element_exception("Priority queue underflow");
        return make_pair(pq[1], keys[pq[1]]);
    }
    pair<int, Key> pop() {
        if (N == 0) throw no_such_element_exception("Priority queue underflow");
        int minInd = pq[1]; Key minKey = keys[minInd]; exch(1, N--); sink(1); qp[minInd] = -1;
        return make_pair(minInd, minKey);
    }
    Key keyOf(int i) {
        assert(0 <= i && i < maxN);
        if (!containsIndex(i)) throw no_such_element_exception("index is not in the priority queue");
        else return keys[i];
    }
    void changeKey(int i, Key key) {
        assert(0 <= i && i < maxN);
        if (!containsIndex(i)) throw no_such_element_exception("index is not in the priority queue");
        Key old = keys[i]; keys[i] = key;
        if (cmp(old, key)) swim(qp[i]);
        else if (cmp(key, old)) sink(qp[i]);
    }
    void remove(int i) {
        assert(0 <= i && i < maxN);
        if (!containsIndex(i)) throw no_such_element_exception("index is not in the priority queue");
        int index = qp[i]; exch(index, N--); swim(index); sink(index); qp[i] = -1;
    }
};
```

1.1.2 Skew Heap

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

// Heap supporting merges
// comparator convention is same as priority_queue in STL
// Time Complexity:
//   constructor, empty, top, size: O(1)
//   pop, push, merge: O(log N)
template <class Value, class Comparator = less<Value>> struct SkewHeap {
    Comparator cmp;
    struct Node { Value val; Node *left = nullptr, *right = nullptr; Node (Value val) : val(val) {} };
    int sz = 0; Node *root = nullptr;
    Node *merge(Node *a, Node *b) {
        if (!a || !b) return a ? a : b;
        if (cmp(a->val, b->val)) swap(a, b);
        a->right = merge(b, a->right); swap(a->left, a->right);
        return a;
    }
    SkewHeap() {}
    bool empty() { return !root; }
```

```
    Value top() { return root->val; }
    Value pop() {
        Value ret = root->val; root = merge(root->left, root->right); sz--;
        return ret;
    }
    void push(Value val) { root = merge(root, new Node(val)); sz++; }
    void merge(SkewHeap &h) { root = merge(root, h.root); sz += h.sz; }
    int size() { return sz; }
};
```

1.1.3 Incremental Skew Heap

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

// Heap supporting merges and increments
// comparator convention is same as priority_queue in STL
// Time Complexity:
//   constructor, empty, top, increment, size: O(1)
//   pop, push, merge: O(log N)
template <class Value, class Comparator = less<Value>> struct IncrementalSkewHeap {
    Comparator cmp;
    struct Node { Value val, delta = 0; Node *left = nullptr, *right = nullptr; Node (Value val) : val(val) {} };
    int sz = 0; Node *root = nullptr;
    void propagate(Node *a) {
        a->val += a->delta;
        if (a->left) a->left->delta += a->delta;
        if (a->right) a->right->delta += a->delta;
        a->delta = 0;
    }
    Node *merge(Node *a, Node *b) {
        if (!a || !b) return a ? a : b;
        propagate(a); propagate(b);
        if (cmp(a->val, b->val)) swap(a, b);
        a->right = merge(b, a->right); swap(a->left, a->right);
        return a;
    }
    IncrementalSkewHeap() {}
    bool empty() { return !root; }
    Value top() { propagate(root); return root->val; }
    Value pop() {
        propagate(root); Value ret = root->val; root = merge(root->left, root->right); sz--;
        return ret;
    }
    void push(Value val) { root = merge(root, new Node(val)); sz++; }
    void increment(Value delta) { if (root) root->delta += delta; }
    void merge(IncrementalSkewHeap &h) { root = merge(root, h.root); sz += h.sz; }
    int size() { return sz; }
};
```

1.1.4 Ordered Square Root Array

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Decomposes the array into blocks of size sqrt(n) multiplied by a factor
// The factor should be between 1 and 10, and should be smaller for large N
// Time Complexity:
//   constructor: O(N)
//   insert, erase: O(sqrt(N) + log(N))
//   pop_front: O(sqrt(N))
//   pop_back: O(1) amortized
//   front, back, empty, size: O(1)
//   at, accessor, contains, floor, ceiling, above, below: O(log(N))
//   values: O(N)
// Memory Complexity: O(N)
template <class Value, class Comparator = less<Value>>
struct OrderedSqrtArray {
    Comparator cmp; int n, SCALE_FACTOR; vector<vector<Value>> a; vector<int> prefixSZ;
    pair<int, int> ceiling_ind(const Value val) const {
        int lo = 0, hi = (int) a.size(), mid;
        while (lo < hi) {
            mid = lo + (hi - lo) / 2;
            if (cmp(a[mid].back(), val)) lo = mid + 1;
            else hi = mid;
        }
        if (lo == (int) a.size()) return {(int) a.size(), 0};
        int i = lo; lo = 0, hi = (int) a[i].size();
        while (lo < hi) {
            mid = lo + (hi - lo) / 2;
            if (cmp(a[i][mid], val)) lo = mid + 1;
            else hi = mid;
        }
        return {i, lo};
    }
    pair<int, int> floor_ind(const Value val) const {
        int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
```

```
        if (cmp(val, a[mid].front())) hi = mid - 1;
        else lo = mid + 1;
    }
    if (hi == -1) return {-1, 0};
    int i = hi; lo = 0, hi = ((int) a[i].size()) - 1;
    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (cmp(val, a[i][mid])) hi = mid - 1;
        else lo = mid + 1;
    }
    return {i, hi};
}

pair<int, int> above_ind(const Value val) const {
    int lo = 0, hi = (int) a.size(), mid;
    while (lo < hi) {
        mid = lo + (hi - lo) / 2;
        if (cmp(val, a[mid].back())) hi = mid;
        else lo = mid + 1;
    }
    if (lo == (int) a.size()) return {(int) a.size(), 0};
    int i = lo; lo = 0, hi = (int) a[i].size();
    while (lo < hi) {
        mid = lo + (hi - lo) / 2;
        if (cmp(val, a[i][mid])) hi = mid;
        else lo = mid + 1;
    }
    return {i, lo};
}
pair<int, int> below_ind(const Value val) const {
    int lo = 0, hi = ((int) a.size()) - 1, mid;
    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (cmp(a[mid].front(), val)) lo = mid + 1;
        else hi = mid - 1;
    }
    if (hi == -1) return {-1, 0};
    int i = hi; lo = 0, hi = ((int) a[i].size()) - 1;
    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (cmp(a[i][mid], val)) lo = mid + 1;
        else hi = mid - 1;
    }
    return {i, hi};
}
OrderedSqrtArray(const int SCALE_FACTOR = 1) : n(0), SCALE_FACTOR(SCALE_FACTOR) {}
template <typename It> OrderedSqrtArray(const It st, const It en, const int SCALE_FACTOR = 1) :
        n(en - st), SCALE_FACTOR(SCALE_FACTOR) {
    assert(n >= 0); assert(is_sorted(st, en, cmp));
    int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
    for (It i = st; i < en; i += sqrtn) {
        a.emplace_back(i, min(i + sqrtn, en));
        prefixSZ.push_back(0);
    }
    for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
}
OrderedSqrtArray(initializer_list<Value> il, const int SCALE_FACTOR = 1) :
        n(il.end() - il.begin()), SCALE_FACTOR(SCALE_FACTOR) {
    assert(n >= 0); assert(is_sorted(il.begin(), il.end(), cmp));
    int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
    for (auto i = il.begin(); i < il.end(); i += sqrtn) {
        a.emplace_back(i, min(i + sqrtn, il.end()));
        prefixSZ.push_back(0);
    }
    for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
}
void insert(const Value val) {
    pair<int, int> i = above_ind(val);
    if (n++ == 0) { a.emplace_back(); prefixSZ.push_back(0); }
    if (i.first == (int) a.size()) a[--i.first].push_back(val);
    else a[i.first].insert(a[i.first].begin() + i.second, val);
    int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
    if ((int) a[i.first].size() > 2 * sqrtn) {
        a.emplace(a.begin() + i.first + 1, a[i.first].begin() + sqrtn, a[i.first].end());
        a[i.first].resize(sqrtn);
        prefixSZ.push_back(0);
    }
    for (int j = i.first + 1; j < (int) a.size(); j++) prefixSZ[j] = prefixSZ[j - 1] + (int) a[j - 1].size();
}
bool erase(const Value val) {
    pair<int, int> i = ceiling_ind(val);
    if (i.first == (int) a.size() || a[i.first][i.second] != val) return false;
    --n; a[i.first].erase(a[i.first].begin() + i.second);
    if (a[i.first].empty()) { a.erase(a.begin() + i.first); prefixSZ.pop_back(); }
    for (int j = i.first + 1; j < (int) a.size(); j++) prefixSZ[j] = prefixSZ[j - 1] + (int) a[j - 1].size();
    return true;
}
void pop_front() {
    assert(n > 0); --n; a.front().erase(a.front().begin());
    if (a.front().empty()) { a.erase(a.begin()); prefixSZ.pop_back(); }
    for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
}
void pop_back() {
    assert(n > 0); --n; a.back().pop_back();
    if (a.back().empty()) { a.pop_back(); prefixSZ.pop_back(); }
}
```

```
    const Value &at(const int k) const {
        assert(0 <= k && k < n); int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        return a[hi][k - prefixSZ[hi]];
    }
    const Value &operator [](const int k) const { return at(k); }
    const Value &front() const { assert(n > 0); return a.front().front(); }
    const Value &back() const { assert(n > 0); return a.back().back(); }
    bool empty() const { return n == 0; }
    int size() const { return n; }
    bool contains(const Value val) const {
        pair<int, int> i = ceiling_ind(val);
        return i.first != (int) a.size() && a[i.first][i.second] == val;
    }
    pair<int, Value> floor(const Value val) const {
        pair<int, int> i = floor_ind(val);
        if (i.first == -1) throw no_such_element_exception("call to floor() resulted in no such value");
        return {prefixSZ[i.first] + i.second, a[i.first][i.second]};
    }
    pair<int, Value> ceiling(const Value val) const {
        pair<int, int> i = ceiling_ind(val);
        if (i.first == (int) a.size()) throw no_such_element_exception("call to ceiling() resulted in no such value");
        return {prefixSZ[i.first] + i.second, a[i.first][i.second]};
    }
    pair<int, Value> above(const Value val) const {
        pair<int, int> i = above_ind(val);
        if (i.first == (int) a.size()) throw no_such_element_exception("call to above() resulted in no such value");
        return {prefixSZ[i.first] + i.second, a[i.first][i.second]};
    }
    pair<int, Value> below(const Value val) const {
        pair<int, int> i = below_ind(val);
        if (i.first == -1) throw no_such_element_exception("call to below() resulted in no such value");
        return {prefixSZ[i.first] + i.second, a[i.first][i.second]};
    }
    vector<Value> values() const {
        vector<Value> ret;
        for (auto &&ai : a) for (auto &&aij : ai) ret.push_back(aij);
        return ret;
    }
};
```

1.1.5 Ordered Root Array

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Decomposes the array into N ^ (1 / R) containers of size N ^ ((R - 1) / R) multiplied by a factor
// The factor should be between 1 and 10, and should be smaller for large N
// Time Complexity:
//   constructor: O(N)
//   insert, erase: O(N ^ (1 / R) + log(N))
//   pop_front: O(N ^ (1 / R))
//   pop_back: O(1) amortized
//   front, back, empty, size: O(1)
//   at, accessor, contains, floor, ceiling, above, below: O(log(N))
//   values: O(N)
// Memory Complexity: O(N)
template <const int R, class Value, class Container, class Comparator = less<Value>>
struct OrderedRootArray {
    Comparator cmp; int n, SCALE_FACTOR; vector<Container> a; vector<int> prefixSZ;
    int ceiling_ind(const Value val) const {
        int lo = 0, hi = (int) a.size(), mid;
        while (lo < hi) {
            mid = lo + (hi - lo) / 2;
            if (cmp(a[mid].back(), val)) lo = mid + 1;
            else hi = mid;
        }
        return lo;
    }
    int floor_ind(const Value val) const {
        int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (cmp(val, a[mid].front())) hi = mid - 1;
            else lo = mid + 1;
        }
        return hi;
    }
    int above_ind(const Value val) const {
        int lo = 0, hi = (int) a.size(), mid;
        while (lo < hi) {
            mid = lo + (hi - lo) / 2;
            if (cmp(val, a[mid].back())) hi = mid;
            else lo = mid + 1;
        }
        return lo;
```

```
}
int below_ind(const Value val) const {
    int lo = 0, hi = ((int) a.size()) - 1, mid;
    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (cmp(a[mid].front(), val)) lo = mid + 1;
        else hi = mid - 1;
    }
    return hi;
}
OrderedRootArray(const int SCALE_FACTOR = 1) : n(0), SCALE_FACTOR(SCALE_FACTOR) {}
template <typename It> OrderedRootArray(const It st, const It en, const int SCALE_FACTOR = 1) :
        n(en - st), SCALE_FACTOR(SCALE_FACTOR) {
    assert(n >= 0); assert(is_sorted(st, en, cmp));
    int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
    for (It i = st; i < en; i += rootn) {
        a.emplace_back(i, min(i + rootn, en), SCALE_FACTOR);
        prefixSZ.push_back(0);
    }
    for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
}
OrderedRootArray(initializer_list<Value> il, const int SCALE_FACTOR = 1) :
        n(il.end() - il.begin()), SCALE_FACTOR(SCALE_FACTOR) {
    assert(n >= 0); assert(is_sorted(il.begin(), il.end(), cmp));
    int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
    for (auto i = il.begin(); i < il.end(); i += rootn) {
        a.emplace_back(i, min(i + rootn, il.end()), SCALE_FACTOR);
        prefixSZ.push_back(0);
    }
    for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
}
void insert(const Value val) {
    int i = above_ind(val);
    if (n++ == 0) { a.emplace_back(SCALE_FACTOR); prefixSZ.push_back(0); }
    if (i == (int) a.size()) a[--i].insert(val);
    else a[i].insert(val);
    int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
    if ((int) a[i].size() > 2 * rootn) {
        vector<Value> b;
        while (a[i].size() > rootn) { b.push_back(a[i].back()); a[i].pop_back(); }
        reverse(b.begin(), b.end()); a.emplace(a.begin() + i + 1, b.begin(), b.end(), SCALE_FACTOR); prefixSZ.push_back(0);
    }
    for (int j = i + 1; j < (int) a.size(); j++) prefixSZ[j] = prefixSZ[j - 1] + (int) a[j - 1].size();
}
bool erase(const Value val) {
    int i = ceiling_ind(val);
    if (i == (int) a.size()) return false;
    if (!a[i].erase(val)) return false;
    --n;
    if (a[i].empty()) { a.erase(a.begin() + i); prefixSZ.pop_back(); }
    for (int j = i + 1; j < (int) a.size(); j++) prefixSZ[j] = prefixSZ[j - 1] + (int) a[j - 1].size();
    return true;
}
void pop_front() {
    assert(n > 0); --n; a.front().pop_front();
    if (a.front().empty()) { a.erase(a.begin()); prefixSZ.pop_back(); }
    for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
}
void pop_back() {
    assert(n > 0); --n; a.back().pop_back();
    if (a.back().empty()) { a.pop_back(); prefixSZ.pop_back(); }
}
const Value &at(const int k) const {
    assert(0 <= k && k < n); int lo = 0, hi = ((int) a.size()) - 1, mid;
    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (k < prefixSZ[mid]) hi = mid - 1;
        else lo = mid + 1;
    }
    return a[hi].at(k - prefixSZ[hi]);
}
const Value &operator [](const int k) const { return at(k); }
const Value &front() const { assert(n > 0); return a.front().front(); }
const Value &back() const { assert(n > 0); return a.back().back(); }
bool empty() const { return n == 0; }
int size() const { return n; }
bool contains(const Value val) const {
    int i = ceiling_ind(val);
    return i != (int) a.size() && a[i].contains(val);
}
pair<int, Value> floor(const Value val) const {
    int i = floor_ind(val);
    if (i == -1) throw no_such_element_exception("call to floor() resulted in no such value");
    pair<int, Value> j = a[i].floor(val);
    return {prefixSZ[i] + j.first, j.second};
}
pair<int, Value> above(const Value val) const {
    int i = above_ind(val);
    if (i == (int) a.size()) throw no_such_element_exception("call to above() resulted in no such value");
    pair<int, Value> j = a[i].above(val);
    return {prefixSZ[i] + j.first, j.second};
}
pair<int, Value> below(const Value val) const {
    int i = below_ind(val);
    if (i == -1) throw no_such_element_exception("call to below() resulted in no such value");
    pair<int, Value> j = a[i].below(val);
```

```
        return {prefixSZ[i] + j.first, j.second};
    }
    pair<int, Value> ceiling(const Value val) const {
        int i = ceiling_ind(val);
        if (i == (int) a.size()) throw no_such_element_exception("call to ceiling() resulted in no such value");
        pair<int, Value> j = a[i].ceiling(val);
        return {prefixSZ[i] + j.first, j.second};
    }
    vector<Value> values() const {
        vector<Value> ret;
        for (auto &&ai : a) for (auto &&aij : ai.values()) ret.push_back(aij);
        return ret;
    }
};
```

1.1.6 Square Root Array

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Decomposes the array into blocks of size sqrt(n) multiplied by a factor
// The factor should be between 1 and 10, and should be smaller for large N
// Time Complexity:
//   constructor: O(N)
//   insert, erase: O(sqrt(N) + log(N))
//   push_front, pop_front: O(sqrt(N))
//   push_back, pop_back: O(1) amortized
//   front, back, empty, size: O(1)
//   at, accessor: O(log(N))
//   values: O(N)
// Memory Complexity: O(N)
template <class Value> struct SqrtArray {
    int n, SCALE_FACTOR; vector<vector<Value>> a; vector<int> prefixSZ;
    SqrtArray(const int SCALE_FACTOR = 1) : n(0), SCALE_FACTOR(SCALE_FACTOR) {}
    SqrtArray(const int n, const int SCALE_FACTOR) : n(n), SCALE_FACTOR(SCALE_FACTOR) {
        assert(n >= 0); int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
        for (int i = 0; i < n; i += sqrtn) { a.emplace_back(min(sqrtn, n - i)); prefixSZ.push_back(0); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    template <typename It> SqrtArray(const It st, const It en, const int SCALE_FACTOR = 1) : n(en - st), SCALE_FACTOR(SCALE_FACTOR)
         {
        assert(n >= 0); int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
        for (It i = st; i < en; i += sqrtn) { a.emplace_back(i, min(i + sqrtn, en)); prefixSZ.push_back(0); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    SqrtArray(initializer_list<Value> il, const int SCALE_FACTOR = 1) : n(il.end() - il.begin()), SCALE_FACTOR(SCALE_FACTOR) {
        assert(n >= 0); int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
        for (auto i = il.begin(); i < il.end(); i += sqrtn) { a.emplace_back(i, min(i + sqrtn, il.end())); prefixSZ.push_back(0); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void insert(int k, const Value val) { // inserts value before kth index
        assert(0 <= k && k <= n);
        if (n++ == 0) { a.emplace_back(); prefixSZ.push_back(0); }
        int lo = 0, hi = (int) (a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        k -= prefixSZ[hi]; int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
        if (hi == -1) a[hi += (int) a.size()].push_back(val);
        else a[hi].insert(a[hi].begin() + k, val);
        if ((int) a[hi].size() > 2 * sqrtn) {
            a.emplace(a.begin() + hi + 1, a[hi].begin() + sqrtn, a[hi].end()); a[hi].resize(sqrtn); prefixSZ.push_back(0);
        }
        for (int i = hi + 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void push_front(const Value val) {
        if (n++ == 0) { a.emplace_back(); prefixSZ.push_back(0); }
        a.front().insert(a.front().begin(), val); int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
        if ((int) a.front().size() > 2 * sqrtn) {
            a.emplace(a.begin() + 1, a.front().begin() + sqrtn, a.front().end()); a.front().resize(sqrtn); prefixSZ.push_back(0);
        }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void push_back(const Value val) {
        if (n++ == 0) { a.emplace_back(); prefixSZ.push_back(0); }
        a.back().push_back(val); int sqrtn = (int) sqrt(n) * SCALE_FACTOR;
        if ((int) a.back().size() > 2 * sqrtn) {
            a.emplace_back(a.back().begin() + sqrtn, a.back().end()); a[(int) a.size() - 2].resize(sqrtn);
            prefixSZ.push_back(prefixSZ[(int) a.size() - 2] + (int) a[(int) a.size() - 2].size());
        }
    }
    void erase(const int k) {
        assert(0 <= k && k < n); --n; int lo = 0, hi = (int) (a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
```

```
        a[hi].erase(a[hi].begin() + k - prefixSZ[hi]);
        if (a[hi].empty()) { a.erase(a.begin() + hi); prefixSZ.pop_back(); }
        for (int i = hi + 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void pop_front() {
        assert(n > 0); --n; a.front().erase(a.front().begin());
        if (a.front().empty()) { a.erase(a.begin()); prefixSZ.pop_back(); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void pop_back() {
        assert(n > 0); --n; a.back().pop_back();
        if (a.back().empty()) { a.pop_back(); prefixSZ.pop_back(); }
    }
    Value &operator [](int k) {
        assert(0 <= k && k < n); int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        return a[hi][k - prefixSZ[hi]];
    }
    const Value &at(const int k) const {
        assert(0 <= k && k < n); int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        return a[hi][k - prefixSZ[hi]];
    }
    const Value &operator [](const int k) const { return at(k); }
    const Value &front() const { assert(n > 0); return a.front().front(); }
    const Value &back() const { assert(n > 0); return a.back().back(); }
    bool empty() const { return n == 0; }
    int size() const { return n; }
    vector<Value> values() const {
        vector<Value> ret;
        for (auto &&ai : a) for (auto &&aij : ai) ret.push_back(aij);
        return ret;
    }
};
```

1.1.7 Root Array

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Decomposes the array into N ^ (1 / R) containers of size N ^ ((R - 1) / R) multiplied by a factor
// The factor should be between 1 and 10, and should be smaller for large N
// Time Complexity:
//   constructor: O(N)
//   insert, erase: O(N ^ (1 / R) + log(N))
//   push_front, pop_front: O(N ^ (1 / R))
//   push_back, pop_back: O(1) amortized
//   front, back, empty, size: O(1)
//   at, accessor: O(log(N))
//   values: O(N)
// Memory Complexity: O(N)
template <const int R, class Value, class Container> struct RootArray {
    int n, SCALE_FACTOR; vector<Container> a; vector<int> prefixSZ;
    RootArray(const int SCALE_FACTOR = 1) : n(0), SCALE_FACTOR(SCALE_FACTOR) {}
    RootArray(const int n, const int SCALE_FACTOR) : n(n), SCALE_FACTOR(SCALE_FACTOR) {
        assert(n >= 0); int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
        for (int i = 0; i < n; i += rootn) { a.emplace_back(min(rootn, n - i)); prefixSZ.push_back(0); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    template <typename It> RootArray(const It st, const It en, const int SCALE_FACTOR = 1) : n(en - st), SCALE_FACTOR(SCALE_FACTOR)
        {
        assert(n >= 0); int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
        for (It i = st; i < en; i += rootn) { a.emplace_back(i, min(i + rootn, st + n), SCALE_FACTOR); prefixSZ.push_back(0); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    RootArray(initializer_list<Value> il, const int SCALE_FACTOR = 1) : n(il.end() - il.begin()), SCALE_FACTOR(SCALE_FACTOR) {
        assert(n >= 0); int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
        for (auto i = il.begin(); i < il.end(); i += rootn) {
            a.emplace_back(i, min(i + rootn, il.end()), SCALE_FACTOR);
            prefixSZ.push_back(0);
        }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void insert(int k, const Value val) { // inserts value before kth index
        assert(0 <= k && k <= n);
        if (n++ == 0) { a.emplace_back(SCALE_FACTOR); prefixSZ.push_back(0); }
        int lo = 0, hi = (int) (a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
```

```
        k -= prefixSZ[hi]; int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
        if (hi == -1) a[hi += (int) a.size()].push_back(val);
        else a[hi].insert(k, val);
        if ((int) a[hi].size() > 2 * rootn) {
            vector<Value> b;
            while (a[hi].size() > rootn) { b.push_back(a[hi].back()); a[hi].pop_back(); }
            reverse(b.begin(), b.end()); a.emplace(a.begin() + hi + 1, b.begin(), b.end(), SCALE_FACTOR); prefixSZ.push_back(0);
        }
        for (int i = hi + 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void push_front(const Value val) {
        if (n++ == 0) { a.emplace_back(SCALE_FACTOR); prefixSZ.push_back(0); }
        a.front().push_front(val);
        int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
        if ((int) a.front().size() > 2 * rootn) {
            vector<Value> b;
            while (a.front().size() > rootn) { b.push_back(a.front().back()); a.front().pop_back(); }
            reverse(b.begin(), b.end()); a.emplace(a.begin() + 1, b.begin(), b.end(), SCALE_FACTOR); prefixSZ.push_back(0);
        }
    }
    void push_back(const Value val) {
        if (n++ == 0) { a.emplace_back(SCALE_FACTOR); prefixSZ.push_back(0); }
        a.back().push_back(val);
        int rootn = (int) pow(n, (double) (R - 1) / R) * SCALE_FACTOR;
        if ((int) a.back().size() > 2 * rootn) {
            vector<Value> b;
            while (a.back().size() > rootn) { b.push_back(a.back().back()); a.back().pop_back(); }
            reverse(b.begin(), b.end()); a.emplace_back(b.begin(), b.end(), SCALE_FACTOR);
            prefixSZ.push_back(prefixSZ[(int) a.size() - 2] + (int) a[(int) a.size() - 2].size());
        }
    }
    void erase(const int k) {
        assert(0 <= k && k < n); --n; int lo = 0, hi = (int) (a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        a[hi].erase(k - prefixSZ[hi]);
        if (a[hi].empty()) { a.erase(a.begin() + hi); prefixSZ.pop_back(); }
        for (int i = hi + 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void pop_front() {
        assert(n > 0); --n; a.front().pop_front();
        if (a.front().empty()) { a.erase(a.begin()); prefixSZ.pop_back(); }
        for (int i = 1; i < (int) a.size(); i++) prefixSZ[i] = prefixSZ[i - 1] + (int) a[i - 1].size();
    }
    void pop_back() {
        assert(n > 0); --n; a.back().pop_back();
        if (a.back().empty()) { a.pop_back(); prefixSZ.pop_back(); }
    }
    Value &operator [](int k) {
        assert(0 <= k && k < n); int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        return a[hi][k - prefixSZ[hi]];
    }
    const Value &at(const int k) const {
        assert(0 <= k && k < n); int lo = 0, hi = ((int) a.size()) - 1, mid;
        while (lo <= hi) {
            mid = lo + (hi - lo) / 2;
            if (k < prefixSZ[mid]) hi = mid - 1;
            else lo = mid + 1;
        }
        return a[hi].at(k - prefixSZ[hi]);
    }
    const Value &operator [](const int k) const { return at(k); }
    const Value &front() const { assert(n > 0); return a.front().front(); }
    const Value &back() const { assert(n > 0); return a.back().back(); }
    bool empty() const { return n == 0; }
    int size() const { return n; }
    vector<Value> values() const {
        vector<Value> ret;
        for (auto &&ai : a) for (auto &&aij : ai.values()) ret.push_back(aij);
        return ret;
    }
};
```

1.1.8 Radix Heap

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

// 32-bit Radix Heap
// Comparator convention is same as priority_queue in STL
// Time Complexity:
//   top: O(B) where B is the number of bits
//   empty, size, pop, push: O(1)
template <class T, class Comparator = less<uint32_t>> struct RadixHeap {
    Comparator cmp; int n; uint32_t last; vector<pair<uint32_t, T>> x[33];
    int bsr(uint32_t a) { return a ? 31 - __builtin_clz(a) : -1; }
    void aux(const pair<uint32_t, T> &p) { x[bsr(p.first ^ last) + 1].push_back(p); }
```

```
    RadixHeap() : n(0), last(0) {}
    bool empty() const { return 0 == n; }
    int size() const { return n; }
    pair<uint32_t, T> top() {
        assert(n > 0);
        if (x[0].empty()) {
            int i = 1;
            while (x[i].empty()) ++i;
            last = x[i][0].first;
            for (int j = 1; j < int(x[i].size()); j++) if (cmp(last, x[i][j].first)) last = x[i][j].first;
            for (auto &&p : x[i]) aux(p);
            x[i].clear();
        }
        return x[0].back();
    }
    void pop() { assert(n > 0); top(); n--; x[0].pop_back(); }
    void push(uint32_t key, T value) { n++; aux({key, value}); }
};
```

1.1.9 Radix Heap 64

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

// 64-bit Radix Heap
// Comparator convention is same as priority_queue in STL
// Time Complexity:
//   top: O(B) where B is the number of bits
//   empty, size, pop, push: O(1)
template <class T, class Comparator = less<uint64_t>> struct RadixHeap64 {
    Comparator cmp; int n; uint64_t last; vector<pair<uint64_t, T>> x[65];
    int bsr(uint64_t a) { return a ? 63 - __builtin_clzll(a) : -1; }
    void aux(const pair<uint64_t, T> &p) { x[bsr(p.first ^ last) + 1].push_back(p); }
    RadixHeap64() : n(0), last(0) {}
    bool empty() const { return 0 == n; }
    int size() const { return n; }
    pair<uint64_t, T> top() {
        assert(n > 0);
        if (x[0].empty()) {
            int i = 1;
            while (x[i].empty()) ++i;
            last = x[i][0].first;
            for (int j = 1; j < int(x[i].size()); j++) if (cmp(last, x[i][j].first)) last = x[i][j].first;
            for (auto &&p : x[i]) aux(p);
            x[i].clear();
        }
        return x[0].back();
    }
    void pop() { assert(n > 0); top(); n--; x[0].pop_back(); }
    void push(uint64_t key, T value) { n++; aux({key, value}); }
};
```

1.1.10 RMQ Sparse Table

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

// Sparse Table for Range Minimum Queries
// Time Complexity:
//   init: O(N log N)
//   query: O(1)
template <const int MAXN, const int LGN, class T> struct RMQSparseTable {
    T A[MAXN]; int ST[LGN][MAXN];
    int minInd(int l, int r) {  return A[l] <= A[r] ? l : r; }
    void init(int N) {
        int lg = 32 - __builtin_clz(N); iota(ST[0], ST[0] + N, 0);
        for (int i = 0; i < lg - 1; i++) for (int j = 0; j + (1 << i) < N; j++)
            ST[i + 1][j] = minInd(ST[i][j], ST[i][j + (1 << i)]);
    }
    int query(int l, int r) { // zero-indexed, inclusive
        int i = 31 - __builtin_clz(r - l + 1); l = ST[i][l]; r = ST[i][r - (1 << i) + 1];
        return minInd(l, r);
    }
};
```

1.1.11 Union Find

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

// Union Find / Disjoint Sets with Half Compression
// Time Complexity:
//   init: O(N)
//   find, join, connected, getSize: Inverse Ackerman
// Memory Complexity: O(N)
template <const int MAXN> struct UnionFind {
    int UF[MAXN], cnt;
    void init(int N) { cnt = N; fill(UF, UF + MAXN, -1); }
    int find(int v) { return UF[v] < 0 ? v : UF[v] = find(UF[v]); }
    bool join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return false;
        if (UF[v] > UF[w]) swap(v, w);
        UF[v] += UF[w]; UF[w] = v; cnt--;
```

```
        return true;
    }
    bool connected(int v, int w) { return find(v) == find(w); }
    int getSize(int v) { return -UF[find(v)]; }
};
```

1.1.12 Union Find Undo

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

// Union Find / Disjoint Sets supporting undos
// Time Complexity:
//   init: O(N)
//   find, join, connected, getSize: O(log N)
//   undo: O(1)
// Memory Complexity: O(N)
template <const int MAXN> struct UnionFindUndo {
    int UF[MAXN], cnt; stack<pair<pair<int, int>, int>> history;
    void init(int N) { cnt = N; fill(UF, UF + MAXN, -1); }
    int find(int v) { while (UF[v] >= 0) v = UF[v]; return v; }
    bool join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return false;
        if (UF[v] > UF[w]) swap(v, w);
        history.push({{v, w}, UF[w]}); UF[v] += UF[w]; UF[w] = v; cnt--;
        return true;
    }
    void undo() {
        int v = history.top().first.first, w = history.top().first.second, ufw = history.top().second;
        history.pop(); UF[w] = ufw; UF[v] -= UF[w]; cnt++;
    }
    bool connected(int v, int w) { return find(v) == find(w); }
    int getSize(int v) { return -UF[find(v)]; }
};
```

1.2 Trees

1.2.1 Size Balanced Tree

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Size Balanced Binary Search Set
// Time Complexity:
//   constructor, empty, size: O(1)
//   keyValuePairs: O(N)
//   all other operators: O(log N)
// Memory Complexity: O(N)
template <typename Key, typename Value, typename Comparator = less<Key>> struct SBT {
    Comparator cmp; vector<Key> KEY; vector<Value> VAL; vector<int> SZ, L, R; int root = 0;
    void update(int x) { SZ[x] = 1 + SZ[L[x]] + SZ[R[x]]; }
    int rotateRight(int x) { int y = L[x]; L[x] = R[y]; R[y] = x; update(x); update(y); return y; }
    int rotateLeft(int x) { int y = R[x]; R[x] = L[y]; L[y] = x; update(x); update(y); return y; }
    int maintain(int x, bool flag) {
        if (flag) {
            if (SZ[L[x]] < SZ[L[R[x]]]) { R[x] = rotateRight(R[x]); x = rotateLeft(x); }
            else if (SZ[L[x]] < SZ[R[R[x]]]) x = rotateLeft(x);
            else return x;
        } else {
            if (SZ[R[x]] < SZ[R[L[x]]]) { L[x] = rotateLeft(L[x]); x = rotateRight(x); }
            else if (SZ[R[x]] < SZ[L[L[x]]]) x = rotateRight(x);
            else return x;
        }
        L[x] = maintain(L[x], false); R[x] = maintain(R[x], true); x = maintain(x, true); x = maintain(x, false);
        return x;
    }
    int get(int x, Key key) {
        if (!x) return 0;
        if (cmp(key, KEY[x])) return get(L[x], key);
        else if (cmp(KEY[x], key)) return get(R[x], key);
        else return x;
    }
    int put(int x, Key key, Value val) {
        if (!x) {
            if (KEY.empty()) KEY.push_back(key);
            if (VAL.empty()) VAL.push_back(val);
            KEY.push_back(key); VAL.push_back(val); SZ.push_back(1); L.push_back(0); R.push_back(0);
            return int(KEY.size()) - 1;
        }
        if (cmp(key, KEY[x])) { int l = put(L[x], key, val); L[x] = l; }
        else if (cmp(KEY[x], key)) { int r = put(R[x], key, val); R[x] = r; }
        else { VAL[x] = val; return x; }
        update(x); return maintain(x, key > KEY[x]);
    }
    int removeMin(int x) {
        if (!L[x]) return R[x];
        L[x] = removeMin(L[x]); update(x); return x;
```

```
}
int removeMax(int x) {
    if (!R[x]) return L[x];
    R[x] = removeMax(R[x]); update(x); return x;
}
int getMin(int x) { return L[x] ? getMin(L[x]) : x; }
int getMax(int x) { return R[x] ? getMax(R[x]) : x; }
int remove(int x, Key key) {
    if (cmp(key, KEY[x])) L[x] = remove(L[x], key);
    else if (cmp(KEY[x], key)) R[x] = remove(R[x], key);
    else {
        if (!L[x]) return R[x];
        else if (!R[x]) return L[x];
        else { int y = x; x = getMin(R[y]); R[x] = removeMin(R[y]); L[x] = L[y]; }
    }
    update(x); return x;
}
int floor(int x, Key key) {
    if (!x) return 0;
    if (!cmp(key, KEY[x]) && !cmp(KEY[x], key)) return x;
    if (cmp(key, KEY[x])) return floor(L[x], key);
    int y = floor(R[x], key); return y ? y : x;
}
int ceiling(int x, Key key) {
    if (!x) return 0;
    if (!cmp(key, KEY[x]) && !cmp(KEY[x], key)) return x;
    if (cmp(KEY[x], key)) return ceiling(R[x], key);
    int y = ceiling(L[x], key); return y ? y : x;
}
int select(int x, int k) {
    if (!x) return 0;
    int t = SZ[L[x]];
    if (t > k) return select(L[x], k);
    else if (t < k) return select(R[x], k - t - 1);
    return x;
}
int getRank(int x, Key key) {
    if (!x) return 0;
    if (cmp(key, KEY[x])) return getRank(KEY[x], key);
    else if (cmp(KEY[x], key)) return 1 + SZ[L[x]] + getRank(R[x], key);
    else return SZ[L[x]];
}
void keyValuePairsInOrder(int x, vector<pair<Key, Value>> &queue) {
    if (!x) return;
    keyValuePairsInOrder(L[x], queue); queue.push_back({KEY[x], VAL[x]}); keyValuePairsInOrder(R[x], queue);
}
void keyValuePairs(int x, vector<pair<Key, Value>> &queue, Key lo, Key hi) {
    if (!x) return;
    if (cmp(lo, KEY[x])) keyValuePairs(L[x], queue, lo, hi);
    if (!cmp(KEY[x], lo) && !cmp(hi, KEY[x])) queue.push_back({KEY[x], VAL[x]});
    if (cmp(KEY[x], hi)) keyValuePairs(R[x], queue, lo, hi);
}
SBT() : SZ(1), L(1), R(1) {}
void clear() {
    KEY.clear(); VAL.clear(); SZ.clear(); L.clear(); R.clear();
    SZ.push_back(0); L.push_back(0); R.push_back(0);
}
bool empty() { return root == 0; }
int size() { return SZ[root]; }
Value get(Key key) {
    int x = get(root, key);
    if (!x) throw no_such_element_exception("no such key is in the symbol table");
    return VAL[x];
}
bool contains(Key key) { return get(root, key) != 0; }
void put(Key key, Value val) { root = put(root, key, val); }
void remove(Key key) { if (contains(key)) root = remove(root, key); }
void removeMin() {
    if (empty()) throw runtime_error("called removeMin() with empty symbol table");
    root = removeMin(root);
}
void removeMax() {
    if (empty()) throw runtime_error("called removeMax() with empty symbol table");
    root = removeMax(root);
}
pair<Key, Value> getMin() {
    if (empty()) throw runtime_error("called getMin() with empty symbol table");
    int x = getMin(root); return {KEY[x], VAL[x]};
}
pair<Key, Value> getMax() {
    if (empty()) throw runtime_error("called getMax() with empty symbol table");
    int x = getMax(root); return {KEY[x], VAL[x]};
}
pair<Key, Value> floor(Key key) {
    if (empty()) throw runtime_error("called floor() with empty symbol table");
    int x = floor(root, key);
    if (!x) throw no_such_element_exception("call to floor() resulted in no such value");
    return {KEY[x], VAL[x]};
}
pair<Key, Value> ceiling(Key key) {
    if (empty()) throw runtime_error("called ceiling() with empty symbol table");
    int x = ceiling(root, key);
    if (!x) throw no_such_element_exception("call to ceiling() resulted in no such value");
    return {KEY[x], VAL[x]};
}
pair<Key, Value> select(int k) {
```

```
        if (k < 0 || k >= size()) throw invalid_argument("k is not in range 0 to size");
        int x = select(root, k); return {KEY[x], VAL[x]};
    }
    int getRank(Key key) { return getRank(root, key); }
    vector<pair<Key, Value>> keyValuePairs() {
        vector<pair<Key, Value>> queue; keyValuePairsInOrder(root, queue);
        return queue;
    }
    vector<pair<Key, Value>> keyValuePairs(Key lo, Key hi) {
        vector<pair<Key, Value>> queue; keyValuePairs(root, queue, lo, hi);
        return queue;
    }
    int size(Key lo, Key hi) {
        if (cmp(hi, lo)) return 0;
        if (contains(hi)) return getRank(hi) - getRank(lo) + 1;
        else return getRank(hi) - getRank(lo);
    }
};
```

1.2.2 Size Balanced Tree Set

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

class no_such_element_exception: public runtime_error {
public:
    no_such_element_exception(): runtime_error("No such element exists"){}
    no_such_element_exception(string message): runtime_error(message){}
};

// Size Balanced Binary Search Set
// Time Complexity:
//   constructor, empty, size: O(1)
//   values: O(N)
//   all other operators: O(log N)
// Memory Complexity: O(N)
template <typename Value, typename Comparator = less<Value>> struct SBTSet {
    Comparator cmp; vector<Value> VAL; vector<int> SZ, L, R; int root = 0;
    void update(int x) { SZ[x] = 1 + SZ[L[x]] + SZ[R[x]]; }
    int rotateRight(int x) { int y = L[x]; L[x] = R[y]; R[y] = x; update(x); update(y); return y; }
    int rotateLeft(int x) { int y = R[x]; R[x] = L[y]; L[y] = x; update(x); update(y); return y; }
    int maintain(int x, bool flag) {
        if (flag) {
            if (SZ[L[x]] < SZ[L[R[x]]]) { R[x] = rotateRight(R[x]); x = rotateLeft(x); }
            else if (SZ[L[x]] < SZ[R[R[x]]]) x = rotateLeft(x);
            else return x;
        } else {
            if (SZ[R[x]] < SZ[R[L[x]]]) { L[x] = rotateLeft(L[x]); x = rotateRight(x); }
            else if (SZ[R[x]] < SZ[L[L[x]]]) x = rotateRight(x);
            else return x;
        }
        L[x] = maintain(L[x], false); R[x] = maintain(R[x], true); x = maintain(x, true); x = maintain(x, false);
        return x;
    }
    bool contains(int x, Value val) {
        if (!x) return false;
        else if (cmp(val, VAL[x])) return contains(L[x], val);
        else if (cmp(VAL[x], val)) return contains(R[x], val);
        return true;
    }
    int add(int x, Value val) {
        if (!x) {
            if (VAL.empty()) VAL.push_back(val);
            VAL.push_back(val); SZ.push_back(1); L.push_back(0); R.push_back(0);
            return int(VAL.size()) - 1;
        }
        if (cmp(val, VAL[x])) { int l = add(L[x], val); L[x] = l; }
        else { int r = add(R[x], val); R[x] = r; }
        update(x); return maintain(x, !cmp(val, VAL[x]));
    }
    int removeMin(int x) {
        if (!L[x]) return R[x];
        L[x] = removeMin(L[x]); update(x); return x;
    }
    int removeMax(int x) {
        if (!R[x]) return L[x];
        R[x] = removeMax(R[x]); update(x); return x;
    }
    int getMin(int x) { return L[x] ? getMin(L[x]) : x; }
    int getMax(int x) { return R[x] ? getMax(R[x]) : x; }
    int remove(int x, Value val) {
        if (cmp(val, VAL[x])) L[x] = remove(L[x], val);
        else if (cmp(VAL[x], val)) R[x] = remove(R[x], val);
        else {
            if (!L[x]) return R[x];
            else if (!R[x]) return L[x];
            else { int y = x; x = getMin(R[y]); R[x] = removeMin(R[y]); L[x] = L[y]; }
        }
        update(x); return x;
    }
    int floor(int x, Value val) {
        if (!x) return 0;
        if (!cmp(val, VAL[x]) && !cmp(VAL[x], val)) return x;
        if (cmp(val, VAL[x])) return floor(L[x], val);
        int y = floor(R[x], val); return y ? y : x;
    }
```

```
    int ceiling(int x, Value val) {
        if (!x) return 0;
        if (!cmp(val, VAL[x]) && !cmp(VAL[x], val)) return x;
        if (cmp(VAL[x], val)) return ceiling(R[x], val);
        int y = ceiling(L[x], val); return y ? y : x;
    }
    int select(int x, int k) {
        if (!x) return 0;
        int t = SZ[L[x]];
        if (t > k) return select(L[x], k);
        else if (t < k) return select(R[x], k - t - 1);
        return x;
    }
    int getRank(int x, Value val) {
        if (!x) return 0;
        if (!cmp(VAL[x], val)) return getRank(L[x], val);
        else return 1 + SZ[L[x]] + getRank(R[x], val);
    }
    void valuesInOrder(int x, vector<Value> &queue) {
        if (!x) return;
        valuesInOrder(L[x], queue); queue.push_back(VAL[x]); valuesInOrder(R[x], queue);
    }
    void values(int x, vector<Value> &queue, Value lo, Value hi) {
        if (!x) return;
        if (cmp(lo, VAL[x])) values(L[x], queue, lo, hi);
        if (!cmp(VAL[x], lo) && !cmp(hi, VAL[x])) queue.push_back(VAL[x]);
        if (cmp(VAL[x], hi)) values(R[x], queue, lo, hi);
    }
    SBTSet() : SZ(1), L(1), R(1) {}
    void clear() {
        VAL.clear(); SZ.clear(); L.clear(); R.clear();
        SZ.push_back(0); L.push_back(0); R.push_back(0);
    }
    bool empty() { return root == 0; }
    int size() { return SZ[root]; }
    bool contains(Value val) { return contains(root, val); }
    void add(Value val) { root = add(root, val); }
    void remove(Value val) { if (contains(val)) root = remove(root, val); }
    void removeMin() {
        if (empty()) throw runtime_error("called removeMin() with empty symbol table");
        root = removeMin(root);
    }
    void removeMax() {
        if (empty()) throw runtime_error("called removeMax() with empty symbol table");
        root = removeMax(root);
    }
    Value getMin() {
        if (empty()) throw runtime_error("called getMin() with empty symbol table");
        return VAL[getMin(root)];
    }
    Value getMax() {
        if (empty()) throw runtime_error("called getMax() with empty symbol table");
        return VAL[getMax(root)];
    }
    Value floor(Value val) {
        if (empty()) throw runtime_error("called floor() with empty symbol table");
        int x = floor(root, val);
        if (!x) throw no_such_element_exception("call to floor() resulted in no such value");
        return VAL[x];
    }
    Value ceiling(Value val) {
        if (empty()) throw runtime_error("called ceiling() with empty symbol table");
        int x = ceiling(root, val);
        if (!x) throw no_such_element_exception("call to ceiling() resulted in no such value");
        return VAL[x];
    }
    Value select(int k) {
        if (k < 0 || k >= size()) throw invalid_argument("k is not in range 0 to size");
        return VAL[select(root, k)];
    }
    int getRank(Value val) { return getRank(root, val); }
    vector<Value> values() {
        vector<Value> queue; valuesInOrder(root, queue);
        return queue;
    }
    vector<Value> values(Value lo, Value hi) {
        vector<Value> queue; values(root, queue, lo, hi);
        return queue;
    }
    int size(Value lo, Value hi) {
        if (cmp(hi, lo)) return 0;
        if (contains(hi)) return getRank(hi) - getRank(lo) + 1;
        else return getRank(hi) - getRank(lo);
    }
};
```

1.2.3 Fenwick Tree

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

// Fenwick Tree or Binary Indexed Tree supporting point updates and range queries
// Time Complexity:
//   init: O(N)
//   update, rsq: O(log N)
// Memory Complexity: O(N)
```

```cpp
template <const int MAXN, class T> struct FenwickTree {
    T BIT[MAXN];
    void init() { fill(BIT, BIT + MAXN, 0); }
    void update(int i, T v) { for (; i < MAXN; i += i & -i) BIT[i] += v; }
    T rsq(int i) { T ret = 0; for (; i > 0; i -= i & -i) ret += BIT[i]; return ret; }
    T rsq(int a, int b) { return rsq(b) - rsq(a - 1); }
};
```

1.2.4 Fenwick Tree Binary Search

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

// FenwickTree supporting point updates and range queries, as well as
// support for lower_bound and upper_bound binary searching the cumulative sum
// Time Complexity:
//   init: O(N)
//   update, rsq, lower_bound, upper_bound: O(log N)
// Memory Complexity: O(N)
template <const int MAXN, class T> struct FenwickTreeBinarySearch {
    T BIT[MAXN]; int lg = 31 - __builtin_clz(MAXN);
    void init() { fill(BIT, BIT + MAXN, 0); }
    void update(int i, T v) { for (; i < MAXN; i += i & -i) BIT[i] += v; }
    T rsq(int i) { T ret = 0; for (; i > 0; i -= i & -i) ret += BIT[i]; return ret; }
    T rsq(int a, int b) { return rsq(b) - rsq(a - 1); }
    int lower_bound(T value) {
        int ind = 0, i = 0; T sum = 0;
        for (int x = lg; x >= 0; x--) {
            i = ind | (1 << x);
            if (i >= MAXN) continue;
            if (sum + BIT[i] < value) { ind = i; sum += BIT[i]; }
        }
        return ind + 1;
    }
    int upper_bound(T value) {
        int ind = 0, i = 0; T sum = 0;
        for (int x = lg; x >= 0; x--) {
            i = ind | (1 << x);
            if (i >= MAXN) continue;
            if (sum + BIT[i] <= value) { ind = i; sum += BIT[i]; }
        }
        return ind + 1;
    }
};
```

1.2.5 Fenwick Tree Range Point

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

// Fenwick Tree or Binary Indexed Tree supporting range updates and point queries
// Time Complexity:
//   init: O(N)
//   update, getValue: O(log N)
// Memory Complexity: O(N)
template <const int MAXN, class T> struct FenwickTree {
    T BIT[MAXN];
    void init() { fill(BIT, BIT + MAXN, 0); }
    void update(int i, T v) { for (; i < MAXN; i += i & -i) BIT[i] += v; }
    void update(int a, int b, T v) { update(a, v), update(b + 1, -v); }
    T getValue(int i) { T ret = 0; for (; i > 0; i -= i & -i) ret += BIT[i]; return ret; }
};
```

1.2.6 Fenwick Tree Range

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

// Fenwick Tree or Binary Indexed Tree supporting range updates and range queries
// Time Complexity:
//   constructor: O(N)
//   update, rsq: O(log N)
// Memory Complexity: O(N)
template <class T> struct FenwickTreeRange {
    int size; vector<T> BIT1, BIT2;
    T rsq(vector<T> &BIT, int i) { T ret = 0; for (; i > 0; i -= i & -i) ret += BIT[i]; return ret; }
    void update(vector<T> &BIT, int i, T v) { for (; i <= size; i += i & -i) BIT[i] += v; }
    FenwickTreeRange(int size) : size(size), BIT1(size + 1, 0), BIT2(size + 1, 0) {}
    T rsq(int ind) { return rsq(BIT1, ind) * ind - rsq(BIT2, ind); }
    T rsq(int a, int b) { return rsq(b) - rsq(a - 1); }
    void update(int a, int b, T value) {
        update(BIT1, a, value); update(BIT1, b + 1, -value);
        update(BIT2, a, value * ((T) a - 1)); update(BIT2, b + 1, -value * (T) b);
    }
};
```

1.2.7 Fenwick Tree 2D

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

// Fenwick Tree or Binary Indexed Tree supporting point updates and range queries in 2 dimensions
// Time Complexity:
```

```
//   init: O(NM)
//   update, rsq: O(log N log M)
// Memory Complexity: O(NM)
template <const int MAXN, const int MAXM, class T> struct FenwickTree2D {
    T BIT[MAXN][MAXM];
    void init() { for (int i = 0; i < MAXN; i++) fill(BIT[i], BIT[i] + MAXM, 0); }
    void update(int x, int y, T v) {
        for (int i = x; i < MAXN; i += i & -i) for (int j = y; j < MAXM; j += j & -j) BIT[i][j] += v;
    }
    T rsq(int x, int y) {
        T ret = 0;
        for (int i = x; i > 0; i -= i & -i) for (int j = y; j > 0; j -= j & -j) ret += BIT[i][j];
        return ret;
    }
    T rsq(int x1, int y1, int x2, int y2) {
        return rsq(x2, y2) + rsq(x1 - 1, y1 - 1) - rsq(x1 - 1, y2) - rsq(x2, y1 - 1);
    }
};
```

1.2.8 Fenwick Tree ND

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

// N dimensional Fenwick Tree
// Time Complexity:
//   constructor: O(product of dims)
//   update: O(product of logs of dims)
//   rsq: O(2^N * (product of logs of dims))
// Memory Complexity: O(product of dims)
template <typename T> struct FenwickTreeND {
    int prod, n; vector<int> dim, sufProd; vector<T> BIT;
    T rsq(vector<int> &ind, int curDim, int pos) {
        T sum = 0;
        if (curDim == n) sum += BIT[pos];
        else for (int i = ind[curDim]; i > 0; i -= i & -i)
            sum += rsq(ind, curDim + 1, pos + (i - 1) * sufProd[curDim]);
        return sum;
    }
    T rsq(vector<int> &start, vector<int> &end, int curDim, int pos) {
        T sum = 0;
        if (curDim == n) sum += BIT[pos];
        else {
            for (int i = end[curDim]; i > 0; i -= i & -i)
                sum += rsq(start, end, curDim + 1, pos + (i - 1) * sufProd[curDim]);
            for (int i = start[curDim] - 1; i > 0; i -= i & -i)
                sum -= rsq(start, end, curDim + 1, pos + (i - 1) * sufProd[curDim]);
        }
        return sum;
    }
    void update(vector<int> &ind, int curDim, int pos, T value) {
        if (curDim == n) BIT[pos] += value;
        else for (int i = ind[curDim]; i <= dim[curDim]; i += i & -i)
            update(ind, curDim + 1, pos + (i - 1) * sufProd[curDim], value);
    }
    FenwickTreeND(vector<int> &dim) : prod(1), n(dim.size()), dim(dim) {
        for (int i = n - 1; i >= 0; i--) { sufProd.push_back(prod); prod *= dim[i]; }
        reverse(sufProd.begin(), sufProd.end());
        for (int i = 0; i < prod; i++) BIT.push_back(0);
    }
    T rsq(vector<int> &ind) { return rsq(ind, 0, 0); }
    T rsq(vector<int> &start, vector<int> &end) { return rsq(start, end, 0, 0); }
    void update(vector<int> &ind, T value) { update(ind, 0, 0, value); }
};
```

1.2.9 Fenwick Tree Quadratic

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

// FenwickTree supporting range updates with updates in the form of
// adding v, 2v, 3v, ... to the interval [l, r], and range queries
// Time Complexity:
//   constructor: O(N)
//   update, rsq: O(log N)
// Memory Complexity: O(N)
template <class T> struct FenwickTreeQuadratic {
    int size; vector<T> con, lin, quad;
    T rsq(vector<T> &BIT, int i) { T ret = 0; for (; i > 0; i -= i & -i) ret += BIT[i]; return ret; }
    void update(vector<T> &BIT, int i, T v) { for (; i <= size; i += i & -i) BIT[i] += v; }
    FenwickTreeQuadratic(int size) : size(size), con(size + 1), lin(size + 1), quad(size + 1) {}
    T rsq(int ind) { return (rsq(quad, ind) * (T) ind * (T) ind + rsq(lin, ind) * (T) ind + rsq(con, ind)) / (T) 2; }
    T rsq(int a, int b) { return rsq(b) - rsq(a - 1); }
    void update(int a, int b, T value) {
        int s = a - 1, len = b - a + 1;
        update(quad, a, value); update(quad, b + 1, -value);
        update(lin, a, value * ((T) 1 - (T) 2 * (T) s)); update(lin, b + 1, -value * ((T) 1 - (T) 2 * (T) s));
        update(con, a, value * ((T) s * (T) s - (T) s)); update(con, b + 1, -value * (((T) s * (T) s - (T) s)));
        update(con, b + 1, value * ((T) len * (T) (len + 1)));
    }
};
```

1.2.10 Fenwick Tree Polynomial

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

// FenwickTree supporting range updates with updates in the form of
// adding v, 2^k * v, 3^k * v, ... to the interval [l, r], and range queries
// Time Complexity:
//   constructor: O(N + K^3)
//   update: O(K (K + log N))
//   rsq: O(K log N)
// Memory Complexity: O(KN + K^3)
template <class T> struct FenwickTreePolynomial {
    T powMod(T base, T pow, T mod) {
        T x = 1, y = base % mod;
        for (; pow > 0; pow /= 2, y = y * y % mod) if (pow % 2 == 1) x = x * y % mod;
        return x;
    }
    int size, maxK; T mod;
    vector<vector<T>> BIT;
    vector<vector<T>> cof {{0, 1},
                           {0, 1, 1},
                           {0, 1, 3, 2},
                           {0, 0, 6, 12, 6},
                           {0, -4, 0, 40, 60, 24},
                           {0, 0, -60, 0, 300, 360, 120},
                           {0, 120, 0, -840, 0, 2520, 2520, 720},
                           {0, 0, 3360, 0, -11760, 0, 23520, 20160, 5040},
                           {0, -12096, 0, 80640, 0, -169344, 0, 241920, 181440, 40320},
                           {0, 0, -544320, 0, 1814400, 0, -2540160, 0, 2721600, 1814400, 362880},
                           {0, 3024000, 0, -19958400, 0, 39916800, 0, -39916800, 0, 33264000, 19958400, 3628800}};
    vector<vector<vector<T>>> sumDiff;
    vector<T> factorial;
    vector<vector<T>> pascal;
    T invDen;
    T rsq(vector<T> &BIT, int i) { T ret = 0; for (; i > 0; i -= i & -i) (ret += BIT[i]) %= mod; return ret; }
    void update(vector<T> &BIT, int i, T v) { for (; i <= size; i += i & -i) (BIT[i] += v) %= mod; }
    FenwickTreePolynomial(int size, int maxK, T modbigPrime) : size(size), maxK(maxK), mod(modbigPrime), BIT(maxK + 2), sumDiff(maxK
        + 1), pascal(maxK + 2) {
        assert(maxK >= 0 && maxK <= 10);
        factorial.push_back(1);
        for (int i = 1; i < maxK + 2; i++) factorial.push_back(factorial.back() * i);
        for (int i = 0; i < maxK + 2; i++) for (int j = 0; j <= size; j++) BIT[i].push_back(0);
        for (int k = 0; k < maxK + 1; k++) {
            for (int i = 0; i < maxK + 2; i++) {
                sumDiff[k].push_back(vector<T>());
                for (int j = 0; j < maxK + 2 - i; j++) sumDiff[k][i].push_back(0);
            }
        }
        for (int i = 0; i < maxK + 2; i++) {
            pascal[i].push_back(1);
            for (int j = 1; j <= i; j++) pascal[i].push_back(pascal[i].back() * (i - j + 1) / j);
        }
        for (int k = 0; k < maxK + 1; k++) {
            for (int i = 0; i < k + 2; i++) {
                int sign = 1;
                for (int j = 0; j <= i; j++) {
                    sumDiff[k][i - j][j] = (sumDiff[k][i - j][j] + (((cof[k][i] * pascal[i][j]) % mod * (factorial[maxK + 1] /
                        factorial[k + 1])) % mod) * sign) % mod;
                    sign = -sign;
                }
            }
        }
        invDen = pow3(factorial[maxK + 1], mod - 2, mod);
    }
    T rsq(int ind) {
        T sum = rsq(BIT[maxK + 1], ind);
        for (int i = maxK; i >= 0; i--) sum = ((sum * ind) % mod + rsq(BIT[i], ind)) % mod;
        if (sum < 0) sum += mod;
        return sum * invDen % mod;
    }
    T rsq(int a, int b) {
        T sum = (rsq(b) - rsq(a - 1)) % mod;
        if (sum < 0) sum += mod;
        return sum;
    }
    void update(int a, int b, T value, int k) {
        value %= mod;
        if (value < 0) value += mod;
        T s = a - 1, len = b - a + 1, mult;
        for (int i = 0; i < k + 2; i++) {
            mult = sumDiff[k][i][k + 1 - i];
            for (int j = k - i; j >= 0; j--) mult = ((mult * s) % mod + sumDiff[k][i][j]) % mod;
            update(BIT[i], a, value * mult % mod);
            update(BIT[i], b + 1, -value * mult % mod);
        }
        mult = cof[k][k + 1];
        for (int i = k; i >= 0; i--) mult = ((mult * len) % mod + cof[k][i]) % mod;
        if (mult < 0) mult += mod;
        mult = (mult * (factorial[maxK + 1] / factorial[k + 1])) % mod;
        update(BIT[0], b + 1, value * mult % mod);
    }
};
```

1.2.11 Bottom Up Segment Trees

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

```cpp
using namespace std;

// single assignment and modifications, range query
template <const int MAXN> struct SegmentTree_SAM_RQ {
    using Data = int; int N; Data T[2 * MAXN];
    const Data def = 0; // default value
    const Data qdef = INT_MIN; // query default value
    // operation must be associative (but not necessarily commutative)
    Data merge(Data l, Data r); // to be implemented
    Data applyVal(Data o, Data n); // to be implemented
    template <class it> void init(it st, it en) {
        N = en - st; for (int i = 0; i < N; i++) T[N + i] = *(st + i);
        for (int i = N - 1; i > 0; i--) T[i] = merge(T[i << 1], T[i << 1 | 1]);
    }
    void init(int size) { N = size; for (int i = 1; i < 2 * N; i++) T[i] = def; }
    void update(int i, Data v) { for (i += N - 1, T[i] = applyVal(T[i], v); i >>= 1;) T[i] = merge(T[i << 1], T[i << 1 | 1]); }
    Data query(int l, int r) {
        Data ql = qdef, qr = qdef;
        for (l += N - 1, r += N - 1; l <= r; l >>= 1, r >>= 1) {
            if (l & 1) ql = merge(ql, T[l++]);
            if (!(r & 1)) qr = merge(T[r--], qr);
        }
        return merge(ql, qr);
    }
};

// range modification, single query
// works when order of modifications does not affect result
template <const int MAXN> struct SegmentTree_RM_SQ {
    using Data = int; int N; Data T[2 * MAXN];
    const Data def = 0; // default value
    bool final = false; // whether the pushAll function has been called
    Data merge(Data l, Data r); // to be implemented
    template <class it> void init(it st, it en) { N = en - st; for (int i = 0; i < N; i++) T[N + i] = *(st + i), T[i] = 0; }
    void init(int size) { N = size; for (int i = 0; i < N; i++) T[N + i] = def, T[i] = 0; }
    void update(int l, int r, Data v) {
        for (l += N - 1, r += N - 1; l <= r; l >>= 1, r >>= 1) {
            if (l & 1) { T[l] = merge(T[l], v); l++; }
            if (!(r & 1)) { T[r] = merge(T[r], v); r--; }
        }
    }
    Data query(int i) {
        if (final) return T[N + i - 1];
        Data q = 0; for (i += N - 1; i > 0; i >>= 1) q = merge(q, T[i]);
        return q;
    }
    void pushAll() {
        for (int i = 1; i < N; i++) { T[i << 1] = merge(T[i << 1], T[i]); T[i << 1 | 1] = merge(T[i << 1 | 1], T[i]); T[i] = 0; }
        final = true;
    }
};

// range assignments/modifications, range query
template <const int MAXN> struct SegmentTree_RAM_RQ {
private:
    using Data = int; using Lazy = int; int N, H; Data T[2 * MAXN]; Lazy L[MAXN]; // L stores lazy values
    const Data def = 0; // default value
    const Data qdef = 0; // query default value
    const Lazy ldef = 0; // lazy default value
    // operation must be associative (but not necessarily commutative)
    Data merge(Data l, Data r); // to be implemented
    Data getSegmentVal(Lazy v, int k); // to be implemented
    Lazy mergeLazy(Lazy l, Lazy r); // to be implemented
    Data applyVal(Data o, Data n); // to be implemented
    void apply(int i, Lazy v, int k) {
        T[i] = applyVal(T[i], getSegmentVal(v, k));
        if (i < N) L[i] = mergeLazy(L[i], v);
    }
    void pushup(int i) {
        for (int k = 2; i > 1; k <<= 1) { i >>= 1; T[i] = L[i] == ldef ? merge(T[i << 1], T[i << 1 | 1]) : getSegmentVal(L[i], k); }
    }
    void propagate(int i) {
        for (int h = H, k = 1 << (H - 1); h > 0; h--, k >>= 1) {
            int ii = i >> h;
            if (L[ii] != ldef) { apply(ii << 1, L[ii], k); apply(ii << 1 | 1, L[ii], k); L[ii] = ldef; }
        }
    }
    template <class it> void init(it st, it en) {
        N = en - st; H = 0; for (int i = 1; i <= N; H++) i <<= 1;
        for (int i = 0; i < N; i++) { T[N + i] = *(st + i); L[i] = ldef; }
        for (int i = N - 1; i > 0; i--) T[i] = merge(T[i << 1], T[i << 1 | 1]);
    }
    void init(int size) {
        N = size; H = 0;
        for (int i = 1; i <= N; H++) i <<= 1;
        for (int i = 1; i < 2 * N; i++) T[i] = def;
        for (int i = 1; i < N; i++) L[i] = ldef;
    }
    void update(int l, int r, Lazy v) {
        l += N - 1; r += N - 1; propagate(l); propagate(r); int l0 = l, r0 = r, k = 1;
        for (; l <= r; l >>= 1, r >>= 1, k <<= 1) {
            if (l & 1) apply(l++, v, k);
            if (!(r & 1)) apply(r--, v, k);
        }
        pushup(l0); pushup(r0);
    }
```

```
    Data query(int l, int r) {
        l += N - 1; r += N - 1; propagate(l); propagate(r); Data ql = qdef, qr = qdef;
        for (; l <= r; l >>= 1, r >>= 1) {
            if (l & 1) ql = merge(ql, T[l++]);
            if (!(r & 1)) qr = merge(T[r--], qr);
        }
        return merge(ql, qr);
    }
};
```

1.2.12 Top Down Segment Trees

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

template <const int MAXN> struct SegmentTree {
    using Data = int; const Data vdef = 0, qdef = 0;
    Data merge(const Data &l, const Data &r); // to be implemented
    Data apply(const Data &x, const Data &v); // to be implemented
    Data T[MAXN * 4], A[MAXN]; int N;
    void build(int cur, int cL, int cR) {
        if (cL == cR) { T[cur] = A[cL]; return; }
        int m = cL + (cR - cL) / 2; build(cur * 2, cL , m); build(cur * 2 + 1, m + 1, cR);
        T[cur] = merge(T[cur * 2], T[cur * 2 + 1]);
    }
    void update(int cur, int cL, int cR, int ind, const Data &val) {
        if (cL > ind || cR < ind) return;
        if (cL == cR) { T[cur] = apply(T[cur], val); return; }
        int m = cL + (cR - cL) / 2; update(cur * 2, cL, m, ind, val); update(cur * 2 + 1, m + 1, cR, ind, val);
        T[cur] = merge(T[cur * 2], T[cur * 2 + 1]);
    }
    Data query(int cur, int cL, int cR, int l, int r) {
        if (cL > r || cR < l) return qdef;
        if (cL >= l && cR <= r) return T[cur];
        int m = cL + (cR - cL) / 2;
        return merge(query(cur * 2, cL, m, l, r), query(cur * 2 + 1, m + 1, cR, l, r));
    }
    template <class It> SegmentTree(It st, It en): N(en - st) {
        for (int i = 1; i <= N; i++) A[i] = *(st + i - 1); build(1, 1, N);
    }
    SegmentTree(int size) : N(size) { fill(A + 1, A + N, vdef); build(1, 1, size); }
    void update(int ind, const Data &val) { update(1, 1, N, ind, val); }
    Data query(int l, int r) { return query(1, 1, N, l, r); }
};

template <const int MAXN> struct LazySegmentTree {
    using Data = int; using Lazy = int; const Data vdef = 0, qdef = 0; const Lazy ldef = 0;
    Data merge(const Data &l, const Data &r); // to be implemented
    Data apply(const Data &x, const Lazy &v); // to be implemented
    Lazy getSegmentVal(const Lazy &v, int len); // to be implemented
    Lazy mergeLazy(const Lazy &l, const Lazy &r); // to be implemented
    Data T[MAXN * 4], A[MAXN]; Lazy L[MAXN * 4]; int N;
    void propagate(int cur, int cL, int cR) {
        if (T[cur].lazy != ldef) {
            int m = cL + (cR - cL) / 2;
            T[cur * 2] = apply(T[cur * 2], getSegmentVal(L[cur], m - cL + 1));
            L[cur * 2] = mergeLazy(L[cur * 2], L[cur]);
            T[cur * 2 + 1] = apply(T[cur * 2 + 1], getSegmentVal(L[cur], cR - m));
            L[cur * 2 + 1] = mergeLazy(L[cur * 2 + 1], L[cur]);
            L[cur].lazy = ldef;
        }
    }
    void build(int cur, int cL, int cR) {
        if (cL == cR) { T[cur] = A[cL]; L[cur] = ldef; return; }
        int m = cL + (cR - cL) / 2; build(cur * 2, cL , m); build(cur * 2 + 1, m + 1, cR);
        T[cur] = merge(T[cur * 2], T[cur * 2 + 1]);
    }
    void update(int cur, int cL, int cR, int l, int r, const Lazy &val) {
        if (cL > r || cR < l) return;
        if (cL != cR) propagate(cur, cL, cR);
        if (cL >= l && cR <= r) {
            T[cur] = apply(T[cur], getSegmentVal(val, cR - cL + 1));
            L[cur] = mergeLazy(L[cur], val);
            return;
        }
        int m = cL + (cR - cL) / 2; update(cur * 2, cL, m, l, r, val); update(cur * 2 + 1, m + 1, cR, l, r, val);
        T[cur] = merge(T[cur * 2], T[cur * 2 + 1]);
    }
    Data query(int cur, int cL, int cR, int l, int r) {
        if (cL != cR) propagate(cur, cL, cR);
        if (cL > r || cR < l) return qdef;
        if (cL >= l && cR <= r) return T[cur];
        int m = cL + (cR - cL) / 2;
        return merge(query(cur * 2, cL, m, l, r), query(cur * 2 + 1, m + 1, cR, l, r));
    }
    template <class It> LazySegmentTree(It st, It en): N(en - st) {
        for (int i = 1; i <= N; i++) A[i] = *(st + i - 1); build(1, 1, N);
    }
    LazySegmentTree(int size) : N(size) { fill(A + 1, A + N, vdef); build(1, 1, size); }
    void update(int ind, const Lazy &val) { update(1, 1, N, ind, val); }
    Data query(int l, int r) { return query(1, 1, N, l, r); }
};

struct PersistentSegmentTree {
    using Data = int; using Lazy = int; static const Data vdef = 0, qdef = 0; static const Lazy ldef = 0;
    Data merge(const Data &l, const Data &r); // to be implemented
```

```
    Data apply(const Data &x, const Lazy &v); // to be implemented
    Lazy getSegmentVal(const Lazy &v, int len); // to be implemented
    Lazy mergeLazy(const Lazy &l, const Lazy &r); // to be implemented
    struct Node {
        Node *left = nullptr, *right = nullptr;
        Data val = PersistentSegmentTree::vdef;
        Lazy lazy = PersistentSegmentTree::ldef;
    };
    void propagate(Node *cur, int cL, int cR) {
        if (cur->lazy != ldef) {
            int m = cL + (cR - cL) / 2;
            if (cur->left == nullptr) cur->left = new Node();
            cur->left->val = apply(cur->left->val, getSegmentVal(cur->lazy, m - cL + 1));
            cur->left->lazy = mergeLazy(cur->left->lazy, cur->lazy);
            if (cur->right == nullptr) cur->right = new Node();
            cur->right->val = apply(cur->right->val, getSegmentVal(cur->lazy, cR - m));
            cur->right->lazy = mergeLazy(cur->right->lazy, cur->lazy);
            cur->lazy = ldef;
        }
    }
    int N; vector<Node*> roots = {new Node()};
    template <class It> Node *build(int cL, int cR, It st) {
        Node *ret = new Node();
        if (cL == cR) { ret->val = *(st + cL - 1); return ret; }
        int m = cL + (cR - cL) / 2;
        ret->left = build(cL, m, st); ret->right = build(m + 1, cR, st);
        ret->val = merge(ret->left->val, ret->right->val);
        return ret;
    }
    Node *update(Node *cur, int cL, int cR, int l, int r, const Lazy &val) {
        if (cL > r || cR < l) return cur;
        Node *ret = new Node();
        if (cur) {
            ret->left = cur->left; ret->right = cur->right; ret->val = cur->val, ret->lazy = cur->lazy;
        }
        assert(ret);
        if (cL != cR) propagate(ret, cL, cR);
        if (cL >= l && cR <= r) {
            ret->val = apply(ret->val, getSegmentVal(val, cR - cL + 1));
            ret->lazy = mergeLazy(ret->lazy, val);
            return ret;
        }
        int m = cL + (cR - cL) / 2;
        ret->left = update(ret->left, cL, m, l, r, val);
        ret->right = update(ret->right, m + 1, cR, l, r, val);
        if (ret->left && ret->right) ret->val = merge(ret->left->val, ret->right->val);
        else if (ret->left) ret->val = ret->left->val;
        else if (ret->right) ret->val = ret->right->val;
        return ret;
    }
    Data query(Node *cur, int cL, int cR, int l, int r) {
        if (!cur || cL > r || cR < l) return qdef;
        if (cL != cR) propagate(cur, cL, cR);
        if (cL >= l && cR <= r) return cur->val;
        int m = cL + (cR - cL) / 2;
        return merge(query(cur->left, cL, m, l, r), query(cur->right, m + 1, cR, l, r));
    }
    template <class It> PersistentSegmentTree(It st, It en): N(en - st) { roots.push_back(build(1, N, st)); }
    PersistentSegmentTree(int size) : N(size) {}
    void update(int l, int r, int val) { roots.push_back(update(roots.back(), 1, N, l, r, val)); }
    Data query(int rootInd, int l, int r) { return query(roots[rootInd], 1, N, l, r); }
    void revert(int x) { roots.push_back(roots[x]); }
};
```

1.2.13 Implicit Treap

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

// Implicit Treap supporting point updates and range queries
// Time Complexity:
//   constructor: O(N)
//   updateVal, queryRange: O(log N)
// Memory Complexity: O(N)
struct ImplcitTreap {
    seed_seq seq {
        (uint64_t) chrono::duration_cast<chrono::nanoseconds>(chrono::high_resolution_clock::now().time_since_epoch()).count(),
        (uint64_t) __builtin_ia32_rdtsc(),
        (uint64_t) (uintptr_t) make_unique<char>().get()
    };
    mt19937 rng; uniform_real_distribution<double> dis;
    using Data = int; const Data vdef = 0;
    vector<Data> VAL, SBTR; vector<int> L, R, SZ; vector<double> PRI; int root = -1;
    int makeNode(Data val) {
        VAL.push_back(val); SBTR.push_back(val); L.push_back(-1); R.push_back(-1); SZ.push_back(1); PRI.push_back(dis(rng));
        return int(VAL.size()) - 1;
    }
    int size(int x) { return x == -1 ? 0 : SZ[x]; }
    Data sbtrVal(int x) { return x == -1 ? vdef : SBTR[x]; }
    Data merge(Data l, Data r); // to be implemented
    Data applyVal(Data o, Data n); // to be implemented
    void apply(int x, Data d) {
        if (x == -1) return;
        VAL[x] = applyVal(VAL[x], d); SBTR[x] = applyVal(SBTR[x], d);
    }
    void update(int x) {
```

```
        if (x == -1) return;
        SZ[x] = 1; SBTR[x] = VAL[x];
        if (L[x] != -1) { SZ[x] += SZ[L[x]]; SBTR[x] = merge(SBTR[x], SBTR[L[x]]); }
        if (R[x] != -1) { SZ[x] += SZ[R[x]]; SBTR[x] = merge(SBTR[x], SBTR[R[x]]); }
    }
    void merge(int &x, int l, int r) {
        if (l == -1 || r == -1) { x = l == -1 ? r : l; }
        else if (PRI[l] > PRI[r]) { merge(R[l], R[l], r); x = l; }
        else { merge(L[r], l, L[r]); x = r; }
        update(x);
    }
    void split(int x, int &l, int &r, int lsz) {
        if (x == -1) { l = r = -1; return; }
        if (lsz <= size(L[x])) { split(L[x], l, L[x], lsz); r = x; }
        else { split(R[x], R[x], r, lsz - size(L[x]) - 1); l = x; }
        update(x);
    }
    ImplcitTreap(int N) : rng(seq), dis(0.0, 1.0) {
        for (int i = 0; i < N; i++) merge(root, root, makeNode(vdef));
    }
    template <class It> ImplcitTreap(It st, It en) : rng(seq), dis(0.0, 1.0) {
        for (It i = st; i < en; i++) merge(root, root, makeNode(*i));
    }
    void updateVal(int i, Data val) {
        int left, right, mid; split(root, left, mid, i); split(mid, mid, right, 1);
        apply(mid, val); update(mid); merge(root, left, mid); merge(root, root, right);
    }
    Data queryRange(int l, int r) {
        int left, right, mid; split(root, left, mid, l); split(mid, mid, right, r - l + 1);
        Data ret = sbtrVal(mid); merge(root, left, mid); merge(root, root, right);
        return ret;
    }
};
```

1.2.14 Lazy Implicit Treap

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

// Implicit Treap supporting range updates and queries
// Time Complexity:
//   constructor: O(N)
//   updateRange, queryRange: O(log N)
// Memory Complexity: O(N)
struct LazyImplcitTreap {
    seed_seq seq {
        (uint64_t) chrono::duration_cast<chrono::nanoseconds>(chrono::high_resolution_clock::now().time_since_epoch()).count(),
        (uint64_t) __builtin_ia32_rdtsc(),
        (uint64_t) (uintptr_t) make_unique<char>().get()
    };
    mt19937 rng; uniform_real_distribution<double> dis;
    using Data = int; using Lazy = int; const Data vdef = 0; const Lazy ldef = 0;
    vector<Data> VAL, SBTR; vector<Lazy> LZ; vector<int> L, R, SZ; vector<double> PRI; int root = -1;
    int makeNode(Data val) {
        VAL.push_back(val); SBTR.push_back(val); LZ.push_back(ldef);
        L.push_back(-1); R.push_back(-1); SZ.push_back(1); PRI.push_back(dis(rng));
        return int(VAL.size()) - 1;
    }
    int size(int x) { return x == -1 ? 0 : SZ[x]; }
    Data sbtrVal(int x) { return x == -1 ? vdef : SBTR[x]; }
    Data merge(Data l, Data r); // to be implemented
    Lazy getSegmentVal(Lazy v, int k); // to be implemented
    Lazy mergeLazy(Lazy l, Lazy r); // to be implemented
    Data applyLazy(Data d, Lazy l); // to be implemented
    void apply(int x, Lazy v) {
        if (x == -1) return;
        VAL[x] = applyLazy(VAL[x], v); SBTR[x] = applyLazy(SBTR[x], getSegmentVal(v, SZ[x])); LZ[x] = mergeLazy(LZ[x], v);
    }
    void propagate(int x) {
        if (x == -1 || LZ[x] == ldef) return;
        if (L[x] != -1) apply(L[x], LZ[x]);
        if (R[x] != -1) apply(R[x], LZ[x]);
        LZ[x] = ldef;
    }
    void update(int x) {
        if (x == -1) return;
        SZ[x] = 1; SBTR[x] = VAL[x];
        if (L[x] != -1) { SZ[x] += SZ[L[x]]; SBTR[x] = merge(SBTR[x], SBTR[L[x]]); }
        if (R[x] != -1) { SZ[x] += SZ[R[x]]; SBTR[x] = merge(SBTR[x], SBTR[R[x]]); }
    }
    void merge(int &x, int l, int r) {
        propagate(l); propagate(r);
        if (l == -1 || r == -1) { x = l == -1 ? r : l; }
        else if (PRI[l] > PRI[r]) { merge(R[l], R[l], r); x = l; }
        else { merge(L[r], l, L[r]); x = r; }
        update(x);
    }
    void split(int x, int &l, int &r, int lsz) {
        if (x == -1) { l = r = -1; return; }
        propagate(x);
        if (lsz <= size(L[x])) { split(L[x], l, L[x], lsz); r = x; }
        else { split(R[x], R[x], r, lsz - size(L[x]) - 1); l = x; }
        update(x);
    }
    LazyImplcitTreap(int N) : rng(seq), dis(0.0, 1.0) {
```

```
        for (int i = 0; i < N; i++) merge(root, root, makeNode(vdef));
    }
    template <class It> LazyImplcitTreap(It st, It en) : rng(seq), dis(0.0, 1.0) {
        for (It i = st; i < en; i++) merge(root, root, makeNode(*i));
    }
    void updateRange(int l, int r, Lazy val) {
        int left, right, mid; split(root, left, mid, l); split(mid, mid, right, r - l + 1);
        apply(mid, val); merge(root, left, mid); merge(root, root, right);
    }
    Data queryRange(int l, int r) {
        int left, right, mid; split(root, left, mid, l); split(mid, mid, right, r - l + 1);
        Data ret = sbtrVal(mid); merge(root, left, mid); merge(root, root, right);
        return ret;
    }
};
```

1.3 Geometry

1.3.1 Point

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

using T = double; const T EPS = 1e-9;

struct Point {
    T x, y;
    Point(T x = 0, T y = 0) : x(x), y(y) {}
    double polarRadius() const { return sqrt(x * x + y * y); }
    double theta() const { return atan2(y, x); }
    double angleTo(const Point &that) const { return atan2(that.x - x, that.y - y); }
    static int ccw(const Point &a, const Point &b, const Point &c) {
        T area2 = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
        if (area2 < -T(EPS)) return -1;
        else if (area2 > T(EPS)) return +1;
        else return 0;
    }
    static T area2(const Point &a, const Point &b, const Point &c) {
        return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    }
    Point rotate(const Point &that, double theta) const {
        T x = that.x + (x - that.x) * cos(theta) - (y - that.y) * sin(theta);
        T y = that.y + (x - that.x) * sin(theta) + (y - that.y) * cos(theta);
        return Point(x, y);
    }
    double distanceTo(const Point &that) const {
        T dx = x - that.x, dy = y - that.y;
        return sqrt(dx * dx + dy * dy);
    }
    T distanceSquaredTo(const Point &that) const {
        T dx = x - that.x, dy = y - that.y;
        return dx * dx + dy * dy;
    }
    bool onSegment(const Point &p, const Point &q) const {
        return x <= max(p.x, q.x) + T(EPS) && x >= min(p.x, q.x) - T(EPS) && y <= max(p.y, q.y) + T(EPS) && y >= min(p.y, q.y) - T(
            EPS);
    }
    static bool intersects(const Point &p1, const Point &q1, const Point &p2, const Point &q2) {
        int o1 = Point::ccw(p1, q1, p2), o2 = Point::ccw(p1, q1, q2), o3 = Point::ccw(p2, q2, p1), o4 = Point::ccw(p2, q2, q1);
        if (o1 != o2 && o3 != o4) return true;
        if (o1 == 0 && p2.onSegment(p1, q1)) return true;
        if (o2 == 0 && q2.onSegment(p1, q1)) return true;
        if (o3 == 0 && p1.onSegment(p2, q2)) return true;
        if (o4 == 0 && q1.onSegment(p2, q2)) return true;
        return false;
    }
    static Point intersection(const Point &p1, const Point &q1, const Point &p2, const Point &q2) {
        T A1 = q1.y - p1.y, B1 = p1.x - q1.x, C1 = A1 * p1.x + B1 * p1.y;
        T A2 = q2.y - p2.y, B2 = p2.x - q2.x, C2 = A2 * p2.x + B2 * p2.y, det = A1 * B2 - A2 * B1;
        if (abs(det) <= T(EPS)) throw runtime_error("The lines do not intersect");
        return Point((B2 * C1 - B1 * C2) / det, (A1 * C2 - A2 * C1) / det);
    }
    int compareTo(const Point &that) const {
        if (y < that.y - T(EPS)) return -1;
        if (y > that.y + T(EPS)) return +1;
        if (x < that.x - T(EPS)) return -1;
        if (x > that.x + T(EPS)) return +1;
        return 0;
    }
    bool operator == (const Point &other) const { return abs(x - other.x) <= T(EPS) && abs(y - other.y) <= T(EPS); }
    bool operator != (const Point &other) const { return abs(x - other.x) > T(EPS) || abs(y - other.y) > T(EPS); }
    static bool xOrderLt(const Point &p, const Point &q) { return p.x < q.x - T(EPS); }
    static bool xOrderLe(const Point &p, const Point &q) { return p.x <= q.x + T(EPS); }
    static bool xOrderGt(const Point &p, const Point &q) { return p.x > q.x + T(EPS); }
    static bool xOrderGe(const Point &p, const Point &q) { return p.x >= q.x - T(EPS); }
    static bool yOrderLt(const Point &p, const Point &q) { return p.y < q.y - T(EPS); }
    static bool yOrderLe(const Point &p, const Point &q) { return p.y <= q.y + T(EPS); }
    static bool yOrderGt(const Point &p, const Point &q) { return p.y > q.y + T(EPS); }
    static bool yOrderGe(const Point &p, const Point &q) { return p.y >= q.y - T(EPS); }
    static bool rOrderLt(const Point &p, const Point &q) { return (p.x*p.x + p.y*p.y) < (q.x*q.x + q.y*q.y) - T(EPS); }
    static bool rOrderLe(const Point &p, const Point &q) { return (p.x*p.x + p.y*p.y) <= (p.x*p.x + p.y*p.y) + T(EPS); }
    static bool rOrderGt(const Point &p, const Point &q) { return (p.x*p.x + p.y*p.y) > (q.x*q.x + q.y*q.y) + T(EPS); }
    static bool rOrderGe(const Point &p, const Point &q) { return (p.x*p.x + p.y*p.y) >= (q.x*q.x + q.y*q.y) - T(EPS); }
    bool polarOrderLt(const Point &q1, const Point &q2) const { return ccw(*this, q1, q2) > 0; }
    bool polarOrderLe(const Point &q1, const Point &q2) const { return ccw(*this, q1, q2) >= 0; }
```

```
    bool polarOrderGt(const Point &q1, const Point &q2) const { return ccw(*this, q1, q2) < 0; }
    bool polarOrderGe(const Point &q1, const Point &q2) const { return ccw(*this, q1, q2) <= 0; }
    bool atan2OrderLt(const Point &q1, const Point &q2) const { return angleTo(q1) - angleTo(q2) < T(EPS); }
    bool atan2OrderLe(const Point &q1, const Point &q2) const { return angleTo(q1) - angleTo(q2) <= -T(EPS); }
    bool atan2OrderGt(const Point &q1, const Point &q2) const { return angleTo(q1) - angleTo(q2) < T(EPS); }
    bool atan2OrderGe(const Point &q1, const Point &q2) const { return angleTo(q1) - angleTo(q2) <= -T(EPS); }
    bool distanceToOrderLt(const Point &p, const Point &q) { return distanceSquaredTo(p) - distanceSquaredTo(q) < T(EPS); }
    bool distanceToOrderLe(const Point &p, const Point &q) { return distanceSquaredTo(p) - distanceSquaredTo(q) <= -T(EPS); }
    bool distanceToOrderGt(const Point &p, const Point &q) { return distanceSquaredTo(p) - distanceSquaredTo(q) < T(EPS); }
    bool distanceToOrderGe(const Point &p, const Point &q) { return distanceSquaredTo(p) - distanceSquaredTo(q) <= -T(EPS); }
};

struct Point_hash {
    size_t operator ()(const Point &p) const {
        return 31 * hash<T> {}(p.x) + hash<T> {}(p.y);
    }
};
```

1.3.2 Vector

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

using T = double; const T EPS = 1e-9;

struct Vector {
    int d; vector<T> data;
    Vector() {}
    Vector(int d) : d(d), data(d, 0) {}
    template <class It> Vector(It st, It en) : d(d), data(st, en) {}
    int dimension() const { return d; }
    T dot(const Vector &that) const {
        if (d != that.d) throw invalid_argument("Dimensions don't agree");
        T sum = 0;
        for (int i = 0; i < d; i++) sum = sum + (data[i] * that.data[i]);
        return sum;
    }
    T operator * (const Vector &that) const { return dot(that); }
    T cross2D(const Vector &that) const {
        if (d != 2 || that.d != 2) throw invalid_argument("Vectors must be 2-dimensional");
        return data[0] * that.data[1] - data[1] * that.data[0];
    }
    T operator | (const Vector &that) const { return cross2D(that); }
    Vector cross3D(const Vector &that) const {
        if (d != 3 || that.d != 3) throw invalid_argument("Vectors must be 3-dimensional");
        Vector c(d);
        c.data[0] = data[1] * that.data[2] - data[2] * that.data[1];
        c.data[1] = data[2] * that.data[0] - data[0] * that.data[2];
        c.data[2] = data[0] * that.data[1] - data[1] * that.data[0];
        return c;
    }
    Vector operator ^ (const Vector &that) const { return cross3D(that); }
    double magnitude() const { return sqrt(dot(*this)); }
    double distanceTo(const Vector &that) const {
        if (d != that.d) throw invalid_argument("Dimensions don't agree");
        return minus(that).magnitude();
    }
    Vector plus(const Vector &that) const {
        if (d != that.d) throw invalid_argument("Dimensions don't agree");
        Vector c(d);
        for (int i = 0; i < d; i++) c.data[i] = data[i] + that.data[i];
        return c;
    }
    Vector operator + (const Vector &that) const { return plus(that); }
    Vector minus(const Vector &that) const {
        if (d != that.d) throw invalid_argument("Dimensions don't agree");
        Vector c(d);
        for (int i = 0; i < d; i++) c.data[i] = data[i] - that.data[i];
        return c;
    }
    Vector operator - (const Vector &that) const { return minus(that); }
    T cartesian(int i) const { return data[i]; }
    T &operator [](int i) { return data[i]; }
    T operator [](int i) const { return data[i]; }
    Vector scale(double alpha) const {
        Vector c(d);
        for (int i = 0; i < d; i++) c.data[i] = alpha * data[i];
        return c;
    }
    Vector operator * (double alpha) const { return scale(alpha); }
    Vector operator / (double alpha) const { return scale(1.0 / alpha); }
    Vector direction() const {
        if (magnitude() <= EPS) throw runtime_error("Zero-vector has no direction");
        return scale(1.0 / magnitude());
    }
    double directionCosine(int i) const { return data[i] / magnitude(); }
    Vector projectionOn(const Vector &that) const {
        if (d != that.d) throw invalid_argument("Dimensions don't agree");
        return that.scale(dot(that) / that.dot(that));
    }
    Vector rotate(Vector &that, double theta) const {
        if (d == 2 && that.d == 2) {
            Vector r(2);
            r.data[0] = that.data[0] + (data[0] - that.data[0]) * cos(theta) - (data[1] - that.data[1]) * sin(theta);
            r.data[1] = that.data[1] + (data[0] - that.data[0]) * sin(theta) + (data[1] - that.data[1]) * cos(theta);
            return r;
        } else if (d == 3 && that.d == 3) {
```

```
            return scale(cos(theta)).plus(that.direction().cross3D(*this).scale(sin(theta))).plus(that.direction().
                scale(that.direction().dot(*this)).scale(1.0 - cos(theta)));
        } else if (d == that.d) {
            throw invalid_argument("Vectors must be 2-dimensional or 3-dimensional");
        } else {
            throw invalid_argument("Dimensions don't agree");
        }
    }
    bool operator < (const Vector &that) const {
        for (int i = 0; i < d; i++) if (abs(data[i] - that.data[i]) > EPS) return data[i] < that.data[i];
        return false;
    }
};
```

1.3.3 Rectangle

```
#pragma once
#include <bits/stdc++.h>
#include "Point.h"
using namespace std;

using T = double; const T EPS = 1e-9;

struct Rectangle {
    T xmin, ymin, xmax, ymax;
    Rectangle(T xmin, T ymin, T xmax, T ymax) : xmin(xmin), ymin(ymin), xmax(xmax), ymax(ymax) {
        assert(xmin <= xmax);
        assert(ymin <= ymax);
    }
    T width() const { return xmax - xmin; }
    T height() const { return ymax - ymin; }
    bool intersects(Rectangle that) const {
        return xmax - that.xmin >= -EPS && ymax - that.ymin >= -EPS
            && that.xmax - xmin >= -EPS && that.ymax - ymin >= -EPS;
    }
    bool contains(const Point &p) const {
        return (p.x - xmin >= -EPS) && (xmax - p.x >= -EPS) && (p.y - ymin >= -EPS) && (ymax - p.y >= -EPS);
    }
    double distanceTo(const Point &p) const { return sqrt(distanceSquaredTo(p)); }
    T distanceSquaredTo(const Point &p) const {
        T dx = 0, dy = 0;
        if (p.x < xmin) dx = p.x - xmin;
        else if (p.x > xmax) dx = p.x - xmax;
        if (p.y < ymin) dy = p.y - ymin;
        else if (p.y > ymax) dy = p.y - ymax;
        return dx * dx + dy * dy;
    }
    bool operator == (const Rectangle &that) const {
        if (abs(xmin - that.xmin) > EPS) return false;
        if (abs(ymin - that.ymin) > EPS) return false;
        if (abs(xmax - that.xmax) > EPS) return false;
        if (abs(ymax - that.ymax) > EPS) return false;
        return true;
    }
    bool operator != (const Rectangle &that) const {
        if (abs(xmin - that.xmin) > EPS) return true;
        if (abs(ymin - that.ymin) > EPS) return true;
        if (abs(xmax - that.xmax) > EPS) return true;
        if (abs(ymax - that.ymax) > EPS) return true;
        return false;
    }
};

struct Rectangle_hash {
    size_t operator ()(const Rectangle &r) const {
        return 31 * (31 * (31 * hash<T> {}(r.xmin) + hash<T> {}(r.ymin)) + hash<T> {}(r.xmax)) + hash<T> {}(r.ymax);
    }
};
```

1.3.4 Kd Tree

```
#pragma once
#include <bits/stdc++.h>
#include "Point.h"
#include "Rectangle.h"
using namespace std;

using T = double; const T EPS = 1e-9;

struct KdTree {
    bool VERTICAL = false, HORIZONTAL = true;
    T XMIN, YMIN, XMAX, YMAX;
    struct Node {
        Point p; Rectangle r;
        Node *leftUp = nullptr, *rightDown = nullptr;
        Node(const Point &p, const Rectangle &r) : p(p), r(r) {}
    };
    int cnt; Node *root;
    template <class It> Node *construct(Node *n, It points, int lo, int hi, bool partition, T xmin, T ymin, T xmax, T ymax) {
        if (lo > hi) return nullptr;
        int mid = lo + (hi - lo) / 2;
        if (partition == VERTICAL) nth_element(points + lo, points + mid, points + hi + 1, Point::xOrderLt);
        else nth_element(points + lo, points + mid, points + hi + 1, Point::yOrderLt);
        Point p = *(points + mid);
        n = new Node(p, Rectangle(xmin, ymin, xmax, ymax));
        if (partition == VERTICAL) {
            n->leftUp = construct(n->leftUp, points, lo, mid - 1, !partition, xmin, ymin, n->p.x, ymax);
            n->rightDown = construct(n->rightDown, points, mid + 1, hi, !partition, n->p.x, ymin, xmax, ymax);
```

```
        } else {
            n->leftUp = construct(n->leftUp, points, lo, mid - 1, !partition, xmin, ymin, xmax, n->p.y);
            n->rightDown = construct(n->rightDown, points, mid + 1, hi, !partition, xmin, n->p.y, xmax, ymax);
        }
        return n;
    }
    Node *insert(Node *n, const Point &p, bool partition, T xmin, T ymin, T xmax, T ymax) {
        if (nullptr == n) {
            cnt++;
            return new Node(p, Rectangle(xmin, ymin, xmax, ymax));
        }
        if (n->p == p) return n;
        if (partition == VERTICAL) {
            if (Point::xOrderLt(p, n->p)) n->leftUp = insert(n->leftUp, p, !partition, xmin, ymin, n->p.x, ymax);
            else n->rightDown = insert(n->rightDown, p, !partition, n->p.x, ymin, xmax, ymax);
        } else {
            if (Point::yOrderLt(p, n->p)) n->leftUp = insert(n->leftUp, p, !partition, xmin, ymin, xmax, n->p.y);
            else n->rightDown = insert(n->rightDown, p, !partition, xmin, n->p.y, xmax, ymax);
        }
        return n;
    }
    bool contains(Node *n, const Point &p, bool partition) {
        if (nullptr == n) return false;
        if (n->p == p) return true;
        if (partition == VERTICAL) {
            if (Point::xOrderLt(p, n->p)) return contains(n->leftUp, p, !partition);
            else return contains(n->rightDown, p, !partition);
        } else {
            if (Point::yOrderLt(p, n->p)) return contains(n->leftUp, p, !partition);
            else return contains(n->rightDown, p, !partition);
        }
    }
    void range(Node *n, queue<Point> &q, const Rectangle &rect) {
        if (nullptr == n || !rect.intersects(n->r)) return;
        if (rect.contains(n->p)) q.push(n->p);
        range(n->leftUp, q, rect); range(n->rightDown, q, rect);
    }
    Point *findNearest(Node *n, const Point &p, Point *nearest) {
        if (nullptr == n || (nullptr != nearest && nearest->distanceSquaredTo(p) < n->r.distanceSquaredTo(p))) return nearest;
        if (nullptr == nearest || n->p.distanceSquaredTo(p) < nearest->distanceSquaredTo(p)) nearest = &(n->p);
        if (nullptr != n->leftUp && n->leftUp->r.contains(p)) {
            nearest = findNearest(n->leftUp, p, nearest);
            nearest = findNearest(n->rightDown, p, nearest);
        } else {
            nearest = findNearest(n->rightDown, p, nearest);
            nearest = findNearest(n->leftUp, p, nearest);
        }
        return nearest;
    }
    KdTree(T xmin, T ymin, T xmax, T ymax) : XMIN(xmin), YMIN(ymin), XMAX(xmax), YMAX(ymax), cnt(0), root(nullptr) {}
    template <class It> KdTree(T xmin, T ymin, T xmax, T ymax, It st, It en) :
            XMIN(xmin), YMIN(ymin), XMAX(xmax), YMAX(ymax), cnt(en - st) {
        root = construct(root, st, 0, cnt - 1, VERTICAL, XMIN, YMIN, XMAX, YMAX);
    }
    bool empty() { return cnt == 0; }
    int size() { return cnt; }
    void insert(const Point &p) { root = insert(root, p, VERTICAL, XMIN, YMIN, XMAX, YMAX); }
    bool contains(const Point &p) { return contains(root, p, VERTICAL); }
    queue<Point> range(const Rectangle &rect) { queue<Point> q; range(root, q, rect); return q; }
    Point *findNearest(const Point &p) {
        if (empty()) return nullptr;
        return findNearest(root, p, nullptr);
    }
};
```

1.4 Graph

1.4.1 Euler Tour Treap

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

// Euler Tour Treap supporting vertex updates and vertex/subtree queries
// Time Complexity:
//   constructor: O(N)
//   treeRoot, connected, addEdge, cutParent: O(log N)
//   updateVertex, queryVertexValue, querySubtreeValue: O(log N)
// Memory Complexity: O(N)
struct EulerTourTreap {
    seed_seq seq {
        (uint64_t) chrono::duration_cast<chrono::nanoseconds>(chrono::high_resolution_clock::now().time_since_epoch()).count(),
        (uint64_t) __builtin_ia32_rdtsc(),
        (uint64_t) (uintptr_t) make_unique<char>().get()
    };
    mt19937 rng; uniform_real_distribution<double> dis;
    using Data = int; const Data vdef = 0; const bool ISPRE = true, ISPOST = false;
    vector<Data> VAL, SBTR; vector<int> PRE, POST, VERT, L, R, P, SZ; vector<double> PRI; vector<bool> TYPE;
    int makeNode(int vert, bool type, Data val) {
        VAL.push_back(val); SBTR.push_back(val); VERT.push_back(vert); TYPE.push_back(type);
        L.push_back(-1); R.push_back(-1); P.push_back(-1); SZ.push_back(1); PRI.push_back(dis(rng));
        return int(VAL.size()) - 1;
    }
    int size(int x) { return x == -1 ? 0 : SZ[x]; }
    Data val(int x) { return x == -1 ? vdef : VAL[x]; }
    Data sbtrVal(int x) { return x == -1 ? vdef : SBTR[x]; }
```

```
    Data merge(Data l, Data r); // to be implemented
    Data applyVal(Data o, Data n); // to be implemented
    void apply(int x, Data d) {
        if (x == -1) return;
        VAL[x] = applyVal(VAL[x], d); SBTR[x] = applyVal(SBTR[x], d);
    }
    void update(int x) {
        if (x == -1) return;
        SZ[x] = 1; SBTR[x] = VAL[x];
        if (L[x] != -1) { P[L[x]] = x; SZ[x] += SZ[L[x]]; SBTR[x] = merge(SBTR[x], SBTR[L[x]]); }
        if (R[x] != -1) { P[R[x]] = x; SZ[x] += SZ[R[x]]; SBTR[x] = merge(SBTR[x], SBTR[R[x]]); }
    }
    void merge(int &x, int l, int r) {
        if (l == -1 || r == -1) { x = l == -1 ? r : l; }
        else if (PRI[l] > PRI[r]) { merge(R[l], R[l], r); x = l; }
        else { merge(L[r], l, L[r]); x = r; }
        update(x);
    }
    void split(int x, int &l, int &r, int lsz) {
        if (x == -1) { l = r = -1; return; }
        P[x] = -1;
        if (lsz <= size(L[x])) { split(L[x], l, L[x], lsz); r = x; }
        else { split(R[x], R[x], r, lsz - size(L[x]) - 1); l = x; }
        update(x);
    }
    int min(int x) {
        if (x == -1) return 0;
        while (L[x] != -1) x = L[x];
        return x;
    }
    int root(int x) {
        if (x == -1) return 0;
        while (P[x] != -1) x = P[x];
        return x;
    }
    int index(int x) {
        int ind = size(L[x]);
        for (; P[x] != -1; x = P[x]) if (L[P[x]] != x) ind += 1 + size(L[P[x]]);
        return ind;
    }
    EulerTourTreap(int N) : rng(seq), dis(0.0, 1.0) {
        for (int i = 0, dummy = -1; i < N; i++) {
            PRE.push_back(makeNode(i, ISPRE, vdef)); POST.push_back(makeNode(i, ISPOST, vdef));
            merge(dummy, PRE.back(), POST.back());
        }
    }
    template <class It> EulerTourTreap(It st, It en) : rng(seq), dis(0.0, 1.0) {
        int dummy = -1;
        for (It i = st; i < en; i++) {
            PRE.push_back(makeNode(i, ISPRE, *i)); POST.push_back(makeNode(i, ISPOST, *i));
            merge(dummy, PRE.back(), POST.back());
        }
    }
    int treeRoot(int v) { return VERT[min(root(PRE[v]))]; }
    bool connected(int v, int w) { return treeRoot(v) == treeRoot(w); }
    void addEdge(int v, int w) {
        int l, r; split(root(PRE[v]), l, r, index(PRE[v]) + 1);
        merge(l, l, root(PRE[w])); merge(l, l, r);
    }
    void cutParent(int v) {
        int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, index(POST[v]) + 1);
        merge(l, l, r);
    }
    Data getVertexValue(int v) {
        int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, 1);
        Data ret = val(m); merge(l, l, m); merge(l, l, r);
        return ret;
    }
    Data getSubtreeValue(int v) { // value may be doubled due to double counting of pre and post
        int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, index(POST[v]) + 1);
        Data ret = sbtrVal(m); merge(l, l, m); merge(l, l, r);
        return ret;
    }
    void updateVertex(int v, Data val) {
        int l, preV, m, postV, r; split(root(PRE[v]), l, preV, index(PRE[v])); split(preV, preV, m, 1);
        split(m, m, postV, index(POST[v])); split(postV, postV, r, 1);
        apply(preV, val); apply(postV, val); merge(l, l, preV); merge(l, l, m); merge(l, l, postV); merge(l, l, r);
    }
};
```

1.4.2 Lazy Euler Tour Treap

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

// Euler Tour Treap supporting vertex/subtree updates and vertex/subtree queries
// Time Complexity:
//   constructor: O(N)
//   treeRoot, connected, addEdge, cutParent: O(log N)
//   updateVertex, updateSubtreeValue, queryVertexValue, querySubtreeValue: O(log N)
// Memory Complexity: O(N)
struct LazyEulerTourTreap {
    seed_seq seq {
        (uint64_t) chrono::duration_cast<chrono::nanoseconds>(chrono::high_resolution_clock::now().time_since_epoch()).count(),
        (uint64_t) __builtin_ia32_rdtsc(),
        (uint64_t) (uintptr_t) make_unique<char>().get()
```

```
};
mt19937 rng; uniform_real_distribution<double> dis;
using Data = int; using Lazy = int; const Data vdef = 0; const Lazy ldef = 0; const bool ISPRE = true, ISPOST = false;
vector<Data> VAL, SBTR; vector<Lazy> LZ; vector<int> PRE, POST, VERT, L, R, P, SZ; vector<double> PRI; vector<bool> TYPE;
int makeNode(int vert, bool type, Data val) {
    VAL.push_back(val); SBTR.push_back(val); LZ.push_back(ldef); VERT.push_back(vert); TYPE.push_back(type);
    L.push_back(-1); R.push_back(-1); P.push_back(-1); SZ.push_back(1); PRI.push_back(dis(rng));
    return int(VAL.size()) - 1;
}
int size(int x) { return x == -1 ? 0 : SZ[x]; }
Data val(int x) { return x == -1 ? vdef: VAL[x]; }
Data sbtrVal(int x) { return x == -1 ? vdef : SBTR[x]; }
Data merge(Data l, Data r); // to be implemented
Lazy getSegmentVal(Lazy v, int k); // to be implemented
Lazy mergeLazy(Lazy l, Lazy r); // to be implemented
Data applyLazy(Data d, Lazy l); // to be implemented
void apply(int x, Lazy v) {
    if (x == -1) return;
    VAL[x] = applyLazy(VAL[x], v); SBTR[x] = applyLazy(SBTR[x], getSegmentVal(v, SZ[x])); LZ[x] = mergeLazy(LZ[x], v);
}
void propagate(int x) {
    if (x == -1 || LZ[x] == ldef) return;
    if (L[x] != -1) apply(L[x], LZ[x]);
    if (R[x] != -1) apply(R[x], LZ[x]);
    LZ[x] = ldef;
}
void update(int x) {
    if (x == -1) return;
    SZ[x] = 1; SBTR[x] = VAL[x];
    if (L[x] != -1) { P[L[x]] = x; SZ[x] += SZ[L[x]]; SBTR[x] = merge(SBTR[x], SBTR[L[x]]); }
    if (R[x] != -1) { P[R[x]] = x; SZ[x] += SZ[R[x]]; SBTR[x] = merge(SBTR[x], SBTR[R[x]]); }
}
void merge(int &x, int l, int r) {
    propagate(l); propagate(r);
    if (l == -1 || r == -1) { x = l == -1 ? r : l; }
    else if (PRI[l] > PRI[r]) { merge(R[l], R[l], r); x = l; }
    else { merge(L[r], l, L[r]); x = r; }
    update(x);
}
void split(int x, int &l, int &r, int lsz) {
    if (x == -1) { l = r = -1; return; }
    propagate(x); P[x] = -1;
    if (lsz <= size(L[x])) { split(L[x], l, L[x], lsz); r = x; }
    else { split(R[x], R[x], r, lsz - size(L[x]) - 1); l = x; }
    update(x);
}
int min(int x) {
    if (x == -1) return 0;
    while (L[x] != -1) x = L[x];
    return x;
}
int root(int x) {
    if (x == -1) return 0;
    while (P[x] != -1) x = P[x];
    return x;
}
int index(int x) {
    int ind = size(L[x]);
    for (; P[x] != -1; x = P[x]) if (L[P[x]] != x) ind += 1 + size(L[P[x]]);
    return ind;
}
LazyEulerTourTreap(int N) : rng(seq), dis(0.0, 1.0) {
    for (int i = 0, dummy = -1; i < N; i++) {
        PRE.push_back(makeNode(i, ISPRE, vdef)); POST.push_back(makeNode(i, ISPOST, vdef));
        merge(dummy, PRE.back(), POST.back());
    }
}
template <class It> LazyEulerTourTreap(It st, It en) : rng(seq), dis(0.0, 1.0) {
    int dummy = -1;
    for (It i = st; i < en; i++) {
        PRE.push_back(makeNode(i, ISPRE, *i)); POST.push_back(makeNode(i, ISPOST, *i));
        merge(dummy, PRE.back(), POST.back());
    }
}
int treeRoot(int v) { return VERT[min(root(PRE[v]))]; }
bool connected(int v, int w) { return treeRoot(v) == treeRoot(w); }
void addEdge(int v, int w) {
    int l, r; split(root(PRE[v]), l, r, index(PRE[v]) + 1);
    merge(l, l, root(PRE[w])); merge(l, l, r);
}
void cutParent(int v) {
    int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, index(POST[v]) + 1);
    merge(l, l, r);
}
Data getVertexValue(int v) {
    int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, 1);
    Data ret = val(m); merge(l, l, m); merge(l, l, r);
    return ret;
}
Data getSubtreeValue(int v) { // value may be doubled due to double counting of pre and post
    int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, index(POST[v]) + 1);
    Data ret = sbtrVal(m); merge(l, l, m); merge(l, l, r);
    return ret;
}
void updateVertex(int v, Data val) {
    int l, preV, m, postV, r; split(root(PRE[v]), l, preV, index(PRE[v])); split(preV, preV, m, 1);
```

```
        split(m, m, postV, index(POST[v])); split(postV, postV, r, l);
        apply(preV, val); apply(postV, val); merge(l, l, preV); merge(l, l, m); merge(l, l, postV); merge(l, l, r);
    }
    void updateSubtree(int v, Data val) {
        int l, m, r; split(root(PRE[v]), l, m, index(PRE[v])); split(m, m, r, index(POST[v]) + 1);
        apply(m, val); merge(l, l, m); merge(l, l, r);
    }
};
```

1.4.3 Link Cut Tree

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

// Link Cut Tree supporting vertex updates and path queries
// Time Complexity:
//   constructor: O(N)
//   makeRoot, findRoot, lca, link, cut, cutParent, updateVertex, queryPath: O(log N)
// Memory Complexity: O(N)
struct LinkCutTree {
    using Data = int; const Data vdef = 0;
    vector<Data> VAL, SBTR; vector<int> L, R, P, SZ; vector<bool> REV;
    int makeNode(Data val) {
        VAL.push_back(val); SBTR.push_back(val); REV.push_back(false);
        L.push_back(-1); R.push_back(-1); P.push_back(-1); SZ.push_back(1);
        return int(VAL.size()) - 1;
    }
    LinkCutTree(int N) { for (int i = 0; i < N; i++) makeNode(vdef); }
    template <class It> LinkCutTree(It st, It en) { for (It i = st; i < en; i++) makeNode(*i); }
    bool isRoot(int x) { return P[x] == -1 || (x != L[P[x]] && x != R[P[x]]); }
    int size(int x) { return x == -1 ? 0 : SZ[x]; }
    Data sbtr(int x) { return x == -1 ? vdef : SBTR[x]; }
    Data merge(Data l, Data r); // to be implemented
    Data applyVal(Data o, Data n); // to be implemented
    void apply(int x, Data d) {
        if (x == -1) return;
        VAL[x] = applyVal(VAL[x], d); SBTR[x] = applyVal(SBTR[x], d);
    }
    void propagate(int x) {
        if (REV[x]) {
            REV[x] = false; swap(L[x], R[x]);
            if (L[x] != -1) REV[L[x]] = !REV[L[x]];
            if (R[x] != -1) REV[R[x]] = !REV[R[x]];
        }
    }
    void update(int x) { SBTR[x] = merge(merge(sbtr(L[x]), VAL[x]), sbtr(R[x])); SZ[x] = 1 + size(L[x]) + size(R[x]); }
    void connect(int ch, int par, bool hasCh, bool isL) {
        if (ch != -1) P[ch] = par;
        if (hasCh) {
            if (isL) L[par] = ch;
            else R[par] = ch;
        }
    }
    void rotate(int x) {
        int p = P[x], g = P[p]; bool isRootP = isRoot(p); bool isL = x == L[p];
        connect(isL ? R[x] : L[x], p, true, isL); connect(p, x, true, !isL); connect(x, g, !isRootP, isRootP ? false : p == L[g]);
        update(p);
    }
    void splay(int x) {
        while (!isRoot(x)) {
            int p = P[x], g = P[p];
            if (!isRoot(p)) propagate(g);
            propagate(p); propagate(x);
            if (!isRoot(p)) rotate((x == L[p]) == (p == L[g]) ? p : x);
            rotate(x);
        }
        propagate(x); update(x);
    }
    int expose(int x) {
        int last = -1;
        for (int y = x; y != -1; y = P[y]) { splay(y); L[y] = last; last = y; }
        splay(x); return last;
    }
    void makeRoot(int x) { expose(x); REV[x] = !REV[x]; }
    int findRoot(int x) {
        expose(x);
        while (R[x] != -1) x = R[x];
        splay(x); return x;
    }
    bool connected(int x, int y) {
        if (findRoot(x) != findRoot(y)) return false;
        if (x == y) return true;
        expose(x); expose(y); return P[x] != -1;
    }
    int lca(int x, int y) {
        if (findRoot(x) != findRoot(y)) return -1;
        expose(x); return expose(y);
    }
    bool link(int x, int y) {
        if (connected(x, y)) return false;
        makeRoot(x); P[x] = y; return true;
    }
    bool cut(int x, int y) {
        if (!connected(x, y)) return false;
        makeRoot(x); expose(y);
```

```
        if (x != R[y] || L[x] != -1) return false;
        R[y] = P[R[y]] = -1; return true;
    }
    bool cutParent(int x) {
        expose(x);
        if (R[x] == -1) return false;
        R[x] = P[R[x]] = -1;
        return true;
    }
    bool updateVertex(int x, Data val) { makeRoot(x); apply(x, val); return true; }
    int queryPath(int from, int to) {
        if (!connected(from, to)) return vdef;
        makeRoot(from); expose(to); return sbtr(to);
    }
};
```

1.4.4 Lazy Link Cut Tree

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

// Link Cut Tree supporting path updates and path queries
// Time Complexity:
//   constructor: O(N)
//   makeRoot, findRoot, lca, link, cut, cutParent, updatePath, queryPath: O(log N)
// Memory Complexity: O(N)
struct LazyLinkCutTree {
    using Data = int; using Lazy = int; const Data vdef = 0; const Lazy ldef = 0;
    vector<Data> VAL, SBTR; vector<Lazy> LZ; vector<int> L, R, P, SZ; vector<bool> REV;
    int makeNode(Data val) {
        VAL.push_back(val); SBTR.push_back(val); LZ.push_back(ldef); REV.push_back(false);
        L.push_back(-1); R.push_back(-1); P.push_back(-1); SZ.push_back(1);
        return int(VAL.size()) - 1;
    }
    LazyLinkCutTree(int N) { for (int i = 0; i < N; i++) makeNode(vdef); }
    template <class It> LazyLinkCutTree(It st, It en) { for (It i = st; i < en; i++) makeNode(*i); }
    bool isRoot(int x) { return P[x] == -1 || (x != L[P[x]] && x != R[P[x]]); }
    int size(int x) { return x == -1 ? 0 : SZ[x]; }
    Data sbtr(int x) { return x == -1 ? vdef : SBTR[x]; }
    Data merge(Data l, Data r); // to be implemented
    Lazy getSegmentVal(Lazy v, int k); // to be implemented
    Lazy mergeLazy(Lazy l, Lazy r); // to be implemented
    Data applyLazy(Data d, Lazy l); // to be implemented
    void apply(int x, Lazy v) {
        if (x == -1) return;
        VAL[x] = applyLazy(VAL[x], v); SBTR[x] = applyLazy(SBTR[x], getSegmentVal(v, SZ[x])); LZ[x] = mergeLazy(LZ[x], v);
    }
    void propagate(int x) {
        if (REV[x]) {
            REV[x] = false; swap(L[x], R[x]);
            if (L[x] != -1) REV[L[x]] = !REV[L[x]];
            if (R[x] != -1) REV[R[x]] = !REV[R[x]];
        }
        VAL[x] = applyLazy(VAL[x], LZ[x]); SBTR[x] = applyLazy(SBTR[x], getSegmentVal(LZ[x], SZ[x]));
        if (L[x] != -1) LZ[L[x]] = mergeLazy(LZ[L[x]], LZ[x]);
        if (R[x] != -1) LZ[R[x]] = mergeLazy(LZ[R[x]], LZ[x]);
        LZ[x] = ldef;
    }
    void update(int x) {
        SBTR[x] = merge(merge(sbtr(L[x]), applyLazy(VAL[x], LZ[x])), sbtr(R[x]));
        SZ[x] = 1 + size(L[x]) + size(R[x]);
    }
    void connect(int ch, int par, bool hasCh, bool isL) {
        if (ch != -1) P[ch] = par;
        if (hasCh) {
            if (isL) L[par] = ch;
            else R[par] = ch;
        }
    }
    void rotate(int x) {
        int p = P[x], g = P[p]; bool isRootP = isRoot(p); bool isL = x == L[p];
        connect(isL ? R[x] : L[x], p, true, isL); connect(p, x, true, !isL); connect(x, g, !isRootP, isRootP ? false : p == L[g]);
        update(p);
    }
    void splay(int x) {
        while (!isRoot(x)) {
            int p = P[x], g = P[p];
            if (!isRoot(p)) propagate(g);
            propagate(p); propagate(x);
            if (!isRoot(p)) rotate((x == L[p]) == (p == L[g]) ? p : x);
            rotate(x);
        }
        propagate(x); update(x);
    }
    int expose(int x) {
        int last = -1;
        for (int y = x; y != -1; y = P[y]) { splay(y); L[y] = last; last = y; }
        splay(x); return last;
    }
    void makeRoot(int x) { expose(x); REV[x] = !REV[x]; }
    int findRoot(int x) {
        expose(x);
        while (R[x] != -1) x = R[x];
        splay(x); return x;
    }
    bool connected(int x, int y) {
```

```
        if (findRoot(x) != findRoot(y)) return false;
        if (x == y) return true;
        expose(x); expose(y); return P[x] != -1;
    }
    int lca(int x, int y) {
        if (findRoot(x) != findRoot(y)) return -1;
        expose(x); return expose(y);
    }
    bool link(int x, int y) {
        if (connected(x, y)) return false;
        makeRoot(x); P[x] = y; return true;
    }
    bool cut(int x, int y) {
        if (!connected(x, y)) return false;
        makeRoot(x); expose(y);
        if (x != R[y] || L[x] != -1) return false;
        R[y] = P[R[y]] = -1; return true;
    }
    bool cutParent(int x) {
        expose(x);
        if (R[x] == -1) return false;
        R[x] = P[R[x]] = -1;
        return true;
    }
    bool updatePath(int from, int to, Lazy val) {
        if (!connected(from, to)) return vdef;
        makeRoot(from); expose(to); apply(to, val); return sbtr(to);
    }
    int queryPath(int from, int to) {
        if (!connected(from, to)) return vdef;
        makeRoot(from); expose(to); return sbtr(to);
    }
};
```

1.4.5 Top Tree

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

struct Data {
    int mn, mx, sum, size;
    Data(int mn = INT_MAX, int mx = INT_MIN, int sum = 0, int size = 0) : mn(mn), mx(mx), sum(sum), size(size) {}
};
struct Lazy {
    int lz, fl;
    Lazy(int lz = 0, int fl = 0): lz(lz), fl(fl) {}
};

// Top Tree supporting path and subtree, updates and queries
// Time Complexity:
//   constructor: O(N)
//   makeRoot, findParent, link, cutParent: O(log N)
//   updatePath, updateSubtree, queryPath, querySubtree: O(log N)
// Memory Complexity: O(N)
struct TopTree {
    const int INC = 1, ASSIGN = 2;
    Data merge(const Data &l, const Data &r) {
        return Data(min(l.mn, r.mn), max(l.mx, r.mx), l.sum + r.sum, l.size + r.size);
    }
    int apply(int val, int lz, int fl) {
        if (fl == ASSIGN) return lz;
        else if (fl == INC) return val + lz;
        else return val;
    }
    Data applyLazy(const Data &d, const Lazy &lz) {
        return d.size ? Data(apply(d.mn, lz.lz, lz.fl), apply(d.mx, lz.lz, lz.fl), apply(d.sum, lz.lz * d.size, lz.fl), d.size) : d;
    }
    Lazy mergeLazy(const Lazy &ch, const Lazy &par) {
        return Lazy(apply(ch.lz, par.lz, par.fl), max(ch.fl, par.fl));
    }
    vector<Data> PATH, SBTR, FTR; vector<Lazy> LZPATH, LZSBTR;
    vector<array<int, 4>> CH; vector<int> VAL, P; vector<bool> REV, INR;
    int makeNode(int val, bool inr) {
        VAL.push_back(val); Data d = inr ? Data() : Data(val, val, val, 1); Lazy lz = Lazy(0, 0);
        PATH.push_back(d); SBTR.push_back(d); FTR.push_back(d); LZPATH.push_back(lz); LZSBTR.push_back(lz);
        CH.push_back({0, 0, 0, 0}); P.push_back(0); REV.push_back(false); INR.push_back(inr);
        return int(VAL.size()) - 1;
    }
    TopTree(int N) { makeNode(0, false); for (int i = 1; i <= N; i++) makeNode(0, false); }
    template <class It> TopTree(It st, It en) { makeNode(0, false); for (It i = st; i < en; i++) makeNode(*i, false); }
    bool isRoot(int x, bool t) {
        if (t) return !P[x] || !INR[x] || !INR[P[x]];
        else return !P[x] || (x != CH[P[x]][0] && x != CH[P[x]][1]);
    }
    void rev(int x) { REV[x] = !REV[x]; swap(CH[x][0], CH[x][1]); }
    void pushPath(int x, const Lazy &lz) {
        LZPATH[x] = mergeLazy(LZPATH[x], lz); PATH[x] = applyLazy(PATH[x], lz);
        VAL[x] = apply(VAL[x], lz.lz, lz.fl); SBTR[x] = merge(PATH[x], FTR[x]);
    }
    void pushSbtr(int x, const Lazy &lz, bool ftr) {
        LZSBTR[x] = mergeLazy(LZSBTR[x], lz); SBTR[x] = applyLazy(SBTR[x], lz); FTR[x] = applyLazy(FTR[x], lz);
        if (ftr) pushPath(x, lz);
    }
    void propagate(int x) {
        if (x) {
            if (REV[x]) {
```

```
            REV[x] = false;
            for (int i = 0; i < 2; i++) if (CH[x][i]) rev(CH[x][i]);
        }
        for (int i = 0; i < 4; i++) if (CH[x][i]) pushSbtr(CH[x][i], LZSBTR[x], i >= 2);
        for (int i = 0; i < 2; i++) if (CH[x][i]) pushPath(CH[x][i], LZPATH[x]);
        LZSBTR[x] = LZPATH[x] = Lazy();
    }
}
void update(int x) {
    if (x) {
        PATH[x] = SBTR[x] = FTR[x] = Data();
        if (!INR[x]) PATH[x] = SBTR[x] = Data(VAL[x], VAL[x], VAL[x], 1);
        for (int i = 0; i < 2; i++) {
            if (CH[x][i]) {
                PATH[x] = merge(PATH[x], PATH[CH[x][i]]);
                FTR[x] = merge(FTR[x], FTR[CH[x][i]]);
            }
        }
        for (int i = 0; i < 4; i++) if (CH[x][i]) SBTR[x] = merge(SBTR[x], SBTR[CH[x][i]]);
        for (int i = 2; i < 4; i++) if (CH[x][i]) FTR[x] = merge(FTR[x], SBTR[CH[x][i]]);
    }
}
int findInd(int x) {
    for (int i = 0; i < 4; i++) if (CH[P[x]][i] == x) return i;
    assert(false); return -1;
}
void connect(int ch, int par, int i) {
    if (ch) P[ch] = par;
    CH[par][i] = ch;
}
void rotate(int x, int t) {
    assert(t == 0 || t == 2);
    int p = P[x], g = P[p];
    bool isL = x == CH[p][t];
    if (g) connect(x, g, findInd(p));
    else P[x] = 0;
    connect(CH[x][t ^ isL], p, t ^ !isL); connect(p, x, t ^ isL); update(p);
}
void splay(int x, int t) {
    assert(t == 0 || t == 2);
    while (!isRoot(x, t)) {
        int p = P[x], g = P[p];
        if (!isRoot(p, t)) rotate((x == CH[p][t]) == (p == CH[g][t]) ? p : x, t);
        rotate(x, t);
    }
    update(x);
}
void add(int x, int y) {
    propagate(y);
    for (int i = 2; i < 4; i++) if (!CH[y][i]) { connect(x, y, i); return; }
    int z = makeNode(0, true), w = 0;
    for(w = y; INR[CH[w][2]]; w = CH[w][2]) propagate(CH[w][2]);
    connect(CH[w][2], z, 2); connect(x, z, 3); connect(z, w, 2); splay(z, 2);
}
void remove(int x) {
    if (INR[P[x]]) { connect(CH[P[x]][findInd(x) ^ 1], P[P[x]], findInd(P[x])); splay(P[P[x]], 2); }
    else connect(0, P[x], findInd(x));
    P[x] = 0;
}
void expose(int x) {
    stack<int> stk; int y = x, z;
    for (z = x; z; z = P[z]) stk.push(z);
    while (!stk.empty()) { propagate(stk.top()); stk.pop(); }
    splay(x, 0);
    if (CH[x][1]) { z = CH[x][1]; CH[x][1] = 0; add(z, x); update(x); }
    while (P[x]) {
        for (z = P[x]; INR[z]; z = P[z]);
        splay(z, 0);
        if (CH[z][1]) { connect(CH[z][1], P[x], findInd(x)); splay(P[x], 2); }
        else remove(x);
        connect(x, z, 1); update(z); x = z;
    }
    splay(y, 0);
}
void makeRoot(int x) { expose(x); rev(x); }
int findParent(int x) {
    expose(x); propagate(CH[x][0]);
    for (x = CH[x][0]; x && CH[x][1]; x = CH[x][1]) propagate(CH[x][1]);
    return x;
}
int findRoot(int x) { for (; P[x]; x = P[x]); return x; }
int cutParent(int x) {
    int y = findParent(x);
    if (y) { expose(y); remove(x); update(y); }
    return y;
}
void link(int x, int y) {
    makeRoot(x); int p = cutParent(x);
    if (findRoot(x) != findRoot(y)) p = y;
    if (p) { expose(p); add(x, p); update(p); }
}
void updateSubtree(int x, Lazy lz) {
    expose(x); VAL[x] = apply(VAL[x], lz.lz, lz.fl);
    for (int i = 2; i < 4; i++) if (CH[x][i]) pushSbtr(CH[x][i], lz, true);
    update(x);
}
```

```
    void updatePath(int x, int y, Lazy lz) { makeRoot(x); expose(y); splay(x, 0); pushPath(x, lz); }
    Data querySubtree(int x) {
        expose(x); Data ret = (VAL[x], VAL[x], VAL[x], 1);
        for (int i = 2; i < 4; i++) ret = merge(ret, SBTR[CH[x][i]]);
        return ret;
    }
    Data queryPath(int x, int y) { makeRoot(x); expose(y); splay(x, 0); return PATH[x]; }
};
```

1.5 String

1.5.1 Trie

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

struct Trie {
    struct Node { unordered_map<char, Node*> child; int cnt = 0, prefixCnt = 0; };
    void insert(Node *root, string &s) {
        Node *cur = root;
        for (char c : s) {
            if (cur->child.count(c)) cur = cur->child[c];
            else cur = cur->child[c] = new Node();
            cur->prefixCnt++;
        }
        cur->cnt++;
    }
    int count(Node *root, string &s) {
        Node *cur = root;
        for (char c : s) {
            if (cur->child.count(c)) cur = cur->child[c];
            else return 0;
        }
        return cur->cnt;
    }
    int prefixCount(Node *root, string &s) {
        Node *cur = root;
        for (char c : s) {
            if (cur->child.count(c)) cur = cur->child[c];
            else return 0;
        }
        return cur->prefixCnt;
    }
    Node *root = new Node();
    Trie() {}
    void insert(string &s) { insert(root, s); }
    bool contains(string &s) { return count(root, s) >= 1; }
    bool hasPrefix(string &s) { return prefixCount(root, s) >= 1; }
    int count(string &s) { return count(root, s); }
    int prefixCount(string &s) { return prefixCount(root, s); }
};
```

2 Algorithms

2.1 Dynamic Programming

2.1.1 Coin Change

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

// Computes the number of ways to make chance for exactly T dollars given N coin values
// Time Complexity: O(TN)
// Memory Complexity: O(T + N)
template <const int MAXT, const int MAXN> struct CoinChange {
    int dp[MAXT], C[MAXN];
    int solve(int T, int N) {
        dp[0] = 1; fill(dp + 1, dp + T + 1, 0);
        for (int i = 0; i < N; i++) for (int j = 1; j <= T; j++) if (C[i] <= j) dp[j] += dp[j - C[i]];
        return dp[T];
    }
};
```

2.1.2 Minimum Coin Change

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

// Computes the minimum number of coins to make chance for exactly T dollars given N coin values
// Time Complexity: O(TN)
// Memory Complexity: O(T + N)
template <const int MAXT, const int MAXN> struct CoinChange {
    int dp[MAXT], C[MAXN];
    int solve(int T, int N) {
        dp[0] = 0; fill(dp + 1, dp + T + 1, INT_MAX - 1);
        for (int i = 1; i <= T; i++) for (int j = 0; j < N; j++) if (C[j] <= i) dp[i] = min(dp[i], dp[i - C[j]] + 1);
        return dp[T];
    }
};
```

2.1.3 Egg Dropping

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

// Computes the number of trials required to determine which of the F floors will
// an egg break on, using E eggs, assuming eggs cannot be resued if broken
// Time Complexity:
//   preCompute: O(E^2)
//   query: O(log E)
//   querySingle: O(E log F)
// Memory Complexity: O(E^2)
template <const int MAXE> struct EggDropping {
    long long dp[MAXE][MAXE];
    void preCompute() {
        for (int i = 1; i < MAXE; i++) {
            dp[i][0] = 0; dp[i][1] = 1;
            for (int j = 2; j < MAXE; j++) dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1] + 1;
        }
    }
    int query(int E, int F) { return lower_bound(dp[E], dp[E] + MAXE, F) - dp[E]; }
    long long f(int x, int E, int F) {
        long long sum = 0, term = 1;
        for (int i = 1; i <= E && sum < F; i++) sum += ((term *= x - i + 1) /= i);
        return sum;
    }
    int querySingle(int E, int F) {
        int lo = 1, hi = F, mid;
        while (lo < hi) {
            mid = lo + (hi - lo) / 2;
            if (f(mid, E, F) < F) lo = mid + 1;
            else hi = mid;
        }
        return lo;
    }
};
```

2.1.4 Hamiltonian Cycle

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

// Computes the shortest Hamiltonian Cycle (returning to starting position)
// Time Complexity: O(N^2 * 2^N)
// Memory Complexity: O(N * 2^N)
template <const int MAXN, class T> struct HamiltonianCycle {
    T INF, dist[MAXN][MAXN], dp[1 << MAXN][MAXN]; int order[MAXN];
    HamiltonianCycle(T INF) : INF(INF) {}
    T solve(int N) {
        T ret = INF;
        for (int i = 0; i < (1 << N); i++) for (int j = 0; j < N; j++) dp[i][j] = INF;
        dp[1][0] = 0;
        for (int i = 1; i < (1 << N); i += 2) for (int j = 0; j < N; j++) if (i & (1 << j))
            for (int k = 1; k < N; k++) if (!(i & (1 << k))) dp[i ^ (1 << k)][k] = min(dp[i ^ (1 << k)][k], dp[i][j] + dist[j][k]);
        for (int i = 1; i < N; i++) ret = min(ret, dp[(1 << N) - 1][i] + dist[i][0]);
        int curPos = 0, curState = (1 << N) - 1;
        for (int i = N - 1; i >= 0; i--) {
            int next = -1;
            for (int j = 0; j < N; j++)
                if ((curState & (1 << j)) && (next == -1 || dp[curState][j] + dist[j][curPos] < dp[curState][next] + dist[next][
                    curPos]))
                    next = j;
            order[i] = curPos = next; curState ^= 1 << curPos;
        }
        return ret;
    }
};
```

2.1.5 Hamiltonian Path

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

// Computes the shortest Hamiltonian Path (does not have to return to starting position)
// Time Complexity: O(N^2 * 2^N)
// Memory Complexity: O(N * 2^N)
template <const int MAXN, class T> struct HamiltonianPath {
    T INF, dist[MAXN][MAXN], dp[1 << MAXN][MAXN]; int order[MAXN];
    HamiltonianPath(T INF) : INF(INF) {}
    T run(int N) {
        T ret = INF;
        for (int i = 0; i < (1 << N); i++) for (int j = 0; j < N; j++) dp[i][j] = INF;
        for (int i = 0; i < N; i++) dp[1 << i][i] = 0;
        for (int i = 1; i < (1 << N); i++) for (int j = 0; j < N; j++) if (i & (1 << j))
            for (int k = 0; k < N; k++) if (!(i & (1 << k))) dp[i ^ (1 << k)][k] = min(dp[i ^ (1 << k)][k], dp[i][j] + dist[j][k]);
        for (int i = 0; i < N; i++) ret = min(ret, dp[(1 << N) - 1][i]);
        int curState = (1 << N) - 1, last = -1;
        for (int i = N - 1; i >= 0; i--) {
            int next = -1;
            for (int j = 0; j < N; j++)
                if ((curState & (1 << j)) && (next == -1 || dp[curState][j] + (last == -1 ? 0 : dist[j][last]) < dp[curState][next]
                    + (last == -1 ? 0 : dist[next][last])))
                    next = j;
            order[i] = last = next; curState ^= 1 << last;
        }
```

```
        return ret;
    }
};
```

2.1.6 0-1 Knapsack

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

// Computes the maximum value that can be obtained by items in a knapsack (from a selection of N items)
// that can hold a maximum of M weight, not allowing for repeated instances of items
// Time Complexity: O(NM)
// Space Complexity: O(N + M)
template <const int MAXN, const int MAXM, class TV> struct ZeroOneKnapsack {
    int W[MAXN]; TV V[MAXN], dp[MAXM];
    TV solve(int N, int M) {
        fill(dp, dp + M + 1, 0);
        for (int i = 0; i < N; i++) for (int j = M; j >= W[i]; j--) dp[j] = max(dp[j - W[i]] + V[i]);
        return dp[M];
    }
};
```

2.1.7 Unbounded Knapsack

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

// Computes the maximum value that can be obtained by items in a knapsack (from a selection of N items)
// that can hold a maximum of M weight, allowing for repeated instances of items
// Time Complexity: O(NM)
// Space Complexity: O(N + M)
template <const int MAXN, const int MAXM, class TV> struct UnboundedKnapsack {
    int W[MAXN]; TV V[MAXN], dp[MAXM];
    TV solve(int N, int M) {
        for (int j = 0; j <= M; j++) for (int i = 0; i < N; i++) if (W[i] <= j) dp[j] = max(dp[j], dp[j - W[i]] + V[i]);
        return dp[M];
    }
};
```

2.1.8 Bounded Knapsack

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

// Computes the maximum value that can be obtained by item in a knapsack (from a selection of N items)
// that can hold a maximum of M weight, allowing for a maximum of Fi items for item i
// Time Complexity: O(NM log M)
// Space Complexity: O(N log M + M)
template <const int MAXN, const int MAXM, class TV> struct BoundedKnapsack {
    TV dp[MAXM]; vector<pair<int, TV>> items;
    void addItem(int w, TV v, int f) {
        int sum = 0, mult = 1;
        for (f = min(f, MAXM / w); sum + mult < f; sum += mult, mult *= 2) items.emplace_back(mult * w, mult * v);
        items.emplace_back((f - sum) * w, (f - sum) * v);
    }
    void reset() { items.clear(); }
    TV solve(int N, int M) {
        fill(dp, dp + M + 1, 0);
        for (auto &&i : items) for (int j = M; j >= i.first; j--) dp[j] = max(dp[j], dp[j - i.first] + i.second);
        return dp[M];
    }
};
```

2.1.9 ZigZag

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

// Computes the length of the longest zigzag subsequence
// A zigzag sequence alternates between increasing and decreasing (it can start with either)
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct ZigZag {
    int A[MAXN], dp[2][MAXN];
    int solve(int N) {
        dp[0][0] = dp[1][0] = 1;
        for (int i = 1; i < N; i++) {
            if (A[i] > A[i - 1]) { dp[0][i] = max(dp[1][i - 1] + 1, dp[0][i - 1]); dp[1][i] = dp[1][i - 1]; }
            else if (A[i] < A[i - 1]) { dp[1][i] = max(dp[0][i - 1] + 1, dp[1][i - 1]); dp[0][i] = dp[0][i - 1]; }
            else { dp[0][i] = dp[0][i - 1]; dp[1][i] = dp[1][i - 1]; }
        }
        return max(dp[0][N - 1], dp[1][N - 1]);
    }
};
```

2.1.10 Unique ZigZag

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

// Computes the number of unique zigzag sequences of a specified length N, modulo a number
// A zigzag sequence alternates between increasing and decreasing (it can start with either)
```

```cpp
// Time Complexity: O(N^2)
// Memory Complexity: O(N)
template <const int MAXN, const long long MOD> struct UniqueZigZag {
    long long dp[2][MAXN];
    long long solve(int N) {
        dp[0][0] = dp[1][0] = 0; dp[1][1] = 1;
        for (int i = 2; i <= N; i++) for (int j = 1; j <= i; j++) {
            if (i % 2 == 0) { dp[0][j] = (dp[1][j - 1] + dp[0][j - 1]) % MOD; dp[0][j + 1] = dp[0][j]; }
            else { dp[1][j] = (dp[0][i] - dp[0][j - 1] + dp[1][j - 1] + MOD) % MOD; dp[1][j + 1] = dp[1][j]; }
        }
        return dp[N % 2][N] % MOD;
    }
};
```

2.1.11 Longest Common Integer Subsequence

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

// Computes the longest common subsequence between 2 arrays of integers
// Time Complexity: O(N^2)
// Memory Complexity: O(N^2)
template <const int MAXN, const int MAXM = MAXN> struct LCIS {
    int A[MAXN], B[MAXM], dp[MAXN][MAXM], subsequence[max(MAXN, MAXM)], len;
    int solve(int N, int M) {
        for (int i = 1; i <= N; i++) for (int j = 1; j <= M; j++) {
            if (A[i - 1] == B[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;
            else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
        len = dp[N][M];
        for (int i = N, j = M, k = len - 1; i > 0 && j > 0;) {
            if (A[i - 1] == B[j - 1]) { subsequence[k--] = A[i - 1]; i--; j--; }
            else if (dp[i - 1][j] > dp[i][j - 1]) i--;
            else j--;
        }
        return len;
    }
};
```

2.1.12 Longest Increasing Subsequence

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

// Computes the length of the longest increasing subsequence
// Time Complexity: O(N log N)
// Memory Complexity: O(N)
template <const int MAXN> struct LongestIncreasingSubsequence {
    int A[MAXN], dp[MAXN];
    int solve(int N) {
        int ret = 0;
        for (int i = 0; i < N; i++) {
            int j = lower_bound(dp, dp + ret, A[i]) - dp;
            dp[j] = A[i]; if (j == ret) ret++;
        }
        return ret;
    }
};
```

2.1.13 Maximum Nonconsecutive Subsequence Sum

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

// Computes the maximum sum of a non sonsecutive subsequence
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxNonConsecutiveSum {
    int A[MAXN], dp[MAXN];
    int solve(int N) {
        dp[0] = A[0]; dp[1] = max(dp[0], A[1]);
        for (int i = 2; i < N; i++) dp[i] = max(dp[i - 2] + A[i], dp[i - 1]);
        return dp[N - 1];
    }
};
```

2.1.14 Maximum Subarray Sum

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

// Computes the subarray with the maximum sum using Kadane's Algorithm
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxSubarraySum {
    int A[MAXN], maxSum, st, en; // st and en are the starting and ending (inclusive) indices of the subarray
    int solve(int N) {
        maxSum = st = 0; en = -1; int curMax = 0, curSt = 0;
        for (int i = 0; i < N; i++) {
            curMax += A[i]; if (curMax < 0) { curMax = 0; curSt = i + 1; }
            if (maxSum < curMax) { maxSum = curMax; st = curSt; en = i; }
        }
        return maxSum;
    }
```

```
};
```

2.1.15 Maximum Subarray Sum With Skip

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

// Computes the subarray with the maximum sum, removing at most one element from the array
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxSubarraySumSkip {
    int A[MAXN], fw[MAXN], bw[MAXN];
    int solve(int N) {
        int curMax = fw[0] = A[0], maxSum = A[0];
        for (int i = 1; i < N; i++) { fw[i] = curMax = max(A[i], curMax + A[i]); maxSum = max(maxSum, curMax); }
        curMax = maxSum = bw[N - 1] = A[N - 1];
        for (int i = N - 2; i >= 0; i--) { bw[i] = curMax = max(A[i], curMax + A[i]); maxSum = max(maxSum, curMax); }
        for (int i = 1; i < N - 1; i++) maxSum = max(maxSum, fw[i - 1] + bw[i + 1]);
        return maxSum;
    }
};
```

2.1.16 Maximum Zero Submatrix

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

// Computes the area of the largest submatrix that contains only 0s
// Time Complexity: O(NM)
// Memory Complexity: O(NM)
template <const int MAXN, const int MAXM = MAXN> struct MaxZeroSubmatrix {
    int A[MAXN][MAXM], H[MAXN][MAXM];
    int solve(int N, int M) {
        stack<int> s; int ret = 0;
        for (int j = 0; j < M; j++) for (int i = N - 1; i >= 0; i--) H[i][j] = A[i][j] ? 0 : 1 + (i == N - 1 ? 0 : H[i + 1][j]);
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                int minInd = j;
                while (!s.empty() && H[i][s.top()] >= H[i][j]) {
                    ret = max(ret, (j - s.top()) * (H[i][s.top()]));
                    minInd = s.top(); s.pop(); H[i][minInd] = H[i][j];
                }
                s.push(minInd);
            }
            while (!s.empty()) ret = max(ret, (M - s.top()) * H[i][s.top()]); s.pop();
        }
        return ret;
    }
};
```

2.1.17 Partitions

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

// counts the number of partitions of a number such that the sum is equal to n
// Time Complexity: O(N^2)
// Memory Complexity: O(N)
template <const int MAXN> struct Partitions1 {
    long long dp[MAXN];
    long long solve(int N, long long mod) {
        dp[0] = 1;
        for (int i = 1; i <= N; i++) for (int j = i; j <= N; j++) dp[j] = (dp[j] + dp[j - 1]) % mod;
        return dp[N] % mod;
    }
};

// counts the number of partitions of size k, for a number such that the sum is equal to n
// Time Complexity: O(NK)
// Memory Complexity: O(NK)
template <const int MAXN, const int MAXK> struct Partitions2 {
    long long dp[MAXN][MAXK];
    long long solve(int N, int K, long long mod) {
        dp[0][1] = 1;
        for (int i = 1; i <= N; i++) for (int j = 1; j <= min(i, K); j++) dp[i][j] = (dp[i - 1][j - 1] + dp[i - j][j]) % mod;
        return dp[N][K] % mod;
    }
};
```

2.1.18 Convex Hull Optimization

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

// Supports adding lines in the form f(x) = mx + b and finding the maximum value of f(x) at any given x
// Comparator convention is same as priority_queue in STL
// Time Complexity:
//   addLine, getMax: O(1)
// Memory Complexity: O(N) where N is the total number of lines added
template <const int MAXN, class T, class Comparator = less<T>> struct ConvexHullOptimization {
    Comparator cmp; T M[MAXN], B[MAXN]; int front = 0, back = 0;
    void addLine(T m, T b) { // in non decreasing order of slope, as sorted by the comparator
        while (back >= 2 && (B[back - 2] - B[back - 1]) * (m - M[back - 1]) >= (B[back - 1] - b) * (M[back - 1] - M[back - 2])) back
            --;
```

```
        M[back] = m, B[back++] = b;
    }
    T getMax(T x) { // in non decreasing order of x, regardless of comparator
        while (front < back - 1 && !cmp(M[front + 1] * x + B[front + 1], M[front] * x + B[front])) front++;
        return M[front] * x + B[front];
    }
};
```

2.1.19 Knuth Optimization

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

// Knuth's Dynamic Programming Optimization
// Must statisfy dp[i][j] = min(dp[i][k] + dp[k][j] + cost(i, k, j)) for i <= k <= j
// and A[i][j - 1] <= A[i][j] <= A[i + 1][j], where A[i][j] is the optimal value of k
// for dp[i][j]
// Time Complexity: Reduces the runtime from O(N^3) to O(N^2)
// Memory Complexity: O(N^2)
template <const int MAXN, const int INF> struct KnuthOptimization {
    int dp[MAXN][MAXN], mid[MAXN][MAXN];
    int cost(int l, int m, int r); // to be implemented
    int solve(int N) {
        for (int l = N - 1; l >= 0; l--) for (int r = l; r <= N; r++) {
            if (r - l < 2) { dp[l][r] = 0; mid[l][r] = l; continue; }
            dp[l][r] = INF;
            for (int m = mid[l][r - 1]; m <= mid[l + 1][r]; m++) {
                int temp = dp[l][m] + dp[m][r] + cost(l, m ,r);
                if (dp[l][r] > temp) { dp[l][r] = temp; mid[l][r] = m; }
            }
        }
        return dp[0][N];
    }
};
```

2.2 Meet in the Middle

2.2.1 Closest Subset Sum

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

// Finds the closest non-empty subset sum to a value, using a meet in the middle approach
// Time Complexity: Reduces the runtime from O(N * 2^N) to O(N * 2^(N/2))
// Memory Complexity: O(2^(N/2))
template <const int MAXN, class T> struct ClosestSubsetSum {
    T A[MAXN]; vector<T> ans; // ans contains the closest non-empty subset sum to value
    void solveHalf(vector<T> &half, vector<pair<T, int>> &sum) {
        for (int i = 1; i < (1 << int(half.size())); i++) {
            T curSum = 0;
            for (int j = 0; j < int(half.size()); j++) if (i & (1 << j)) curSum += half[j];
            sum.push_back(make_pair(curSum, i));
        }
        sort(sum.begin(), sum.end()); sum.resize(unique(sum.begin(), sum.end()) - sum.begin());
    }
    T solve(int N, T value) { // returns the closest sum, elements are stored in ans
        ans.clear(); T minDiff = numeric_limits<T>::max(), closestSum; int evenPerm, oddPerm;
        vector<T> even, odd; even.reserve(N - N / 2); odd.reserve(N / 2);
        vector<pair<T, int>> evenSum, oddSum; evenSum.reserve(1 << (N - N / 2)); oddSum.reserve(1 << (N / 2));
        for (int i = 0; i < N; i++) {
            if (i % 2 == 0) even.push_back(A[i]);
            else odd.push_back(A[i]);
        }
        solveHalf(even, evenSum); solveHalf(odd, oddSum);
        if (int(evenSum.size()) > 0 && abs(value - evenSum[0].first) < minDiff) {
            evenPerm = evenSum[0].second; oddPerm = 0; closestSum = evenSum[0].first; minDiff = abs(value - evenSum[0].first);
        }
        if (int(oddSum.size()) > 0 && abs(value - oddSum[0].first) < minDiff) {
            evenPerm = 0; oddPerm = oddSum[0].second; closestSum = oddSum[0].first; minDiff = abs(value - oddSum[0].first);
        }
        for (int i = 0; i < int(evenSum.size()); i++) {
            int j = lower_bound(oddSum.begin(), oddSum.end(), make_pair(value - evenSum[i].first, -1)) - oddSum.begin();
            if (j == int(oddSum.size())) {
                if (j > 0 && abs(value - (evenSum[i].first + oddSum[j - 1].first)) < minDiff) {
                    evenPerm = evenSum[i].second; oddPerm = oddSum[j - 1].second;
                    closestSum = evenSum[i].first + oddSum[j - 1].first;
                    minDiff = abs(value - (evenSum[i].first + oddSum[j - 1].first));
                }
            } else if (evenSum[i].first + oddSum[j].first != value) {
                if (j > 0 && abs(value - (evenSum[i].first + oddSum[j - 1].first)) < minDiff) {
                    evenPerm = evenSum[i].second; oddPerm = oddSum[j - 1].second;
                    closestSum = evenSum[i].first + oddSum[j - 1].first;
                    minDiff = abs(value - (evenSum[i].first + oddSum[j - 1].first));
                }
                if (abs(value - (evenSum[i].first + oddSum[j].first)) < minDiff) {
                    evenPerm = evenSum[i].second; oddPerm = oddSum[j].second;
                    closestSum = evenSum[i].first + oddSum[j].first;
                    minDiff = abs(value - (evenSum[i].first + oddSum[j].first));
                }
            } else {
                if (abs(value - (evenSum[i].first + oddSum[j].first)) < minDiff) {
                    evenPerm = evenSum[i].second; oddPerm = oddSum[j].second;
                    closestSum = evenSum[i].first + oddSum[j].first;
```

```
                    minDiff = abs(value - (evenSum[i].first + oddSum[j].first));
                }
            }
        }
        for (int i = 0; i < int(even.size()); i++) if (evenPerm & (1 << i)) ans.push_back(i * 2);
        for (int i = 0; i < int(odd.size()); i++) if (oddPerm & (1 << i)) ans.push_back(i * 2 + 1);
        return closestSum;
    }
};
```

2.2.2 Minimum Subset Difference

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

// Divides a set into two disjoint non-empty subsets with the smallest absolute difference
// of sums between them, using a meet in the middle approach
// Time Complexity: Reduces the runtime from O(N * 3^N) to O(N * 3^(N/2))
// Memory Complexity: O(3^(N/2))
template <const int MAXN, class T> struct MinSubsetDifference {
    T A[MAXN]; vector<T> setA, setB; // setA and setB contain the two disjoint non-empty subset
    void solveHalf(vector<T> &half, vector<pair<T, int>> &diff) {
        int perms = 1;
        for (int i = 0; i < int(half.size()); i++) perms *= 3;
        for (int i = 1; i < perms; i++) {
            int cur = i; T curDiff = 0;
            for (int j = 0; j < int(half.size()); j++) {
                if (cur % 3 == 1) curDiff += half[j];
                else if (cur % 3 == 2) curDiff -= half[j];
                cur /= 3;
            }
            diff.push_back(make_pair(curDiff, i));
        }
        sort(diff.begin(), diff.end()); diff.resize(unique(diff.begin(), diff.end()) - diff.begin());
    }
    T solve(int N) { // returns the smallest absolute difference of sums, set is split into setA and setB
        T minDiff = numeric_limits<T>::max(); vector<T> even, odd; even.reserve(N - N / 2); odd.reserve(N / 2);
        vector<pair<T, int>> evenDiff, oddDiff; evenDiff.reserve(1 << (N - N / 2)); oddDiff.reserve(1 << (N / 2));
        int evenPerm, oddPerm; setA.clear(); setB.clear();
        for (int i = 0; i < N; i++) {
            if (i % 2 == 0) even.push_back(A[i]);
            else odd.push_back(A[i]);
        }
        solveHalf(even, evenDiff); solveHalf(odd, oddDiff); int p = 0, q = 0;
        if (int(evenDiff.size()) > 0 && abs(evenDiff[0].first) < minDiff) {
            evenPerm = evenDiff[0].second; oddPerm = 0; minDiff = abs(evenDiff[0].first);
        }
        if (int(oddDiff.size()) > 0 && abs(oddDiff[0].first) < minDiff) {
            evenPerm = 0; oddPerm = oddDiff[0].second; minDiff = abs(oddDiff[0].first);
        }
        while (p < int(evenDiff.size()) && q < int(oddDiff.size())) {
            if (abs(evenDiff[p].first - oddDiff[q].first) < minDiff) {
                evenPerm = evenDiff[p].second; oddPerm = oddDiff[q].second; minDiff = abs(evenDiff[p].first - oddDiff[q].first);
            }
            if (evenDiff[p].first < oddDiff[q].first) p++;
            else if (evenDiff[p].first > oddDiff[q].first) q++;
            else break;
        }
        for (int i = 0; i < (int) even.size(); i++) {
            if (evenPerm % 3 == 1) setA.push_back(i * 2);
            else if (evenPerm % 3 == 2) setB.push_back(i * 2);
            evenPerm /= 3;
        }
        for (int i = 0; i < (int) odd.size(); i++) {
            if (oddPerm % 3 == 1) setB.push_back(i * 2 + 1);
            else if (oddPerm % 3 == 2) setA.push_back(i * 2 + 1);
            oddPerm /= 3;
        }
        return minDiff;
    }
};
```

2.2.3 Subset Sum Count Less

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

// Computes the number of subsets that sum less than or equal to a value
// using a meet in the middle approach
// Time Complexity: Reduces the runtime from O(N * 2^N) to O(N * 2^(N/2))
// Memory Complexity: O(2^(N/2))
template <const int MAXN, class T> struct SubsetSumCountLess {
    T A[MAXN];
    long long solveHalf(vector<T> &half, vector<T> &sum, T value) {
        long long ret = 0;
        for (int i = 1; i < (1 << int(half.size())); i++) {
            T curSum = 0;
            for (int j = 0; j < int(half.size()); j++) if (i & (1 << j)) curSum += half[j];
            sum.push_back(curSum);
            if (curSum <= value) ret++;
        }
        sort(sum.begin(), sum.end());
        return ret;
    }
    long long solve(int N, T value) {
```

```
        long long ret = 0; vector<T> even, odd; even.reserve(N - N / 2); odd.reserve(N / 2);
        vector<T> evenSum, oddSum; evenSum.reserve(1 << (N - N / 2)); oddSum.reserve(1 << (N / 2));
        for (int i = 0; i < N; i++) {
            if (i % 2 == 0) even.push_back(A[i]);
            else odd.push_back(A[i]);
        }
        ret += solveHalf(even, evenSum, value); ret += solveHalf(odd, oddSum, value);
        if (value >= 0) ret++;
        for (int i = 0, j = int(oddSum.size()) - 1; i < int(evenSum.size()); i++) {
            while (j >= 0 && evenSum[i] + oddSum[j] > value) j--;
            ret += j + 1;
        }
        return ret;
    }
};
```

2.3 Sliding Window

2.3.1 Maximum Subarray For Each Subarray of Size K

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

// Computes the maximum element for each contiguous subarray of size K
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxSubarrayK {
    int A[MAXN], dq[MAXN], ans[MAXN];
    void solve(int N, int K) {
        int front = 0, back = 0;
        for (int i = 0; i < N; i++) {
            while (back - front > 0 && dq[front] <= i - K) front++;
            while (back - front > 0 && A[dq[back - 1]] <= A[i]) back--;
            dq[back++] = i;
            if (i >= K - 1) ans[i - K + 1] = A[dq[front]];
        }
    }
};
```

2.3.2 Maximum Subarray Sum of Size K

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

// Computes the maximum sum of a subarray of size K
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxSubarraySumK {
    int maxSum, st, en, A[MAXN];
    int solve(int N, int K) {
        maxSum = 0; st = 0; en = K;
        for (int i = 0; i < K; i++) maxSum += A[i];
        int curSum = maxSum;
        for (int i = K; i < N; i++) if ((curSum += A[i] - A[i - K]) > maxSum) {
            st = i - K + 1; en = i + 1; maxSum = curSum;
        }
        return maxSum;
    }
};
```

2.3.3 Maximum Subarray Sum of Size K or Less

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

// Computes the maximum subarray sum of size K or less
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxSubarraySumK {
    int A[MAXN], dq[MAXN];
    int solve(int N, int K) {
        int front = 0, back = 0, ans = 0;
        for (int i = 1; i < N; i++) A[i] += A[i - 1];
        for (int i = 0; i < N; i++) {
            while (back - front > 0 && dq[front] < i - K) front++;
            while (back - front > 0 && A[dq[back - 1]] >= A[i]) back--;
            dq[back++] = i; ans = max(ans, A[i] - A[dq[front]]);
        }
        return ans;
    }
};
```

2.3.4 Maximum Subarray Sum of Size K or More

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

// Computes the maximum subarray sum of size K or more
// Time Complexity: O(N)
// Memory Complexity: O(N)
template <const int MAXN> struct MaxSubarraySumK {
    int A[MAXN], maxSum[MAXN];
    int solve(int N, int K) {
```

```
    maxSum[0] = A[0];
        int curMax = A[0], sum = 0;
        for (int i = 1; i < N; i++) maxSum[i] = curMax = max(A[i], curMax + A[i]);
        for (int i = 0; i < K; i++) sum += A[i];
        int ans = sum;
        for (int i = K; i < N; i++) { sum += A[i] - A[i - K]; ans = max(ans, max(sum, sum + maxSum[i - K])); }
        return ans;
    }
};
```

2.4 Geometry

2.4.1 Convex Hull

```
#pragma once
#include <bits/stdc++.h>
#include "../../datastructures/geometry/Point.h"
using namespace std;
using namespace std::placeholders;

// Computes the convex hull of points
// Time Complexity: O(N log N)
// Memory Complexity: O(N)
template <const int MAXN> struct ConvexHull {
    Point P[MAXN]; vector<Point> hull;
    void clear() { hull.clear(); }
    void run(int N) {
        sort(P, P + N, Point::yOrderLt);
        if (N > 1) sort(P + 1, P + N, bind(&Point::polarOrderLt, P[0], _1, _2));
        hull.push_back(P[0]);
        int k1, k2;
        for (k1 = 1; k1 < N; k1++) if (P[0] != P[k1]) break;
        if (k1 == N) return;
        for (k2 = k1 + 1; k2 < N; k2++) if (Point::ccw(P[0], P[k1], P[k2]) != 0) break;
        hull.push_back(P[k2 - 1]);
        for (int i = k2; i < N; i++) {
            while (hull.size() >= 2 && Point::ccw(hull[hull.size() - 2], hull[hull.size() - 1], P[i]) <= 0)
                hull.pop_back();
            hull.push_back(P[i]);
        }
    }
};
```

2.4.2 Closest Pair

```
#pragma once
#include <bits/stdc++.h>
#include "../../datastructures/geometry/Point.h"
using namespace std;

// Computes the closest pair of points
// Time Complexity: O(N log N)
// Memory Complexity: O(N)
template <const int MAXN> struct ClosestPair {
    Point P[MAXN], pointsByY[MAXN], aux[MAXN], best1, best2; double bestDist;
    void merge(int lo, int mid, int hi) {
        for (int k = lo; k <= hi; k++) aux[k] = pointsByY[k];
        int i = lo, j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid) pointsByY[k] = aux[j++];
            else if (j > hi) pointsByY[k] = aux[i++];
            else if (aux[j].compareTo(aux[i]) < 0) pointsByY[k] = aux[j++];
            else pointsByY[k] = aux[i++];
        }
    }
    double closest(int lo, int hi) {
        if (hi <= lo) return numeric_limits<double>::infinity();
        int mid = lo + (hi - lo) / 2; Point median = P[mid];
        double delta = min(closest(lo, mid), closest(mid + 1, hi));
        merge(lo, mid, hi); int m = 0;
        for (int i = lo; i <= hi; i++) if (abs(pointsByY[i].x - median.x) < delta) aux[m++] = pointsByY[i];
        for (int i = 0; i < m; i++) {
            for (int j = i + 1; (j < m) && (aux[j].y - aux[i].y < delta); j++) {
                double distance = aux[i].distanceTo(aux[j]);
                if (distance < delta) {
                    delta = distance;
                    if (distance < bestDist) { bestDist = delta; best1 = aux[i]; best2 = aux[j]; }
                }
            }
        }
        return delta;
    }
    void solve(int N) {
        if (N <= 1) return;
        sort(P, P + N, Point::xOrderLt);
        for (int i = 0; i < N - 1; i++) {
            if (P[i] == P[i + 1]) {
                bestDist = 0; best1 = P[i]; best2 = P[i + 1];
                return;
            }
        }
        for (int i = 0; i < N; i++) pointsByY[i] = P[i];
        closest(0, N - 1);
    }
};
```

2.4.3 Farthest Pair

```cpp
#pragma once
#include <bits/stdc++.h>
#include "../../datastructures/geometry/Point.h"
#include "ConvexHull.h"
using namespace std;

// Computes the farthest pair of points
// Time Complexity: O(N log N)
// Memory Complexity: O(N)
template <const int MAXN> struct FarthestPair {
    Point P[MAXN], hull[MAXN], best1, best2; double bestDist;
    ConvexHull<MAXN> H;
    void solve(int N) {
        bestDist = -numeric_limits<double>::infinity();
        if (N <= 1) return;
        copy(P, P + N, H.P);
        H.run(N);
        int M = 1;
        for (auto &&p : H.hull) hull[M++] = p;
        M--;
        if (M == 1) return;
        if (M == 2) {
            best1 = hull[1]; best2 = hull[2];
            bestDist = best1.distanceTo(best2);
            return;
        }
        int k = 2;
        while (Point::area2(hull[M], hull[1], hull[k + 1]) > area2(hull[M], hull[1], hull[k])) k++;
        for (int i = 1, j = k; i <= k && j <= M; i++) {
            if (hull[i].distanceTo(hull[j]) > bestDist) {
                best1 = hull[i]; best2 = hull[j];
                bestDist = hull[i].distanceTo(hull[j]);
            }
            while ((j < M) && Point::area2(hull[i], hull[i + 1], hull[j + 1]) > Point::area2(hull[i], hull[i + 1], hull[j])) {
                j++;
                double distanceSquared = hull[i].distanceTo(hull[j]);
                if (distanceSquared > bestDist) {
                    best1 = hull[i]; best2 = hull[j];
                    bestDist = hull[i].distanceTo(hull[j]);
                }
            }
        }
    }
};
```

2.5 Graph

2.5.1 Search

2.5.1.1 Depth First Search

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

// Depth First Traversal of a graph
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct DFS {
    bool vis[MAXV]; int dep[MAXV], to[MAXV]; vector<int> adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); }
    void addBiEdge(int v, int w) { addEdge(v, w); addEdge(w, v); }
    void reset() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void dfs(int v, int d) {
        vis[v] = true; dep[v] = d;
        for (int w : adj[v]) if (!vis[w]) { to[w] = v; dfs(w, d + 1); }
    }
    void run(int s) { fill(vis, vis + MAXV, false); fill(to, to + MAXV, -1); dfs(s, 0); }
};
```

2.5.1.2 BreadthFirstSearch

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

// Breadth First Traversal of a graph
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct BFS {
    int dist[MAXV], to[MAXV]; vector<int> adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); }
    void addBiEdge(int v, int w) { addEdge(v, w); addEdge(w, v); }
    void reset() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int s) {
        fill(dist, dist + MAXV, INT_MAX); fill(to, to + MAXV, -1);
        queue<int> q; dist[s] = 0; q.push(s);
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int w : adj[v]) if (dist[w] == INT_MAX) { dist[w] = dist[v] + 1; to[w] = v; q.push(w); }
        }
    }
};
```

2.5.2 Shortest Path

2.5.2.1 Classical Dijkstra Single Source Shortest Path

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

// Classical Dijkstra's single source shortest path algorithm for weighted graphs with non negative weights
// Time Complexity: O(V^2)
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct ClassicalDijkstraSSSP {
    unit INF, dist[MAXV]; pair<int, unit> to[MAXV]; bool done[MAXV]; vector<pair<int, unit>> adj[MAXV];
    ClassicalDijkstraSSSP(unit INF) : INF(INF) {}
    void addEdge(int v, int w, unit weight) { adj[v].emplace_back(w, weight); }
    void addBiEdge(int v, int w, unit weight) { addEdge(v, w, weight); addEdge(w, v, weight); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V, int s) {
        fill(dist, dist + V, INF); fill(to, to + V, make_pair(-1, 0)); fill(done, done + V, false); dist[s] = 0;
        for (int v = 0; v < V - 1; v++) {
            int minV = -1;
            for (int w = 0; w < V; v++) if (!done[w] && (minV == -1 || dist[minV] > dist[w])) minV = w;
            done[minV] = true;
            for (auto &&e : adj[minV]) if (dist[e.first] > dist[minV] + e.second) {
                dist[e.first] = dist[minV] + e.second; to[e.first] = {v, e.second};
            }
        }
    }
};
```

2.5.2.2 Dijkstra Single Source Shortest Path

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

// Dijstra's single source shortest path algorithm for weighted graphs without negative cycles
// Time Complexity: O(E log E) or O(E log V) if an indexed priority queue is used
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct DijkstraSSSP {
    unit INF, dist[MAXV]; pair<int, unit> to[MAXV]; vector<pair<int, unit>> adj[MAXV]; DijkstraSSSP(unit INF) : INF(INF) {}
    void addEdge(int v, int w, unit weight) { adj[v].emplace_back(w, weight); }
    void addBiEdge(int v, int w, unit weight) { addEdge(v, w, weight); addEdge(w, v, weight); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V, int s) {
        priority_queue<pair<unit, int>, vector<pair<unit, int>>, greater<pair<unit, int>>> PQ;
        fill(dist, dist + V, INF); fill(to, to + V, make_pair(-1, 0)); PQ.emplace(dist[s] = 0, s);
        while (!PQ.empty()) {
            unit d = PQ.top().first; int v = PQ.top().second; PQ.pop();
            if (d > dist[v]) continue;
            for (auto &&e : adj[v]) if (dist[e.first] > dist[v] + e.second) {
                to[e.first] = {v, e.second}; PQ.emplace(dist[e.first] = dist[v] + e.second, e.first);
            }
        }
    }
};
```

2.5.2.3 Source Path Faster Algorithm

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

// Shortest Path Faster Algorithm for weighted graphs without negative cycles
// Time Complexity: O(VE) in the worst case, O(E) on average
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct SPFA {
    unit INF, dist[MAXV]; SPFA(unit INF) : INF(INF) {}
    bool inQueue[MAXV]; pair<int, unit> to[MAXV]; vector<pair<int, unit>> adj[MAXV];
    void addEdge(int v, int w, unit weight) { adj[v].emplace_back(w, weight); }
    void addBiEdge(int v, int w, unit weight) { addEdge(v, w, weight); addEdge(w, v, weight); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V, int s) {
        fill(dist, dist + V, INF); fill(to, to + V, make_pair(-1, 0)); fill(inQueue, inQueue + V, true);
        deque<int> DQ; DQ.push_back(s); inQueue[s] = true;
        while (!DQ.empty()) {
            int v = DQ.front(); DQ.pop_front(); inQueue[v] = false;
            for (auto &&e : adj[v]) if (dist[e.first] > dist[v] + e.second) {
                to[e.first] = {v, e.second};
                if (!inQueue[e.first]) {
                    if (!DQ.empty() && dist[e.first] <= dist[DQ.front()]) DQ.push_front(e.first);
                    else DQ.push_back(e.first);
                    inQueue[e.first] = true;
                }
            }
        }
    }
};
```

2.5.2.4 Bellman Ford Single Source Shortest Path and Negative Cycle Detection

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

// Bellman Ford's single source shortest path algorithm for weighted graphs with negative cycles
// Can be used to detect negative cycles
```

```cpp
// Time Compleity: O(VE)
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct BellmanFordSSSP {
    struct Edge { int v, w; unit weight; };
    unit INF, dist[MAXV]; pair<int, unit> to[MAXV]; vector<Edge> edges; bool hasNegativeCycle; BellmanFordSSSP(unit INF) : INF(INF)
        {}
    void addEdge(int v, int w, unit weight) { edges.push_back({v, w, weight}); }
    void addBiEdge(int v, int w, unit weight) { addEdge(v, w, weight); addEdge(w, v, weight); }
    void clear() { edges.clear(); }
    void run(int V, int s) {
        fill(dist, dist + V, INF); fill(to, to + V, make_pair(-1, 0)); hasNegativeCycle = false; dist[s] = 0;
        for (int i = 0; i < V - 1; i++) for (auto &&e : edges) if (dist[e.v] < INF && dist[e.w] > dist[e.v] + e.weight) {
            dist[e.w] = dist[e.v] + e.weight; to[e.w] = {e.v, e.weight};
        }
        bool inCycle = true;
        while (inCycle) {
            inCycle = false;
            for (auto &&e : edges) if (dist[e.v] < INF && dist[e.w] > -INF && dist[e.w] > dist[e.v] + e.weight) {
                dist[e.w] = -INF; inCycle = true; hasNegativeCycle = true;
            }
        }
    }
};
```

2.5.2.5 Floyd Warshall All Pairs Shortest Path

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

// Floyd Warshall's all pairs shortest path algorithm for weighted graphs
// Can be used to detect negative cycles
// Time Complexity: O(V^3)
// Memory Complexity: O(V^2)
template <const int MAXV, class unit> struct FloydWarshallAPSP {
    unit INF, dist[MAXV][MAXV]; bool hasNegativeCycle; FloydWarshallAPSP(unit INF) : INF(INF) {}
    void init() { for (int i = 0; i < MAXV; i++) fill(dist[i], dist[i] + MAXV, INF); }
    void addEdge(int v, int w, unit weight) { dist[v][w] = min(dist[v][w], weight); }
    void addBiEdge(int v, int w, unit weight) { addEdge(v, w, weight); addEdge(w, v, weight); }
    void run(int V) {
        hasNegativeCycle = false;
        for (int v = 0; v < V; v++) dist[v][v] = 0;
        for (int u = 0; u < V; u++) for (int v = 0; v < V; v++) for (int w = 0; w < V; w++)
            if (dist[v][u] < INF && dist[u][w] < INF && dist[v][w] > dist[v][u] + dist[u][w]) dist[v][w] = dist[v][u] + dist[u][w];
        for (int u = 0; u < V; u++) for (int v = 0; v < V; v++) for (int w = 0; w < V; w++)
            if (dist[w][w] < 0 && dist[u][w] < INF && dist[w][v] < INF) { dist[u][v] = -INF; hasNegativeCycle = true; break; }
    }
};
```

2.5.2.6 Johnson All Pairs Shortest Path

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

// Johnson's all pairs shortest path algorithm for weighted graphs
// Can be used to detect negative cycles
// Time Complexity: O(VE log E) or O(VE log V) if an indexed priority queue is used
// Memory Complexity: O(V^2 + E)
template <const int MAXV, class unit> struct JohnsonAPSP {
    unit INF, dist[MAXV][MAXV], h[MAXV]; JohnsonAPSP(unit INF) : INF(INF) {}
    pair<int, unit> to[MAXV][MAXV]; bool hasNegativeCycle; vector<pair<int, unit>> adj[MAXV];
    void addEdge(int v, int w, unit weight) { adj[v].emplace_back(w, weight); }
    void addBiEdge(int v, int w, unit weight) { addEdge(v, w, weight); addEdge(w, v, weight); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V) {
        for (int v = 0; v < V; v++) { fill(dist[v], dist[v] + V, INF); fill(to[v], to[v] + V, make_pair(-1, 0)); }
        fill(h, h + V, INF); h[V] = 0;
        for (int v = 0; v < V; v++) adj[V].emplace_back(v, 0);
        for (int i = 0; i < V; i++) for (int v = 0; v <= V; v++) for (auto &&e : adj[v])
            if (h[v] < INF && h[e.first] > h[v] + e.second) h[e.first] = h[v] + e.second;
        adj[V].clear();
        bool inCycle = true;
        while (inCycle) {
            inCycle = false;
            for (int v = 0; v <= V; v++) for (auto &&e : adj[v]) {
                if (h[v] < INF && h[e.first] > -INF && h[e.first] > h[v] + e.second) {
                    h[e.first] = -INF; inCycle = true; hasNegativeCycle = true;
                }
            }
        }
        adj[V].clear();
        if (hasNegativeCycle) return;
        for (int s = 0; s < V; s++) {
            priority_queue<pair<unit, int>, vector<pair<unit, int>> , greater<pair<unit, int>>> PQ;
            PQ.emplace(dist[s][s] = 0, s);
            while (!PQ.empty()) {
                unit d = PQ.top().first; int v = PQ.top().second; PQ.pop();
                if (d > dist[s][v]) continue;
                for (auto &&e : adj[v]) if (dist[s][e.first] > dist[s][v] + e.second + h[v] - h[e.first]) {
                    to[s][e.first] = {v, e.second}; PQ.emplace(dist[s][e.first] = dist[s][v] + e.second + h[v] - h[e.first], e.first
                        );
                }
            }
        }
        for (int v = 0; v < V; v++) for (int w = 0; w < V; w++) dist[v][w] = dist[v][w] - h[v] + h[w];
```

```
    }
};
```

2.5.3 Components

2.5.3.1 Biconnected Components

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

// Computes the articulation points and bridges of a graph,
// and decomposes the graph into biconnected ocmponents
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct Biconnected {
    int low[MAXV], pre[MAXV], cur; bool articulation[MAXV]; stack<pair<int, int>> s;
    vector<int> adj[MAXV]; vector<pair<int, int>> bridges; vector<unordered_set<int>> components;
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void dfs(int v, int prev) {
        int children = 0; pre[v] = low[v] = cur++;
        for (int w : adj[v]) {
            if (pre[w] == -1) {
                children++; s.emplace(v, w); dfs(w, v); low[v] = min(low[v], low[w]);
                if ((prev == v && children > 1) || (prev != v && low[w] >= pre[v])) {
                    articulation[v] = true; components.emplace_back();
                    while (s.top().first != v || s.top().second != w) {
                        components.back().insert(s.top().first); components.back().insert(s.top().second); s.pop();
                    }
                    components.back().insert(s.top().first); components.back().insert(s.top().second); s.pop();
                }
                if (low[w] == pre[w]) bridges.emplace_back(v, w);
            } else if (w != prev && pre[w] < low[v]) {
                low[v] = pre[w]; s.emplace(v, w);
            }
        }
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); bridges.clear(); components.clear(); }
    void run(int V) {
        cur = 0;
        for (int v = 0; v < V; v++) { low[v] = pre[v] = -1; articulation[v] = false; }
        for (int v = 0; v < V; v++) {
            if (pre[v] == -1) dfs(v, v);
            if (!s.empty()) {
                components.emplace_back();
                while (!s.empty()) {
                    components.back().insert(s.top().first); components.back().insert(s.top().second); s.pop();
                }
            }
        }
    }
};
```

2.5.3.2 Connected Components

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

// Computes the connected components of a graph
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct ConnectedComponents {
    int id[MAXV]; bool vis[MAXV];
    vector<int> adj[MAXV]; vector<vector<int>> components;
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void dfs(int v) {
        vis[v] = true; id[v] = components.size() - 1;
        components.back().push_back(v);
        for (int w : adj[v]) if (!vis[w]) dfs(w);
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); components.clear(); }
    void run(int V) {
        fill(vis, vis + MAXV, false);
        for (int v = 0; v < V; v++) if (!vis[v]) { components.emplace_back(); dfs(v); }
    }
};
```

2.5.3.3 Kosaraju Sharir Strongly Connected Components

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

// Determines the strong components in a directed graph using Kosaraju and Sharir's algorithm
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct KosarajuSharirSCC {
    int id[MAXV]; bool vis[MAXV]; vector<int> adj[MAXV], rev[MAXV]; vector<vector<int>> components; stack<int> revPost;
    void addEdge(int v, int w) { adj[v].push_back(w); }
    void postOrder(int v) {
        vis[v] = true;
        for (int w : rev[v]) if (!vis[v]) postOrder(w);
        revPost.push(v);
    }
    void dfs(int v) {
```

```
        vis[v] = true; id[v] = components.size() - 1; components.back().push_back(v);
        for (int w : adj[v]) if (!vis[v]) dfs(w);
    }
    void clear() { for (int i = 0; i < MAXV; i++) { adj[i].clear(); rev[i].clear(); } components.clear(); }
    void run(int V) {
        fill(vis, vis + V, false);
        for (int v = 0; v < V; v++) for (int w : adj[v]) rev[w].push_back(v);
        for (int v = 0; v < V; v++) if (!vis[v]) postOrder(v);
        fill(vis, vis + V, false);
        while (!revPost.empty()) {
            int v = revPost.top(); revPost.pop();
            if (!vis[v]) { components.emplace_back(); dfs(v); }
        }
    }
};
```

2.5.3.4 Tarjan Strongly Connected Components

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

// Determines the strong components in a directed graph using Tarjan's algorithm
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct TarjanSCC {
    int id[MAXV], low[MAXV], pre; bool vis[MAXV]; vector<int> adj[MAXV]; vector<vector<int>> components; stack<int> s;
    void addEdge(int v, int w) { adj[v].push_back(w); }
    void dfs(int v) {
        vis[v] = true; int mn = low[v] = pre++; s.push(v);
        for (int w : adj[v]) {
            if (!vis[w]) dfs(w);
            if (low[w] < mn) mn = low[w];
        }
        if (mn < low[v]) { low[v] = mn; return; }
        int w; components.emplace_back();
        do {
            w = s.top(); s.pop();
            id[w] = components.size() - 1; components.back().push_back(w); low[w] = INT_MAX;
        } while (w != v);
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); components.clear(); }
    void run(int V) {
        fill(vis, vis + V, false); pre = 0;
        for (int v = 0; v < V; v++) if (!vis[v]) dfs(v);
    }
};
```

2.5.3.5 Bron Kerbosch Max Clique

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

// Computes the maximum (weighted) clique in an undirected graph, where each vertex
// can have a weight (default is 1)
// A clique is defined to be a subset of a graph such that every pair of vertices
// in the subset are connected by an edge
// Time Complexity: O(3^(V/3))
// Memory Complexity: O(V^2)
template <const int MAXV> struct BronKerboschMaxClique {
    bool adj[MAXV][MAXV]; int W[MAXV];
    void init() { for (int i = 0; i < MAXV; W[i++] = 1) for (int j = 0; j < MAXV; j++) adj[i][j] = 0; }
    int solve(int V, long long cur, long long pool, long long excl) {
        if (pool == 0 && excl == 0) {
            int cnt = 0;
            for (int i = 0; i < V; i++) if ((cur & (1LL << i)) > 0) cnt += W[i];
            return cnt;
        }
        int res = 0, j = 0;
        for (int i = 0; i < V; i++) if ((pool & (1LL << i)) > 0 || (excl & (1LL << i)) > 0) j = i;
        for (int i = 0; i < V; i++) {
            if ((pool & (1LL << i)) == 0 || adj[i][j]) continue;
            int ncurr = cur, npool = 0, nexcl = 0; ncurr |= 1LL << i;
            for (int k = 0; k < V; k++) if (adj[i][k]) { npool |= pool & (1LL << k); nexcl |= excl & (1LL << k); }
            res = max(res, solve(V, ncurr, npool, nexcl)); pool &= ~(1LL << i); excl |= 1 >> i;
        }
        return res;
    }
    int solve(int V) { return solve(V, 0, (1LL << V) - 1, 0); }
};
```

2.5.3.6 Stoer Wagner Min Cut

```
#pragma once
#include <bits/stdc++.h>
#include "../../datastructures/IndexedPQ.h"
#include "../../datastructures/UnionFind.h"
using namespace std;

// Computes the minimum cut for a weighted graph
// A cut is a partition of the vertices into two nonempty subsets
// A crossing edges is an edge with endpoints in both subsets
// The cost of a cut is the sum of the weights of the crossing edges
// Time Complexity: O(V (V + E) log V)
// Memory Complexity: O(V + E)
template <const int MAXN, class unit> struct StoerWagnerMinCut {
```

```
    struct Graph {
        vector<pair<int, unit>> adj[MAXN];
        void addEdge(int v, int w, unit weight) { adj[v].emplace_back(w, weight); adj[w].emplace_back(v, weight); }
        void clear() { for (int i = 0; i < MAXN; i++) adj[i].clear(); }
    } G;
    bool vis[MAXN], cut[MAXN]; UnionFind<MAXN> uf; unit INF; StoerWagnerMinCut(unit INF) : INF(INF) {}
    void addEdge(int v, int w, unit weight) { G.addEdge(v, w, weight); }
    struct CutPhase { unit weight; int s, t; };
    void makeCut(int V, int t, UnionFind<MAXN> &uf) { for (int v = 0; v < V; v++) cut[v] = uf.connected(v, t); }
    void minCutPhase(int V, CutPhase &cp) {
        IndexedPQ<unit, less<unit>> pq(V);
        for (int v = 0; v < V; v++) if (v != cp.s && !vis[v]) pq.push(v, 0);
        pq.push(cp.s, INF);
        while (!pq.empty()) {
            int v = pq.top().first; pq.pop(); cp.s = cp.t; cp.t = v;
            for (auto &&e : G.adj[v]) if (pq.containsIndex(e.first)) pq.changeKey(e.first, pq.keyOf(e.first) + e.second);
        }
        cp.weight = 0;
        for (auto &&e : G.adj[cp.t]) cp.weight += e.second;
    }
    Graph contractEdge(int V, int s, int t) {
        Graph H;
        for (int v = 0; v < V; v++) {
            for (auto &&e : G.adj[v]) {
                if ((v == s && e.first == t) || (v == t && e.first == s)) continue;
                if (v < e.first) {
                    if (e.first == t) H.addEdge(v, s, e.second);
                    else if (v == t) H.addEdge(e.first, s, e.second);
                    else H.addEdge(v, e.first, e.second);
                }
            }
        }
        return H;
    }
    unit minCut(int V, int root = 0) {
        unit ret = INF; CutPhase cp = {0, root, root}; uf.init(V); fill(vis, vis + V, false); fill(cut, cut + V, false);
        for (int v = V; v > 1; v--) {
            minCutPhase(V, cp);
            if (cp.weight < ret) { ret = cp.weight; makeCut(V, cp.t, uf); }
            G = contractEdge(V, cp.s, cp.t); vis[cp.t] = true; uf.join(cp.s, cp.t);
        }
        return ret;
    }
    void clear() { G.clear(); }
};
```

2.5.3.7 Centroid Decomposition

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

// Finds the centroid of each component of a tree, and splits the component at that vertex
// Can be used to create a centroid tree, which has depth O(log V)
// Time Complexity:
//    getCentroid: O(ComponentSize)
//    bfs: O(V log V)
// Memory Complexity: O(V)
template <const int MAXV> struct CentroidDecomposition {
    vector<int> adj[MAXV]; bool exclude[MAXV]; int par[MAXV], T[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    int getSize(int v, int prev) {
        int size = 1;
        for (int w : adj[v]) if (w != prev && !exclude[w]) size += getSize(w, v);
        return size;
    }
    int getCentroid(int v, int prev, int treeSize) {
        int n = treeSize, size = 1; bool hasCentroid = true;
        for (int w : adj[v]) {
            if (w == prev || exclude[w]) continue;
            int ret = getCentroid(w, v, treeSize);
            if (ret > 0) return ret;
            hasCentroid &= -ret <= n / 2; size += -ret;
        }
        return (hasCentroid &= n - size <= n / 2) ? v : -size;
    }
    void init() { fill(exclude, exclude + MAXV, false); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    int getCentroid(int v) {
        int c = getCentroid(v, -1, getSize(v, -1)); exclude[c] = true;
        return c;
    }
    void bfs(int root = 0) {
        queue<pair<int, int>> q; q.emplace(root, -1);
        while (!q.empty()) {
            int v = q.front().first, c = getCentroid(v, -1, getSize(v, -1));
            par[c] = q.front().second; q.pop(); exclude[c] = true;
            for (int w : adj[c]) if (!exclude[w]) q.emplace(w, c);
        }
    }
};
```

2.5.3.8 Dynamic Connectivity Divide and Conquer

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

```cpp
// Support queries for the number of components in a graph, after edges have been added or removed
// Divide and Conquer Solution
// Time Complexity: O(V + Q log Q)
// Memory Complexity: O(V + Q)
template <const int MAXV, const int MAXQ> struct DynamicConnectivityDivAndConq {
    int Q = 0, cnt, UF[MAXV]; vector<int> ans; unordered_map<int, int> present[MAXV]; stack<pair<pair<int ,int>, int>> history;
    struct Query { int type, v, w, otherTime; } q[MAXQ];
    int find(int v) { while (UF[v] >= 0) v = UF[v]; return v; }
    bool join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return false;
        if (UF[v] > UF[w]) swap(v, w);
        history.push({{v, w}, UF[w]}); UF[v] += UF[w]; UF[w] = v; cnt--;
        return true;
    }
    void undo() {
        int v = history.top().first.first, w = history.top().first.second, ufw = history.top().second;
        history.pop(); UF[w] = ufw; UF[v] -= UF[w]; cnt++;
    }
    void solve(int l, int r) {
        if (l == r && q[l].type == 0) ans.push_back(cnt);
        if (l >= r) return;
        int m = l + (r - l) / 2, curSize = history.size();
        for (int i = m + 1; i <= r; i++) if (q[i].otherTime < l) join(q[i].v, q[i].w);
        solve(l, m);
        while ((int) history.size() > curSize) undo();
        for (int i = l; i <= m; i++) if (q[i].otherTime > r) join(q[i].v, q[i].w);
        solve(m + 1, r);
        while ((int) history.size() > curSize) undo();
    }
    void clear() { for (int i = 0; i < MAXV; i++) present[i].clear(); ans.clear(); Q = 0; }
    void addEdge(int v, int w) {
        if (v > w) swap(v, w);
        present[v][w] = Q; q[Q++] = {1, v, w, INT_MAX};
    }
    void removeEdge(int v, int w) {
        if (v > w) swap(v, w);
        int insTime = present[v][w]; q[Q] = {-1, v, w, insTime}; q[insTime].otherTime = Q++; present[v].erase(w);
    }
    void query() { q[Q] = {0, -1, -1, Q}; Q++; }
    void solve(int V) {
        cnt = V; fill(UF, UF + MAXV, -1);
        solve(0, Q - 1);
    }
};
```

2.5.3.9 Dynamic Connectivity Sqrt Decomposition

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

// Support queries for the number of components in a graph, after edges have been added or removed
// Sqrt Decomposition Solution
// Time Complexity: O(V * (Q / B) + Q * B)
// Memory Complexity: O(V + Q)
template <const int MAXV, const int MAXQ, const int BLOCKSZ> struct DynamicConnectivitySqrtDecomp {
    int Q = 0, vis[MAXV], root[MAXV]; vector<int> ans; unordered_set<int> adj[MAXV], toRem[MAXV]; unordered_map<int, int> adj2[MAXV
        ];
    struct Query { int type, v, w; } q[MAXQ];
    void dfs1(int v, int r) {
        root[v] = r;
        for (int w : adj[v]) if (root[w] == -1 && !toRem[v].count(w)) dfs1(w, r);
    }
    bool dfs2(int v, int t, int i) {
        if (v == t) return true;
        vis[v] = i;
        for (auto && e : adj2[v]) if (vis[e.first] != i && dfs2(e.first, t, i)) return true;
        return false;
    }
    void clear() { for (int i = 0; i < MAXV; i++) { adj[i].clear(); toRem[i].clear(); adj2[i].clear(); } ans.clear(); Q = 0; }
    void addEdge(int v, int w) { q[Q++] = {1, v, w}; }
    void removeEdge(int v, int w) { q[Q++] = {-1, v, w}; }
    void query() { q[Q++] = {0, -1, -1}; }
    void solve(int V) {
        int cnt = V;
        fill(vis, vis + MAXV, -1);
        for (int st = 0; st < Q; st += BLOCKSZ) {
            fill(root, root + V, -1);
            for (int i = st; i < min(st + BLOCKSZ, Q); i++) {
                if (q[i].type != -1) continue;
                toRem[q[i].v].insert(q[i].w); toRem[q[i].w].insert(q[i].v);
            }
            for (int v = 0; v < V; v++) {
                if (root[v] == -1) dfs1(v, v);
                adj2[v].clear();
            }
            for (int v = 0; v < V; v++) {
                for (int w : adj[v]) if (root[v] != root[w]) adj2[root[v]][root[w]]++;
                toRem[v].clear();
            }
            for (int i = st; i < min(st + BLOCKSZ, Q); i++) {
                if (q[i].type == 1) {
                    adj[q[i].v].insert(q[i].w); adj[q[i].w].insert(q[i].v);
                    int rv = root[q[i].v], rw = root[q[i].w];
```

```
                    if (rv != rw) {
                        if (!dfs2(rv, rw, i)) cnt--;
                        adj2[rv][rw]++; adj2[rw][rv]++;
                    }
                } else if (q[i].type == -1) {
                    adj[q[i].v].erase(q[i].w); adj[q[i].w].erase(q[i].v);
                    int rv = root[q[i].v], rw = root[q[i].w];
                    if (rv != rw) {
                        if (--adj2[rv][rw] == 0) adj2[rv].erase(rw);
                        if (--adj2[rw][rv] == 0) adj2[rw].erase(rv);
                        if (!dfs2(rv, rw, i)) cnt++;
                    }
                } else ans.push_back(cnt);
            }
        }
    }
};
```

2.5.3.10 Dynamic Biconnectivity

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

// Supports queries for the number of bridges in a graph after edges have been added
// Time Complexity: O(V + Q)
// Memory Complexity O(V + Q)
template <const int MAXV, const int MAXQ> struct DynamicBiconnectivity {
    int Q = 0, UF[MAXV], par[MAXV], dep[MAXV]; vector<int> adj[MAXV], ans;
    struct Query { int type, v, w; } q[MAXQ];
    int find(int v) { return UF[v] < 0 ? v : UF[v] = find(UF[v]); }
    bool join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return false;
        if (UF[v] > UF[w]) swap(v, w);
        UF[v] += UF[w]; UF[w] = v;
        return true;
    }
    void dfs(int v, int prev, int d) {
        dep[v] = d; par[v] = prev;
        for (int w : adj[v]) if (w != prev) dfs(w, v, d + 1);
    }
    void init() { fill(UF, UF + MAXV, -1); Q = 0; }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); ans.clear(); }
    void addEdge(int v, int w) {
        if (join(v, w)) { adj[v].push_back(w); adj[w].push_back(v); q[Q++] = {1, v, w}; }
        else q[Q++] = {2, v, w};
    }
    void query() { q[Q++] = {0, -1, -1}; }
    void solve(int V) {
        fill(par, par + MAXV, -1); fill(UF, UF + MAXV, -1); int cnt = 0;
        for (int v = 0; v < V; v++) if (par[v] == -1) dfs(v, -1, 0);
        for (int i = 0; i < Q; i++) {
            if (q[i].type == 0) ans.push_back(cnt);
            else if (q[i].type == 1) cnt++;
            else {
                int a = q[i].v, b = q[i].w;
                while ((a = find(a)) != (b = find(b))) {
                    if (dep[a] < dep[b]) swap(a, b);
                    UF[a] = find(par[a]); a = par[a]; cnt--;
                }
            }
        }
    }
};
```

2.5.4 Minimum Spanning Tree

2.5.4.1 Prim Minimum Spanning Tree

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

// Computes the minimum spanning tree using Prims's algorithm.
// Time Complexity: O(E log E) or O(E log V) if an indexed priority queue is used
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct PrimMST {
    unit INF; PrimMST(unit INF) : INF(INF) {}
    struct Edge { int v, w; unit weight; };
    pair<int, unit> to[MAXV]; unit weight, cost[MAXV]; vector<Edge> mst; bool vis[MAXV];
    vector<pair<int, unit>> adj[MAXV];
    void addEdge(int v, int w, unit weight) { adj[v].emplace_back(w, weight); adj[w].emplace_back(v, weight); }
    unit run(int V) {
        weight = 0; fill(vis, vis + V, false); fill(cost, cost + V, INF); fill(to, to + V, make_pair(-1, 0));
        priority_queue<pair<unit, int>, vector<pair<unit, int>>, greater<pair<unit, int>>> PQ;
        for (int s = 0; s < V; s++) {
            if (vis[s]) continue;
            PQ.emplace(cost[s] = 0, s);
            while (!PQ.empty()) {
                int v = PQ.top().second; PQ.pop(); vis[v] = true;
                for (auto &&e : adj[v]) if (!vis[e.first] && e.second < cost[e.first]) {
                    to[e.first] = {v, e.second}; PQ.emplace(cost[e.first] = e.second, e.first);
                }
            }
        }
```

```
        for (int v = 0; v < V; v++) if (to[v].first != -1) {
            mst.push_back({v, to[v].first, to[v].second}); weight += to[v].second;
        }
        return weight;
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); mst.clear(); }
};
```

2.5.4.2 Kruskal Minimum Spanning Tree

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

// Computes the minimum spanning tree using Kruskal's algorithm.
// Time Complexity: O(E log E)
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct KruskalMST {
    struct Edge {
        int v, w; unit weight;
        bool operator < (const Edge &e) const { return weight < e.weight; }
    };
    int UF[MAXV]; vector<Edge> edges, mst; unit weight;
    void addEdge(int v, int w, unit weight) { edges.push_back({v, w, weight}); }
    int find(int v) { return UF[v] < 0 ? v : UF[v] = find(UF[v]); }
    bool join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return false;
        if (UF[v] > UF[w]) swap(v, w);
        UF[v] += UF[w]; UF[w] = v;
        return true;
    }
    unit run(int V) {
        weight = 0; fill(UF, UF + V, -1); sort(edges.begin(), edges.end());
        for (auto &&e : edges) {
            if (int(mst.size()) >= V - 1) break;
            if (join(e.v, e.w)) { mst.push_back(e); weight += e.weight; }
        }
        return weight;
    }
    void clear() { edges.clear(); mst.clear(); }
};
```

2.5.4.3 Boruvka Minimum Spanning Tree

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

// Computes the minimum spanning tree using Boruvka's algorithm.
// Time Complexity: O(E log V)
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct BoruvkaMST {
    struct Edge { int v, w; unit weight; };
    int UF[MAXV], closest[MAXV]; vector<Edge> edges, mst; unit weight;
    void addEdge(int v, int w, unit weight) { edges.push_back({v, w, weight}); }
    int find(int v) { return UF[v] < 0 ? v : UF[v] = find(UF[v]); }
    bool join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return false;
        if (UF[v] > UF[w]) swap(v, w);
        UF[v] += UF[w]; UF[w] = v;
        return true;
    }
    unit run(int V) {
        weight = 0; fill(UF, UF + V, -1);
        for (int t = 1; t < V && int(mst.size()) < V - 1; t *= 2) {
            fill(closest, closest + V, -1);
            for (int e = 0; e < int(edges.size()); e++) {
                int i = find(edges[e].v), j = find(edges[e].w);
                if (i == j) continue;
                if (closest[i] == -1 || edges[e].weight < edges[closest[i]].weight) closest[i] = e;
                if (closest[j] == -1 || edges[e].weight < edges[closest[j]].weight) closest[j] = e;
            }
            for (int i = 0; i < V; i++) {
                if (closest[i] == -1) continue;
                int v = edges[closest[i]].v, w = edges[closest[i]].w;
                if (join(v, w)) { mst.push_back(edges[closest[i]]); weight += edges[closest[i]].weight; }
            }
        }
        return weight;
    }
    void clear() { edges.clear(); mst.clear(); }
};
```

2.5.4.4 Offline Dynamic MST

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

// Supports queries for the minimum spanning tree after an edges's weight has been changed
// Offline Sqrt Decomposition Solution
// Time Complexity: O(V + E * (Q / B) + Q * B)
// Memory Complexity: O(V + E + Q)
template <const int MAXV, const int MAXE, const int MAXQ, const int BLOCKSZ, class unit> struct OfflineDynamicMST {
    struct Edge { int v, w; unit weight; };
```

```
    int Q = 0, flag[MAXE], ind[MAXQ], curFlag, UF[2][MAXV], stamp[2], vis[2][MAXV]; unit ans[MAXQ]; pair<int, unit> q[MAXQ];
    set<pair<unit, int>> small, large; vector<Edge> edges;
    void prop(int i, int v) { if (vis[i][v] != stamp[i]) { vis[i][v] = stamp[i]; UF[i][v] = -1; } }
    int find(int i, int v) { prop(i, v); return UF[i][v] < 0 ? v : UF[i][v] = find(i, UF[i][v]); }
    bool join(int i, int v, int w) {
        v = find(i, v); w = find(i, w);
        if (v == w) return false;
        if (UF[i][v] > UF[i][w]) swap(v, w);
        UF[i][v] += UF[i][w]; UF[i][w] = v;
        return true;
    }
    void addEdge(int v, int w, unit weight) { edges.push_back({v, w, weight}); }
    void query(int i, unit weight) { q[Q++] = {i, weight}; }
    void solve(int V) {
        int E = edges.size(); fill(flag, flag + E, -1); curFlag = 0; unit forest, mst;
        for (int i = 0; i < E; i++) large.emplace(edges[i].weight, i);
        for (int i = 0; i < 2; i++) fill(UF[i], UF[i] + V, -1);
        for (int l = 0; l < Q; l += BLOCKSZ) {
            int r = min(l + BLOCKSZ - 1, Q - 1), cnt = 0; curFlag++; stamp[0]++; small.clear();
            for (int i = l; i <= r; i++) {
                flag[q[i].first] = curFlag; join(0, edges[q[i].first].v, edges[q[i].first].w);
                small.emplace(edges[q[i].first].weight, q[i].first);
            }
            stamp[1]++;
            for (auto &&p : large) if (flag[p.second] != curFlag)
                if (join(1, edges[p.second].v, edges[p.second].w)) ind[cnt++] = p.second;
            stamp[1]++; forest = 0;
            for (int i = 0; i < cnt; i++) {
                if (join(0, edges[ind[i]].v, edges[ind[i]].w)) {
                    join(1, edges[ind[i]].v, edges[ind[i]].w);
                    forest += edges[ind[i]].weight;
                } else small.emplace(edges[ind[i]].weight, ind[i]);
            }
            for (int i = l; i <= r; i++) {
                large.erase({edges[q[i].first].weight, q[i].first}); small.erase({edges[q[i].first].weight, q[i].first});
                edges[q[i].first].weight = q[i].second;
                large.emplace(edges[q[i].first].weight, q[i].first); small.emplace(edges[q[i].first].weight, q[i].first);
                stamp[0]++; mst = 0;
                for (auto &&p: small) if (join(0, find(1, edges[p.second].v), find(1, edges[p.second].w)))
                    mst += edges[p.second].weight;
                ans[i] = forest + mst;
            }
        }
    }
    void clear() { small.clear(); large.clear(); edges.clear(); Q = 0; }
};
```

2.5.5 Lowest Common Ancestor

2.5.5.1 LCA using Sparse Table

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

// Supports queries for the lowest common ancestor of 2 vertices in a tree using a sparse table
// Time Complexity:
//    run: O(V log V)
//    lca: O(log V)
template <const int MAXV, const int MAXLGV> struct LCA_SparseTable {
    int dep[MAXV], par[MAXLGV][MAXV]; vector<int> adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void dfs(int v, int prev, int d) {
        dep[v] = d; par[0][v] = prev;
        for (int w : adj[v]) if (w != prev) dfs(w, v, d + 1);
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V, int root = 0) {
        for (int i = 0; i < MAXLGV; i++) fill(par[i], par[i] + V, -1);
        dfs(root, -1, 0);
        for (int i = 1; i < MAXLGV; i++) for (int j = 0; j < V; j++) if (par[i - 1][j] != -1) par[i][j] = par[i - 1][par[i - 1][j]];
    }
    int lca(int v, int w) {
        if (dep[v] < dep[w]) swap(v, w);
        for (int i = MAXLGV - 1; i >= 0; i--) if (par[i][v] != -1 && dep[par[i][v]] >= dep[w]) v = par[i][v];
        if (v == w) return v;
        for (int i = MAXLGV - 1; i >= 0; i--) if (par[i][v] != par[i][w]) { v = par[i][v]; w = par[i][w]; }
        return par[0][v];
    }
};
```

2.5.5.2 LCA using Euler Tour

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

// Supports queries for the lowest common ancestor of 2 vertices in a tree by computing the euler tour of the tree
// Time Complexity:
//    run: O(V)
//    lca: O(log V)
template <const int MAXV> struct LCA_Euler {
    int to[MAXV], ind[MAXV], vert[MAXV], head[MAXV], ord[MAXV * 4], size, cnt, curInd; vector<int> adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void dfs(int v, int prev) {
        int cur = ind[v] = curInd++; vert[cur] = v;
```

```
        for (int w : adj[v]) {
            if (w == prev) continue;
            ord[size + cnt++] = cur;
            if (head[cur] == -1) head[cur] = cnt - 1;
            dfs(w, v);
        }
        ord[size + cnt++] = cur;
        if (head[cur] == -1) head[cur] = cnt - 1;
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V, int root = 0) {
        size = 2 * V - 1; curInd = cnt = 0; fill(head, head + V, -1); dfs(root, -1);
        for (int i = 2 * size - 2; i > 1; i -= 2) ord[i / 2] = min(ord[i], ord[i ^ 1]);
    }
    int lca(int v, int w) {
        int lo = head[ind[v]], hi = head[ind[w]], ret = INT_MAX; if (lo > hi) swap(lo, hi);
        for (lo += size, hi += size; lo <= hi; lo = (lo + 1) / 2, hi = (hi - 1) / 2) ret = min(ret, min(ord[lo], ord[hi]));
        return ret;
    }
};
```

2.5.5.3 LCA using Range Minimum Query

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

// Supports queries for the lowest common ancestor of 2 vertices in a tree by reduing the problem
// to a range minimum query
// Time Complexity:
//    run: O(V log V)
//    lca: O(1)
template <const int MAXV, const int MAXLGV> struct LCA_RMQ {
    int ind, head[MAXV], dep[MAXV], rmq[MAXLGV][2 * MAXV]; vector<int> adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void dfs(int v, int prev, int d) {
        dep[v] = d; rmq[0][head[v] = ind++] = v;
        for (int w : adj[v]) if (w != prev) { dfs(w, v, d + 1); rmq[0][ind++] = v; }
    }
    int minDep(int v, int w) { return dep[v] < dep[w] ? v : w; }
    int RMQ(int l, int r) { int i = 31 - __builtin_clz(r - l + 1); return minDep(rmq[i][l], rmq[i][r - (1 << i) + 1]); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int V, int root = 0) {
        ind = 0; dfs(root, -1, 0); int lg = 32 - __builtin_clz(V * 2 - 1);
        for (int i = 0; i < lg; i++) for (int j = 0; j < ind; j++) rmq[i + 1][j] = minDep(rmq[i][j], rmq[i][min(j + (1 << i), ind -
            1)]);
    }
    int lca(int v, int w) { if (head[v] > head[w]) swap(v, w); return RMQ(head[v], head[w]); }
};
```

2.5.5.4 Tarjan Offline LCA

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

// Supports queries for the lowest common ancestor of 2 vertices in a tree using Tarjan's Offline algorithm
// Time Complexity: O(V + Q)
template <const int MAXV, const int MAXQ> struct TarjanOfflineLCA {
    const bool WHITE = false, BLACK = true; bool color[MAXV]; int par[MAXV], ans[MAXQ], UF[MAXV], Q = 0;
    vector<int> adj[MAXV]; vector<pair<int, int>> qAdj[MAXV];
    int find(int v) { return UF[v] < 0 ? v : UF[v] = find(UF[v]); }
    void join(int v, int w) {
        v = find(v); w = find(w);
        if (v == w) return;
        if (UF[v] > UF[w]) swap(v, w);
        UF[v] += UF[w]; UF[w] = v;
    }
    void dfs(int v, int prev) {
        par[v] = v;
        for (int w : adj[v]) if (w != prev) { dfs(w, v); join(v, w); par[find(v)] = v; }
        color[v] = BLACK;
        for (auto &&q : qAdj[v]) if (color[q.first] == BLACK) ans[q.second] = par[find(q.first)];
    }
    void clear() { for (int i = 0; i < MAXV; i++) { adj[i].clear(); qAdj[i].clear(); } Q = 0; }
    void run(int root = 0) { fill(UF, UF + MAXV, -1); fill(color, color + MAXV, WHITE); dfs(root, -1); }
    void query(int v, int w) { qAdj[v].emplace_back(w, Q); qAdj[w].emplace_back(v, Q); Q++; }
};
```

2.5.6 Queries

2.5.6.1 Heavy Light Decomposition

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

// Decomposes a tree into chains, such that a path from any vertex to the root will cover at most log V chains
// Time Complexity:
//    run: O(V)
//    lca: O(log V)
//    queryPath: O(log V) * (complexity of query)
// Memory Complexity: O(V)
template <const int MAXV> struct HLD {
    int dep[MAXV], par[MAXV], chain[MAXV], size[MAXV], head[MAXV], ind[MAXV], vert[MAXV], chainNum, curInd;
    vector<int> adj[MAXV];
```

```
    void dfs(int v, int prev, int d) {
        dep[v] = d; par[v] = prev; size[v] = 1;
        for (int w : adj[v]) if (w != prev) { dfs(w, v, d + 1); size[v] += size[w]; }
    }
    void hld(int v, int prev) {
        if (head[chainNum] == -1) head[chainNum] = v; chain[v] = chainNum; ind[v] = curInd; vert[curInd++] = v; int maxInd = -1;
        for (int w : adj[v]) if (w != prev && (maxInd == -1 || size[maxInd] < size[w])) maxInd = w;
        if (maxInd != -1) hld(maxInd, v);
        for (int w : adj[v]) if (w != prev && w != maxInd) { chainNum++; hld(w, v); }
    }
    int merge(int a, int b); // to be implemented
    int query(int l, int r, bool up); // to be implemented
    int queryUp(int v, int w, bool up, bool includeW) {
        int ans = 0;
        while (chain[v] != chain[w]) {
            ans = up ? merge(ans, query(ind[head[chain[v]]], ind[v], up)) : merge(query(ind[head[chain[v]]], ind[v], up), ans);
            v = par[head[chain[v]]];
        }
        if (!includeW && v == w) return ans;
        return up ? merge(ans, query(ind[w] + !includeW, ind[v], up)) : merge(query(ind[w] + !includeW, ind[v], up), ans);
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
    void run(int root = 0) { chainNum = 0; curInd = 1; fill(head, head + MAXV, -1); dfs(root, -1, 0); hld(root, -1); }
    void addEdge(int a, int b) { adj[a].pb(b); adj[b].pb(a); }
    int lca(int v, int w) {
        while (chain[v] != chain[w]) {
            if (dep[head[chain[v]]] < dep[head[chain[w]]]) w = par[head[chain[w]]];
            else v = par[head[chain[v]]];
        }
        if (dep[v] < dep[w]) return v;
        return w;
    }
    int queryPath(int v, int w) {
        int lcavertex = lca(v, w);
        return merge(queryUp(v, lcavertex, true, false), queryUp(w, lcavertex, false, true));
    }
};
```

2.5.6.2 Mo's Algorithm for Trees

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

// Mo's algorithm on a tree, used to count the number of distinct integers on a path between two nodes
// Time Complexity: O(V + Q log Q + Q * max(B, Q / B) * (update complexity))
// Memory Complexity: O(V log V + Q)
template <const int MAXV, const int MAXQ, const int BLOCKSZ, const int MAXLGV> struct MoTree {
    struct Query {
        int l, r, lca, ind, block;
        bool operator < (const Query &q) const { return block == q.block ? r < q.r : block < q.block; }
    };
    int head[MAXV], dep[MAXV], rmq[MAXLGV][2 * MAXV], pre[MAXV], post[MAXV], vert[MAXV * 2], cnt[MAXV], ans[MAXQ], val[MAXV], temp[
        MAXV];
    int Q = 0, ind1, ind2, curAns; vector<int> adj[MAXV]; Query q[MAXQ]; bool vis[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void query(int v, int w) { q[Q++] = {v, w, 0, 0, 0}; }
    void dfs(int v, int prev, int d) {
        dep[v] = d; rmq[0][head[v] = ind1++] = v; vert[pre[v] = ind2++] = v;
        for (int w : adj[v]) if (w != prev) { dfs(w, v, d + 1); rmq[0][ind1++] = v; }
        vert[post[v] = ind2++] = v;
    }
    int minDep(int v, int w) { return dep[v] < dep[w] ? v : w; }
    int RMQ(int l, int r) { int i = 31 - __builtin_clz(r - l + 1); return minDep(rmq[i][l], rmq[i][r - (1 << i) + 1]); }
    int lca(int v, int w) { if (head[v] > head[w]) swap(v, w); return RMQ(head[v], head[w]); }
    void add(int x) { if (cnt[x]++ == 0) curAns++; }
    void rem(int x) { if (--cnt[x] == 0) curAns--; }
    void update(int v) {
        if (vis[v]) rem(val[v]);
        else add(val[v]);
        vis[v] = !vis[v];
    }
    void run(int V) {
        ind1 = ind2 = 0; int lg = 32 - __builtin_clz(V * 2 - 1); fill(cnt, cnt + V, 0); fill(vis, vis + V, false); dfs(0, -1, 0);
        for (int i = 0; i < lg; i++) for (int j = 0; j < ind1; j++) rmq[i + 1][j] = minDep(rmq[i][j], rmq[i][min(j + (1 << i), ind1
            - 1)]);
        copy(val, val + V, temp); sort(temp, temp + V); int k = unique(temp, temp + V) - temp;
        for (int v = 0; v < V; v++) val[v] = lower_bound(temp, temp + k, val[v]) - temp;
        for (int i = 0; i < Q; i++) {
            int v = q[i].l, w = q[i].r; q[i].lca = lca(v, w);
            if (pre[v] > pre[w]) swap(v, w);
            if (q[i].lca == v) {
                q[i].l = pre[v]; q[i].r = pre[w];
            } else {
                q[i].l = post[v]; q[i].r = pre[w];
            }
            q[i].ind = i; q[i].block = q[i].l / BLOCKSZ;
        }
        sort(q, q + Q); int l = q[0].l, r = l - 1; curAns = 0;
        for (int i = 0; i < Q; i++) {
            while (l < q[i].l) update(vert[l++]);
            while (l > q[i].l) update(vert[--l]);
            while (r < q[i].r) update(vert[++r]);
            while (r > q[i].r) update(vert[r--]);
            if (q[i].lca != vert[l] && q[i].lca != vert[r]) update(q[i].lca);
            ans[q[i].ind] = curAns;
```

```
            if (q[i].lca != vert[l] && q[i].lca != vert[r]) update(q[i].lca);
        }
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); Q = 0; }
};
```

2.5.6.3 Disjoint Union Sets for Trees

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

// Supports queries on graphs that involve finding the number of nodes in a subtree
// with a certain property
// Time Complexity: O(V log V + Q)
// Memory Complexity: O(V + Q)
template <const int MAXV, const int MAXQ> struct DSUTree {
    int Q = 0, ans[MAXQ], color[MAXV], temp[MAXV], size[MAXV], cnt[MAXV]; bool isHeavy[MAXV];
    vector<int> adj[MAXV]; vector<pair<int, int>> q[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void query(int v, int val) { q[v].emplace_back(val, Q++); }
    void getSize(int v, int prev) {
        size[v] = 1;
        for (int w : adj[v]) if (w != prev) { getSize(w, v); size[v] += size[w]; }
    }
    void add(int v, int prev, int delta) {
        cnt[color[v]] += delta;
        for (int w : adj[v]) if (w != prev && !isHeavy[w]) add(w, v, delta);
    }
    void dfs(int v, int prev, bool keep) {
        int maxSize = -1, heavyInd = -1;
        for (int w : adj[v]) if (w != prev && size[w] > maxSize) { maxSize = size[w]; heavyInd = w; }
        for (int w : adj[v]) if (w != prev && w != heavyInd) dfs(w, v, 0);
        if (heavyInd != -1) { dfs(heavyInd, v, 1); isHeavy[heavyInd] = 1; }
        add(v, prev, 1);
        for (auto &&qq : q[v]) ans[qq.second] = cnt[qq.first];
        if (heavyInd != -1) isHeavy[heavyInd] = 0;
        if (!keep) add(v, prev, -1);
    }
    void run(int V) {
        copy(color, color + V, temp); sort(temp, temp + V); int k = unique(temp, temp + V) - temp;
        for (int v = 0; v < V; v++) color[v] = lower_bound(temp, temp + k, color[v]) - temp;
        fill(isHeavy, isHeavy + V, false);
        getSize(0, -1); dfs(0, -1, 0);
    }
    void clear() { for (int i = 0; i < MAXV; i++) { adj[i].clear(); q[i].clear(); } Q = 0; }
};
```

2.5.7 Cycle

2.5.7.1 Cycle

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

// Finds a cycle in an undirected graph (including self loops and parallel edges)
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct Cycle {
    bool vis[MAXV]; int to[MAXV]; vector<int> adj[MAXV], cycle;
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    bool hasSelfLoop(int V) {
        for (int v = 0; v < V; v++) for (int w : adj[v]) {
            if (v != w) continue;
            cycle.clear(); cycle.push_back(v); cycle.push_back(v);
            return true;
        }
        return false;
    }
    bool hasParallelEdges(int V) {
        fill(vis, vis + V, false);
        for (int v = 0; v < V; v++) {
            for (int w : adj[v]) {
                if (vis[w]) {
                    cycle.clear(); cycle.push_back(v); cycle.push_back(w); cycle.push_back(v);
                    return true;
                }
                vis[w] = true;
            }
            for (int w : adj[v]) vis[w] = false;
        }
        return false;
    }
    void dfs(int v, int prev) {
        vis[v] = true;
        for (int w : adj[v]) {
            if (!cycle.empty()) return;
            if (!vis[w]) { to[w] = v; dfs(w, v); }
            else if (w != prev) {
                cycle.clear();
                for (int x = v; x != w; x = to[x]) cycle.push_back(x);
                cycle.push_back(w); cycle.push_back(v);
            }
        }
    }
```

```
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); cycle.clear(); }
    void run(int V) {
        if (hasSelfLoop(V) || hasParallelEdges(V)) return;
        fill(vis, vis + V, false);
        for (int v = 0; v < V; v++) if (!vis[v]) dfs(v, -1);
    }
};
```

2.5.7.2 Directed Cycle

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

// Finds a cycle in a directed graph
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct DirectedCycle {
    bool vis[MAXV], onStack[MAXV]; int to[MAXV]; vector<int> adj[MAXV], cycle;
    void dfs(int v) {
        vis[v] = onStack[v] = true;
        for (int w : adj[v]) {
            if (!cycle.empty()) return;
            else if (!vis[w]) { to[w] = v; dfs(w); }
            else if (onStack[w]) {
                cycle.clear();
                for (int x = v; x != w; x = to[x]) cycle.push_back(x);
                cycle.push_back(w); cycle.push_back(v);
            }
        }
        onStack[v] = false;
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); cycle.clear(); }
    void run(int V) {
        fill(vis, vis + MAXV, false);
        for (int v = 0; v < V; v++) if (!vis[v] && cycle.empty()) dfs(v);
    }
};
```

2.5.8 Matching

2.5.8.1 Bipartite

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

// Computes whether an undirected graph is bipartite, or whether it has an odd cycle.
// Time Complexity: O(V + E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct Bipartite {
    const bool WHITE = false, BLACK = true; bool bipartite, color[MAXV], vis[MAXV]; int to[MAXV]; vector<int> oddCycle, adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void bfs(int s) {
        queue<int> q; color[s] = WHITE; vis[s] = true; q.push(s);
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int w : adj[v]) {
                if (!vis[w]) { vis[w] = true; to[w] = v; color[w] = !color[v]; q.push(w); }
                else if (color[w] == color[v]) {
                    bipartite = false; oddCycle.clear(); stack<int> stk; int x = v, y = w;
                    while (x != y) { stk.push(x); oddCycle.push_back(y); x = to[x]; y = to[y]; }
                    stk.push(x);
                    while (!stk.empty()) { oddCycle.push_back(stk.top()); stk.pop(); }
                    oddCycle.push_back(w);
                    return;
                }
            }
        }
    }
    void run(int V) {
        bipartite = true; fill(color, color + V, WHITE); fill(vis, vis + V, false);
        for (int v = 0; v < V && bipartite; v++) if (!vis[v]) bfs(v);
    }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); oddCycle.clear(); }
};
```

2.5.8.2 Hopcroft-Karp Maximum Matching

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

// Computes the maximum matching (ans minimum vertex cover) on an unweighted bipartite graph
// Time Complexity: O((V + E) sqrt V)
// Memory Complexity: O(V + E)
template <const int MAXV> struct HopcroftKarpMaxMatch {
    int cardinality, mate[MAXV], dist[MAXV], pathDist; vector<int> adj[MAXV], typeA; bool color[MAXV];
    void addEdge(int v, int w) { adj[v].pb(w); adj[w].pb(v); }
    bool hasPath() {
        queue<int> q;
        for (int v : typeA) {
            if (mate[v] == -1) { dist[v] = 0; q.push(v); }
            else dist[v] = INT_MAX;
        }
        pathDist = INT_MAX;
```

```
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int w : adj[v]) {
                if (mate[w] == -1) {
                    if (pathDist == INT_MAX) pathDist = dist[v] + 1;
                } else if (dist[mate[w]] == INT_MAX) {
                    dist[mate[w]] = dist[v] + 1;
                    if (pathDist == INT_MAX) q.push(mate[w]);
                }
            }
        }
        return pathDist != INT_MAX;
    }
    bool dfs(int v) {
        for (int w : adj[v]) {
            if (mate[w] == -1) {
                if (pathDist == dist[v] + 1) { mate[w] = v; mate[v] = w; return true; }
            } else if (dist[mate[w]] == dist[v] + 1) {
                if (dfs(mate[w])) { mate[w] = v; mate[v] = w; return true; }
            }
        }
        dist[v] = INT_MAX; return false;
    }
    void init() { fill(mate, mate + MAXV, -1); fill(color, color + MAXV, false); }
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); typeA.clear(); }
    int getMaxMatch() {
        cardinality = 0;
        for (int v = 0; v < MAXV; v++) if (color[v]) typeA.push_back(v);
        while (hasPath()) for (int v : typeA) if (mate[v] == -1 && dfs(v)) cardinality++;
        return cardinality;
    }
};
```

2.5.8.3 Edmonds Matching

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

// Computes the maximum matching in a general undirected graph, such that each vertex
// is incident with at most one edge in the matching
// Time Complexity: O(V^2 * E)
// Memory Complexity: O(V + E)
template <const int MAXV> struct EdmondsMatching {
    int cardinality, match[MAXV], par[MAXV], id[MAXV], vis2[MAXV], stamp; bool vis[MAXV], blossom[MAXV]; vector<int> adj[MAXV];
    void addEdge(int v, int w) { adj[v].push_back(w); adj[w].push_back(v); }
    void markPath(int i, int b, int j) {
        for (; id[i] != b; i = par[match[i]]) { blossom[id[i]] = blossom[id[match[i]]] = true; par[i] = j; j = match[i]; }
    }
    int lca(int i, int j) {
        stamp++;
        while (true) {
            vis2[i = id[i]] = stamp;
            if (match[i] == -1) break;
            i = par[match[i]];
        }
        while (true) {
            if (vis2[j = id[j]] == stamp) return j;
            j = par[match[j]];
        }
    }
    int getAugmentingPath(int V, int s) {
        fill(par, par + V, -1); fill(vis, vis + V, false); iota(id, id + V, 0); queue<int> q; vis[q] = true; q.push(s);
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int w : adj[v]) {
                if (id[v] == id[w] || match[v] == w) continue;
                if (w == s || (match[w] != -1 && par[match[w]] != -1)) {
                    int newBase = lca(v, w); fill(blossom, blossom + V, false); markPath(v, newBase, w); markPath(w, newBase, v);
                    for (int i = 0; i < V; i++) {
                        if (!blossom[id[i]]) continue;
                        id[i] = newBase;
                        if (!vis[i]) { vis[i] = true; q.push(i); }
                    }
                } else if (par[w] == -1) {
                    par[w] = v;
                    if (match[w] == -1) return w;
                    vis[match[w]] = true;
                    q.push(match[w]);
                }
            }
        }
        assert(false);
        return -1;
    }
    int getMaxMatch(int V) {
        fill(match, match + V, -1); fill(vis2, vis2 + V, 0); stamp = 0;
        for (int i = 0; i < V; i++) {
            if (match[i] != -1) continue;
            int v = getAugmentingPath(V, i);
            while (v != -1) { int pv = par[v], ppv = match[pv]; match[v] = pv; match[pv] = v; v = ppv; }
        }
        cardinality = 0;
        for (int i = 0; i < V; i++) if (match[i] != -1) cardinality++;
        return cardinality / 2;
    }
```

```
    void clear() { for (int i = 0; i < MAXV; i++) adj[i].clear(); }
};
```

2.5.8.4 Hungarian Algorithm

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

// Solves the assignment problem of matching W workers to J jobs with the minimum cost
// where each worked can only be assigned to 1 job and each job can only be assigned to 1 worker
// Time Complexity: O(max(W, J)^3)
// Memory Complexity: O(max(W, J)^2)
template <const int MAXWJ, class unit> struct HungarianAlgorithm {
    unit INF, EPS; HungarianAlgorithm(unit INF, unit EPS) : INF(INF), EPS(EPS) {}
    int rows, cols, dim; bool committedWorkers[MAXWJ];
    unit matrix[MAXWJ][MAXWJ], costMatrix[MAXWJ][MAXWJ], labelByWorker[MAXWJ], labelByJob[MAXWJ], minSlackValueByJob[MAXWJ],
         totalCost;
    int minSlackWorkerByJob[MAXWJ], matchJobByWorker[MAXWJ], matchWorkerByJob[MAXWJ], parentWorkerByCommittedJob[MAXWJ];
    void match(int w, int j) { matchJobByWorker[w] = j; matchWorkerByJob[j] = w; }
    void updateLabeling(unit slack) {
        for (int w = 0; w < dim; w++) if (committedWorkers[w]) labelByWorker[w] += slack;
        for (int j = 0; j < dim; j++) {
            if (parentWorkerByCommittedJob[j] != -1) labelByJob[j] -= slack;
            else minSlackValueByJob[j] -= slack;
        }
    }
    void reduce() {
        for (int w = 0; w < dim; w++) {
            unit minCost = INF;
            for (int j = 0; j < dim; j++) minCost = min(minCost, costMatrix[w][j]);
            for (int j = 0; j < dim; j++) costMatrix[w][j] -= minCost;
        }
        for (int j = 0; j < dim; j++) {
            unit minCost = INF;
            for (int w = 0; w < dim; w++) minCost = min(minCost, costMatrix[w][j]);
            for (int w = 0; w < dim; w++) costMatrix[w][j] -= minCost;
        }
    }
    void computeInitialSolution() {
        for (int j = 0; j < dim; j++) labelByJob[j] = INF;
        for (int w = 0; w < dim; w++) for (int j = 0; j < dim; j++)
            if (costMatrix[w][j] < labelByJob[j]) labelByJob[j] = costMatrix[w][j];
    }
    void greedyMatch() {
        for (int w = 0; w < dim; w++) for (int j = 0; j < dim; j++)
            if (matchJobByWorker[w] == -1 && matchWorkerByJob[j] == -1 && costMatrix[w][j] - labelByWorker[w] - labelByJob[j] <= EPS
                )
                match(w, j);
    }
    int getUnmatchedWorker() {
        for (int w = 0; w < dim; w++) if (matchJobByWorker[w] == -1) return w;
        return dim;
    }
    void initialize(int w) {
        for (int i = 0; i < dim; i++) { committedWorkers[i] = false; parentWorkerByCommittedJob[i] = -1; }
        committedWorkers[w] = true;
        for (int j = 0; j < dim; j++) {
            minSlackValueByJob[j] = costMatrix[w][j] - labelByWorker[w] - labelByJob[j];
            minSlackWorkerByJob[j] = w;
        }
    }
    void execute() {
        while (true) {
            int minSlackWorker = -1, minSlackJob = -1; unit minSlackValue = INF;
            for (int j = 0; j < dim; j++) {
                if (parentWorkerByCommittedJob[j] == -1) {
                    if (minSlackValueByJob[j] < minSlackValue) {
                        minSlackValue = minSlackValueByJob[j]; minSlackWorker = minSlackWorkerByJob[j]; minSlackJob = j;
                    }
                }
            }
            if (minSlackValue > EPS) updateLabeling(minSlackValue);
            parentWorkerByCommittedJob[minSlackJob] = minSlackWorker;
            if (matchWorkerByJob[minSlackJob] == -1) {
                int committedJob = minSlackJob, parentWorker = parentWorkerByCommittedJob[committedJob];
                while (true) {
                    int temp = matchJobByWorker[parentWorker]; match(parentWorker, committedJob); committedJob = temp;
                    if (committedJob == -1) break;
                    parentWorker = parentWorkerByCommittedJob[committedJob];
                }
                return;
            } else {
                int worker = matchWorkerByJob[minSlackJob]; committedWorkers[worker] = true;
                for (int j = 0; j < dim; j++) {
                    if (parentWorkerByCommittedJob[j] == -1) {
                        unit slack = costMatrix[worker][j] - labelByWorker[worker] - labelByJob[j];
                        if (minSlackValueByJob[j] > slack) { minSlackValueByJob[j] = slack; minSlackWorkerByJob[j] = worker; }
                    }
                }
            }
        }
    }
    void init() { for (int i = 0; i < dim; i++) fill(matrix[i], matrix[i] + dim, 0); }
    unit run(int workers, int jobs) {
        dim = max(workers, jobs); rows = workers; cols = jobs;
```

```
        for (int i = 0; i < dim; i++) {
            labelByWorker[i] = labelByJob[i] =  minSlackValueByJob[i] = 0;
            minSlackWorkerByJob[i] = 0;
            matchJobByWorker[i] = matchWorkerByJob[i] = parentWorkerByCommittedJob[i] = -1;
            committedWorkers[i] = false;
            for (int j = 0; j < dim; j++) costMatrix[i][j] = matrix[i][j];
        }
        reduce(); computeInitialSolution(); greedyMatch();
        int w = getUnmatchedWorker();
        while (w < dim) { initialize(w); execute(); w = getUnmatchedWorker(); }
        for (w = 0; w < rows; w++) if (matchJobByWorker[w] >= cols) matchJobByWorker[w] = -1;
        for (int j = 0; j < cols; j++) if (matchWorkerByJob[j] >= rows) matchWorkerByJob[j] = -1;
        totalCost = 0;
        for (int w = 0; w < rows; w++) if (matchJobByWorker[w] != -1) totalCost += matrix[w][matchJobByWorker[w]];
        return totalCost;
    }
};
```

2.5.8.5 Stable Marriage

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

// Given N people of type A and N people of type B, and a list of their ranked choices
// for partners, the goal is to arrange N pairs such that if a person x of type A perfers
// a person y of type B more than their current partner, then person y preters their
// current partner more than x
// Time Complexity: O(N^2)
// Memory Complexity: O(N^2)
template <const int MAXN> struct StableMarriage {
    int aPrefs[MAXN], bPrefs[MAXN], curA[MAXN], mateA[MAXN], mateB[MAXN], bRanks[MAXN][MAXN];
    void solve(int N) {
        for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) bRanks[i][bPrefs[i][j]] = j;
        queue<int> q;
        for (int i = 0; i < N; i++) { curA[i] = mateB[i] = -1; q.push(i); }
        while (!q.empty()) {
            int A = q.front(); q.pop();
            while (true) {
                int B = aPrefs[A][++curA[A]];
                if (mateB[B] == -1) {
                    mateB[B] = A;
                    break;
                } else if (bRanks[B][A] < bRanks[B][mateB[B]]) {
                    q.push(mateB[B]); mateB[B] = A;
                    break;
                }
            }
        }
        for (int i = 0; i < N; i++) mateA[mateB[i]] = i;
    }
};
```

2.5.9 Network Flow

2.5.9.1 Edmonds Karp Max Flow

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

// Computes the maximum flow and minimum cut in a flow network using Edmonds Karp's algorithm
// Time Complexity: O(V * E^2)
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct EdmondsKarpMaxFlow {
    unit INF, EPS; EdmondsKarpMaxFlow(unit INF, unit EPS) : INF(INF), EPS(EPS) {}
    struct Edge {
        int to; unit cap, origCap; int next;
        Edge(int to, unit cap, int next) : to(to), cap(cap), origCap(cap), next(next) {}
    };
    int to[MAXV], last[MAXV]; bool vis[MAXV], cut[MAXV]; vector<Edge> e; unit maxFlow, minCut;
    void addEdge(int v, int w, unit vw, unit wv = 0) {
        e.emplace_back(w, vw, last[v]); last[v] = int(e.size()) - 1;
        e.emplace_back(v, wv, last[w]); last[w] = int(e.size()) - 1;
    }
    bool bfs(int V, int s, int t) {
        fill(vis, vis + V, false); fill(to, to + V, -1); queue<int> q; q.push(s); vis[s] = true;
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int i = last[v]; i != -1; i = e[i].next)
                if (e[i].cap > EPS && !vis[e[i].to]) { vis[e[i].to] = true; to[e[i].to] = i; q.push(e[i].to); }
        }
        return vis[t];
    }
    void init() { fill(last, last + MAXV, -1); fill(cut, cut + MAXV, false); }
    void clear() { e.clear(); }
    unit getFlow(int V, int s, int t) {
        maxFlow = 0;
        while (bfs(V, s, t)) {
            unit bottle = INF;
            for (int v = t; v != s; v = e[to[v] ^ 1].to) bottle = min(bottle, e[to[v]].cap);
            for (int v = t; v != s; v = e[to[v] ^ 1].to) { e[to[v]].cap -= bottle; e[to[v] ^ 1].cap += bottle; }
            maxFlow += bottle;
        }
        return maxFlow;
    }
```

```
    void inferMinCutDfs(int v) {
        cut[v] = true;
        for (int i = last[v]; i != -1; i = e[i].next) if (e[i].cap > 0 && !cut[e[i].to]) inferMinCutDfs(e[i].to);
    }
    unit inferMinCut(int V, int s) {
        inferMinCutDfs(s); minCut = 0;
        for (int v = 0; v < V; v++) if (cut[v]) for (int i = last[v]; i != -1; i = e[i].next)
            if (!cut[e[i].to]) minCut += e[i].origCap;
        return minCut;
    }
};
```

2.5.9.2 Dinic Max Flow

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

// Computes the maximum flow and minimum cut in a flow network using Dinic's algorithm
// Time Complexity: O(V^2 * E), O(min(V^(2/3), E^(1/2)) * E) for unit capacities
// Memory Complexity: O(V + E)
template <const int MAXV, class unit> struct DinicMaxFlow {
    unit INF, EPS; DinicMaxFlow(unit INF, unit EPS) : INF(INF), EPS(EPS) {}
    struct Edge {
        int to; unit cap, origCap; int next;
        Edge(int to, unit cap, int next) : to(to), cap(cap), origCap(cap), next(next) {}
    };
    int level[MAXV], last[MAXV]; bool vis[MAXV], cut[MAXV]; vector<Edge> e; unit maxFlow, minCut;
    void addEdge(int v, int w, unit vw, unit wv = 0) {
        e.emplace_back(w, vw, last[v]); last[v] = int(e.size()) - 1;
        e.emplace_back(v, wv, last[w]); last[w] = int(e.size()) - 1;
    }
    bool bfs(int V, int s, int t) {
        fill(level, level + V, -1); level[s] = 0; queue<int> q; q.push(s);
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int i = last[v]; i != -1; i = e[i].next)
                if (e[i].cap > EPS && level[e[i].to] == -1) { level[e[i].to] = level[v] + 1; q.push(e[i].to); }
        }
        return level[t] != -1;
    }
    unit dfs(int v, int t, unit flow) {
        if (v == t) return flow;
        unit ret = 0;
        for (int i = last[v]; i != -1; i = e[i].next) if (e[i].cap > EPS && level[e[i].to] == level[v] + 1) {
            unit res = dfs(e[i].to, t, min(flow, e[i].cap));
            ret += res; e[i].cap -= res; e[i ^ 1].cap += res; flow -= res;
            if (abs(flow) <= EPS) break;
        }
        return ret;
    }
    void init() { fill(last, last + MAXV, -1); fill(cut, cut + MAXV, false); }
    void clear() { e.clear(); }
    unit getFlow(int V, int s, int t) {
        maxFlow = 0;
        while (bfs(V, s, t)) maxFlow += dfs(s, t, INF);
        return maxFlow;
    }
    void inferMinCutDfs(int v) {
        cut[v] = true;
        for (int i = last[v]; i != -1; i = e[i].next) if (e[i].cap > 0 && !cut[e[i].to]) inferMinCutDfs(e[i].to);
    }
    unit inferMinCut(int V, int s) {
        inferMinCutDfs(s); minCut = 0;
        for (int v = 0; v < V; v++) if (cut[v]) for (int i = last[v]; i != -1; i = e[i].next)
            if (!cut[e[i].to]) minCut += e[i].origCap;
        return minCut;
    }
};
```

2.5.9.3 Max Flow Min Cost

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

// Computes the maximum flow using a path with the minimum cost
// Time Complexity: O(VEB log V) where B the the upper bound on the largest supply of any node
// Memory Complexity: O(V + E)
template <const int MAXV, class flowUnit, class costUnit> struct MaxFlowMinCost {
    flowUnit FLOW_INF, FLOW_EPS; costUnit COST_INF, COST_SMALL_INF;
    MaxFlowMinCost(flowUnit FLOW_INF, flowUnit FLOW_EPS, costUnit COST_INF, costUnit COST_SMALL_INF) :
        FLOW_INF(FLOW_INF), FLOW_EPS(FLOW_EPS), COST_INF(COST_INF), COST_SMALL_INF(COST_SMALL_INF) {}
    struct Edge {
        int from, to; flowUnit cap; costUnit origCost, cost; int next;
        Edge(int from, int to, flowUnit cap, costUnit cost, int next) :
            from(from), to(to), cap(cap), origCost(cost), cost(cost), next(next) {}
    };
    int last[MAXV], prev[MAXV], index[MAXV]; costUnit phi[MAXV], dist[MAXV]; vector<Edge> e; bool hasNegativeEdgeCost;
    void addEdge(int u, int v, flowUnit flow, costUnit cost) {
        if (cost < 0) hasNegativeEdgeCost = true;
        e.emplace_back(u, v, flow, cost, last[u]); last[u] = int(e.size()) - 1;
        e.emplace_back(v, u, 0, -cost, last[v]); last[v] = int(e.size()) - 1;
    }
    void bellmanFord(int V, int s, int t) {
        fill(phi, phi + V, COST_SMALL_INF); phi[s] = 0;
        for (int j = 0; j < V - 1; j++) for (int i = 0; i < int(e.size()); i++)
```

```
            if (e[i].cap > FLOW_EPS) phi[e[i].to] = min(phi[e[i].to], phi[e[i].from] + e[i].cost);
    }
    bool dijkstra(int V, int s, int t) {
        fill(dist, dist + V, COST_INF); fill(prev, prev + V, -1); fill(index, index + V, -1);
        priority_queue<pair<costUnit, int>, vector<pair<costUnit, int>>, greater<pair<costUnit, int>>> PQ;
        PQ.emplace(dist[s] = 0, s);
        while (!PQ.empty()) {
            pair<costUnit, int> v = PQ.top(); PQ.pop();
            if (v.first > dist[v.second]) continue;
            for (int next = last[v.second]; next != -1; next = e[next].next) {
                if (abs(e[next].cap) <= FLOW_EPS) continue;
                costUnit d = dist[v.second] + e[next].cost + phi[v.second] - phi[e[next].to];
                if (dist[e[next].to] <= d) continue;
                prev[e[next].to] = v.second; index[e[next].to] = next;
                PQ.emplace(dist[e[next].to] = d, e[next].to);
            }
        }
        return dist[t] != COST_INF;
    }
    void init() { fill(last, last + MAXV, -1); hasNegativeEdgeCost = false; }
    void clear() { e.clear(); }
    pair<flowUnit, costUnit> getMaxFlowMinCost(int V, int s, int t) {
        flowUnit flow = 0; costUnit cost = 0; fill(phi, phi + V, 0);
        if (hasNegativeEdgeCost) bellmanFord(V, s, t);
        while (dijkstra(V, s, t)) {
            flowUnit aug = FLOW_INF; int cur = t;
            while (prev[cur] != -1) { aug = min(aug, e[index[cur]].cap); cur = prev[cur]; }
            flow += aug; cur = t;
            while (prev[cur] != -1) {
                e[index[cur]].cap -= aug; e[index[cur] ^ 1].cap += aug;
                cost += aug * e[index[cur]].origCost; cur = prev[cur];
            }
            for (int v = 0; v < V; v++) if (dist[v] != COST_INF) phi[v] += dist[v];
        }
        return {flow, cost};
    }
};
```

2.5.10 Dynamic Programming

2.5.10.1 Maximum Nonadjacent Sum

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

// Computes the maximum sum of non adjacent vertices on any path between
// a source an (optional) destination vertex on a Directed Acyclic Graph
// Time Complexity: O(V)
// Memory Complexity: O(V)
template <const int MAXV> struct MaxNonadjacentSum {
    vector<int> adj[MAXV]; int val[MAXV], dp[MAXV][2];
    void addEdge(int v, int w) { adj[v].push_back(w); }
    int dfs(int v, int t, bool take) {
        if (v == t) return take;
        if (dp[v][take] != -1) return dp[v][take];
        int ret = INT_MIN;
        for (int w : adj[v]) {
            if (take) ret = max(ret, dfs(w, false) + val[v]);
            ret = max(ret, dfs(w, true));
        }
        return dp[v][take] = ret;
    }
    int solve(int s, int t) {
        fill(dp, dp + MAXV, -1);
        return dfs(s, t, true);
    }
};
```

2.6 Math

2.6.1 Bisection Root Finding

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

// Finds the root of a function using a bisection search
// Time Complexity: O(log((hi - lo) / EPS)) * (cost to compute f(x))
template <class T, class F> T bisection_root_finding(F f, T lo, T hi, T EPS) {
    T mid;
    do {
        mid = lo + (hi - lo) / 2;
        if (f(mid) < 0) lo = mid;
        else hi = mid;
    } while (hi - lo >= EPS);
    return mid;
}
```

2.6.2 Newton Root Finding

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

```cpp
// Find the root of a function using Newton's Method, given an initial guess x0
// Time Complexity: O(-log(EPS)) * cost to compute f(x), faster in practice
template <class T, class F> T newton(F f, F df, T x0, T EPS) {
    T cur = x0, next = x0;
    do { cur = next; next = cur - f(cur) / df(cur); } while (abs(next - cur) > EPS);
    return next;
}
```

2.6.3 Ternary Search for Maximum or Minimum

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

// Ternary search to find the minimum of a function
// Time Complexity: O(log((hi - lo) / EPS)) * (cost to compute f(x))
template <class T, class F> ternary_search(F f, T lo, T hi, T EPS) {
    T m1, m2;
    do {
        m1 = lo + (hi - lo) / 3; m2 = hi - (hi - lo) / 3;
        if (f(m1) < f(m2)) hi = m2; // f(m1) < f(m2) for minimum, f(m1) > f(m2) for maximum
        else lo = m1;
    } while (abs(hi - lo) >= EPS);
    return lo + (hi - lo) / 2;
}

// Ternary search to find the minimum of a function at integral values
// Time Complexity: O(log(hi - lo)) * (cost to compute f(x))
template <class T, class F> ternary_search(F f, T lo, T hi) {
    static_assert(is_integral<T>::value, "T must be an integral type"); T mid;
    while (lo < hi) {
        mid = lo + (hi - lo) / 2;
        if (f(mid) < f(mid + 1)) hi = mid; // f(mid) < f(mid + 1) for minimum, f(mid) > f(mid + 1) for maximum
        else lo = mid + 1;
    }
    return lo;
}
```

2.6.4 Combinatorics

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

// n! % m
// Time Complexity: O(n)
template <class T> T factorial(T n, T m) {
    T ret = 1;
    for (int i = 2; i <= n; i++) ret = (ret * i) % m;
    return ret;
}

// n! % p for a prime p
// Time Complexity: O(p log n)
template <class T> T factorialPrime(T n, T p) {
    T ret = 1, h = 0;
    while (n > 1) {
        ret = (ret * ((n / p) % 2 == 1 ? p - 1 : 1)) % p; h = n % p;
        for (int i = 2; i <= h; i++) ret = (ret * i) % p;
        n /= p;
    }
    return ret;
}

// a * b % mod
// Time Complexity: O(log b)
template <class T> T multMod(T a, T b, T mod) {
    T x = 0, y = a % mod;
    for (; b > 0; b /= 2, y = (y + y) % mod) if (b % 2 == 1) x = (x + y) % mod;
    return x;
}

// base ^ pow % mod
// Time Complexity: O(log pow)
template <class T> T powMod(T base, T pow, T mod) {
    T x = 1, y = base % mod;
    for (; pow > 0; pow /= 2, y = y * y % mod) if (pow % 2 == 1) x = x * y % mod;
    return x;
}

// Modular Multiplicative Inverse of i in Zp for a prime p
// Time Complexity: O(log p)
template <class T> T multInv(T i, T p) {
    return powMod(i, p - 2, p) % p;
}

// i / j % p for a prime p
// Time Complexity: O(log p)
template <class T> T divMod(T i, T j, T p) {
    return i * powMod(j, p - 2, p) % p;
}

// n choose k
// Time Complexity: O(min(k, n - k))
long long choose(int n, int k) {
    if (n < k) return 0;
    if (k > n - k) k = n - k;
    long long ret = 1;
    for (int i = 0; i < k; i++) ret = ret * (n - i) / (i + 1);
```

```
    return ret;
}

// n choose k % p for a prime p
// Time Complexity: O(min(k, n - k))
template <class T> T choose(int n, int k, T p) {
    if (n < k) return 0;
    if (k > n - k) k = n - k;
    T num = 1, den = 1;
    for (int i = 0; i < k; i++) { num = (num * (n - i)) % p; den = (den * (i + 1)) % p; }
    return divMod(num, den, p);
}

// n choose k % p
// Time Complexity: O(log p) if factorials are precomputed
template <class T> T fastChoose(int n, int k, T p) {
    return divMod(divMod(factorial(n, p), factorial(k, p), p), factorial(n - k, p), p);
}

// choosing k elements from n items with replacement, modulo p
// Time Complexity: O(log p) if factorials are precomputed
template <class T> T multiChoose(int n, int k, T p) {
    return fastChoose(n + k - 1, k, p);
}

// n permute k
// Time Complexity: O(min(k, n - k))
long long permute(int n, int k) {
    if (n < k) return 0;
    if (k > n - k) k = n - k;
    long long ret = 1;
    for (int i = 0; i < k; i++) ret = ret * (n - i);
    return ret;
}

// n permute k % p
// Time Complexity: O(min(k, n - k))
template <class T> T permute(int n, int k, T p) {
    if (n < k) return 0;
    if (k > n - k) k = n - k;
    T ret = 1;
    for (int i = 0; i < k; i++) ret = ret * (n - i) % p;
    return ret;
}

// n permute k % p
// Time Complexity: O(log p) if factorials are precomputed
template <class T> T fastPermute(int n, int k, T p) {
    return divMod(factorial(n, p), factorial(k, p), p);
}
```

2.6.5 Greatest Common Divisor

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

// Time Complexity of all functions: O(log max(a, b))

// Computes gcd(a, b)
template <typename T> T gcd(T a, T b) {
    while (b != 0) { T temp = b; b = a % b; a = temp; }
    return abs(a);
}

// Computes gcd(a b)
template <typename T> T gcdRec(T a, T b) { return b == 0 ? abs(a) : gcdRec(b, a % b); }

// Extended Euclidean Algorithm to compute x and y, where ax + by = gcd(a, b)
template <typename T> T EEA(T a, T b, T &x, T &y) {
    if (b == 0) { x = 1; y = 0; return abs(a); }
    T x1, y1, g = EEA(b, a % b, x1, y1); y = x1 - (a / b) * y1; x = y1;
    return g;
}

// Computes the multiplicative inverse of a in Zn
template <typename T> T multInv(T a, T n) {
    T x, y;
    assert(EEA(a, n, x, y) == 1); // otherwise, there is no inverse
    return (x % n + n) % n;
}

// Solves the linear congruence ax = c mod m
// return the value of x, and the modulus of the answer
template <typename T> pair<T, T> solveCongruence(T a, T c, T m) {
    T x, y, g = EEA(a, m, x, y);
    assert(c % g == 0); // otherwise there is no solution
    x = (x % m + m) % m; x = (x * c / g) % (m / g);
    return make_pair(x, m / g);
}
```

2.6.6 Primes

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

// Determines whether N is prime
// Time Complexity: O(sqrt N)
// Memory Complexity: O(1)
```

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

// Returns the prime factors of N
// Time Complexity: O(sqrt N)
// Memory Complexity: O(1)
vector<long long> primeFactor(long long x) {
    vector<long long> ret;
    for (long long i = 2; i * i <= x; i++) while (x % i == 0) { ret.push_back(i); x /= i; }
    if (x > 1) ret.push_back(x);
    return ret;
}

template <class T> T powMod(T base, T pow, T mod) {
    T x = 1, y = base % mod;
    for (; pow > 0; pow /= 2, y = y * y % mod) if (pow % 2 == 1) x = x * y % mod;
    return x;
}

// Determines whether N is prime using the Miller Rabin Primality Test
// Time Complexity: O((log N)^3) * iterations
// Memory Complexity: O(1)
template <class T> bool millerRabin(T N, int iterations) {
    if (N < 2 || (N != 2 && N % 2 == 0)) return false;
    mt19937_64 rng(time(0)); T s = N - 1;
    while (s % 2 == 0) s /= 2;
    for (int i = 0; i < iterations; i++) {
        T temp = s, r = powMod(T(rng() % (N - 1) + 1), temp, N);
        while (temp != N - 1 && r != 1 && r != N - 1) { r = r * r % N; temp *= 2; }
        if (r != N - 1 && temp % 2 == 0) return false;
    }
    return true;
}

// Sieve of Erathosthenes to identify primes and the smallest prime factor of each number
// Time Complexity:
//   sieve: O(N)
//   primeFactor: O(log x)
// Memory Complexity: O(N)
template <const int MAXN> struct Sieve {
    bool isPrime[MAXN], SPF[MAXN]; vector<int> primes;
    void run(int N) {
        primes.clear(); fill(isPrime, isPrime + N + 1, true); isPrime[0] = isPrime[1] = false;
        for (int i = 2; i <= N; i++) {
            if (isPrime[i]) { primes.push_back(i); SPF[i] = i; }
            for (int j = 0; j < (int) primes.size() && i * primes[j] <= N && primes[j] <= SPF[i]; j++) {
                isPrime[i * primes[j]] = false; SPF[i * primes[j]] = primes[j];
            }
        }
    }
    vector<int> primeFactor(int x) {
        vector<int> ret;
        while (x != 1) { ret.push_back(SPF[x]); x /= SPF[x]; }
        return ret;
    }
};

// Segmented Sieve of Erathosthenes to identify primes between st and en
// Time Complexity: O(en - st)
// Memory Complexity: O(sqrt(en) + en - st)
template <const int MAXE, const int MAXRANGE> struct SegmentedSieve {
    bool sieve1[int(sqrt(MAXE)) + 5], sieve2[MAXRANGE]; vector<long long> primes;
    void run(long long st, long long en) { // [st, en] (inclusive)
        primes.clear(); st = max(st, 2LL); long long sqrtEn = sqrt(en);
        fill(sieve1, sieve1 + sqrtEn + 1, false); fill(sieve2, sieve2 + en - st + 1, false); sieve1[0] = sieve1[1] = true;
        for (long long i = 2; i <= sqrtEn; i++) {
            if (sieve1[i]) continue;
            for (long long j = i * i; j <= sqrtEn; j += i) sieve1[j] = true;
            for (long long j = (st + i - 1) / i * i; j <= en; j += i) if (j != i && !sieve2[j - st]) sieve2[j - st] = true;
        }
        for (long i = 0; i < en - st + 1; i++) if (!sieve2[i]) primes.push_back(st + i);
    }
};

// Determines the factors of all numbers from 1 to N
// Time Complexity: O(N log N)
// Memory Complexity: O(N log N)
template <const int MAXN> struct Factors {
    vector<int> factors[MAXN];
    void run(int N) { for (int i = 1; i <= N; i++) for (int j = i; j <= N; j += i) factors[j].push_back(i); }
    void clear() { for (int i = 0; i < MAXN; i++) factors[i].clear(); }
};
```

2.6.7 Fast Fourier Transformations

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

// Time Complexity of all functions: O(log max(size(a), size(b)))

const double PI = 3.14159265358979323846;

void fft(vector<complex<double>> &a, bool invert) {
    int N = int(a.size());
```

```
    for (int i = 1, j = 0; i < N; i++) {
        int bit = N >> 1;
        for (; j >= bit; bit >>= 1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (int len = 2; len <= N; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        complex<double> wlen(cos(ang), sin(ang));
        for (int i = 0; i < N; i += len) {
            complex<double> w(1, 0);
            for (int j = 0; j < len / 2; j++) {
                complex<double> u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v; a[i + j + len / 2] = u - v; w *= wlen;
            }
        }
    }
    if (invert) for (int i = 0; i < N; i++) a[i] /= N;
}

// Multiplies 2 large integers
void multiplyInteger(vector<int> &a, vector<int> &b, vector<int> &res) {
    vector<complex<double>> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int N = int(a.size()) + int(b.size());
    while (N & (N - 1)) N++;
    fa.resize(N); fb.resize(N); fft(fa, false); fft(fb, false);
    for (int i = 0; i < N; i++) fa[i] *= fb[i];
    fft(fa, true); res.resize(N);
    int carry = 0;
    for (int i = 0; i < N; i++) { res[i] = (int) (fa[i].real() + 0.5) + carry; carry = res[i] / 10; res[i] %= 10; }
    while (int(res.size()) > 1 && res.back() == 0) res.pop_back();
}

// Multiplies 2 polynomial
template <class T> void multiplyPolynomial(vector<T> &a, vector<T> &b, vector<T> &res) {
    vector<complex<double>> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int N = int(a.size()) + int(b.size()) - 1; bool isIntegral = is_integral<T>::value;
    while (N & (N - 1)) N++;
    fa.resize(N); fb.resize(N); fft(fa, false); fft(fb, false);
    for (int i = 0; i < N; i++) fa[i] *= fb[i];
    fft(fa, true); res.resize(N);
    for (int i = 0; i < N; i++) res[i] = isIntegral ? round(real(fa[i])) : real(fa[i]);
    while (int(res.size()) > 1 && res.back() == 0) res.pop_back();
}
```

2.6.8 Gaussian Elimination

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

// Solve the equation Ax = b for Matrix A, and Vectors b, x
// Time Complexity: O(N^3)
// Memory Complexity: O(N^2)
template <const int MAXN> struct GaussianElimination {
    double EPS, A[MAXN][MAXN], b[MAXN], x[MAXN]; GaussianElimination(double EPS) : EPS(EPS) {}
    void solve(int N) {
        for (int p = 0; p < N; p++) {
            int m = p;
            for (int i = p + 1; i < N; i++) if (abs(A[i][p]) > abs(A[m][p])) m = i;
            swap(A[p], A[m]); swap(b[p], b[m]);
            if (abs(A[p][p]) <= EPS) throw runtime_error("Matrix is singular or nearly singular");
            for (int i = p + 1; i < N; i++) {
                double alpha = A[i][p] / A[p][p]; b[i] -= alpha * b[p];
                for (int j = p; j < N; j++) A[i][j] -= alpha * A[p][j];
            }
        }
        fill(x, x + N, 0);
        for (int i = N - 1; i >= 0; i--) {
            double sum = 0.0;
            for (int j = i + 1; j < N; j++) sum += A[i][j] * x[j];
            x[i] = (b[i] - sum) / A[i][i];
        }
    }
};
```

2.6.9 Matrix

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

// Metrix data structure
template <class T> struct Matrix : vector<vector<T>> {
    int N, M; Matrix(int N, int M) : N(N), M(M) { for (int i = 0; i < N; i++) this->push_back(vector<T>(M, 0)); }
};

// Returns the identiy matrix of dimension N
// Time Complexity: O(N^2)
template <class T> Matrix<T> identity(int N) {
    Matrix<T> A(N, N);
    for (int i = 0; i < N; i++) A[i][i] = 1;
    return A;
}

// Returns A + B
// Time Complexity: O(N^2)
```

```
template <class T> add(const Matrix<T> &A, const Matrix<T> &B) {
    assert(A.N == B.N && A.M == B.M); Matrix<T> C(A.N, A.M);
    for (int i = 0; i < C.N; i++) for (int j = 0; j < C.M; j++) C[i][j] = A[i][j] + B[i][j];
    return C;
}

// Returns A - B
// Time Complexity: O(N^2)
template <class T> sub(const Matrix<T> &A, const Matrix<T> &B) {
    assert(A.N == B.N && A.M == B.M); Matrix<T> C(A.N, A.M);
    for (int i = 0; i < C.N; i++) for (int j = 0; j < C.M; j++) C[i][j] = A[i][j] - B[i][j];
    return C;
}

// Returns A * B
// Time Complexity: O(N^3)
template <class T> times(const Matrix<T> &A, const Matrix<T> &B) {
    assert(A.M == B.N); Matrix<T> C(A.N, B.M);
    for (int i = 0; i < A.N; i++) for (int j = 0; j < B.M; j++) for (int k = 0; k < A.M; k++) C[i][j] += A[i][k] * B[k][j];
    return C;
}

// Returns A ^ pow
// Time Complexity: O(N^3 log pow)
template <class T> pow(const Matrix<T> &A, long long pow) {
    assert(A.N == A.M);
    Matrix<T> x = identity(A.N), y = A;
    for (; pow > 0; pow /= 2, y = times(y, y)) if (pow % 2 == 1) x = times(x, y);
    return x;
}
```

2.6.10 XOR Satisfiability

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

// Solves the boolean satisfiability problem for so the rows of A when xored evaluate to b
// If A[i][j] = 1, then the value xj is xored in equation i
// If A[i][j] = 0, then the value !xj is xored in equation i
// Time Complexity: O(N^3)
// Memory Complexity: O(N^2)
template <const int MAXN> struct GaussianElimination {
    bool A[MAXN][MAXN], b[MAXN], x[MAXN];
    void solve(int N) {
        for (int p = 0; p < N; p++) {
            int m = p;
            for (int i = 0; i < p; i++) {
                int j = i;
                while (j < N && !A[i][j]) j++;
                if (j == p) m = i;
            }
            for (int i = p + 1; i < N; i++) if (A[i][p]) m = i;
            if (!A[m][p]) continue;
            swap(A[p], A[m]); swap(b[p], b[m]);
            for (int i = 0; i < N; i++) if (p != i && A[i][p]) {
                b[i] ^= b[p];
                for (int j = p; j < N; j++) A[i][j] ^= A[p][j];
            }
        }
        copy(b, b + N, x);
        for (int i = N - 1; i >= 0; i--) {
            if (!A[i][i] && x[i]) throw runtime_error("Matrix has no solution");
            if (x[i]) for (int j = i - 1; j >= 0; j--) { x[j] ^= A[j][i]; A[j][i] = false; }
        }
        return b;
    }
};
```

2.7 Queries

2.7.1 Mo's Algorithm

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

// Mo's algorithm, used to count the number of distinct integers in a subarray
// Time Complexity: O(N + Q log Q + Q * max(B, Q / B) * (update complexity))
// Memory Complexity: O(N + Q)
template <const int MAXN, const int MAXQ, const int BLOCKSZ> struct Mo {
    struct Query {
        int l, r, ind, block;
        bool operator < (const Query &q) const { return block == q.block ? r < q.r : block < q.block; }
    };
    int Q = 0, cnt[MAXN], ans[MAXQ], val[MAXN], temp[MAXN], curAns; Query q[MAXQ];
    void query(int l, int r) { q[Q] = {l, r, Q, l / BLOCKSZ}; Q++; }
    void add(int x) { if (cnt[x]++ == 0) curAns++; }
    void rem(int x) { if (--cnt[x] == 0) curAns--; }
    void run(int N) {
        fill(cnt, cnt + N, 0); copy(val, val + N, temp); sort(temp, temp + N); int k = unique(temp, temp + N) - temp;
        for (int i = 0; i < N; i++) val[i] = lower_bound(temp, temp + k, val[i]) - temp;
        sort(q, q + Q); int l = q[0].l, r = l - 1; curAns = 0;
        for (int i = 0; i < Q; i++) {
            while (l < q[i].l) rem(val[l++]);
            while (l > q[i].l) add(val[--l]);
            while (r < q[i].r) add(val[++r]);
```

```
            while (r > q[i].r) rem(val[r--]);
            ans[q[i].ind] = curAns;
        }
    }
    void clear() { Q = 0; }
};
```

2.7.2 Mo's Algorithm with Updates

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

// Mo's algorithm, supporting point updates, used to count the number of distinct integers in a subarray
// Time Complexity: O(N + Q * ((N/B)^2 + B * (update complexity)))
// Memory Complexity: O(N + Q)
template <const int MAXN, const int MAXQ, const int BLOCKSZ> struct MoUpdates {
    int Q = 0, cnt[MAXN + MAXQ], ans[MAXQ], val[MAXN], temp[MAXN + MAXQ], curAns;
    pair<int, int> q[MAXQ]; bool isQuery[MAXQ]; vector<int> qs[MAXN / BLOCKSZ + 5][MAXN / BLOCKSZ + 5];
    void add(int x) { if (cnt[x]++ == 0) curAns++; }
    void rem(int x) { if (--cnt[x] == 0) curAns--; }
    void upd(int i, int x) { rem(val[i]); add(val[i] = x); }
    void query(int l, int r) { q[Q] = {l, r}; isQuery[Q++] = true; }
    void update(int i, int x) { q[Q] = {i, x}; isQuery[Q++] = false; }
    void run(int N) {
        int cur = N, blocks = (N - 1) / BLOCKSZ + 1; fill(cnt, cnt + N + Q, 0);
        for (int i = 0; i < Q; i++) if (!isQuery[i]) temp[cur++] = q[i].second;
        copy(val, val + N, temp); sort(temp, temp + cur); int k = unique(temp, temp + cur) - temp;
        for (int i = 0; i < N; i++) val[i] = lower_bound(temp, temp + k, val[i]) - temp;
        for (int i = 0; i < Q; i++) {
            if (isQuery[i]) qs[q[i].first / BLOCKSZ][q[i].second / BLOCKSZ].push_back(i);
            else {
                q[i].second = lower_bound(temp, temp + k, q[i].second) - temp;
                for (int bl = 0; bl <= q[i].first / BLOCKSZ; bl++) for (int br = q[i].first / BLOCKSZ; br < blocks; br++)
                    qs[bl][br].push_back(i);
            }
        }
        int l = 0, r = l - 1; curAns = 0; stack<pair<int, int>> revert;
        for (int bl = 0; bl < blocks; bl++) for (int br = bl; br < blocks; br++) {
            for (int i : qs[bl][br]) {
                if (isQuery[i]) {
                    while (l < q[i].first) rem(val[l++]);
                    while (l > q[i].first) add(val[--l]);
                    while (r < q[i].second) add(val[++r]);
                    while (r > q[i].second) rem(val[r--]);
                    ans[i] = curAns;
                } else {
                    revert.emplace(q[i].first, val[q[i].first]);
                    if (l <= q[i].first && q[i].first <= r) upd(q[i].first, q[i].second);
                    else val[q[i].first] = q[i].second;
                }
            }
            while (!revert.empty()) {
                pair<int, int> x = revert.top(); revert.pop();
                if (l <= x.first && x.first <= r) upd(x.first, x.second);
                else val[x.first] = x.second;
            }
        }
    }
    void clear() { Q = 0; }
};
```

2.8 Search

2.8.1 Binary Search

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

// returns the first value in the range [lo, hi) where f(x) == 0
// if there is no value in [lo, hi) where f(x) == 0, then it returns -1
// assumes that f(x) is non decreasing in the range [lo, hi)
// Time Complexity: O(log (hi - lo)) * (cost to compute f(x))
template <class T, class F> T getExact(T lo, T hi, F f) {
    hi--;
    while (lo <= hi) {
        T mid = lo + (hi - lo) / 2;
        auto fmid = f(mid);
        if (fmid < 0) lo = mid + 1;
        else if (fmid > 0) hi = mid - 1;
        else return mid;
    }
    return -1;
}

// returns the first value in the range [lo, hi) where f(x) is true
// if no value in [lo, hi) satisfies f(x), then it returns hi
// assumes that all values where f(x) is true are greater than all values where f(x) is false
// Time Complexity: O(log (hi - lo)) * (cost to compute f(x))
template <class T, class F> T getFirst(T lo, T hi, F f) {
    while (lo < hi) {
        T mid = lo + (hi - lo) / 2;
        if (f(mid)) hi = mid;
        else lo = mid + 1;
    }
```

```
    return lo;
}

// returns the last value in the range [lo, hi) where f(x) is true
// if no value in [lo, hi) satisfies f(x), then it returns lo - 1
// assumes that all values where f(x) is true are less than all values where f(x) is false
// Time Complexity: O(log (hi - lo)) * (cost to compute f(x))
template <class T, class F> T getLast(T lo, T hi, F f) {
    hi--;
    while (lo <= hi) {
        T mid = lo + (hi - lo) / 2;
        if (f(mid)) lo = mid + 1;
        else hi = mid - 1;
    }
    return hi;
}
```

2.8.2 Interpolation Search

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

// returns the first value in the range [lo, hi) where f(x) == key
// if there is no value in [lo, hi) where f(x) == key, then it returns -1
// assumes that f(x) is uniform and non decreasing in the range [lo, hi)
// Time Complexity: O(log log (hi - lo)) * (cost to compute f(x))
template <class T, class F> T interpolationSearch(T lo, T hi, F f, T key) {
    hi--;
    while (lo < hi) {
        T guess = ((key - f(lo)) / f(hi - lo)) * (hi - lo) + lo;
        auto fguess = f(guess);
        if (fguess < key) lo = guess + 1;
        else if (fguess > key) hi = guess - 1;
        else return guess;
    }
    return -1;
}
```

2.9 Sort

2.9.1 Counting Sort

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

// Sorts an array using counting sort, works with integer values
// Time Complexity: O(N + K) worst case for N elements over a range of K
// Memory Complexity: O(N + K) additional memory
template <class It> void counting_sort(It st, It en) {
    int n = en - st;
    if (n == 0) return;
    int maxVal = INT_MIN, minVal = INT_MAX, *count = new int[maxVal - minVal + 1], *b = new int[n];
    for (It i = st; i < en; i++) { maxVal = max(maxVal, *i); minVal = min(minVal, *i); }
    for (int i = 0; i <= maxVal - minVal; i++) count[i] = 0;
    for (It i = st; i < en; i++) count[*i - minVal]++;
    for (int i = 1; i <= maxVal - minVal; i++) count[i] += count[i - 1];
    for (It i = st; i < en; i++) b[--count[*i - minVal]] = *i;
    for (It i = st; i < en; i++) *i = b[i - st];
    delete[] (count); delete[] (b);
}
```

2.9.2 Heap Sort

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

// Sorts an array using heap sort
// Time Complexity: O(N log N) worse case
// Memory Complexity: O(1) additional memory

template <class It> void heap_sort(It st, It en) {
    int n = en - st;
    for (int i = n / 2; i >= 1; i--) {
        int k = i;
        while (2 * k <= n) {
            int j = 2 * k;
            if (j < n && st[j - 1] < st[j]) j++;
            if (st[k - 1] >= st[j - 1]) break;
            swap(st[k - 1], st[j - 1]); k = j;
        }
    }
    while (n > 1) {
        swap(st[0], st[n - 1]); n--; int k = 1;
        while (2 * k <= n) {
            int j = 2 * k;
            if (j < n && st[j - 1] < st[j]) j++;
            if (st[k - 1] >= st[j - 1]) break;
            swap(st[k - 1], st[j - 1]); k = j;
        }
    }
}

////////////////// COMPARATOR SORT //////////////////
```

```
template <class It, class Comparator> void heap_sort(It st, It en, Comparator cmp) {
    int n = en - st;
    for (int i = n / 2; i >= 1; i--) {
        int k = i;
        while (2 * k <= n) {
            int j = 2 * k;
            if (j < n && cmp(st[j - 1], st[j])) j++;
            if (!cmp(st[k - 1], st[j - 1])) break;
            swap(st[k - 1], st[j - 1]); k = j;
        }
    }
    while (n > 1) {
        swap(st[0], st[n - 1]); n--; int k = 1;
        while (2 * k <= n) {
            int j = 2 * k;
            if (j < n && cmp(st[j - 1], st[j])) j++;
            if (!cmp(st[k - 1], st[j - 1])) break;
            swap(st[k - 1], st[j - 1]); k = j;
        }
    }
}
```

2.9.3 Merge Sort

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

// Sorts an array using merge sort
// Time Complexity: O(N log N) worse case
// Memory Complexity: O(N) additional memory

const int INSERTION_SORT_CUTOFF = 8;

template <class SrcIt, class DstIt>
void merge_sort(SrcIt src_st, SrcIt src_en, DstIt dst_st, DstIt dst_en) {
    int n = src_en - src_st;
    if (n <= INSERTION_SORT_CUTOFF) {
        for (int i = 0; i < n; i++) for (int j = i; j > 0 && dst_st[j] < dst_st[j - 1]; j--)
            swap(dst_st[j], dst_st[j - 1]);
        return;
    }
    int mid = (n - 1) / 2;
    merge_sort(dst_st, dst_st + mid + 1, src_st, src_st + mid + 1);
    merge_sort(dst_st + mid + 1, dst_en, src_st + mid + 1, src_en);
    if (src_st[mid + 1] >= src_st[mid]) { for (int i = 0; i < n; i++) dst_st[i] = src_st[i]; return; }
    for (int i = 0, j = mid + 1, k = 0; k < n; k++) {
        if (i > mid) dst_st[k] = src_st[j++];
        else if (j >= n) dst_st[k] = src_st[i++];
        else if (src_st[j] < src_st[i]) dst_st[k] = src_st[j++];
        else dst_st[k] = src_st[i++];
    }
}

template <class It> void merge_sort(It st, It en) {
    typedef typename std::iterator_traits<It>::value_type T;
    int n = en - st; T *aux = new T[n];
    for (int i = 0; i < n; i++) aux[i] = st[i];
    merge_sort(aux, aux + n, st, en); delete[] (aux);
}

////////////////// COMPARATOR SORT //////////////////

template <class SrcIt, class DstIt, class Comparator>
void merge_sort(SrcIt src_st, SrcIt src_en, DstIt dst_st, DstIt dst_en, Comparator cmp) {
    int n = src_en - src_st;
    if (n <= INSERTION_SORT_CUTOFF) {
        for (int i = 0; i < n; i++) for (int j = i; j > 0 && cmp(dst_st[j], dst_st[j - 1]); j--)
            swap(dst_st[j], dst_st[j - 1]);
        return;
    }
    int mid = (n - 1) / 2;
    merge_sort(dst_st, dst_st + mid + 1, src_st, src_st + mid + 1, cmp);
    merge_sort(dst_st + mid + 1, dst_en, src_st + mid + 1, src_en, cmp);
    if (!cmp(src_st[mid + 1], src_st[mid])) { for (int i = 0; i < n; i++) dst_st[i] = src_st[i]; return; }
    for (int i = 0, j = mid + 1, k = 0; k < n; k++) {
        if (i > mid) dst_st[k] = src_st[j++];
        else if (j >= n) dst_st[k] = src_st[i++];
        else if (cmp(src_st[j], src_st[i])) dst_st[k] = src_st[j++];
        else dst_st[k] = src_st[i++];
    }
}

template <class It, class Comparator> void merge_sort(It st, It en, Comparator cmp) {
    typedef typename std::iterator_traits<It>::value_type T;
    int n = en - st; T *aux = new T[n];
    for (int i = 0; i < n; i++) aux[i] = st[i];
    merge_sort(aux, aux + n, st, en, cmp); delete[] (aux);
}
```

2.9.4 Merge Sort Bottom Up

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

// Sorts an array using bottom up merge sort
```

```cpp
// Time Complexity: O(N log N) worse case
// Memory Complexity: O(N) additional memory

const int INSERTION_SORT_CUTOFF = 7;

template <class It> void merge_sort_buttom_up(It st, It en) {
    typedef typename std::iterator_traits<It>::value_type T;
    int n = en - st; T *aux = new T[n];
    for (int i = 0; i < n; i++) aux[i] = st[i];
    bool flag = true; int len = 1;
    for (; len * 2 <= INSERTION_SORT_CUTOFF && len < n; len *= 2);
    for (int lo = 0; lo < n; lo += len + len) {
        int hi = min(lo + len + len - 1, n - 1);
        for (int i = lo; i <= hi; i++) for (int j = i; j > lo && aux[j] < aux[j - 1]; j--)
            swap(aux[j], aux[j - 1]);
    }
    len *= 2;
    for (; len < n; len *= 2) {
        if (flag) {
            for (int lo = 0; lo < n; lo += len + len) {
                int mid = lo + len - 1, hi = min(lo + len + len - 1, n - 1);
                if (mid + 1 < n && aux[mid + 1] >= aux[mid]) {
                    for (int i = lo; i <= hi; i++) st[i] = aux[i];
                } else {
                    int i = lo, j = mid + 1;
                    for (int k = lo; k <= hi; k++) {
                        if (i > mid) st[k] = aux[j++];
                        else if (j > hi) st[k] = aux[i++];
                        else if (aux[j] < aux[i]) st[k] = aux[j++];
                        else st[k] = aux[i++];
                    }
                }
            }
        } else {
            for (int lo = 0; lo < n; lo += len + len) {
                int mid = lo + len - 1, hi = min(lo + len + len - 1, n - 1);
                if (mid + 1 < n && st[mid + 1] >= st[mid]) {
                    for (int i = lo; i <= hi; i++) aux[i] = st[i];
                } else {
                    int i = lo, j = mid + 1;
                    for (int k = lo; k <= hi; k++) {
                        if (i > mid) aux[k] = st[j++];
                        else if (j > hi) aux[k] = st[i++];
                        else if (st[j] < st[i]) aux[k] = st[j++];
                        else aux[k] = st[i++];
                    }
                }
            }
        }
        flag = !flag;
    }
    if (flag) for (int i = 0; i < n; i++) st[i] = aux[i];
    delete[] (aux);
}

////////////////// COMPARATOR SORT //////////////////

template <class It, class Comparator> void merge_sort_bottom_up(It st, It en, Comparator cmp) {
    typedef typename std::iterator_traits<It>::value_type T;
    int n = en - st; T *aux = new T[n];
    for (int i = 0; i < n; i++) aux[i] = st[i];
    bool flag = true; int len = 1;
    for (; len * 2 <= INSERTION_SORT_CUTOFF && len < n; len *= 2);
    for (int lo = 0; lo < n; lo += len + len) {
        int hi = min(lo + len + len - 1, n - 1);
        for (int i = lo; i <= hi; i++) for (int j = i; j > lo && cmp(aux[j], aux[j - 1]); j--)
            swap(aux[j], aux[j - 1]);
    }
    len *= 2;
    for (; len < n; len *= 2) {
        if (flag) {
            for (int lo = 0; lo < n; lo += len + len) {
                int mid = lo + len - 1, hi = min(lo + len + len - 1, n - 1);
                if (mid + 1 < n && !cmp(aux[mid + 1], aux[mid])) {
                    for (int i = lo; i <= hi; i++) st[i] = aux[i];
                } else {
                    int i = lo, j = mid + 1;
                    for (int k = lo; k <= hi; k++) {
                        if (i > mid) st[k] = aux[j++];
                        else if (j > hi) st[k] = aux[i++];
                        else if (cmp(aux[j], aux[i])) st[k] = aux[j++];
                        else st[k] = aux[i++];
                    }
                }
            }
        } else {
            for (int lo = 0; lo < n; lo += len + len) {
                int mid = lo + len - 1, hi = min(lo + len + len - 1, n - 1);
                if (mid + 1 < n && !cmp(st[mid + 1], st[mid])) {
                    for (int i = lo; i <= hi; i++) aux[i] = st[i];
                } else {
                    int i = lo, j = mid + 1;
                    for (int k = lo; k <= hi; k++) {
                        if (i > mid) aux[k] = st[j++];
                        else if (j > hi) aux[k] = st[i++];
                        else if (cmp(st[j], st[i])) aux[k] = st[j++];
```

```
                    else aux[k] = st[i++];
                }
            }
        }
    }
    flag = !flag;
}
if (flag) for (int i = 0; i < n; i++) st[i] = aux[i];
delete[] (aux);
}
```

2.9.5 Quick Sort 3 Way

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

// Sorts an array using 3-way quick sort
// Time Complexity: O(N^2) worse case, O(N log N) average case
// Memory Complexity: O(1) additional memory

const int INSERTION_SORT_CUTOFF = 8;
const int MEDIAN_OF_3_CUTOFF = 40;

template <class It> inline It median3(It a, It b, It c) {
    return (*a < *b ? (*b < *c ? b : (*a < *c ? c : a)) : (*c < *b ? b : (*c < *a ? c : a)));
}

template <class It> void quick_sort_3_way(It st, It en) {
    int n = en - st;
    if (n <= INSERTION_SORT_CUTOFF) {
        for (int i = 0; i < n; i++) for (int j = i; j > 0 && st[j] < st[j - 1]; j--)
            swap(st[j], st[j - 1]);
        return;
    } else if (n <= MEDIAN_OF_3_CUTOFF) {
        It m = median3(st, st + n / 2, en - 1); swap(*st, *m);
    } else {
        int eps = n / 8; It mid = st + n / 2;
        It ninther = median3(median3(st, st + eps, st + eps + eps),
                median3(mid - eps, mid, mid + eps),
                median3(en - 1 - eps - eps, en - 1 - eps, en - 1));
        swap(*st, *ninther);
    }
    int i = 0, j = n, p = 0, q = n; auto v = st[0];
    while (true) {
        while (st[++i] < v) if (i == n - 1) break;
        while (v < st[--j]) if (j == 0) break;
        if (i == j && st[i] == v) swap(st[++p], st[i]);
        if (i >= j) break;
        swap(st[i], st[j]);
        if (st[i] == v) swap(st[++p], st[i]);
        if (st[j] == v) swap(st[--q], st[j]);
    }
    i = j + 1;
    for (int k = 0; k <= p; k++, j--) swap(st[k], st[j]);
    for (int k = n - 1; k >= q; k--, i++) swap(st[k], st[i]);
    quick_sort_3_way(st, st + j + 1); quick_sort_3_way(st + i, en);
}

////////////////// COMPARATOR SORT //////////////////

template <class It, class Comparator> void quick_sort_3_way(It st, It en, Comparator cmp) {
    int n = en - st;
    if (n <= INSERTION_SORT_CUTOFF) {
        for (int i = 0; i < n; i++) for (int j = i; j > 0 && cmp(st[j], st[j - 1]); j--)
            swap(st[j], st[j - 1]);
        return;
    } else if (n <= MEDIAN_OF_3_CUTOFF) {
        It m = median3(st, st + n / 2, en - 1); swap(*st, *m);
    } else {
        int eps = n / 8; It mid = st + n / 2;
        It ninther = median3(median3(st, st + eps, st + eps + eps),
                median3(mid - eps, mid, mid + eps),
                median3(en - 1 - eps - eps, en - 1 - eps, en - 1));
        swap(*st, *ninther);
    }
    int i = 0, j = n, p = 0, q = n; auto v = st[0];
    while (true) {
        while (cmp(st[++i], v)) if (i == n - 1) break;
        while (cmp(v, st[--j])) if (j == 0) break;
        if (i == j && !cmp(st[i], v) && !cmp(v, st[i])) swap(st[++p], st[i]);
        if (i >= j) break;
        swap(st[i], st[j]);
        if (!cmp(st[i], v) && !cmp(v, st[i])) swap(st[++p], st[i]);
        if (!cmp(st[j], v) && !cmp(v, st[j])) swap(st[--q], st[j]);
    }
    i = j + 1;
    for (int k = 0; k <= p; k++, j--) swap(st[k], st[j]);
    for (int k = n - 1; k >= q; k--, i++) swap(st[k], st[i]);
    quick_sort_3_way(st, st + j + 1, cmp); quick_sort_3_way(st + i, en, cmp);
}
```

2.9.6 Quick Sort Dual Pivot

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

```cpp
// Sorts an array using dual pivot quick sort
// Time Complexity: O(N^2) worse case, O(N log N) average case
// Memory Complexity: O(1) additional memory

const int INSERTION_SORT_CUTOFF = 8;

template <class It> void quick_sort_dual(It st, It en) {
    int n = en - st;
    if (n <= 1) return;
    if (n <= INSERTION_SORT_CUTOFF) { // insertion sort
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0 && st[j] < st[j - 1]; j--) {
                swap(st[j], st[j - 1]);
            }
        }
        return;
    }
    if (st[n - 1] < st[0]) swap(st[0], st[n - 1]);
    int lt = 1, gt = n - 2, i = 1;
    while (i <= gt) {
        if (st[i] < st[0]) swap(st[i++], st[lt++]);
        else if (st[n - 1] < st[i]) swap(st[i], st[gt--]);
        else i++;
    }
    swap(st[0], st[--lt]);
    swap(st[n - 1], st[++gt]);
    quick_sort_dual(st, st + lt);
    if (st[lt] < st[gt]) quick_sort_dual(st + lt + 1, st + gt);
    quick_sort_dual(st + gt + 1, en);
}

////////////////// COMPARATOR SORT //////////////////

template <class It, class Comparator> void quick_sort_dual(It st, It en, Comparator cmp) {
    int n = en - st;
    if (n <= 1) return;
    if (n <= INSERTION_SORT_CUTOFF) { // insertion sort
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0 && cmp(st[j], st[j - 1]); j--) {
                swap(st[j], st[j - 1]);
            }
        }
        return;
    }
    if (cmp(st[n - 1], st[0])) swap(st[0], st[n - 1]);
    int lt = 1, gt = n - 2, i = 1;
    while (i <= gt) {
        if (cmp(st[i], st[0])) swap(st[i++], st[lt++]);
        else if (cmp(st[n - 1], st[i])) swap(st[i], st[gt--]);
        else i++;
    }
    swap(st[0], st[--lt]);
    swap(st[n - 1], st[++gt]);
    quick_sort_dual(st, st + lt);
    if (cmp(st[lt], st[gt])) quick_sort_dual(st + lt + 1, st + gt);
    quick_sort_dual(st + gt + 1, en);
}
```

2.9.7 Shell Sort

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

// Sorts an array using shell sort
// Time Complexity: O(N^(4/3)) worse case
// Memory Complexity: O(1) additional memory

template <class It> void shell_sort(It st, It en) {
    int n = en - st, h = 1;
    while (h < n * 4 / 9) h = h * 9 / 4 + 1;
    for (; h >= 1; h = h * 4 / 9) for (int i = h; i < n; i++)
        for (int j = i; j >= h && st[j] < st[j - h]; j -= h) swap(st[j], st[j - h]);
}

////////////////// COMPARATOR SORT //////////////////

template <class It, class Comparator> void shell_sort(It st, It en, Comparator cmp) {
    int n = en - st, h = 1;
    while (h < n * 4 / 9) h = h * 9 / 4 + 1;
    for (; h >= 1; h = h * 4 / 9) for (int i = h; i < n; i++)
        for (int j = i; j >= h && cmp(st[j], st[j - h]); j -= h) swap(st[j], st[j - h]);
}
```

2.9.8 Count Inversions

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

// Counts the number of inversions in an array, while also sorting using merge sort
// Time Complexity: O(N log N) worse case
// Memory Complexity: O(N) additional memory

template <class SrcIt, class DstIt>
long long count_inversions(SrcIt src_st, SrcIt src_en, DstIt dst_st, DstIt dst_en) {
    int n = src_en - src_st;
    if (n <= 1) return 0;
```

```
    int mid = (n - 1) / 2; long long ret = 0;
    ret += count_inversions(dst_st, dst_st + mid + 1, src_st, src_st + mid + 1);
    ret += count_inversions(dst_st + mid + 1, dst_en, src_st + mid + 1, src_en);
    for (int i = 0, j = mid + 1, k = 0; k < n; k++) {
        if (i > mid) { dst_st[k] = src_st[j++]; }
        else if (j >= n) { dst_st[k] = src_st[i++]; ret += j - (mid + 1); }
        else if (src_st[j] < src_st[i]) { dst_st[k] = src_st[j++]; ret += mid + 1 - i; }
        else { dst_st[k] = src_st[i++]; ret += j - (mid + 1); }
    }
    return ret;
}

template <class It> long long count_inversions(It st, It en) {
    typedef typename std::iterator_traits<It>::value_type T;
    int n = en - st; T *aux = new T[n];
    for (int i = 0; i < n; i++) aux[i] = st[i];
    long long ret = count_inversions(aux, aux + n, st, en);
    delete[] (aux);
    return ret / 2;
}

////////////////// COMPARATOR SORT //////////////////

template <class SrcIt, class DstIt, class Comparator>
long long count_inversions(SrcIt src_st, SrcIt src_en, DstIt dst_st, DstIt dst_en, Comparator cmp) {
    int n = src_en - src_st;
    if (n <= 1) return 0;
    int mid = (n - 1) / 2; long long ret = 0;
    ret += count_inversions(dst_st, dst_st + mid + 1, src_st, src_st + mid + 1, cmp);
    ret += count_inversions(dst_st + mid + 1, dst_en, src_st + mid + 1, src_en, cmp);
    for (int i = 0, j = mid + 1, k = 0; k < n; k++) {
        if (i > mid) { dst_st[k] = src_st[j++]; }
        else if (j >= n) { dst_st[k] = src_st[i++]; ret += j - (mid + 1); }
        else if (cmp(src_st[j], src_st[i])) { dst_st[k] = src_st[j++]; ret += mid + 1 - i; }
        else { dst_st[k] = src_st[i++]; ret += j - (mid + 1); }
    }
    return ret;
}

template <class It, class Comparator> long long count_inversions(It st, It en, Comparator cmp) {
    typedef typename std::iterator_traits<It>::value_type T;
    int n = en - st; T *aux = new T[n];
    for (int i = 0; i < n; i++) aux[i] = st[i];
    long long ret = count_inversions(aux, aux + n, st, en, cmp);
    delete[] (aux);
    return ret / 2;
}
```

2.10 String

2.10.1 KMP String Search

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

// KMP String Search
// Time Complexity:
//   preCompute: O(len(pat))
//   search, multiSearch: O(len(txt))
// Memory Complexity: O(len(pat)) additional memory
template <const int MAXS> struct KMP {
    int LCP[MAXS]; string pat; vector<int> matches;
    void preCompute(const string &pat) {
        this->pat = pat; LCP[0] = -1;
        for (int i = 0, j = -1; i < pat.length(); i++, j++, LCP[i] = (i != pat.length() && pat[i] == pat[j]) ? LCP[j] : j)
            while (j >= 0 && pat[i] != pat[j]) j = LCP[j];
    }
    int search(const string &txt) { // returns the first index of a match, -1 if none
        for (int i = 0, j = 0, n = txt.length(); i < n; i++, j++) {
            while (j >= 0 && txt[i] != pat[j]) j = LCP[j];
            if (j == pat.length() - 1) return i - j;
        }
        return -1;
    }
    void multiSearch(const string &txt) { // finds all matches
        matches.clear();
        for (int i = 0, j = 0; i < txt.length(); i++, j++) {
            while (j >= 0 && (j == pat.length() || txt[i] != pat[j])) j = LCP[j];
            if (j == pat.length() - 1) matches.push_back(i - j);
        }
    }
};
```

2.10.2 Manacher Palindrome

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

// Computes the longest palindromic substring centered at each half index
// Time Complexity: O(S)
// Memory Complexity: O(S) additional memory
template <const int MAXS> struct ManacherPalindrome {
    int p[MAXS]; string s; char t[MAXS * 2];
    void preProcess(const string &s) {
```

```
        this->s = s; t[0] = '$'; t[s.length() * 2 + 1] = '#'; t[s.length() * 2 + 2] = '@';
        for (int i = 0; i < int(s.length()); i++) { t[2 * i + 1] = '#'; t[2 * i + 2] = s[i]; }
        int center = 0, right = 0; fill(p, p + int(s.length()) * 2 + 3, 0);
        for (int i = 1; i < int(s.length()) * 2 + 2; i++) {
            int mirror = 2 * center - i;
            if (right > i) p[i] = min(right - i, p[mirror]);
            while (t[i + (1 + p[i])] == t[i - (1 + p[i])]) p[i]++;
            if (i + p[i] > right) { center = i; right = i + p[i]; }
        }
    }
    string longestPalindromicSubstring() { // longest palindromic substring
        int length = 0, center = 0;
        for (int i = 1; i < int(s.length()) * 2 + 2; i++) if (p[i] > length) { length = p[i]; center = i; }
        return s.substr((center - 1 - length) / 2, (center - 1 + length) / 2 - (center - 1 - length) / 2);
    }
    string longestPalindromicSubstring(int i) { // longest palindromic substring centered at index i/2
        int length = p[i + 2], center = i + 2;
        return s.substr((center - 1 - length) / 2, (center - 1 + length) / 2 - (center - 1 - length) / 2);
    }
};
```

2.10.3 Z Algorithm

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

// Computes an array z, where z[i] is the length of the longest substring starting at S[i],
// which is also a prefix of S (maximum k such that S[i + j] == S[j] for all 0 <= j < k)
// Time Complexity: O(S)
// Memory Complexity: O(S) additional memory
template <const int MAXS> struct ZAlgorithm {
    int z[MAXS];
    void run(const string &S) {
        int l = 0, r = 0;
        if (int(S.length()) > 0) z[0] = int(S.length());
        for (int i = 1; i < int(S.length()); i++) {
            if (i > r) {
                l = r = i;
                while (r < int(S.length()) && S[r] == S[r - l]) r++;
                r--; z[i] = r - l + 1;
            } else {
                int j = i - l;
                if (z[j] < r - i + 1) z[i] = z[j];
                else {
                    l = i;
                    while (r < int(S.length()) && S[r] == S[r - l]) r++;
                    r--; z[i] = r - l + 1;
                }
            }
        }
    }
};
```

2.10.4 Longest Common Substring

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

// Computes the longest common substring between 2 strings
// Time Complexity: O(len(s1) * len(s2))
// Memory Complexity: O(len(s2))
template <const int MAXS> struct LongestCommonSubstring {
    int dp[2][MAXS]; string substring;
    int solve(const string &s1, const string &s2) {
        int st = 0, len = 0;
        for (int i = 0; i < 2; i++) fill(dp[i], dp[i] + s2.length() + 1, 0);
        for (int i = 1; i <= int(s1.length()); i++) {
            for (int j = 1; j <= int(s2.length()); j++) {
                if (s1[i - 1] == s2[j - 1]) {
                    dp[i % 2][j] = 1 + dp[1 - i % 2][j - 1];
                    if (dp[i % 2][j] > len) { len = dp[i % 2][j]; st = i - dp[i % 2][j]; }
                }
            }
        }
        substring = s1.substr(st, len);
        return len;
    }
};
```

2.10.5 Minimum Edit Distance

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

// Computes the minimum edit distance between 2 strings, allowing for custom penalties
// Time Complexity: O(len(s1) * len(s2))
// Memory Complexity: O(len(s2))
template <const int MAXS> struct MinEditDistance {
    int dp[2][MAXS]; string substring;
    int solve(const string &s1, const string &s2, int repPen = 1, int insPen = 1, int delPen = 1) {
        for (int i = 0; i < 2; i++) fill(dp[i], dp[i] + s2.length() + 1, 0);
        for (int i = 0; i <= int(s1.length()); i++) {
            for (int j = 0; j <= int(s2.length()); j++) {
                if (i == 0) dp[i % 2][j] = j;
```

```
                else if (j == 0) dp[i % 2][j] = i;
                else if (s1[i - 1] == s2[j - 1]) dp[i % 2][j] = dp[1 - i % 2][j - 1];
                else dp[i % 2][j] = min(min(dp[1 - i % 2][j - 1] + repPen, dp[1 - i % 2][j] + insPen), dp[i % 2][j - 1] + delPen);
            }
        }
        return dp[int(s1.length()) % 2][int(s2.length())];
    }
};
```

2.10.6 Suffix Automata

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

// Suffix Automata
// Time Complexity:
//   add: O(S)
//   addLetter: amortized O(1)
//   LCS: O(S)
// Memory Complexity: O(S * ALPHABET_SIZE)
template <const int ALPHABET_SIZE, const int OFFSET> class SuffixAutomata {
    vector<array<int, ALPHABET_SIZE>> to; vector<int> len, link; int last;
    void init() {
        to.clear(); len.clear(); link.clear(); last = 0;
        to.emplace_back(); to.back().fill(-1); len.push_back(0); link.push_back(0);
    }
    void addLetter(char c) {
        c -= OFFSET; int p = last, q;
        if (to[p][c] != -1) {
            q = to[p][c];
            if (len[q] == len[p] + 1) { last = q; return; }
            to.push_back(to[q]); len.push_back(len[p] + 1); link.push_back(link[q]); link[q] = int(link.size()) - 1;
            while (to[p][c] == q) { to[p][c] = int(to.size()) - 1; p = link[p]; }
        } else {
            last = int(to.size()); to.emplace_back(); to.back().fill(-1); len.push_back(len[p] + 1); link.push_back(0);
            while (to[p][c] == -1) { to[p][c] = last; p = link[p]; }
            if (to[p][c] == last) { link[last] = p; return; }
            q = to[p][c];
            if (len[q] == len[p] + 1) { link[last] = q; return; }
            to.push_back(to[q]); len.push_back(len[p] + 1);
            link.push_back(link[q]); link[q] = int(link.size()) - 1; link[last] = int(link.size()) - 1;
            while (to[p][c] == q) { to[p][c] = int(to.size()) - 1; p = link[p]; }
        }
    }
    void add(const string &s) {
        last = 0;
        for (auto &&c : s) addLetter(c);
    }
    // longest common substring
    int LCS(const string &s) {
        int p = 0, curLen = 0, ret = 0;
        for (char c : s) {
            c -= OFFSET;
            while (p != 0 && to[p][c] == -1) { p = link[p]; curLen = len[p]; }
            if (to[p][c] != -1) { p = to[p][c]; curLen++; }
            ret = max(ret, curLen);
        }
        return ret;
    }
};
```