Contents

1 Funciones C++

#include <algorithm> #include <numeric>

Algo	Params	Funcion
sort, stable_sort	f, l	ordena el intervalo
nth_element	f, nth, l	*void* ordena el n-esimo, y particiona el resto
fill, fill_n	f, l / n, elem	*void* llena [f, l) o [f, f+n) con elem
lower_bound, upper_bound	f, l, elem	*it* al primer / ultimo donde se puede insertar elem para que quede ordenada
binary_search	f, l, elem	*bool* esta elem en [f, l)
copy	f, l, resul	hace resul+i=f+i $\forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	*it* encuentra i $\in$[f,l) tq. i=elem, pred(i), i$\in$[f2,l2)
count, count_if	f, l, elem/pred	cuenta elem, pred(i)
search	f, l, f2, l2	busca [f2,l2) $\in$ [f,l)
replace, replace_if	f, l, old / pred, new	cambia old / pred(i) por new
reverse	f, l	da vuelta
partition, stable_partition	f, l, pred	pred(i) ad, !pred(i) atras
min_element, max_element	f, l, [comp]	*it* min, max de [f,l]
lexicographical_compare	f1,l1,f2,l2	*bool* con [f1,l1]¡[f2,l2]
next/prev_permutation	f,l	deja en [f,l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f,l), hace un heap de [f,l)
is_heap	f,l	*bool* es [f,l) un heap
accumulate	f,l,i,[op]	T = $\sum$/oper de [f,l)
inner_product	f1, l1, f2, i	T = i + [f1, l1) . [f2, ...)
partial_sum	f, l, r, [op]	r+i = $\sum$/oper de [f,f+i] $\forall i$ $\in$[f,l)
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.

Specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
%	A % followed by another % will write a single %.	%

2 Compile

2.1 Compile

```
g++-13 nombre.cpp -o nombre (compilar)
./nombre (ejecutar)
g++ -std=c++23 -Wall -Wshadow -g -fsanitize=undefined -fsanitize=address
     -D_GLIBCXX_DEBUG nombre.cpp -o nombre
```

2.2 Template

```
#include <bits/stdc++.h>
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
using namespace std;
#define pb push_back
#define ll long long
#define s second
#define f first
#define MOD 1000000007
#define INF 100000000000000

void solve(){

```

```
}

int main() {
  ios_base::sync_with_stdio(false);  cin.tie(0);  cout.tie(0);
  int t;cin>>t;for(int T=0;T<t;T++)
  solve();
}
```

3 Data Structures

3.1 BIT

```
/*
  Binary Indexed Tree (Fenwick Tree) Fast Implementation
  ---------------------------------
  Indexing: 1-based
  Bounds: [1, MAXN)
  Time Complexity:
    - update(x, val): O(log n) -> adds val to index x
    - get(x): O(log n) -> prefix sum from 1 to x
  Space Complexity: O(n)
*/

const int MAXN = 10000;
int bit[MAXN];

// Add `val` to index `x`
void update(int x, int val) {
  for (; x < MAXN; x += x & -x) {
    bit[x] += val;
  }
}

// Get prefix sum from 1 to `x`
int get(int x) {
  int ans = 0;
  for (; x > 0; x -= x & -x) {
    ans += bit[x];
  }
  return ans;
}
```

3.2 Bitset

```
bitset<3001> b[3001];

//set() Set the bit value at the given index to 1.
//reset() Set the bit value at the given index to 0.
//flip() Toggle the bit value at the given index.
//test() Check if the bit value at the given index is 1 or 0.
//count() Count the number of set bits.
//any() Checks if any bit is set
//all() Check if all bit is set.
//none() Check if no bit is set.
//to_string() Convert the bitset to a string representation.

#pragma GCC target("popcnt")
(int) __builtin_popcount(x);
(int) __builtin_popcountll(x);
__builtin_clz(x); // count leading zeros

// declare bitset
bitset<64> b;
```

3.3 Bit Trie

```
/*
  Bit Trie (Binary Trie for Integers)
  ---------------------------------
  Indexing: 0-based
  Bit Width: [0, MAX_BIT] inclusive (e.g., 31 for 32-bit integers)
  Time Complexity:
    - Insert: O(MAX_BIT)
    - Query: O(MAX_BIT)
  Space Complexity: O(N * MAX_BIT) nodes (in worst case, 1 per bit per
      number)
*/

const int K = 2; // Each node has 2 branches (bit 0 or 1)
const int MAX_BIT = 30; // Max bit position (for 32-bit integers)

struct Vertex {
  int next[K]; // next[0] = child for bit 0, next[1] = child for bit 1

  Vertex() {
    fill(begin(next), end(next), -1); // -1 means no child
  }
```

```
};

vector<Vertex> trie; // Trie nodes

// Inserts a number into the binary trie
void insert(int num) {
  int v = 0; // Start from root
  for (int j = MAX_BIT; j >= 0; --j) {
    int c = (num >> j) & 1; // Extract j-th bit (0 or 1)
    if (trie[v].next[c] == -1) {
      trie[v].next[c] = trie.size();
      trie.emplace_back(); // Add new node
    }
    v = trie[v].next[c]; // Move to next node
  }
}
```

3.4 Disjoint Set Union Bipartite

```
/*
  DSU with Parity - Bipartite Checker
  -----------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive

  Features:
    - Tracks parity (even/odd length) of paths in each component
    - Can be used to detect odd-length cycles (non-bipartite components)
    - Supports dynamic edge additions

  Functions:
    - make_set(v): initializes singleton component
    - find_set(v): returns root of v and its parity relative to root
    - add_edge(a, b): merges two components and checks for bipartite
        violation
    - is_bipartite(v): returns whether component containing v is
        bipartite
*/

const int MAXN = 100005; // Set according to constraints

pair<int, int> parent[MAXN]; // parent[v] = {root, parity_from_root_to_v
    }
int rank[MAXN];              // Union by rank
bool bipartite[MAXN];        // bipartite[root] = true if component is
    bipartite

// Create a new set for node v
void make_set(int v) {
  parent[v] = {v, 0};   // Self-rooted, even parity to self
  rank[v] = 0;
  bipartite[v] = true;
}

// Find the root of v and track parity along the path (0 = even, 1 = odd
    )
pair<int, int> find_set(int v) {
  if (v != parent[v].first) {
    auto [par, parity] = parent[v];
    auto root = find_set(par);
    parent[v] = {root.first, parity ^ root.second}; // Path compression
        with parity update
  }
  return parent[v];
}

// Adds an edge between a and b, merges components and checks for odd
    cycles
void add_edge(int a, int b) {
  auto [ra, pa] = find_set(a); // ra = root of a, pa = parity from root
      to a
  auto [rb, pb] = find_set(b); // rb = root of b, pb = parity from root
      to b

  if (ra == rb) {
    // Same component: edge (a, b) adds a cycle -> check parity
    if ((pa ^ pb) == 0) {
      bipartite[ra] = false; // Found odd-length cycle
    }
  } else {
    // Merge smaller rank under larger
    if (rank[ra] < rank[rb]) swap(ra, rb), swap(pa, pb);

    // Make rb child of ra; update parity of root
    parent[rb] = {ra, pa ^ pb ^ 1};

```

```
    bipartite[ra] &= bipartite[rb]; // Component is only bipartite if both were
    if (rank[ra] == rank[rb]) rank[ra]++;
  }
}

// Check if the component containing v is bipartite
bool is_bipartite(int v) {
  return bipartite[find_set(v).first];
}
```

3.5 Disjoint Set Union

```
/*
  Disjoint Set Union (Union-Find) with Rollback
  ---------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, N-1]

  Features:
    - Path compression + union by size
    - Optional rollback to previous state
    - Supports dynamic connectivity in offline algorithms (e.g., divide & conquer)

  Functions:
    - get(x): find root of x
    - connected(a, b): check if a and b are in same component
    - size(x): size of component containing x
    - unite(x, y): union x and y, returns true if merged
    - time(): current rollback timestamp
    - rollback(t): revert to state at timestamp t
*/

struct DSU {
  vector<int> e;                    // e[x] < 0 -> root; size = -e[x]; e[x] >= 0 -> parent
  vector<pair<int, int>> st;        // rollback stack: stores (index, old value)

  DSU(int N) : e(N, -1) {}

  // Find with path compression
  int get(int x) {
    return e[x] < 0 ? x : get(e[x]);
  }

  // Check if x and y belong to the same component
  bool connected(int a, int b) {
    return get(a) == get(b);
  }

  // Return size of component containing x
  int size(int x) {
    return -e[get(x)];
  }

  // Union by size, returns true if union happened
  bool unite(int x, int y) {
    x = get(x), y = get(y);
    if (x == y) return false;
    if (e[x] > e[y]) swap(x, y); // Ensure x has larger size (more negative)
    st.push_back({x, e[x]});
    st.push_back({y, e[y]});
    e[x] += e[y];
    e[y] = x;
    return true;
  }

  // Return current rollback timestamp
  int time() {
    return (int)st.size();
  }

  // Roll back to previous state at time t
  void rollback(int t) {
    for (int i = time(); i-- > t;) {
      e[st[i].first] = st[i].second;
    }
    st.resize(t);
  }
};
```

3.6 Dynamic Conectivity

```cpp
/*
  Offline Dynamic Connectivity - Segment Tree + Rollback DSU
  ----------------------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1]

  Features:
    - Handles dynamic edge insertions and deletions over time
    - Answers queries of type: "how many connected components at time t ?"
    - All operations are processed offline

  Components:
    - DSU with rollback (stores a stack of previous states)
    - Segment tree over time to store active edge intervals
*/

// Rollbackable Disjoint Set Union (Union-Find)
struct DSU {
  vector<int> e; // e[x] < 0 means x is a root, and size is -e[x]
  vector<pair<int, int>> st; // Stack for rollback (stores changed values)
  int cnt; // Current number of connected components

  DSU() {}
  DSU(int N) : e(N, -1), cnt(N) {}

  // Find root of x with path compression
  int get(int x) {
    return e[x] < 0 ? x : get(e[x]);
  }

  // Check if x and y are connected
  bool connected(int a, int b) {
    return get(a) == get(b);
  }

  // Size of component containing x
  int size(int x) {
    return -e[get(x)];
  }

  // Union two components; record state for rollback
  bool unite(int x, int y) {
    x = get(x), y = get(y);
    if (x == y) return false; // Already connected

    if (e[x] > e[y]) swap(x, y); // Union by size: ensure x is larger

    // Save state for rollback
    st.push_back({x, e[x]});
    st.push_back({y, e[y]});

    e[x] += e[y]; // Merge y into x
    e[y] = x;
    cnt--; // One fewer component
    return true;
  }

  // Undo last union
  void rollback() {
    auto [x1, y1] = st.back(); st.pop_back();
    e[x1] = y1;
    auto [x2, y2] = st.back(); st.pop_back();
    e[x2] = y2;
    cnt++;
  }
};

// Represents a single union operation (on edge u-v)
struct query {
  int v, u;
  bool united;
  query(int _v, int _u) : v(_v), u(_u), united(false) {}
};

// Segment Tree for storing edge intervals [l, r]
struct QueryTree {
  vector<vector<query>> t; // Each node stores queries that are active in that time segment
  DSU dsu;
  int T; // Number of total operations (time steps)

  QueryTree() {}
  QueryTree(int _T, int n) : T(_T) {
    dsu = DSU(n);
```

```
    t.resize(4 * T + 4);
  }

  // Internal segment tree add function
  void add(int v, int l, int r, int ul, int ur, query& q) {
    if (ul > ur) return;
    if (l == ul && r == ur) {
      t[v].push_back(q);
      return;
    }
    int mid = (l + r) / 2;
    add(2 * v, l, mid, ul, min(ur, mid), q);
    add(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
  }

  // Public wrapper: add a query in interval [l, r]
  void add_query(query q, int l, int r) {
    add(1, 0, T - 1, l, r, q);
  }

  // Traverse the segment tree and simulate unions with rollback
  void dfs(int v, int l, int r, vector<int>& ans) {
    // Apply all union operations in this segment node
    for (query& q : t[v]) {
      q.united = dsu.unite(q.v, q.u);
    }

    if (l == r) {
      ans[l] = dsu.cnt; // Save answer for time l
    } else {
      int mid = (l + r) / 2;
      dfs(2 * v, l, mid, ans);
      dfs(2 * v + 1, mid + 1, r, ans);
    }

    // Rollback all operations applied in this node
    for (query& q : t[v]) {
      if (q.united)
        dsu.rollback();
    }
  }
};

```

```
int main() {
  int n, k;
  cin >> n >> k; // n nodes, k operations
  if (k == 0) return 0;
  QueryTree st(k, n);
  map<pair<int, int>, int> mp; // Edge -> start time
  vector<int> ans(k);          // Answers for '?' queries
  vector<int> qmarks;          // Indices of '?' queries
  // Parse all k operations
  for (int i = 0; i < k; i++) {
    char c;
    cin >> c;
    if (c == '?') {
      qmarks.push_back(i); // Save query index
      continue;
    }
    int u, v;
    cin >> u >> v;
    u--; v--;
    if (u > v) swap(u, v); // Normalize edge direction

    if (c == '+') {
      mp[{u, v}] = i; // Mark time edge is added
    } else {
      // Edge removed: store active interval
      st.add_query(query(u, v), mp[{u, v}], i);
      mp[{u, v}] = -1;
    }
  }
  // Any edge still active is added until end of timeline
  for (auto [edge, start] : mp) {
    if (start != -1) {
      st.add_query(query(edge.first, edge.second), start, k - 1);
    }
  }
  // Process the tree to compute all '?'-query answers
  st.dfs(1, 0, k - 1, ans);
  // Output results of all '?'
  for (int x : qmarks) {
    cout << ans[x] << '\n';
  }
}
```

3.7 Fenwick Tree

```cpp
/*
  Fenwick Tree (Binary Indexed Tree)
  ----------------------------------
  Indexing: 0-based
  Bounds: [0, n-1] inclusive
  Time Complexity:
    - add(x, v): O(log n)
    - sum(x): O(log n) -> prefix sum over [0, x)
    - rangeSum(l, r): O(log n) -> sum over [l, r)
    - select(k): O(log n) -> smallest x such that prefix sum >= k (works
        for monotonic cumulative sums)

  Space Complexity: O(n)
*/

template <typename T>
struct Fenwick {
  int n;
  std::vector<T> a;

  Fenwick(int n_ = 0) {
    init(n_);
  }

  // Initialize BIT of size n
  void init(int n_) {
    n = n_;
    a.assign(n, T{});
  }

  // Add value `v` to position `x`
  void add(int x, const T &v) {
    for (int i = x + 1; i <= n; i += i & -i) {
      a[i - 1] = a[i - 1] + v;
    }
  }

  // Compute prefix sum on range [0, x)
  T sum(int x) const {
    T ans{};
    for (int i = x; i > 0; i -= i & -i) {
      ans = ans + a[i - 1];
    }
    return ans;
  }

  // Compute sum over range [l, r)
  T rangeSum(int l, int r) const {
    return sum(r) - sum(l);
  }

  // Find the smallest x such that sum[0, x) > k (if exists), or returns
      n
  int select(const T &k) const {
    int x = 0;
    T cur{};
    for (int i = 1 << std::__lg(n); i; i >>= 1) {
      if (x + i <= n && cur + a[x + i - 1] <= k) {
        cur = cur + a[x + i - 1];
        x += i;
      }
    }
    return x;
  }
};
// Fenwick<int> bit(n);
```

3.8 Fenwick Tree 2D

```cpp
/*
  2D Fenwick Tree (Binary Indexed Tree)
  -------------------------------------
  Indexing: 1-based
  Bounds: [1, n] inclusive
  Time Complexity:
    -update(x, y, v): O(log^2 n)
    -get(x, y): sum of rectangle [1,1] to [x,y]
    -get1(x1, y1, x2, y2): sum over rectangle [x1,y1] to [x2,y2]
  Space Complexity: O(n^2)

  -Can be adapted for rectangular grids by using n, m separately
*/

struct Fenwick2D {
```

```
  vector<vector<ll>> b; // 2D BIT array
  int n;                // Grid size (1-based)

  Fenwick2D(int _n) : b(_n + 5, vector<ll>(_n + 5, 0)), n(_n) {}

  // Add 'val' to cell (x, y)
  void update(int x, int y, int val) {
    for (; x <= n; x += (x & -x)) {
      for (int j = y; j <= n; j += (j & -j)) {
        b[x][j] += val;
      }
    }
  }

  // Get sum of rectangle [(1,1) to (x,y)]
  ll get(int x, int y) {
    ll ans = 0;
    for (; x > 0; x -= x & -x) {
      for (int j = y; j > 0; j -= j & -j) {
        ans += b[x][j];
      }
    }
    return ans;
  }

  // Get sum over subrectangle [(x1,y1) to (x2,y2)]
  ll get1(int x1, int y1, int x2, int y2) {
    return get(x2, y2)-get(x1-1, y2)-get(x2, y1-1) + get(x1-1, y1-1);
  }
};
// Usage example:
Fenwick2D fw(n);
fw.update(3, 4, 5);                 // add 5 to (3, 4)
ll sum = fw.get(3, 4);              // sum from (1,1) to (3,4)
ll range = fw.get1(2, 2, 5, 5);     // sum in rectangle [(2,2)-(5,5)]
```

3.9 Linked List

```
/*
  Linked list
  ------------------------------------
  Time Complexity:
    -insert: O(1)
  Space Complexity: O(n)

  -Can be adapted for rectangular grids by using n, m separately
*/
// === Constructors and Destructor ===
// list()               // Default constructor, creates an empty list.
// list(n, val)         // Creates a list with 'n' elements, all
    initialized to 'val'.
// list(first, last)    // Creates a list from another range (e.g.,
    another list or array).
// list(const list& other) // Copy constructor.
// list(list&& other)   // Move constructor.
// ~list()              // Destructor, destroys the list.

// === Iterators ===
// begin()              // Returns an iterator to the first element of
    the list.
// end()                // Returns an iterator to the position just
    past the last element.
// rbegin()             // Returns a reverse iterator to the last
    element.
// rend()               // Returns a reverse iterator to the position
    just before the first element.
// insert(it, val)      // Inserts 'val' before the position of the
    iterator 'it'.
// erase(it)            // Removes the element at the position of the
    iterator 'it'.
// erase(first, last)   // Removes a range of elements from 'first' to
    'last'.
// push_back(val)       // Adds 'val' to the end of the list.
// push_front(val)      // Adds 'val' to the front of the list.
// pop_back()           // Removes the last element of the list.
// pop_front()          // Removes the first element of the list.
// clear()              // Removes all elements from the list.

// === Capacity ===
// empty()              // Returns true if the list is empty, otherwise
     false.
// size()               // Returns the number of elements in the list.
// max_size()           // Returns the maximum number of elements the
    list can hold.

// === Modifiers ===
```

```
// insert(it, n, val)    // Inserts 'n' copies of 'val' before the position of 'it'.
// insert(it, first, last) // Inserts a range of elements before the position of 'it'.
// remove(val)           // Removes all occurrences of 'val' from the list.
// remove_if(pred)       // Removes all elements that satisfy the predicate 'pred'.
// unique()              // Removes consecutive duplicate elements from the list.
// merge(other)          // Merges two sorted lists into one sorted list.
// merge(other, pred)    // Merges two sorted lists into one sorted list using custom comparison 'pred'.
// splice(it, other)     // Transfers elements from another list 'other' to the list at position 'it'.
// splice(it, other, it2) // Transfers an element from another list 'other' at position 'it2' to the list at position 'it'.
// splice(it, other, first, last) // Transfers a range from another list 'other' from 'first' to 'last' to the list at position 'it'.
// reverse()             // Reverses the order of elements in the list.
// sort()                // Sorts the elements in the list in ascending order.
// sort(comp)            // Sorts the elements in the list using custom comparison 'comp'.
// swap(other)           // Swaps the content of the list with another list 'other'.

// === Element Access ===
// front()               // Returns a reference to the first element in the list.
// back()                // Returns a reference to the last element in the list.

// === Search ===
// find(val)             // Returns an iterator to the first occurrence of 'val'.
// find_if(pred)         // Returns an iterator to the first element satisfying the predicate 'pred'.
// count(val)            // Returns the number of occurrences of 'val' in the list.
// count_if(pred)        // Returns the number of elements satisfying the predicate 'pred'.
// equal_range(val)      // Returns a pair of iterators representing the range of elements equal to 'val'.

// === Other Functions ===
// swap(other)           // Swaps the contents of the list with another list 'other'.
// emplace(it, args...)  // Constructs and inserts an element at the position 'it' using 'args...' (perfect forwarding).
// emplace_front(args...) // Constructs and inserts an element at the front using 'args...' (perfect forwarding).
// emplace_back(args...)  // Constructs and inserts an element at the back using 'args...' (perfect forwarding).
```

3.10 Merge Sort Tree

```
/*
  Merge Sort Tree (Segment Tree of Sorted Arrays)
  -----------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive
  Time Complexity:
    - build(): O(n log n)
    - q(l, r, x): O(log^2 n) -> number of elements <= x in [l, r)
  Space Complexity: O(n log n)

  Features:
    - Supports frequency/count queries: "how many values <= x in range [l, r)?"
    - Static array (no point updates unless rebuilt)
*/

const int MAXN = 100005;        // size of original array
const int MAXT = 2 * MAXN;      // size of segment tree (2n)

vector<int> t[MAXT];    // Segment tree: each node holds sorted vector
int a[MAXN];            // Original array
int n;                  // Size of array

// Build merge sort tree (bottom-up)
void build() {
  // Fill leaves
  for (int i = 0; i < n; i++) {
    t[i + n].push_back(a[i]);
```

```
  }

  // Merge children into parent
  for (int i = n - 1; i > 0; i--) {
    auto &left = t[2 * i], &right = t[2 * i + 1];
    merge(left.begin(), left.end(), right.begin(), right.end(),
        back_inserter(t[i]));
  }
}

// Query how many elements <= `x` in range [l, r)
int q(int l, int r, int x) {
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l & 1) {
      res += upper_bound(t[l].begin(), t[l].end(), x) - t[l].begin();
      l++;
    }
    if (r & 1) {
      r--;
      res += upper_bound(t[r].begin(), t[r].end(), x) - t[r].begin();
    }
  }
  return res;
}
// Read n and array a, then call build()
```

3.11 Minimum Cartesian Tree

```
/*
  Min Cartesian Tree
  ------------------
  Indexing: 0-based
  Time Complexity: O(n)
  Space Complexity: O(n)
  Tree Properties:
    - Binary tree where in-order traversal = original array
    - Tree satisfies min-heap property: parent <= children
    - `par[i]`: parent of node i
    - `sons[i][0]`: left child, `sons[i][1]`: right child
    - `root`: index of root node

  Use cases:
    - RMQ construction
    - LCA over RMQ via Cartesian Tree
*/

struct min_cartesian_tree {
  vector<int> par;           // parent for each node
  vector<vector<int>> sons;  // left and right children
  int root;

  void init(int n, const vector<int> &arr) {
    par.assign(n, -1);
    sons.assign(n, vector<int>(2, -1)); // 0 = left, 1 = right
    stack<int> st;

    for (int i = 0; i < n; i++) {
      int last = -1;

      // Maintain increasing stack -> build min Cartesian Tree
      // Change > to < for max Cartesian Tree
      while (!st.empty() && arr[st.top()] > arr[i]) {
        last = st.top();
        st.pop();
      }

      if (!st.empty()) {
        par[i] = st.top();
        sons[st.top()][1] = i;
      }
      if (last != -1) {
        par[last] = i;
        sons[i][0] = last;
      }

      st.push(i);
    }

    for (int i = 0; i < n; i++) {
      if (par[i] == -1) {
        root = i;
      }
    }
  }
};
```

```cpp
// Example usage:
vector<int> a = {4, 2, 6, 1, 3};
min_cartesian_tree ct;
ct.init(a.size(), a);
cout << "Root index: " << ct.root << '\n';
```

3.12 Multi Ordered Set

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T> using oset = __gnu_pbds::tree<
    T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update
>;

//en main

oset<pair<int,int>> name;
    map<int,int> cuenta;
    function<void(int)> meter = [&] (int val) {
        name.insert({val,++cuenta[val]});
    };
    auto quitar = [&] (int val) {
        name.erase({val,cuenta[val]--});
    };

meter(x);
quitar(y);
multioset.order_of_key({y+1,-1})-multioset.order_of_key({x,0})
```

3.13 Ordered Set

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T> using oset = __gnu_pbds::tree<
    T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update
>;
// order_of_key() primero mayor o igual;
// find_by_order() apuntador al elemento k;
// oset<pair<int,int>> os;
// os.insert({1,2});
// os.insert({2,3});
// os.insert({5,6});
// ll k=os.order_of_key({2,0});
// cout<<k<<endl; // 1
// pair<int,int> p=*os.find_by_order(k);
// cout<<p.f<<" "<<p.s<<endl; // 2 3
// os.erase(p);
// p=*os.find_by_order(k);
// cout<<p.f<<" "<<p.s<<endl; // 5 6


// check if upperbound or lowerbound does what you want
// because they give better time.

// to allow repetitions
#define ordered_set tree<int, null_type,less_equal<int>, rb_tree_tag,
    tree_order_statistics_node_update>

// to not allow repetitions
#define ordered_set tree<int, null_type,less<int>, rb_tree_tag,
    tree_order_statistics_node_update>

//order_of_key(x): number of items are strictly smaller than x

//find_by_order(k) iterator to the kth element
```

3.14 Palindromic Tree

```cpp
/*
  Palindromic Tree (Eertree)
  --------------------------
  Indexing: 0-based
  Time Complexity:
    - extend(i): O(1) amortized
    - calc_occurrences(): O(n)
  Space Complexity: O(n)

  Features:
    - Each node represents a unique palindromic substring
    - Efficient online construction
    - `oc`: how many times this palindrome occurs as suffix
    - `cnt`: number of palindromic suffixes in its subtree
    - `link`: suffix link to longest proper palindromic suffix
```

```
*/

const int N = 3e5 + 9;

struct PalindromicTree {
  struct node {
    int nxt[26];     // transitions by character
    int len;         // length of palindrome
    int st, en;      // start and end indices in string
    int link;        // suffix link
    int cnt = 0;     // number of palindromic suffixes
    int oc = 0;      // occurrences of the palindrome
  };

  string s;
  vector<node> t;
  int sz, last;

  PalindromicTree() {}
  PalindromicTree(const string &_s) {
    s = _s;
    int n = s.size();
    t.clear();
    t.resize(n + 5); // up to n + 2 distinct palindromes
    sz = 2;
    last = 2;
    // Root 1: imaginary (-1 length), simplifies links
    t[1].len = -1;
    t[1].link = 1;
    // Root 2: length 0, link to root 1
    t[2].len = 0;
    t[2].link = 1;
  }

  // Extend tree with s[pos], returns 1 if a new node is created
  int extend(int pos) {
    int cur = last;
    int ch = s[pos] - 'a';
    // Find longest suffix palindrome that can be extended
    while (true) {
      int curlen = t[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) break;
      cur = t[cur].link;
    }

    if (t[cur].nxt[ch]) {
      // Already exists
      last = t[cur].nxt[ch];
      t[last].oc++;
      return 0;
    }
    // Create new node
    sz++;
    last = sz;
    t[sz].oc = 1;
    t[sz].len = t[cur].len + 2;
    t[cur].nxt[ch] = sz;
    t[sz].en = pos;
    t[sz].st = pos - t[sz].len + 1;

    if (t[sz].len == 1) {
      t[sz].link = 2;
      t[sz].cnt = 1;
      return 1;
    }
    // Compute suffix link
    while (true) {
      cur = t[cur].link;
      int curlen = t[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
        t[sz].link = t[cur].nxt[ch];
        break;
      }
    }
    t[sz].cnt = 1 + t[t[sz].link].cnt;
    return 1;
  }
  // Accumulate total occurrences for each palindrome node
  void calc_occurrences() {
    for (int i = sz; i >= 3; i--) {
      t[t[i].link].oc += t[i].oc;
    }
  }
};

int main() {
```

```
  string s;
  cin >> s;
  PalindromicTree t(s);
  for (int i = 0; i < s.size(); i++) {
    t.extend(i);
  }
  t.calc_occurrences();
  long long total = 0;
  for (int i = 3; i <= t.sz; i++) {
    total += t.t[i].oc;
  }
  cout << total << '\n'; // Total palindromic substrings
  return 0;
}
```

3.15 Persistent Array

```
/*
  Persistent Array (via Persistent Segment Tree)
  ----------------------------------------------
  Indexing: 0-based
  Bounds: [0, n-1]
  Time Complexity:
    - point update: O(log n)
    - point query: O(log n)
  Space Complexity: O(log n) per update/version

  Features:
    - Supports point updates with full version history
    - Allows querying any version at any index
*/

struct Node {
  int val;
  Node *l, *r;

  // Leaf node with value
  Node(int x) : val(x), l(nullptr), r(nullptr) {}

  // Internal node with children (value is not used)
  Node(Node *ll, Node *rr) : val(0), l(ll), r(rr) {}
};

int n;
int a[100001];        // Initial array
Node *roots[100001];  // Roots of all versions (0-based)

// Build version 0 from initial array
Node *build(int l = 0, int r = n - 1) {
  if (l == r) return new Node(a[l]);
  int mid = (l + r) / 2;
  return new Node(build(l, mid), build(mid + 1, r));
}

// Create new version with a[pos] = val
Node *update(Node *node, int pos, int val, int l = 0, int r = n - 1) {
  if (l == r) return new Node(val);
  int mid = (l + r) / 2;
  if (pos <= mid)
    return new Node(update(node->l, pos, val, l, mid), node->r);
  else
    return new Node(node->l, update(node->r, pos, val, mid + 1, r));
}

// Query value at position `pos` in a given version (node)
int query(Node *node, int pos, int l = 0, int r = n - 1) {
  if (l == r) return node->val;
  int mid = (l + r) / 2;
  if (pos <= mid) return query(node->l, pos, l, mid);
  else return query(node->r, pos, mid + 1, r);
}

// External helper: get value at index in version
int get_item(int index, int version) {
  return query(roots[version], index);
}

// External helper: make new version based on `prev_version`, updating
    one index
void update_item(int index, int value, int prev_version, int new_version
    ) {
  roots[new_version] = update(roots[prev_version], index, value);
}

// Initializes version 0 from given array
void init_arr(int nn, int *init) {
```

```
  n = nn;
  for (int i = 0; i < n; i++) a[i] = init[i];
  roots[0] = build();
}
```

3.16 Persistent Segment Tree

```
/*
  Persistent Segment Tree (Point Updates, Range Queries)
  ------------------------------------------------------
  Indexing: 1-based
  Bounds: [1, n]
  Time Complexity:
    - Build: O(n)
    - Point update: O(log n) -> returns new version
    - Range query: O(log n)
    - Copy version: O(1)

  Features:
    - Each update creates a new tree version with shared unchanged nodes
    - Supports querying over any version
    - Useful in rollback problems, version history, and functional
        programming
*/

struct Node {
  ll val;     // segment sum
  Node *l, *r;

  // Leaf node
  Node(ll x) : val(x), l(nullptr), r(nullptr) {}

  // Internal node with children
  Node(Node *_l, Node *_r) {
    l = _l;
    r = _r;
    val = 0;
    if (l) val += l->val;
    if (r) val += r->val;
  }

  // Version clone (used when copying tree version directly)
  Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
};

int n, sz = 1;
ll a[200001];          // Input array (1-indexed)
Node *t[200001];       // Roots of different versions (t[version_id])

// Build initial segment tree from array a[1..n]
Node *build(int l = 1, int r = n) {
  if (l == r) return new Node(a[l]);
  int mid = (l + r) / 2;
  return new Node(build(l, mid), build(mid + 1, r));
}

// Update position `pos` with `val` in given `node` version
Node *update(Node *node, int pos, int val, int l = 1, int r = n) {
  if (l == r) return new Node(val); // replace leaf
  int mid = (l + r) / 2;
  if (pos <= mid)
    return new Node(update(node->l, pos, val, l, mid), node->r);
  else
    return new Node(node->l, update(node->r, pos, val, mid + 1, r));
}

// Query sum over range [a, b] in given version
ll query(Node *node, int a, int b, int l = 1, int r = n) {
  if (r < a || l > b) return 0;          // No overlap
  if (l >= a && r <= b) return node->val; // Total overlap
  int mid = (l + r) / 2;
  return query(node->l, a, b, l, mid) + query(node->r, a, b, mid + 1, r)
      ;
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(NULL);

  int q;
  cin >> n >> q;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }

  // Build version 0
  t[0] = build();
```

```
  sz = 1;

  while (q--) {
    int ty;
    cin >> ty;

    if (ty == 1) {
      // Point update: create new version from t[k] with a[pos] = x
      int k, pos, x;
      cin >> k >> pos >> x;
      t[k] = update(t[k], pos, x);

    } else if (ty == 2) {
      // Range query on version k over [l, r]
      int k, l, r;
      cin >> k >> l >> r;
      cout << query(t[k], l, r) << '\n';

    } else if (ty == 3) {
      // Clone version k into new version
      int k;
      cin >> k;
      t[sz++] = new Node(t[k]);
    }
  }
  return 0;
}
```

3.17 Segment Tree

```
/*
  Segment Tree (Iterative, Range Minimum Query)
  ---------------------------------------------
  Indexing: 0-based
  Bounds: [0, n-1] inclusive
  Time Complexity:
    - update(pos, val): O(log n)
    - get(l, r): O(log n) -> query min in range [l, r)
  Space Complexity: O(2n)
*/
struct SegmentTree {
  vector<ll> a; // segment tree array
  int n;        // number of elements in original array

  SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}
  // Update position `pos` to value `val`
  void update(int pos, ll val) {
    pos += n;      // move to leaf
    a[pos] = val;  // set value
    for (pos /= 2; pos > 0; pos /= 2) {
      a[pos] = min(a[2 * pos], a[2 * pos + 1]); // update parent
    }
  }
  // Get minimum value in range [l, r)
  ll get(int l, int r) {
    ll res = 1e18;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) res = min(res, a[l++]); // if l is right child
      if (r & 1) res = min(res, a[--r]); // if r is left child
    }
    return res;
  }
};
```

3.18 Segment Tree 2D

```
/*
  2D Segment Tree (Sum over Rectangles)
  -------------------------------------
  Indexing: 0-based
  Grid Size: n * m
  Time Complexity:
    - build: O(nm log n log m)
    - point update: O(log n log m)
    - range query [x1..x2][y1..y2]: O(log n log m)
  Space Complexity: O(4n x 4m)
*/

const int MAXN = 505;
int n, m;             // grid dimensions
int a[MAXN][MAXN];    // input grid
int t[4 * MAXN][4 * MAXN]; // segment tree

// Build the tree along y-axis (internal to each x-interval)
void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
  if (ly == ry) {
```

```
    if (lx == rx)
      t[vx][vy] = a[lx][ly];
    else
      t[vx][vy] = t[vx * 2][vy] + t[vx * 2 + 1][vy];
  } else {
    int my = (ly + ry) / 2;
    build_y(vx, lx, rx, vy * 2, ly, my);
    build_y(vx, lx, rx, vy * 2 + 1, my + 1, ry);
    t[vx][vy] = t[vx][vy * 2] + t[vx][vy * 2 + 1];
  }
}

// Build the tree along x-axis and call build_y for each
void build_x(int vx, int lx, int rx) {
  if (lx != rx) {
    int mx = (lx + rx) / 2;
    build_x(vx * 2, lx, mx);
    build_x(vx * 2 + 1, mx + 1, rx);
  }
  build_y(vx, lx, rx, 1, 0, m - 1);
}

// Query along y-axis in a fixed x-node
int sum_y(int vx, int vy, int tly, int try_, int ly, int ry) {
  if (ly > ry) return 0;
  if (ly == tly && ry == try_) return t[vx][vy];
  int tmy = (tly + try_) / 2;
  return sum_y(vx, vy * 2, tly, tmy, ly, min(ry, tmy))
       + sum_y(vx, vy * 2 + 1, tmy + 1, try_, max(ly, tmy + 1), ry);
}

// Query sum in rectangle [lx..rx][ly..ry]
int sum_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
  if (lx > rx) return 0;
  if (lx == tlx && trx == rx)
    return sum_y(vx, 1, 0, m - 1, ly, ry);
  int tmx = (tlx + trx) / 2;
  return sum_x(vx * 2, tlx, tmx, lx, min(rx, tmx), ly, ry)
       + sum_x(vx * 2 + 1, tmx + 1, trx, max(lx, tmx + 1), rx, ly, ry);
}

// Update along y-axis for fixed x-node
void update_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int
    y, int new_val) {
  if (ly == ry) {
    if (lx == rx)
      t[vx][vy] = new_val;
    else
      t[vx][vy] = t[vx * 2][vy] + t[vx * 2 + 1][vy];
  } else {
    int my = (ly + ry) / 2;
    if (y <= my)
      update_y(vx, lx, rx, vy * 2, ly, my, x, y, new_val);
    else
      update_y(vx, lx, rx, vy * 2 + 1, my + 1, ry, x, y, new_val);
    t[vx][vy] = t[vx][vy * 2] + t[vx][vy * 2 + 1];
  }
}

// Update point (x, y) to new_val
void update_x(int vx, int lx, int rx, int x, int y, int new_val) {
  if (lx != rx) {
    int mx = (lx + rx) / 2;
    if (x <= mx)
      update_x(vx * 2, lx, mx, x, y, new_val);
    else
      update_x(vx * 2 + 1, mx + 1, rx, x, y, new_val);
  }
  update_y(vx, lx, rx, 1, 0, m - 1, x, y, new_val);
}
```

3.19 Segment Tree Dynamic

```
/*
  Dynamic Segment Tree (Point Add, Range Sum)
  -------------------------------------------
  Indexing: [0, INF) or any large bounded range
  Time Complexity:
    - add(k, x): O(log U)
    - get_sum(l, r): O(log U)
      where U = range size (e.g., 1e9 if implicit bounds)

  Space Complexity: O(nodes visited or created) -> worst O(log U) per
      operation
*/

```

```
struct Vertex {
  int left, right;        // interval [left, right)
  int sum = 0;            // sum of elements in this interval
  Vertex *left_child = nullptr, *right_child = nullptr;

  Vertex(int lb, int rb) {
    left = lb;
    right = rb;
  }

  // Create children lazily only when needed
  void extend() {
    if (!left_child && left + 1 < right) {
      int mid = (left + right) / 2;
      left_child = new Vertex(left, mid);
      right_child = new Vertex(mid, right);
    }
  }

  // Add `x` to position `k`
  void add(int k, int x) {
    extend();
    sum += x;
    if (left_child) {
      if (k < left_child->right)
        left_child->add(k, x);
      else
        right_child->add(k, x);
    }
  }

  // Get sum over interval [lq, rq)
  int get_sum(int lq, int rq) {
    if (lq <= left && right <= rq)
      return sum;
    if (rq <= left || right <= lq)
      return 0;
    extend();
    return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq);
  }
};

Vertex *root = new Vertex(0, 1e9); // Range [0, 1e9)
root->add(5, 10);       // a[5] += 10
root->add(1000, 20);    // a[1000] += 20
cout << root->get_sum(0, 10) << '\n';     // sum of [0, 10) = 10
cout << root->get_sum(0, 2000) << '\n';   // sum of [0, 2000) = 30
```

3.20 Segment Tree Lazy Types

```
struct max_t {
  ll val;
  static const ll null_v = -1LL << 61;
  max_t(): val(0) {}
  max_t(ll v): val(v) {}
  max_t op(max_t& other) {
    return max_t(max(val, other.val));
  }
  max_t lazy_op(max_t& v, int size) {
    return max_t(val + v.val);
  }
};

struct min_t {
  ll val;
  static const ll null_v = 1LL << 61;
  min_t(): val(0) {}
  min_t(ll v): val(v) {}
  min_t op(min_t& other) {
    return min_t(min(val, other.val));
  }
  min_t lazy_op(min_t& v, int size) {
    return min_t(val + v.val);
  }
};

struct sum_t {
  ll val;
  static const ll null_v = 0;
  sum_t(): val(0) {}
  sum_t(ll v): val(v) {}
  sum_t op(sum_t& other) {
    return sum_t(val + other.val);
  }
  sum_t lazy_op(sum_t& v, int size) {
    return sum_t(val + v.val * size);
```

```
  }
};
```

3.21 Segment Tree Lazy

```
/*
  Lazy Segment Tree (Range Update, Range Query)
  ---------------------------------------------
  Indexing: 0-based
  Bounds: [0, n-1]
  Time Complexity:
    - build: O(n)
    - update(l, r, v): O(log n)
    - query(l, r): O(log n)
  Space Complexity: O(4n)
*/

// See SegTreeLazy_types for num_t structs

const num_t num_t::null_v = num_t(0);


template <typename num_t>
struct segtree {
  int n;
  vector<num_t> tree, lazy;

  // Initialize segment tree with array of size s
  void init(int s, long long* arr) {
    n = s;
    tree.assign(4 * n, num_t());
    lazy.assign(4 * n, num_t());
    init(0, 0, n - 1, arr);
  }

  // Build segment tree from array
  num_t init(int i, int l, int r, long long* arr) {
    if (l == r) return tree[i] = num_t(arr[l]);

    int mid = (l + r) / 2;
    num_t left = init(2 * i + 1, l, mid, arr);
    num_t right = init(2 * i + 2, mid + 1, r, arr);
    return tree[i] = left.op(right);
  }

  // Public wrapper: update range [l, r] with value v
  void update(int l, int r, num_t v) {
    if (l > r) return;
    update(0, 0, n - 1, l, r, v);
  }

  // Internal recursive update
  num_t update(int i, int tl, int tr, int ql, int qr, num_t v) {
    eval_lazy(i, tl, tr);

    if (tr < ql || qr < tl) return tree[i]; // no overlap
    if (ql <= tl && tr <= qr) {
      lazy[i].val += v.val;
      eval_lazy(i, tl, tr);
      return tree[i];
    }

    int mid = (tl + tr) / 2;
    num_t a = update(2 * i + 1, tl, mid, ql, qr, v);
    num_t b = update(2 * i + 2, mid + 1, tr, ql, qr, v);
    return tree[i] = a.op(b);
  }

  // Public wrapper: query sum in range [l, r]
  num_t query(int l, int r) {
    if (l > r) return num_t::null_v;
    return query(0, 0, n - 1, l, r);
  }

  // Internal recursive query
  num_t query(int i, int tl, int tr, int ql, int qr) {
    eval_lazy(i, tl, tr);

    if (ql <= tl && tr <= qr) return tree[i]; // total overlap
    if (tr < ql || qr < tl) return num_t::null_v; // no overlap

    int mid = (tl + tr) / 2;
    num_t a = query(2 * i + 1, tl, mid, ql, qr);
    num_t b = query(2 * i + 2, mid + 1, tr, ql, qr);
    return a.op(b);
  }
```

```
82
83    // Push down pending lazy updates to children
84    void eval_lazy(int i, int l, int r) {
85      tree[i] = tree[i].lazy_op(lazy[i], r - l + 1);
86      if (l != r) {
87        lazy[2 * i + 1].val += lazy[i].val;
88        lazy[2 * i + 2].val += lazy[i].val;
89      }
90      lazy[i] = num_t(); // reset lazy at this node
91    }
92  };
```

3.22 Segment Tree Lazy Range Set

```
1   /*
2     Lazy Segment Tree (Range Set + Range Add + Range Sum)
3     -----------------------------------------------------
4     Indexing: 0-based
5     Bounds: [0, N-1]
6
7     Features:
8       - Supports range set (assign value), range add (increment), and
            range sum queries
9       - Properly prioritizes lazy set > lazy add
10  */
11
12  const int maxN = 1e5 + 5;
13  int N, Q;
14  int a[maxN];
15
16  struct node {
17    ll val = 0;      // range sum
18    ll lzAdd = 0;    // pending addition
19    ll lzSet = 0;    // pending set (non-zero means active)
20  };
21
22  node tree[maxN << 2];
23
24  #define lc (p << 1)
25  #define rc ((p << 1) | 1)
26
27  // Update current node based on its children
28  inline void pushup(int p) {
29    tree[p].val = tree[lc].val + tree[rc].val;
30  }
31
32  // Push lazy values down to children
33  void pushdown(int p, int l, int mid, int r) {
34    // Range set overrides any pending add
35    if (tree[p].lzSet != 0) {
36      tree[lc].lzSet = tree[rc].lzSet = tree[p].lzSet;
37      tree[lc].val = (mid - l + 1) * tree[p].lzSet;
38      tree[rc].val = (r - mid) * tree[p].lzSet;
39      tree[lc].lzAdd = tree[rc].lzAdd = 0;
40      tree[p].lzSet = 0;
41    }
42    // Otherwise propagate add
43    else if (tree[p].lzAdd != 0) {
44      if (tree[lc].lzSet == 0) tree[lc].lzAdd += tree[p].lzAdd;
45      else {
46        tree[lc].lzSet += tree[p].lzAdd;
47        tree[lc].lzAdd = 0;
48      }
49      if (tree[rc].lzSet == 0) tree[rc].lzAdd += tree[p].lzAdd;
50      else {
51        tree[rc].lzSet += tree[p].lzAdd;
52        tree[rc].lzAdd = 0;
53      }
54      tree[lc].val += (mid - l + 1) * tree[p].lzAdd;
55      tree[rc].val += (r - mid) * tree[p].lzAdd;
56      tree[p].lzAdd = 0;
57    }
58  }
59
60  // Build tree from array a[0..N-1]
61  void build(int p, int l, int r) {
62    tree[p].lzAdd = tree[p].lzSet = 0;
63    if (l == r) {
64      tree[p].val = a[l];
65      return;
66    }
67    int mid = (l + r) >> 1;
68    build(lc, l, mid);
69    build(rc, mid + 1, r);
70    pushup(p);
71  }
```

```

// Add ‘val‘ to all elements in [a, b]
void add(int p, int l, int r, int a, int b, ll val) {
  if (a > r || b < l) return;
  if (a <= l && r <= b) {
    tree[p].val += (r - l + 1) * val;
    if (tree[p].lzSet == 0) tree[p].lzAdd += val;
    else tree[p].lzSet += val;
    return;
  }
  int mid = (l + r) >> 1;
  pushdown(p, l, mid, r);
  add(lc, l, mid, a, b, val);
  add(rc, mid + 1, r, a, b, val);
  pushup(p);
}

// Set all elements in [a, b] to ‘val‘
void set(int p, int l, int r, int a, int b, ll val) {
  if (a > r || b < l) return;
  if (a <= l && r <= b) {
    tree[p].val = (r - l + 1) * val;
    tree[p].lzAdd = 0;
    tree[p].lzSet = val;
    return;
  }
  int mid = (l + r) >> 1;
  pushdown(p, l, mid, r);
  set(lc, l, mid, a, b, val);
  set(rc, mid + 1, r, a, b, val);
  pushup(p);
}

// Query sum over [a, b]
ll query(int p, int l, int r, int a, int b) {
  if (a > r || b < l) return 0;
  if (a <= l && r <= b) return tree[p].val;
  int mid = (l + r) >> 1;
  pushdown(p, l, mid, r);
  return query(lc, l, mid, a, b) + query(rc, mid + 1, r, a, b);
}
// Example usage
N = 5;
a[0] = 2, a[1] = 4, a[2] = 3, a[3] = 1, a[4] = 5;
build(1, 0, N - 1);
set(1, 0, N - 1, 1, 3, 7);     // a[1..3] = 7
add(1, 0, N - 1, 2, 4, 2);     // a[2..4] += 2
cout << query(1, 0, N - 1, 0, 4) << '\n'; // total sum
```

3.23 Segment Tree Max Subarray Sum

```
const ll inf=1e18;

struct Node {
  ll maxi, l_max, r_max, sum;
  Node(ll _maxi, ll _l_max, ll _r_max, ll _sum){
    maxi=_maxi;
    l_max=_l_max;
    r_max=_r_max;
    sum=_sum;
  }
  Node operator+(Node b) {
    return {max(max(maxi, b.maxi), r_max + b.l_max),
      max(l_max, sum + b.l_max), max(b.r_max, r_max + b.sum),
      sum + b.sum};
  }

};

struct SegmentTreeMaxSubSum{
  int n;
  vector<Node> t;
  SegmentTreeMaxSubSum(int _n) : n(_n), t(2 * _n, Node(-inf, -inf, -inf,
       -inf)) {}
  void update(int pos, ll val) {
    t[pos += n] = Node(val, val, val, val);
    for (pos>>=1; pos ; pos >>= 1) {
      t[pos] = t[2*pos]+t[2*pos+1];
    }
  }
  Node query(int l, int r) {
    Node node_l = Node(-inf, -inf, -inf, -inf);
    Node node_r = Node(-inf, -inf, -inf, -inf);
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        node_l=node_l+t[l++];
```

```
      }
      if (r & 1) {
        node_r=t[--r]+node_r;
      }
    }
    return node_l+node_r;
  }
};
```

3.24 Segment Tree Range Update

```
/*
  Segment Tree (Range Min Update, Point Query)
  -------------------------------------------
  Indexing: 0-based
  Bounds: [0, n-1]
  Time Complexity:
    - update(l, r, val): O(log n) -> applies min(val) over [l, r)
    - get(pos): O(log n) -> minimum affecting position pos
  Space Complexity: O(2n)
*/

struct SegmentTree {
  vector<ll> a; // a[i] = min value affecting segment i
  int n;

  SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}

  // Get the effective minimum at position `pos`
  ll get(int pos) {
    ll res = 1e18;
    for (pos += n; pos > 0; pos >>= 1) {
      res = min(res, a[pos]);
    }
    return res;
  }

  // Apply min(val) to all positions in [l, r)
  void update(int l, int r, ll val) {
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        a[l] = min(a[l], val);
        l++;
      }
      if (r & 1) {
        --r;
        a[r] = min(a[r], val);
      }
    }
  }
};
```

3.25 Segment Tree Struct Types

```
// Sum segment tree
struct sum_t{
  ll val;
  static const long long null_v = 0;

  sum_t(): val(null_v) {}
  sum_t(long long v): val(v) {}

  sum_t operator + (const sum_t &a) const {
    sum_t ans;
    ans.val = val + a.val;
    return ans;
  }
};
// Min segment tree
struct min_t{
  ll val;
  static const long long null_v = 1e18;

  min_t(): val(null_v) {}
  min_t(long long v): val(v) {}

  min_t operator + (const min_t &a) const {
    min_t ans;
    ans.val = min(val, a.val);
    return ans;
  }
};
// Max segment tree
struct max_t{
  ll val;
  static const long long null_v = -1e18;
```

```
33
34    max_t(): val(null_v) {}
35    max_t(long long v): val(v) {}
36
37    max_t operator + (const max_t &a) const {
38      max_t ans;
39      ans.val = max(val, a.val);
40      return ans;
41    }
42  };
43  // GCD segment tree
44  struct gcd_t{
45    ll val;
46    static const long long null_v = 0;
47
48    gcd_t(): val(null_v) {}
49    gcd_t(long long v): val(v) {}
50
51    gcd_t operator + (const gcd_t &a) const {
52      gcd_t ans;
53      ans.val = gcd(val, a.val);
54      return ans;
55    }
56  };
```

3.26 Segment Tree Struct

```
1   // works as a 0-indexed segtree (not lazy)
2   template <typename num_t>
3   struct segtree
4   {
5     int n, k;
6     vector<num_t> tree;
7     void init(int s, vector<ll> arr)
8     {
9       n = s;
10      k = 0;
11      while ((1 << k) < n)
12        k++;
13      tree = vector<num_t>(2 * (1 << k) + 1);
14      for (int i = 0; i < n; i++)
15      {
16        tree[(1 << k) + i] = arr[i];
17      }
18      for (int i = (1 << k) - 1; i > 0; i--)
19      {
20        tree[i] = tree[i * 2] + tree[i * 2 + 1];
21      }
22    }
23    void update(int a, ll b)
24    {
25      a += (1 << k);
26      tree[a] = b;
27      for (a /= 2; a >= 1; a /= 2)
28      {
29        tree[a] = tree[a * 2] + tree[a * 2 + 1];
30      }
31    }
32    num_t find(int a, int b)
33    {
34      a += (1 << k);
35      b += (1 << k);
36      num_t s;
37      while (a <= b)
38      {
39        if (a % 2 == 1)
40          s = s + tree[a++];
41        if (b % 2 == 0)
42          s = s + tree[b--];
43        a /= 2;
44        b /= 2;
45      }
46      return s;
47    }
48  };
```

3.27 Segment Tree Walk

```
1   /*
2     Segment Tree Walk - Find First Position >= val
3     ----------------------------------------------
4     Indexing: 0-based
5     Bounds: [0, n-1]
6     Time Complexity:
7       - build: O(n)
8       - update(pos, val): O(log n)
```

```
    - get(L, R): O(log n) -> min value in [L, R]
    - query(L, R, val): O(log n) -> find first index in [L, R] where a[i] >= val

  Features:
    - Stores original value array in segment tree form
    - Maps original indices to tree positions for fast updates
    - Allows efficient walk to find constrained elements (e.g. lower bound >= val)
*/

struct SegmentTreeWalk {
  vector<ll> a;           // segment tree values
  vector<int> final_pos;  // maps index i to position in tree (leaf)
  int n;

  SegmentTreeWalk(int _n) : a(4 * _n, 1e18), final_pos(_n), n(_n) {}

  // Build segment tree from array `vals[0..n-1]`, start with node=1, l=0, r=n-1
  void build(int l, int r, int node, const vector<ll> &vals) {
    if (l == r) {
      final_pos[l] = node;
      a[node] = vals[l];
    } else {
      int mid = (l + r) / 2;
      build(l, mid, node * 2, vals);
      build(mid + 1, r, node * 2 + 1, vals);
      a[node] = min(a[node * 2], a[node * 2 + 1]);
    }
  }

  // Update value at original index `pos` to `val`
  void update(int pos, ll val) {
    pos = final_pos[pos]; // leaf position
    a[pos] = val;
    for (pos /= 2; pos > 0; pos /= 2)
      a[pos] = min(a[pos * 2], a[pos * 2 + 1]);
  }

  // Get min value in [L, R], with current node interval [l, r] and root `node`
  ll get(int l, int r, int L, int R, int node) {
    if (L > R) return 1e18;
    if (l == L && r == R) return a[node];
    int mid = (l + r) / 2;
    return min(
      get(l, mid, L, min(R, mid), node * 2),
      get(mid + 1, r, max(L, mid + 1), R, node * 2 + 1)
    );
  }

  // Find first position in [L, R] with a[i] >= val, starting from node interval [l, r]
  pair<ll, ll> query(int l, int r, int L, int R, int node, int val) {
    if (l > R || r < L) return {-1, 0};             // out of query bounds
    if (a[node] < val) return {-1, 0};              // all values < val
    if (l == r) return {a[node], l};                // leaf node that satisfies

    int mid = (l + r) / 2;
    auto left = query(l, mid, L, R, node * 2, val);
    if (left.first != -1) return left;
    return query(mid + 1, r, L, R, node * 2 + 1, val);
  }
};
// Example usage:
int n = 8;
vector<ll> vals = {4, 2, 7, 1, 9, 5, 6, 3};
SegmentTreeWalk st(n);
st.build(0, n - 1, 1, vals);
```

3.28 Sparse Table

```
/*
  Sparse Table (Range Minimum Query)
  ----------------------------------
  Indexing: 0-based
  Bounds: [0, n-1]
  Time Complexity:
    - Build: O(n log n)
    - Query: O(1)
  Space Complexity: O(n log n)

  Features:
```

```
    - Immutable RMQ (no updates)
    - Works for idempotent operations like min, max, gcd
*/

const int MAXN = 100005;
const int K = 30; // floor(log2(MAXN))
int lg[MAXN + 1]; // log base 2 of each i
int st[K + 1][MAXN]; // st[k][i] = min in [i, i + 2^k - 1]

vector<int> a; // input array
int n;

// Returns min in range [L, R]
int mini(int L, int R) {
  int len = R - L + 1;
  int i = lg[len];
  return min(st[i][L], st[i][R - (1 << i) + 1]);
}

int main() {
  cin >> n;
  a.resize(n);
  for (int i = 0; i < n; i++) cin >> a[i];
  // Precompute binary logs
  lg[1] = 0;
  for (int i = 2; i <= n; i++) {
    lg[i] = lg[i / 2] + 1;
  }
  // Initialize 2^0 intervals
  for (int i = 0; i < n; i++) {
    st[0][i] = a[i];
  }
  // Build sparse table
  for (int k = 1; k <= K; k++) {
    for (int i = 0; i + (1 << k) <= n; i++) {
      st[k][i] = min(st[k - 1][i], st[k - 1][i + (1 << (k - 1))]);
    }
  }
  // Example usage
  int q; cin >> q;
  while (q--) {
    int l, r;
    cin >> l >> r;
    cout << mini(l, r) << '\n';
  }
  return 0;
}
```

3.29 Sparse Table 2D

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

const int N = 505, LG = 10;

int st[N][N][LG][LG];
int a[N][N], lg2[N];

int yo(int x1, int y1, int x2, int y2) {
  x2++;
  y2++;
  int a = lg2[x2 - x1], b = lg2[y2 - y1];
  return max(
         max(st[x1][y1][a][b], st[x2 - (1 << a)][y1][a][b]),
         max(st[x1][y2 - (1 << b)][a][b], st[x2 - (1 << a)][y2 - (1 << b
             )][a][b])
       );
}

void build(int n, int m) { // 0 indexed
  for (int i = 2; i < N; i++) lg2[i] = lg2[i >> 1] + 1;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      st[i][j][0][0] = a[i][j];
    }
  }
  for (int a = 0; a < LG; a++) {
    for (int b = 0; b < LG; b++) {
      if (a + b == 0) continue;
      for (int i = 0; i + (1 << a) <= n; i++) {
        for (int j = 0; j + (1 << b) <= m; j++) {
          if (!a) {
            st[i][j][a][b] = max(st[i][j][a][b - 1], st[i][j + (1 << (b
                - 1))][a][b - 1]);
          } else {
            st[i][j][a][b] = max(st[i][j][a - 1][b], st[i + (1 << (a -
```

```
                1))][j][a - 1][b]);
          }
        }
      }
    }
  }
}

string s[N];
int l[N][N], u[N][N];

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  int n, m;
  cin >> n >> m;
  for (int i = 0; i < n; i++) {
    cin >> s[i];
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (!j) l[i][j] = 1;
      else l[i][j] = 1 + (s[i][j - 1] <= s[i][j] ? l[i][j - 1] : 0);
    }
  }
  for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
      if (!i) u[i][j] = 1;
      else u[i][j] = 1 + (s[i - 1][j] <= s[i][j] ? u[i - 1][j] : 0);
    }
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      int nw = 1, mnx = u[i][j], mny = l[i][j];
      for (int len = 1; len <= min(i, j); len++) {
        mnx = min(mnx, u[i][j - len]);
        mny = min(mny, l[i - len][j]);
        if (min(mnx, mny) >= len + 1) nw++;
        else break;
      }
      a[i][j] = nw;
    }
  }
  build(n, m);
  int q;
  cin >> q;
  while (q--) {
    int x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;
    x1--, y1--;
    x2--;
    y2--;
    int l = 1, r = min(x2 - x1 + 1, y2 - y1 + 1), ans = 0;
    while (l <= r) {
      int mid = l + r >> 1;
      if (yo(x1 + mid - 1, y1 + mid - 1, x2, y2) >= mid) ans = mid, l =
          mid + 1;
      else r = mid - 1;
    }
    cout << ans << '\n';
  }
  return 0;
}
// https://www.codechef.com/problems/CENS20B
```

3.30 Square Root Decomposition

```
/*
  Sqrt Decomposition (String Block Cut and Move)
  ----------------------------------------------
  Operation:
    - Supports moving substrings using block cut logic
    - Rebuilds when too many blocks (for performance)

  Indexing: 0-based
  String Bounds: [0, n)
  Time Complexity:
    - cut(a, b): O(sqrt(n))
    - rebuildDecomp(): O(n)
  When to rebuild: after too many block splits

  Use case: performing multiple cut/paste operations efficiently on
      large strings
*/

```

```cpp
const int MAXI = 350; // = sqrt(n), for n up to 1e5

int n, numBlocks;
string s;

struct Block {
  int l, r; // indices into string s
  int sz() const { return r - l; }
};

Block blocks[2 * MAXI];     // current block array
Block newBlocks[2 * MAXI];  // used temporarily during cutting

// Rebuilds the entire decomposition into 1 block (or balanced ones)
void rebuildDecomp() {
  string newS = s;
  int k = 0;
  // Flatten string using current block structure
  for (int i = 0; i < numBlocks; i++) {
    for (int j = blocks[i].l; j < blocks[i].r; j++) {
      newS[k++] = s[j];
    }
  }
  // Reset to one big block
  numBlocks = 1;
  blocks[0] = {0, n};
  s = newS;
}

// Cut [a, b) into a separate region and reorder it to the end
void cut(int a, int b) {
  int pos = 0, curBlock = 0;
  // Pass 1: Split blocks to isolate [a, b)
  for (int i = 0; i < numBlocks; i++) {
    Block B = blocks[i];
    bool containsA = (pos < a && pos + B.sz() > a);
    bool containsB = (pos < b && pos + B.sz() > b);
    int cutA = B.l + a - pos;
    int cutB = B.l + b - pos;

    if (containsA && containsB) {
      newBlocks[curBlock++] = {B.l, cutA};
      newBlocks[curBlock++] = {cutA, cutB};
      newBlocks[curBlock++] = {cutB, B.r};
    } else if (containsA) {
      newBlocks[curBlock++] = {B.l, cutA};
      newBlocks[curBlock++] = {cutA, B.r};
    } else if (containsB) {
      newBlocks[curBlock++] = {B.l, cutB};
      newBlocks[curBlock++] = {cutB, B.r};
    } else {
      newBlocks[curBlock++] = B;
    }

    pos += B.sz();
  }

  // Pass 2: Reorder - move [a, b) to the end
  pos = 0;
  numBlocks = 0;

  // First add all blocks not in [a, b)
  for (int i = 0; i < curBlock; i++) {
    if (pos < a || pos >= b)
      blocks[numBlocks++] = newBlocks[i];
    pos += newBlocks[i].sz();
  }

  // Then add blocks in [a, b)
  pos = 0;
  for (int i = 0; i < curBlock; i++) {
    if (pos >= a && pos < b)
      blocks[numBlocks++] = newBlocks[i];
    pos += newBlocks[i].sz();
  }
}
// Example usage
int main() {
  cin >> s;
  n = s.size();
  numBlocks = 1;
  blocks[0] = {0, n};

  int q; cin >> q;
  while (q--) {
    int a, b;
```

```
    cin >> a >> b;
    cut(a, b); // move [a, b) to the end

    if (numBlocks > MAXI) rebuildDecomp();
  }

  rebuildDecomp(); // flatten before output
  cout << s << '\n';
}
```

3.31 Treap

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }

template<class F> void each(Node* n, F f) {
  if (n) { each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
    auto pa = split(n->l, k);
    n->l = pa.second;
    n->recalc();
    return {pa.first, n};
  } else {
    auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
    n->r = pa.first;
    n->recalc();
    return {n, pa.second};
  }
}

Node* merge(Node* l, Node* r) {
  if (!l) return r;
  if (!r) return l;
  if (l->y > r->y) {
    l->r = merge(l->r, r);
    l->recalc();
    return l;
  } else {
    r->l = merge(l, r->l);
    r->recalc();
    return r;
  }
}

Node* ins(Node* t, Node* n, int pos) {
  auto pa = split(t, pos);
  return merge(merge(pa.first, n), pa.second);
}

// Example application: move the range [l, r) to index k
void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
  if (k <= l) t = merge(ins(a, b, k), c);
  else t = merge(a, ins(c, b, k - r));
}

// Usage
// create treap
// Node* name=nullptr;
// insert element
// name=ins(name, new Node(val), pos);
// Node* x = new Node(val);
// name = ins(name, x, pos);
// merge two treaps (name before x)
// name=merge(name, x);
// split treap (this will split treap in two treaps,
// first with elements [0, pos) and second with elements [pos, n))
// pa will be pair of two treaps
// auto pa = split(name, pos);
// move range [l, r) to index k
// move(name, l, r, k);
// iterate over treap
// each(name, [&](int val) {
//     cout << val << ' ';
```

```
// });
```

3.32 Treap 2

```
typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};

int cnt (pitem it) {
    return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void push (pitem it) {
    if (it && it->rev) {
        it->rev = false;
        swap (it->l, it->r);
        if (it->l)  it->l->rev ^= true;
        if (it->r)  it->r->rev ^= true;
    }
}

void merge (pitem & t, pitem l, pitem r) {
    push (l);
    push (r);
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)
        split (t->l, l, t->l, key, add),  r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}

void reverse (pitem t, int l, int r) {
    pitem t1, t2, t3;
    split (t, t1, t2, l);
    split (t2, t2, t3, r-l+1);
    t2->rev ^= true;
    merge (t, t1, t2);
    merge (t, t, t3);
}

void output (pitem t) {
    if (!t)  return;
    push (t);
    output (t->l);
    printf ("%d ", t->value);
    output (t->r);
}
```

3.33 Treap With Inversion

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  bool rev = 0;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
  void push();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
void Node::push() {
  if (rev) {
    rev = 0;
```

```
    swap(l, r);
    if (l) l->rev ^= 1;
    if (r) r->rev ^= 1;
  }
}

template<class F> void each(Node* n, F f) {
  if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  n->push();
  if (cnt(n->l) >= k) {
    auto pa = split(n->l, k);
    n->l = pa.second;
    n->recalc();
    return {pa.first, n};
  } else {
    auto pa = split(n->r, k - cnt(n->l) - 1);
    n->r = pa.first;
    n->recalc();
    return {n, pa.second};
  }
}

Node* merge(Node* l, Node* r) {
  if (!l) return r;
  if (!r) return l;
  l->push();
  r->push();
  if (l->y > r->y) {
    l->r = merge(l->r, r);
    l->recalc();
    return l;
  } else {
    r->l = merge(l, r->l);
    r->recalc();
    return r;
  }
}

Node* ins(Node* t, Node* n, int pos) {
  auto pa = split(t, pos);
  return merge(merge(pa.first, n), pa.second);
}

// Example application: reverse the range [l, r]
void reverse(Node*& t, int l, int r) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l);
  tie(b,c) = split(b, r - l + 1);
  b->rev ^= 1;
  t = merge(merge(a, b), c);
}

void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l);
  tie(b,c) = split(b, r - l);
  if (k <= l) t = merge(ins(a, b, k), c);
  else t = merge(a, ins(c, b, k - r));
}
```

4 Dynamic Programming

4.1 CHT Deque

```
/*
  Convex Hull Trick (CHT) - Min Query with Increasing Slopes
  ----------------------------------------------------------
  Indexing: 1-based for `a`, `dp`, `s`
  Bounds:
    - i from 1 to m // number of elements in the array
    - j from 1 to p // number of transitions
  Time Complexity: O(m * p)
  Requires:
    - Lines inserted in increasing slope order for min query
    - Queries made with increasing x values

  dp[i][j] = min over k < i of { dp[k][j-1] + a[i] * (i - k) + s[i] - s[
      k+1] }
  dp[i][j] = min over k < i of { dp[k][j-1] + cost(k, i) }

  We reformulate:
    y = m * x + c
```

```
    line: m = -k - 1, c = dp[k][j-1] - s[k+1]
    eval: m * a[i] + c + a[i] * i + s[i]
*/

struct Line {
  ll a, b; // y = ax + b
  Line(ll A, ll B) : a(A), b(B) {}
  ll eval(ll x) const {
    return a * x + b;
  }
  // Returns intersection x-coordinate with another line
  double intersect(const Line& other) const {
    return (double)(other.b - b) / (a - other.a);
  }
};

// this finds the minimum and slope in increasing
// Deques for each dp stage
deque<Line> cht[p+1];
// Fill dp
cht[0].push_back(Line(-1, -s[1])); // base case
for (int i = 1; i <= m; i++) {
  for (int j = p; j >= 1; j--) {
    if (j > i) continue;
    // Maintain front of deque to find minimum
    while (cht[j - 1].size() >= 2 && cht[j - 1][1].eval(a[i]) <= cht[j -
        1][0].eval(a[i])) {
      cht[j - 1].pop_front();
    }
    // Evaluate best line
    dp[i][j] = cht[j - 1].front().eval(a[i]) + a[i] * i + s[i];
    // Create new line for current i
    Line curr(-i - 1, dp[i][j] - s[i + 1]);
    // Maintain convexity: remove worse lines from back
    while (cht[j].size() >= 2) {
      Line& l1 = cht[j][cht[j].size() - 2];
      Line& l2 = cht[j].back();
      if (curr.intersect(l1) <= l2.intersect(l1)) {
        cht[j].pop_back();
      } else break;
    }
    cht[j].push_back(curr);
  }
}
```

4.2 Digit DP

```
/*
  Digit Dynamic Programming (Digit DP)
  ------------------------------------
  Goal: Count numbers in range [0, x] that do not have two adjacent
      equal digits.
  State:
    - pos: current digit position
    - last: digit placed at previous position (0 to 9)
    - f: tight flag (0 = must match prefix of x, 1 = already below x)
    - z: leading zero flag (1 = still in leading zero zone)
  Notes:
    - Solve up to x using `solve(x)`
    - Can be modified to count palindromes, digits divisible by 3, etc.
*/

vector<int> num;
ll DP[20][20][2][2]; // pos, last digit, f (tight), z (leading zero)

ll g(int pos, int last, int f, int z) {
  if (pos == num.size()) return 1; // reached end, valid number
  if (DP[pos][last][f][z] != -1) return DP[pos][last][f][z];
  ll res = 0;
  int limit = f ? 9 : num[pos]; // upper digit bound based on tight flag
  for (int dgt = 0; dgt <= limit; dgt++) {
    // Skip if digit equals last (unless it's a leading zero)
    if (dgt == last && !(dgt == 0 && z == 1)) continue;
    int nf = f;
    if (!f && dgt < limit) nf = 1;
    if (z && dgt == 0) res += g(pos + 1, dgt, nf, 1); // still leading
        zeros
    else res += g(pos + 1, dgt, nf, 0); // now in significant digits
  }
  return DP[pos][last][f][z] = res;
}

ll solve(ll x) {
  if (x == -1) return 0;
  num.clear();
  while (x > 0) {
```

```
    num.push_back(x % 10);
    x /= 10;
  }
  reverse(num.begin(), num.end());
  memset(DP, -1, sizeof(DP));
  return g(0, 0, 0, 1);
}
```

4.3 Divide and Conquer DP

```
/*
  Divide and Conquer DP Optimization
  ----------------------------------
  Problem:
    - dp[i][j] = min over k <= j of { dp[i-1][k] + C(k, j) }
    - C(k, j) must satisfy the quadrangle inequality:
        - C(a, c) + C(b, d) <= C(a, d) + C(b, c) for a <= b <= c <= d
    - or monotonicity of opt[i][j] <= opt[i][j+1]

  Indexing: 0-based
  Time Complexity: O(m * n * log n)
  Space Complexity: O(n)

  dp_cur[j]: current dp[i][j] layer
  dp_before[j]: previous dp[i-1][j] layer
*/

int n, m;
vector<ll> dp_before, dp_cur;
ll C(int i, int j); // Cost function defined by user

// Recursively compute dp_cur[l..r] with optimal k in [optl, optr]
void compute(int l, int r, int optl, int optr) {
  if (l > r) return;
  int mid = (l + r) / 2;
  pair<ll, int> best = {LLONG_MAX, -1};
  for (int k = optl; k <= min(mid, optr); k++) {
    ll val = (k > 0 ? dp_before[k - 1] : 0) + C(k, mid);
    if (val < best.first) best = {val, k};
  }
  dp_cur[mid] = best.first;
  int opt = best.second;
  compute(l, mid - 1, optl, opt);
  compute(mid + 1, r, opt, optr);
}

// Entry point: computes dp[m-1][n-1]
ll solve() {
  dp_before.assign(n, 0);
  dp_cur.assign(n, 0);
  for (int i = 0; i < n; i++) {
    dp_before[i] = C(0, i);
  }
  for (int i = 1; i < m; i++) {
    compute(0, n - 1, 0, n - 1);
    dp_before = dp_cur;
  }
  return dp_before[n - 1];
}
```

4.4 Edit Distance

```
/*
  Given strings s and t, compute the minimum number of operations
  (insert, delete, substitute) to convert s into t.
  Indexing: 0-based (strings), DP is 1-based with offset
  dp[i][j] = cost to convert s[0..i-1] into t[0..j-1]
  Time Complexity: O(n * m)
  Transitions:
    - insert: dp[i][j-1] + 1
    - delete: dp[i-1][j] + 1
    - replace/match: dp[i-1][j-1] + (s[i-1] != t[j-1])
*/
const int MAXN = 5005;
int dp[MAXN][MAXN];

string s, t; cin >> s >> t;
int n = s.length(), m = t.length();
// Initialize all to a large number
for (int i = 0; i <= n; i++) {
  fill(dp[i], dp[i] + m + 1, 1e9);
}
dp[0][0] = 0;
for (int i = 0; i <= n; i++) {
  for (int j = 0; j <= m; j++) {
    if (j) { // insert
```

```
      dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
    }
    if (i) { // delete
      dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
    }
    if (i && j) { // replace or match
      int cost = (s[i - 1] != t[j - 1]) ? 1 : 0;
      dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + cost);
    }
  }
}
```

4.5 Knuth's Algorithm

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

const int N = 1010;
using ll = long long;
/*
Knuths optimization works for optimization over sub arrays
for which optimal middle point depends monotonously on the end points.
Let mid[l,r] be the first middle point for (l,r) sub array which gives
    optimal result.
It can be proven that mid[l,r-1] <= mid[l,r] <= mid[l+1,r]
- this means monotonicity of mid by l and r.
Applying this optimization reduces time complexity from O(k^3) to O(k^2)
because with fixed s (sub array length) we have m_right(l) = mid[l+1][r]
     = m_left(l+1).
That's why nested l and m loops require not more than 2k iterations
    overall.
*/

int n, k;
int a[N], mid[N][N];
ll res[N][N];
ll solve() {
  for (int s = 0; s <= k; s++) {          // s - length of the subarray
    for (int l = 0; l + s <= k; l++) {  // l - left point
      int r = l + s;                    // r - right point
      if (s < 2) {
        res[l][r] = 0;              // base case- nothing to break
        mid[l][r] = l;              // mid is equal to left border
        continue;
      }
      int mleft = mid[l][r - 1];
      int mright = mid[l + 1][r];
      res[l][r] = 2e18;
      for (int m = mleft; m <= mright; m++) {    // iterating for m in
          the bounds only
        ll tmp = res[l][m] + res[m][r] + (a[r] - a[l]);
        if (res[l][r] > tmp) {                  // relax current solution
          res[l][r] = tmp;
          mid[l][r] = m;
        }
      }
    }
  }
  ll ans = res[0][k];
  return ans;
}
int main() {
  int i, j, m;
  while(cin >> n >> k) {
    for(i = 1; i <= k; i++) cin >> a[i];
    a[0] = 0;
    a[k + 1] = n;
    k++;
    cout << solve() << endl;
  }
  return 0;
}
// https://vjudge.net/problem/ZOJ-2860
```

4.6 LCS

```
string s, t; cin >> s >> t;
int n=s.length(), m=t.length();
int dp[n+1][m+1];
memset(dp, 0, sizeof(dp));
for(int i=1;i<=n;i++){
    for(int j=1;j<=m;j++){
        dp[i][j]=max(dp[i-1][j], dp[i][j-1]);
        if(s[i-1]==t[j-1]){
            dp[i][j]=dp[i-1][j-1]+1;
        }
```

```
    }
}
int i=n, j=m;
string ans="";
while(i && j){
    if(s[i-1]==t[j-1]){
        ans+=s[i-1];
        i--; j--;
    }
    else if(dp[i][j-1]>=dp[i-1][j]){
        j--;
    }
    else{
        i--;
    }
}
reverse(all(ans));
cout << ans << endl;

// For two permutations one can create new array that will map each element from the first permutation to the second.
// For each element a[i] in the first permutatio, you find which j is a[i] == b[j].
// After creating this new array, run LIS (Longest Increasing subsequence).
```

4.7 Line Container

```
/*
  Line Container (Dynamic Convex Hull Trick)
  ------------------------------------------
  Supports:
    - Adding lines: y = k * x + m
    - Querying maximum y at given x
  Indexing: arbitrary, supports any x
  Time Complexity:
    - add(): amortized O(log n)
    - query(x): O(log n)
  Space Complexity: O(n)
  For min queries: negate slopes and intercepts on insert and result on query
  Structure:
    - Stores lines in slope-sorted order (k)
    - Each line keeps its intersection point 'p' with the next line
    - Binary search on 'p' to answer queries
*/
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }  // Sort by slope
  bool operator<(ll x) const { return p < x; }             // Query comparator
};

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // Floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }
  // Update intersection point x->p with y
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, false;
    if (x->k == y->k)
      x->p = (x->m > y->m ? inf : -inf); // higher line wins
    else
      x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  // Add new line: y = k * x + m
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    // Remove dominated lines after y
    while (isect(y, z)) z = erase(z);
    // Remove dominated lines before y
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    // Further cleanup to preserve order
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  // Query max y at given x
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
```

```
  }
};
// Example usage:
LineContainer cht;
cht.add(3, 5);      // y = 3x + 5
cht.add(2, 7);      // y = 2x + 7
cout << cht.query(4) << '\n'; // max y at x = 4
```

4.8 Longest Increasing Subsequence

```
/*
  Longest Increasing Subsequence + (Recover Sequence) O(n log n)
  ----------------------------------------------
  If no recovery is needed, use dp[] only.
  dp.size() gives the length of LIS.
  For non-decreasing use upper_bound instead of lower_bound.
*/
vector<int> dp;    // smallest tail values of LIS length i+1
vector<int> dp_index;    // index in original array
vector<int> parent(n, -1);    // parent[i] = index of previous element in LIS
vector<int> last_pos(n + 1);  // last_pos[len] = index in v[] ending LIS of length len
for (int i = 0; i < n; i++) {
  auto it = lower_bound(dp.begin(), dp.end(), v[i]);
  int len = it - dp.begin();
  if (it == dp.end()) {
    dp.push_back(v[i]);
    dp_index.push_back(i); // Ignore if no recovery
  } else {
    *it = v[i];
    dp_index[len] = i; // Ignore if no recovery
  }
  if (len > 0) parent[i] = dp_index[len - 1]; // Ignore if no recovery
}
// Reconstruct LIS
vector<int> lis;
int pos = dp_index.back();
while (pos != -1) {
  lis.push_back(v[pos]);
  pos = parent[pos];
}
reverse(lis.begin(), lis.end());
```

5 Flow

5.1 Dinic

```
// Si en el grafo todos los vertices distintos
// de s y t cumplen que solo tienen una arista
// de entrada o una de salida la y dicha arista
// tiene capacidad 1 entonces la complejidad es
// O(E sqrt(v))

// si todas las aristas tinen capacidad 1
// el algoritmo tiene complejidad O(E sqrt(E))

// to find min cut run bfs from source and find all vertices that can be reached
// edges between vertices that can be reached and the ones that cant are the min cut

struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
```

```
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap - edges[id].
                flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].
                flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

5.2 Hopcroft-Karp

```
// maximum matching in bipartite graph
vector<int> match, dist;
vector<vector<int>> g;
int n, m, k;
bool bfs()
{
  queue<int> q;
  // The alternating path starts with unmatched nodes
  for (int node = 1; node <= n; node++)
  {
    if (!match[node])
    {
      q.push(node);
      dist[node] = 0;
    }
    else
    {
      dist[node] = INF;
    }
  }

  dist[0] = INF;

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

```
    int node = q.front();
    q.pop();
    if (dist[node] >= dist[0])
    {
      continue;
    }
    for (int son : g[node])
    {
      // If the match of son is matched
      if (dist[match[son]] == INF)
      {
        dist[match[son]] = dist[node] + 1;
        q.push(match[son]);
      }
    }
  }
  // Returns true if an alternating path has been found
  return dist[0] != INF;
}

// Returns true if an augmenting path has been found starting from vertex node
bool dfs(int node)
{
  if (node == 0)
  {
    return true;
  }
  for (int son : g[node])
  {
    if (dist[match[son]] == dist[node] + 1 && dfs(match[son]))
    {
      match[node] = son;
      match[son] = node;
      return true;
    }
  }
  dist[node] = INF;
  return false;
}

int hopcroft_karp()
{
  int cnt = 0;
  // While there is an alternating path
  while (bfs())
  {
    for (int node = 1; node <= n; node++)
    {
      // If node is unmatched but we can match it using an augmenting path
      if (!match[node] && dfs(node))
      {
        cnt++;
      }
    }
  }
  return cnt;
}
// usage
// n numero de puntos en la izquierda
// m numero de puntos en la derecha
// las aristas se guardan en g
// los puntos estan 1 indexados
// el punto 1 de m es el punto n+1 de g
// hopcroft_karp() devuelve el tamano del maximo matching
// match contiene el match de cada punto
// si match de i es 0, entonces i no esta matcheado
//
// https://judge.yosupo.jp/submission/247277
```

5.3 Hungarian

```
#define forn(i,n) for(int i=0;i<int(n);++i)
#define forsn(i,s,n) for(int i=s;i<int(n);++i)
#define forall(i,c) for(typeof(c.begin()) i=c.begin();i!=c.end();++i)
#define DBG(X) cerr << #X << " = " << X << endl;
typedef vector<int> vint;
typedef vector<vint> vvint;

void showmt();

/* begin notebook */

#define MAXN 256
#define INFTO 0x7f7f7f7f
```

```
int n;
int mt[MAXN][MAXN]; // Matriz de costos (X * Y)
int xy[MAXN], yx[MAXN]; // Matching resultante (X->Y, Y->X)

int lx[MAXN], ly[MAXN], slk[MAXN], slkx[MAXN], prv[MAXN];
char S[MAXN], T[MAXN];

void updtree(int x) {
  forn(y, n) if (lx[x] + ly[y] - mt[x][y] < slk[y]) {
    slk[y] = lx[x] + ly[y] - mt[x][y];
    slkx[y] = x;
  }
}
int hungar() {
  forn(i, n) {
    ly[i] = 0;
    lx[i] = *max_element(mt[i], mt[i]+n);
  }
  memset(xy, -1, sizeof(xy));
  memset(yx, -1, sizeof(yx));

  forn(m, n) {
    memset(S, 0, sizeof(S));
    memset(T, 0, sizeof(T));
    memset(prv, -1, sizeof(prv));
    memset(slk, 0x7f, sizeof(slk));
    queue<int> q;
    #define bpone(e, p) { q.push(e); prv[e] = p; S[e] = 1; updtree(e); }
    forn(i, n) if (xy[i] == -1) { bpone(i, -2); break; }

    int x=0, y=-1;
    while (y==-1) {
      while (!q.empty() && y==-1) {
        x = q.front(); q.pop();
        forn(j, n) if (mt[x][j] == lx[x] + ly[j] && !T[j]) {
          if (yx[j] == -1) { y = j; break; }
          T[j] = 1;
          bpone(yx[j], x);
        }
      }
      if (y!=-1) break;
      int dlt = INFTO;
      forn(j, n) if (!T[j]) dlt = min(dlt, slk[j]);
      forn(k, n) {
        if (S[k]) lx[k] -= dlt;
        if (T[k]) ly[k] += dlt;
        if (!T[k]) slk[k] -= dlt;
      }
//      q = queue<int>();
      forn(j, n) if (!T[j] && !slk[j]) {
        if (yx[j] == -1) {
          x = slkx[j]; y = j; break;
        } else {
          T[j] = 1;
          if (!S[yx[j]]) bpone(yx[j], slkx[j]);
        }
      }
    }
    if (y!=-1) {
      for(int p = x; p != -2; p = prv[p]) {
        yx[y] = p;
        int ty = xy[p]; xy[p] = y; y = ty;
      }
    } else break;
  }
  int res = 0;
  forn(i, n) res += mt[i][xy[i]];
  return res;
}
```

5.4 Max Flow Min Cost

```
// dado un acomodo de flujos con costos
// devuelve el costo minimo para un flujo especificado

struct Edge
{
  int from, to, capacity, cost;
  Edge(int _from, int _to, int _capacity, int _cost)
  {
    from = _from;
    to = _to;
    capacity = _capacity;
    cost = _cost;
  }
};
```

```
15
16  vector<vector<int>> adj, cost, capacity;
17
18  const int INF = 1e9;
19
20  void shortest_paths(int n, int v0, vector<int> &d, vector<int> &p)
21  {
22    d.assign(n, INF);
23    d[v0] = 0;
24    vector<bool> inq(n, false);
25    queue<int> q;
26    q.push(v0);
27    p.assign(n, -1);
28
29    while (!q.empty())
30    {
31      int u = q.front();
32      q.pop();
33      inq[u] = false;
34      for (int v : adj[u])
35      {
36        if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v])
37        {
38          d[v] = d[u] + cost[u][v];
39          p[v] = u;
40          if (!inq[v])
41          {
42            inq[v] = true;
43            q.push(v);
44          }
45        }
46      }
47    }
48  }
49
50  int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t)
51  {
52    adj.assign(N, vector<int>());
53    cost.assign(N, vector<int>(N, 0));
54    capacity.assign(N, vector<int>(N, 0));
55    for (Edge e : edges)
56    {
57      adj[e.from].push_back(e.to);
58      adj[e.to].push_back(e.from);
59      cost[e.from][e.to] = e.cost;
60      cost[e.to][e.from] = -e.cost;
61      capacity[e.from][e.to] = e.capacity;
62    }
63
64    int flow = 0;
65    int cost = 0;
66    vector<int> d, p;
67    while (flow < K)
68    {
69      shortest_paths(N, s, d, p);
70      if (d[t] == INF)
71        break;
72
73      // find max flow on that path
74      int f = K - flow;
75      int cur = t;
76      while (cur != s)
77      {
78        f = min(f, capacity[p[cur]][cur]);
79        cur = p[cur];
80      }
81
82      // apply flow
83      flow += f;
84      cost += f * d[t];
85      cur = t;
86      while (cur != s)
87      {
88        capacity[p[cur]][cur] -= f;
89        capacity[cur][p[cur]] += f;
90        cur = p[cur];
91      }
92    }
93
94    if (flow < K)
95      return -1;
96    else
97      return cost;
98  }
```

5.5 Max Flow

```cpp
long long max_flow(vector<vector<int>> adj, vector<vector<long long>>
    capacity,
                   int source, int sink)
{
  int n = adj.size();
  vector<int> parent(n, -1);
  // Find a way from the source to sink on a path with non-negative
      capacities
  auto reachable = [&]() -> bool
  {
    queue<int> q;
    q.push(source);
    while (!q.empty())
    {
      int node = q.front();
      q.pop();
      for (int son : adj[node])
      {
        long long w = capacity[node][son];
        if (w <= 0 || parent[son] != -1)
          continue;
        parent[son] = node;
        q.push(son);
      }
    }
    return parent[sink] != -1;
  };

  long long flow = 0;
  // While there is a way from source to sink with non-negative
      capacities
  while (reachable())
  {
    int node = sink;
    // The minimum capacity on the path from source to sink
    long long curr_flow = LLONG_MAX;
    while (node != source)
    {
      curr_flow = min(curr_flow, capacity[parent[node]][node]);
      node = parent[node];
    }
    node = sink;
    while (node != source)
    {
      // Subtract the capacity from capacity edges
      capacity[parent[node]][node] -= curr_flow;
      // Add the current flow to flow backedges
      capacity[node][parent[node]] += curr_flow;
      node = parent[node];
    }
    flow += curr_flow;
    fill(parent.begin(), parent.end(), -1);
  }

  return flow;
}



//vector<vector<long long>> capacity(n, vector<long long>(n));
//vector<vector<int>> adj(n);
//adj[a].push_back(b);
//adj[b].push_back(a);
//capacity[a][b] += c;
```

5.6 Min Cost Max Flow

```cpp
/**
 *  If costs can be negative, call setpi before maxflow, but note that
     negative cost cycles are not supported.
 *  To obtain the actual flow, look at positive values only
 * Time: $O(F E \log(V))$ where F is max flow. $O(VE)$ for setpi.
 */
#include <bits/stdc++.h>
using namespace std;

#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

```

```cpp
#pragma once

// #include <bits/extc++.h> /// include-line, keep-include

const ll INF = numeric_limits<ll>::max() / 4;

struct MCMF {
  struct edge {
    int from, to, rev;
    ll cap, cost, flow;
  };
  int N;
  vector<vector<edge>> ed;
  vi seen;
  vector<ll> dist, pi;
  vector<edge*> par;

  MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}

  void addEdge(int from, int to, ll cap, ll cost) {
    if (from == to) return;
    ed[from].push_back(edge{ from,to,sz(ed[to]),cap,cost,0 });
    ed[to].push_back(edge{ to,from,sz(ed[from])-1,0,-cost,0 });
  }

  void path(int s) {
    fill(all(seen), 0);
    fill(all(dist), INF);
    dist[s] = 0; ll di;

    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(N);
    q.push({ 0, s });

    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (edge& e : ed[s]) if (!seen[e.to]) {
        ll val = di - pi[e.to] + e.cost;
        if (e.cap - e.flow > 0 && val < dist[e.to]) {
          dist[e.to] = val;
          par[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({ -dist[e.to], e.to });
          else
            q.modify(its[e.to], { -dist[e.to], e.to });
        }
      }
    }
    rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
  }

  pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (edge* x = par[t]; x; x = par[x->from])
        fl = min(fl, x->cap - x->flow);

      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
    }
    rep(i,0,N) for(edge& e : ed[i]) totcost += e.cost * e.flow;
    return {totflow, totcost/2};
  }

  // If some costs can be negative, call this before maxflow:
  void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      rep(i,0,N) if (pi[i] != INF)
        for (edge& e : ed[i]) if (e.cap)
          if ((v = pi[i] + e.cost) < pi[e.to])
            pi[e.to] = v, ch = 1;
    assert(it >= 0); // negative cost cycle
  }
};
```

5.7 Push Relabel

```cpp
const int inf = 1000000000;

```

```
int n;
vector<vector<int>> capacity, flow;
vector<int> height, excess, seen;
queue<int> excess_vertices;

void push(int u, int v) {
  int d = min(excess[u], capacity[u][v] - flow[u][v]);
  flow[u][v] += d;
  flow[v][u] -= d;
  excess[u] -= d;
  excess[v] += d;
  if (d && excess[v] == d)
    excess_vertices.push(v);
}

void relabel(int u) {
  int d = inf;
  for (int i = 0; i < n; i++) {
    if (capacity[u][i] - flow[u][i] > 0)
      d = min(d, height[i]);
  }
  if (d < inf)
    height[u] = d + 1;
}

void discharge(int u) {
  while (excess[u] > 0) {
    if (seen[u] < n) {
      int v = seen[u];
      if (capacity[u][v] - flow[u][v] > 0 && height[u] > height[v])
        push(u, v);
      else
        seen[u]++;
    } else {
      relabel(u);
      seen[u] = 0;
    }
  }
}

int max_flow(int s, int t) {
  height.assign(n, 0);
  height[s] = n;
  flow.assign(n, vector<int>(n, 0));
  excess.assign(n, 0);
  excess[s] = inf;
  for (int i = 0; i < n; i++) {
    if (i != s)
      push(s, i);
  }
  seen.assign(n, 0);

  while (!excess_vertices.empty()) {
    int u = excess_vertices.front();
    excess_vertices.pop();
    if (u != s && u != t)
      discharge(u);
  }

  int max_flow = 0;
  for (int i = 0; i < n; i++)
    max_flow += flow[i][t];
  return max_flow;
}
```

6 Geometry

6.1 Point Struct

```
typedef long long T;
struct pt {
  T x,y;
  pt operator+(pt p) {return {x+p.x, y+p.y};}
  pt operator-(pt p) {return {x-p.x, y-p.y};}
  pt operator*(T d) {return {x*d, y*d};}
  pt operator/(T d) {return {x/d, y/d};}
};

// cross product
// positivo si el segundo esta en sentido antihorario
// 0 si el angulo es 180
// negativo si el segundo esta en sentido horario
T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}

// dot product
// positivo si el angulo entre los vectores es agudo
```

```
// 0 si son perpendiculares
// negativo si el angulo es obtuso
T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}

T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}

T dist(pt a,pt b){
  pt aux=b-a;
  return sqrtl(aux.x*aux.x+aux.y*aux.y);
}
```

6.2 Sort Points

```
// This comparator sorts the points clockwise
// starting from the first quarter

bool getQ(Point a){
  if(a.y!=0){
    if(a.y>0)return 0;
    return 1;
  }
  if(a.x>0)return 0;
  return 1;
}
bool comp(Point a, Point b){
  if(getQ(a)!=getQ(b))return getQ(a)<getQ(b);
  return a*b>0;
}
```

6.3 Point Struct2

```
#include <bits/stdc++.h>
using namespace std;
using ld = long double;
const ld eps = 1e-9, inf = numeric_limits<ld>::max(), pi = acos(-1);
// For use with integers, just set eps=0 and everything remains the same
bool geq(ld a, ld b){return a-b >= -eps;}     //a >= b
bool leq(ld a, ld b){return b-a >= -eps;}     //a <= b
bool ge(ld a, ld b){return a-b > eps;}        //a > b
bool le(ld a, ld b){return b-a > eps;}        //a < b
bool eq(ld a, ld b){return abs(a-b) <= eps;}  //a == b
bool neq(ld a, ld b){return abs(a-b) > eps;}  //a != b

struct point{
  ld x, y;
  point(): x(0), y(0){}
  point(ld x, ld y): x(x), y(y){}

  point operator+(const point & p) const{return point(x + p.x, y + p.y)
      ;}
  point operator-(const point & p) const{return point(x - p.x, y - p.y)
      ;}
  point operator*(const ld & k) const{return point(x * k, y * k);}
  point operator/(const ld & k) const{return point(x / k, y / k);}

  point operator+=(const point & p){*this = *this + p; return *this;}
  point operator-=(const point & p){*this = *this - p; return *this;}
  point operator*=(const ld & p){*this = *this * p; return *this;}
  point operator/=(const ld & p){*this = *this / p; return *this;}

  point rotate(const ld & a) const{return point(x*cos(a) - y*sin(a), x*
      sin(a) + y*cos(a));}
  point perp() const{return point(-y, x);}
  ld ang() const{
    ld a = atan2l(y, x); a += le(a, 0) ? 2*pi : 0; return a;
  }
  ld dot(const point & p) const{return x * p.x + y * p.y;}
  ld cross(const point & p) const{return x * p.y - y * p.x;}
  ld norm() const{return x * x + y * y;}
  ld length() const{return sqrtl(x * x + y * y);}
  point unit() const{return (*this) / length();}

  bool operator==(const point & p) const{return eq(x, p.x) && eq(y, p.y)
      ;}
  bool operator!=(const point & p) const{return !(*this == p);}
  bool operator<(const point & p) const{return le(x, p.x) || (eq(x, p.x)
       && le(y, p.y));}
  bool operator>(const point & p) const{return ge(x, p.x) || (eq(x, p.x)
       && ge(y, p.y));}
  bool half(const point & p) const{return le(p.cross(*this), 0) || (eq(p
      .cross(*this), 0) && le(p.dot(*this), 0));}
};

istream &operator>>(istream &is, point & p){return is >> p.x >> p.y;}
ostream &operator<<(ostream &os, const point & p){return os << "(" << p.
    x << ",␣" << p.y << ")";}

```

```cpp
int sgn(ld x){
  if(ge(x, 0)) return 1;
  if(le(x, 0)) return -1;
  return 0;
}

void polarSort(vector<point> & P, const point & o, const point & v){
  //sort points in P around o, taking the direction of v as first angle
  sort(P.begin(), P.end(), [&](const point & a, const point & b){
    return point((a - o).half(v), 0) < point((b - o).half(v), (a - o).
        cross(b - o));
  });
}
```

6.4 Antipodal Pairs

```cpp
vector<pair<int, int>> antipodalPairs(vector<point> & P){
  vector<pair<int, int>> ans;
  int n = P.size(), k = 1;
  auto f = [&](int u, int v, int w){return abs((P[v%n]-P[u%n]).cross(P[w
      %n]-P[u%n]));};
  while(ge(f(n-1, 0, k+1), f(n-1, 0, k))) ++k;
  for(int i = 0, j = k; i <= k && j < n; ++i){
    ans.emplace_back(i, j);
    while(j < n-1 && ge(f(i, i+1, j+1), f(i, i+1, j)))
      ans.emplace_back(i, ++j);
  }
  return ans;
}
```

6.5 Area and Perimeter

```cpp
ld perimeter(vector<point> & P){
  int n = P.size();
  ld ans = 0;
  for(int i = 0; i < n; i++){
    ans += (P[i] - P[(i + 1) % n]).length();
  }
  return ans;
}

ld area(vector<point> & P){
  int n = P.size();
  ld ans = 0;
  for(int i = 0; i < n; i++){
    ans += P[i].cross(P[(i + 1) % n]);
  }
  return abs(ans / 2);
}
```

6.6 Area Union Circles

```cpp
struct circ{
  point c;
  ld r;
  circ() {}
  circ(const point & c, ld r): c(c), r(r) {}
  set<pair<ld, ld>> ranges;

  void disable(ld l, ld r){
    ranges.emplace(l, r);
  }

  auto getActive() const{
    vector<pair<ld, ld>> ans;
    ld maxi = 0;
    for(const auto & dis : ranges){
      ld l, r;
      tie(l, r) = dis;
      if(l > maxi){
        ans.emplace_back(maxi, l);
      }
      maxi = max(maxi, r);
    }
    if(!eq(maxi, 2*pi)){
      ans.emplace_back(maxi, 2*pi);
    }
    return ans;
  }
};

ld areaUnionCircles(const vector<circ> & circs){
  vector<circ> valid;
  for(const circ & curr : circs){
    if(eq(curr.r, 0)) continue;
    circ nuevo = curr;
    for(circ & prev : valid){
```

```
      if(circleInsideCircle(prev.c, prev.r, nuevo.c, nuevo.r)){
        nuevo.disable(0, 2*pi);
      }else if(circleInsideCircle(nuevo.c, nuevo.r, prev.c, prev.r)){
        prev.disable(0, 2*pi);
      }else{
        auto cruce = intersectionCircles(prev.c, prev.r, nuevo.c, nuevo.
            r);
        if(cruce.size() == 2){
          ld a1 = (cruce[0] - prev.c).ang();
          ld a2 = (cruce[1] - prev.c).ang();
          ld b1 = (cruce[1] - nuevo.c).ang();
          ld b2 = (cruce[0] - nuevo.c).ang();
          if(a1 < a2){
            prev.disable(a1, a2);
          }else{
            prev.disable(a1, 2*pi);
            prev.disable(0, a2);
          }
          if(b1 < b2){
            nuevo.disable(b1, b2);
          }else{
            nuevo.disable(b1, 2*pi);
            nuevo.disable(0, b2);
          }
        }
      }
    }
    valid.push_back(nuevo);
  }
  ld ans = 0;
  for(const circ & curr : valid){
    for(const auto & range : curr.getActive()){
      ld l, r;
      tie(l, r) = range;
      ans += curr.r*(curr.c.x * (sin(r) - sin(l)) - curr.c.y * (cos(r) -
           cos(l))) + curr.r*curr.r*(r-l);
    }
  }
  return ans/2;
};
```

6.7 Centroid

```
point centroid(vector<point> & P){
  point num;
  ld den = 0;
  int n = P.size();
  for(int i = 0; i < n; ++i){
    ld cross = P[i].cross(P[(i + 1) % n]);
    num += (P[i] + P[(i + 1) % n]) * cross;
    den += cross;
  }
  return num / (3 * den);
}
```

6.8 Circle Inside Circle

```
int circleInsideCircle(const point & c1, ld r1, const point & c2, ld r2)
    {
  //test if circle 2 is inside circle 1
  //returns "-1" if 2 touches internally 1, "1" if 2 is inside 1, "0" if
       they overlap
  ld l = r1 - r2 - (c1 - c2).length();
  return (ge(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
}
```

6.9 Circle Outside Circle

```
int circleOutsideCircle(const point & c1, ld r1, const point & c2, ld r2
    ){
  //test if circle 2 is outside circle 1
  //returns "-1" if they touch externally, "1" if 2 is outside 1, "0" if
       they overlap
  ld l = (c1 - c2).length() - (r1 + r2);
  return (ge(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
}
```

6.10 Closest Pair of Points

```
bool comp1(const point & a, const point & b){
  return le(a.y, b.y);
}
pair<point, point> closestPairOfPoints(vector<point> P){
  sort(P.begin(), P.end(), comp1);
  set<point> S;
  ld ans = inf;
  point p, q;
```

```
  int pos = 0;
  for(int i = 0; i < P.size(); ++i){
    while(pos < i && geq(P[i].y - P[pos].y, ans)){
      S.erase(P[pos++]);
    }
    auto lower = S.lower_bound({P[i].x - ans - eps, -inf});
    auto upper = S.upper_bound({P[i].x + ans + eps, -inf});
    for(auto it = lower; it != upper; ++it){
      ld d = (P[i] - *it).length();
      if(le(d, ans)){
        ans = d;
        p = P[i];
        q = *it;
      }
    }
    S.insert(P[i]);
  }
  return {p, q};
}
```

6.11 Common Tangents

```
vector<vector<point>> tangents(const point & c1, ld r1, const point & c2
    , ld r2, bool inner){
  //returns a vector of segments or a single point
  if(inner) r2 = -r2;
  point d = c2 - c1;
  ld dr = r1 - r2, d2 = d.norm(), h2 = d2 - dr*dr;
  if(eq(d2, 0) || le(h2, 0)) return {};
  point v = d*dr/d2;
  if(eq(h2, 0)) return {{c1 + v*r1}};
  else{
    point u = d.perp()*sqrt(h2)/d2;
    return {{c1 + (v - u)*r1, c2 + (v - u)*r2}, {c1 + (v + u)*r1, c2 + (
        v + u)*r2}};
  }
}
```

6.12 Convex Hull

```
vector<point> convexHull(vector<point> P){
  sort(P.begin(), P.end());
  vector<point> L, U;
  for(int i = 0; i < P.size(); i++){
    while(L.size() >= 2 && leq((L[L.size() - 2] - P[i]).cross(L[L.size()
        - 1] - P[i]), 0)){
      L.pop_back();
    }
    L.push_back(P[i]);
  }
  for(int i = P.size() - 1; i >= 0; i--){
    while(U.size() >= 2 && leq((U[U.size() - 2] - P[i]).cross(U[U.size()
        - 1] - P[i]), 0)){
      U.pop_back();
    }
    U.push_back(P[i]);
  }
  L.pop_back();
  U.pop_back();
  L.insert(L.end(), U.begin(), U.end());
  return L;
}
```

6.13 Segment Intersection with Ray

```
bool crossesRay(const point & a, const point & b, const point & p){
  return (geq(b.y, p.y) - geq(a.y, p.y)) * sgn((a - p).cross(b - p)) >
      0;
}
```

6.14 Cut Polygon

```
vector<point> cutPolygon(const vector<point> & P, const point & a, const
     point & v){
  //returns the part of the convex polygon P on the left side of line a+
      tv
  int n = P.size();
  vector<point> lhs;
  for(int i = 0; i < n; ++i){
    if(geq(v.cross(P[i] - a), 0)){
      lhs.push_back(P[i]);
    }
    if(intersectLineSegmentInfo(a, v, P[i], P[(i+1)%n]) == 1){
      point p = intersectLines(a, v, P[i], P[(i+1)%n] - P[i]);
      if(p != P[i] && p != P[(i+1)%n]){
        lhs.push_back(p);
      }
    }
```

```
  }
  return lhs;
}
```

6.15 Delaunay Triangulation

```
//Delaunay triangulation in O(n log n)
const point inf_pt(inf, inf);

struct QuadEdge{
  point origin;
  QuadEdge* rot = nullptr;
  QuadEdge* onext = nullptr;
  bool used = false;
  QuadEdge* rev() const{return rot->rot;}
  QuadEdge* lnext() const{return rot->rev()->onext->rot;}
  QuadEdge* oprev() const{return rot->onext->rot;}
  point dest() const{return rev()->origin;}
};

QuadEdge* make_edge(const point & from, const point & to){
  QuadEdge* e1 = new QuadEdge;
  QuadEdge* e2 = new QuadEdge;
  QuadEdge* e3 = new QuadEdge;
  QuadEdge* e4 = new QuadEdge;
  e1->origin = from;
  e2->origin = to;
  e3->origin = e4->origin = inf_pt;
  e1->rot = e3;
  e2->rot = e4;
  e3->rot = e2;
  e4->rot = e1;
  e1->onext = e1;
  e2->onext = e2;
  e3->onext = e4;
  e4->onext = e3;
  return e1;
}

void splice(QuadEdge* a, QuadEdge* b){
  swap(a->onext->rot->onext, b->onext->rot->onext);
  swap(a->onext, b->onext);
}

void delete_edge(QuadEdge* e){
  splice(e, e->oprev());
  splice(e->rev(), e->rev()->oprev());
  delete e->rot;
  delete e->rev()->rot;
  delete e;
  delete e->rev();
}

QuadEdge* connect(QuadEdge* a, QuadEdge* b){
  QuadEdge* e = make_edge(a->dest(), b->origin);
  splice(e, a->lnext());
  splice(e->rev(), b);
  return e;
}

bool left_of(const point & p, QuadEdge* e){
  return ge((e->origin - p).cross(e->dest() - p), 0);
}

bool right_of(const point & p, QuadEdge* e){
  return le((e->origin - p).cross(e->dest() - p), 0);
}

ld det3(ld a1, ld a2, ld a3, ld b1, ld b2, ld b3, ld c1, ld c2, ld c3) {
  return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) + a3 * (b1
      * c2 - c1 * b2);
}

bool in_circle(const point & a, const point & b, const point & c, const
    point & d) {
  ld det = -det3(b.x, b.y, b.norm(), c.x, c.y, c.norm(), d.x, d.y, d.
      norm());
  det += det3(a.x, a.y, a.norm(), c.x, c.y, c.norm(), d.x, d.y, d.norm()
      );
  det -= det3(a.x, a.y, a.norm(), b.x, b.y, b.norm(), d.x, d.y, d.norm()
      );
  det += det3(a.x, a.y, a.norm(), b.x, b.y, b.norm(), c.x, c.y, c.norm()
      );
  return ge(det, 0);
}

```

```
pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<point> & P){
  if(r - l + 1 == 2){
    QuadEdge* res = make_edge(P[l], P[r]);
    return {res, res->rev()};
  }
  if(r - l + 1 == 3){
    QuadEdge *a = make_edge(P[l], P[l + 1]), *b = make_edge(P[l + 1], P[
        r]);
    splice(a->rev(), b);
    int sg = sgn((P[l + 1] - P[l]).cross(P[r] - P[l]));
    if(sg == 0)
      return {a, b->rev()};
    QuadEdge* c = connect(b, a);
    if(sg == 1)
      return {a, b->rev()};
    else
      return {c->rev(), c};
  }
  int mid = (l + r) / 2;
  QuadEdge *ldo, *ldi, *rdo, *rdi;
  tie(ldo, ldi) = build_tr(l, mid, P);
  tie(rdi, rdo) = build_tr(mid + 1, r, P);
  while(true){
    if(left_of(rdi->origin, ldi)){
      ldi = ldi->lnext();
      continue;
    }
    if(right_of(ldi->origin, rdi)){
      rdi = rdi->rev()->onext;
      continue;
    }
    break;
  }
  QuadEdge* basel = connect(rdi->rev(), ldi);
  auto valid = [&basel](QuadEdge* e){return right_of(e->dest(), basel)
      ;};
  if(ldi->origin == ldo->origin)
    ldo = basel->rev();
  if(rdi->origin == rdo->origin)
    rdo = basel;
  while(true){
    QuadEdge* lcand = basel->rev()->onext;
    if(valid(lcand)){
      while(in_circle(basel->dest(), basel->origin, lcand->dest(), lcand
          ->onext->dest())){
        QuadEdge* t = lcand->onext;
        delete_edge(lcand);
        lcand = t;
      }
    }
    QuadEdge* rcand = basel->oprev();
    if(valid(rcand)){
      while(in_circle(basel->dest(), basel->origin, rcand->dest(), rcand
          ->oprev()->dest())){
        QuadEdge* t = rcand->oprev();
        delete_edge(rcand);
        rcand = t;
      }
    }
    if(!valid(lcand) && !valid(rcand))
      break;
    if(!valid(lcand) || (valid(rcand) && in_circle(lcand->dest(), lcand
        ->origin, rcand->origin, rcand->dest())))
      basel = connect(rcand, basel->rev());
    else
      basel = connect(basel->rev(), lcand->rev());
  }
  return {ldo, rdo};
}

vector<tuple<point, point, point>> delaunay(vector<point> & P){
  sort(P.begin(), P.end());
  auto res = build_tr(0, (int)P.size() - 1, P);
  QuadEdge* e = res.first;
  vector<QuadEdge*> edges = {e};
  while(le((e->dest() - e->onext->dest()).cross(e->origin - e->onext->
      dest()), 0))
    e = e->onext;
  auto add = [&P, &e, &edges](){
    QuadEdge* curr = e;
    do{
      curr->used = true;
      P.push_back(curr->origin);
      edges.push_back(curr->rev());
      curr = curr->lnext();
    }while(curr != e);
```

```
  };
  add();
  P.clear();
  int kek = 0;
  while(kek < (int)edges.size())
    if(!(e = edges[kek++])->used)
      add();
  vector<tuple<point, point, point>> ans;
  for(int i = 0; i < (int)P.size(); i += 3){
    ans.emplace_back(P[i], P[i + 1], P[i + 2]);
  }
  return ans;
}
```

6.16 Diameter and Width

```
pair<ld, ld> diameterAndWidth(vector<point> & P){
  int n = P.size(), k = 0;
  auto dot = [&](int a, int b){return (P[(a+1)%n]-P[a]).dot(P[(b+1)%n]-P
      [b]);};
  auto cross = [&](int a, int b){return (P[(a+1)%n]-P[a]).cross(P[(b+1)%
      n]-P[b]);};
  ld diameter = 0;
  ld width = inf;
  while(ge(dot(0, k), 0)) k = (k+1) % n;
  for(int i = 0; i < n; ++i){
    while(ge(cross(i, k), 0)) k = (k+1) % n;
    //pair: (i, k)
    diameter = max(diameter, (P[k] - P[i]).length());
    width = min(width, distancePointLine(P[i], P[(i+1)%n] - P[i], P[k]))
        ;
  }
  return {diameter, width};
}
```

6.17 Distance Between Point and Circle

```
ld distancePointCircle(const point & c, ld r, const point & p){
  //point p, circle with center c and radius r
  return max((ld)0, (p - c).length() - r);
}
```

6.18 Distance Between Point and Line

```
ld distancePointLine(const point & a, const point & v, const point & p){
  //line: a + tv, point p
  return abs(v.cross(p - a)) / v.length();
}
```

6.19 Example of Geometry

```
int main(){
  /*vector<pair<point, point>> centers = {{point(-2, 5), point(-8, -7)},
       {point(14, 4), point(18, 6)}, {point(9, 20), point(9, 28)},
                       {point(21, 20), point(21, 29)}, {point(8, -10),
                           point(14, -10)}, {point(24, -6), point(34, -6)
                           },
                       {point(34, 8), point(36, 9)}, {point(50, 20),
                           point(56, 24.5)}};
  vector<pair<ld, ld>> radii = {{7, 4}, {3, 5}, {4, 4}, {4, 5}, {3, 3},
      {4, 6}, {5, 1}, {10, 2.5}};
  int n = centers.size();
  for(int i = 0; i < n; ++i){
    cout << "\n" << centers[i].first << " " << radii[i].first << " " <<
        centers[i].second << " " << radii[i].second << "\n";
    auto extLines = tangents(centers[i].first, radii[i].first, centers[i
        ].second, radii[i].second, false);
    cout << "Exterior tangents:\n";
    for(auto par : extLines){
      for(auto p : par){
        cout << p << " ";
      }
      cout << "\n";
    }
    auto intLines = tangents(centers[i].first, radii[i].first, centers[i
        ].second, radii[i].second, true);
    cout << "Interior tangents:\n";
    for(auto par : intLines){
      for(auto p : par){
        cout << p << " ";
      }
      cout << "\n";
    }
  }*/

  /*int n;
  cin >> n;
```

```
  vector<point> P(n);
  for(auto & p : P) cin >> p;
  auto triangulation = delaunay(P);
  for(auto triangle : triangulation){
    cout << get<0>(triangle) << " " << get<1>(triangle) << " " << get
        <2>(triangle) << "\n";
  }*/

  /*int n;
  cin >> n;
  vector<point> P(n);
  for(auto & p : P) cin >> p;
  auto ans = smallestEnclosingCircle(P);
  cout << ans.first << " " << ans.second << "\n";*/

  /*vector<point> P;
  srand(time(0));
  for(int i = 0; i < 1000; ++i){
    P.emplace_back(rand() % 1000000000, rand() % 1000000000);
  }
  point o(rand() % 1000000000, rand() % 1000000000), v(rand() %
      1000000000, rand() % 1000000000);
  polarSort(P, o, v);
  auto ang = [&](point p){
    ld th = atan2(p.y, p.x);
    if(th < 0) th += acosl(-1)*2;
    ld t = atan2(v.y, v.x);
    if(t < 0) t += acosl(-1)*2;
    if(th < t) th += acosl(-1)*2;
    return th;
  };
  for(int i = 0; i < P.size()-1; ++i){
    assert(leq(ang(P[i] - o), ang(P[i+1] - o)));
  }*/
  return 0;
}
```

6.20 Half Plane Intersection

```
struct plane{
  point a, v;
  plane(): a(), v(){}
  plane(const point& a, const point& v): a(a), v(v){}

  point intersect(const plane& p) const{
    ld t = (p.a - a).cross(p.v) / v.cross(p.v);
    return a + v*t;
  }

  bool outside(const point& p) const{ // test if point p is strictly
      outside
    return le(v.cross(p - a), 0);
  }

  bool inside(const point& p) const{ // test if point p is inside or in
      the boundary
    return geq(v.cross(p - a), 0);
  }

  bool operator<(const plane& p) const{ // sort by angle
    auto lhs = make_tuple(v.half({1, 0}), ld(0), v.cross(p.a - a));
    auto rhs = make_tuple(p.v.half({1, 0}), v.cross(p.v), ld(0));
    return lhs < rhs;
  }

  bool operator==(const plane& p) const{ // paralell and same directions
      , not really equal
    return eq(v.cross(p.v), 0) && ge(v.dot(p.v), 0);
  }
};

vector<point> halfPlaneIntersection(vector<plane> planes){
  planes.push_back({{0, -inf}, {1, 0}});
  planes.push_back({{inf, 0}, {0, 1}});
  planes.push_back({{0, inf}, {-1, 0}});
  planes.push_back({{-inf, 0}, {0, -1}});
  sort(planes.begin(), planes.end());
  planes.erase(unique(planes.begin(), planes.end()), planes.end());
  deque<plane> ch;
  deque<point> poly;
  for(const plane& p : planes){
    while(ch.size() >= 2 && p.outside(poly.back())) ch.pop_back(), poly.
        pop_back();
    while(ch.size() >= 2 && p.outside(poly.front())) ch.pop_front(),
        poly.pop_front();
    if(p.v.half({1, 0}) && poly.empty()) return {};
```

```
    ch.push_back(p);
    if(ch.size() >= 2) poly.push_back(ch[ch.size()-2].intersect(ch[ch.
        size()-1]));
  }
  while(ch.size() >= 3 && ch.front().outside(poly.back())) ch.pop_back()
      , poly.pop_back();
  while(ch.size() >= 3 && ch.back().outside(poly.front())) ch.pop_front
      (), poly.pop_front();
  poly.push_back(ch.back().intersect(ch.front()));
  return vector<point>(poly.begin(), poly.end());
}

vector<point> halfPlaneIntersectionRandomized(vector<plane> planes){
  point p = planes[0].a;
  int n = planes.size();
  random_shuffle(planes.begin(), planes.end());
  for(int i = 0; i < n; ++i){
    if(planes[i].inside(p)) continue;
    ld lo = -inf, hi = inf;
    for(int j = 0; j < i; ++j){
      ld A = planes[j].v.cross(planes[i].v);
      ld B = planes[j].v.cross(planes[j].a - planes[i].a);
      if(ge(A, 0)){
        lo = max(lo, B/A);
      }else if(le(A, 0)){
        hi = min(hi, B/A);
      }else{
        if(ge(B, 0)) return {};
      }
      if(ge(lo, hi)) return {};
    }
    p = planes[i].a + planes[i].v*lo;
  }
  return {p};
}
```

6.21 Incircle

```
pair<point, ld> getCircle(const point & m, const point & n, const point
    & p){
  //find circle that passes through points p, q, r
  point c = intersectLines((n + m) / 2, (n - m).perp(), (p + n) / 2, (p
      - n).perp());
  ld r = (c - m).length();
  return {c, r};
}
```

6.22 Intersection of Two Circles

```
vector<point> intersectionCircles(const point & c1, ld r1, const point &
     c2, ld r2){
  //circle 1 with center c1 and radius r1
  //circle 2 with center c2 and radius r2
  point d = c2 - c1;
  ld d2 = d.norm();
  if(eq(d2, 0)) return {}; //concentric circles
  ld pd = (d2 + r1*r1 - r2*r2) / 2;
  ld h2 = r1*r1 - pd*pd/d2;
  point p = c1 + d*pd/d2;
  if(eq(h2, 0)) return {p}; //circles touch at one point
  else if(le(h2, 0)) return {}; //circles don't intersect
  else{
    point u = d.perp() * sqrt(h2/d2);
    return {p - u, p + u};
  }
}
```

6.23 Intersection Line and Circle

```
vector<point> intersectLineCircle(const point & a, const point & v,
    const point & c, ld r){
  //line a+tv, circle with center c and radius r
  ld h2 = r*r - v.cross(c - a) * v.cross(c - a) / v.norm();
  point p = a + v * v.dot(c - a) / v.norm();
  if(eq(h2, 0)) return {p}; //line tangent to circle
  else if(le(h2, 0)) return {}; //no intersection
  else{
    point u = v.unit() * sqrt(h2);
    return {p - u, p + u}; //two points of intersection (chord)
  }
}
```

6.24 Intersection Polygon and Circle

```
ld signed_angle(const point & a, const point & b){
  return sgn(a.cross(b)) * acosl(a.dot(b) / (a.length() * b.length()));
}
```

```

ld intersectPolygonCircle(const vector<point> & P, const point & c, ld r
    ){
  //Gets the area of the intersection of the polygon with the circle
  int n = P.size();
  ld ans = 0;
  for(int i = 0; i < n; ++i){
    point p = P[i], q = P[(i+1)%n];
    bool p_inside = (pointInCircle(c, r, p) != 0);
    bool q_inside = (pointInCircle(c, r, q) != 0);
    if(p_inside && q_inside){
      ans += (p - c).cross(q - c);
    }else if(p_inside && !q_inside){
      point s1 = intersectSegmentCircle(p, q, c, r)[0];
      point s2 = intersectSegmentCircle(c, q, c, r)[0];
      ans += (p - c).cross(s1 - c) + r*r * signed_angle(s1 - c, s2 - c);
    }else if(!p_inside && q_inside){
      point s1 = intersectSegmentCircle(c, p, c, r)[0];
      point s2 = intersectSegmentCircle(p, q, c, r)[0];
      ans += (s2 - c).cross(q - c) + r*r * signed_angle(s1 - c, s2 - c);
    }else{
      auto info = intersectSegmentCircle(p, q, c, r);
      if(info.size() <= 1){
        ans += r*r * signed_angle(p - c, q - c);
      }else{
        point s2 = info[0], s3 = info[1];
        point s1 = intersectSegmentCircle(c, p, c, r)[0];
        point s4 = intersectSegmentCircle(c, q, c, r)[0];
        ans += (s2 - c).cross(s3 - c) + r*r * (signed_angle(s1 - c, s2 -
             c) + signed_angle(s3 - c, s4 - c));
      }
    }
  }
  return abs(ans)/2;
}
```

6.25 Intersection Segment and Circle

```
vector<point> intersectSegmentCircle(const point & a, const point & b,
    const point & c, ld r){
  //segment ab, circle with center c and radius r
  vector<point> P = intersectLineCircle(a, b - a, c, r), ans;
  for(const point & p : P){
    if(pointInSegment(a, b, p)) ans.push_back(p);
  }
  return ans;
}
```

6.26 Line Intersection

```
int intersectLinesInfo(const point & a1, const point & v1, const point &
     a2, const point & v2){
  //lines a1+tv1 and a2+tv2
  ld det = v1.cross(v2);
  if(eq(det, 0)){
    if(eq((a2 - a1).cross(v1), 0)){
      return -1; //infinity points
    }else{
      return 0; //no points
    }
  }else{
    return 1; //single point
  }
}

point intersectLines(const point & a1, const point & v1, const point &
    a2, const point & v2){
  //lines a1+tv1, a2+tv2
  //assuming that they intersect
  ld det = v1.cross(v2);
  return a1 + v1 * ((a2 - a1).cross(v2) / det);
}
```

6.27 Minkowski Sum

```
vector<point> minkowskiSum(vector<point> A, vector<point> B){
  int na = (int)A.size(), nb = (int)B.size();
  if(A.empty() || B.empty()) return {};

  rotate(A.begin(), min_element(A.begin(), A.end()), A.end());
  rotate(B.begin(), min_element(B.begin(), B.end()), B.end());

  int pa = 0, pb = 0;
  vector<point> M;

  while(pa < na && pb < nb){
    M.push_back(A[pa] + B[pb]);
```

```
    ld x = (A[(pa + 1) % na] - A[pa]).cross(B[(pb + 1) % nb] - B[pb]);
    if(leq(x, 0)) pb++;
    if(geq(x, 0)) pa++;
  }

  while(pa < na) M.push_back(A[pa++] + B[0]);
  while(pb < nb) M.push_back(B[pb++] + A[0]);

  return M;
}
```

6.28 Point in Circle

```
int pointInCircle(const point & c, ld r, const point & p){
  //test if point p is inside the circle with center c and radius r
  //returns "0" if it's outside, "-1" if it's in the perimeter, "1" if
      it's inside
  ld l = (p - c).length() - r;
  return (le(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
}
```

6.29 Point in Convex Hull

```
//point in convex polygon in O(log n)
//make sure that P is convex and in ccw
//before the queries, do the preprocess on P:
// rotate(P.begin(), min_element(P.begin(), P.end()), P.end());
// int right = max_element(P.begin(), P.end()) - P.begin();
//returns 0 if p is outside, 1 if p is inside, -1 if p is in the
    perimeter
int pointInConvexPolygon(const vector<point> & P, const point & p, int
    right){
  if(p < P[0] || P[right] < p) return 0;
  int orientation = sgn((P[right] - P[0]).cross(p - P[0]));
  if(orientation == 0){
    if(p == P[0] || p == P[right]) return -1;
    return (right == 1 || right + 1 == P.size()) ? -1 : 1;
  }else if(orientation < 0){
    auto r = lower_bound(P.begin() + 1, P.begin() + right, p);
    int det = sgn((p - r[-1]).cross(r[0] - r[-1])) - 1;
    if(det == -2) det = 1;
    return det;
  }else{
    auto l = upper_bound(P.rbegin(), P.rend() - right - 1, p);
    int det = sgn((p - l[0]).cross((l == P.rbegin() ? P[0] : l[-1]) - l
        [0])) - 1;
    if(det == -2) det = 1;
    return det;
  }
}
```

6.30 Point in Line

```
bool pointInLine(const point & a, const point & v, const point & p){
  //line a+tv, point p
  return eq((p - a).cross(v), 0);
}
```

6.31 Point in Perimeter

```
bool pointInPerimeter(const vector<point> & P, const point & p){
  int n = P.size();
  for(int i = 0; i < n; i++){
    if(pointInSegment(P[i], P[(i + 1) % n], p)){
      return true;
    }
  }
  return false;
}
```

6.32 Point in Polygon

```
int pointInPolygon(const vector<point> & P, const point & p){
  if(pointInPerimeter(P, p)){
    return -1; //point in the perimeter
  }
  int n = P.size();
  int rays = 0;
  for(int i = 0; i < n; i++){
    rays += crossesRay(P[i], P[(i + 1) % n], p);
  }
  return rays & 1; //0: point outside, 1: point inside
}
```

6.33 Point in Segment

```
bool pointInSegment(const point & a, const point & b, const point & p){
  //segment ab, point p
```

```
  return pointInLine(a, b - a, p) && leq((a - p).dot(b - p), 0);
}
```

6.34 Points Tangency

```
pair<point, point> pointsOfTangency(const point & c, ld r, const point & p){
  //point p (outside the circle), circle with center c and radius r
  point v = (p - c).unit() * r;
  ld d2 = (p - c).norm(), d = sqrt(d2);
  point v1 = v * (r / d), v2 = v.perp() * (sqrt(d2 - r*r) / d);
  return {c + v1 - v2, c + v1 + v2};
}
```

6.35 Projection Point Circle

```
point projectionPointCircle(const point & c, ld r, const point & p){
  //point p (outside the circle), circle with center c and radius r
  return c + (p - c).unit() * r;
}
```

6.36 Segment Intersection

```
int intersectLineSegmentInfo(const point & a, const point & v, const point & c, const point & d){
  //line a+tv, segment cd
  point v2 = d - c;
  ld det = v.cross(v2);
  if(eq(det, 0)){
    if(eq((c - a).cross(v), 0)){
      return -1; //infinity points
    }else{
      return 0; //no point
    }
  }else{
    return sgn(v.cross(c - a)) != sgn(v.cross(d - a)); //1: single point, 0: no point
  }
}

int intersectSegmentsInfo(const point & a, const point & b, const point & c, const point & d){
  //segment ab, segment cd
  point v1 = b - a, v2 = d - c;
  int t = sgn(v1.cross(c - a)), u = sgn(v1.cross(d - a));
  if(t == u){
    if(t == 0){
      if(pointInSegment(a, b, c) || pointInSegment(a, b, d) ||
          pointInSegment(c, d, a) || pointInSegment(c, d, b)){
        return -1; //infinity points
      }else{
        return 0; //no point
      }
    }else{
      return 0; //no point
    }
  }else{
    return sgn(v2.cross(a - c)) != sgn(v2.cross(b - c)); //1: single point, 0: no point
  }
}
```

6.37 Smallest Enclosing Circle

```
pair<point, ld> mec2(vector<point> & S, const point & a, const point & b, int n){
  ld hi = inf, lo = -hi;
  for(int i = 0; i < n; ++i){
    ld si = (b - a).cross(S[i] - a);
    if(eq(si, 0)) continue;
    point m = getCircle(a, b, S[i]).first;
    ld cr = (b - a).cross(m - a);
    if(le(si, 0)) hi = min(hi, cr);
    else lo = max(lo, cr);
  }
  ld v = (ge(lo, 0) ? lo : le(hi, 0) ? hi : 0);
  point c = (a + b) / 2 + (b - a).perp() * v / (b - a).norm();
  return {c, (a - c).norm()};
}

pair<point, ld> mec(vector<point> & S, const point & a, int n){
  random_shuffle(S.begin(), S.begin() + n);
  point b = S[0], c = (a + b) / 2;
  ld r = (a - c).norm();
  for(int i = 1; i < n; ++i){
    if(ge((S[i] - c).norm(), r)){
      tie(c, r) = (n == S.size() ? mec(S, S[i], i) : mec2(S, a, S[i], i)
```

```
          );
    }
  }
  return {c, r};
}

pair<point, ld> smallestEnclosingCircle(vector<point> S){
  assert(!S.empty());
  auto r = mec(S, S[0], S.size());
  return {r.first, sqrt(r.second)};
}
```

6.38 Smallest Enclosing Rectangle

```
pair<ld, ld> smallestEnclosingRectangle(vector<point> & P){
  int n = P.size();
  auto dot = [&](int a, int b){return (P[(a+1)%n]-P[a]).dot(P[(b+1)%n]-P
      [b]);};
  auto cross = [&](int a, int b){return (P[(a+1)%n]-P[a]).cross(P[(b+1)%
      n]-P[b]);};
  ld perimeter = inf, area = inf;
  for(int i = 0, j = 0, k = 0, m = 0; i < n; ++i){
    while(ge(dot(i, j), 0)) j = (j+1) % n;
    if(!i) k = j;
    while(ge(cross(i, k), 0)) k = (k+1) % n;
    if(!i) m = k;
    while(le(dot(i, m), 0)) m = (m+1) % n;
    //pairs: (i, k) , (j, m)
    point v = P[(i+1)%n] - P[i];
    ld h = distancePointLine(P[i], v, P[k]);
    ld w = distancePointLine(P[j], v.perp(), P[m]);
    perimeter = min(perimeter, 2 * (h + w));
    area = min(area, h * w);
  }
  return {area, perimeter};
}
```

6.39 Vantage Point Tree

```
struct vantage_point_tree{
  struct node
  {
    point p;
    ld th;
    node *l, *r;
  }*root;

  vector<pair<ld, point>> aux;

  vantage_point_tree(vector<point> &ps){
    for(int i = 0; i < ps.size(); ++i)
      aux.push_back({ 0, ps[i] });
    root = build(0, ps.size());
  }

  node *build(int l, int r){
    if(l == r)
      return 0;
    swap(aux[l], aux[l + rand() % (r - l)]);
    point p = aux[l++].second;
    if(l == r)
      return new node({ p });
    for(int i = l; i < r; ++i)
      aux[i].first = (p - aux[i].second).dot(p - aux[i].second);
    int m = (l + r) / 2;
    nth_element(aux.begin() + l, aux.begin() + m, aux.begin() + r);
    return new node({ p, sqrt(aux[m].first), build(l, m), build(m, r) })
        ;
  }

  priority_queue<pair<ld, node*>> que;

  void k_nn(node *t, point p, int k){
    if(!t)
      return;
    ld d = (p - t->p).length();
    if(que.size() < k)
      que.push({ d, t });
    else if(ge(que.top().first, d)){
      que.pop();
      que.push({ d, t });
    }
    if(!t->l && !t->r)
      return;
    if(le(d, t->th)){
      k_nn(t->l, p, k);
      if(leq(t->th - d, que.top().first))
```

```
        k_nn(t->r, p, k);
    }else{
      k_nn(t->r, p, k);
      if(leq(d - t->th, que.top().first))
        k_nn(t->l, p, k);
    }
  }

  vector<point> k_nn(point p, int k){
    k_nn(root, p, k);
    vector<point> ans;
    for(; !que.empty(); que.pop())
      ans.push_back(que.top().second->p);
    reverse(ans.begin(), ans.end());
    return ans;
  }
};
```

7 Graphs

7.1 2Sat

```
struct TwoSatSolver {
  int n_vars;                    // Number of boolean variables
  int n_vertices;                // Total vertices in the implication
      graph (2 per variable)
  vector<vector<int>> adj;       // Implication graph: adj[i] contains
      edges from node i
  vector<vector<int>> adj_t;     // Transposed graph for Kosaraju's
      algorithm
  vector<bool> used;             // Visited marker for DFS
  vector<int> order;             // Finishing order of vertices (DFS1)
  vector<int> comp;              // Component ID for each node (DFS2)
  vector<bool> assignment;       // Final truth assignment for each
      variable

  // Constructor initializes all data structures
  TwoSatSolver(int _n_vars)
    : n_vars(_n_vars),
      n_vertices(2 * _n_vars),
      adj(n_vertices),
      adj_t(n_vertices),
      used(n_vertices),
      comp(n_vertices, -1),
      assignment(n_vars) {
    order.reserve(n_vertices); // Pre-allocate memory for efficiency
  }

  // First DFS pass for Kosaraju's algorithm (on original graph)
  void dfs1(int v) {
    used[v] = true;
    for (int u : adj[v]) {
      if (!used[u])
        dfs1(u);
    }
    order.push_back(v); // Save the vertex post-DFS for reverse ordering
  }

  // Second DFS pass on the transposed graph to label components
  void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : adj_t[v]) {
      if (comp[u] == -1)
        dfs2(u, cl);
    }
  }

  // Solves the 2-SAT problem using Kosaraju's algorithm
  bool solve_2SAT() {
    // 1st pass: fill the order vector
    order.clear();
    used.assign(n_vertices, false);
    for (int i = 0; i < n_vertices; ++i) {
      if (!used[i])
        dfs1(i);
    }

    // 2nd pass: find SCCs in reverse postorder
    comp.assign(n_vertices, -1);
    for (int i = 0, j = 0; i < n_vertices; ++i) {
      int v = order[n_vertices - i - 1]; // Reverse postorder
      if (comp[v] == -1)
        dfs2(v, j++);
    }

    // Assign values to variables based on component comparison
```

```
    assignment.assign(n_vars, false);
    for (int i = 0; i < n_vertices; i += 2) {
      if (comp[i] == comp[i + 1])
        return false; // Contradiction: variable and its negation are in the same SCC
      assignment[i / 2] = comp[i] > comp[i + 1]; // True if var's component comes after its negation
    }
    return true;
  }

  // Adds a disjunction (a v b) to the implication graph
  // 'na' and 'nb' indicate negation: if true means !a or !b
  // Variables are 0-indexed. Bounds are inclusive for each literal (i.e., 0 to n_vars - 1)
  void add_disjunction(int a, bool na, int b, bool nb) {
    // Each variable `x` has two nodes:
    // x => 2*x, !x => 2*x + 1
    // We encode (a v b) as (!a -> b) and (!b -> a)
    a = 2 * a ^ na;
    b = 2 * b ^ nb;
    int neg_a = a ^ 1;
    int neg_b = b ^ 1;

    adj[neg_a].push_back(b);
    adj[neg_b].push_back(a);
    adj_t[b].push_back(neg_a);
    adj_t[a].push_back(neg_b);
  }
};
```

7.2 Articulation Points

```
/*
  Articulation Points (Cut Vertices) in an Undirected Graph
  ---------------------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive
  Time Complexity: O(V + E)
  Space Complexity: O(V)

  Use Case:
    - Identifies vertices whose removal increases the number of connected components.
    - Works on undirected graphs (connected or disconnected).
*/

int n; // Number of nodes in the graph
vector<vector<int>> adj; // Adjacency list of the undirected graph

vector<bool> visited; // Marks if a node was visited during DFS
vector<int> tin, low; // tin[v]: discovery time; low[v]: lowest discovery time reachable from subtree
int timer; // Global time counter for DFS

// DFS traversal to identify articulation points
void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
  int children = 0;
  for (int to : adj[v]) {
    if (to == p) continue; // Skip the parent edge
    if (visited[to]) {
      // Back edge
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      // Articulation point condition for non-root
      if (low[to] >= tin[v] && p != -1) {
        // v is an articulation point
        // handle_cutpoint(v);
      }
      ++children;
    }
  }
  // Articulation point condition for root
  if (p == -1 && children > 1) {
    // v is an articulation point
    // handle_cutpoint(v);
  }
}

// Initializes structures and launches DFS
void find_cutpoints() {
  timer = 0;
```

```
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);

  for (int i = 0; i < n; ++i) {
    if (!visited[i])
      dfs(i);
  }
}
```

7.3 Bellman-Ford

```
/*
  Bellman-Ford (SPFA variant) for Shortest Paths
  ----------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive
  Time Complexity: O(V * E) worst-case (amortized better)
  Space Complexity: O(V + E)

  Features:
    - Handles negative edge weights
    - Detects negative weight cycles (returns false if one exists)
    - Works on directed or undirected graphs

  Path Reconstruction:
    - To recover the path from source `s` to any node `u`:
        vector<int> path;
        for (int v = u; v != -1; v = parent[v])
          path.push_back(v);
        reverse(path.begin(), path.end());
*/

const int INF = 1<<30; // Large value to represent "infinity"
vector<vector<pair<int, int>>> adj; // adj[v] = list of (neighbor,
    weight) pairs
vector<int> parent; // parent(n, -1) for path reconstruction

// SPFA implementation to find shortest paths from source s
// d[i] will contain shortest distance from s to i
// Returns false if a negative cycle is detected
// For path reconstruction add vector<int>& parent as parameter
bool spfa(int s, vector<int>& d, vector<int>& parent) {
  int n = adj.size();
  d.assign(n, INF);
  vector<int> cnt(n, 0);       // Count how many times each node has
      been relaxed
  vector<bool> inqueue(n, false); // Tracks if a node is currently in
      queue
  queue<int> q;

  d[s] = 0;
  q.push(s);
  inqueue[s] = true;

  while (!q.empty()) {
    int v = q.front();
    q.pop();
    inqueue[v] = false;

    for (auto edge : adj[v]) {
      int to = edge.first;
      int len = edge.second;

      if (d[v] + len < d[to]) {
        parent[to] = v; // For path reconstruction
        d[to] = d[v] + len;
        if (!inqueue[to]) {
          q.push(to);
          inqueue[to] = true;
          cnt[to]++;
          if (cnt[to] > n)
            return false; // Negative weight cycle detected
        }
      }
    }
  }

  return true; // No negative cycles; shortest paths computed
}
```

7.4 Bipartite Checker

```
/*
  Bipartite Graph Checker (BFS-based)
  -----------------------------------
```

```
  Indexing: 0-based
  Time Complexity: O(V + E)
  Space Complexity: O(V)

  Handles disconnected graphs
*/

int n; // Number of nodes
vector<vector<int>> adj; // Adjacency list of the undirected graph

vector<int> side(n, -1); // -1 = unvisited, 0/1 = sides of bipartition
bool is_bipartite = true;
queue<int> q;

for (int st = 0; st < n; ++st) {
  if (side[st] == -1) {
    q.push(st);
    side[st] = 0; // Start with side 0
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int u : adj[v]) {
        if (side[u] == -1) {
          // Assign opposite side to neighbor
          side[u] = side[v] ^ 1;
          q.push(u);
        } else {
          // Conflict: adjacent nodes on same side
          is_bipartite &= side[u] != side[v];
        }
      }
    }
  }
}

cout << (is_bipartite ? "YES" : "NO") << endl;
```

7.5 Bipartite Maximum Matching

```
/*
  Maximum Bipartite Matching (Kuhn's Algorithm)
  ---------------------------------------------
  Indexing: 0-based
  Time Complexity: O(N * (E + N)) worst case
  Space Complexity: O(N + K + E)

  Input:
    - n: number of nodes on the left side
    - k: number of nodes on the right side
    - g: adjacency list where g[v] contains all right nodes adjacent to
       left node v

  Output:
    - Prints the pairs (left, right) in the matching
    - mt[r] = l means right node r is matched to left node l
*/

int n, k; // n: number of left nodes, k: number of right nodes
vector<vector<int>> g; // g[l]: list of right-side neighbors of left
    node l
vector<int> mt;        // mt[r]: matched left node for right node r (or
    -1 if unmatched)
vector<bool> used;     // used[l]: visited status for left node l during
     DFS

// Try to find an augmenting path from left node v
bool try_kuhn(int v) {
  if (used[v])
    return false;
  used[v] = true;
  for (int to : g[v]) {
    if (mt[to] == -1 || try_kuhn(mt[to])) {
      mt[to] = v;
      return true;
    }
  }
  return false;
}

int main() {
  //... reading the graph ...

  mt.assign(k, -1); // Right-side nodes initially unmatched
  for (int v = 0; v < n; ++v) {
    used.assign(n, false); // Reset visited for each left node
    try_kuhn(v);
```

```
  }
  // Output matched pairs (left+1, right+1 for 1-based output)
  for (int i = 0; i < k; ++i) {
    if (mt[i] != -1)
      printf("%d %d\n", mt[i] + 1, i + 1);
  }
  return 0;
}
```

7.6 Bipartite Minimum Maximum Matching2

```
/*
  THIS CODE HAS NOT BEEN TESTED BEFOREHAND
  Maximum Bipartite Matching - Hopcroft-Karp Algorithm
  -----------------------------------------------------
  Indexing: 0-based
  Time Complexity: O(sqrt(V) * E)
  Space Complexity: O(V + E)

  Input:
    - n: number of nodes on the left
    - m: number of nodes on the right
    - g[l]: adjacency list for left node l (list of right-side neighbors
        )

  Output:
    - match_right[r] = l means right node r is matched to left node l
    - match_left[l] = r means left node l is matched to right node r
    - Function returns total number of matching pairs
*/

const int INF = 1e9;

int n, m; // Number of nodes on left and right
vector<vector<int>> g;       // Adjacency list: g[l] contains neighbors
    of left node l
vector<int> match_left, match_right; // match_left[l], match_right[r]
vector<int> dist;            // Distance levels for BFS

// BFS to build layer graph, returns true if there's at least one
    unmatched node on the right reachable from left
bool bfs() {
  queue<int> q;
  dist.assign(n, INF);

  for (int l = 0; l < n; ++l) {
    if (match_left[l] == -1) {
      dist[l] = 0;
      q.push(l);
    }
  }

  bool found = false;

  while (!q.empty()) {
    int l = q.front(); q.pop();
    for (int r : g[l]) {
      int matched_l = match_right[r];
      if (matched_l == -1) {
        found = true; // Free right node reachable -> potential
            augmenting path
      } else if (dist[matched_l] == INF) {
        dist[matched_l] = dist[l] + 1;
        q.push(matched_l);
      }
    }
  }

  return found;
}

// DFS to find augmenting paths in the layered graph
bool dfs(int l) {
  for (int r : g[l]) {
    int matched_l = match_right[r];
    if (matched_l == -1 || (dist[matched_l] == dist[l] + 1 && dfs(
        matched_l))) {
      match_left[l] = r;
      match_right[r] = l;
      return true;
    }
  }
  dist[l] = INF;
  return false;
}

```

```cpp
// Main function to compute maximum matching
int hopcroft_karp() {
  match_left.assign(n, -1);
  match_right.assign(m, -1);

  int matching = 0;
  while (bfs()) {
    for (int l = 0; l < n; ++l) {
      if (match_left[l] == -1 && dfs(l)) {
        ++matching;
      }
    }
  }
  return matching;
}

int main() {
  cin >> n >> m;
  g.assign(n, {});
  int e; cin >> e;
  while (e--) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v); // u in [0, n-1], v in [0, m-1]
  }

  int res = hopcroft_karp();
  cout << "Maximum matching size: " << res << '\n';
  for (int r = 0; r < m; ++r) {
    if (match_right[r] != -1) {
      cout << match_right[r] + 1 << " " << r + 1 << '\n'; // 1-based output
    }
  }
}
```

7.7 Block Cut Tree

```cpp
/*
  Block-Cut Tree from Biconnected Components
  ------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive
  Time Complexity: O(V + E)
  Space Complexity: O(V + E)

  Features:
    - Identifies articulation points (cut vertices)
    - Extracts all biconnected components (BCCs)
    - Constructs the Block-Cut Tree:
        - Each BCC becomes a node in the tree
        - Each articulation point becomes its own node
        - An edge connects a BCC-node to each cutpoint in it

  Output:
    - `is_cutpoint`: true if node is an articulation point
    - `id[v]`: node ID of `v` in the block-cut tree
    - Returns the block-cut tree as an adjacency list
*/

vector<vector<int>> biconnected_components(vector<vector<int>> &g, // Adjacency list of the undirected graph
                                           vector<bool> &is_cutpoint, // Output vector (resized internally)
                                           vector<int> &id) { // Output vector (resized internally)

  int n = g.size();
  vector<vector<int>> comps; // Stores all biconnected components
  vector<int> stk;           // Stack of visited nodes for current component
  vector<int> num(n), low(n); // DFS discovery time and low-link values
  is_cutpoint.assign(n, false);

  // DFS to find BCCs and articulation points
  function<void(int, int, int&)> dfs = [&](int node, int parent, int & timer) {
    num[node] = low[node] = ++timer;
    stk.push_back(node);
    for (int son : g[node]) {
      if (son == parent) continue;
      if (num[son]) {
        // Back edge
        low[node] = min(low[node], num[son]);
```

```
    } else {
      dfs(son, node, timer);
      low[node] = min(low[node], low[son]);
      // Check articulation point condition
      if (low[son] >= num[node]) {
        is_cutpoint[node] = (num[node] > 1 || num[son] > 2); // For root and non-root
        comps.push_back({node});
        while (comps.back().back() != son) {
          comps.back().push_back(stk.back());
          stk.pop_back();
        }
      }
    }
  }
};

int timer = 0;
dfs(0, -1, timer);

id.resize(n); // Maps each original node to its block-cut tree node ID

// Build block-cut tree using articulation points and BCCs
function<vector<vector<int>>()> build_tree = [&]() {
  vector<vector<int>> t(1); // Dummy index 0 (not used)
  int node_id = 1; // Start assigning block-cut tree IDs from 1
  // Assign unique tree node IDs to cutpoints
  for (int node = 0; node < n; ++node) {
    if (is_cutpoint[node]) {
      id[node] = node_id++;
      t.push_back({});
    }
  }
  // Assign each component a new node and connect it to its cutpoints
  for (auto &comp : comps) {
    int bcc_node = node_id++;
    t.push_back({});
    for (int u : comp) {
      if (!is_cutpoint[u]) {
        id[u] = bcc_node;
      } else {
        t[bcc_node].push_back(id[u]);
        t[id[u]].push_back(bcc_node);
      }
    }
  }
  return t;
};

return build_tree(); // Return the block-cut tree
}
```

7.8 Blossom

```
/*
  Edmonds' Blossom Algorithm (Maximum Matching in General Graphs)
  ---------------------------------------------------------------
  Indexing: 1-based
  Node Bounds: [1, n]
  Time Complexity: O(n^3) in worst case
  Space Complexity: O(n^2)

  Features:
    - Handles odd-length cycles (blossoms)
    - Works on any undirected graph (not just bipartite)
    - Uses BFS with blossom contraction and path augmentation

  Input:
    - n: number of vertices
    - add_edge(u, v): undirected edges between nodes (1 <= u,v <= n)

  Output:
    - maximum_matching(): returns size of max matching
    - match[u]: matched vertex for node u (or 0 if unmatched)
*/

const int N = 2009;
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());

struct Blossom {
  int vis[N];        // vis[u]: -1 = unvisited, 0 = in queue, 1 = outer layer
  int par[N];        // par[u]: parent in alternating tree
  int orig[N];       // orig[u]: base of blossom u belongs to
  int match[N];      // match[u]: matched partner of u (0 if unmatched)
  int aux[N];        // aux[u]: visit marker for LCA
```

```
  int t;              // global timestamp for LCA markers
  int n;              // number of nodes
  bool ad[N];         // ad[u]: whether u is reachable in an alternating
      path
  vector<int> g[N];   // g[u]: adjacency list
  queue<int> Q;       // BFS queue

  // Constructor: initializes data for n nodes
  Blossom() {}
  Blossom(int _n) {
    n = _n;
    t = 0;
    for (int i = 0; i <= n; ++i) {
      g[i].clear();
      match[i] = par[i] = vis[i] = aux[i] = ad[i] = orig[i] = 0;
    }
  }

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

  // Augment the matching along the alternating path from u to v
  void augment(int u, int v) {
    int pv = v, nv;
    do {
      pv = par[v];
      nv = match[pv];
      match[v] = pv;
      match[pv] = v;
      v = nv;
    } while (u != pv);
  }

  int lca(int v, int w) {
    ++t; // Increment timestamp for LCA markers
    while (true) {
      if (v) {
        if (aux[v] == t) return v;
        aux[v] = t;
        v = orig[par[match[v]]]; // Move to the parent in the
            alternating tree
      }
      swap(v, w);
    }
  }

  // Contract a blossom from v and w with common ancestor a
  void blossom(int v, int w, int a) {
    while (orig[v] != a) {
      par[v] = w;
      w = match[v];
      ad[v] = true;
      if (vis[w] == 1) Q.push(w), vis[w] = 0;
      orig[v] = orig[w] = a;
      v = par[w];
    }
  }

  // Find augmenting path starting from unmatched node u
  bool bfs(int u) {
    fill(vis + 1, vis + n + 1, -1);
    iota(orig + 1, orig + n + 1, 1);
    Q = queue<int>();
    Q.push(u);
    vis[u] = 0;

    while (!Q.empty()) {
      int v = Q.front(); Q.pop();
      ad[v] = true;
      for (int x : g[v]) {
        if (vis[x] == -1) {
          par[x] = v;
          vis[x] = 1;
          if (!match[x]) {
            augment(u, x);
            return true;
          }
          Q.push(match[x]);
          vis[match[x]] = 0;
        } else if (vis[x] == 0 && orig[v] != orig[x]) {
          int a = lca(orig[v], orig[x]);
          blossom(x, v, a);
          blossom(v, x, a);
        }
```

```
      }
    }
    return false;
  }

  // Computes maximum matching and returns the size
  int maximum_matching() {
    int ans = 0;
    vector<int> p(n - 1);
    iota(p.begin(), p.end(), 1);
    shuffle(p.begin(), p.end(), rnd);
    for (int i = 1; i <= n; ++i) {
      shuffle(g[i].begin(), g[i].end(), rnd);
    }

    // Greedy matching: try to match unmatched nodes directly
    for (int u : p) {
      if (!match[u]) {
        for (int v : g[u]) {
          if (!match[v]) {
            match[u] = v;
            match[v] = u;
            ++ans;
            break;
          }
        }
      }
    }

    // Augmenting path phase
    for (int i = 1; i <= n; ++i) {
      if (!match[i] && bfs(i)) ++ans;
    }

    return ans;
  }
} M;

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

  int t;
  cin >> t;
  while (t--) {
    int n, m;
    cin >> n >> m;
    M = Blossom(n);
    // Read all edges
    for (int i = 0; i < m; i++) {
      int u, v;
      cin >> u >> v;
      M.add_edge(u, v);
    }
    // Compute max matching
    int matched = M.maximum_matching();
    if (matched * 2 == n) {
      // Perfect matching
      cout << 0 << '\n';
    } else {
      // Find reachable unmatched nodes in alternating trees
      memset(M.ad, 0, sizeof M.ad);
      for (int i = 1; i <= n; i++) {
        if (M.match[i] == 0) M.bfs(i);
      }
      int unmatched_reachable = 0;
      for (int i = 1; i <= n; i++) {
        unmatched_reachable += M.ad[i];
      }
      cout << unmatched_reachable << '\n';
    }
  }
  return 0;
}
```

7.9 Bridges

```
/*
  Bridge-Finding in an Undirected Graph
  -------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive
  Time Complexity: O(V + E)
  Space Complexity: O(V)

  Input:
```

```
    n    - Number of nodes in the graph
    adj  - Adjacency list of the undirected graph

  Output:
    - Call `find_bridges()` to populate bridge information.
    - Modify the DFS `Bridge` section to store or print the bridges.
      A bridge is an edge (v, to) such that removing it increases the
          number of connected components.
*/

int n; // Number of nodes
vector<vector<int>> adj; // Adjacency list

vector<bool> visited; // Marks visited nodes
vector<int> tin, low; // tin[v]: discovery time; low[v]: lowest ancestor
     reachable
int timer; // Global DFS timer

// DFS to detect bridges
void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
  for (int to : adj[v]) {
    if (to == p) continue; // Skip edge to parent
    if (visited[to]) {
      // Back edge
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      // Bridge condition: if no back edge connects subtree rooted at `
          to` to ancestors of `v`
      if (low[to] > tin[v]) {
        // (v, to) is a bridge
        // Example: bridges.push_back({v, to});
      }
    }
  }
}

// Initialize tracking structures and run DFS
void find_bridges() {
  timer = 0;
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  for (int i = 0; i < n; ++i) {
    if (!visited[i])
      dfs(i);
  }
}
```

7.10 Bridges Online

```
/*
  Online Bridge-Finding (Dynamic Edge Insertion)
  ----------------------------------------------
  Indexing: 0-based
  Node Bounds: [0, n-1] inclusive
  Time Complexity:
    - Amortized O(log^2N) per edge addition
  Space Complexity: O(V)

  Features:
    - Maintains the number of bridges dynamically as edges are added one
        by one.
    - Detects if adding an edge merges different 2-edge-connected
        components.
    - No deletions supported.

  Input:
    init(n)     - Initializes the data structure for a graph with n
        nodes.
    add_edge(a, b) - Adds an undirected edge between nodes a and b.

  Output:
    `bridges` - Global variable representing the current number of
        bridges.
*/

vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges; // Number of bridges in the graph
int lca_iteration;
vector<int> last_visit;

// Initializes the data structures
```

```cpp
void init(int n) {
  par.resize(n);
  dsu_2ecc.resize(n);
  dsu_cc.resize(n);
  dsu_cc_size.resize(n);
  last_visit.assign(n, 0);
  lca_iteration = 0;
  bridges = 0;

  for (int i = 0; i < n; ++i) {
    par[i] = -1;
    dsu_2ecc[i] = i;
    dsu_cc[i] = i;
    dsu_cc_size[i] = 1;
  }
}

// Finds the representative of the 2-edge-connected component of node v
int find_2ecc(int v) {
  if (v == -1) return -1;
  return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
}

// Finds the connected component representative of the component
    containing v
int find_cc(int v) {
  v = find_2ecc(v);
  return dsu_cc[v] == v ? v : dsu_cc[v] = find_cc(dsu_cc[v]);
}

// Makes node v the root of its tree, rerouting parent pointers upward
void make_root(int v) {
  int root = v;
  int child = -1;
  while (v != -1) {
    int p = find_2ecc(par[v]);
    par[v] = child;
    dsu_cc[v] = root;
    child = v;
    v = p;
  }
  dsu_cc_size[root] = dsu_cc_size[child];
}

// Merges paths from a and b to their lowest common ancestor in the 2ECC
    forest
void merge_path(int a, int b) {
  ++lca_iteration;
  vector<int> path_a, path_b;
  int lca = -1;

  while (lca == -1) {
    if (a != -1) {
      a = find_2ecc(a);
      path_a.push_back(a);
      if (last_visit[a] == lca_iteration) {
        lca = a;
        break;
      }
      last_visit[a] = lca_iteration;
      a = par[a];
    }
    if (b != -1) {
      b = find_2ecc(b);
      path_b.push_back(b);
      if (last_visit[b] == lca_iteration) {
        lca = b;
        break;
      }
      last_visit[b] = lca_iteration;
      b = par[b];
    }
  }

  // Merge all nodes on path_a and path_b into the same 2ECC
  for (int v : path_a) {
    dsu_2ecc[v] = lca;
    if (v == lca) break;
    --bridges;
  }
  for (int v : path_b) {
    dsu_2ecc[v] = lca;
    if (v == lca) break;
    --bridges;
  }
}
```

```

// Adds an undirected edge between a and b and updates bridge count
void add_edge(int a, int b) {
  a = find_2ecc(a);
  b = find_2ecc(b);
  if (a == b) return; // Already in the same 2ECC

  int ca = find_cc(a);
  int cb = find_cc(b);

  if (ca != cb) {
    // Bridge found - connects two different components
    ++bridges;
    // Union by size
    if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
      swap(a, b);
      swap(ca, cb);
    }
    make_root(a);
    par[a] = b;
    dsu_cc[a] = b;
    dsu_cc_size[cb] += dsu_cc_size[a];
  } else {
    // No new bridge, but must merge paths to unify 2ECCs
    merge_path(a, b);
  }
}

// Example usage
int main() {
  init(n);
  for (auto [u, v] : edges) {
    add_edge(u, v);
    cout << "Current bridge count: " << bridges << '\n';
  }
}
```

7.11 Dijkstra

```
vector<vector<pair<int, int>>> adj(n); // Adjacency list (node, weight)
vector<ll> dist(n, 1LL << 61); // Distance array initialized to infinity

priority_queue<pair<ll, int>> q; // Max-heap, so we push negative
    weights to simulate min-heap
dist[0] = 0; // Starting node distance
q.push({0, 0}); // (distance, vertex)

while (!q.empty()) {
  auto [w, v] = q.top(); q.pop();
  w = -w; // Convert back to positive
  if (w > dist[v]) continue; // Skip outdated entry
  for (auto [u, cost] : adj[v]) {
    if (dist[v] + cost < dist[u]) {
      dist[u] = dist[v] + cost;
      q.push({-dist[u], u}); // Push updated distance (negated)
    }
  }
}
```

7.12 Eulerian Path

An Eulerian Path is a path that passes through every edge once. For an undirected graph an eulerian path exists if the degree of every node is even or the degree of exactly two nodes is odd. In the first case, the eulerian path is also an eulerian circuit or cycle. In a directed graph, an eulerian path exists if at most one noda has $out_i - in_i = 1$ and at most one node has $in_i - out_i = 1$. A cycle exists if $in_i - out_i = 0$ for all i.

```
/*
  Eulerian Path (Hierholzer's Algorithm)
  --------------------------------------
  Time Complexity: O(E)
  Space Complexity: O(V + E)

  Input:
    - g: adjacency list of the graph
        * Directed: vector<vector<pair<int, int>>> g
            where g[v] = list of {to, edge_index}
        * Undirected: vector<vector<int>> g
            where g[v] = list of neighbors
    - seen: vector<bool> seen(E) - only needed for directed version
    - path: vector<int> path - will be filled in reverse order of
        traversal
            reverse(path.begin(), path.end());
*/

// Directed Version //
```

```cpp
void dfs_directed(int node) {
  while (!g[node].empty()) {
    auto [son, idx] = g[node].back();
    g[node].pop_back();
    if (seen[idx]) continue;  // Skip if edge already visited
    seen[idx] = true;
    dfs_directed(son);
  }
  path.push_back(node); // Post-order insertion (reverse of actual path)
}

// Undirected Version //
void dfs_undirected(int node) {
  while (!g[node].empty()) {
    int son = g[node].back();
    g[node].pop_back();
    dfs_undirected(son);
  }
  path.push_back(node); // Post-order insertion
}
```

7.13 Floyd-Warshall

```cpp
/*
  Floyd-Warshall Algorithm (All-Pairs Shortest Paths)
  ---------------------------------------------------
  Indexing: 0-based
  Time Complexity: O(V^3)
  Space Complexity: O(V^2)

  Input:
    - d: distance matrix of size n x n
        * d[i][j] should be initialized as:
            - 0 if i == j
            - weight of edge (i, j) if exists
            - INF (e.g. 1e18) otherwise
*/

vector<vector<ll>> d(n, vector<ll>(n, 1e18)); // distance matrix

// This version is by default adapted for UNDIRECTED graphs.
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) { // For directed graphs, use j = 0; j < n; j++
      long long new_dist = d[i][k] + d[k][j];
      if (new_dist < d[i][j]) {
        d[i][j] = d[j][i] = new_dist; // update both directions for undirected graph
      }
    }
  }
}
```

7.14 Kruskal

```cpp
/*
  Kruskal's Algorithm (Minimum Spanning Tree - MST)
  -------------------------------------------------
  Indexing: 0-based for nodes in edges
  Time Complexity: O(E log E)
  Space Complexity: O(N)

  Input:
    - N: number of nodes
    - edges: list of weighted edges in form {weight, {u, v}}

  Output:
    - Returns total weight of the MST if the graph is connected
    - Returns -1 if MST cannot be formed (i.e., graph is disconnected)

  Note:
    - Requires a Disjoint Set Union (DSU) / Union-Find data structure with:
        - unite(a, b): merges components, returns true if successful
        - size(v): returns size of component containing v
*/

template <class T>
T kruskal(int N, vector<pair<T, pair<int, int>>> edges) {
  sort(edges.begin(), edges.end()); // Sort by weight (non-decreasing)
  T ans = 0;
  DSU D(N); // Disjoint Set Union for N nodes
  for (auto &[w, uv] : edges) {
    int u = uv.first, v = uv.second;
```

```
    if (D.unite(u, v)) {
      ans += w; // Add edge to MST if u and v are in different components
    }
  }
  // Check if MST spans all nodes (i.e., one component of size N)
  return (D.size(0) == N ? ans : -1);
}
```

7.15 Marriage

```
/*
  Male-Optimal Stable Marriage Problem (Gale-Shapley Algorithm)
  -------------------------------------------------------------
  Indexing: 0-based
  Bounds: 0 <= i, j < n
  Time Complexity: O(n^2)
  Space Complexity: O(n^2)

  Input:
    - n: Number of men/women (equal)
    - gv[i][j]: j-th most preferred woman for man i
    - om[i][j]: j-th most preferred man for woman i
        * Both are permutations of {0, ..., n-1}
        * om must be inverted to get om[w][m] = woman w's ranking of man m

  Output:
    - pm[i]: Woman matched to man i (i.e. pairings)
    - pv[i]: Man matched to woman i
*/

#define MAXN 1000
int gv[MAXN][MAXN], om[MAXN][MAXN]; // Male and female preference lists
int pv[MAXN], pm[MAXN];             // pv[woman] = man, pm[man] = woman
int pun[MAXN];                      // pun[man] = next woman to propose to

void stableMarriage(int n) {
  fill_n(pv, n, -1);   // All women initially unmatched
  fill_n(pm, n, -1);   // All men initially unmatched
  fill_n(pun, n, 0);   // Each man starts at his top preference

  int unmatched = n;   // Number of free men
  int i = n - 1;       // Current man index (rotates over all men)

  #define engage pm[j] = i; pv[i] = j;

  while (unmatched) {
    while (pm[i] == -1) {
      int j = gv[i][pun[i]++]; // Next woman on man i's list

      if (pv[j] == -1) {
        // Woman j is free -> engage with man i
        unmatched--;
        engage;
      } else if (om[j][i] < om[j][pv[j]]) {
        // Woman j prefers i over her current partner
        int loser = pv[j];
        pm[loser] = -1;
        engage;
        i = loser; // Reconsider the rejected man
      }
    }

    // Move to next unmatched man
    i--;
    if (i < 0) i = n - 1;
  }

  #undef engage
}
```

7.16 SCC

```
/*
  Strongly Connected Components (Kosaraju's Algorithm)
  ----------------------------------------------------
  Indexing: 0-based
  Time Complexity: O(V + E)
  Space Complexity: O(V + E)

  Input:
    - n: number of nodes
    - m: number of directed edges
    - adj: original graph
```

```
    - adjr: reversed graph

  Output:
    - comp[i]: component ID of node i
    - order[]: nodes in reverse post-order (1st DFS)
    - nc: is the number of unique comp values
*/

vector<vector<int>> adj, adjr;
vector<bool> vis;
vector<int> order, comp;

// First DFS: post-order on original graph
void dfs(int v) {
  vis[v] = true;
  for (int u : adj[v]) {
    if (!vis[u])
      dfs(u);
  }
  order.push_back(v); // Record post-order
}

// Second DFS: assign component IDs on reversed graph
void dfsr(int v, int k) {
  vis[v] = true;
  comp[v] = k;
  for (int u : adjr[v]) {
    if (!vis[u])
      dfsr(u, k);
  }
}

void solve() {
  int n, m;
  cin >> n >> m;
  adj.assign(n, vector<int>());
  adjr.assign(n, vector<int>());
  comp.resize(n);
  // Read edges and build both original and reversed graphs
  for (int i = 0; i < m; i++) {
    int a, b;
    cin >> a >> b;
    a--; b--;
    adj[a].push_back(b);
    adjr[b].push_back(a);
  }
  // First pass: DFS on original graph to get order
  vis.assign(n, false);
  order.clear();
  for (int i = 0; i < n; i++) {
    if (!vis[i]) dfs(i);
  }
  // Second pass: DFS on reversed graph using reverse post-order
  vis.assign(n, false);
  int nc = 0;
  for (int i = n - 1; i >= 0; i--) {
    int v = order[i];
    if (!vis[v]) {
      dfsr(v, nc++);
    }
  }
  // comp[i] now holds the component ID for node i (0-based)
  // nc = number of strongly connected components
}
```

7.17 Stoer-Wagner

```
/*
                                    Stoer - Wagner
  ------------------------------------------------------------------------------

  Solves the minimum cut problem in undirected weighted graphs with non-
      negative weights.

  Time Complexity: O(V ^ 3)
  Space Complexity: O(V ^ 2)
*/

mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
struct StoerWagner {
  int n;
  long long G[N][N], dis[N];
  int idx[N];
  bool vis[N];
  const long long inf = 1e18;
  StoerWagner() {}
```

```
  StoerWagner(int _n) {
    n = _n;
    memset(G, 0, sizeof G);
  }
  void add_edge(int u, int v, long long w) { //undirected edge, multiple
       edges are merged into one edge
    if (u != v) {
      G[u][v] += w;
      G[v][u] += w;
    }
  }
  long long solve() {
    long long ans = inf;
    for (int i = 0; i < n; ++ i) idx[i] = i + 1;
    shuffle(idx, idx + n, rnd);
    while (n > 1) {
      int t = 1, s = 0;
      for (int i = 1; i < n; ++ i) {
        dis[idx[i]] = G[idx[0]][idx[i]];
        if (dis[idx[i]] > dis[idx[t]]) t = i;
      }
      memset(vis, 0, sizeof vis);
      vis[idx[0]] = true;
      for (int i = 1; i < n; ++ i) {
        if (i == n - 1) {
          if (ans > dis[idx[t]]) ans = dis[idx[t]]; //idx[s] - idx[t] is
               in two halves of the  mincut
          if (ans == 0) return 0;
          for (int j = 0; j < n; ++ j) {
            G[idx[s]][idx[j]] += G[idx[j]][idx[t]];
            G[idx[j]][idx[s]] += G[idx[j]][idx[t]];
          }
          idx[t] = idx[-- n];
        }
        vis[idx[t]] = true;
        s = t;
        t = -1;
        for (int j = 1; j < n; ++ j) {
          if (!vis[idx[j]]) {
            dis[idx[j]] += G[idx[s]][idx[j]];
            if (t == -1 || dis[idx[t]] < dis[idx[j]]) t = j;
          }
        }
      }
      for (int i = 1; i < n; ++ i) {
        int u = idx[i];
        for (int j = 0; j < i; ++ j) {
          int v = idx[j];
          G[u][v] = G[v][u] = G[u][v] - G[v][u];
        }
      }
      n --;
    }
    return ans;
  }
};

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int t, cs = 0;
  cin >> t;
  while (t--) {
    int n, m;
    cin >> n >> m;
    StoerWagner st(n);
    while (m--) {
      int u, v, w;
      cin >> u >> v >> w;
      st.add_edge(u, v, w);
    }
    cout << "Case␣#" << ++cs << ":␣" << st.solve() << '\n';
  }
  return 0;
}
```

8 Linear Algebra

8.1 Gaussian Elimination

```
/*
  Gaussian elimination
  ---------------------------------
  Utilized for solving linear systems of equations.

  If the system is modulo 2, use GaussianEliminationModulo instead.

  Time Complexity: O(n ^ 3)
*/


const double eps = 1e-9;
int Gauss(vector<vector<double>> a, vector<double> &ans) {
  int n = (int)a.size(), m = (int)a[0].size() - 1;
```

```
  vector<int> pos(m, -1);
  double det = 1; int rank = 0;
  for(int col = 0, row = 0; col < m && row < n; ++col) {
    int mx = row;
    for(int i = row; i < n; i++) if(fabs(a[i][col]) > fabs(a[mx][col]))
        mx = i;
    if(fabs(a[mx][col]) < eps) {det = 0; continue;}
    for(int i = col; i <= m; i++) swap(a[row][i], a[mx][i]);
    if (row != mx) det = -det;
    det *= a[row][col];
    pos[col] = row;
    for(int i = 0; i < n; i++) {
      if(i != row && fabs(a[i][col]) > eps) {
        double c = a[i][col] / a[row][col];
        for(int j = col; j <= m; j++) a[i][j] -= a[row][j] * c;
      }
    }
    ++row; ++rank;
  }
  ans.assign(m, 0);
  for(int i = 0; i < m; i++) {
    if(pos[i] != -1) ans[i] = a[pos[i]][m] / a[pos[i]][i];
  }
  for(int i = 0; i < n; i++) {
    double sum = 0;
    for(int j = 0; j < m; j++) sum += ans[j] * a[i][j];
    if(fabs(sum - a[i][m]) > eps) return -1; //no solution
  }
  for(int i = 0; i < m; i++) if(pos[i] == -1) return 2; //infinte
      solutions
  return 1; //unique solution
}
int main() {
  int n, m; cin >> n >> m;
  vector< vector<double> > v(n);
  for(int i = 0; i < n; i++) {
    for(int j = 0; j <= m; j++) {
      double x; cin >> x; v[i].push_back(x);
    }
  }
  vector<double> ans;
  int k = Gauss(v, ans);
  if(k) for(int i = 0; i < n; i++) cout << fixed << setprecision(5) <<
      ans[i] << ' ';
  else cout << "no solution\n";
  return 0;
}
```

8.2 Gaussian Elimination Modulo

```
/*
  Gaussian elimination Modulo 2
  ---------------------------------
  Utilized for solving linear systems of equations modulo2.

  Time Complexity: O(n ^ 3)
*/

//n = number of equations, m = number of variables
int Gauss(int n, int m, vector<bitset<N>> a, bitset<N> &ans) {
  //reversing for lexocgraphically largest solution
  for (int i = 0; i < n; i++) {
      bitset<N> tmp;
      for (int j = 0; j < m; j++) tmp[j] = a[i][m - j - 1];
      tmp[m] = a[i][m];
      a[i] = tmp;
  }
  int rank = 0, det = 1;
  vector<int> pos(N, -1);
  for(int col = 0, row = 0; col < m && row < n; ++col) {
    int mx = row;
    for(int i = row; i < n; ++i) if(a[i][col]) { mx = i; break; }
    if(!a[mx][col]) { det = 0; continue; }
    swap(a[mx], a[row]);
    if(row != mx) det = det == 0 ? 0 : 1;
    det &= a[row][col];
    pos[col] = row;
    //forward elimination
    for(int i = row + 1; i < n; ++i) if (i != row && a[i][col]) a[i] ^=
        a[row];
    ++row, ++rank;
  }
  ans.reset();
  //backward substituition
  for (int i = m - 1; i >= 0; i--) {
      if (pos[i] == -1) ans[i] = true;
```

```
      else {
        int k = pos[i];
        for (int j = i + 1; j < m; j++) if (a[k][j]) ans[i] = ans[i] ^
            ans[j];
        ans[i] = ans[i] ^ a[k][m];
      }
  }
  for(int i = rank; i < n; ++i) if(a[i][m]) return -1; //no solution
  //reversing again beacuse we reversed earlier
  bitset<N> tmp;
  for (int j = 0; j < m; j++) tmp[j] = ans[m - j - 1];
  ans = tmp;
  int free_var = 0;
  for(int i = 0; i < m; ++i) if(pos[i] == -1) free_var++;
  return free_var; //has solution
}
string read() {
   string t;
   if(!(cin >> t)) return "";
   if(t.empty() || t == "and") return "";
   while(t[0] == '(') t.erase(t.begin());
   while(t.back() == ')') t.pop_back();
   return t;
}
bool is_var(string t) { return t.size() > 0 && t[0] == 'x'; }
int get_var(string t) { return atoi(t.substr(1).c_str()) - 1; }
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, m; cin >> n >> m;
  vector<bitset<N>> bs(n, bitset<N>(0));
  for(int i = 0; i < n; i++) {
     string s;
     bool eq = 1;
     while((s = read()).size() > 0) {
        if(is_var(s)) {
          int x = get_var(s);
          bs[i][x] = bs[i][x] ^ 1;
        }
        else if(s == "not") eq ^= 1;
     }
     bs[i][m] = eq;
  }
  bitset<N> ans;
  int ok = Gauss(n, m, bs, ans);
  if (ok == -1) cout << "impossible\n";
  else {
    for (int i = 0; i < m; i++) cout << "FT"[ans[i]]; cout << '\n';
  }
   return 0;
}
//https://codeforces.com/gym/101908/problem/M
```

8.3 Simplex

```
/*
Parametric Self-Dual Simplex method
Solve a canonical LP:
  min or max. c x
  s.t. A x <= b
    x >= 0
*/
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-9, oo = numeric_limits<double>::infinity();

typedef vector<double> vec;
typedef vector<vec> mat;

pair<vec, double> simplexMethodPD(const mat &A, const vec &b, const vec
    &c, bool mini = true){
  int n = c.size(), m = b.size();
  mat T(m + 1, vec(n + m + 1));
  vector<int> base(n + m), row(m);

  for(int j = 0; j < m; ++j){
    for(int i = 0; i < n; ++i)
      T[j][i] = A[j][i];
    row[j] = n + j;
    T[j][n + j] = 1;
    base[n + j] = 1;
    T[j][n + m] = b[j];
  }

  for(int i = 0; i < n; ++i)
    T[m][i] = c[i] * (mini ? 1 : -1);
```

```
31
32    while(true){
33      int p = 0, q = 0;
34      for(int i = 0; i < n + m; ++i)
35        if(T[m][i] <= T[m][p])
36          p = i;
37
38      for(int j = 0; j < m; ++j)
39        if(T[j][n + m] <= T[q][n + m])
40          q = j;
41
42      double t = min(T[m][p], T[q][n + m]);
43
44      if(t >= -eps){
45        vec x(n);
46        for(int i = 0; i < m; ++i)
47          if(row[i] < n) x[row[i]] = T[i][n + m];
48        return {x, T[m][n + m] * (mini ? -1 : 1)}; // optimal
49      }
50
51      if(t < T[q][n + m]){
52        // tight on c -> primal update
53        for(int j = 0; j < m; ++j)
54          if(T[j][p] >= eps)
55            if(T[j][p] * (T[q][n + m] - t) >= T[q][p] * (T[j][n + m] - t))
56              q = j;
57
58        if(T[q][p] <= eps)
59          return {vec(n), oo * (mini ? 1 : -1)}; // primal infeasible
60      }else{
61        // tight on b -> dual update
62        for(int i = 0; i < n + m + 1; ++i)
63          T[q][i] = -T[q][i];
64
65        for(int i = 0; i < n + m; ++i)
66          if(T[q][i] >= eps)
67            if(T[q][i] * (T[m][p] - t) >= T[q][p] * (T[m][i] - t))
68              p = i;
69
70        if(T[q][p] <= eps)
71          return {vec(n), oo * (mini ? -1 : 1)}; // dual infeasible
72      }
73
74      for(int i = 0; i < m + n + 1; ++i)
75        if(i != p) T[q][i] /= T[q][p];
76
77      T[q][p] = 1; // pivot(q, p)
78      base[p] = 1;
79      base[row[q]] = 0;
80      row[q] = p;
81
82      for(int j = 0; j < m + 1; ++j){
83        if(j != q){
84          double alpha = T[j][p];
85          for(int i = 0; i < n + m + 1; ++i)
86            T[j][i] -= T[q][i] * alpha;
87        }
88      }
89    }
90
91    return {vec(n), oo};
92  }
93
94  int main(){
95    int m, n;
96    bool mini = true;
97    cout << "Numero de restricciones: ";
98    cin >> m;
99    cout << "Numero de incognitas: ";
100   cin >> n;
101   mat A(m, vec(n));
102   vec b(m), c(n);
103   for(int i = 0; i < m; ++i){
104     cout << "Restriccion #" << (i + 1) << ": ";
105     for(int j = 0; j < n; ++j){
106       cin >> A[i][j];
107     }
108     cin >> b[i];
109   }
110   cout << "[0]Max o [1]Min?: ";
111   cin >> mini;
112   cout << "Coeficientes de " << (mini ? "min" : "max") << " z: ";
113   for(int i = 0; i < n; ++i){
114     cin >> c[i];
115   }
116   cout.precision(6);
```

```
  auto ans = simplexMethodPD(A, b, c, mini);
  cout << (mini ? "Min" : "Max") << " z = " << ans.second << ", cuando: \n";
  for(int i = 0; i < ans.first.size(); ++i){
    cout << "x_" << (i + 1) << " = " << ans.first[i] << "\n";
  }
  return 0;
}
```

9 Math

9.1 BinPow

```
ll binpow(ll a, ll b){
    ll r=1;
    while(b){
        if(b%2)
            r=(r*a)%MOD;
        a=(a*a)%MOD;
        b/=2;
    }
    return r;
}

ll divide(ll a, ll b){
    return ((a%MOD)*binpow(b, MOD-2))%MOD;
}
void inverses(long long p) {
  inv[MAXN] = exp(fac[MAXN], p - 2, p);
  for (int i = MAXN; i >= 1; i--) { inv[i - 1] = inv[i] * i % p; }
}
```

9.2 Diophantine

If one solution is (x_0, y_0) all solutions can be obtained by $x = x_0 + k * \frac{b}{gcd(a,b)}$ and $y = y_0 - k * \frac{a}{gcd(a,b)}$.

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}



//n variables
vector<ll> find_any_solution(vector<ll> a, ll c) {
  int n = a.size();
  vector<ll> x;
  bool all_zero = true;
  for (int i = 0; i < n; i++) {
    all_zero &= a[i] == 0;
  }
  if (all_zero) {
    if (c) return {};
    x.assign(n, 0);
    return x;
  }
  ll g = 0;
  for (int i = 0; i < n; i++) {
    g = __gcd(g, a[i]);
  }
  if (c % g != 0) return {};
  if (n == 1) {
    return {c / a[0]};
  }
  vector<ll> suf_gcd(n);
```

```
  suf_gcd[n - 1] = a[n - 1];
  for (int i = n - 2; i >= 0; i--) {
    suf_gcd[i] = __gcd(suf_gcd[i + 1], a[i]);
  }
  ll cur = c;
  for (int i = 0; i + 1 < n; i++) {
    ll x0, y0, g;
    // solve for a[i] * x + suf_gcd[i + 1] * (y / suf_gcd[i + 1]) = cur
    bool ok = find_any_solution(a[i], suf_gcd[i + 1], cur, x0, y0, g);
    assert(ok);
    {
      // trying to minimize x0 in case x0 becomes big
      // it is needed for this problem, not needed in general
      ll shift = abs(suf_gcd[i + 1] / g);
      x0 = (x0 % shift + shift) % shift;
    }
    x.push_back(x0);

    // now solve for the next suffix
    cur -= a[i] * x0;
  }
  x.push_back(a[n - 1] == 0 ? 0 : cur / a[n - 1]);
  return x;
}
```

9.3 Discrete Logarithm

Finds discrete logarithm in $O(\sqrt{(m)})$.

```
// Returns minimum x for which a ^ x % m = b % m, a and m are coprime.
int solve(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;

    int an = 1;
    for (int i = 0; i < n; ++i)
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

    for (int p = 1, cur = 1; p <= n; ++p) {
        cur = (cur * 1ll * an) % m;
        if (vals.count(cur)) {
            int ans = n * p - vals[cur];
            return ans;
        }
    }
    return -1;
}

// Returns minimum x for which a ^ x % m = b % m.
int solve(int a, int b, int m) {
    a %= m, b %= m;
    int k = 1, add = 0, g;
    while ((g = gcd(a, m)) > 1) {
        if (b == k)
            return add;
        if (b % g)
            return -1;
        b /= g, m /= g, ++add;
        k = (k * 1ll * a / g) % m;
    }

    int n = sqrt(m) + 1;
    int an = 1;
    for (int i = 0; i < n; ++i)
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

    for (int p = 1, cur = k; p <= n; ++p) {
        cur = (cur * 1ll * an) % m;
        if (vals.count(cur)) {
            int ans = n * p - vals[cur] + add;
            return ans;
        }
    }
    return -1;
```

```
}
```

9.4 Divisors

```
ll numberOfDivisors(ll num) {
  ll total = 1;
  for (int i = 2; (ll)i * i <= num; i++) {
    if (num % i == 0){
      int e = 0;
      do{
        e++;
        num /= i;
      } while (num % i == 0);
      total *= e + 1;
    }
  }
  if (num > 1) total *= 2;
  return total;
}

ll SumOfDivisors(ll num) {
  ll total = 1;
  for (int i = 2; (ll)i * i <= num; i++) {
    if (num % i == 0) {
      int e = 0;
      do {
        e++;
        num /= i;
      } while (num % i == 0);
      ll sum = 0, pow = 1;
      do {
        sum += pow;
        pow *= i;
      } while (e-- > 0);
      total *= sum;
    }
  }
  if (num > 1) total *= (1 + num);
  return total;
}
```

9.5 Euler Totient (Phi)

```
//counts coprimes to each number from 1 to n
vector<int> phi1(int n) {
  vector<int> phi(n + 1);
  for (int i = 0; i <= n; i++) phi[i] = i;
  for (int i = 2; i <= n; i++) {
      if (phi[i] == i) {
        for (int j = i; j <= n; j += i)
          phi[j] -= phi[j] / i;
      }
  }
  return phi;
}
```

9.6 Fibonacci

```
// Fibonacci in O(log n)
void fib(ll n, ll&x, ll&y){
  if(n==0){
    x = 0;
    y = 1;
    return ;
  }
  if(n&1){
    fib(n-1, y, x);
    y=(y+x)%MOD;
  } else {
    ll a, b;
    fib(n>>1, a, b);
    y = (a*a+b*b)%MOD;
    x = (a*b + a*(b-a+MOD))%MOD;
  }
}
// Usage
ll x, y;
fib(10, x, y);
cout << x << "␣" << y << endl;
// This will output 55 89
```

9.7 Matrix Exponentiation

```
struct Mat {
  int n, m;
  vector<vector<int>> a;
  Mat() { }
```

```
  Mat(int _n, int _m) {n = _n; m = _m; a.assign(n, vector<int>(m, 0)); }
  Mat(vector< vector<int> > v) { n = v.size(); m = n ? v[0].size() : 0;
      a = v; }
  inline void make_unit() {
    assert(n == m);
    for (int i = 0; i < n; i++)  {
      for (int j = 0; j < n; j++) a[i][j] = i == j;
    }
  }
  inline Mat operator + (const Mat &b) {
    assert(n == b.n && m == b.m);
    Mat ans = Mat(n, m);
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < m; j++) {
        ans.a[i][j] = (a[i][j] + b.a[i][j]) % mod;
      }
    }
    return ans;
  }
  inline Mat operator - (const Mat &b) {
    assert(n == b.n && m == b.m);
    Mat ans = Mat(n, m);
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < m; j++) {
        ans.a[i][j] = (a[i][j] - b.a[i][j] + mod) % mod;
      }
    }
    return ans;
  }
  inline Mat operator * (const Mat &b) {
    assert(m == b.n);
    Mat ans = Mat(n, b.m);
    for(int i = 0; i < n; i++) {
      for(int j = 0; j < b.m; j++) {
        for(int k = 0; k < m; k++) {
          ans.a[i][j] = (ans.a[i][j] + 1LL * a[i][k] * b.a[k][j] % mod)
              % mod;
        }
      }
    }
    return ans;
  }
  inline Mat pow(long long k) {
    assert(n == m);
    Mat ans(n, n), t = a; ans.make_unit();
    while (k) {
      if (k & 1) ans = ans * t;
      t = t * t;
      k >>= 1;
    }
    return ans;
  }
  inline Mat& operator += (const Mat& b) { return *this = (*this) + b; }
  inline Mat& operator -= (const Mat& b) { return *this = (*this) - b; }
  inline Mat& operator *= (const Mat& b) { return *this = (*this) * b; }
  inline bool operator == (const Mat& b) { return a == b.a; }
  inline bool operator != (const Mat& b) { return a != b.a; }
};

// Usage
// Mat a(n, n);
// Mat b(n, n);
// Mat c = a * b;
// Mat d = a + b;
// Mat e = a - b;
// Mat f = a.pow(k);
// a.a[i][j] = x;
```

9.8 Miller Rabin Deterministic

```
using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
```

```
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};


bool MillerRabin(ll n) {
    if (n < 2)
        return false;

    int r = 0;
    ll d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

9.9 Mobius

```
int mob[N];
void mobius() {
  mob[1] = 1;
  for (int i = 2; i < N; i++){
    mob[i]--;
    for (int j = i + i; j < N; j += i) {
      mob[j] -= mob[i];
    }
  }
}
```

9.10 Prefix Sum Phi

```
vector<ll> sieve(kMaxV + 1,0);
vector<ll> phi(kMaxV + 1,0);

void primes()
{
  phi[1]=1;
  vector<ll> pr;
  for(int i=2;i<kMaxV;i++){
    if(sieve[i]==0){
      sieve[i]=i;
      pr.pb(i);
      phi[i]=i-1;
    }
    for(auto p:pr){
      if(p>sieve[i]||i*p>=kMaxV)break;
      sieve[i*p]=p;
      phi[i*p]=(p==sieve[i]?p:p-1)*phi[i];
    }
  }
  for(int i=1;i<kMaxV;i++){
    phi[i]+=phi[i-1];
    phi[i]%=MOD;
  }
}

map<ll,ll> m;
ll PHI(ll a){
  if(a<kMaxV)return phi[a];
  if(m.count(a))return m[a];
  // if(a<3)return 1;
  m[a]=((((a%MOD)*((a+1)%MOD))%MOD)*inverse(2));
  m[a]%=MOD;
  long long i=2;
  while(i<=a){
      long long j=a/i;
      j=a/j;
      m[a]+=MOD;
      m[a]-=((j-i+1)*PHI(a/i))%MOD;
      m[a]%=MOD;
```

```
      i=j+1;
  }
  m[a]%=MOD;
  return m[a];
}
```

9.11 Sieve

```
const int kMaxV = 1e6;

int sieve[kMaxV + 1];

//stores some prime (not necessarly the minimum one)
void primes()
{
  for (int i = 4; i <= kMaxV; i += 2)
    sieve[i] = 2;
  for (int i = 3; i <= kMaxV / i; i += 2)
  {
    if (sieve[i])
      continue;
    for (int j = i * i; j <= kMaxV; j += i)
      sieve[j] = i;
    }
}

vector<int> PrimeFactors(int x)
{
  if (x == 1)
    return {};

  unordered_set<int> primes;
  while (sieve[x])
  {
    primes.insert(sieve[x]);
    x /= sieve[x];
  }
  primes.insert(x);
  return {primes.begin(), primes.end()};
}
```

9.12 Identities

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

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

$\sigma(n) = O(\log(\log(n)))$ (number of divisors of n)

$$F_{2n+1} = F_n^2 + F_{n+1}^2$$

$$F_{2n} = F_{n+1}^2 - F_{n-1}^2$$

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

$$F_{n+i}F_{n+j} - F_nF_{n+i+j} = (-1)^n F_i F_j$$

(Möbius Inv. Formula) $\mu(p^k) = [k = 0] - [k = 1]$ Let $g(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n} g(d)\mu\left(\frac{n}{d}\right)$.

(Dirichlet Convolution) Let f, g be arithmetic functions, then $(f * g)(n) = \sum_{d|n} f(d)g\left(\frac{n}{d}\right)$. If f, g are multiplicative, then so is $f * g$.

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

Lucas' Theorem: $\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p}$ where $m = \sum_{i=0}^{k} m_i p^i$ and $n = \sum_{i=0}^{k} n_i p^i$.

9.13 Burnside's Lemma

Dado un grupo G de permutaciones y un conjunto X de n elementos, el número de órbitas de X bajo la acción de G es igual al promedio del número de puntos fijos de las permutaciones en G.
Formalmente, el número de órbitas es $\frac{1}{|G|}\sum_{g\in G} f(g)$ donde $f(g)$ es el número de puntos fijos de g.
Ejemplo: Dado un collar con n cuentas y 2 colores, el número de collares diferentes que se pueden formar es $\frac{1}{n}\sum_{i=0}^{n} f(i)$ donde $f(i)$ es el número de collares que quedan fijos bajo una rotación de i posiciones.
Para contar el número de collares que quedan fijos bajo una rotación de i posiciones, se puede usar la fórmula $f(i) = 2^{\gcd(i,n)}$.
Para un collar de n cuentas y k colores, el número de collares diferentes que se pueden formar es $\frac{1}{n}\sum_{i=0}^{n} k^{\gcd(i,n)}$
Ejemplo: Dado un cubo con 6 caras y k colores, el número de cubos diferentes que se pueden formar es $\frac{1}{24}\sum_{i=0}^{24} f(i)$ donde $f(i)$ es el número de cubos que quedan fijos bajo una rotación de i posiciones. Esta formula es igual a $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$

9.14 Recursion

Sea $f(n) = \sum_{i=1}^{k} a_i f(n-i)$ entonces podemos considerar la matriz:

$$\begin{bmatrix} f(n) \\ f(n-1) \\ \vdots \\ f(n-k+1) \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \\ \vdots \\ f(n-k) \end{bmatrix}$$

De aqui podemos calcular $f(n)$ con exponenciación de matrices.

$$\begin{bmatrix} f(n) \\ f(n-1) \\ \vdots \\ f(n-k+1) \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}^{n-k} \begin{bmatrix} f(k) \\ f(k-1) \\ \vdots \\ f(1) \end{bmatrix}$$

9.15 Theorems

Koeing's Theorem: La cardinalidad del emparejamiento maximo de una grafica bipartita es igual al minimum vertex cover.
Hall's Theorem: Una grafica bipartita G tiene un emparejamiento que cubre todos los nodos de G si y solo si para todo subconjunto S de nodos de G, el número de vecinos de S es mayor o igual a $|S|$.
Kuratowski's Theorem: Una grafica es plana si y solo si no contiene un subgrafo homeomorfo a $K_{3,3}$ o K_5.

9.16 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c-1}, c \neq 1$$

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

9.17 Catalan numbers

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

$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2}C_n,\ C_{n+1} = \sum C_i C_{n-i}$$

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested. If prefix is given, number of ways is $\binom{n}{remaining_c losed}$-$\binom{n}{remaining_c losed+1}$.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

9.18 Cayley's formula

Number of labeled trees of n vertices is n^{n-2}. Number of rooted forest of n vertices is $(n+1)^{n-1}$.

9.19 Geometric series

Infinite
$a + ar + ar^2 + ar^3 + \ldots\ldots + \sum_{k=0}^{\infty} ar^k$
Sum=$\frac{a}{1-r}$
Finite
$a + ar + ar^2 + ar^3 + \ldots\ldots + \sum_{k=0}^{n} ar^k$
Sum=$\frac{a(1-r^{n+1})}{1-r}$

9.20 Estimates For Divisors

$\sum_{d|n} d = O(n \log\log n)$.
The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

9.21 Sum of divisors

$$\sum d|n = \frac{{p_1}^{\alpha_1+1}-1}{p_1-1} + \frac{{p_2}^{\alpha_2+1}-1}{p_2-1} + \ldots. + \frac{{p_n}^{\alpha_n+1}-1}{p_n-1}$$

9.22 Pythagorean Triplets

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2),\ \ b = k \cdot (2mn),\ \ c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

9.23 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor \frac{n!}{e} \right\rceil$$

10 Game Theory

10.1 Sprague-Grundy theorem

`https://codeforces.com/blog/entry/66040` Dado un juego con pilas p_1, p_2, ..., p_n sea $g(p)$ el nimber de la pila p, entonces el nimber del juego es $g(p_1) \oplus g(p_2) \oplus ... \oplus g(p_n)$. Para calcular el nimber de una pila, se puede usar la fórmula $g(r) = mex(\{g(r_1), g(r_2), ..., g(r_k)\})$ donde $r_1, r_2, ..., r_k$ son los posibles estados a los que se puede llegar desde r y $g(r) = 0$ si r es un estado perdedor.

11 Fórmulas y notas

11.1 Números de Stirling del primer tipo

$\begin{bmatrix} n \\ k \end{bmatrix}$ representa el número de permutaciones de n elementos en exactamente k ciclos disjuntos.

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1$$

$$\begin{bmatrix} 0 \\ n \end{bmatrix} = \begin{bmatrix} n \\ 0 \end{bmatrix} = 0 \qquad , \quad n > 0$$

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} \qquad , \quad k > 0$$

$$\sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix} = n!$$

$$\sum_{k=0}^{\infty} \begin{bmatrix} n \\ k \end{bmatrix} x^k = \prod_{k=0}^{n-1} (x+k)$$

11.2 Números de Stirling del segundo tipo

$\begin{Bmatrix} n \\ k \end{Bmatrix}$ representa el número de formas de particionar un conjunto de n objetos distinguibles en k subconjuntos no vacíos.

$$\begin{Bmatrix} 0 \\ 0 \end{Bmatrix} = 1$$

$$\begin{Bmatrix} 0 \\ n \end{Bmatrix} = \begin{Bmatrix} n \\ 0 \end{Bmatrix} = 0 \qquad , \quad n > 0$$

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = k\begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} \qquad , \quad k > 0$$

$$= \sum_{j=0}^{k} \frac{j^n}{j!} \cdot \frac{(-1)^{k-j}}{(k-j)!}$$

11.3 Números de Euler

$\left\langle\!\left\langle \begin{smallmatrix} n \\ k \end{smallmatrix} \right\rangle\!\right\rangle$ representa el número de permutaciones de 1 a n en donde exactamente k números son mayores que el número anterior, es decir, cuántas permutaciones tienen k "ascensos".

$$\left\langle\!\left\langle \begin{matrix} 1 \\ 0 \end{matrix} \right\rangle\!\right\rangle = 1$$

$$\left\langle\!\left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle\!\right\rangle = (n-k)\left\langle\!\left\langle \begin{matrix} n-1 \\ k-1 \end{matrix} \right\rangle\!\right\rangle + (k+1)\left\langle\!\left\langle \begin{matrix} n-1 \\ k \end{matrix} \right\rangle\!\right\rangle \qquad , \quad n \geq 2$$

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

$$\sum_{k=0}^{n-1} \left\langle\!\left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle\!\right\rangle = n!$$

11.4 Números de Catalan

$$C_0 = 1$$

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \sum_{j=0}^{n-1} C_j C_{n-1-j}$$

$$\sum_{n=0}^{\infty} C_n x^n = \frac{1-\sqrt{1-4x}}{2x}$$

11.5 Números de Bell

B_n representa el número de formas de particionar un conjunto de n elementos.

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\} = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k$$

$$\sum_{n=0}^{\infty} \frac{B_n}{n!} x^n = e^{e^x - 1}$$

11.6 Números de Bernoulli

$$B_0{}^+ = 1$$

$$B_n{}^+ = 1 - \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k{}^+}{n-k+1}$$

$$\sum_{n=0}^{\infty} \frac{B_n{}^+ x^n}{n!} = \frac{x}{1-e^{-x}} = \frac{1}{\frac{1}{1!} - \frac{x}{2!} + \frac{x^2}{3!} - \frac{x^3}{4!} + \cdots}$$

11.7 Fórmula de Faulhaber

$$S_p(n) = \sum_{k=1}^{n} k^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k{}^+ n^{p+1-k}$$

11.8 Función Beta

$$B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = 2\int_0^{\pi/2} \sin^{2x-1}(\theta)\cos^{2x-1}(\theta)d\theta$$
$$= \int_0^1 t^{x-1}(1-t)^{y-1}dt = \int_0^{\infty} \frac{t^{x-1}}{(1+t)^{x+y}}dt$$

11.9 Función zeta de Riemann

La siguiente fórmula converge rápido para valores pequeños de n ($n \approx 20$):

$$\zeta(s) \approx \frac{1}{d_0(1-2^{1-s})} \sum_{k=1}^{n} \frac{(-1)^{k-1} d_k}{k^s}$$

$$d_k = \sum_{j=k}^{n} \frac{4^j}{n+j} \binom{n+j}{2j}$$

11.10 Funciones generadoras

$$\sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k \right) x^n = \frac{1}{1-x} \sum_{n=0}^{\infty} a_n x^n$$

$$\sum_{n=0}^{\infty} \binom{n+k-1}{k-1} x^n = \frac{1}{(1-x)^k}$$

$$\sum_{n=0}^{\infty} p_n x^n = \frac{1}{\prod_{k=1}^{\infty}(1-x^k)} = \frac{1}{\sum_{n=-\infty}^{\infty} (-1)^n x^{\frac{1}{2}n(3n+1)}}$$

$$\sum_{p=0}^{\infty} \frac{S_p(n)}{p!} x^p = \frac{e^{x(n+1)} - e^x}{e^x - 1}$$

$$\sum_{n=0}^{\infty} n^k x^n = \frac{\sum_{i=0}^{k-1} \left\langle\!\left\langle {k \atop i} \right\rangle\!\right\rangle x^{i+1}}{(1-x)^{k+1}} \quad , \quad k \geq 1$$

Sean $a_1, a_2, \ldots, a_n$ números complejos. Sean $p_m = \sum_{i=1}^{n} a_i^m$ y s_m el m-ésimo polinomio elemental simétrico de $a_1, a_2, \ldots, a_n$. Entonces se cumple que $xS'(x) + P(x)S(x) = 0$, donde $P(x) = \sum_{m=1}^{\infty} p_m x^m$ y $S(x) = \prod_{i=1}^{n}(1 - a_i x) = \sum_{m=0}^{n} (-1)^m s_m x^m$.

11.11 Números armónicos

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \approx \ln(n) + \gamma + \frac{1}{2n} - \frac{1}{12n^2}$$

$$\gamma \approx 0.57721566490153286060606512$$

11.12 Aproximación de Stirling

$$\ln(n!) \approx n\ln(n) - n + \frac{1}{2}\ln(2\pi n)$$

$$\#\text{ de dígitos de } n! = 1 + \left\lfloor n \log\left(\frac{n}{e}\right) + \frac{1}{2}\log(2\pi n) \right\rfloor \quad (n \geq 30)$$

11.13 Ternas pitagóricas

- Una terna de enteros positivos (a, b, c) es pitagórica si $a^2 + b^2 = c^2$. Además es primitiva si $\gcd(a, b, c) = 1$.
- Generador de ternas primitivas:

$$\begin{aligned} a &= m^2 - n^2 \\ b &= 2mn \\ c &= m^2 + n^2 \end{aligned}$$

  donde $n \geq 1$, $m > n$, $\gcd(m, n) = 1$ y m, n tienen distinta paridad.
- Árbol de ternas pitagóricas primitivas: al multiplicar cualquiera de estas matrices:

$$\begin{pmatrix} 1 & -2 & 2 \\ 2 & -1 & 2 \\ 2 & -2 & 3 \end{pmatrix} \quad , \quad \begin{pmatrix} -1 & 2 & 2 \\ -2 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 3 \end{pmatrix}$$

  por una terna primitiva $\mathbf{v^T}$, obtenemos otra terna primitiva diferente. En particular, si empezamos con $\mathbf{v} = (3, 4, 5)$, podremos generar todas las ternas primitivas.

11.14 Árbol de Stern–Brocot

Todos los racionales positivos se pueden representar como un árbol binario de búsqueda completo infinito con raíz $\frac{1}{1}$.

- Dado un racional $q = [a_0; a_1, a_2, \ldots, a_k]$ donde $a_k \neq 1$, sus hijos serán $[a_0; a_1, a_2, \ldots, a_k + 1]$ y $[a_0; a_1, a_2, \ldots, a_k - 1, 2]$, y su padre será $[a_0; a_1, a_2, \ldots, a_k - 1]$.
- Para hallar el camino de la raíz $\frac{1}{1}$ a un racional q, se usa búsqueda binaria iniciando con $L = \frac{0}{1}$ y $R = \frac{1}{0}$. Para hallar M se supone que $L = \frac{a}{b}$ y $R = \frac{c}{d}$, entonces $M = \frac{a+c}{b+d}$.

11.15 Combinatoria

- Principio de las casillas: al colocar n objetos en k lugares hay al menos $\lceil \frac{n}{k} \rceil$ objetos en un mismo lugar.
- Número de funciones: sean A y B conjuntos con $m = |A|$ y $n = |B|$. Sea $f : A \to B$:
  - Si $m \leq n$, entonces hay $m!\binom{n}{m}$ funciones inyectivas f.
  - Si $m = n$, entonces hay $n!$ funciones biyectivas f.
  - Si $m \geq n$, entonces hay $n!\left\{ {m \atop n} \right\}$ funciones suprayectivas f.
- Barras y estrellas: ¿cuántas soluciones en los enteros no negativos tiene la ecuación $\sum_{i=1}^{k} x_i = n$? Tiene $\binom{n+k-1}{k-1}$ soluciones.
- ¿Cuántas soluciones en los enteros positivos tiene la ecuación $\sum_{i=1}^{k} x_i = n$? Tiene $\binom{n-1}{k-1}$ soluciones.
- Desordenamientos: $a_0 = 1$, $a_1 = 0$, $a_n = (n-1)(a_{n-1} + a_{n-2}) = na_{n-1} + (-1)^n$.
- Sea $f(x)$ una función. Sea $g_n(x) = xg'_{n-1}(x)$ con $g_0(x) = f(x)$. Entonces $g_n(x) = \sum_{k=0}^{n} \left\{ {n \atop k} \right\} x^k f^{(k)}(x)$.
- Supongamos que tenemos $m + 1$ puntos: $(0, y_0)$, $(1, y_1)$, …, (m, y_m). Entonces el polinomio $P(x)$ de grado m que pasa por todos ellos es:

$$P(x) = \left[\prod_{i=0}^{m} (x - i) \right] (-1)^m \sum_{i=0}^{m} \frac{y_i (-1)^i}{(x - i) i! (m - i)!}$$

- Sea $a_0, a_1, \ldots$ una recurrencia lineal homogénea de grado d dada por $a_n = \sum_{i=1}^{d} b_i a_{n-i}$ para $n \geq d$ con términos iniciales $a_0, a_1, \ldots, a_{d-1}$. Sean $A(x)$ y $B(x)$ las funciones generadoras de las sucesiones a_n y b_n respectivamente, entonces se cumple que $A(x) = \dfrac{A_0(x)}{1 - B(x)}$, donde $A_0(x) = \sum_{i=0}^{d-1} \left[a_i - \sum_{j=0}^{i-1} a_j b_{i-j} \right] x^i$.
- Si queremos obtener otra recurrencia c_n tal que $c_n = a_{kn}$, las raíces del polinomio característico de c_n se obtienen al elevar todas las raíces del polinomio característico de a_n a la k-ésima potencia; y sus términos iniciales serán $a_0, a_k, \ldots, a_{k(d-1)}$.

11.16 Grafos

- Sea d_n el número de grafos con n vértices etiquetados: $d_n = 2^{\binom{n}{2}}$.
- Sea c_n el número de grafos conexos con n vértices etiquetados. Tenemos la recurrencia: $c_1 = 1$ y $d_n = \sum_{k=1}^{n} \binom{n-1}{k-1} c_k d_{n-k}$. También se cumple, usando funciones generadoras exponenciales, que $C(x) = 1 + \ln(D(x))$.

- Sea t_n el número de torneos fuertemente conexos en n nodos etiquetados. Tenemos la recurrencia $t_1 = 1$ y $d_n = \sum_{k=1}^{n} \binom{n}{k} t_k d_{n-k}$. Usando funciones generadoras exponenciales, tenemos que $T(x) = 1 - \frac{1}{D(x)}$.
- Número de spanning trees en un grafo completo con n vértices etiquetados: n^{n-2}.
- Número de bosques etiquetados con n vértices y k componentes conexas: kn^{n-k-1}.
- Para un grafo no dirigido simple G con n vértices etiquetados de 1 a n, sea $Q = D - A$, donde D es la matriz diagonal de los grados de cada nodo de G y A es la matriz de adyacencia de G. Entonces el número de spanning trees de G es igual a cualquier cofactor de Q.
- Sea G un grafo. Se define al polinomio $P_G(x)$ como el polinomio cromático de G, en donde $P_G(k)$ nos dice cuántas k-coloraciones de los vértices admite G. Ejemplos comunes:
  - Grafo completo de n nodos: $P(x) = x(x-1)(x-2)\dots(x-(n-1))$
  - Grafo vacío de n nodos: $P(x) = x^n$
  - Árbol de n nodos: $P(x) = x(x-1)^{n-1}$
  - Ciclo de n nodos: $P(x) = (x-1)^n + (-1)^n(x-1)$

11.17 Teoría de números

$$(f * e)(n) = f(n)$$
$$(\varphi * \mathbf{1})(n) = n$$
$$(\mu * \mathbf{1})(n) = e(n)$$
$$\varphi(n^k) = n^{k-1}\varphi(n)$$
$$\sum_{\substack{k=1 \\ \gcd(k,n)=1}}^{n} k = \frac{n\varphi(n)}{2} \quad , \quad n \geq 2$$
$$\sum_{k=1}^{n} \operatorname{lcm}(k,n) = \frac{n}{2} + \frac{n}{2}\sum_{d|n} d\varphi(d) = \frac{n}{2} + \frac{n}{2}\prod_{p^a|n} \frac{p^{2a+1}+1}{p+1}$$
$$\sum_{k=1}^{n} \gcd(k,n) = \sum_{d|n} d\varphi\left(\frac{n}{d}\right) = \prod_{p^a|n} p^{a-1}(1+(a+1)(p-1))$$

- Lifting the exponent: sea p un primo, x, y enteros y n un entero positivo tal que $p \mid x-y$ pero $p \nmid x$ ni $p \nmid y$. Entonces:
  - Si p es impar: $v_p(x^n - y^n) = v_p(x-y) + v_p(n)$
  - Si $p = 2$ y n es par: $v_p(x^n - y^n) = v_p(x-y) + v_p(n) + v_p(x+y) - 1$

  donde $v_p(n)$ es el exponente de p en la factorización en primos de n.
- Suma de dos cuadrados: sea $\chi_4(n)$ una función multiplicativa igual a 1 si $n \equiv 1 \mod 4$, -1 si $n \equiv 3 \mod 4$ y cero en otro caso. Entonces, el número de soluciones enteras (a,b) de la ecuación $a^2 + b^2 = n$ es $4(\chi_4 * 1)(n) = 4\sum_{d|n} \chi_4(d)$.
- Teorema de Lucas:
$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{k_i} \pmod{p}$$
$$m = \sum_{i=0}^{k} m_i p^i \quad , \quad n = \sum_{i=0}^{k} n_i p^i$$
$$0 \leq m_i, n_i < p$$
- Sean $a, b, c \in \mathbb{Z}$ con $a \neq 0$ y $b \neq 0$. La ecuación $ax + by = c$ tiene como soluciones:
$$x = \frac{x_0 c - bk}{d}$$
$$y = \frac{y_0 c + ak}{d}$$
para toda $k \in \mathbb{Z}$ si y solo si $d|c$, donde $ax_0 + by_0 = \gcd(a,b) = d$ (Euclides extendido). Si a y b tienen el mismo signo, hay exactamente $\max\left(\left\lfloor \frac{x_0 c}{|b|} \right\rfloor + \left\lfloor \frac{y_0 c}{|a|} \right\rfloor + 1, 0\right)$ soluciones no negativas. Si tienen el signo distinto, hay infinitas soluciones no negativas.
- Dada una función aritmética f con $f(1) \neq 0$, existe otra función aritmética g tal que $(f * g)(n) = e(n)$, dada por:
$$g(1) = \frac{1}{f(1)}$$
$$g(n) = -\frac{1}{f(1)} \sum_{d|n, d<n} f\left(\frac{n}{d}\right) g(d) \quad , \quad n > 1$$
- Sean $h(n) = \sum_{k=1}^{n} f\left(\left\lfloor \frac{n}{k} \right\rfloor\right) g(k)$, $G(n) = \sum_{k=1}^{n} g(k)$ y $m = \lfloor \sqrt{n} \rfloor$, entonces:
$$h(n) = \sum_{k=1}^{\lfloor n/m \rfloor} f\left(\left\lfloor \frac{n}{k} \right\rfloor\right) g(k) + \sum_{k=1}^{m-1} \left(G\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - G\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right)\right) f(k)$$

- Sean $F(n) = \sum_{k=1}^{n} f(k)$, $G(n) = \sum_{k=1}^{n} g(k)$, $h(n) = (f * g)(n) = \sum_{d|n} f(d)g\left(\frac{n}{d}\right)$ y $H(n) = \sum_{k=1}^{n} h(k)$, entonces:

$$H(n) = \sum_{k=1}^{n} f(k)G\left(\left\lfloor \frac{n}{k} \right\rfloor\right)$$

- Sean $\Phi_p(n) = \sum_{k=1}^{n} k^p \varphi(k)$ y $M_p(n) = \sum_{k=1}^{n} k^p \mu(k)$. Aplicando lo anterior, podemos calcular $\Phi_p(n)$ y $M_p(n)$ con complejidad $O(n^{2/3})$ si precalculamos con fuerza bruta los primeros $\lfloor n^{2/3} \rfloor$ valores, y para los demás, usamos las siguientes recurrencias (DP con `map`):

$$\Phi_p(n) = S_{p+1}(n) - \sum_{k=2}^{\lfloor n/m \rfloor} k^p \Phi_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - \sum_{k=1}^{m-1} \left(S_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - S_p\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right)\right) \Phi_p(k)$$

$$M_p(n) = 1 - \sum_{k=2}^{\lfloor n/m \rfloor} k^p M_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - \sum_{k=1}^{m-1} \left(S_p\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - S_p\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right)\right) M_p(k)$$

- En general, si queremos hallar $F(n)$ y existe una función mágica $g(n)$ tal que $G(n)$ y $H(n)$ se puedan calcular en $O(1)$, entonces:

$$F(n) = \frac{1}{g(1)}\left[H(n) - \sum_{k=2}^{\lfloor n/m \rfloor} g(k)F\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - \sum_{k=1}^{m-1} \left(G\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - G\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right)\right) F(k)\right]$$

11.18 Primos

10^2+1, 10^3+9, 10^4+7, 10^5+3, 10^6+3, 10^7+19, 10^8+7, 10^9+7, $10^{10}+19$, $10^{11}+3$, $10^{12}+39$, $10^{13}+37$, $10^{14}+31$, $10^{15}+37$, $10^{16}+61$, $10^{17}+3$, $10^{18}+3$.
10^2-3, 10^3-3, 10^4-27, 10^5-9, 10^6-17, 10^7-9, 10^8-11, 10^9-63, $10^{10}-33$, $10^{11}-23$, $10^{12}-11$, $10^{13}-29$, $10^{14}-27$, $10^{15}-11$, $10^{16}-63$, $10^{17}-3$, $10^{18}-11$.

11.19 Números primos de Mersenne

Números primos de la forma $M_p = 2^p - 1$ con p primo. Todos los números perfectos pares son de la forma $2^{p-1}M_p$ y viceversa.
Los primeros 47 valores de p son: 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951, 30402457, 32582657, 37156667, 42643801, 43112609.

11.20 Números primos de Fermat

Números primos de la forma $F_p = 2^{2^p} + 1$, solo se conocen cinco: 3, 5, 17, 257, 65537.
Un polígono de n lados es construible si y solo si n es el producto de algunas potencias de dos y distintos primos de Fermat.

12 More Topics

12.1 2D Prefix Sum

```
int b[MAXN][MAXN];
int a[MAXN][MAXN];
for (int i = 1; i <= N; i++) {
  for (int j = 1; j <= N; j++) {
    b[i][j] = a[i][j] + b[i - 1][j] +
              b[i][j - 1] - b[i - 1][j - 1];
  }
}
for (int q = 0; q < Q; q++) {
  int from_row, to_row, from_col, to_col;
  cin >> from_row >> from_col >> to_row >> to_col;
  cout << b[to_row][to_col] - b[from_row - 1][to_col] -
            b[to_row][from_col - 1] +
            b[from_row - 1][from_col - 1]
       << '\n';
}
```

12.2 Custom Comparators

```
bool cmp(const Edge &x, const Edge &y) { return x.w < y.w; }

sort(a.begin(), a.end(), cmp);

set<int, greater<int>> a;
map<int, string, greater<int>> b;
priority_queue<int, vector<int>, greater<int>> c;
```

12.3 Day of the Week

```
int dayOfWeek(int d, int m, lli y){
  if(m == 1 || m == 2){
    m += 12;
    --y;
```

```
  }
  int k = y % 100;
  lli j = y / 100;
  return (d + 13*(m+1)/5 + k + k/4 + j/4 + 5*j) % 7;
}
```

12.4 Directed MST

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

const int N = 3e5 + 9;

const long long inf = 1e18;

template<typename T> struct PQ {
  long long sum = 0;
  priority_queue<T, vector<T>, greater<T>> Q;
  void push(T x) { x.w -= sum; Q.push(x); }
  T pop() { auto ans = Q.top(); Q.pop(); ans.w += sum; return ans; }
  int size() { return Q.size(); }
  void add(long long x) { sum += x; }
  void merge(PQ &&x) {
    if (size() < x.size()) swap(sum, x.sum), swap(Q, x.Q);
    while (x.size()) {
      auto tmp = x.pop();
      tmp.w -= sum;
      Q.push(tmp);
    }
  }
};
struct edge {
  int u, v; long long w;
  bool operator > (const edge &rhs) const { return w > rhs.w; }
};
struct DSU {
  vector<int> par;
  DSU (int n) : par(n, -1) {}
  int root(int i) { return par[i] < 0 ? i : par[i] = root(par[i]); }
  void set_par(int c, int p) { par[c] = p; }
};
// returns parents of each vertex
// each edge should be distinct
// it assumes that a solution exists (all vertices are reachable from
    root)
// 0 indexed
// Takes ~300ms for n = 2e5
vector<int> DMST(int n, int root, const vector<edge> &edges) {
  vector<int> u(2 * n - 1, -1), par(2 * n - 1, -1);
  edge par_edge[2 * n - 1];
  vector<int> child[2 * n - 1];

  PQ<edge> Q[2 * n - 1];
  for (auto e : edges) Q[e.v].push(e);
  for (int i = 0; i < n; i++) Q[(i + 1) % n].push({i, (i + 1) % n, inf})
      ;

  int super = n;
  DSU dsu(2 * n - 1);
  int head = 0;
  while (Q[head].size()) {
    auto x = Q[head].pop();
    int nxt_root = dsu.root(x.u);
    if (nxt_root == head) continue;
    u[head] = nxt_root;
    par_edge[head] = x;
    if (u[nxt_root] == -1) head = nxt_root;
    else {
      int j = nxt_root;
      do {
        Q[j].add(-par_edge[j].w);
        Q[super].merge(move(Q[j]));
        assert(u[j] != -1);
        child[super].push_back(j);
        j = dsu.root(u[j]);
      } while (j != nxt_root);
      for (auto u : child[super]) par[u] = super, dsu.set_par(u, super);
      head = super++;
    }
  }
  vector<int> res(2 * n - 1, -1);
  queue<int> q; q.push(root);
  while (q.size()) {
    int u = q.front();
    q.pop();
    while (par[u] != -1) {
```

```
      for (auto v : child[par[u]]) {
        if (v != u) {
          res[par_edge[v].v] = par_edge[v].u;
          q.push(par_edge[v].v);
          par[v] = -1;
        }
      }
      u = par[u];
    }
  }
  res[root] = root; res.resize(n);
  return res;
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, m, root; cin >> n >> m >> root;
  vector<edge> edges(m);
  for (auto &e : edges) cin >> e.u >> e.v >> e.w;
  auto res = DMST(n, root, edges);

  unordered_map<int, int> W[n];
  for (auto u : edges) W[u.v][u.u] = u.w;

  long long ans = 0;
  for (int i = 0; i < n; i++) if (i != root) ans += W[i][res[i]];
  cout << ans << '\n';
  for (auto x : res) cout << x << ' '; cout << '\n';
  return 0;
}
// https://judge.yosupo.jp/problem/directedmst
// http://www.cs.tau.ac.il/~zwick/grad-algo-13/directed-mst.pdf
```

12.5 GCD Convolution

```
/*
  GCD Convolution via Multiple Zeta and Mobius Transforms
  -------------------------------------------------------
  Given two sequences A[0..n] and B[0..n], computes C where
    C[d] = sum_{i, j: gcd(i, j) = d} A[i] * B[j]
  Indexing: 1-based, vectors of size n+1 (ignore index 0)
  Bounds:
    - i, j, d in [1..n]
  Time Complexity: O(n log log n + n log n) per convolution
  Space Complexity: O(n)
  Steps:
    1. Multiple Zeta Transform: for each prime p, v[i] += v[i*p]
    2. Pointwise multiply transformed A and B
    3. Multiple Mobius Transform: for each prime p, v[i] -= v[i*p]
*/
// Enumerate primes up to n in O(n)
vector<int> PrimeEnumerate(int n) {
  vector<bool> isPrime(n+1, true);
  vector<int> P;
  for (int i = 2; i <= n; i++) {
    if (isPrime[i]) P.push_back(i);
    for (int p : P) {
      if (i * p > n) break;
      isPrime[i*p] = false;
      if (i % p == 0) break;
    }
  }
  return P;
}
// Multiple Zeta Transform (over divisibility poset)
template<typename T>
void MultipleZetaTransform(vector<T>& v) {
  int n = (int)v.size() - 1;
  for (int p : PrimeEnumerate(n)) {
    for (int i = n/p; i >= 1; i--) {
      v[i] += v[i * p];
    }
  }
}
// Multiple Mobius Transform (inverse of zeta)
template<typename T>
void MultipleMobiusTransform(vector<T>& v) {
  int n = (int)v.size() - 1;
  for (int p : PrimeEnumerate(n)) {
    for (int i = 1; i * p <= n; i++) {
      v[i] -= v[i * p];
    }
  }
}
// GCD convolution of A and B, both size n+1
template<typename T>
```

```cpp
vector<T> GCDConvolution(vector<T> A, vector<T> B) {
  int n = (int)A.size() - 1;
  MultipleZetaTransform(A);
  MultipleZetaTransform(B);
  for (int i = 1; i <= n; i++) {
    A[i] *= B[i];  // pointwise multiplication
  }
  MultipleMobiusTransform(A);
  return A;  // result C[0..n], where C[d] = sum_{gcd(i,j)=d} A[i]*B[j]
}
// Example usage
vector<long long> A(n+1), B(n+1);
for (int i = 1; i <= n; i++) cin >> A[i];
for (int i = 1; i <= n; i++) cin >> B[i];
auto C = GCDConvolution<long long>(A, B);
for (int d = 1; d <= n; d++){
  cout << C[d] << (d==n?'\n':' ');
}
```

12.6 int128

```cpp
//cout for __int128
ostream &operator<<(ostream &os, const __int128 & value){
  char buffer[64];
  char *pos = end(buffer) - 1;
  *pos = '\0';
  __int128 tmp = value < 0 ? -value : value;
  do{
    --pos;
    *pos = tmp % 10 + '0';
    tmp /= 10;
  }while(tmp != 0);
  if(value < 0){
    --pos;
    *pos = '-';
  }
  return os << pos;
}

//cin for __int128
istream &operator>>(istream &is, __int128 & value){
  char buffer[64];
  is >> buffer;
  char *pos = begin(buffer);
  int sgn = 1;
  value = 0;
  if(*pos == '-'){
    sgn = -1;
    ++pos;
  }else if(*pos == '+'){
    ++pos;
  }
  while(*pos != '\0'){
    value = (value << 3) + (value << 1) + (*pos - '0');
    ++pos;
  }
  value *= sgn;
  return is;
}


ll mult(__int128 a, __int128 b){ return ((a*1LL*b)%MOD + MOD)%MOD; }
```

12.7 Iterating Over All Subsets

```cpp
for (int mk = 0; mk < (1 << k); mk++) {
  Ap[mk] = 0;
  for (int s = mk;; s = (s - 1) & mk) {
    Ap[mk] += A[s];
    if (!s) break;
  }
}
```

12.8 LCM Convolution

```cpp
/*
  LCM Convolution via Divisor Zeta and Mobius Transforms
  -----------------------------------------------------
  Given two sequences A[1..n] and B[1..n], computes C where
    C[d] = sum_{i, j: lcm(i, j) = d} A[i] * B[j]
  Indexing: 1-based (vectors of size n+1, ignore index 0)
  Bounds:
    - i, j, d in [1..n]
  Time Complexity: O(n log log n + n log n)
  Space Complexity: O(n)
  Steps:
    1. Divisor Zeta Transform: for each prime p, v[i*p] += v[i]
```

```
    2. Pointwise multiply transformed A and B
    3. Divisor Mobius Transform: for each prime p, v[i*p] -= v[i]
*/
// Sieve primes up to n in O(n)
vector<int> PrimeEnumerate(int n) {
  vector<bool> isPrime(n+1, true);
  vector<int> P;
  for (int i = 2; i <= n; i++) {
    if (isPrime[i]) P.push_back(i);
    for (int p : P) {
      if (i * p > n) break;
      isPrime[i*p] = false;
      if (i % p == 0) break;
    }
  }
  return P;
}
// Divisor Zeta Transform: v[d] = sum_{d|k} original_v[k]
template<typename T>
void DivisorZetaTransform(vector<T>& v) {
  int n = (int)v.size() - 1;
  auto primes = PrimeEnumerate(n);
  for (int p : primes) {
    for (int i = 1; i * p <= n; i++) {
      v[i * p] += v[i];
    }
  }
}
// Divisor Mobius Transform (inverse of zeta)
template<typename T>
void DivisorMobiusTransform(vector<T>& v) {
  int n = (int)v.size() - 1;
  auto primes = PrimeEnumerate(n);
  for (int p : primes) {
    for (int i = n / p; i >= 1; i--) {
      v[i * p] -= v[i];
    }
  }
}
// LCM convolution of A and B (each size n+1)
template<typename T>
vector<T> LCMConvolution(vector<T> A, vector<T> B) {
  int n = (int)A.size() - 1;
  DivisorZetaTransform(A);
  DivisorZetaTransform(B);
  for (int d = 1; d <= n; d++) {
    A[d] *= B[d];  // pointwise multiplication
  }
  DivisorMobiusTransform(A);
  return A;  // result C[1..n], C[d] = sum_{lcm(i,j)=d} A[i]*B[j]
}
// Example usage
vector<long long> A(n+1), B(n+1);
for (int i = 1; i <= n; i++) cin >> A[i];
for (int i = 1; i <= n; i++) cin >> B[i];
auto C = LCMConvolution<long long>(A, B);
for (int d = 1; d <= n; d++){
  cout << C[d] << (d==n?'\n':' ');
}
```

12.9 Manhattan MST

```
/*
  Manhattan MST Edge Generation
  -----------------------------
  Given n points ps[0..n-1], returns a list of candidate edges
    (weight, u, v) for the Manhattan MST, where
    weight = |x[u]-x[v]| + |y[u]-y[v]|.
  Indexing: 0-based for points and returned edges
  Time Complexity: O(n log n)
*/
struct Point {
  ll x, y;
};

vector<tuple<ll,int,int>> manhattan_mst_edges(vector<Point> ps) {
  int n = ps.size();
  vector<int> order(n);
  iota(order.begin(), order.end(), 0);
  vector<tuple<ll,int,int>> edges;
  edges.reserve(n * 4);
  // Repeat for 4 orientations
  for (int r = 0; r < 4; r++) {
    // Sort by x+y ascending
    sort(order.begin(), order.end(), [&](int i, int j) {
      return ps[i].x + ps[i].y < ps[j].x + ps[j].y;
```

```
    });
    // Active map: key = x-coordinate, value = point index
    map<ll,int,greater<ll>> active;
    for (int i : order) {
      // For all active points with x >= ps[i].x
      for (auto it = active.lower_bound(ps[i].x); it != active.end();
          active.erase(it++)) {
        int j = it->second;
        if (ps[i].x - ps[i].y > ps[j].x - ps[j].y) break;
        assert(ps[i].x >= ps[j].x && ps[i].y >= ps[j].y);
        edges.emplace_back({(ps[i].x - ps[j].x) + (ps[i].y - ps[j].y), i
            , j});
      }
      active[ps[i].x] = i;
    }
    // Rotate points 90 degrees: (x,y) -> (y,-x)
    for (auto &p : ps) {
      ll tx = p.x;
      p.x = p.y;
      p.y = -tx;
    }
  }
  return edges;
}
```

12.10 Max Manhattan Distance

```
/*
  Max Manhattan distance
  ----------------------------------
  Finds maximum manhattan distance for any two points with d dimensions.
  Generalized code for d dimensions.
  Time Complexity: (n * 2 ^ d * d)
*/


long long ans = 0;
for (int msk = 0; msk < (1 << d); msk++) {
    long long mx = LLONG_MIN, mn = LLONG_MAX;
    for (int i = 0; i < n; i++) {
        long long cur = 0;
        for (int j = 0; j < d; j++) {
            if (msk & (1 << j)) cur += p[i][j];
            else cur -= p[i][j];
        }
        mx = max(mx, cur);
        mn = min(mn, cur);
    }
    ans = max(ans, mx - mn);
}
```

12.11 Mo

```
/*
  Mo's Algorithm (Sqrt Decomposition for Offline Queries)
  ----------------------------------------------------------
  Problem: Answer q range queries [L, R] over array of length n
           using add/remove operations efficiently
  Indexing: 0-based
  Bounds:
    - arr[0..n-1]
    - queries input as 1-based and converted to 0-based
  Time Complexity: O((n + q) * sqrt(n)) per query batch
  Space Complexity: O(n + q)
  Usage:
    - Fill in logic for add/remove operations (update cur)
    - Fill answers[] indexed by original query order
*/

const int MAXN = 200500;
ll n, q;
ll arr[MAXN];      // input array
ll cnt[1000005];   // frequency count
ll answers[MAXN];  // output array
ll cur = 0;        // current query result
ll BLOCK_SIZE;

pair< pair<ll, ll>, ll> queries[MAXN]; // {{L, R}, query_index}

// Sort by block and then by R
inline bool cmp(const pair< pair<ll, ll>, ll> &x, const pair< pair<ll,
    ll>, ll> &y) {
  ll block_x = x.first.first / BLOCK_SIZE;
  ll block_y = y.first.first / BLOCK_SIZE;
  if (block_x != block_y) return block_x < block_y;
  return x.first.second < y.first.second;
```

```
}

int main() {
  cin >> n >> q;
  BLOCK_SIZE = sqrt(n);
  for (int i = 0; i < n; i++) cin >> arr[i];
  for (int i = 0; i < q; i++) {
    int l, r;
    cin >> l >> r;
    --l; --r; // convert to 0-based
    queries[i] = {{l, r}, i};
  }
  sort(queries, queries + q, cmp);
  ll l = 0, r = -1;
  for (int i = 0; i < q; i++) {
    int left = queries[i].first.first;
    int right = queries[i].first.second;
    // Expand to right
    while (r < right) {
      r++;
      // Operations to add arr[r], implement exactly here
    }
    // Shrink from right
    while (r > right) {
      // Operations to remove arr[r], implement exactly here
      r--;
    }
    // Expand to left
    while (l < left) {
      // Operations to remove arr[l], implement exactly here
      l++;
    }
    // Shrink from left
    while (l > left) {
      l--;
      // Operations to add arr[l], implement exactly here
    }
    answers[queries[i].second] = cur; // Current answer
  }
  for (int i = 0; i < q; i++) cout << answers[i] << '\n';
}
```

12.12 MOD INT

```
template<int MOD>
struct ModInt {
  ll v;
  ModInt(ll _v = 0) {v = (-MOD < _v && _v < MOD) ? _v : _v % MOD; if (v < 0) v += MOD;}
  ModInt& operator += (const ModInt &other) {v += other.v; if (v >= MOD) v -= MOD; return *this;}
  ModInt& operator -= (const ModInt &other) {v -= other.v; if (v < 0) v += MOD; return *this;}
  ModInt& operator *= (const ModInt &other) {v = v * other.v % MOD; return *this;}
  ModInt& operator /= (const ModInt &other) {return *this *= inverse(other);}
  bool operator == (const ModInt &other) const {return v == other.v;}
  bool operator != (const ModInt &other) const {return v != other.v;}
  friend ModInt operator + (ModInt a, const ModInt &b) {return a += b;}
  friend ModInt operator - (ModInt a, const ModInt &b) {return a -= b;}
  friend ModInt operator * (ModInt a, const ModInt &b) {return a *= b;}
  friend ModInt operator / (ModInt a, const ModInt &b) {return a /= b;}
  friend ModInt operator - (const ModInt &a) {return 0 - a;}
  friend ModInt power(ModInt a, ll b) {ModInt ret(1); while (b > 0) {if (b & 1) ret *= a; a *= a; b >>= 1;} return ret;}
  friend ModInt inverse(ModInt a) {return power(a, MOD - 2);}
  friend istream& operator >> (istream &is, ModInt &m) {is >> m.v; m.v = (-MOD < m.v && m.v < MOD) ? m.v : m.v % MOD; if (m.v < 0) m.v += MOD; return is;}
  friend ostream& operator << (ostream &os, const ModInt &m) {return os << m.v;}
};
```

12.13 Next Permutation

```
sort(v.begin(),v.end());
while(next_permutation(v.begin(),v.end())){
  for(auto u:v){
    cout<<u<<"␣";
  }
  cout<<endl;
}

string s="asdfassd";
sort(s.begin(),s.end());
while(next_permutation(s.begin(),s.end())){
```

```
  cout<<s<<endl;
}
```

12.14 Next and Previous Smaller/Greater Element

```
/*
  Next and Previous Smaller / Greater Elements
  --------------------------------
  Given: array a[0..n-1]
  nextSmaller[i]: index of next element smaller than a[i], or n if none
  prevSmaller[i]: index of previous element smaller than a[i], or -1 if
      none
  For GREATER, replace a[s.top()] < a[i] with a[s.top()] > a[i]
  Indexing: 0-based
*/
vector<int> nextSmaller(vector<int> a, int n) {
  stack<int> s;
  vector<int> res(n, n); // default: no smaller to right
  for (int i = 0; i < n; i++) {
    while (!s.empty() && a[s.top()] > a[i]) {
      res[s.top()] = i;
      s.pop();
    }
    s.push(i);
  }
  return res;
}
vector<int> prevSmaller(vector<int> a, int n) {
  stack<int> s;
  vector<int> res(n, -1); // default: no smaller to left
  for (int i = n - 1; i >= 0; i--) {
    while (!s.empty() && a[s.top()] > a[i]) {
      res[s.top()] = i;
      s.pop();
    }
    s.push(i);
  }
  return res;
}
```

12.15 Parallel Binary Search

```
/*
  Parallel Binary Search
  ----------------------
  Solves: k independent queries where each answer lies in range [lo, hi]
  - At each iteration, test the midpoint for a batch of queries
  Indexing: 0-based
  Bounds:
    - i in [0, m-1] for the structure version
    - q in [0, k-1] for each of the k queries
  Time Complexity: O((m + k) * log M)
  Space Complexity: O(k + m)
  You must:
    - Reset data structures before each iteration
    - Implement apply_change(i) and check_query(q)
*/
int lo[MAXN], hi[MAXN];     // binary search bounds for each query
vector<int> tocheck[MAXN];  // tocheck[mid] = list of queries to check
    at mid

bool done = false;
while (!done) {
  done = true;
   // Reset changes of structure to 0 before each iteration
  // Assign queries to current midpoint
  for (int q = 0; q < k; q++) {
    if (lo[q] < hi[q]) {
      tocheck[(lo[q] + hi[q]) / 2].push_back(q);
    }
  }
  // Apply changes and evaluate queries
  for (int i = 0; i < m; i++) {
    // Apply change for ith query to your data structure
    apply_change(i);
    for (int q : tocheck[i]) {
      done = false; // at least one query is not done
      // Evaluate query q using the updated structure
      if (operationToCheck(q)) {
        hi[q] = i; // condition is true, try earlier
      } else {
        lo[q] = i + 1; // condition false, try later
      }
    }
  }
}
```

12.16 Random Number Generators

```cpp
//to avoid hacks
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
//you can also just write seed_value if hacks are not an issue

// rng() for generating random numbers between 0 and 2<<31-1

// for generating numbers with uniform probability in range
uniform_int_distribution<int>(0, n)(rng)
std::normal_distribution<> normal_dist(mean, 2)
exponential_distribution

// for shuffling array
shuffle(permutation.begin(), permutation.end(), rng);
```

12.17 setprecision

```cpp
cout<<fixed<<setprecision(10);
```

12.18 Ternary Search

```cpp
/*
  Ternary Search (for continuous unimodal functions)
  ---------------------------------------------------
  Finds the maximum (or minimum) value of a unimodal function f(x) in [l
      , r]
  Indexing: continuous domain (double)
  Bounds:
    - Search range is [l, r]
    - Precision controlled by eps
  Time Complexity: O(log((r - l) / eps))
  - For minimum, reverse the condition (if f1 > f2 then l = m1)
*/
double ternary_search(double l, double r) {
  double eps = 1e-9; // precision error
  while (r - l > eps) {
    double m1 = l + (r - l) / 3;
    double m2 = r - (r - l) / 3;
    double f1 = f(m1); // evaluate function at m1
    double f2 = f(m2); // evaluate function at m2
    if (f1 < f2)
      l = m1;  // move left bound up if increasing
    else
      r = m2;  // move right bound down if decreasing
  }
  return f(l); // return value at the best point
}
```

12.19 Ternary Search Int

```cpp
int lo = -1, hi = n;
while (hi - lo > 1){
    int mid = (hi + lo)>>1;
    if (f(mid) > f(mid + 1)) hi = mid;
    else lo = mid;
}
//lo + 1 is the answer
```

12.20 XOR Convolution

```cpp
/*
  Fast Walsh-Hadamard Transform (FWHT) for XOR Convolution
  ---------------------------------------------------------
  Given two arrays A[0..N-1], B[0..N-1], with N = 1<<k (power of two).
  Computes C[d] = sum_{i xor j = d} A[i] * B[j].
  Indexing: 0-based
  Bounds:
    - N must be a power of two
  Time Complexity: O(N * log(N))
  Steps:
    1. fwht(A, false); fwht(B, false); // forward transform
    2. for i in [0..N): C[i] = A[i] * B[i]
    3. fwht(C, true);                  // inverse transform
*/
void FWHT (int A[], int k, int inv) {
  for (int j = 0; j < k; j++)
    for (int i = 0; i < (1 << k); i++)
      if (~i & (1 << j)) {
        int p0 = A[i];
        int p1 = A[i | (1 << j)];
        A[i] = p0 + p1;
        A[i | (1 << j)] = p0 - p1;
        if (inv) {
          A[i] /= 2;
          A[i | (1 << j)] /= 2;
        }
```

```
    }
}
void XOR_conv (int A[], int B[], int C[], int k) {
  FWHT(A, k, false);
  FWHT(B, k, false);
  for (int i = 0; i < (1 << k); i++)
    C[i] = A[i] * B[i];
  FWHT(A, k, true);
  FWHT(B, k, true);
  FWHT(C, k, true);
}
// Example usage:
int A[1 << 20], B[1 << 20], C[1 << 20];
XOR_conv(A, B, C, 20);
for (int d = 0; d < (1<<20); d++){
  cout << C[d] << (d==(1<<20)?'\n':' ');
}
```

12.21 XOR Basis

```
/*
  XOR Basis (Linear Basis over GF(2))
  -----------------------------------
  Supports insertion of numbers into a basis and querying the maximum
  XOR of any subset with a given value.

  Bit Width: up to D bits (here D = 60 for 64-bit integers)
  Indexing: basis[0] is for bit 0 (least significant), basis[D-1] for
      bit D-1
  Bounds:
    - Values x inserted must satisfy 0 <= x < 2^D
  Time Complexity:
    - insert(x):   O(D)
    - getMax(x):   O(D)
  Space Complexity: O(D)

  Notes:
    - We maintain one basis vector per bit position.
    - On insert, we reduce x by existing basis vectors, then store it
      in the highest bit it still has.
    - To query max XOR, we greedily try to xor with basis vectors
      from highest bit down.
*/
```

```
const int D = 31; // Maximum number of bits in the numbers
int basis[D]; // basis[i] keeps the mask of the vector whose f value is
    i
int sz; // Current size of the basis

void insertVector(int mask) {
  //turn for around if you want max xor
  for (int i = 0; i < D; i++) {
    if ((mask & 1 << i) == 0) continue; // continue if i != f(mask)
    if (!basis[i]) { // If there is no basis vector with the ith bit set
        , then insert this vector into the basis
      basis[i] = mask;
      ++sz;
      return;
    }
    mask ^= basis[i]; // Otherwise subtract the basis vector from this
        vector
  }
}

// V2: If you dont need the basis sorted.
vector<ll> basis;
void add(ll x)
{
  for (int i = 0; i < basis.size(); i++)
  {
    x = min(x, x ^ basis[i]);
  }
  if (x != 0)
  {
    basis.pb(x);
  }
}
```

12.22 XOR Basis Online

```
int rebuild_and_delete (int id) {
  int pos = 0, mn = inf, p2 = 0;
  for (int i = 0; i < basis.size(); i++) {
      if (basis[i].id == id) {
          pos = i;
      }
  }
```

```
  int bits = 0;
  for (int i = 0; i < basis.size(); i++) {
      if (basis[i].mask & (1 << pos)) {
          if (mn > basis[i].high) {
              mn = basis[i].high;
              p2 = i;
          }
          bits ^= 1 << basis[i].high;
      }
  }

  if (p2 != pos) {
      swap(basis[p2].id, basis[pos].id);
      for (auto &i : basis) {
          swap_bits(i.mask, pos, p2); // just swaps pos-th and p2-th bit
              in i.mask
      }
      pos = p2;
  }
  for (int i = 0; i < basis.size(); i++) {
      if (i != pos) {
          if (basis[i].mask & (1 << pos)) {
              basis[i].val ^= basis[pos].val;
              basis[i].mask ^= basis[pos].mask;
          }
      }
  }
  int good = (1 << pos) - 1;
  int other = ((1 << len(basis)) - 1) ^ (good | (1 << pos));
  basis.erase(basis.begin() + pos);
  for (auto &i : basis) {
      i.mask = ((i.mask & other) >> 1) | (i.mask & good);
  }
  return bits;
}

C++
41 lines
1161 bytes


int get_the_same_high_bit (int bits, vector <int> &val) {
  for (auto &i : basis) {
      if (__builtin_popcount(val[i.id] & bits) & 1) {
          return i.id;
      }
  }
  return -1;
}

C++
8 lines
200 bytes

Thus, we have learned to delete in O(m2).

Here is full version of the code:
Code

//#pragma GCC optimize("Ofast", "unroll-loops")
//#pragma GCC target("sse", "sse2", "sse3", "ssse3", "sse4")

#include <bits/stdc++.h>

#define all(a) a.begin(),a.end()
#define len(a) (int)(a.size())
#define mp make_pair
#define pb push_back
#define fir first
#define sec second
#define fi first
#define se second

using namespace std;

typedef pair<int, int> pii;
typedef long long ll;
typedef long double ld;

template<typename T>
bool umin(T &a, T b) {
  if (b < a) {
      a = b;
      return true;
  }
  return false;
```

```
}
template<typename T>
bool umax(T &a, T b) {
  if (a < b) {
      a = b;
      return true;
  }
  return false;
}

#ifdef KIVI
#define DEBUG for (bool _FLAG = true; _FLAG; _FLAG = false)
#define LOG(...) print(#__VA_ARGS__" ::", __VA_ARGS__) << endl
template <class ...Ts> auto &print(Ts ...ts) { return ((cerr << ts << "␣
    "), ...); }
#else
#define DEBUG while (false)
#define LOG(...)
#endif

const int max_n = -1, inf = 1000111222;


const int C = 20;

struct node {
  int val, id, mask, high;
};


inline int get_high (int x) {
  if (x == 0) {
      return -1;
  }
  return 31 - __builtin_clz(x);
}

inline void swap_bits (int &x, int a, int b) {
  int x1 = bool(x & (1 << a));
  int x2 = bool(x & (1 << b));
  x ^= (x1 ^ x2) << a;
  x ^= (x1 ^ x2) << b;
}


struct xor_basis {
  vector <node> basis;

  inline bool add (int x, int id) {
      int mask = 0;
      for (auto &i : basis) {
          if (umin(x, x ^ i.val)) {
              mask ^= i.mask;
          }
      }
      if (x) {
          mask |= 1 << len(basis);
          for (auto &i : basis) {
              if (umin(i.val, i.val ^ x)) {
                  i.mask ^= mask;
              }
          }
          basis.pb(node{val: x, id: id, mask: mask, high: get_high(x)});
          return true;
      }
      return false;
  }



  inline int get_the_same_high_bit (int bits, const vector <int> &val) {
      for (auto &i : basis) {
          if (__builtin_popcount(val[i.id] & bits) & 1) {
              return i.id;
          }
      }
      return -1;
  }


  inline int rebuild_and_delete (int id) {
      int pos = 0, mn = inf, p2 = 0;
      for (int i = 0; i < len(basis); i++) {
          if (basis[i].id == id) {
              pos = i;
          }
      }
```

```
        int bits = 0;
        for (int i = 0; i < len(basis); i++) {
            if (basis[i].mask & (1 << pos)) {
                if (umin(mn, basis[i].high)) {
                    p2 = i;
                }
                bits ^= 1 << basis[i].high;
            }
        }

        if (p2 != pos) {
            swap(basis[p2].id, basis[pos].id);
            for (auto &i : basis) {
                swap_bits(i.mask, pos, p2);
            }
            pos = p2;
        }
        for (int i = 0; i < len(basis); i++) {
            if (i != pos) {
                if (basis[i].mask & (1 << pos)) {
                    basis[i].val ^= basis[pos].val;
                    basis[i].mask ^= basis[pos].mask;
                }
            }
        }
        int good = (1 << pos) - 1;
        int other = ((1 << len(basis)) - 1) ^ (good | (1 << pos));
        basis.erase(basis.begin() + pos);
        for (auto &i : basis) {
            i.mask = ((i.mask & other) >> 1) | (i.mask & good);
        }
        return bits;
    }

};

template<int max_bit> // not inclusive
struct xor_basis_online {

    vector <xor_basis> basises[max_bit + 1];

    int mx;

    vector <pii> where;
    vector <int> val;

    xor_basis_online() : mx(0), cur_id(0) {}

    int cur_id;

    inline int add (int x) {
        val.pb(x);
        where.pb(make_pair(-1, -1));
        int id = cur_id++;
        if (x == 0) {
            return id;
        }
        for (int i = mx; i >= 0; i--) {
            if (basises[i].empty()) {
                continue;
            }
            if (basises[i].back().add(x, id)) {
                basises[i + 1].pb(basises[i].back());
                basises[i].pop_back();
                umax(mx, i + 1);
                for (auto &j : basises[i + 1].back().basis) {
                    where[j.id] = make_pair(i + 1, len(basises[i + 1]) -
                        1);
                }
                return id;
            }
        }
        basises[1].pb(xor_basis());
        basises[1].back().add(x, id);
        where.back() = make_pair(1, len(basises[1]) - 1);
        umax(mx, 1);
        return id;
    }

    inline int move_back (int sz, int pos) {
        int to = len(basises[sz]) - 1;
        if (to == pos) {
            return pos;
        }
        for (auto &i : basises[sz][pos].basis) {
            where[i.id].second = to;
```

```
        }
        for (auto &i : basises[sz][to].basis) {
            where[i.id].second = pos;
        }
        swap(basises[sz][pos], basises[sz][to]);
        return to;
    }

    inline void del (int id) {
        if (val[id] == 0) {
            return;
        }
        int sz, pos;
        tie(sz, pos) = where[id];
        pos = move_back(sz, pos);
        while (true) {
            int bits = basises[sz].back().rebuild_and_delete(id);
            int i = sz - 1;
            while (i > 0 && basises[i].empty()) {
                --i;
            }
            int new_id = -1;
            if (i > 0) {
                new_id = basises[i].back().get_the_same_high_bit(bits, val
                    );
            }
            if (new_id == -1) {
                if (sz > 1) {
                    basises[sz - 1].pb(basises[sz].back());
                    for (auto &j : basises[sz - 1].back().basis) {
                        where[j.id] = make_pair(sz - 1, len(basises[sz -
                            1]) - 1);
                    }
                }
                basises[sz].pop_back();
                if (mx > 0 && basises[mx].empty()) {
                    --mx;
                }
                return;
            }
            int cur = val[new_id];
            assert(basises[sz].back().add(cur, new_id));
            int new_sz = i;
            int new_pos = len(basises[i]) - 1;
            where[new_id] = make_pair(sz, pos);
            id = new_id;
            sz = new_sz;
            pos = new_pos;
        }
    }

    inline int size () {
        return mx;
    }
};

int main() {
//    freopen("input.txt", "r", stdin);
//    freopen("output.txt", "w", stdout);

    ios_base::sync_with_stdio(0);
    cin.tie(0);


    // solution to https://basecamp.eolymp.com/uk/problems/11732
    int n, q, add;
    cin >> n >> add;
    vector <int> p(n);
    xor_basis_online<19> t;
    vector <int> now(n);
    for (int i = 0; i < n; i++) {
        cin >> p[i];
        now[i] = t.add(i ^ p[i]);
    }
    cin >> q;
    int ans = t.size();
    cout << ans << '\n';
    for (int i = 0, l, r; i < q; i++) {
        cin >> l >> r;
        l = (l + ans * add) % n;
        r = (r + ans * add) % n;
        if (l != r) {
            t.del(now[l]);
            t.del(now[r]);
            swap(p[l], p[r]);
            now[l] = t.add(l ^ p[l]);
```

```
            now[r] = t.add(r ^ p[r]);
        }
        ans = t.size();
        cout << ans << '\n';
    }
}
```

13 Polynomials

13.1 Berlekamp Massey

```
template<typename T>
vector<T> berlekampMassey(const vector<T> &s) {
    vector<T> c;    // the linear recurrence sequence we are building
    vector<T> oldC; // the best previous version of c to use (the one
        with the rightmost left endpoint)
    int f = -1;     // the index at which the best previous version of c
         failed on
    for (int i=0; i<(int)s.size(); i++) {
        // evaluate c(i)
        // delta = s_i - \sum_{j=1}^n c_j s_{i-j}
        // if delta == 0, c(i) is correct
        T delta = s[i];
        for (int j=1; j<=(int)c.size(); j++)
            delta -= c[j-1] * s[i-j];   // c_j is one-indexed, so we
                actually need index j - 1 in the code
        if (delta == 0)
            continue;   // c(i) is correct, keep going
        // now at this point, delta != 0, so we need to adjust it
        if (f == -1) {
            // this is the first time we're updating c
            // s_i was the first non-zero element we encountered
            // we make c of length i + 1 so that s_i is part of the base
                 case
            c.resize(i + 1);
            mt19937 rng(chrono::steady_clock::now().time_since_epoch().
                count());
            for (T &x : c)
                x = rng();  // just to prove that the initial values don
                    't matter in the first step, I will set to random
                    values
            f = i;
        } else {
            // we need to use a previous version of c to improve on this
                 one
            // apply the 5 steps to build d
            // 1. set d equal to our chosen sequence
            vector<T> d = oldC;
            // 2. multiply the sequence by -1
            for (T &x : d)
                x = -x;
            // 3. insert a 1 on the left
            d.insert(d.begin(), 1);
            // 4. multiply the sequence by delta / d(f + 1)
            T df1 = 0;  // d(f + 1)
            for (int j=1; j<=(int)d.size(); j++)
                df1 += d[j-1] * s[f+1-j];
            assert(df1 != 0);
            T coef = delta / df1;   // storing this in outer variable so
                 it's O(n^2) instead of O(n^2 log MOD)
            for (T &x : d)
                x *= coef;
            // 5. insert i - f - 1 zeros on the left
            vector<T> zeros(i - f - 1);
            zeros.insert(zeros.end(), d.begin(), d.end());
            d = zeros;
            // now we have our new recurrence: c + d
            vector<T> temp = c; // save the last version of c because it
                 might have a better left endpoint
            c.resize(max(c.size(), d.size()));
            for (int j=0; j<(int)d.size(); j++)
                c[j] += d[j];
            // finally, let's consider updating oldC
            if (i - (int) temp.size() > f - (int) oldC.size()) {
                // better left endpoint, let's update!
                oldC = temp;
                f = i;
            }
        }
    }
    return c;
}
```

13.2 FFT

```
using cd = complex<double>;
```

```cpp
const double PI = acos(-1);
//declare size of vectors used like this
const int MAXN=2<<19;

void fft(vector<cd> & 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);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
  cd u = a[i+j], v = a[i+j+len/2] * w;
  a[i+j] = u + v;
  a[i+j+len/2] = u - v;
  w *= wlen;
      }
    }
  }

  if (invert) {
    for (cd & x : a)
      x /= n;
  }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

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

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

//normalizing for when mult is between 2 big numbers and not polynomials
int carry = 0;
for (int i = 0; i < n; i++){
    result[i] += carry;
    carry = result[i] / 10;
    result[i] %= 10;
}
```

13.3 NTT

```cpp
// number theory transform

const int MOD = 998244353, ROOT = 3;
// const int MOD = 7340033, ROOT = 5;
// const int MOD = 167772161, ROOT = 3;
// const int MOD = 469762049, ROOT = 3;

int power(int base, int exp) {
    int res = 1;
    while (exp) {
        if (exp % 2) res = 1LL * res * base % MOD;
        base = 1LL * base * base % MOD;
        exp /= 2;
    }
    return res;
}

void ntt(vector<int>& a, bool invert) {
  int n = a.size();
```

```
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1) j ^= bit;
    j ^= bit;
    if (i < j) swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    int wlen = power(ROOT, (MOD - 1) / len);
    if (invert) wlen = power(wlen, MOD - 2);
    for (int i = 0; i < n; i += len) {
      int w = 1;
      for (int j = 0; j < len / 2; j++) {
        int u = a[i + j], v = 1LL * a[i + j + len / 2] * w % MOD;
        a[i + j] = u + v < MOD ? u + v : u + v - MOD;
        a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + MOD;
        w = 1LL * w * wlen % MOD;
      }
    }
  }
  if (invert) {
    int n_inv = power(n, MOD - 2);
    for (int& x : a) x = 1LL * x * n_inv % MOD;
  }
}

vector<int> multiply(vector<int>& a, vector<int>& b) {
  int n = 1;
  while (n < a.size() + b.size()) n <<= 1;
  a.resize(n), b.resize(n);
  ntt(a, false), ntt(b, false);
  for (int i = 0; i < n; i++) a[i] = 1LL * a[i] * b[i] % MOD;
  ntt(a, true);
  return a;
}
// usage
// vector<int> a = {1, 2, 3}, b = {4, 5, 6};
// vector<int> c = multiply(a, b);
// for (int x : c) cout << x << " ";
```

13.4 Roots NTT

```
1*2^0 + 1 = 2, 1, 1
1*2^1 + 1 = 3, 2, 2
1*2^2 + 1 = 5, 2, 3
2*2^3 + 1 = 17, 2, 9
1*2^4 + 1 = 17, 3, 6
3*2^5 + 1 = 97, 19, 46
3*2^6 + 1 = 193, 11, 158
2*2^7 + 1 = 257, 9, 200
1*2^8 + 1 = 257, 3, 86
15*2^9 + 1 = 7681, 62, 1115
12*2^10 + 1 = 15361, 49, 1254
6*2^11 + 1 = 12289, 7, 8778
3*2^12 + 1 = 12289, 41, 4496
5*2^13 + 1 = 40961, 12, 23894
4*2^14 + 1 = 65537, 15, 30584
2*2^15 + 1 = 65537, 9, 7282
1*2^16 + 1 = 65537, 3, 21846
6*2^17 + 1 = 786433, 8, 688129
3*2^18 + 1 = 786433, 5, 471860
11*2^19 + 1 = 5767169, 12, 3364182
7*2^20 + 1 = 7340033, 5, 4404020
11*2^21 + 1 = 23068673, 38, 21247462
25*2^22 + 1 = 104857601, 21, 49932191
20*2^23 + 1 = 167772161, 4, 125829121
10*2^24 + 1 = 167772161, 2, 83886081
5*2^25 + 1 = 167772161, 17, 29606852
7*2^26 + 1 = 469762049, 30, 15658735
15*2^27 + 1 = 2013265921, 137, 749463956
12*2^28 + 1 = 3221225473, 8, 2818572289
6*2^29 + 1 = 3221225473, 14, 1150437669
3*2^30 + 1 = 3221225473, 13, 1734506024
35*2^31 + 1 = 75161927681, 93, 44450602392
18*2^32 + 1 = 77309411329, 106, 5105338484
```

14 Strings

14.1 Hashed String

```

/*
                                        Hashed string
  ------------------------------------------------------------------
  Class for hashing string. Allows retrieval of hashes of any substring
      in the string.

```

```
  Double hash or use big mod values to avoid problems with collisions

  Time Complexity(Construction): O(n)
  Space Complexity: O(n)
*/

const ll MOD = 212345678987654321LL;
const ll base = 33;

class HashedString {
  private:
  // change M and B if you want
  static const long long M = 1e9 + 9;
  static const long long B = 9973;

  // pow[i] contains B^i % M
  static vector<long long> pow;

  // p_hash[i] is the hash of the first i characters of the given string
  vector<long long> p_hash;

  public:
  HashedString(const string &s) : p_hash(s.size() + 1) {
    while (pow.size() < s.size()) { pow.push_back((pow.back() * B) % M);
        }

    p_hash[0] = 0;
    for (int i = 0; i < s.size(); i++) {
      p_hash[i + 1] = ((p_hash[i] * B) % M + s[i]) % M;
    }
  }

  // Returns hash of substring [start, end]
  long long get_hash(int start, int end) {
    long long raw_val =
        (p_hash[end + 1] - (p_hash[start] * pow[end - start + 1]));
    return (raw_val % M + M) % M;
  }
};
// you cant skip this
vector<long long> HashedString::pow = {1};
```

14.2 KMP

```
/*
                                KMP
  -----------------------------------------------------------
  Computes the prefix function for a string.

  Maximum length of substring that ends at position i and is proper
      prefix (not equal to string itself) of string
  pf[i]  is the length of the longest proper prefix of the substrings
      [0.....i]$  which is also a suffix of this substring.

  For matching, one can append the string with a delimiter like $
      between them

  Time Complexity: O(n)
  Space Complexity: O(n)
*/

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

// Counts how many times each prefix occurs
// Same thing can be done for two strings but only considering indices
    of second string
vector<int> count_occurrences_of_prefixes(vector<int> pf){
  int n=(int)pf.size();
  vector<int> ans(n + 1);
  for (int i = 0; i < n; i++)
    ans[pi[i]]++;
  for (int i = n-1; i > 0; i--)
    ans[pi[i-1]] += ans[i];
  for (int i = 0; i <= n; i++)
```

```
    ans[i]++;
}

// Computes automaton for string
// useful for not having to recalculate KMP of string s
// can be utilized when the second string (the one in which we are
    trying to count occurences)
// is very large
void compute_automaton(string s, vector<vector<int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = KMP(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
      for (int c = 0; c < 26; c++) {
          if (i > 0 && 'a' + c != s[i])
              aut[i][c] = aut[pi[i-1]][c];
          else
              aut[i][c] = i + ('a' + c == s[i]);
      }
  }
}
```

14.3 Least Rotation String

```
/*
                          Min cyclic shift
  ---------------------------------------------------------------
  Finds the lexicographically minimum cyclic shift of a string

  Time Complexity: O(n)
  Space Complexity: O(n)
*/

string least_rotation(string s)
{
  s += s;
  vector<int> f(s.size(), -1);
  int k = 0;
  for(int j = 1; j < s.size(); j++)
  {
    char sj = s[j];
    int i = f[j - k - 1];
    while(i != -1 && sj != s[k + i + 1])
    {
      if(sj < s[k + i + 1]){
        k = j - i - 1;
      }
      i = f[i];
    }
    if(sj != s[k + i + 1])
    {
      if(sj < s[k]){
        k = j;
      }
      f[j - k] = -1;
    }
    else
      f[j - k] = i + 1;
  }
  return s.substr(k, s.size() / 2);
}
```

14.4 Manacher

```
/*
                              Manacher
  ---------------------------------------------------------------
  Computes the length of the longest palindrome centered at position i.

  p[i] is length of biggest palindrome centered in this position.
  Be careful with characters that are inserted to account for odd and
      even palindromes

  Time Complexity: O(n)
  Space Complexity: O(n)

*/

// Number of palindromes centered at each position

vector<int> manacher_odd(string s)
{
  int n = s.size();
  s = "$" + s + "^";
  vector<int> p(n + 2);
```

```
  int l = 1, r = 1;
  for (int i = 1; i <= n; i++)
  {
    p[i] = max(0, min(r - i, p[l + (r - i)]));
    while (s[i - p[i]] == s[i + p[i]])
    {
      p[i]++;
    }
    if (i + p[i] > r)
    {
      l = i - p[i], r = i + p[i];
    }
  }
  return vector<int>(begin(p) + 1, end(p) - 1);
}
vector<int> manacher(string s)
{
  string t;
  for (auto c : s)
  {
    t += string("#") + c;
  }
  auto res = manacher_odd(t + "#");
  return vector<int>(begin(res) + 1, end(res) - 1);
}

// usage
// vector<int> p = manacher("abacaba");
// this will return {2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2}
// vector<int> p = manacher("abaaba");
// this will return {2, 1, 4, 1, 2, 7, 2, 1, 4, 1, 2}
```

14.5 Suffix Array

```
/*
                        Suffix Array
  ------------------------------------------------------------
  Computes the suffix array of a string in O(n log n).
  Sorted array of all cyclic shifts of a string.

  If you want sorted suffixes append $ to the end of the string.
  lc is longest common prefix. Lcp of two substrings j > i is min(lc[i],
      ....., lc[j - 1]).

  To compute Largest common substring of multiple strings
  Join all strings separating them with special character like $ (it has
      to be different for each string)
  Sliding window on lcp array (all string have to appear on  the sliding
      window and
  the lcp of the interval will give the length of the substring that
     appears on all strings)

  Time Complexity: O(n log n)
  Space Complexity: O(n)

*/

struct SuffixArray
{
  int n;
  string t;
  vector<int> sa, rk, lc;
  SuffixArray(const std::string &s)
  {
    n = s.length();
    t = s;
    sa.resize(n);
    lc.resize(n - 1);
    rk.resize(n);
    std::iota(sa.begin(), sa.end(), 0);
    std::sort(sa.begin(), sa.end(), [&](int a, int b)
              { return s[a] < s[b]; });
    rk[sa[0]] = 0;
    for (int i = 1; i < n; ++i)
      rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
    int k = 1;
    std::vector<int> tmp, cnt(n);
    tmp.reserve(n);
    while (rk[sa[n - 1]] < n - 1)
    {
      tmp.clear();
      for (int i = 0; i < k; ++i)
        tmp.push_back(n - k + i);
      for (auto i : sa)
        if (i >= k)
          tmp.push_back(i - k);
```

```
    std::fill(cnt.begin(), cnt.end(), 0);
    for (int i = 0; i < n; ++i)
      ++cnt[rk[i]];
    for (int i = 1; i < n; ++i)
      cnt[i] += cnt[i - 1];
    for (int i = n - 1; i >= 0; --i)
      sa[--cnt[rk[tmp[i]]]] = tmp[i];
    std::swap(rk, tmp);
    rk[sa[0]] = 0;
    for (int i = 1; i < n; ++i)
      rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i
          - 1] + k == n || tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
    k *= 2;
  }
  for (int i = 0, j = 0; i < n; ++i)
  {
    if (rk[i] == 0)
    {
      j = 0;
    }
    else
    {
      for (j -= j > 0; i + j < n && sa[rk[i] - 1] + j < n && s[i + j]
          == s[sa[rk[i] - 1] + j];)
        ++j;
      lc[rk[i] - 1] = j;
    }
  }
}

// Finds if string p appears as substring in the string
// might now work perfectly
int search(string &p){
    int tam = p.size();
    int l = 0, r = n;

    string tmp = "";
    while(r > l) {
        int m = l + (r-l)/2;
        tmp = t.substr(sa[m], min(n-sa[m], tam));
        if(tmp >= p){
            r = m;
        } else {
            l = m + 1;
        }
    }
    if(l < n) {
        tmp = t.substr(sa[l], min(n-sa[l], tam));
    } else{
        return -1;
    }
    if(tmp == p){
        return l;
    } else {
        return -1;
    }
}


// Counts number of times a string p appears as substring in string
int count(string &p) {
    int x = search(p);
    if(x == -1) return 0;
    int cnt = 0;
    int tam = p.size();
    int maxx = 0;
    while((1 << maxx) + x < n) maxx++;
    int y = x;
    for(int i = maxx-1; i >= 0; i--) {
        if(x + (1 << i) >= n) continue;
        string tmp = t.substr(sa[x + (1 << i)], min(n-sa[x + (1 << i)
            ], tam));
        if(tmp == p) x += (1 << i);
    }
    return x-y+1;
}
};
int main() {
    cin.tie(0)->sync_with_stdio(0);
    string s; cin >> s;
    SuffixArray SA(s);

    int q; cin >> q;
    for(int t = 0; t < q; t++) {
        string tmp; cin >> tmp;
        cout << SA.count(tmp) << endl;
```

```
    }

    return 0;
}
```

14.6 Suffix Automaton

```
/*
                            Suffix Automaton
  ----------------------------------------------------------------
  Constructs suffix automaton for a given string.
  Be careful with overlapping substrings.

  Firstposition if first position string ends in.
  If you want starting index you need to substract length of the string
      being searched.

  len is length of longest string of state

  Time Complexity(Construction): O(n)
  Space Complexity: O(n)
*/


struct state {
  int len, link, firstposition;
  vector<int> inv_link; // can skip for almost everything
  map<char, int> next;
};

const int MAXN = 100000;
state st[MAXN * 2];
ll cnt[MAXN*2], cntPaths[MAXN*2], cntSum[MAXN*2], cnt1[2 * MAXN];
int sz, last;

// call this first
void initSuffixAutomaton() {
  st[0].len = 0;
  st[0].link = -1;
  sz++;
  last = 0;
}

// construction is O(n)
void insertChar(char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  st[cur].firstposition=st[last].len;
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
      st[p].next[c] = cur;
      p = st[p].link;
  }
  if (p == -1) {
      st[cur].link = 0;
  } else {
      int q = st[p].next[c];
      if (st[p].len + 1 == st[q].len) {
          st[cur].link = q;
      } else {
          int clone = sz++;
          st[clone].len = st[p].len + 1;
          st[clone].next = st[q].next;
          st[clone].link = st[q].link;
          st[clone].firstposition=st[q].firstposition;
          while (p != -1 && st[p].next[c] == q) {
              st[p].next[c] = clone;
              p = st[p].link;
          }
          st[q].link = st[cur].link = clone;
      }
  }
  last = cur;
  cnt[last]=1;
}

// searches for the starting position in O(len(s)). Returns starting
    index of first ocurrence or -1 if it does not appear.
int search(string s){
  int cur=0, i=0, n=(int)s.length();
  while(i<n){
    if(!st[cur].next.count(s[i])) return -1;
    cur=st[cur].next[s[i]];
    i++;
  }
  //sumar 2 si se quiere 1 indexado
```

```
  return st[cur].firstposition-n+1;
}

void dfs(int cur){
  cntPaths[cur]=1;
  for(auto [x, y]:st[cur].next){
    if(cntPaths[y]==0) dfs(y);
    cntPaths[cur]+=cntPaths[y];
  }
}

// Counts how many paths exist from state. How many substrings exist
    from a specific state.
// Stored in cntPaths
void countPaths(){
  dfs(0);
}

// Computes the number of times each state appears
void countOcurrences(){
  vector<pair<int, int>> a;
  for(int i=sz-1;i>0;i--){
    a.push_back({st[i].len, i});
  }
  sort(a.begin(), a.end());
  for(int i=sz-2;i>=0;i--){
    cnt[st[a[i].second].link]+=cnt[a[i].second];
  }
}

void dfs1(int cur){
  for(auto [x, y]:st[cur].next){
    if(cntSum[y]==cnt[y]) dfs1(y);
    cntSum[cur]+=cntSum[y];
  }
}

// Computes the number of times each state or any of its children appear
     in the string.
void countSumOcurrences(){
  for(int i=0;i<sz;i++){
    cntSum[i]=cnt[i];
  }
  dfs1(0);
}


// Counts number of paths that can reach specific state.
void countPathsReverse(){
  cnt1[0]=1;
  queue<int> q;
  q.push(0);
  vector<int> in(2*MAXN, 0);
  for(int i=0;i<sz;i++){
    for(auto [x, y]:st[i].next){
      in[y]++;
    }
  }
  while((int)q.size()){
    int cur=q.front();
    q.pop();
    for(auto [x, y]:st[cur].next){
      cnt1[y]+=cnt1[cur];
      in[y]--;
      if(in[y]==0){
        q.push(y);
      }
    }
  }
}

// Computes the kth smallest string that appears on the string (counting
     repetitions)
string kthSmallest(ll k){
  string s="";
  int cur=0;
  while(k>0){
    for(auto [c, y]:st[cur].next){
      if(k>cntSum[y]) k-=cntSum[y];
      else{
        k-=cnt[y];
        s+=c;
        cur=y;
        break;
      }
    }
```

```
  }
  return s;
}


// Computes the kth smallest string that appears on the string (without counting repetitions)
string kthSmallestDistinct(ll k){
  string s="";
  int cur=0;
  while(k>0){
    for(auto [c, y]:st[cur].next){
      if(k>cntPaths[y]) k-=cntPaths[y];
      else{
        k--;
        s+=c;
        cur=y;
        break;
      }
    }
  }
  return s;
}


// Precomputation to find all occurrences of a substring
void precoumpte_for_all_ocurrences(){
  for (int v = 1; v < sz; v++) {
    st[st[v].link].inv_link.push_back(v);
  }
}

// Finding all occurences of substring in string
// P_length is length of substring
// v is state where first ocurrence happens
// be careful as indices can appear multiple times due to clone states
// if you want to avoid duplicate positions utilize set or have a flag for each state to know if it is clone or not
void output_all_occurrences(int v, int P_length) {
  cout << st[v].firstposition - P_length + 1 << endl;
  for (int u : st[v].inv_link)
    output_all_occurrences(u, P_length);
}


//longest common substring
//build automaton for s first
string lcs (string S, string T) {
    int v = 0, l = 0, best = 0, bestpos = 0;
    for (int i = 0; i < T.size(); i++) {
        while (v && !st[v].next.count(T[i])) {
            v = st[v].link ;
            l = st[v].len;
        }
        if (st[v].next.count(T[i])) {
            v = st [v].next[T[i]];
            l++;
        }
        if (l > best) {
            best = l;
            bestpos = i;
        }
    }
    return T.substr(bestpos - best + 1, best);
}


int main(){
  ios_base::sync_with_stdio(false); cin.tie(NULL);
  string s; cin >> s;
  initSuffixAutomaton();
  for(char c:s){
    insertChar(c);
  }
}
```

14.7 Trie Ahocorasick

```
/*
                        Trie - AhoCorasick
  ---------------------------------------------------------------
  Builds a trie for subset of strings and computes suffix links.
  KATCL implementation is cleaner.

  Time Complexity(Construction): O(m) where m is sum
   of lengths of strings
```

```
  Space Complexity: O(m)
*/



const int K = 26;

struct Vertex {
    int next[K];
    bool output = false;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].output = true;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

14.8 Z Function

```
/*
                              Z_function
  -----------------------------------------------------------
  Computes the z_function for any string.
  ith element is equal to the greatest number of characters starting
      from the position i that coincide with the first characters of s

  z[i] length of the longest string that is, at the same time, a prefix
      of s and a prefix of the suffix of $s$ starting at  i.

  to compress string, one can run z_function and then find the smallest
      i that divides n such that i + z[i] = n

  Time Complexity: O(n)
  Space Complexity: O(n)
*/


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

```
    }
    while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }
    if(i + z[i] > r) {
      l = i;
      r = i + z[i];
    }
  }
  return z;
}

// usage
// vector<int> z = z_function("abacaba");
// this will return {0, 0, 1, 0, 3, 0, 1}
// vector<int> z = z_function("aaaaa");
// this will return {0, 4, 3, 2, 1}
// vector<int> z = z_function("aaabaab");
// this will return {0, 2, 1, 0, 2, 1, 0}
```

15 Trees

15.1 Centroid Decomposition

```
/*
                          Centroid Decomposition
  ------------------------------------------------------------------
  Finds the centroid decomposition of a given tree.
  Any vertex can have at most log n centroid ancestors

  The code below is the solution to Xenia and tree.
  Given tree, queries of two types:
  1) u - color vertex u
  2) v - print minimum distance of vertex v to any colored vertex before


  Time Complexity: O(n log n)
  Space Complexity: O(n log n)
*/
const int MAXN=200005;

vector<int> adj[MAXN];
vector<bool> is_removed(MAXN, false);
vector<int> subtree_size(MAXN, 0);
vector<int> dis(MAXN, 1e9);
vector<vector<pair<int, int>>> ancestor(MAXN);

int get_subtree_size(int node, int parent = -1) {
  subtree_size[node] = 1;
  for (int child : adj[node]) {
    if (child == parent || is_removed[child]) { continue; }
    subtree_size[node] += get_subtree_size(child, node);
  }
  return subtree_size[node];
}

int get_centroid(int node, int tree_size, int parent = -1) {
  for (int child : adj[node]) {
    if (child == parent || is_removed[child]) { continue; }
    if (subtree_size[child] * 2 > tree_size) {
      return get_centroid(child, tree_size, node);
    }
  }
  return node;
}

void getDist(int cur, int centroid, int p=-1, int dist=1){
    for (int child:adj[cur]){
        if(child==p ||is_removed[child])
            continue;
        dist++;
        getDist(child, centroid, cur, dist);
        dist--;
    }
    ancestor[cur].push_back(make_pair(centroid, dist));
}

void update(int cur){
    for (int i=0;i<ancestor[cur].size();i++){
        dis[ancestor[cur][i].first]=min(dis[ancestor[cur][i].first],
            ancestor[cur][i].second);
    }
    dis[cur]=0;
}

int query(int cur){
```

```
    int mini=dis[cur];
    for (int i=0;i<ancestor[cur].size();i++){
       mini=min(mini, ancestor[cur][i].second+dis[ancestor[cur][i].first
           ]);
    }
    return mini;
}

void build_centroid_decomp(int node = 1) {
  int centroid = get_centroid(node, get_subtree_size(node));

  for (int child : adj[centroid]) {
    if (is_removed[child]) { continue; }
    getDist(child, centroid, centroid);
  }

  is_removed[centroid] = true;

  for (int child : adj[centroid]) {
    if (is_removed[child]) { continue; }
    build_centroid_decomp(child);
  }
}
```

15.2 Heavy Light Decomposition

```
/*
                    Heavy Light Decomposition(HLD)
  ---------------------------------------------------------------------
  Constructs the heavy light decomposition of a tree

  Splits the tree into several paths so that we can reach the root
      vertex from any v  by traversing at most log n paths.
  In addition, none of these paths intersect with another.

  Time Complexity(Creation): O(n log n)
  Time Complexity(Query): O((log n) ^ 2) usually, depending on the query
       itself
  Space Complexity: O(n)
*/

//call dfs1 first
struct SegmentTree {
  vector<ll> a;
  int n;

  SegmentTree(int _n) : a(2 * _n, 0), n(_n) {}

  void update(int pos, ll val) {
    for (a[pos += n] = val; pos > 1; pos >>= 1) {
      a[pos / 2] = (a[pos]^a[pos ^ 1]);
    }
  }

  ll get(int l, int r) {
    ll res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        res^=a[l++];
      }
      if (r & 1) {
        res^=a[--r];
      }
    }
    return res;
  }
};


const int MAXN=500005;
vector<int> adj[MAXN];
SegmentTree st(MAXN);
int a[MAXN], sz[MAXN], to[MAXN], dpth[MAXN], s[MAXN], par[MAXN];
int cnt=0;

void dfs1(int cur, int p){
    sz[cur]=1;
    for(int x:adj[cur]){
        if(x==p) continue;
        dpth[x]=dpth[cur]+1;
        par[x]=cur;
        dfs1(x, cur);
        sz[cur]+=sz[x];
    }
}

```

```
void dfs(int cur, int p, int l){
    st.update(cnt, a[cur]);
    s[cur]=cnt++;
    to[cur]=l;
    int g=-1;
    for(int x:adj[cur]){
        if(x==p) continue;
        if(g==-1 || sz[g]<sz[x]){
            g=x;
        }
    }
    if(g==-1) return;
    dfs(g, cur, l);
    for(int x:adj[cur]){
        if(x==p || x==g) continue;
        dfs(x, cur, x);
    }
}

int query(int u, int v){
    int res=0;
    while(to[u]!=to[v]){
        if(dpth[to[u]]<dpth[to[v]]) swap(u, v);
        res^=st.get(s[to[u]], s[u]+1);
        u=par[to[u]];
    }
    if(dpth[u]>dpth[v]) swap(u, v);
    res^=st.get(s[u], s[v]+1);
    return res;
}



//alternate implementation
vector<int> parent, depth, heavy, head, pos;
int cur_pos;

int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}

void decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c, adj);
    }
}

void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;

    dfs(0, adj);
    decompose(0, 0, adj);
}

int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b
            ]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])
```

```
        swap(a, b);
    int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res;
}
```

15.3 Lowest Common Ancestor (LCA)

```
/*
                        LCA(Lowest Common Ancestor)
  ------------------------------------------------------------------------
  Computes the lowest common ancestor of two vertices in a tree.

  Be careful as implementation is indexed starting with 1

  Time Complexity(Creation): O(n log n)
  Time Complexity(Query): O(log n)
  Space Complexity: O(n log n)
*/

const int N=200005;
vector<int> adj[N];
vector<int> start(N), end1(N), depth(N);
vector<vector<int>> t(N, vi(32));
int timer=0;
int n, l;
// l=(int)ceil(log2(n))
// call dfs(1, 1, 0)
// 1 indexed, dont use 0 indexing


void dfs(int cur, int p, int cnt){
    depth[cur]=cnt;
    t[cur][0]=p;
    start[cur]=timer++;
    for(int i=1;i<=l;i++){
        t[cur][i]=t[t[cur][i-1]][i-1];
    }
    for(int x:adj[cur]){
        if(x==p) continue;
        dfs(x, cur, cnt+1);
    }
    end1[cur]=++timer;
}

bool ancestor(int u, int v){
    return start[u]<=start[v] && end1[u]>=end1[v];
}

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

15.4 Tree Diameter

```
/*
                            Tree Diameter
  ------------------------------------------------------------------------
  Finds the vertex most distant to vertex on which function is called.

  The first value is the vertex itself and the second value is the
      distance.

  To find diameter run algorithm twice, first on random vertex and then
      on the vertex that is farthest away.

  The vertex that is the farthest away from any vertex in tree must be
      an endpoint of the diameter.

  Time Complexity: O(n)
  Space Complexity: O(n)
*/

pair<int, int> dfs(const vector<vector<int>> &tree, int node = 1,
  int previous = 0, int length = 0) {
  pair<int, int> max_path = {node, length};
```

```
  for (const int &i : tree[node]) {
    if (i == previous) { continue; }
    pair<int, int> other = dfs(tree, i, node, length + 1);
    if (other.second > max_path.second) { max_path = other; }
  }
  return max_path;
}
```

16 Scripts

16.1 build.sh

This file should be called before stress.sh or validate.sh. build.sh name.cpp

```
g++ -static -DLOCAL -lm -s -x c++ -Wall -Wextra -O2 -std=c++17 -o $1 $1.
    cpp
```

16.2 stress.sh

Format is stress.sh Awrong Aslow Agen Numtests

```
#!/usr/bin/env bash

for ((testNum=0;testNum<$4;testNum++))
do
    ./$3 > input
    ./$2 < input > outSlow
    ./$1 < input > outWrong
    H1=‘md5sum outWrong‘
    H2=‘md5sum outSlow‘
    if !(cmp -s "outWrong" "outSlow")
    then
        echo "Error␣found!"
        echo "Input:"
        cat input
        echo "Wrong␣Output:"
        cat outWrong
        echo "Slow␣Output:"
        cat outSlow
        exit
    fi
done
echo Passed $4 tests
```

16.3 validate.sh

Format is validate.sh Awrong Avalidator Agen NumTests

```
#!/usr/bin/env bash

for ((testNum=0;testNum<$4;testNum++))
do
    ./$3 > input
    ./$1 < input > out
    cat input out > data
    ./$2 < data > res
    result=$(cat res)
    if [ "${result:0:2}" != "OK" ];
    then
        echo "Error␣found!"
        echo "Input:"
        cat input
        echo "Output:"
        cat out
        echo "Validator␣Result:"
        cat res
        exit
    fi
done
echo Passed $4 tests
```