

Contents

1	Funciones C++	3	4.2	Digit DP	29
2	Compile	3	4.3	Divide and Conquer DP	30
2.1	Compile	3	4.4	Edit Distance	30
2.2	Template	3	4.5	LCS	31
3	Data Structures	4	4.6	Line Container	31
3.1	BIT	4	4.7	Longest Increasing Subsequence	31
3.2	Bitset	4	5	Flow	32
3.3	Bit Trie	4	5.1	Dinic	32
3.4	Disjoint Set Union Bipartite	5	5.2	Hopcroft-Karp	33
3.5	Disjoint Set Union	6	5.3	Hungarian	34
3.6	Dynamic Conectivity	6	5.4	Max Flow Min Cost	35
3.7	Fenwick Tree	9	5.5	Max Flow	36
3.8	Fenwick Tree 2D	9	5.6	Min Cost Max Flow	37
3.9	Merge Sort Tree	10	5.7	Push Relabel	38
3.10	Minimum Cartesian Tree	11	6	Geometry	39
3.11	Multi Ordered Set	11	6.1	Point Struct	39
3.12	Ordered Set	12	6.2	Sort Points	39
3.13	Palindromic Tree	12	7	Graphs	39
3.14	Persistent Array	13	7.1	2Sat	39
3.15	Persistent Segment Tree	14	7.2	Articulation Points	41
3.16	Segment Tree	16	7.3	Bellman-Ford	41
3.17	Segment Tree 2D	16	7.4	Bipartite Checker	42
3.18	Segment Tree Dynamic	17	7.5	Bipartite Maximum Matching	43
3.19	Segment Tree Lazy Types	18	7.6	Block Cut Tree	43
3.20	Segment Tree Lazy	18	7.7	Blossom	45
3.21	Segment Tree Lazy Range Set	20	7.8	Bridges	47
3.22	Segment Tree Max Subarray Sum	21	7.9	Bridges Online	48
3.23	Segment Tree Range Update	22	7.10	Dijkstra	49
3.24	Segment Tree Struct Types	22	7.11	Eulerian Path	50
3.25	Segment Tree Struct	23	7.12	Floyd-Warshall	50
3.26	Segment Tree Walk	23	7.13	Kruskal	51
3.27	Sparse Table	24	7.14	Marriage	51
3.28	Square Root Decomposition	25	7.15	SCC	52
3.29	Treap	26	8	Linear Algebra	53
3.30	Treap 2	27	8.1	Simplex	53
3.31	Treap With Inversion	28	9	Math	54
4	Dynamic Programming	29	9.1	BinPow	54
4.1	CHT Deque	29	9.2	Diophantine	54

9.3 Discrete Logarithm	55	11.18XOR Convolution	67
9.4 Divisors	56	11.19XOR Basis	67
9.5 Euler Totient (Phi)	57	12 Polynomials	68
9.6 Fibonacci	57	12.1 Berlekamp Massey	68
9.7 Matrix Exponentiation	57	12.2 FFT	68
9.8 Miller Rabin Deterministic	58	12.3 NTT	69
9.9 Mobius	59	12.4 Roots NTT	70
9.10 Prefix Sum Phi	59	13 Scripts	70
9.11 Sieve	59	13.1 build.sh	70
9.12 Identities	60	13.2 stress.sh	70
9.13 Burnside's Lemma	60	13.3 validate.sh	71
9.14 Recursion	60	14 Strings	71
9.15 Theorems	60	14.1 Hashed String	71
9.16 Sums	61	14.2 KMP	72
9.17 Catalan numbers	61	14.3 Least Rotation String	72
9.18 Cayley's formula	61	14.4 Manacher	73
9.19 Geometric series	61	14.5 Suffix Array	74
9.20 Estimates For Divisors	61	14.6 Suffix Automaton	75
9.21 Sum of divisors	61	14.7 Trie Ahocorasick	78
9.22 Pythagorean Triplets	61	14.8 Z Function	79
9.23 Derangements	61	15 Trees	79
10 Game Theory	61	15.1 Centroid Decomposition	79
10.1 Sprague-Grundy theorem	61	15.2 Heavy Light Decomposition	81
11 More Topics	62	15.3 Lowest Common Ancestor (LCA)	82
11.1 2D Prefix Sum	62	15.4 Tree Diameter	83
11.2 Custom Comparators	62		
11.3 Day of the Week	62		
11.4 GCD Convolution	62		
11.5 int128	62		
11.6 Iterating Over All Subsets	63		
11.7 LCM Convolution	63		
11.8 Manhattan MST	64		
11.9 Mo	64		
11.10MOD INT	65		
11.11Next Permutation	65		
11.12Next and Previous Smaller/Greater Element	65		
11.13Parallel Binary Search	66		
11.14Random Number Generators	66		
11.15setprecision	66		
11.16Ternary Search	66		
11.17Ternary Search Int	67		

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]	$T = \sum / \text{oper de } [f, l)$
inner_product	f1, l1, f2, i	$T = i + [f1, l1) \cdot [f2, \dots)$
partial_sum	f, l, r, [op]	$r+i = \sum / \text{oper de } [f, f+i) \forall i \in [f, l)$
__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

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

2.2 Template

```
1 #include <bits/stdc++.h>
2 #pragma GCC optimize("O3,unroll-loops")
3 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
4 using namespace std;
5 #define pb push_back
6 #define ll long long
7 #define s second
8 #define f first
9 #define MOD 1000000007
10 #define INF 1000000000000000
11
12 void solve(){
13
```

```

14 }
15
16 int main() {
17     ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);
18     int t;cin>>t;for(int T=0;T<t;T++)
19         solve();
20 }

```

3 Data Structures

3.1 BIT

```

1  /*
2   Binary Indexed Tree (Fenwick Tree) Fast Implementation
3   -----
4   Indexing: 1-based
5   Bounds: [1, MAXN)
6   Time Complexity:
7       - update(x, val): O(log n) -> adds val to index x
8       - get(x): O(log n) -> prefix sum from 1 to x
9   Space Complexity: O(n)
10  */
11
12  const int MAXN = 10000;
13  int bit[MAXN];
14
15  // Add 'val' to index 'x'
16  void update(int x, int val) {
17      for (; x < MAXN; x += x & -x) {
18          bit[x] += val;
19      }
20  }
21
22  // Get prefix sum from 1 to 'x'
23  int get(int x) {
24      int ans = 0;
25      for (; x > 0; x -= x & -x) {
26          ans += bit[x];
27      }
28      return ans;
29  }

```

3.2 Bitset

```

1  bitset<3001> b[3001];
2
3  //set() Set the bit value at the given index to 1.
4  //reset() Set the bit value at the given index to 0.
5  //flip() Toggle the bit value at the given index.
6  //test() Check if the bit value at the given index is 1 or 0.
7  //count() Count the number of set bits.
8  //any() Checks if any bit is set
9  //all() Check if all bit is set.
10 //none() Check if no bit is set.
11 //to_string() Convert the bitset to a string representation.
12
13 #pragma GCC target("popcnt")
14 (int) __builtin_popcount(x);
15 (int) __builtin_popcountll(x);
16 __builtin_clz(x); // count leading zeros
17
18 // declare bitset
19 bitset<64> b;

```

3.3 Bit Trie

```

1  /*
2   Bit Trie (Binary Trie for Integers)
3   -----
4   Indexing: 0-based
5   Bit Width: [0, MAX_BIT] inclusive (e.g., 31 for 32-bit integers)
6   Time Complexity:
7       - Insert: O(MAX_BIT)
8       - Query: O(MAX_BIT)
9   Space Complexity: O(N * MAX_BIT) nodes (in worst case, 1 per bit per
10                      number)
11  */
12
13  const int K = 2; // Each node has 2 branches (bit 0 or 1)
14  const int MAX_BIT = 30; // Max bit position (for 32-bit integers)
15
16  struct Vertex {
17      int next[K]; // next[0] = child for bit 0, next[1] = child for bit 1
18
19      Vertex() {
20          fill(begin(next), end(next), -1); // -1 means no child
21      }
22  }

```

```

21 };
22
23 vector<Vertex> trie; // Trie nodes
24
25 // Inserts a number into the binary trie
26 void insert(int num) {
27     int v = 0; // Start from root
28     for (int j = MAX_BIT; j >= 0; --j) {
29         int c = (num >> j) & 1; // Extract j-th bit (0 or 1)
30         if (trie[v].next[c] == -1) {
31             trie[v].next[c] = trie.size();
32             trie.emplace_back(); // Add new node
33         }
34         v = trie[v].next[c]; // Move to next node
35     }
36 }

```

3.4 Disjoint Set Union Bipartite

```

1  /*
2  DSU with Parity - Bipartite Checker
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, n-1] inclusive
6
7  Features:
8  - Tracks parity (even/odd length) of paths in each component
9  - Can be used to detect odd-length cycles (non-bipartite components)
10 - Supports dynamic edge additions
11
12 Functions:
13 - make_set(v): initializes singleton component
14 - find_set(v): returns root of v and its parity relative to root
15 - add_edge(a, b): merges two components and checks for bipartite
    violation
16 - is_bipartite(v): returns whether component containing v is
    bipartite
17 */
18
19 const int MAXN = 100005; // Set according to constraints
20
21 pair<int, int> parent[MAXN]; // parent[v] = {root, parity_from_root_to_v
    }

```

```

22 int rank[MAXN]; // Union by rank
23 bool bipartite[MAXN]; // bipartite[root] = true if component is
    bipartite
24
25 // Create a new set for node v
26 void make_set(int v) {
27     parent[v] = {v, 0}; // Self-rooted, even parity to self
28     rank[v] = 0;
29     bipartite[v] = true;
30 }
31
32 // Find the root of v and track parity along the path (0 = even, 1 = odd
    )
33 pair<int, int> find_set(int v) {
34     if (v != parent[v].first) {
35         auto [par, parity] = parent[v];
36         auto root = find_set(par);
37         parent[v] = {root.first, parity ^ root.second}; // Path compression
            with parity update
38     }
39     return parent[v];
40 }
41
42 // Adds an edge between a and b, merges components and checks for odd
    cycles
43 void add_edge(int a, int b) {
44     auto [ra, pa] = find_set(a); // ra = root of a, pa = parity from root
        to a
45     auto [rb, pb] = find_set(b); // rb = root of b, pb = parity from root
        to b
46
47     if (ra == rb) {
48         // Same component: edge (a, b) adds a cycle -> check parity
49         if ((pa ^ pb) == 0) {
50             bipartite[ra] = false; // Found odd-length cycle
51         }
52     } else {
53         // Merge smaller rank under larger
54         if (rank[ra] < rank[rb]) swap(ra, rb), swap(pa, pb);
55
56         // Make rb child of ra; update parity of root
57         parent[rb] = {ra, pa ^ pb ^ 1};
58     }

```

```

59     bipartite[ra] &= bipartite[rb]; // Component is only bipartite if
        both were
60     if (rank[ra] == rank[rb]) rank[ra]++;
61 }
62 }
63
64 // Check if the component containing v is bipartite
65 bool is_bipartite(int v) {
66     return bipartite[find_set(v).first];
67 }

```

3.5 Disjoint Set Union

```

1  /*
2  Disjoint Set Union (Union-Find) with Rollback
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, N-1]
6
7  Features:
8      - Path compression + union by size
9      - Optional rollback to previous state
10     - Supports dynamic connectivity in offline algorithms (e.g., divide
        & conquer)
11
12  Functions:
13      - get(x): find root of x
14      - connected(a, b): check if a and b are in same component
15      - size(x): size of component containing x
16      - unite(x, y): union x and y, returns true if merged
17      - time(): current rollback timestamp
18      - rollback(t): revert to state at timestamp t
19  */
20
21 struct DSU {
22     vector<int> e;                // e[x] < 0 -> root; size = -e[x]; e
        [x] >= 0 -> parent
23     vector<pair<int, int>> st;    // rollback stack: stores (index, old
        value)
24
25     DSU(int N) : e(N, -1) {}
26
27     // Find with path compression

```

```

28     int get(int x) {
29         return e[x] < 0 ? x : get(e[x]);
30     }
31
32     // Check if x and y belong to the same component
33     bool connected(int a, int b) {
34         return get(a) == get(b);
35     }
36
37     // Return size of component containing x
38     int size(int x) {
39         return -e[get(x)];
40     }
41
42     // Union by size, returns true if union happened
43     bool unite(int x, int y) {
44         x = get(x), y = get(y);
45         if (x == y) return false;
46         if (e[x] > e[y]) swap(x, y); // Ensure x has larger size (more
        negative)
47         st.push_back({x, e[x]});
48         st.push_back({y, e[y]});
49         e[x] += e[y];
50         e[y] = x;
51         return true;
52     }
53
54     // Return current rollback timestamp
55     int time() {
56         return (int)st.size();
57     }
58
59     // Roll back to previous state at time t
60     void rollback(int t) {
61         for (int i = time(); i-- > t;) {
62             e[st[i].first] = st[i].second;
63         }
64         st.resize(t);
65     }
66 };

```

3.6 Dynamic Conectivity

```

1  /*
2  Offline Dynamic Connectivity - Segment Tree + Rollback DSU
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, n-1]
6
7  Features:
8  - Handles dynamic edge insertions and deletions over time
9  - Answers queries of type: "how many connected components at time t
10     ?"
11  - All operations are processed offline
12
13  Components:
14  - DSU with rollback (stores a stack of previous states)
15  - Segment tree over time to store active edge intervals
16
17  */
18  // Rollbackable Disjoint Set Union (Union-Find)
19  struct DSU {
20      vector<int> e; // e[x] < 0 means x is a root, and size is -e[x]
21      vector<pair<int, int>> st; // Stack for rollback (stores changed
22          values)
23      int cnt; // Current number of connected components
24
25      DSU() {}
26      DSU(int N) : e(N, -1), cnt(N) {}
27
28      // Find root of x with path compression
29      int get(int x) {
30          return e[x] < 0 ? x : get(e[x]);
31      }
32
33      // Check if x and y are connected
34      bool connected(int a, int b) {
35          return get(a) == get(b);
36      }
37
38      // Size of component containing x
39      int size(int x) {
40          return -e[get(x)];
41      }
42
43      // Union two components; record state for rollback

```

```

42  bool unite(int x, int y) {
43      x = get(x), y = get(y);
44      if (x == y) return false; // Already connected
45
46      if (e[x] > e[y]) swap(x, y); // Union by size: ensure x is larger
47
48      // Save state for rollback
49      st.push_back({x, e[x]});
50      st.push_back({y, e[y]});
51
52      e[x] += e[y]; // Merge y into x
53      e[y] = x;
54      cnt--; // One fewer component
55      return true;
56  }
57
58  // Undo last union
59  void rollback() {
60      auto [x1, y1] = st.back(); st.pop_back();
61      e[x1] = y1;
62      auto [x2, y2] = st.back(); st.pop_back();
63      e[x2] = y2;
64      cnt++;
65  }
66  };
67
68  // Represents a single union operation (on edge u-v)
69  struct query {
70      int v, u;
71      bool united;
72      query(int _v, int _u) : v(_v), u(_u), united(false) {}
73  };
74
75  // Segment Tree for storing edge intervals [l, r]
76  struct QueryTree {
77      vector<vector<query>> t; // Each node stores queries that are active
78          in that time segment
79      DSU dsu;
80      int T; // Number of total operations (time steps)
81
82      QueryTree() {}
83      QueryTree(int _T, int n) : T(_T) {
84          dsu = DSU(n);

```

```

84     t.resize(4 * T + 4);
85 }
86
87 // Internal segment tree add function
88 void add(int v, int l, int r, int ul, int ur, query& q) {
89     if (ul > ur) return;
90     if (l == ul && r == ur) {
91         t[v].push_back(q);
92         return;
93     }
94     int mid = (l + r) / 2;
95     add(2 * v, l, mid, ul, min(ur, mid), q);
96     add(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
97 }
98
99 // Public wrapper: add a query in interval [l, r]
100 void add_query(query q, int l, int r) {
101     add(1, 0, T - 1, l, r, q);
102 }
103
104 // Traverse the segment tree and simulate unions with rollback
105 void dfs(int v, int l, int r, vector<int>& ans) {
106     // Apply all union operations in this segment node
107     for (query& q : t[v]) {
108         q.united = dsu.unite(q.v, q.u);
109     }
110
111     if (l == r) {
112         ans[l] = dsu.cnt; // Save answer for time l
113     } else {
114         int mid = (l + r) / 2;
115         dfs(2 * v, l, mid, ans);
116         dfs(2 * v + 1, mid + 1, r, ans);
117     }
118
119     // Rollback all operations applied in this node
120     for (query& q : t[v]) {
121         if (q.united)
122             dsu.rollback();
123     }
124 }
125 };
126

```

```

127 int main() {
128     int n, k;
129     cin >> n >> k; // n nodes, k operations
130     if (k == 0) return 0;
131     QueryTree st(k, n);
132     map<pair<int, int>, int> mp; // Edge -> start time
133     vector<int> ans(k); // Answers for '?' queries
134     vector<int> qmarks; // Indices of '?' queries
135     // Parse all k operations
136     for (int i = 0; i < k; i++) {
137         char c;
138         cin >> c;
139         if (c == '?') {
140             qmarks.push_back(i); // Save query index
141             continue;
142         }
143         int u, v;
144         cin >> u >> v;
145         u--; v--;
146         if (u > v) swap(u, v); // Normalize edge direction
147
148         if (c == '+') {
149             mp[{u, v}] = i; // Mark time edge is added
150         } else {
151             // Edge removed: store active interval
152             st.add_query(query(u, v), mp[{u, v}], i);
153             mp[{u, v}] = -1;
154         }
155     }
156     // Any edge still active is added until end of timeline
157     for (auto [edge, start] : mp) {
158         if (start != -1) {
159             st.add_query(query(edge.first, edge.second), start, k - 1);
160         }
161     }
162     // Process the tree to compute all '?'-query answers
163     st.dfs(1, 0, k - 1, ans);
164     // Output results of all '?'
165     for (int x : qmarks) {
166         cout << ans[x] << '\n';
167     }
168 }

```

3.7 Fenwick Tree

```

1  /*
2  Fenwick Tree (Binary Indexed Tree)
3  -----
4  Indexing: 0-based
5  Bounds: [0, n-1] inclusive
6  Time Complexity:
7      - add(x, v): O(log n)
8      - sum(x): O(log n) -> prefix sum over [0, x)
9      - rangeSum(l, r): O(log n) -> sum over [l, r)
10     - select(k): O(log n) -> smallest x such that prefix sum >= k (works
11       for monotonic cumulative sums)
12
13     Space Complexity: O(n)
14 */
15 template <typename T>
16 struct Fenwick {
17     int n;
18     std::vector<T> a;
19
20     Fenwick(int n_ = 0) {
21         init(n_);
22     }
23
24     // Initialize BIT of size n
25     void init(int n_) {
26         n = n_;
27         a.assign(n, T{});
28     }
29
30     // Add value 'v' to position 'x'
31     void add(int x, const T &v) {
32         for (int i = x + 1; i <= n; i += i & -i) {
33             a[i - 1] = a[i - 1] + v;
34         }
35     }
36
37     // Compute prefix sum on range [0, x)
38     T sum(int x) const {
39         T ans{};
40         for (int i = x; i > 0; i -= i & -i) {

```

```

41             ans = ans + a[i - 1];
42         }
43         return ans;
44     }
45
46     // Compute sum over range [l, r)
47     T rangeSum(int l, int r) const {
48         return sum(r) - sum(l);
49     }
50
51     // Find the smallest x such that sum[0, x) > k (if exists), or returns
52     n
53     int select(const T &k) const {
54         int x = 0;
55         T cur{};
56         for (int i = 1 << std::lg(n); i; i >>= 1) {
57             if (x + i <= n && cur + a[x + i - 1] <= k) {
58                 cur = cur + a[x + i - 1];
59                 x += i;
60             }
61         }
62         return x;
63     };
64     // Fenwick<int> bit(n);

```

3.8 Fenwick Tree 2D

```

1  /*
2  2D Fenwick Tree (Binary Indexed Tree)
3  -----
4  Indexing: 1-based
5  Bounds: [1, n] inclusive
6  Time Complexity:
7      -update(x, y, v): O(log^2 n)
8      -get(x, y): sum of rectangle [1,1] to [x,y]
9      -get1(x1, y1, x2, y2): sum over rectangle [x1,y1] to [x2,y2]
10     Space Complexity: O(n^2)
11
12     -Can be adapted for rectangular grids by using n, m separately
13 */
14
15 struct Fenwick2D {

```

```

16 vector<vector<ll>> b; // 2D BIT array
17 int n;                // Grid size (1-based)
18
19 Fenwick2D(int _n) : b(_n + 5, vector<ll>(_n + 5, 0)), n(_n) {}
20
21 // Add 'val' to cell (x, y)
22 void update(int x, int y, int val) {
23     for (; x <= n; x += (x & -x)) {
24         for (int j = y; j <= n; j += (j & -j)) {
25             b[x][j] += val;
26         }
27     }
28 }
29
30 // Get sum of rectangle [(1,1) to (x,y)]
31 ll get(int x, int y) {
32     ll ans = 0;
33     for (; x > 0; x -= x & -x) {
34         for (int j = y; j > 0; j -= j & -j) {
35             ans += b[x][j];
36         }
37     }
38     return ans;
39 }
40
41 // Get sum over subrectangle [(x1,y1) to (x2,y2)]
42 ll get1(int x1, int y1, int x2, int y2) {
43     return get(x2, y2) - get(x1-1, y2) - get(x2, y1-1) + get(x1-1, y1-1);
44 }
45 };
46 // Usage example:
47 Fenwick2D fw(n);
48 fw.update(3, 4, 5); // add 5 to (3, 4)
49 ll sum = fw.get(3, 4); // sum from (1,1) to (3,4)
50 ll range = fw.get1(2, 2, 5, 5); // sum in rectangle [(2,2)-(5,5)]

```

3.9 Merge Sort Tree

```

1 /*
2 Merge Sort Tree (Segment Tree of Sorted Arrays)
3 -----
4 Indexing: 0-based
5 Node Bounds: [0, n-1] inclusive

```

```

6 Time Complexity:
7   - build():  $O(n \log n)$ 
8   -  $q(l, r, x)$ :  $O(\log^2 n)$   $\rightarrow$  number of elements  $\leq x$  in  $[l, r]$ 
9 Space Complexity:  $O(n \log n)$ 
10
11 Features:
12   - Supports frequency/count queries: "how many values  $\leq x$  in range  $[l, r]$ ?"
13   - Static array (no point updates unless rebuilt)
14 */
15
16 const int MAXN = 100005; // size of original array
17 const int MAXT = 2 * MAXN; // size of segment tree (2n)
18
19 vector<int> t[MAXT]; // Segment tree: each node holds sorted vector
20 int a[MAXN]; // Original array
21 int n; // Size of array
22
23 // Build merge sort tree (bottom-up)
24 void build() {
25     // Fill leaves
26     for (int i = 0; i < n; i++) {
27         t[i + n].push_back(a[i]);
28     }
29
30     // Merge children into parent
31     for (int i = n - 1; i > 0; i--) {
32         auto &left = t[2 * i], &right = t[2 * i + 1];
33         merge(left.begin(), left.end(), right.begin(), right.end(),
34               back_inserter(t[i]));
35     }
36 }
37
38 // Query how many elements  $\leq 'x'$  in range  $[l, r]$ 
39 int q(int l, int r, int x) {
40     int res = 0;
41     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
42         if (l & 1) {
43             res += upper_bound(t[l].begin(), t[l].end(), x) - t[l].begin();
44             l++;
45         }
46         if (r & 1) {
47             r--;

```

```

47     res += upper_bound(t[r].begin(), t[r].end(), x) - t[r].begin();
48 }
49 }
50 return res;
51 }
52 // Read n and array a, then call build()

```

3.10 Minimum Cartesian Tree

```

1  /*
2  Min Cartesian Tree
3  -----
4  Indexing: 0-based
5  Time Complexity: O(n)
6  Space Complexity: O(n)
7  Tree Properties:
8      - Binary tree where in-order traversal = original array
9      - Tree satisfies min-heap property: parent <= children
10     - 'par[i]': parent of node i
11     - 'sons[i][0]': left child, 'sons[i][1]': right child
12     - 'root': index of root node
13
14  Use cases:
15     - RMQ construction
16     - LCA over RMQ via Cartesian Tree
17  */
18
19 struct min_cartesian_tree {
20     vector<int> par;           // parent for each node
21     vector<vector<int>> sons;  // left and right children
22     int root;
23
24     void init(int n, const vector<int> &arr) {
25         par.assign(n, -1);
26         sons.assign(n, vector<int>(2, -1)); // 0 = left, 1 = right
27         stack<int> st;
28
29         for (int i = 0; i < n; i++) {
30             int last = -1;
31
32             // Maintain increasing stack -> build min Cartesian Tree
33             // Change > to < for max Cartesian Tree
34             while (!st.empty() && arr[st.top()] > arr[i]) {

```

```

35         last = st.top();
36         st.pop();
37     }
38
39     if (!st.empty()) {
40         par[i] = st.top();
41         sons[st.top()][1] = i;
42     }
43     if (last != -1) {
44         par[last] = i;
45         sons[i][0] = last;
46     }
47
48     st.push(i);
49 }
50
51 for (int i = 0; i < n; i++) {
52     if (par[i] == -1) {
53         root = i;
54     }
55 }
56 }
57 };
58 // Example usage:
59 vector<int> a = {4, 2, 6, 1, 3};
60 min_cartesian_tree ct;
61 ct.init(a.size(), a);
62 cout << "Root index: " << ct.root << '\n';

```

3.11 Multi Ordered Set

```

1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/tree_policy.hpp>
3  using namespace __gnu_pbds;
4  template <typename T> using oset = __gnu_pbds::tree<
5      T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
6      __gnu_pbds::tree_order_statistics_node_update
7  >;
8
9  //en main
10
11 oset<pair<int,int>> name;
12 map<int,int> cuenta;

```

```

13     function<void(int)> meter = [&] (int val) {
14         name.insert({val,++cuenta[val]});
15     };
16     auto quitar = [&] (int val) {
17         name.erase({val,cuenta[val]--});
18     };
19
20 meter(x);
21 quitar(y);
22 multiset.order_of_key({y+1,-1})-multiset.order_of_key({x,0})

```

3.12 Ordered Set

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T> using oset = __gnu_pbds::tree<
5     T, __gnu_pbds::null_type, less<T>, __gnu_pbds::rb_tree_tag,
6     __gnu_pbds::tree_order_statistics_node_update
7 >;
8 // order_of_key() primero mayor o igual;
9 // find_by_order() apuntador al elemento k;
10 // oset<pair<int,int>> os;
11 // os.insert({1,2});
12 // os.insert({2,3});
13 // os.insert({5,6});
14 // ll k=os.order_of_key({2,0});
15 // cout<<k<<endl; // 1
16 // pair<int,int> p=os.find_by_order(k);
17 // cout<<p.f<<" "<<p.s<<endl; // 2 3
18 // os.erase(p);
19 // p=os.find_by_order(k);
20 // cout<<p.f<<" "<<p.s<<endl; // 5 6
21
22
23 // check if upperbound or lowerbound does what you want
24 // because they give better time.
25
26 // to allow repetitions
27 #define ordered_set tree<int, null_type,less_equal<int>, rb_tree_tag,
28     tree_order_statistics_node_update>
29 // to not allow repetitions

```

```

30 #define ordered_set tree<int, null_type,less<int>, rb_tree_tag,
31     tree_order_statistics_node_update>
32 //order_of_key(x): number of items are strictly smaller than x
33
34 //find_by_order(k) iterator to the kth element

```

3.13 Palindromic Tree

```

1 /*
2     Palindromic Tree (Eertree)
3     -----
4     Indexing: 0-based
5     Time Complexity:
6         - extend(i): O(1) amortized
7         - calc_occurrences(): O(n)
8     Space Complexity: O(n)
9
10    Features:
11        - Each node represents a unique palindromic substring
12        - Efficient online construction
13        - 'oc': how many times this palindrome occurs as suffix
14        - 'cnt': number of palindromic suffixes in its subtree
15        - 'link': suffix link to longest proper palindromic suffix
16 */
17
18 const int N = 3e5 + 9;
19
20 struct PalindromicTree {
21     struct node {
22         int nxt[26];    // transitions by character
23         int len;        // length of palindrome
24         int st, en;     // start and end indices in string
25         int link;       // suffix link
26         int cnt = 0;    // number of palindromic suffixes
27         int oc = 0;     // occurrences of the palindrome
28     };
29
30     string s;
31     vector<node> t;
32     int sz, last;
33
34     PalindromicTree() {}

```

```

35 PalindromicTree(const string &s) {
36     s = _s;
37     int n = s.size();
38     t.clear();
39     t.resize(n + 5); // up to n + 2 distinct palindromes
40     sz = 2;
41     last = 2;
42     // Root 1: imaginary (-1 length), simplifies links
43     t[1].len = -1;
44     t[1].link = 1;
45     // Root 2: length 0, link to root 1
46     t[2].len = 0;
47     t[2].link = 1;
48 }
49
50 // Extend tree with s[pos], returns 1 if a new node is created
51 int extend(int pos) {
52     int cur = last;
53     int ch = s[pos] - 'a';
54     // Find longest suffix palindrome that can be extended
55     while (true) {
56         int curlen = t[cur].len;
57         if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) break;
58         cur = t[cur].link;
59     }
60
61     if (t[cur].nxt[ch]) {
62         // Already exists
63         last = t[cur].nxt[ch];
64         t[last].oc++;
65         return 0;
66     }
67     // Create new node
68     sz++;
69     last = sz;
70     t[sz].oc = 1;
71     t[sz].len = t[cur].len + 2;
72     t[cur].nxt[ch] = sz;
73     t[sz].en = pos;
74     t[sz].st = pos - t[sz].len + 1;
75
76     if (t[sz].len == 1) {
77         t[sz].link = 2;

```

```

78     t[sz].cnt = 1;
79     return 1;
80 }
81 // Compute suffix link
82 while (true) {
83     cur = t[cur].link;
84     int curlen = t[cur].len;
85     if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
86         t[sz].link = t[cur].nxt[ch];
87         break;
88     }
89 }
90 t[sz].cnt = 1 + t[t[sz].link].cnt;
91 return 1;
92 }
93 // Accumulate total occurrences for each palindrome node
94 void calc_occurrences() {
95     for (int i = sz; i >= 3; i--) {
96         t[t[i].link].oc += t[i].oc;
97     }
98 }
99 };
100
101 int main() {
102     string s;
103     cin >> s;
104     PalindromicTree t(s);
105     for (int i = 0; i < s.size(); i++) {
106         t.extend(i);
107     }
108     t.calc_occurrences();
109     long long total = 0;
110     for (int i = 3; i <= t.sz; i++) {
111         total += t.t[i].oc;
112     }
113     cout << total << '\n'; // Total palindromic substrings
114     return 0;
115 }

```

3.14 Persistent Array

```

1  /*
2  Persistent Array (via Persistent Segment Tree)

```

```

3 -----
4 Indexing: 0-based
5 Bounds: [0, n-1]
6 Time Complexity:
7   - point update: O(log n)
8   - point query: O(log n)
9 Space Complexity: O(log n) per update/version
10
11 Features:
12   - Supports point updates with full version history
13   - Allows querying any version at any index
14 */
15
16 struct Node {
17     int val;
18     Node *l, *r;
19
20     // Leaf node with value
21     Node(int x) : val(x), l(nullptr), r(nullptr) {}
22
23     // Internal node with children (value is not used)
24     Node(Node *ll, Node *rr) : val(0), l(ll), r(rr) {}
25 };
26
27 int n;
28 int a[100001]; // Initial array
29 Node *roots[100001]; // Roots of all versions (0-based)
30
31 // Build version 0 from initial array
32 Node *build(int l = 0, int r = n - 1) {
33     if (l == r) return new Node(a[l]);
34     int mid = (l + r) / 2;
35     return new Node(build(l, mid), build(mid + 1, r));
36 }
37
38 // Create new version with a[pos] = val
39 Node *update(Node *node, int pos, int val, int l = 0, int r = n - 1) {
40     if (l == r) return new Node(val);
41     int mid = (l + r) / 2;
42     if (pos <= mid)
43         return new Node(update(node->l, pos, val, l, mid), node->r);
44     else
45         return new Node(node->l, update(node->r, pos, val, mid + 1, r));

```

```

46 }
47
48 // Query value at position 'pos' in a given version (node)
49 int query(Node *node, int pos, int l = 0, int r = n - 1) {
50     if (l == r) return node->val;
51     int mid = (l + r) / 2;
52     if (pos <= mid) return query(node->l, pos, l, mid);
53     else return query(node->r, pos, mid + 1, r);
54 }
55
56 // External helper: get value at index in version
57 int get_item(int index, int version) {
58     return query(roots[version], index);
59 }
60
61 // External helper: make new version based on 'prev_version', updating
62 // one index
63 void update_item(int index, int value, int prev_version, int new_version)
64     {
65     roots[new_version] = update(roots[prev_version], index, value);
66 }
67
68 // Initializes version 0 from given array
69 void init_arr(int nn, int *init) {
70     n = nn;
71     for (int i = 0; i < n; i++) a[i] = init[i];
72     roots[0] = build();
73 }

```

3.15 Persistent Segment Tree

```

1 /*
2 Persistent Segment Tree (Point Updates, Range Queries)
3 -----
4 Indexing: 1-based
5 Bounds: [1, n]
6 Time Complexity:
7   - Build: O(n)
8   - Point update: O(log n) -> returns new version
9   - Range query: O(log n)
10   - Copy version: O(1)
11
12 Features:

```

```

13     - Each update creates a new tree version with shared unchanged nodes
14     - Supports querying over any version
15     - Useful in rollback problems, version history, and functional
        programming
16 */
17
18 struct Node {
19     ll val;      // segment sum
20     Node *l, *r;
21
22     // Leaf node
23     Node(ll x) : val(x), l(nullptr), r(nullptr) {}
24
25     // Internal node with children
26     Node(Node *_l, Node *_r) {
27         l = _l;
28         r = _r;
29         val = 0;
30         if (l) val += l->val;
31         if (r) val += r->val;
32     }
33
34     // Version clone (used when copying tree version directly)
35     Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
36 };
37
38 int n, sz = 1;
39 ll a[200001];      // Input array (1-indexed)
40 Node *t[200001];   // Roots of different versions (t[version_id])
41
42 // Build initial segment tree from array a[1..n]
43 Node *build(int l = 1, int r = n) {
44     if (l == r) return new Node(a[l]);
45     int mid = (l + r) / 2;
46     return new Node(build(l, mid), build(mid + 1, r));
47 }
48
49 // Update position 'pos' with 'val' in given 'node' version
50 Node *update(Node *node, int pos, int val, int l = 1, int r = n) {
51     if (l == r) return new Node(val); // replace leaf
52     int mid = (l + r) / 2;
53     if (pos <= mid)
54         return new Node(update(node->l, pos, val, l, mid), node->r);

```

```

55     else
56         return new Node(node->l, update(node->r, pos, val, mid + 1, r));
57 }
58
59 // Query sum over range [a, b] in given version
60 ll query(Node *node, int a, int b, int l = 1, int r = n) {
61     if (r < a || l > b) return 0;      // No overlap
62     if (l >= a && r <= b) return node->val; // Total overlap
63     int mid = (l + r) / 2;
64     return query(node->l, a, b, l, mid) + query(node->r, a, b, mid + 1, r)
65         ;
66 }
67
68 int main() {
69     ios_base::sync_with_stdio(false); cin.tie(NULL);
70
71     int q;
72     cin >> n >> q;
73     for (int i = 1; i <= n; i++) {
74         cin >> a[i];
75     }
76
77     // Build version 0
78     t[0] = build();
79     sz = 1;
80
81     while (q--) {
82         int ty;
83         cin >> ty;
84
85         if (ty == 1) {
86             // Point update: create new version from t[k] with a[pos] = x
87             int k, pos, x;
88             cin >> k >> pos >> x;
89             t[k] = update(t[k], pos, x);
90
91         } else if (ty == 2) {
92             // Range query on version k over [l, r]
93             int k, l, r;
94             cin >> k >> l >> r;
95             cout << query(t[k], l, r) << '\n';
96
97         } else if (ty == 3) {

```

```

97     // Clone version k into new version
98     int k;
99     cin >> k;
100    t[sz++] = new Node(t[k]);
101    }
102    }
103    return 0;
104    }

```

3.16 Segment Tree

```

1  /*
2  Segment Tree (Iterative, Range Minimum Query)
3  -----
4  Indexing: 0-based
5  Bounds: [0, n-1] inclusive
6  Time Complexity:
7      - update(pos, val): O(log n)
8      - get(l, r): O(log n) -> query min in range [l, r)
9  Space Complexity: O(2n)
10 */
11 struct SegmentTree {
12     vector<ll> a; // segment tree array
13     int n;       // number of elements in original array
14
15     SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}
16     // Update position 'pos' to value 'val'
17     void update(int pos, ll val) {
18         pos += n; // move to leaf
19         a[pos] = val; // set value
20         for (pos /= 2; pos > 0; pos /= 2) {
21             a[pos] = min(a[2 * pos], a[2 * pos + 1]); // update parent
22         }
23     }
24     // Get minimum value in range [l, r)
25     ll get(int l, int r) {
26         ll res = 1e18;
27         for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
28             if (l & 1) res = min(res, a[l++]); // if l is right child
29             if (r & 1) res = min(res, a[--r]); // if r is left child
30         }
31         return res;
32     }
33 }

```

```

33 };

```

3.17 Segment Tree 2D

```

1  /*
2  2D Segment Tree (Sum over Rectangles)
3  -----
4  Indexing: 0-based
5  Grid Size: n * m
6  Time Complexity:
7      - build: O(nm log n log m)
8      - point update: O(log n log m)
9      - range query [x1..x2][y1..y2]: O(log n log m)
10 Space Complexity: O(4n x 4m)
11 */
12
13 const int MAXN = 505;
14 int n, m; // grid dimensions
15 int a[MAXN][MAXN]; // input grid
16 int t[4 * MAXN][4 * MAXN]; // segment tree
17
18 // Build the tree along y-axis (internal to each x-interval)
19 void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
20     if (ly == ry) {
21         if (lx == rx)
22             t[vx][vy] = a[lx][ly];
23         else
24             t[vx][vy] = t[vx * 2][vy] + t[vx * 2 + 1][vy];
25     } else {
26         int my = (ly + ry) / 2;
27         build_y(vx, lx, rx, vy * 2, ly, my);
28         build_y(vx, lx, rx, vy * 2 + 1, my + 1, ry);
29         t[vx][vy] = t[vx][vy * 2] + t[vx][vy * 2 + 1];
30     }
31 }
32
33 // Build the tree along x-axis and call build_y for each
34 void build_x(int vx, int lx, int rx) {
35     if (lx != rx) {
36         int mx = (lx + rx) / 2;
37         build_x(vx * 2, lx, mx);
38         build_x(vx * 2 + 1, mx + 1, rx);
39     }

```

```

40   build_y(vx, lx, rx, 1, 0, m - 1);
41 }
42
43 // Query along y-axis in a fixed x-node
44 int sum_y(int vx, int vy, int tly, int try_, int ly, int ry) {
45     if (ly > ry) return 0;
46     if (ly == tly && ry == try_) return t[vx][vy];
47     int tmy = (tly + try_) / 2;
48     return sum_y(vx, vy * 2, tly, tmy, ly, min(ry, tmy))
49         + sum_y(vx, vy * 2 + 1, tmy + 1, try_, max(ly, tmy + 1), ry);
50 }
51
52 // Query sum in rectangle [lx..rx][ly..ry]
53 int sum_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
54     if (lx > rx) return 0;
55     if (lx == tlx && trx == rx)
56         return sum_y(vx, 1, 0, m - 1, ly, ry);
57     int tmx = (tlx + trx) / 2;
58     return sum_x(vx * 2, tlx, tmx, lx, min(rx, tmx), ly, ry)
59         + sum_x(vx * 2 + 1, tmx + 1, trx, max(lx, tmx + 1), rx, ly, ry);
60 }
61
62 // Update along y-axis for fixed x-node
63 void update_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int
    y, int new_val) {
64     if (ly == ry) {
65         if (lx == rx)
66             t[vx][vy] = new_val;
67         else
68             t[vx][vy] = t[vx * 2][vy] + t[vx * 2 + 1][vy];
69     } else {
70         int my = (ly + ry) / 2;
71         if (y <= my)
72             update_y(vx, lx, rx, vy * 2, ly, my, x, y, new_val);
73         else
74             update_y(vx, lx, rx, vy * 2 + 1, my + 1, ry, x, y, new_val);
75         t[vx][vy] = t[vx][vy * 2] + t[vx][vy * 2 + 1];
76     }
77 }
78
79 // Update point (x, y) to new_val
80 void update_x(int vx, int lx, int rx, int x, int y, int new_val) {
81     if (lx != rx) {

```

```

82         int mx = (lx + rx) / 2;
83         if (x <= mx)
84             update_x(vx * 2, lx, mx, x, y, new_val);
85         else
86             update_x(vx * 2 + 1, mx + 1, rx, x, y, new_val);
87     }
88     update_y(vx, lx, rx, 1, 0, m - 1, x, y, new_val);
89 }

```

3.18 Segment Tree Dynamic

```

1  /*
2  Dynamic Segment Tree (Point Add, Range Sum)
3  -----
4  Indexing: [0, INF) or any large bounded range
5  Time Complexity:
6      - add(k, x): O(log U)
7      - get_sum(l, r): O(log U)
8        where U = range size (e.g., 1e9 if implicit bounds)
9
10 Space Complexity: O(nodes visited or created) -> worst O(log U) per
    operation
11 */
12
13 struct Vertex {
14     int left, right;           // interval [left, right)
15     int sum = 0;               // sum of elements in this interval
16     Vertex *left_child = nullptr, *right_child = nullptr;
17
18     Vertex(int lb, int rb) {
19         left = lb;
20         right = rb;
21     }
22
23     // Create children lazily only when needed
24     void extend() {
25         if (!left_child && left + 1 < right) {
26             int mid = (left + right) / 2;
27             left_child = new Vertex(left, mid);
28             right_child = new Vertex(mid, right);
29         }
30     }
31 }

```

```

32 // Add 'x' to position 'k'
33 void add(int k, int x) {
34     extend();
35     sum += x;
36     if (left_child) {
37         if (k < left_child->right)
38             left_child->add(k, x);
39         else
40             right_child->add(k, x);
41     }
42 }
43
44 // Get sum over interval [lq, rq]
45 int get_sum(int lq, int rq) {
46     if (lq <= left && right <= rq)
47         return sum;
48     if (rq <= left || right <= lq)
49         return 0;
50     extend();
51     return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq);
52 }
53 };
54
55 Vertex *root = new Vertex(0, 1e9); // Range [0, 1e9]
56 root->add(5, 10); // a[5] += 10
57 root->add(1000, 20); // a[1000] += 20
58 cout << root->get_sum(0, 10) << '\n'; // sum of [0, 10] = 10
59 cout << root->get_sum(0, 2000) << '\n'; // sum of [0, 2000] = 30

```

3.19 Segment Tree Lazy Types

```

1 struct max_t {
2     long long val;
3     static const long long null_v = -9223372036854775807LL;
4
5     max_t(): val(0) {}
6     max_t(long long v): val(v) {}
7
8     max_t op(max_t& other) {
9         return max_t(max(val, other.val));
10    }
11
12    max_t lazy_op(max_t& v, int size) {

```

```

13        return max_t(val + v.val);
14    }
15 };
16
17
18 struct min_t {
19     long long val;
20     static const long long null_v = 9223372036854775807LL;
21
22     min_t(): val(0) {}
23     min_t(long long v): val(v) {}
24
25     min_t op(min_t& other) {
26         return min_t(min(val, other.val));
27     }
28
29     min_t lazy_op(min_t& v, int size) {
30         return min_t(val + v.val);
31     }
32 };
33
34
35 struct sum_t {
36     long long val;
37     static const long long null_v = 0;
38
39     sum_t(): val(0) {}
40     sum_t(long long v): val(v) {}
41
42     sum_t op(sum_t& other) {
43         return sum_t(val + other.val);
44     }
45
46     sum_t lazy_op(sum_t& v, int size) {
47         return sum_t(val + v.val * size);
48     }
49 };

```

3.20 Segment Tree Lazy

```

1 /*
2  Lazy Segment Tree (Range Update, Range Query)
3  -----

```

```

4   Indexing: 0-based
5   Bounds: [0, n-1]
6   Time Complexity:
7       - build: O(n)
8       - update(l, r, v): O(log n)
9       - query(l, r): O(log n)
10  Space Complexity: O(4n)
11  */
12
13  // See SegTreeLazy_types for num_t structs
14
15  const num_t num_t::null_v = num_t(0);
16
17  template <typename num_t>
18  struct segtree {
19      int n;
20      vector<num_t> tree, lazy;
21
22      // Initialize segment tree with array of size s
23      void init(int s, long long* arr) {
24          n = s;
25          tree.assign(4 * n, num_t());
26          lazy.assign(4 * n, num_t());
27          init(0, 0, n - 1, arr);
28      }
29
30      // Build segment tree from array
31      num_t init(int i, int l, int r, long long* arr) {
32          if (l == r) return tree[i] = num_t(arr[l]);
33
34          int mid = (l + r) / 2;
35          num_t left = init(2 * i + 1, l, mid, arr);
36          num_t right = init(2 * i + 2, mid + 1, r, arr);
37          return tree[i] = left.op(right);
38      }
39
40      // Public wrapper: update range [l, r] with value v
41      void update(int l, int r, num_t v) {
42          if (l > r) return;
43          update(0, 0, n - 1, l, r, v);
44      }
45  }
46

```

```

47  // Internal recursive update
48  num_t update(int i, int tl, int tr, int ql, int qr, num_t v) {
49      eval_lazy(i, tl, tr);
50
51      if (tr < ql || qr < tl) return tree[i]; // no overlap
52      if (ql <= tl && tr <= qr) {
53          lazy[i].val += v.val;
54          eval_lazy(i, tl, tr);
55          return tree[i];
56      }
57
58      int mid = (tl + tr) / 2;
59      num_t a = update(2 * i + 1, tl, mid, ql, qr, v);
60      num_t b = update(2 * i + 2, mid + 1, tr, ql, qr, v);
61      return tree[i] = a.op(b);
62  }
63
64  // Public wrapper: query sum in range [l, r]
65  num_t query(int l, int r) {
66      if (l > r) return num_t::null_v;
67      return query(0, 0, n - 1, l, r);
68  }
69
70  // Internal recursive query
71  num_t query(int i, int tl, int tr, int ql, int qr) {
72      eval_lazy(i, tl, tr);
73
74      if (ql <= tl && tr <= qr) return tree[i]; // total overlap
75      if (tr < ql || qr < tl) return num_t::null_v; // no overlap
76
77      int mid = (tl + tr) / 2;
78      num_t a = query(2 * i + 1, tl, mid, ql, qr);
79      num_t b = query(2 * i + 2, mid + 1, tr, ql, qr);
80      return a.op(b);
81  }
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  }

```

```

90     lazy[i] = num_t(); // reset lazy at this node
91 }
92 };

```

3.21 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
9     range sum queries
10  - Properly prioritizes lazy set > lazy add
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 }
72
73 // Add 'val' to all elements in [a, b]
74 void add(int p, int l, int r, int a, int b, ll val) {
75     if (a > r || b < l) return;
76     if (a <= l && r <= b) {
77         tree[p].val += (r - l + 1) * val;
78         if (tree[p].lzSet == 0) tree[p].lzAdd += val;
79         else tree[p].lzSet += val;

```

```

80     return;
81 }
82 int mid = (l + r) >> 1;
83 pushdown(p, l, mid, r);
84 add(lc, l, mid, a, b, val);
85 add(rc, mid + 1, r, a, b, val);
86 pushup(p);
87 }
88
89 // Set all elements in [a, b] to 'val'
90 void set(int p, int l, int r, int a, int b, ll val) {
91     if (a > r || b < l) return;
92     if (a <= l && r <= b) {
93         tree[p].val = (r - l + 1) * val;
94         tree[p].lzAdd = 0;
95         tree[p].lzSet = val;
96         return;
97     }
98     int mid = (l + r) >> 1;
99     pushdown(p, l, mid, r);
100    set(lc, l, mid, a, b, val);
101    set(rc, mid + 1, r, a, b, val);
102    pushup(p);
103 }
104
105 // Query sum over [a, b]
106 ll query(int p, int l, int r, int a, int b) {
107     if (a > r || b < l) return 0;
108     if (a <= l && r <= b) return tree[p].val;
109     int mid = (l + r) >> 1;
110     pushdown(p, l, mid, r);
111     return query(lc, l, mid, a, b) + query(rc, mid + 1, r, a, b);
112 }
113 // Example usage
114 N = 5;
115 a[0] = 2, a[1] = 4, a[2] = 3, a[3] = 1, a[4] = 5;
116 build(1, 0, N - 1);
117 set(1, 0, N - 1, 1, 3, 7); // a[1..3] = 7
118 add(1, 0, N - 1, 2, 4, 2); // a[2..4] += 2
119 cout << query(1, 0, N - 1, 0, 4) << '\n'; // total sum

```

3.22 Segment Tree Max Subarray Sum

```

1  const ll inf=1e18;
2
3  struct Node {
4      ll maxi, l_max, r_max, sum;
5
6      Node(ll _maxi, ll _l_max, ll _r_max, ll _sum){
7          maxi=_maxi;
8          l_max=_l_max;
9          r_max=_r_max;
10         sum=_sum;
11     }
12
13     Node operator+(Node b) {
14         return {max(max(maxi, b.maxi), r_max + b.l_max),
15                 max(l_max, sum + b.l_max), max(b.r_max, r_max + b.sum),
16                 sum + b.sum};
17     }
18 };
19
20 struct SegmentTreeMaxSubSum{
21     int n;
22     vector<Node> t;
23
24     SegmentTreeMaxSubSum(int _n) : n(_n), t(2 * _n, Node(-inf, -inf, -inf,
25         -inf)) {}
26
27     void update(int pos, ll val) {
28         t[pos += n] = Node(val, val, val, val);
29         for (pos>>=1; pos ; pos >>= 1) {
30             t[pos] = t[2*pos]+t[2*pos+1];
31         }
32     }
33
34     Node query(int l, int r) {
35         Node node_l = Node(-inf, -inf, -inf, -inf);
36         Node node_r = Node(-inf, -inf, -inf, -inf);
37         for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
38             if (l & 1) {
39                 node_l=node_l+t[l++];
40             }
41             if (r & 1) {
42                 node_r=t[--r]+node_r;

```

```

43     }
44 }
45 return node_l+node_r;
46 }
47 };

```

3.23 Segment Tree Range Update

```

1  /*
2  Segment Tree (Range Min Update, Point Query)
3  -----
4  Indexing: 0-based
5  Bounds: [0, n-1]
6  Time Complexity:
7    - update(l, r, val): O(log n) -> applies min(val) over [l, r)
8    - get(pos): O(log n) -> minimum affecting position pos
9  Space Complexity: O(2n)
10 */
11
12 struct SegmentTree {
13     vector<ll> a; // a[i] = min value affecting segment i
14     int n;
15
16     SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}
17
18     // Get the effective minimum at position 'pos'
19     ll get(int pos) {
20         ll res = 1e18;
21         for (pos += n; pos > 0; pos >>= 1) {
22             res = min(res, a[pos]);
23         }
24         return res;
25     }
26
27     // Apply min(val) to all positions in [l, r)
28     void update(int l, int r, ll val) {
29         for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
30             if (l & 1) {
31                 a[l] = min(a[l], val);
32                 l++;
33             }
34             if (r & 1) {
35                 --r;

```

```

36         a[r] = min(a[r], val);
37     }
38 }
39 }
40 };

```

3.24 Segment Tree Struct Types

```

1  // Sum segment tree
2  struct sum_t{
3      ll val;
4      static const long long null_v = 0;
5
6      sum_t(): val(null_v) {}
7      sum_t(long long v): val(v) {}
8
9      sum_t operator + (const sum_t &a) const {
10         sum_t ans;
11         ans.val = val + a.val;
12         return ans;
13     }
14 };
15 // Min segment tree
16 struct min_t{
17     ll val;
18     static const long long null_v = 1e18;
19
20     min_t(): val(null_v) {}
21     min_t(long long v): val(v) {}
22
23     min_t operator + (const min_t &a) const {
24         min_t ans;
25         ans.val = min(val, a.val);
26         return ans;
27     }
28 };
29 // Max segment tree
30 struct max_t{
31     ll val;
32     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 };

```

3.25 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
8     void init(int s, vector<ll> arr)
9     {
10         n = s;
11         k = 0;
12         while ((1 << k) < n)
13             k++;
14         tree = vector<num_t>(2 * (1 << k) + 1);
15         for (int i = 0; i < n; i++)
16         {
17             tree[(1 << k) + i] = arr[i];
18         }
19         for (int i = (1 << k) - 1; i > 0; i--)

```

```

20     {
21         tree[i] = tree[i * 2] + tree[i * 2 + 1];
22     }
23 }
24
25 void update(int a, ll b)
26 {
27     a += (1 << k);
28     tree[a] = b;
29     for (a /= 2; a >= 1; a /= 2)
30     {
31         tree[a] = tree[a * 2] + tree[a * 2 + 1];
32     }
33 }
34 num_t find(int a, int b)
35 {
36     a += (1 << k);
37     b += (1 << k);
38     num_t s;
39     while (a <= b)
40     {
41         if (a % 2 == 1)
42             s = s + tree[a++];
43         if (b % 2 == 0)
44             s = s + tree[b--];
45         a /= 2;
46         b /= 2;
47     }
48     return s;
49 }
50 };

```

3.26 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)
9         - get(L, R): O(log n) -> min value in [L, R]

```

```

10 - query(L, R, val): O(log n) -> find first index in [L, R] where a[i]
    ] >= val
11
12 Features:
13 - Stores original value array in segment tree form
14 - Maps original indices to tree positions for fast updates
15 - Allows efficient walk to find constrained elements (e.g. lower
    bound >= val)
16
17 */
18 struct SegmentTreeWalk {
19     vector<ll> a;           // segment tree values
20     vector<int> final_pos;  // maps index i to position in tree (leaf)
21     int n;
22
23     SegmentTreeWalk(int _n) : a(4 * _n, 1e18), final_pos(_n), n(_n) {}
24
25     // Build segment tree from array 'vals[0..n-1]', start with node=1, l
26     =0, r=n-1
27     void build(int l, int r, int node, const vector<ll> &vals) {
28         if (l == r) {
29             final_pos[l] = node;
30             a[node] = vals[l];
31         } else {
32             int mid = (l + r) / 2;
33             build(l, mid, node * 2, vals);
34             build(mid + 1, r, node * 2 + 1, vals);
35             a[node] = min(a[node * 2], a[node * 2 + 1]);
36         }
37     }
38
39     // Update value at original index 'pos' to 'val'
40     void update(int pos, ll val) {
41         pos = final_pos[pos]; // leaf position
42         a[pos] = val;
43         for (pos /= 2; pos > 0; pos /= 2)
44             a[pos] = min(a[pos * 2], a[pos * 2 + 1]);
45     }
46
47     // Get min value in [L, R], with current node interval [l, r] and root
48     'node'
49     ll get(int l, int r, int L, int R, int node) {
50         if (L > R) return 1e18;

```

```

49     if (l == L && r == R) return a[node];
50     int mid = (l + r) / 2;
51     return min(
52         get(l, mid, L, min(R, mid), node * 2),
53         get(mid + 1, r, max(L, mid + 1), R, node * 2 + 1)
54     );
55 }
56
57 // Find first position in [L, R] with a[i] >= val, starting from node
58 interval [l, r]
59 pair<ll, ll> query(int l, int r, int L, int R, int node, int val) {
60     if (l > R || r < L) return {-1, 0};           // out of query
61     bounds
62     if (a[node] < val) return {-1, 0};           // all values < val
63     if (l == r) return {a[node], l};             // leaf node that
64     satisfies
65
66     int mid = (l + r) / 2;
67     auto left = query(l, mid, L, R, node * 2, val);
68     if (left.first != -1) return left;
69     return query(mid + 1, r, L, R, node * 2 + 1, val);
70 }
71 };
72 // Example usage:
73 int n = 8;
74 vector<ll> vals = {4, 2, 7, 1, 9, 5, 6, 3};
75 SegmentTreeWalk st(n);
76 st.build(0, n - 1, 1, vals);

```

3.27 Sparse Table

```

1  /*
2  Sparse Table (Range Minimum Query)
3  -----
4  Indexing: 0-based
5  Bounds: [0, n-1]
6  Time Complexity:
7      - Build: O(n log n)
8      - Query: O(1)
9  Space Complexity: O(n log n)
10
11 Features:
12     - Immutable RMQ (no updates)

```

```

13     - Works for idempotent operations like min, max, gcd
14 */
15
16 const int MAXN = 100005;
17 const int K = 30; // floor(log2(MAXN))
18 int lg[MAXN + 1]; // log base 2 of each i
19 int st[K + 1][MAXN]; // st[k][i] = min in [i, i + 2^k - 1]
20
21 vector<int> a; // input array
22 int n;
23
24 // Returns min in range [L, R]
25 int mini(int L, int R) {
26     int len = R - L + 1;
27     int i = lg[len];
28     return min(st[i][L], st[i][R - (1 << i) + 1]);
29 }
30
31 int main() {
32     cin >> n;
33     a.resize(n);
34     for (int i = 0; i < n; i++) cin >> a[i];
35     // Precompute binary logs
36     lg[1] = 0;
37     for (int i = 2; i <= n; i++) {
38         lg[i] = lg[i / 2] + 1;
39     }
40     // Initialize 2^0 intervals
41     for (int i = 0; i < n; i++) {
42         st[0][i] = a[i];
43     }
44     // Build sparse table
45     for (int k = 1; k <= K; k++) {
46         for (int i = 0; i + (1 << k) <= n; i++) {
47             st[k][i] = min(st[k - 1][i], st[k - 1][i + (1 << (k - 1))]);
48         }
49     }
50     // Example usage
51     int q; cin >> q;
52     while (q--) {
53         int l, r;
54         cin >> l >> r;
55         cout << mini(l, r) << '\n';

```

```

56     }
57     return 0;
58 }

```

3.28 Square Root Decomposition

```

1  /*
2  Sqrt Decomposition (String Block Cut and Move)
3  -----
4  Operation:
5      - Supports moving substrings using block cut logic
6      - Rebuilds when too many blocks (for performance)
7
8  Indexing: 0-based
9  String Bounds: [0, n)
10 Time Complexity:
11     - cut(a, b): O(sqrt(n))
12     - rebuildDecomp(): O(n)
13 When to rebuild: after too many block splits
14
15 Use case: performing multiple cut/paste operations efficiently on
16           large strings
17 */
18 const int MAXI = 350; // = sqrt(n), for n up to 1e5
19
20 int n, numBlocks;
21 string s;
22
23 struct Block {
24     int l, r; // indices into string s
25     int sz() const { return r - l; }
26 };
27
28 Block blocks[2 * MAXI]; // current block array
29 Block newBlocks[2 * MAXI]; // used temporarily during cutting
30
31 // Rebuilds the entire decomposition into 1 block (or balanced ones)
32 void rebuildDecomp() {
33     string newS = s;
34     int k = 0;
35     // Flatten string using current block structure
36     for (int i = 0; i < numBlocks; i++) {

```

```

37     for (int j = blocks[i].l; j < blocks[i].r; j++) {
38         newS[k++] = s[j];
39     }
40 }
41 // Reset to one big block
42 numBlocks = 1;
43 blocks[0] = {0, n};
44 s = newS;
45 }
46
47 // Cut [a, b) into a separate region and reorder it to the end
48 void cut(int a, int b) {
49     int pos = 0, curBlock = 0;
50     // Pass 1: Split blocks to isolate [a, b)
51     for (int i = 0; i < numBlocks; i++) {
52         Block B = blocks[i];
53         bool containsA = (pos < a && pos + B.sz() > a);
54         bool containsB = (pos < b && pos + B.sz() > b);
55         int cutA = B.l + a - pos;
56         int cutB = B.l + b - pos;
57
58         if (containsA && containsB) {
59             newBlocks[curBlock++] = {B.l, cutA};
60             newBlocks[curBlock++] = {cutA, cutB};
61             newBlocks[curBlock++] = {cutB, B.r};
62         } else if (containsA) {
63             newBlocks[curBlock++] = {B.l, cutA};
64             newBlocks[curBlock++] = {cutA, B.r};
65         } else if (containsB) {
66             newBlocks[curBlock++] = {B.l, cutB};
67             newBlocks[curBlock++] = {cutB, B.r};
68         } else {
69             newBlocks[curBlock++] = B;
70         }
71
72         pos += B.sz();
73     }
74
75     // Pass 2: Reorder - move [a, b) to the end
76     pos = 0;
77     numBlocks = 0;
78
79     // First add all blocks not in [a, b)

```

```

80     for (int i = 0; i < curBlock; i++) {
81         if (pos < a || pos >= b)
82             blocks[numBlocks++] = newBlocks[i];
83         pos += newBlocks[i].sz();
84     }
85
86     // Then add blocks in [a, b)
87     pos = 0;
88     for (int i = 0; i < curBlock; i++) {
89         if (pos >= a && pos < b)
90             blocks[numBlocks++] = newBlocks[i];
91         pos += newBlocks[i].sz();
92     }
93 }
94 // Example usage
95 int main() {
96     cin >> s;
97     n = s.size();
98     numBlocks = 1;
99     blocks[0] = {0, n};
100
101     int q; cin >> q;
102     while (q--) {
103         int a, b;
104         cin >> a >> b;
105         cut(a, b); // move [a, b) to the end
106
107         if (numBlocks > MAXI) rebuildDecomp();
108     }
109
110     rebuildDecomp(); // flatten before output
111     cout << s << '\n';
112 }

```

3.29 Treap

```

1 struct Node {
2     Node *l = 0, *r = 0;
3     int val, y, c = 1;
4     Node(int val) : val(val), y(rand()) {}
5     void recalc();
6 };
7

```

```

8  int cnt(Node* n) { return n ? n->c : 0; }
9  void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
10
11 template<class F> void each(Node* n, F f) {
12     if (n) { each(n->l, f); f(n->val); each(n->r, f); }
13 }
14
15 pair<Node*, Node*> split(Node* n, int k) {
16     if (!n) return {};
17     if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
18         auto pa = split(n->l, k);
19         n->l = pa.second;
20         n->recalc();
21         return {pa.first, n};
22     } else {
23         auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
24         n->r = pa.first;
25         n->recalc();
26         return {n, pa.second};
27     }
28 }
29
30 Node* merge(Node* l, Node* r) {
31     if (!l) return r;
32     if (!r) return l;
33     if (l->y > r->y) {
34         l->r = merge(l->r, r);
35         l->recalc();
36         return l;
37     } else {
38         r->l = merge(l, r->l);
39         r->recalc();
40         return r;
41     }
42 }
43
44 Node* ins(Node* t, Node* n, int pos) {
45     auto pa = split(t, pos);
46     return merge(merge(pa.first, n), pa.second);
47 }
48
49 // Example application: move the range [l, r) to index k
50 void move(Node*& t, int l, int r, int k) {

```

```

51     Node *a, *b, *c;
52     tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
53     if (k <= l) t = merge(ins(a, b, k), c);
54     else t = merge(a, ins(c, b, k - r));
55 }
56
57 // Usage
58 // create treap
59 // Node* name=nullptr;
60 // insert element
61 // name=ins(name, new Node(val), pos);
62 // Node* x = new Node(val);
63 // name = ins(name, x, pos);
64 // merge two treaps (name before x)
65 // name=merge(name, x);
66 // split treap (this will split treap in two treaps,
67 // first with elements [0, pos) and second with elements [pos, n))
68 // pa will be pair of two treaps
69 // auto pa = split(name, pos);
70 // move range [l, r) to index k
71 // move(name, l, r, k);
72 // iterate over treap
73 // each(name, [&](int val) {
74 //     cout << val << ' ';
75 // });

```

3.30 Treap 2

```

1  typedef struct item * pitem;
2  struct item {
3      int prior, value, cnt;
4      bool rev;
5      pitem l, r;
6  };
7
8  int cnt (pitem it) {
9      return it ? it->cnt : 0;
10 }
11
12 void upd_cnt (pitem it) {
13     if (it)
14         it->cnt = cnt(it->l) + cnt(it->r) + 1;
15 }

```

```

16
17 void push (pitem it) {
18     if (it && it->rev) {
19         it->rev = false;
20         swap (it->l, it->r);
21         if (it->l) it->l->rev ^= true;
22         if (it->r) it->r->rev ^= true;
23     }
24 }
25
26 void merge (pitem & t, pitem l, pitem r) {
27     push (l);
28     push (r);
29     if (!l || !r)
30         t = l ? l : r;
31     else if (l->prior > r->prior)
32         merge (l->r, l->r, r), t = l;
33     else
34         merge (r->l, l, r->l), t = r;
35     upd_cnt (t);
36 }
37
38 void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
39     if (!t)
40         return void( l = r = 0 );
41     push (t);
42     int cur_key = add + cnt(t->l);
43     if (key <= cur_key)
44         split (t->l, l, t->l, key, add), r = t;
45     else
46         split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
47     upd_cnt (t);
48 }
49
50 void reverse (pitem t, int l, int r) {
51     pitem t1, t2, t3;
52     split (t, t1, t2, l);
53     split (t2, t2, t3, r-l+1);
54     t2->rev ^= true;
55     merge (t, t1, t2);
56     merge (t, t, t3);
57 }
58

```

```

59 void output (pitem t) {
60     if (!t) return;
61     push (t);
62     output (t->l);
63     printf ("%d_", t->value);
64     output (t->r);
65 }

```

3.31 Treap With Inversion

```

1 struct Node {
2     Node *l = 0, *r = 0;
3     int val, y, c = 1;
4     bool rev = 0;
5     Node(int val) : val(val), y(rand()) {}
6     void recalc();
7     void push();
8 };
9
10 int cnt(Node* n) { return n ? n->c : 0; }
11 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
12 void Node::push() {
13     if (rev) {
14         rev = 0;
15         swap(l, r);
16         if (l) l->rev ^= 1;
17         if (r) r->rev ^= 1;
18     }
19 }
20
21 template<class F> void each(Node* n, F f) {
22     if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
23 }
24
25 pair<Node*, Node*> split(Node* n, int k) {
26     if (!n) return {};
27     n->push();
28     if (cnt(n->l) >= k) {
29         auto pa = split(n->l, k);
30         n->l = pa.second;
31         n->recalc();
32         return {pa.first, n};
33     } else {

```

```

34     auto pa = split(n->r, k - cnt(n->l) - 1);
35     n->r = pa.first;
36     n->recalc();
37     return {n, pa.second};
38 }
39 }
40
41 Node* merge(Node* l, Node* r) {
42     if (!l) return r;
43     if (!r) return l;
44     l->push();
45     r->push();
46     if (l->y > r->y) {
47         l->r = merge(l->r, r);
48         l->recalc();
49         return l;
50     } else {
51         r->l = merge(l, r->l);
52         r->recalc();
53         return r;
54     }
55 }
56
57 Node* ins(Node* t, Node* n, int pos) {
58     auto pa = split(t, pos);
59     return merge(merge(pa.first, n), pa.second);
60 }
61
62 // Example application: reverse the range [l, r]
63 void reverse(Node& t, int l, int r) {
64     Node *a, *b, *c;
65     tie(a,b) = split(t, l);
66     tie(b,c) = split(b, r - l + 1);
67     b->rev ^= 1;
68     t = merge(merge(a, b), c);
69 }
70
71 void move(Node& t, int l, int r, int k) {
72     Node *a, *b, *c;
73     tie(a,b) = split(t, l);
74     tie(b,c) = split(b, r - l);
75     if (k <= l) t = merge(ins(a, b, k), c);
76     else t = merge(a, ins(c, b, k - r));

```

```

77 }

```

4 Dynamic Programming

4.1 CHT Deque

```

1  /*
2  Convex Hull Trick (CHT) - Min Query with Increasing Slopes
3  -----
4  Indexing: 1-based for 'a', 'dp', 's'
5  Bounds:
6      - i from 1 to m // number of elements in the array
7      - j from 1 to p // number of transitions
8  Time Complexity: O(m * p)
9  Requires:
10     - Lines inserted in increasing slope order for min query
11     - Queries made with increasing x values
12
13     dp[i][j] = min over k < i of { dp[k][j-1] + a[i] * (i - k) + s[i] - s[
14         k+1] }
15     dp[i][j] = min over k < i of { dp[k][j-1] + cost(k, i) }
16
17 We reformulate:
18     y = m * x + c
19     line: m = -k - 1, c = dp[k][j-1] - s[k+1]
20     eval: m * a[i] + c + a[i] * i + s[i]
21 */
22 struct Line {
23     ll a, b; // y = ax + b
24     Line(ll A, ll B) : a(A), b(B) {}
25     ll eval(ll x) const {
26         return a * x + b;
27     }
28     // Returns intersection x-coordinate with another line
29     double intersect(const Line& other) const {
30         return (double)(other.b - b) / (a - other.a);
31     }
32 };
33
34 // this finds the minimum and slope in increasing
35 // Deques for each dp stage
36 deque<Line> cht[p+1];

```

```

37 // Fill dp
38 cht[0].push_back(Line(-1, -s[1])); // base case
39 for (int i = 1; i <= m; i++) {
40     for (int j = p; j >= 1; j--) {
41         if (j > i) continue;
42         // Maintain front of deque to find minimum
43         while (cht[j - 1].size() >= 2 && cht[j - 1][1].eval(a[i]) <= cht[j - 1][0].eval(a[i])) {
44             cht[j - 1].pop_front();
45         }
46         // Evaluate best line
47         dp[i][j] = cht[j - 1].front().eval(a[i]) + a[i] * i + s[i];
48         // Create new line for current i
49         Line curr(-i - 1, dp[i][j] - s[i + 1]);
50         // Maintain convexity: remove worse lines from back
51         while (cht[j].size() >= 2) {
52             Line& l1 = cht[j][cht[j].size() - 2];
53             Line& l2 = cht[j].back();
54             if (curr.intersect(l1) <= l2.intersect(l1)) {
55                 cht[j].pop_back();
56             } else break;
57         }
58         cht[j].push_back(curr);
59     }
60 }

```

4.2 Digit DP

```

1 /*
2  Digit Dynamic Programming (Digit DP)
3  -----
4  Goal: Count numbers in range [0, x] that do not have two adjacent
5        equal digits.
6  State:
7      - pos: current digit position
8      - last: digit placed at previous position (0 to 9)
9      - f: tight flag (0 = must match prefix of x, 1 = already below x)
10     - z: leading zero flag (1 = still in leading zero zone)
11  Notes:
12     - Solve up to x using 'solve(x)'
13     - Can be modified to count palindromes, digits divisible by 3, etc.
14 */

```

```

15 vector<int> num;
16 ll DP[20][20][2][2]; // pos, last digit, f (tight), z (leading zero)
17
18 ll g(int pos, int last, int f, int z) {
19     if (pos == num.size()) return 1; // reached end, valid number
20     if (DP[pos][last][f][z] != -1) return DP[pos][last][f][z];
21     ll res = 0;
22     int limit = f ? 9 : num[pos]; // upper digit bound based on tight flag
23     for (int dgt = 0; dgt <= limit; dgt++) {
24         // Skip if digit equals last (unless it's a leading zero)
25         if (dgt == last && !(dgt == 0 && z == 1)) continue;
26         int nf = f;
27         if (!f && dgt < limit) nf = 1;
28         if (z && dgt == 0) res += g(pos + 1, dgt, nf, 1); // still leading
29                             // zeros
30         else res += g(pos + 1, dgt, nf, 0); // now in significant digits
31     }
32     return DP[pos][last][f][z] = res;
33 }
34
35 ll solve(ll x) {
36     if (x == -1) return 0;
37     num.clear();
38     while (x > 0) {
39         num.push_back(x % 10);
40         x /= 10;
41     }
42     reverse(num.begin(), num.end());
43     memset(DP, -1, sizeof(DP));
44     return g(0, 0, 0, 1);
45 }

```

4.3 Divide and Conquer DP

```

1 /*
2  Divide and Conquer DP Optimization
3  -----
4  Problem:
5      - dp[i][j] = min over k <= j of { dp[i-1][k] + C(k, j) }
6      - C(k, j) must satisfy the quadrangle inequality:
7          - C(a, c) + C(b, d) <= C(a, d) + C(b, c) for a <= b <= c <= d
8      - or monotonicity of opt[i][j] <= opt[i][j+1]
9

```

```

10 Indexing: 0-based
11 Time Complexity: O(m * n * log n)
12 Space Complexity: O(n)
13
14 dp_cur[j]: current dp[i][j] layer
15 dp_before[j]: previous dp[i-1][j] layer
16 */
17
18 int n, m;
19 vector<ll> dp_before, dp_cur;
20 ll C(int i, int j); // Cost function defined by user
21
22 // Recursively compute dp_cur[l..r] with optimal k in [optl, optr]
23 void compute(int l, int r, int optl, int optr) {
24     if (l > r) return;
25     int mid = (l + r) / 2;
26     pair<ll, int> best = {LLONG_MAX, -1};
27     for (int k = optl; k <= min(mid, optr); k++) {
28         ll val = (k > 0 ? dp_before[k - 1] : 0) + C(k, mid);
29         if (val < best.first) best = {val, k};
30     }
31     dp_cur[mid] = best.first;
32     int opt = best.second;
33     compute(l, mid - 1, optl, opt);
34     compute(mid + 1, r, opt, optr);
35 }
36
37 // Entry point: computes dp[m-1][n-1]
38 ll solve() {
39     dp_before.assign(n, 0);
40     dp_cur.assign(n, 0);
41     for (int i = 0; i < n; i++) {
42         dp_before[i] = C(0, i);
43     }
44     for (int i = 1; i < m; i++) {
45         compute(0, n - 1, 0, n - 1);
46         dp_before = dp_cur;
47     }
48     return dp_before[n - 1];
49 }

```

4.4 Edit Distance

```

1 /*
2 Given strings s and t, compute the minimum number of operations
3 (insert, delete, substitute) to convert s into t.
4 Indexing: 0-based (strings), DP is 1-based with offset
5 dp[i][j] = cost to convert s[0..i-1] into t[0..j-1]
6 Time Complexity: O(n * m)
7 Transitions:
8     - insert: dp[i][j-1] + 1
9     - delete: dp[i-1][j] + 1
10    - replace/match: dp[i-1][j-1] + (s[i-1] != t[j-1])
11 */
12 const int MAXN = 5005;
13 int dp[MAXN][MAXN];
14
15 string s, t; cin >> s >> t;
16 int n = s.length(), m = t.length();
17 // Initialize all to a large number
18 for (int i = 0; i <= n; i++) {
19     fill(dp[i], dp[i] + m + 1, 1e9);
20 }
21 dp[0][0] = 0;
22 for (int i = 0; i <= n; i++) {
23     for (int j = 0; j <= m; j++) {
24         if (j) { // insert
25             dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
26         }
27         if (i) { // delete
28             dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
29         }
30         if (i && j) { // replace or match
31             int cost = (s[i - 1] != t[j - 1]) ? 1 : 0;
32             dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + cost);
33         }
34     }
35 }

```

4.5 LCS

```

1 string s, t; cin >> s >> t;
2 int n=s.length(), m=t.length();
3 int dp[n+1][m+1];
4 memset(dp, 0, sizeof(dp));
5 for(int i=1;i<=n;i++){

```

```

6   for(int j=1;j<=m;j++){
7       dp[i][j]=max(dp[i-1][j], dp[i][j-1]);
8       if(s[i-1]==t[j-1]){
9           dp[i][j]=dp[i-1][j-1]+1;
10      }
11  }
12 }
13 int i=n, j=m;
14 string ans="";
15 while(i && j){
16     if(s[i-1]==t[j-1]){
17         ans+=s[i-1];
18         i--; j--;
19     }
20     else if(dp[i][j-1]>=dp[i-1][j]){
21         j--;
22     }
23     else{
24         i--;
25     }
26 }
27 reverse(all(ans));
28 cout << ans << endl;
29
30 // For two permutations one can create new array that will map each
    element from the first permutation to the second.
31 // For each element a[i] in the first permutatio, you find which j is a[
    i] == b[j].
32 // After creating this new array, run LIS (Longest Increasing
    subsequence).

```

4.6 Line Container

```

1  /*
2  Line Container (Dynamic Convex Hull Trick)
3  -----
4  Supports:
5      - Adding lines:  $y = k * x + m$ 
6      - Querying maximum y at given x
7  Indexing: arbitrary, supports any x
8  Time Complexity:
9      - add(): amortized  $O(\log n)$ 
10     - query(x):  $O(\log n)$ 

```

```

11 Space Complexity:  $O(n)$ 
12 For min queries: negate slopes and intercepts on insert and result on
    query
13 Structure:
14     - Stores lines in slope-sorted order (k)
15     - Each line keeps its intersection point 'p' with the next line
16     - Binary search on 'p' to answer queries
17 */
18 struct Line {
19     mutable ll k, m, p;
20     bool operator<(const Line& o) const { return k < o.k; } // Sort by
        slope
21     bool operator<(ll x) const { return p < x; } // Query
        comparator
22 };
23
24 struct LineContainer : multiset<Line, less<>> {
25     // (for doubles, use inf = 1/.0, div(a,b) = a/b)
26     static const ll inf = LLONG_MAX;
27     ll div(ll a, ll b) { // Floored division
28         return a / b - ((a ^ b) < 0 && a % b);
29     }
30     // Update intersection point x->p with y
31     bool isect(iterator x, iterator y) {
32         if (y == end()) return x->p = inf, false;
33         if (x->k == y->k)
34             x->p = (x->m > y->m ? inf : -inf); // higher line wins
35         else
36             x->p = div(y->m - x->m, x->k - y->k);
37         return x->p >= y->p;
38     }
39     // Add new line:  $y = k * x + m$ 
40     void add(ll k, ll m) {
41         auto z = insert({k, m, 0}), y = z++, x = y;
42         // Remove dominated lines after y
43         while (isect(y, z)) z = erase(z);
44         // Remove dominated lines before y
45         if (x != begin() && isect(--x, y))
46             isect(x, y = erase(y));
47         // Further cleanup to preserve order
48         while ((y = x) != begin() && (--x)->p >= y->p)
49             isect(x, erase(y));
50     }

```

```

51 // Query max y at given x
52 ll query(ll x) {
53     assert(!empty());
54     auto l = *lower_bound(x);
55     return l.k * x + l.m;
56 }
57 };
58 // Example usage:
59 LineContainer cht;
60 cht.add(3, 5); // y = 3x + 5
61 cht.add(2, 7); // y = 2x + 7
62 cout << cht.query(4) << '\n'; // max y at x = 4

```

4.7 Longest Increasing Subsequence

```

1  /*
2   Longest Increasing Subsequence + (Recover Sequence) O(n log n)
3   -----
4   If no recovery is needed, use dp[] only.
5   dp.size() gives the length of LIS.
6   For non-decreasing use upper_bound instead of lower_bound.
7  */
8  vector<int> dp; // smallest tail values of LIS length i+1
9  vector<int> dp_index; // index in original array
10 vector<int> parent(n, -1); // parent[i] = index of previous element
    in LIS
11 vector<int> last_pos(n + 1); // last_pos[len] = index in v[] ending LIS
    of length len
12 for (int i = 0; i < n; i++) {
13     auto it = lower_bound(dp.begin(), dp.end(), v[i]);
14     int len = it - dp.begin();
15     if (it == dp.end()) {
16         dp.push_back(v[i]);
17         dp_index.push_back(i); // Ignore if no recovery
18     } else {
19         *it = v[i];
20         dp_index[len] = i; // Ignore if no recovery
21     }
22     if (len > 0) parent[i] = dp_index[len - 1]; // Ignore if no recovery
23 }
24 // Reconstruct LIS
25 vector<int> lis;
26 int pos = dp_index.back();

```

```

27 while (pos != -1) {
28     lis.push_back(v[pos]);
29     pos = parent[pos];
30 }
31 reverse(lis.begin(), lis.end());

```

5 Flow

5.1 Dinic

```

1 // Si en el grafo todos los vertices distintos
2 // de s y t cumplen que solo tienen una arista
3 // de entrada o una de salida la y dicha arista
4 // tiene capacidad 1 entonces la complejidad es
5 // O(E sqrt(v))
6
7 // si todas las aristas tienen capacidad 1
8 // el algoritmo tiene complejidad O(E sqrt(E))
9
10 struct FlowEdge {
11     int v, u;
12     long long cap, flow = 0;
13     FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
14 };
15
16 struct Dinic {
17     const long long flow_inf = 1e18;
18     vector<FlowEdge> edges;
19     vector<vector<int>> adj;
20     int n, m = 0;
21     int s, t;
22     vector<int> level, ptr;
23     queue<int> q;
24
25     Dinic(int n, int s, int t) : n(n), s(s), t(t) {
26         adj.resize(n);
27         level.resize(n);
28         ptr.resize(n);
29     }
30
31     void add_edge(int v, int u, long long cap) {
32         edges.emplace_back(v, u, cap);
33         edges.emplace_back(u, v, 0);

```

```

34     adj[v].push_back(m);
35     adj[u].push_back(m + 1);
36     m += 2;
37 }
38
39 bool bfs() {
40     while (!q.empty()) {
41         int v = q.front();
42         q.pop();
43         for (int id : adj[v]) {
44             if (edges[id].cap - edges[id].flow < 1)
45                 continue;
46             if (level[edges[id].u] != -1)
47                 continue;
48             level[edges[id].u] = level[v] + 1;
49             q.push(edges[id].u);
50         }
51     }
52     return level[t] != -1;
53 }
54
55 long long dfs(int v, long long pushed) {
56     if (pushed == 0)
57         return 0;
58     if (v == t)
59         return pushed;
60     for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
61         int id = adj[v][cid];
62         int u = edges[id].u;
63         if (level[v] + 1 != level[u] || edges[id].cap - edges[id].
64             flow < 1)
65             continue;
66         long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].
67             flow));
68         if (tr == 0)
69             continue;
70         edges[id].flow += tr;
71         edges[id ^ 1].flow -= tr;
72         return tr;
73     }
74     return 0;
75 }

```

```

75     long long flow() {
76         long long f = 0;
77         while (true) {
78             fill(level.begin(), level.end(), -1);
79             level[s] = 0;
80             q.push(s);
81             if (!bfs())
82                 break;
83             fill(ptr.begin(), ptr.end(), 0);
84             while (long long pushed = dfs(s, flow_inf)) {
85                 f += pushed;
86             }
87         }
88         return f;
89     }
90 };

```

5.2 Hopcroft-Karp

```

1 // maximum matching in bipartite graph
2 vector<int> match, dist;
3 vector<vector<int>>> g;
4 int n, m, k;
5 bool bfs()
6 {
7     queue<int> q;
8     // The alternating path starts with unmatched nodes
9     for (int node = 1; node <= n; node++)
10     {
11         if (!match[node])
12         {
13             q.push(node);
14             dist[node] = 0;
15         }
16         else
17         {
18             dist[node] = INF;
19         }
20     }
21
22     dist[0] = INF;
23
24     while (!q.empty())

```

```

25 {
26     int node = q.front();
27     q.pop();
28     if (dist[node] >= dist[0])
29     {
30         continue;
31     }
32     for (int son : g[node])
33     {
34         // If the match of son is matched
35         if (dist[match[son]] == INF)
36         {
37             dist[match[son]] = dist[node] + 1;
38             q.push(match[son]);
39         }
40     }
41 }
42 // Returns true if an alternating path has been found
43 return dist[0] != INF;
44 }
45
46 // Returns true if an augmenting path has been found starting from
47 // vertex node
48 bool dfs(int node)
49 {
50     if (node == 0)
51     {
52         return true;
53     }
54     for (int son : g[node])
55     {
56         if (dist[match[son]] == dist[node] + 1 && dfs(match[son]))
57         {
58             match[node] = son;
59             match[son] = node;
60             return true;
61         }
62     }
63     dist[node] = INF;
64     return false;
65 }
66 int hopcroft_karp()

```

```

67 {
68     int cnt = 0;
69     // While there is an alternating path
70     while (bfs())
71     {
72         for (int node = 1; node <= n; node++)
73         {
74             // If node is unmatched but we can match it using an augmenting
75             // path
76             if (!match[node] && dfs(node))
77             {
78                 cnt++;
79             }
80         }
81     }
82     return cnt;
83 }
84 // usage
85 // n numero de puntos en la izquierda
86 // m numero de puntos en la derecha
87 // las aristas se guardan en g
88 // los puntos estan 1 indexados
89 // el punto 1 de m es el punto n+1 de g
90 // hopcroft_karp() devuelve el tamaño del máximo matching
91 // match contiene el match de cada punto
92 // si match de i es 0, entonces i no está matcheado
93 //
94 // https://judge.yosupo.jp/submission/247277

```

5.3 Hungarian

```

1 #define forn(i,n) for(int i=0;i<int(n);++i)
2 #define forsn(i,s,n) for(int i=s;i<int(n);++i)
3 #define forall(i,c) for(typeof(c.begin()) i=c.begin();i!=c.end();++i)
4 #define DBG(X) cerr << #X << " = " << X << endl;
5 typedef vector<int> vint;
6 typedef vector<vint> vvint;
7
8 void showmt();
9
10 /* begin notebook */
11
12 #define MAXN 256

```

```

13 #define INFTO 0x7f7f7f7f
14 int n;
15 int mt[MAXN][MAXN]; // Matriz de costos (X * Y)
16 int xy[MAXN], yx[MAXN]; // Matching resultante (X->Y, Y->X)
17
18 int lx[MAXN], ly[MAXN], slk[MAXN], slkx[MAXN], prv[MAXN];
19 char S[MAXN], T[MAXN];
20
21 void updtree(int x) {
22     forn(y, n) if (lx[x] + ly[y] - mt[x][y] < slk[y]) {
23         slk[y] = lx[x] + ly[y] - mt[x][y];
24         slkx[y] = x;
25     }
26 }
27 int hungar() {
28     forn(i, n) {
29         ly[i] = 0;
30         lx[i] = *max_element(mt[i], mt[i]+n);
31     }
32     memset(xy, -1, sizeof(xy));
33     memset(yx, -1, sizeof(yx));
34
35     forn(m, n) {
36         memset(S, 0, sizeof(S));
37         memset(T, 0, sizeof(T));
38         memset(prv, -1, sizeof(prv));
39         memset(slk, 0x7f, sizeof(slk));
40         queue<int> q;
41         #define bpone(e, p) { q.push(e); prv[e] = p; S[e] = 1; updtree(e); }
42         forn(i, n) if (xy[i] == -1) { bpone(i, -2); break; }
43
44         int x=0, y=-1;
45         while (y== -1) {
46             while (!q.empty() && y== -1) {
47                 x = q.front(); q.pop();
48                 forn(j, n) if (mt[x][j] == lx[x] + ly[j] && !T[j]) {
49                     if (yx[j] == -1) { y = j; break; }
50                     T[j] = 1;
51                     bpone(yx[j], x);
52                 }
53             }
54             if (y!= -1) break;
55             int dlt = INFTO;

```

```

56         forn(j, n) if (!T[j]) dlt = min(dlt, slk[j]);
57         forn(k, n) {
58             if (S[k]) lx[k] -= dlt;
59             if (T[k]) ly[k] += dlt;
60             if (!T[k]) slk[k] -= dlt;
61         }
62         // q = queue<int>();
63         forn(j, n) if (!T[j] && !slk[j]) {
64             if (yx[j] == -1) {
65                 x = slkx[j]; y = j; break;
66             } else {
67                 T[j] = 1;
68                 if (!S[yx[j]]) bpone(yx[j], slkx[j]);
69             }
70         }
71     }
72     if (y!= -1) {
73         for(int p = x; p != -2; p = prv[p]) {
74             yx[y] = p;
75             int ty = xy[p]; xy[p] = y; y = ty;
76         }
77     } else break;
78 }
79 int res = 0;
80 forn(i, n) res += mt[i][xy[i]];
81 return res;
82 }

```

5.4 Max Flow Min Cost

```

1 // dado un acomodo de flujos con costos
2 // devuelve el costo minimo para un flujo especificado
3
4 struct Edge
5 {
6     int from, to, capacity, cost;
7     Edge(int _from, int _to, int _capacity, int _cost)
8     {
9         from = _from;
10        to = _to;
11        capacity = _capacity;
12        cost = _cost;
13    }

```

```

14 };
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

```

1 long long max_flow(vector<vector<int>> adj, vector<vector<long long>>
   capacity,
2         int source, int sink)
3 {
4     int n = adj.size();
5     vector<int> parent(n, -1);
6     // Find a way from the source to sink on a path with non-negative
       capacities
7     auto reachable = [&]() -> bool
8     {
9         queue<int> q;
10        q.push(source);
11        while (!q.empty())
12        {
13            int node = q.front();
14            q.pop();
15            for (int son : adj[node])
16            {
17                long long w = capacity[node][son];
18                if (w <= 0 || parent[son] != -1)
19                    continue;
20                parent[son] = node;
21                q.push(son);
22            }
23        }
24        return parent[sink] != -1;
25    };
26
27    long long flow = 0;
28    // While there is a way from source to sink with non-negative
       capacities
29    while (reachable())
30    {
31        int node = sink;
32        // The minimum capacity on the path from source to sink
33        long long curr_flow = LLONG_MAX;
34        while (node != source)
35        {
36            curr_flow = min(curr_flow, capacity[parent[node]][node]);
37            node = parent[node];
38        }

```

```

39        node = sink;
40        while (node != source)
41        {
42            // Subtract the capacity from capacity edges
43            capacity[parent[node]][node] -= curr_flow;
44            // Add the current flow to flow backedges
45            capacity[node][parent[node]] += curr_flow;
46            node = parent[node];
47        }
48        flow += curr_flow;
49        fill(parent.begin(), parent.end(), -1);
50    }
51
52    return flow;
53 }
54
55 //vector<vector<long long>> capacity(n, vector<long long>(n));
56 //vector<vector<int>> adj(n);
57 //adj[a].push_back(b);
58 //adj[b].push_back(a);
59 //capacity[a][b] += c;
60
61

```

5.6 Min Cost Max Flow

```

1 /**
2  * If costs can be negative, call setpi before maxflow, but note that
       negative cost cycles are not supported.
3  * To obtain the actual flow, look at positive values only
4  * Time:  $\mathcal{O}(F E \log(V))$  where F is max flow.  $\mathcal{O}(VE)$  for setpi.
5  */
6 #include <bits/stdc++.h>
7 using namespace std;
8
9 #include <ext/pb_ds/priority_queue.hpp>
10 using namespace __gnu_pbds;
11
12 #define rep(i, a, b) for(int i = a; i < (b); ++i)
13 #define all(x) begin(x), end(x)
14 #define sz(x) (int)(x).size()
15 typedef long long ll;
16 typedef pair<int, int> pii;

```

```

17 typedef vector<int> vi;
18
19 #pragma once
20
21 // #include <bits/extc++.h> /// include-line, keep-include
22
23 const ll INF = numeric_limits<ll>::max() / 4;
24
25 struct MCMF {
26     struct edge {
27         int from, to, rev;
28         ll cap, cost, flow;
29     };
30     int N;
31     vector<vector<edge>> ed;
32     vi seen;
33     vector<ll> dist, pi;
34     vector<edge*> par;
35
36     MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}
37
38     void addEdge(int from, int to, ll cap, ll cost) {
39         if (from == to) return;
40         ed[from].push_back(edge{ from,to,sz(ed[to]),cap,cost,0 });
41         ed[to].push_back(edge{ to,from,sz(ed[from])-1,0,-cost,0 });
42     }
43
44     void path(int s) {
45         fill(all(seen), 0);
46         fill(all(dist), INF);
47         dist[s] = 0; ll di;
48
49         __gnu_pbds::priority_queue<pair<ll, int>> q;
50         vector<decltype(q)::point_iterator> its(N);
51         q.push({ 0, s });
52
53         while (!q.empty()) {
54             s = q.top().second; q.pop();
55             seen[s] = 1; di = dist[s] + pi[s];
56             for (edge& e : ed[s]) if (!seen[e.to]) {
57                 ll val = di - pi[e.to] + e.cost;
58                 if (e.cap - e.flow > 0 && val < dist[e.to]) {
59                     dist[e.to] = val;

```

```

60         par[e.to] = &e;
61         if (its[e.to] == q.end())
62             its[e.to] = q.push({ -dist[e.to], e.to });
63         else
64             q.modify(its[e.to], { -dist[e.to], e.to });
65     }
66 }
67
68 rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
69 }
70
71 pair<ll, ll> maxflow(int s, int t) {
72     ll totflow = 0, totcost = 0;
73     while (path(s), seen[t]) {
74         ll fl = INF;
75         for (edge* x = par[t]; x; x = par[x->from])
76             fl = min(fl, x->cap - x->flow);
77
78         totflow += fl;
79         for (edge* x = par[t]; x; x = par[x->from]) {
80             x->flow += fl;
81             ed[x->to][x->rev].flow -= fl;
82         }
83     }
84     rep(i,0,N) for(edge& e : ed[i]) totcost += e.cost * e.flow;
85     return {totflow, totcost/2};
86 }
87
88 // If some costs can be negative, call this before maxflow:
89 void setpi(int s) { // (otherwise, leave this out)
90     fill(all(pi), INF); pi[s] = 0;
91     int it = N, ch = 1; ll v;
92     while (ch-- && it--)
93         rep(i,0,N) if (pi[i] != INF)
94             for (edge& e : ed[i]) if (e.cap)
95                 if ((v = pi[i] + e.cost) < pi[e.to])
96                     pi[e.to] = v, ch = 1;
97     assert(it >= 0); // negative cost cycle
98 }
99 };

```

5.7 Push Relabel

```

1  const int inf = 1000000000;
2
3  int n;
4  vector<vector<int>> capacity, flow;
5  vector<int> height, excess, seen;
6  queue<int> excess_vertices;
7
8  void push(int u, int v) {
9      int d = min(excess[u], capacity[u][v] - flow[u][v]);
10     flow[u][v] += d;
11     flow[v][u] -= d;
12     excess[u] -= d;
13     excess[v] += d;
14     if (d && excess[v] == d)
15         excess_vertices.push(v);
16 }
17
18 void relabel(int u) {
19     int d = inf;
20     for (int i = 0; i < n; i++) {
21         if (capacity[u][i] - flow[u][i] > 0)
22             d = min(d, height[i]);
23     }
24     if (d < inf)
25         height[u] = d + 1;
26 }
27
28 void discharge(int u) {
29     while (excess[u] > 0) {
30         if (seen[u] < n) {
31             int v = seen[u];
32             if (capacity[u][v] - flow[u][v] > 0 && height[u] > height[v])
33                 push(u, v);
34             else
35                 seen[u]++;
36         } else {
37             relabel(u);
38             seen[u] = 0;
39         }
40     }
41 }
42
43 int max_flow(int s, int t) {

```

```

44     height.assign(n, 0);
45     height[s] = n;
46     flow.assign(n, vector<int>(n, 0));
47     excess.assign(n, 0);
48     excess[s] = inf;
49     for (int i = 0; i < n; i++) {
50         if (i != s)
51             push(s, i);
52     }
53     seen.assign(n, 0);
54
55     while (!excess_vertices.empty()) {
56         int u = excess_vertices.front();
57         excess_vertices.pop();
58         if (u != s && u != t)
59             discharge(u);
60     }
61
62     int max_flow = 0;
63     for (int i = 0; i < n; i++)
64         max_flow += flow[i][t];
65     return max_flow;
66 }

```

6 Geometry

6.1 Point Struct

```

1  typedef long long T;
2  struct pt {
3      T x,y;
4      pt operator+(pt p) {return {x+p.x, y+p.y};}
5      pt operator-(pt p) {return {x-p.x, y-p.y};}
6      pt operator*(T d) {return {x*d, y*d};}
7      pt operator/(T d) {return {x/d, y/d};}
8  };
9
10 // cross product
11 // positivo si el segundo esta en sentido antihorario
12 // 0 si el angulo es 180
13 // negativo si el segundo esta en sentido horario
14 T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}
15

```

```

16 // dot product
17 // positivo si el angulo entre los vectores es agudo
18 // 0 si son perpendiculares
19 // negativo si el angulo es obtuso
20 T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}
21
22 T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}
23
24 T dist(pt a,pt b){
25     pt aux=b-a;
26     return sqrtl(aux.x*aux.x+aux.y*aux.y);
27 }

```

6.2 Sort Points

```

1 // This comparator sorts the points clockwise
2 // starting from the first quarter
3
4 bool getQ(Point a){
5     if(a.y!=0){
6         if(a.y>0)return 0;
7         return 1;
8     }
9     if(a.x>0)return 0;
10    return 1;
11 }
12 bool comp(Point a, Point b){
13     if(getQ(a)!=getQ(b))return getQ(a)<getQ(b);
14     return a*b>0;
15 }

```

7 Graphs

7.1 2Sat

```

1 struct TwoSatSolver {
2     int n_vars;                // Number of boolean variables
3     int n_vertices;            // Total vertices in the implication
4     graph (2 per variable)
5     vector<vector<int>> adj;    // Implication graph: adj[i] contains
6     edges from node i
7     vector<vector<int>> adj_t;  // Transposed graph for Kosaraju's
8     algorithm

```

```

6     vector<bool> used;        // Visited marker for DFS
7     vector<int> order;        // Finishing order of vertices (DFS1)
8     vector<int> comp;         // Component ID for each node (DFS2)
9     vector<bool> assignment;  // Final truth assignment for each
10    variable
11
12 // Constructor initializes all data structures
13 TwoSatSolver(int _n_vars)
14 : n_vars(_n_vars),
15   n_vertices(2 * _n_vars),
16   adj(n_vertices),
17   adj_t(n_vertices),
18   used(n_vertices),
19   comp(n_vertices, -1),
20   assignment(n_vars) {
21     order.reserve(n_vertices); // Pre-allocate memory for efficiency
22 }
23
24 // First DFS pass for Kosaraju's algorithm (on original graph)
25 void dfs1(int v) {
26     used[v] = true;
27     for (int u : adj[v]) {
28         if (!used[u])
29             dfs1(u);
30     }
31     order.push_back(v); // Save the vertex post-DFS for reverse ordering
32 }
33
34 // Second DFS pass on the transposed graph to label components
35 void dfs2(int v, int cl) {
36     comp[v] = cl;
37     for (int u : adj_t[v]) {
38         if (comp[u] == -1)
39             dfs2(u, cl);
40     }
41 }
42
43 // Solves the 2-SAT problem using Kosaraju's algorithm
44 bool solve_2SAT() {
45     // 1st pass: fill the order vector
46     order.clear();
47     used.assign(n_vertices, false);
48     for (int i = 0; i < n_vertices; ++i) {

```

```

48     if (!used[i])
49         dfs1(i);
50 }
51
52 // 2nd pass: find SCCs in reverse postorder
53 comp.assign(n_vertices, -1);
54 for (int i = 0, j = 0; i < n_vertices; ++i) {
55     int v = order[n_vertices - i - 1]; // Reverse postorder
56     if (comp[v] == -1)
57         dfs2(v, j++);
58 }
59
60 // Assign values to variables based on component comparison
61 assignment.assign(n_vars, false);
62 for (int i = 0; i < n_vertices; i += 2) {
63     if (comp[i] == comp[i + 1])
64         return false; // Contradiction: variable and its negation are in
                        // the same SCC
65     assignment[i / 2] = comp[i] > comp[i + 1]; // True if var's
                        // component comes after its negation
66 }
67 return true;
68 }
69
70 // Adds a disjunction (a v b) to the implication graph
71 // 'na' and 'nb' indicate negation: if true means !a or !b
72 // Variables are 0-indexed. Bounds are inclusive for each literal (i.e
73 // ., 0 to n_vars - 1)
74 void add_disjunction(int a, bool na, int b, bool nb) {
75     // Each variable 'x' has two nodes:
76     // x => 2*x, !x => 2*x + 1
77     // We encode (a v b) as (!a -> b) and (!b -> a)
78     a = 2 * a ^ na;
79     b = 2 * b ^ nb;
80     int neg_a = a ^ 1;
81     int neg_b = b ^ 1;
82
83     adj[neg_a].push_back(b);
84     adj[neg_b].push_back(a);
85     adj_t[b].push_back(neg_a);
86     adj_t[a].push_back(neg_b);
87 }

```

7.2 Articulation Points

```

1  /*
2  Articulation Points (Cut Vertices) in an Undirected Graph
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, n-1] inclusive
6  Time Complexity: O(V + E)
7  Space Complexity: O(V)
8
9  Use Case:
10     - Identifies vertices whose removal increases the number of
      connected components.
11     - Works on undirected graphs (connected or disconnected).
12 */
13
14 int n; // Number of nodes in the graph
15 vector<vector<int>> adj; // Adjacency list of the undirected graph
16
17 vector<bool> visited; // Marks if a node was visited during DFS
18 vector<int> tin, low; // tin[v]: discovery time; low[v]: lowest
      discovery time reachable from subtree
19 int timer; // Global time counter for DFS
20
21 // DFS traversal to identify articulation points
22 void dfs(int v, int p = -1) {
23     visited[v] = true;
24     tin[v] = low[v] = timer++;
25     int children = 0;
26     for (int to : adj[v]) {
27         if (to == p) continue; // Skip the parent edge
28         if (visited[to]) {
29             // Back edge
30             low[v] = min(low[v], tin[to]);
31         } else {
32             dfs(to, v);
33             low[v] = min(low[v], low[to]);
34             // Articulation point condition for non-root
35             if (low[to] >= tin[v] && p != -1) {
36                 // v is an articulation point
37                 handle_cutpoint(v);
38             }
39             ++children;

```

```

40     }
41 }
42 // Articulation point condition for root
43 if (p == -1 && children > 1) {
44     // v is an articulation point
45     // handle_cutpoint(v);
46 }
47 }
48
49 // Initializes structures and launches DFS
50 void find_cutpoints() {
51     timer = 0;
52     visited.assign(n, false);
53     tin.assign(n, -1);
54     low.assign(n, -1);
55
56     for (int i = 0; i < n; ++i) {
57         if (!visited[i])
58             dfs(i);
59     }
60 }

```

7.3 Bellman-Ford

```

1  /*
2  Bellman-Ford (SPFA variant) for Shortest Paths
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, n-1] inclusive
6  Time Complexity: O(V * E) worst-case (amortized better)
7  Space Complexity: O(V + E)
8
9  Features:
10     - Handles negative edge weights
11     - Detects negative weight cycles (returns false if one exists)
12     - Works on directed or undirected graphs
13
14  Path Reconstruction:
15     - To recover the path from source 's' to any node 'u':
16         vector<int> path;
17         for (int v = u; v != -1; v = parent[v])
18             path.push_back(v);
19         reverse(path.begin(), path.end());

```

```

20  */
21
22  const int INF = 1<<30; // Large value to represent "infinity"
23  vector<vector<pair<int, int>>> adj; // adj[v] = list of (neighbor,
24                                     weight) pairs
25  vector<int> parent; // parent(n, -1) for path reconstruction
26
27  // SPFA implementation to find shortest paths from source s
28  // d[i] will contain shortest distance from s to i
29  // Returns false if a negative cycle is detected
30  // For path reconstruction add vector<int>& parent as parameter
31  bool spfa(int s, vector<int>& d, vector<int>& parent) {
32      int n = adj.size();
33      d.assign(n, INF);
34      vector<int> cnt(n, 0); // Count how many times each node has
35                             // been relaxed
36      vector<bool> inqueue(n, false); // Tracks if a node is currently in
37                                      // queue
38      queue<int> q;
39
40      d[s] = 0;
41      q.push(s);
42      inqueue[s] = true;
43
44      while (!q.empty()) {
45          int v = q.front();
46          q.pop();
47          inqueue[v] = false;
48
49          for (auto edge : adj[v]) {
50              int to = edge.first;
51              int len = edge.second;
52
53              if (d[v] + len < d[to]) {
54                  parent[to] = v; // For path reconstruction
55                  d[to] = d[v] + len;
56                  if (!inqueue[to]) {
57                      q.push(to);
58                      inqueue[to] = true;
59                      cnt[to]++;
60                      if (cnt[to] > n)
61                          return false; // Negative weight cycle detected
62                  }
63              }
64          }
65      }
66      return true;
67  }

```

```

60     }
61   }
62 }
63
64 return true; // No negative cycles; shortest paths computed
65 }

```

7.4 Bipartite Checker

```

1  /*
2   Bipartite Graph Checker (BFS-based)
3   -----
4   Indexing: 0-based
5   Time Complexity: O(V + E)
6   Space Complexity: O(V)
7
8   Handles disconnected graphs
9  */
10
11 int n; // Number of nodes
12 vector<vector<int>> adj; // Adjacency list of the undirected graph
13
14 vector<int> side(n, -1); // -1 = unvisited, 0/1 = sides of bipartition
15 bool is_bipartite = true;
16 queue<int> q;
17
18 for (int st = 0; st < n; ++st) {
19     if (side[st] == -1) {
20         q.push(st);
21         side[st] = 0; // Start with side 0
22         while (!q.empty()) {
23             int v = q.front();
24             q.pop();
25             for (int u : adj[v]) {
26                 if (side[u] == -1) {
27                     // Assign opposite side to neighbor
28                     side[u] = side[v] ^ 1;
29                     q.push(u);
30                 } else {
31                     // Conflict: adjacent nodes on same side
32                     is_bipartite &= side[u] != side[v];
33                 }
34             }
35         }
36     }
37 }

```

```

35     }
36   }
37 }
38
39 cout << (is_bipartite ? "YES" : "NO") << endl;

```

7.5 Bipartite Maximum Matching

```

1  /*
2   Maximum Bipartite Matching (Kuhn's Algorithm)
3   -----
4   Indexing: 0-based
5   Time Complexity: O(N * (E + N)) worst case
6   Space Complexity: O(N + K + E)
7
8   Input:
9   - n: number of nodes on the left side
10  - k: number of nodes on the right side
11  - g: adjacency list where g[v] contains all right nodes adjacent to
12      left node v
13
14   Output:
15   - Prints the pairs (left, right) in the matching
16   - mt[r] = 1 means right node r is matched to left node 1
17  */
18 int n, k; // n: number of left nodes, k: number of right nodes
19 vector<vector<int>> g; // g[l]: list of right-side neighbors of left
20     node l
21 vector<int> mt; // mt[r]: matched left node for right node r (or
22     -1 if unmatched)
23 vector<bool> used; // used[l]: visited status for left node l during
24     DFS
25
26 // Try to find an augmenting path from left node v
27 bool try_kuhn(int v) {
28     if (used[v])
29         return false;
30     used[v] = true;
31     for (int to : g[v]) {
32         if (mt[to] == -1 || try_kuhn(mt[to])) {
33             mt[to] = v;
34             return true;
35         }
36     }
37     return false;
38 }

```

```

32     }
33 }
34 return false;
35 }
36
37 int main() {
38     //... reading the graph ...
39
40     mt.assign(k, -1); // Right-side nodes initially unmatched
41     for (int v = 0; v < n; ++v) {
42         used.assign(n, false); // Reset visited for each left node
43         try_kuhn(v);
44     }
45     // Output matched pairs (left+1, right+1 for 1-based output)
46     for (int i = 0; i < k; ++i) {
47         if (mt[i] != -1)
48             printf("%d %d\n", mt[i] + 1, i + 1);
49     }
50     return 0;
51 }

```

7.6 Block Cut Tree

```

1  /*
2  Block-Cut Tree from Biconnected Components
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, n-1] inclusive
6  Time Complexity: O(V + E)
7  Space Complexity: O(V + E)
8
9  Features:
10 - Identifies articulation points (cut vertices)
11 - Extracts all biconnected components (BCCs)
12 - Constructs the Block-Cut Tree:
13     - Each BCC becomes a node in the tree
14     - Each articulation point becomes its own node
15     - An edge connects a BCC-node to each cutpoint in it
16
17 Output:
18 - 'is_cutpoint': true if node is an articulation point
19 - 'id[v]': node ID of 'v' in the block-cut tree
20 - Returns the block-cut tree as an adjacency list

```

```

21  */
22
23  vector<vector<int>> biconnected_components(vector<vector<int>> &g, //
24      Adjacency list of the undirected graph
25      vector<bool> &is_cutpoint, //
26      Output vector (resized
27      internally)
28      vector<int> &id) { // Output
29      vector (resized
30      internally)
31
32      int n = g.size();
33      vector<vector<int>> comps; // Stores all biconnected components
34      vector<int> stk; // Stack of visited nodes for current
35      component
36      vector<int> num(n), low(n); // DFS discovery time and low-link values
37      is_cutpoint.assign(n, false);
38
39      // DFS to find BCCs and articulation points
40      function<void(int, int, int&)> dfs = [&](int node, int parent, int &
41      timer) {
42          num[node] = low[node] = ++timer;
43          stk.push_back(node);
44          for (int son : g[node]) {
45              if (son == parent) continue;
46              if (num[son]) {
47                  // Back edge
48                  low[node] = min(low[node], num[son]);
49              } else {
50                  dfs(son, node, timer);
51                  low[node] = min(low[node], low[son]);
52                  // Check articulation point condition
53                  if (low[son] >= num[node]) {
54                      is_cutpoint[node] = (num[node] > 1 || num[son] > 2); // For
55                      root and non-root
56                      comps.push_back({node});
57                      while (comps.back().back() != son) {
58                          comps.back().push_back(stk.back());
59                          stk.pop_back();
60                      }
61                  }
62              }
63          }
64      }
65  }

```

```

56 };
57
58 int timer = 0;
59 dfs(0, -1, timer);
60
61 id.resize(n); // Maps each original node to its block-cut tree node ID
62
63 // Build block-cut tree using articulation points and BCCs
64 function<vector<vector<int>>>()> build_tree = [&]() {
65     vector<vector<int>> t(1); // Dummy index 0 (not used)
66     int node_id = 1; // Start assigning block-cut tree IDs from 1
67     // Assign unique tree node IDs to cutpoints
68     for (int node = 0; node < n; ++node) {
69         if (is_cutpoint[node]) {
70             id[node] = node_id++;
71             t.push_back({});
72         }
73     }
74     // Assign each component a new node and connect it to its cutpoints
75     for (auto &comp : comps) {
76         int bcc_node = node_id++;
77         t.push_back({});
78         for (int u : comp) {
79             if (!is_cutpoint[u]) {
80                 id[u] = bcc_node;
81             } else {
82                 t[bcc_node].push_back(id[u]);
83                 t[id[u]].push_back(bcc_node);
84             }
85         }
86     }
87     return t;
88 };
89
90 return build_tree(); // Return the block-cut tree
91 }

```

7.7 Blossom

```

1  /*
2  Edmonds' Blossom Algorithm (Maximum Matching in General Graphs)
3  -----
4  Indexing: 1-based

```

```

5  Node Bounds: [1, n]
6  Time Complexity:  $O(n^3)$  in worst case
7  Space Complexity:  $O(n^2)$ 
8
9  Features:
10     - Handles odd-length cycles (blossoms)
11     - Works on any undirected graph (not just bipartite)
12     - Uses BFS with blossom contraction and path augmentation
13
14  Input:
15     - n: number of vertices
16     - add_edge(u, v): undirected edges between nodes ( $1 \leq u, v \leq n$ )
17
18  Output:
19     - maximum_matching(): returns size of max matching
20     - match[u]: matched vertex for node u (or 0 if unmatched)
21  */
22
23 const int N = 2009;
24 mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
25
26 struct Blossom {
27     int vis[N]; // vis[u]: -1 = unvisited, 0 = in queue, 1 = outer
28     layer
29     int par[N]; // par[u]: parent in alternating tree
30     int orig[N]; // orig[u]: base of blossom u belongs to
31     int match[N]; // match[u]: matched partner of u (0 if unmatched)
32     int aux[N]; // aux[u]: visit marker for LCA
33     int t; // global timestamp for LCA markers
34     int n; // number of nodes
35     bool ad[N]; // ad[u]: whether u is reachable in an alternating
36     path
37     vector<int> g[N]; // g[u]: adjacency list
38     queue<int> Q; // BFS queue
39
40     // Constructor: initializes data for n nodes
41     Blossom() {}
42     Blossom(int _n) {
43         n = _n;
44         t = 0;
45         for (int i = 0; i <= n; ++i) {
46             g[i].clear();
47             match[i] = par[i] = vis[i] = aux[i] = ad[i] = orig[i] = 0;

```

```

46     }
47 }
48
49 void add_edge(int u, int v) {
50     g[u].push_back(v);
51     g[v].push_back(u);
52 }
53
54 // Augment the matching along the alternating path from u to v
55 void augment(int u, int v) {
56     int pv = v, nv;
57     do {
58         pv = par[v];
59         nv = match[pv];
60         match[v] = pv;
61         match[pv] = v;
62         v = nv;
63     } while (u != pv);
64 }
65
66 int lca(int v, int w) {
67     ++t; // Increment timestamp for LCA markers
68     while (true) {
69         if (v) {
70             if (aux[v] == t) return v;
71             aux[v] = t;
72             v = orig[par[match[v]]]; // Move to the parent in the
                                     // alternating tree
73         }
74         swap(v, w);
75     }
76 }
77
78 // Contract a blossom from v and w with common ancestor a
79 void blossom(int v, int w, int a) {
80     while (orig[v] != a) {
81         par[v] = w;
82         w = match[v];
83         ad[v] = true;
84         if (vis[w] == 1) Q.push(w), vis[w] = 0;
85         orig[v] = orig[w] = a;
86         v = par[w];
87     }

```

```

88     }
89
90     // Find augmenting path starting from unmatched node u
91     bool bfs(int u) {
92         fill(vis + 1, vis + n + 1, -1);
93         iota(orig + 1, orig + n + 1, 1);
94         Q = queue<int>();
95         Q.push(u);
96         vis[u] = 0;
97
98         while (!Q.empty()) {
99             int v = Q.front(); Q.pop();
100             ad[v] = true;
101             for (int x : g[v]) {
102                 if (vis[x] == -1) {
103                     par[x] = v;
104                     vis[x] = 1;
105                     if (!match[x]) {
106                         augment(u, x);
107                         return true;
108                     }
109                     Q.push(match[x]);
110                     vis[match[x]] = 0;
111                 } else if (vis[x] == 0 && orig[v] != orig[x]) {
112                     int a = lca(orig[v], orig[x]);
113                     blossom(x, v, a);
114                     blossom(v, x, a);
115                 }
116             }
117         }
118         return false;
119     }
120
121     // Computes maximum matching and returns the size
122     int maximum_matching() {
123         int ans = 0;
124         vector<int> p(n - 1);
125         iota(p.begin(), p.end(), 1);
126         shuffle(p.begin(), p.end(), rnd);
127         for (int i = 1; i <= n; ++i) {
128             shuffle(g[i].begin(), g[i].end(), rnd);
129         }
130

```

```

131 // Greedy matching: try to match unmatched nodes directly
132 for (int u : p) {
133     if (!match[u]) {
134         for (int v : g[u]) {
135             if (!match[v]) {
136                 match[u] = v;
137                 match[v] = u;
138                 ++ans;
139                 break;
140             }
141         }
142     }
143 }
144
145 // Augmenting path phase
146 for (int i = 1; i <= n; ++i) {
147     if (!match[i] && bfs(i)) ++ans;
148 }
149
150 return ans;
151 }
152 } M;
153
154 int main() {
155     ios_base::sync_with_stdio(0);
156     cin.tie(0);
157
158     int t;
159     cin >> t;
160     while (t--) {
161         int n, m;
162         cin >> n >> m;
163         M = Blossom(n);
164         // Read all edges
165         for (int i = 0; i < m; i++) {
166             int u, v;
167             cin >> u >> v;
168             M.add_edge(u, v);
169         }
170         // Compute max matching
171         int matched = M.maximum_matching();
172         if (matched * 2 == n) {
173             // Perfect matching

```

```

174         cout << 0 << '\n';
175     } else {
176         // Find reachable unmatched nodes in alternating trees
177         memset(M.ad, 0, sizeof M.ad);
178         for (int i = 1; i <= n; i++) {
179             if (M.match[i] == 0) M.bfs(i);
180         }
181         int unmatched_reachable = 0;
182         for (int i = 1; i <= n; i++) {
183             unmatched_reachable += M.ad[i];
184         }
185         cout << unmatched_reachable << '\n';
186     }
187 }
188 return 0;
189 }

```

7.8 Bridges

```

1  /*
2  Bridge-Finding in an Undirected Graph
3  -----
4  Indexing: 0-based
5  Node Bounds: [0, n-1] inclusive
6  Time Complexity: O(V + E)
7  Space Complexity: O(V)
8
9  Input:
10     n    - Number of nodes in the graph
11     adj  - Adjacency list of the undirected graph
12
13  Output:
14     - Call 'find_bridges()' to populate bridge information.
15     - Modify the DFS 'Bridge' section to store or print the bridges.
16       A bridge is an edge (v, to) such that removing it increases the
17       number of connected components.
18
19  */
20 int n; // Number of nodes
21 vector<vector<int>> adj; // Adjacency list
22
23 vector<bool> visited; // Marks visited nodes
24 vector<int> tin, low; // tin[v]: discovery time; low[v]: lowest ancestor

```

```

    reachable
24 int timer; // Global DFS timer
25
26 // DFS to detect bridges
27 void dfs(int v, int p = -1) {
28     visited[v] = true;
29     tin[v] = low[v] = timer++;
30     for (int to : adj[v]) {
31         if (to == p) continue; // Skip edge to parent
32         if (visited[to]) {
33             // Back edge
34             low[v] = min(low[v], tin[to]);
35         } else {
36             dfs(to, v);
37             low[v] = min(low[v], low[to]);
38             // Bridge condition: if no back edge connects subtree rooted at 'v'
39             // to ancestors of 'v'
40             if (low[to] > tin[v]) {
41                 // (v, to) is a bridge
42                 // Example: bridges.push_back({v, to});
43             }
44         }
45     }
46
47 // Initialize tracking structures and run DFS
48 void find_bridges() {
49     timer = 0;
50     visited.assign(n, false);
51     tin.assign(n, -1);
52     low.assign(n, -1);
53     for (int i = 0; i < n; ++i) {
54         if (!visited[i])
55             dfs(i);
56     }
57 }

```

7.9 Bridges Online

```

1 /*
2  Online Bridge-Finding (Dynamic Edge Insertion)
3  -----
4  Indexing: 0-based

```

```

5  Node Bounds: [0, n-1] inclusive
6  Time Complexity:
7      - Amortized  $O(\log^2 N)$  per edge addition
8  Space Complexity:  $O(V)$ 
9
10 Features:
11     - Maintains the number of bridges dynamically as edges are added one
12       by one.
13     - Detects if adding an edge merges different 2-edge-connected
14       components.
15     - No deletions supported.
16
17 Input:
18     init(n)      - Initializes the data structure for a graph with n
19                   nodes.
20     add_edge(a, b) - Adds an undirected edge between nodes a and b.
21
22 Output:
23     'bridges' - Global variable representing the current number of
24                 bridges.
25
26 */
27
28 vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
29 int bridges; // Number of bridges in the graph
30 int lca_iteration;
31 vector<int> last_visit;
32
33 // Initializes the data structures
34 void init(int n) {
35     par.resize(n);
36     dsu_2ecc.resize(n);
37     dsu_cc.resize(n);
38     dsu_cc_size.resize(n);
39     last_visit.assign(n, 0);
40     lca_iteration = 0;
41     bridges = 0;
42
43     for (int i = 0; i < n; ++i) {
44         par[i] = -1;
45         dsu_2ecc[i] = i;
46         dsu_cc[i] = i;
47         dsu_cc_size[i] = 1;
48     }

```

```

44 }
45
46 // Finds the representative of the 2-edge-connected component of node v
47 int find_2ecc(int v) {
48     if (v == -1) return -1;
49     return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
50 }
51
52 // Finds the connected component representative of the component
53 // containing v
54 int find_cc(int v) {
55     v = find_2ecc(v);
56     return dsu_cc[v] == v ? v : dsu_cc[v] = find_cc(dsu_cc[v]);
57 }
58
59 // Makes node v the root of its tree, rerouting parent pointers upward
60 void make_root(int v) {
61     int root = v;
62     int child = -1;
63     while (v != -1) {
64         int p = find_2ecc(par[v]);
65         par[v] = child;
66         dsu_cc[v] = root;
67         child = v;
68         v = p;
69     }
70     dsu_cc_size[root] = dsu_cc_size[child];
71 }
72
73 // Merges paths from a and b to their lowest common ancestor in the 2ECC
74 // forest
75 void merge_path(int a, int b) {
76     ++lca_iteration;
77     vector<int> path_a, path_b;
78     int lca = -1;
79
80     while (lca == -1) {
81         if (a != -1) {
82             a = find_2ecc(a);
83             path_a.push_back(a);
84             if (last_visit[a] == lca_iteration) {
85                 lca = a;
86                 break;
87             }
88         }
89         if (b != -1) {
90             b = find_2ecc(b);
91             path_b.push_back(b);
92             if (last_visit[b] == lca_iteration) {
93                 lca = b;
94                 break;
95             }
96         }
97         last_visit[b] = lca_iteration;
98         b = par[b];
99     }
100
101     // Merge all nodes on path_a and path_b into the same 2ECC
102     for (int v : path_a) {
103         dsu_2ecc[v] = lca;
104         if (v == lca) break;
105     }
106     for (int v : path_b) {
107         dsu_2ecc[v] = lca;
108         if (v == lca) break;
109     }
110     --bridges;
111 }
112
113 // Adds an undirected edge between a and b and updates bridge count
114 void add_edge(int a, int b) {
115     a = find_2ecc(a);
116     b = find_2ecc(b);
117     if (a == b) return; // Already in the same 2ECC
118
119     int ca = find_cc(a);
120     int cb = find_cc(b);
121
122     if (ca != cb) {
123         // Bridge found - connects two different components
124         ++bridges;
125         // Union by size
126         if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
127             dsu_cc[cb] = ca;
128         } else {
129             dsu_cc[ca] = cb;
130         }
131     }
132 }

```

```

85 }
86 last_visit[a] = lca_iteration;
87 a = par[a];
88 }
89 if (b != -1) {
90     b = find_2ecc(b);
91     path_b.push_back(b);
92     if (last_visit[b] == lca_iteration) {
93         lca = b;
94         break;
95     }
96     last_visit[b] = lca_iteration;
97     b = par[b];
98 }
99 }
100
101 // Merge all nodes on path_a and path_b into the same 2ECC
102 for (int v : path_a) {
103     dsu_2ecc[v] = lca;
104     if (v == lca) break;
105 }
106 --bridges;
107 }
108 for (int v : path_b) {
109     dsu_2ecc[v] = lca;
110     if (v == lca) break;
111 }
112 --bridges;
113 }
114
115 // Adds an undirected edge between a and b and updates bridge count
116 void add_edge(int a, int b) {
117     a = find_2ecc(a);
118     b = find_2ecc(b);
119     if (a == b) return; // Already in the same 2ECC
120
121     int ca = find_cc(a);
122     int cb = find_cc(b);
123
124     if (ca != cb) {
125         // Bridge found - connects two different components
126         ++bridges;
127         // Union by size
128         if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
129             dsu_cc[cb] = ca;
130         } else {
131             dsu_cc[ca] = cb;
132         }
133     }
134 }

```

```

128     swap(a, b);
129     swap(ca, cb);
130 }
131 make_root(a);
132 par[a] = b;
133 dsu_cc[a] = b;
134 dsu_cc_size[cb] += dsu_cc_size[a];
135 } else {
136     // No new bridge, but must merge paths to unify 2ECCs
137     merge_path(a, b);
138 }
139 }
140
141 // Example usage
142 int main() {
143     init(n);
144     for (auto [u, v] : edges) {
145         add_edge(u, v);
146         cout << "Current_bridge_count:␣" << bridges << '\n';
147     }
148 }

```

7.10 Dijkstra

```

1 vector<vector<pair<int, int>>> adj(n); // Adjacency list (node, weight)
2 vector<ll> dist(n, 1LL << 61); // Distance array initialized to infinity
3
4 priority_queue<pair<ll, int>> q; // Max-heap, so we push negative
   weights to simulate min-heap
5 dist[0] = 0; // Starting node distance
6 q.push({0, 0}); // (distance, vertex)
7
8 while (!q.empty()) {
9     auto [w, v] = q.top(); q.pop();
10    w = -w; // Convert back to positive
11    if (w > dist[v]) continue; // Skip outdated entry
12    for (auto [u, cost] : adj[v]) {
13        if (dist[v] + cost < dist[u]) {
14            dist[u] = dist[v] + cost;
15            q.push({-dist[u], u}); // Push updated distance (negated)
16        }
17    }
18 }

```

7.11 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 node 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 .

```

1  /*
2   Eulerian Path (Hierholzer's Algorithm)
3   -----
4   Time Complexity: O(E)
5   Space Complexity: O(V + E)
6
7   Input:
8   - g: adjacency list of the graph
9       * Directed: vector<vector<pair<int, int>>> g
10         where g[v] = list of {to, edge_index}
11       * Undirected: vector<vector<int>>> g
12         where g[v] = list of neighbors
13   - seen: vector<bool> seen(E) - only needed for directed version
14   - path: vector<int> path - will be filled in reverse order of
        traversal
15         reverse(path.begin(), path.end());
16  */
17
18  // Directed Version //
19  void dfs_directed(int node) {
20      while (!g[node].empty()) {
21          auto [son, idx] = g[node].back();
22          g[node].pop_back();
23          if (seen[idx]) continue; // Skip if edge already visited
24          seen[idx] = true;
25          dfs_directed(son);
26      }
27      path.push_back(node); // Post-order insertion (reverse of actual path)
28  }
29
30  // Undirected Version //
31  void dfs_undirected(int node) {
32      while (!g[node].empty()) {
33          int son = g[node].back();
34          g[node].pop_back();

```

```

35     dfs_undirected(son);
36 }
37 path.push_back(node); // Post-order insertion
38 }

```

7.12 Floyd-Warshall

```

1  /*
2  Floyd-Warshall Algorithm (All-Pairs Shortest Paths)
3  -----
4  Indexing: 0-based
5  Time Complexity: O(V^3)
6  Space Complexity: O(V^2)
7
8  Input:
9  - d: distance matrix of size n x n
10     * d[i][j] should be initialized as:
11       - 0 if i == j
12       - weight of edge (i, j) if exists
13       - INF (e.g. 1e18) otherwise
14 */
15
16 vector<vector<ll>> d(n, vector<ll>(n, 1e18)); // distance matrix
17
18 // This version is by default adapted for UNDIRECTED graphs.
19 for (int k = 0; k < n; k++) {
20     for (int i = 0; i < n; i++) {
21         for (int j = i + 1; j < n; j++) { // For directed graphs, use j = 0;
22             j < n; j++
23             long long new_dist = d[i][k] + d[k][j];
24             if (new_dist < d[i][j]) {
25                 d[i][j] = d[j][i] = new_dist; // update both directions for
26                 undirected graph
27             }
28         }
29     }
30 }

```

7.13 Kruskal

```

1  /*
2  Kruskal's Algorithm (Minimum Spanning Tree - MST)

```

```

3  -----
4  Indexing: 0-based for nodes in edges
5  Time Complexity: O(E log E)
6  Space Complexity: O(N)
7
8  Input:
9  - N: number of nodes
10 - edges: list of weighted edges in form {weight, {u, v}}
11
12 Output:
13 - Returns total weight of the MST if the graph is connected
14 - Returns -1 if MST cannot be formed (i.e., graph is disconnected)
15
16 Note:
17 - Requires a Disjoint Set Union (DSU) / Union-Find data structure
18   with:
19   - unite(a, b): merges components, returns true if successful
20   - size(v): returns size of component containing v
21 */
22 template <class T>
23 T kruskal(int N, vector<pair<T, pair<int, int>>> edges) {
24     sort(edges.begin(), edges.end()); // Sort by weight (non-decreasing)
25     T ans = 0;
26     DSU D(N); // Disjoint Set Union for N nodes
27     for (auto &[w, uv] : edges) {
28         int u = uv.first, v = uv.second;
29         if (D.unite(u, v)) {
30             ans += w; // Add edge to MST if u and v are in different
31                       components
32         }
33     }
34     // Check if MST spans all nodes (i.e., one component of size N)
35     return (D.size(0) == N ? ans : -1);
36 }

```

7.14 Marriage

```

1  /*
2  Male-Optimal Stable Marriage Problem (Gale-Shapley Algorithm)
3  -----
4  Indexing: 0-based
5  Bounds: 0 <= i, j < n

```

```

6   Time Complexity: O(n^2)
7   Space Complexity: O(n^2)
8
9   Input:
10  - n: Number of men/women (equal)
11  - gv[i][j]: j-th most preferred woman for man i
12  - om[i][j]: j-th most preferred man for woman i
13      * Both are permutations of {0, ..., n-1}
14      * om must be inverted to get om[w][m] = woman w's ranking of man
      m
15
16  Output:
17  - pm[i]: Woman matched to man i (i.e. pairings)
18  - pv[i]: Man matched to woman i
19  */
20
21  #define MAXN 1000
22  int gv[MAXN][MAXN], om[MAXN][MAXN]; // Male and female preference lists
23  int pv[MAXN], pm[MAXN];             // pv[woman] = man, pm[man] = woman
24  int pun[MAXN];                      // pun[man] = next woman to propose
      to
25
26  void stableMarriage(int n) {
27      fill_n(pv, n, -1); // All women initially unmatched
28      fill_n(pm, n, -1); // All men initially unmatched
29      fill_n(pun, n, 0); // Each man starts at his top preference
30
31      int unmatched = n; // Number of free men
32      int i = n - 1;    // Current man index (rotates over all men)
33
34      #define engage pm[j] = i; pv[i] = j;
35
36      while (unmatched) {
37          while (pm[i] == -1) {
38              int j = gv[i][pun[i]++]; // Next woman on man i's list
39
40              if (pv[j] == -1) {
41                  // Woman j is free -> engage with man i
42                  unmatched--;
43                  engage;
44              } else if (om[j][i] < om[j][pv[j]]) {
45                  // Woman j prefers i over her current partner
46                  int loser = pv[j];

```

```

47          pm[loser] = -1;
48          engage;
49          i = loser; // Reconsider the rejected man
50      }
51  }
52
53  // Move to next unmatched man
54  i--;
55  if (i < 0) i = n - 1;
56  }
57
58  #undef engage
59  }

```

7.15 SCC

```

1  /*
2  Strongly Connected Components (Kosaraju's Algorithm)
3  -----
4  Indexing: 0-based
5  Time Complexity: O(V + E)
6  Space Complexity: O(V + E)
7
8  Input:
9  - n: number of nodes
10 - m: number of directed edges
11 - adj: original graph
12 - adjr: reversed graph
13
14 Output:
15 - comp[i]: component ID of node i
16 - order[]: nodes in reverse post-order (1st DFS)
17 - nc: is the number of unique comp values
18 */
19
20 vector<vector<int>>> adj, adjr;
21 vector<bool> vis;
22 vector<int> order, comp;
23
24 // First DFS: post-order on original graph
25 void dfs(int v) {
26     vis[v] = true;
27     for (int u : adj[v]) {

```

```

28     if (!vis[u])
29         dfs(u);
30 }
31 order.push_back(v); // Record post-order
32 }
33
34 // Second DFS: assign component IDs on reversed graph
35 void dfsr(int v, int k) {
36     vis[v] = true;
37     comp[v] = k;
38     for (int u : adjr[v]) {
39         if (!vis[u])
40             dfsr(u, k);
41     }
42 }
43
44 void solve() {
45     int n, m;
46     cin >> n >> m;
47     adj.assign(n, vector<int>());
48     adjr.assign(n, vector<int>());
49     comp.resize(n);
50     // Read edges and build both original and reversed graphs
51     for (int i = 0; i < m; i++) {
52         int a, b;
53         cin >> a >> b;
54         a--; b--;
55         adj[a].push_back(b);
56         adjr[b].push_back(a);
57     }
58     // First pass: DFS on original graph to get order
59     vis.assign(n, false);
60     order.clear();
61     for (int i = 0; i < n; i++) {
62         if (!vis[i]) dfs(i);
63     }
64     // Second pass: DFS on reversed graph using reverse post-order
65     vis.assign(n, false);
66     int nc = 0;
67     for (int i = n - 1; i >= 0; i--) {
68         int v = order[i];
69         if (!vis[v]) {
70             dfsr(v, nc++);

```

```

71     }
72 }
73 // comp[i] now holds the component ID for node i (0-based)
74 // nc = number of strongly connected components
75 }

```

8 Linear Algebra

8.1 Simplex

```

1  /*
2  Parametric Self-Dual Simplex method
3  Solve a canonical LP:
4      min or max. c x
5      s.t. A x <= b
6          x >= 0
7  */
8  #include <bits/stdc++.h>
9  using namespace std;
10 const double eps = 1e-9, oo = numeric_limits<double>::infinity();
11
12 typedef vector<double> vec;
13 typedef vector<vec> mat;
14
15 pair<vec, double> simplexMethodPD(const mat &A, const vec &b, const vec
    &c, bool mini = true){
16     int n = c.size(), m = b.size();
17     mat T(m + 1, vec(n + m + 1));
18     vector<int> base(n + m), row(m);
19
20     for(int j = 0; j < m; ++j){
21         for(int i = 0; i < n; ++i)
22             T[j][i] = A[j][i];
23         row[j] = n + j;
24         T[j][n + j] = 1;
25         base[n + j] = 1;
26         T[j][n + m] = b[j];
27     }
28
29     for(int i = 0; i < n; ++i)
30         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_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);
117    auto ans = simplexMethodPD(A, b, c, mini);
118    cout << (mini ? "Min" : "Max") << "_z_" << ans.second << ",_cuando:_";

```

```

119     \n";
120     for(int i = 0; i < ans.first.size(); ++i){
121         cout << "x_" << (i + 1) << " = " << ans.first[i] << "\n";
122     }
123     return 0;

```

9 Math

9.1 BinPow

```

1 ll bnpow(ll a, ll b){
2     ll r=1;
3     while(b){
4         if(b%2)
5             r=(r*a)%MOD;
6         a=(a*a)%MOD;
7         b/=2;
8     }
9     return r;
10 }
11
12 ll divide(ll a, ll b){
13     return ((a%MOD)*bnpow(b, MOD-2))%MOD;
14 }
15 void inverses(long long p) {
16     inv[MAXN] = exp(fac[MAXN], p - 2, p);
17     for (int i = MAXN; i >= 1; i--) { inv[i - 1] = inv[i] * i % p; }
18 }

```

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 $y = y_0 - k * \frac{a}{\gcd(a,b)}$.

```

1 int gcd(int a, int b, int& x, int& y) {
2     if (b == 0) {
3         x = 1;
4         y = 0;
5         return a;
6     }
7     int x1, y1;
8     int d = gcd(b, a % b, x1, y1);
9     x = y1;

```

```

10     y = x1 - y1 * (a / b);
11     return d;
12 }
13
14 bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
15     g = gcd(abs(a), abs(b), x0, y0);
16     if (c % g) {
17         return false;
18     }
19
20     x0 *= c / g;
21     y0 *= c / g;
22     if (a < 0) x0 = -x0;
23     if (b < 0) y0 = -y0;
24     return true;
25 }
26
27
28
29 //n variables
30 vector<ll> find_any_solution(vector<ll> a, ll c) {
31     int n = a.size();
32     vector<ll> x;
33     bool all_zero = true;
34     for (int i = 0; i < n; i++) {
35         all_zero &= a[i] == 0;
36     }
37     if (all_zero) {
38         if (c) return {};
39         x.assign(n, 0);
40         return x;
41     }
42     ll g = 0;
43     for (int i = 0; i < n; i++) {
44         g = __gcd(g, a[i]);
45     }
46     if (c % g != 0) return {};
47     if (n == 1) {
48         return {c / a[0]};
49     }
50     vector<ll> suf_gcd(n);
51     suf_gcd[n - 1] = a[n - 1];
52     for (int i = n - 2; i >= 0; i--) {

```

```

53     suf_gcd[i] = __gcd(suf_gcd[i + 1], a[i]);
54 }
55 ll cur = c;
56 for (int i = 0; i + 1 < n; i++) {
57     ll x0, y0, g;
58     // solve for a[i] * x + suf_gcd[i + 1] * (y / suf_gcd[i + 1]) = cur
59     bool ok = find_any_solution(a[i], suf_gcd[i + 1], cur, x0, y0, g);
60     assert(ok);
61     {
62         // trying to minimize x0 in case x0 becomes big
63         // it is needed for this problem, not needed in general
64         ll shift = abs(suf_gcd[i + 1] / g);
65         x0 = (x0 % shift + shift) % shift;
66     }
67     x.push_back(x0);
68
69     // now solve for the next suffix
70     cur -= a[i] * x0;
71 }
72 x.push_back(a[n - 1] == 0 ? 0 : cur / a[n - 1]);
73 return x;
74 }

```

9.3 Discrete Logarithm

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

```

1 // Returns minimum x for which a ^ x % m = b % m, a and m are coprime.
2 int solve(int a, int b, int m) {
3     a %= m, b %= m;
4     int n = sqrt(m) + 1;
5
6     int an = 1;
7     for (int i = 0; i < n; ++i)
8         an = (an * 11l * a) % m;
9
10    unordered_map<int, int> vals;
11    for (int q = 0, cur = b; q <= n; ++q) {
12        vals[cur] = q;
13        cur = (cur * 11l * a) % m;
14    }
15
16    for (int p = 1, cur = 1; p <= n; ++p) {

```

```

17        cur = (cur * 11l * an) % m;
18        if (vals.count(cur)) {
19            int ans = n * p - vals[cur];
20            return ans;
21        }
22    }
23    return -1;
24 }
25
26 // Returns minimum x for which a ^ x % m = b % m.
27 int solve(int a, int b, int m) {
28     a %= m, b %= m;
29     int k = 1, add = 0, g;
30     while ((g = gcd(a, m)) > 1) {
31         if (b == k)
32             return add;
33         if (b % g)
34             return -1;
35         b /= g, m /= g, ++add;
36         k = (k * 11l * a / g) % m;
37     }
38
39     int n = sqrt(m) + 1;
40     int an = 1;
41     for (int i = 0; i < n; ++i)
42         an = (an * 11l * a) % m;
43
44     unordered_map<int, int> vals;
45     for (int q = 0, cur = b; q <= n; ++q) {
46         vals[cur] = q;
47         cur = (cur * 11l * a) % m;
48     }
49
50     for (int p = 1, cur = k; p <= n; ++p) {
51         cur = (cur * 11l * an) % m;
52         if (vals.count(cur)) {
53             int ans = n * p - vals[cur] + add;
54             return ans;
55         }
56     }
57     return -1;
58 }

```

9.4 Divisors

```

1 long long numberOfDivisors(long long num)
2 {
3     long long total = 1;
4     for (int i = 2; (long long)i * i <= num; i++)
5     {
6         if (num % i == 0)
7         {
8             int e = 0;
9             do
10            {
11                e++;
12                num /= i;
13            } while (num % i == 0);
14            total *= e + 1;
15        }
16    }
17    if (num > 1)
18    {
19        total *= 2;
20    }
21    return total;
22 }
23
24 long long SumOfDivisors(long long num)
25 {
26     long long total = 1;
27
28     for (int i = 2; (long long)i * i <= num; i++)
29     {
30         if (num % i == 0)
31         {
32             int e = 0;
33             do
34             {
35                 e++;
36                 num /= i;
37             } while (num % i == 0);
38
39             long long sum = 0, pow = 1;
40             do
41             {

```

```

42                 sum += pow;
43                 pow *= i;
44             } while (e-- > 0);
45             total *= sum;
46         }
47     }
48     if (num > 1)
49     {
50         total *= (1 + num);
51     }
52     return total;
53 }

```

9.5 Euler Totient (Phi)

```

1 //counts coprimes to each number from 1 to n
2 vector<int> phi1(int n) {
3     vector<int> phi(n + 1);
4     for (int i = 0; i <= n; i++)
5         phi[i] = i;
6
7     for (int i = 2; i <= n; i++) {
8         if (phi[i] == i) {
9             for (int j = i; j <= n; j += i)
10                 phi[j] -= phi[j] / i;
11         }
12     }
13     return phi1;
14 }

```

9.6 Fibonacci

```

1 void fib(ll n, ll&x, ll&y){
2     if(n==0){
3         x = 0;
4         y = 1;
5         return ;
6     }
7
8     if(n&1){
9         fib(n-1, y, x);
10        y=(y+x)%MOD;
11    }else{
12        ll a, b;

```

```

13     fib(n>>1, a, b);
14     y = (a*a+b*b)%MOD;
15     x = (a*b + a*(b-a+MOD))%MOD;
16 }
17 }
18
19 // Usage
20 // ll x, y;
21 // fib(10, x, y);
22 // cout << x << " " << y << endl;
23 // This will output 55 89

```

9.7 Matrix Exponentiation

```

1 struct Mat {
2     int n, m;
3     vector<vector<int>> a;
4     Mat() { }
5     Mat(int _n, int _m) {n = _n; m = _m; a.assign(n, vector<int>(m, 0)); }
6     Mat(vector< vector<int> > v) { n = v.size(); m = n ? v[0].size() : 0;
7         a = v; }
8     inline void make_unit() {
9         assert(n == m);
10        for (int i = 0; i < n; i++) {
11            for (int j = 0; j < n; j++) a[i][j] = i == j;
12        }
13    inline Mat operator + (const Mat &b) {
14        assert(n == b.n && m == b.m);
15        Mat ans = Mat(n, m);
16        for(int i = 0; i < n; i++) {
17            for(int j = 0; j < m; j++) {
18                ans.a[i][j] = (a[i][j] + b.a[i][j]) % mod;
19            }
20        }
21        return ans;
22    }
23    inline Mat operator - (const Mat &b) {
24        assert(n == b.n && m == b.m);
25        Mat ans = Mat(n, m);
26        for(int i = 0; i < n; i++) {
27            for(int j = 0; j < m; j++) {
28                ans.a[i][j] = (a[i][j] - b.a[i][j] + mod) % mod;

```

```

29    }
30    }
31    return ans;
32 }
33 inline Mat operator * (const Mat &b) {
34     assert(m == b.n);
35     Mat ans = Mat(n, b.m);
36     for(int i = 0; i < n; i++) {
37         for(int j = 0; j < b.m; j++) {
38             for(int k = 0; k < m; k++) {
39                 ans.a[i][j] = (ans.a[i][j] + 1LL * a[i][k] * b.a[k][j] % mod)
40                     % mod;
41             }
42         }
43     }
44     return ans;
45 }
46 inline Mat pow(long long k) {
47     assert(n == m);
48     Mat ans(n, n), t = a; ans.make_unit();
49     while (k) {
50         if (k & 1) ans = ans * t;
51         t = t * t;
52         k >>= 1;
53     }
54     return ans;
55 }
56 inline Mat& operator += (const Mat& b) { return *this = (*this) + b; }
57 inline Mat& operator -= (const Mat& b) { return *this = (*this) - b; }
58 inline Mat& operator *= (const Mat& b) { return *this = (*this) * b; }
59 inline bool operator == (const Mat& b) { return a == b.a; }
60 inline bool operator != (const Mat& b) { return a != b.a; }
61 };
62
63 // Usage
64 // Mat a(n, n);
65 // Mat b(n, n);
66 // Mat c = a * b;
67 // Mat d = a + b;
68 // Mat e = a - b;
69 // Mat f = a.pow(k);
70 // a.a[i][j] = x;

```

9.8 Miller Rabin Deterministic

```

1 using u64 = uint64_t;
2 using u128 = __uint128_t;
3
4 u64 binpower(u64 base, u64 e, u64 mod) {
5     u64 result = 1;
6     base %= mod;
7     while (e) {
8         if (e & 1)
9             result = (u128)result * base % mod;
10        base = (u128)base * base % mod;
11        e >>= 1;
12    }
13    return result;
14 }
15
16 bool check_composite(u64 n, u64 a, u64 d, int s) {
17     u64 x = binpower(a, d, n);
18     if (x == 1 || x == n - 1)
19         return false;
20     for (int r = 1; r < s; r++) {
21         x = (u128)x * x % n;
22         if (x == n - 1)
23             return false;
24     }
25     return true;
26 };
27
28
29 bool MillerRabin(ll n) {
30     if (n < 2)
31         return false;
32
33     int r = 0;
34     ll d = n - 1;
35     while ((d & 1) == 0) {
36         d >>= 1;
37         r++;
38     }
39
40     for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
41         if (n == a)

```

```

42         return true;
43         if (check_composite(n, a, d, r))
44             return false;
45     }
46     return true;
47 }

```

9.9 Mobius

```

1 int mob[N];
2 void mobius() {
3     mob[1] = 1;
4     for (int i = 2; i < N; i++){
5         mob[i]--;
6         for (int j = i + i; j < N; j += i) {
7             mob[j] -= mob[i];
8         }
9     }
10 }

```

9.10 Prefix Sum Phi

```

1 vector<ll> sieve(kMaxV + 1,0);
2 vector<ll> phi(kMaxV + 1,0);
3
4 void primes()
5 {
6     phi[1]=1;
7     vector<ll> pr;
8     for(int i=2;i<kMaxV;i++){
9         if(sieve[i]==0){
10            sieve[i]=i;
11            pr.pb(i);
12            phi[i]=i-1;
13        }
14        for(auto p:pr){
15            if(p>sieve[i] || i*p>=kMaxV)break;
16            sieve[i*p]=p;
17            phi[i*p]=(p==sieve[i]?p:p-1)*phi[i];
18        }
19    }
20    for(int i=1;i<kMaxV;i++){
21        phi[i]+=phi[i-1];
22        phi[i]%=MOD;

```

```

23 }
24 }
25
26 map<ll,ll> m;
27 ll PHI(ll a){
28     if(a<kMaxV)return phi[a];
29     if(m.count(a))return m[a];
30     // if(a<3)return 1;
31     m[a]=(((a%MOD)*((a+1)%MOD))%MOD)*inverse(2));
32     m[a]%=MOD;
33     long long i=2;
34     while(i<=a){
35         long long j=a/i;
36         j=a/j;
37         m[a]+=MOD;
38         m[a]-=((j-i+1)*PHI(a/i))%MOD;
39         m[a]%=MOD;
40         i=j+1;
41     }
42     m[a]%=MOD;
43     return m[a];
44 }

```

9.11 Sieve

```

1  const int kMaxV = 1e6;
2
3  int sieve[kMaxV + 1];
4
5  //stores some prime (not necessarily the minimum one)
6  void primes()
7  {
8      for (int i = 4; i <= kMaxV; i += 2)
9          sieve[i] = 2;
10     for (int i = 3; i <= kMaxV / i; i += 2)
11     {
12         if (sieve[i])
13             continue;
14         for (int j = i * i; j <= kMaxV; j += i)
15             sieve[j] = i;
16     }
17 }
18

```

```

19 vector<int> PrimeFactors(int x)
20 {
21     if (x == 1)
22         return {};
23
24     unordered_set<int> primes;
25     while (sieve[x])
26     {
27         primes.insert(sieve[x]);
28         x /= sieve[x];
29     }
30     primes.insert(x);
31     return {primes.begin(), primes.end()};
32 }

```

9.12 Identities

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

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

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

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

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

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

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

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

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

$$f(n) = \sum_{d|n} g(d) \mu\left(\frac{n}{d}\right).$$

$$\text{(Dirichlet Convolution) Let } f, g \text{ be arithmetic functions, then}$$

$$(f * g)(n) = \sum_{d|n} f(d) g\left(\frac{n}{d}\right). \text{ If } f, g \text{ are multiplicative, then so is } f * g.$$

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

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

$$n = \sum_{i=0}^k n_i p^i.$$

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

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

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

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

- 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 $\binom{n}{\text{remaining}_{\text{closed}}} - \binom{n}{\text{remaining}_{\text{closed}}+1}$.
- binary trees 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

$$\begin{array}{l} \text{Infinite} \\ a + ar + ar^2 + ar^3 + \dots + \sum_{k=0}^{\infty} ar^k \\ \text{Sum} = \frac{a}{1-r} \\ \text{Finite} \\ a + ar + ar^2 + ar^3 + \dots + \sum_{k=0}^n ar^k \\ \text{Sum} = \frac{a(1-r^{n+1})}{1-r} \end{array}$$

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

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

9.22 Pythagorean Triplets

The Pythagorean triples are uniquely generated by

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

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.

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

10 Game Theory

10.1 Sprague-Grundy theorem

<https://codeforces.com/blog/entry/66040> Dado un juego con pilas p_1, p_2, \dots, p_n

sea $g(p)$ el nimber de la pila p , entonces el nimber del juego es

$g(p_1) \oplus g(p_2) \oplus \dots \oplus g(p_n)$. Para calcular el nimber de una pila, se puede usar la fórmula $g(r) = mex(\{g(r_1), g(r_2), \dots, g(r_k)\})$ donde r_1, r_2, \dots, 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 More Topics

11.1 2D Prefix Sum

```

1 int b[MAXN][MAXN];
2 int a[MAXN][MAXN];
3
4 for (int i = 1; i <= N; i++) {
5     for (int j = 1; j <= N; j++) {
```

```

6         b[i][j] = a[i][j] + b[i-1][j] +
7             b[i][j-1] - b[i-1][j-1];
8     }
9 }
10
11 for (int q = 0; q < Q; q++) {
12     int from_row, to_row, from_col, to_col;
13     cin >> from_row >> from_col >> to_row >> to_col;
14     cout << b[to_row][to_col] - b[from_row-1][to_col] -
15         b[to_row][from_col-1] +
16         b[from_row-1][from_col-1]
17     << '\n';
18 }
```

11.2 Custom Comparators

```

1 bool cmp(const Edge &x, const Edge &y) { return x.w < y.w; }
2
3 sort(a.begin(), a.end(), cmp);
4
5 set<int, greater<int>> a;
6 map<int, string, greater<int>> b;
7 priority_queue<int, vector<int>, greater<int>> c;
```

11.3 Day of the Week

```

1 int dayOfWeek(int d, int m, lli y){
2     if(m == 1 || m == 2){
3         m += 12;
4         --y;
5     }
6     int k = y % 100;
7     lli j = y / 100;
8     return (d + 13*(m+1)/5 + k + k/4 + j/4 + 5*j) % 7;
9 }
```

11.4 GCD Convolution

```

1 vector<int> PrimeEnumerate(int n) {
2     vector<int> P; vector<bool> B(n+1, 1);
3     for (int i = 2; i <= n; i++) {
4         if (B[i]) P.push_back(i);
5         for (int j : P) { if (i * j > n) break; B[i * j] = 0; if (i % j ==
6             0) break; }
```

```

6   }
7   return P;
8 }
9
10
11 template<typename T>
12 void MultipleZetaTransform(vector<T>& v) {
13     const int n = (int)v.size() - 1;
14     for (int p : PrimeEnumerate(n)) {
15         for (int i = n / p; i; i--)
16             v[i] += v[i * p];
17     }
18 }
19
20 template<typename T>
21 void MultipleMobiusTransform(vector<T>& v) {
22     const int n = (int)v.size() - 1;
23     for (int p : PrimeEnumerate(n)) {
24         for (int i = 1; i * p <= n; i++)
25             v[i] -= v[i * p];
26     }
27 }
28
29 template<typename T>
30 vector<T> GCDCconvolution(vector<T> A, vector<T> B) {
31     MultipleZetaTransform(A);
32     MultipleZetaTransform(B);
33     for (int i = 0; i < A.size(); i++) A[i] *= B[i];
34     MultipleMobiusTransform(A);
35     return A;
36 }

```

11.5 int128

```

1 //cout for __int128
2 ostream &operator<<(ostream &os, const __int128 & value){
3     char buffer[64];
4     char *pos = end(buffer) - 1;
5     *pos = '\0';
6     __int128 tmp = value < 0 ? -value : value;
7     do{
8         --pos;
9         *pos = tmp % 10 + '0';

```

```

10     tmp /= 10;
11 }while(tmp != 0);
12 if(value < 0){
13     --pos;
14     *pos = '-';
15 }
16 return os << pos;
17 }
18
19 //cin for __int128
20 istream &operator>>(istream &is, __int128 & value){
21     char buffer[64];
22     is >> buffer;
23     char *pos = begin(buffer);
24     int sgn = 1;
25     value = 0;
26     if(*pos == '-'){
27         sgn = -1;
28         ++pos;
29     }else if(*pos == '+'){
30         ++pos;
31     }
32     while(*pos != '\0'){
33         value = (value << 3) + (value << 1) + (*pos - '0');
34         ++pos;
35     }
36     value *= sgn;
37     return is;
38 }
39
40
41 ll mult(__int128 a, __int128 b){ return ((a*1LL*b)%MOD + MOD)%MOD; }

```

11.6 Iterating Over All Subsets

```

1 for (int mk = 0; mk < (1 << k); mk++) {
2     Ap[mk] = 0;
3     for (int s = mk;; s = (s - 1) & mk) {
4         Ap[mk] += A[s];
5         if (!s)
6             break;
7     }
8 }

```

11.7 LCM Convolution

```

1  /* Linear Sieve, O(n) */
2  vector<int> PrimeEnumerate(int n) {
3      vector<int> P; vector<bool> B(n + 1, 1);
4      for (int i = 2; i <= n; i++) {
5          if (B[i]) P.push_back(i);
6          for (int j : P) { if (i * j > n) break; B[i * j] = 0; if (i % j ==
7              0) break; }
8      }
9      return P;
10 }

11 template<typename T>
12 void DivisorZetaTransform(vector<T>& v) {
13     const int n = (int)v.size() - 1;
14     for (int p : PrimeEnumerate(n)) {
15         for (int i = 1; i * p <= n; i++)
16             v[i * p] += v[i];
17     }
18 }

19
20 template<typename T>
21 void DivisorMobiusTransform(vector<T>& v) {
22     const int n = (int)v.size() - 1;
23     for (int p : PrimeEnumerate(n)) {
24         for (int i = n / p; i; i--)
25             v[i * p] -= v[i];
26     }
27 }

28
29
30 template<typename T>
31 vector<T> LCMConvolution(vector<T> A, vector<T> B) {
32     DivisorZetaTransform(A);
33     DivisorZetaTransform(B);
34     for (int i = 0; i < A.size(); i++) A[i] *= B[i];
35     DivisorMobiusTransform(A);
36     return A;
37 }

```

11.8 Manhattan MST

```

1 struct point {
2     long long x, y;
3 };
4
5 vector<tuple<long long, int, int>> manhattan_mst_edges(vector<point> ps)
6 {
7     vector<int> ids(ps.size());
8     iota(ids.begin(), ids.end(), 0);
9     vector<tuple<long long, int, int>> edges;
10    for (int rot = 0; rot < 4; rot++) { // for every rotation
11        sort(ids.begin(), ids.end(), [&](int i, int j){
12            return (ps[i].x + ps[i].y) < (ps[j].x + ps[j].y);
13        });
14        map<int, int, greater<int>> active; // (xs, id)
15        for (auto i : ids) {
16            for (auto it = active.lower_bound(ps[i].x); it != active.end();
17                active.erase(it++)) {
18                int j = it->second;
19                if (ps[i].x - ps[i].y > ps[j].x - ps[j].y) break;
20                assert(ps[i].x >= ps[j].x && ps[i].y >= ps[j].y);
21                edges.push_back({(ps[i].x - ps[j].x) + (ps[i].y - ps[j].y), i, j
22                    });
23            }
24            active[ps[i].x] = i;
25        }
26        for (auto &p : ps) { // rotate
27            if (rot & 1) p.x *= -1;
28            else swap(p.x, p.y);
29        }
30    }
31    return edges;
32 }

```

11.9 Mo

```

1  /*
2  Mo's Algorithm (Sqrt Decomposition for Offline Queries)
3  -----
4  Problem: Answer q range queries [L, R] over array of length n
5           using add/remove operations efficiently
6  Indexing: 0-based
7  Bounds:
8  - arr[0..n-1]

```

```

9      - queries input as 1-based and converted to 0-based
10      Time Complexity:  $O((n + q) * \sqrt{n})$  per query batch
11      Space Complexity:  $O(n + q)$ 
12      Usage:
13          - Fill in logic for add/remove operations (update cur)
14          - Fill answers[] indexed by original query order
15  */
16
17  const int MAXN = 200500;
18  ll n, q;
19  ll arr[MAXN];      // input array
20  ll cnt[1000005];   // frequency count
21  ll answers[MAXN];  // output array
22  ll cur = 0;        // current query result
23  ll BLOCK_SIZE;
24
25  pair< pair<ll, ll>, ll> queries[MAXN]; // {{L, R}, query_index}
26
27  // Sort by block and then by R
28  inline bool cmp(const pair< pair<ll, ll>, ll> &x, const pair< pair<ll,
29      ll>, ll> &y) {
30      ll block_x = x.first.first / BLOCK_SIZE;
31      ll block_y = y.first.first / BLOCK_SIZE;
32      if (block_x != block_y) return block_x < block_y;
33      return x.first.second < y.first.second;
34  }
35
36  int main() {
37      cin >> n >> q;
38      BLOCK_SIZE = sqrt(n);
39      for (int i = 0; i < n; i++) cin >> arr[i];
40      for (int i = 0; i < q; i++) {
41          int l, r;
42          cin >> l >> r;
43          --l; --r; // convert to 0-based
44          queries[i] = {{l, r}, i};
45      }
46      sort(queries, queries + q, cmp);
47      ll l = 0, r = -1;
48      for (int i = 0; i < q; i++) {
49          int left = queries[i].first.first;
50          int right = queries[i].first.second;
51          // Expand to right

```

```

51      while (r < right) {
52          r++;
53          // Operations to add arr[r], implement exactly here
54      }
55      // Shrink from right
56      while (r > right) {
57          // Operations to remove arr[r], implement exactly here
58          r--;
59      }
60      // Expand to left
61      while (l < left) {
62          // Operations to remove arr[l], implement exactly here
63          l++;
64      }
65      // Shrink from left
66      while (l > left) {
67          l--;
68          // Operations to add arr[l], implement exactly here
69      }
70      answers[queries[i].second] = cur; // Current answer
71  }
72  for (int i = 0; i < q; i++) cout << answers[i] << '\n';
73  }

```

11.10 MOD INT

```

1  /**
2   * Description: Mod integer class for doing modular arithmetic.
3   * Source: https://github.com/jakobkogler/Algorithm-DataStructures/blob/master/Math/Modular.h
4   * Verification: https://open.kattis.com/problems/modulararithmetic
5   * Time: fast
6   */
7
8  template<int MOD>
9  struct ModInt {
10      long long v;
11      ModInt(long long _v = 0) {v = (-MOD < _v && _v < MOD) ? _v : _v %
12          MOD; if (v < 0) v += MOD;}
13      ModInt& operator += (const ModInt &other) {v += other.v; if (v >=
14          MOD) v -= MOD; return *this;}
15      ModInt& operator -= (const ModInt &other) {v -= other.v; if (v < 0)
16          v += MOD; return *this;}

```

```

14 ModInt& operator *= (const ModInt &other) {v = v * other.v % MOD;
    return *this;}
15 ModInt& operator /= (const ModInt &other) {return *this *= inverse(
    other);}
16 bool operator == (const ModInt &other) const {return v == other.v;}
17 bool operator != (const ModInt &other) const {return v != other.v;}
18 friend ModInt operator + (ModInt a, const ModInt &b) {return a += b
    ;}
19 friend ModInt operator - (ModInt a, const ModInt &b) {return a -= b
    ;}
20 friend ModInt operator * (ModInt a, const ModInt &b) {return a *= b
    ;}
21 friend ModInt operator / (ModInt a, const ModInt &b) {return a /= b
    ;}
22 friend ModInt operator - (const ModInt &a) {return 0 - a;}
23 friend ModInt power(ModInt a, long long b) {ModInt ret(1); while (b
    > 0) {if (b & 1) ret *= a; a *= a; b >>= 1;} return ret;}
24 friend ModInt inverse(ModInt a) {return power(a, MOD - 2);}
25 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;}
26 friend ostream& operator << (ostream &os, const ModInt &m) {return
    os << m.v;}
27 };

```

11.11 Next Permutation

```

1 sort(v.begin(),v.end());
2 while(next_permutation(v.begin(),v.end())){
3     for(auto u:v){
4         cout<<u<<" ";
5     }
6     cout<<endl;
7 }
8
9 string s="asdfassd";
10 sort(s.begin(),s.end());
11 while(next_permutation(s.begin(),s.end())){
12     cout<<s<<endl;
13 }

```

11.12 Next and Previous Smaller/Greater Element

```

1 vector<int> nextSmaller(vector<int> a, int n){

```

```

2     stack<int> s;
3     vector<int> res(n, n);
4     for(int i=0;i<n;i++){
5         while(s.size() && a[s.top()]>a[i]){
6             res[s.top()]=i;
7             s.pop();
8         }
9         s.push(i);
10    }
11    return res;
12 }
13
14 vector<int> prevSmaller(vector<int> a, int n){
15     stack<int> s;
16     vector<int> res(n, -1);
17     for(int i=n-1;i>=0;i--){
18         while(s.size() && a[s.top()]>a[i]){
19             res[s.top()]=i;
20             s.pop();
21         }
22         s.push(i);
23     }
24     return res;
25 }

```

11.13 Parallel Binary Search

```

1 int lo[maxn], hi[maxn];
2 vector<int> tocheck[maxn];
3
4 bool c=true;
5 while(c){
6     c=false;
7     //initialize changes of structure to 0
8
9     for(int i=0;i<k;i++){
10         if(low[i]!=high[i]){
11             check[(low[i]+high[i])/2].pb(i);
12         }
13     }
14
15     for(int i=0;i<m;i++){
16         // apply change for ith query

```

```

17
18     while(check[i].size()){
19         c=true;
20         int x=check[i].back();
21         check[i].pop_back();
22
23         if(operationToCheck){
24             high[x]=i;
25         }
26         else{
27             low[x]=i+1;
28         }
29     }
30 }
31 }

```

11.14 Random Number Generators

```

1 //to avoid hacks
2 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
3 mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
4 //you can also just write seed_value if hacks are not an issue
5
6 // rng() for generating random numbers between 0 and 2<<31-1
7
8 // for generating numbers with uniform probability in range
9 uniform_int_distribution<int>(0, n)(rng)
10 std::normal_distribution<> normal_dist(mean, 2)
11 exponential_distribution
12
13
14 // for shuffling array
15 shuffle(permutation.begin(), permutation.end(), rng);

```

11.15 setprecision

```

1 cout<<fixed<<setprecision(10);

```

11.16 Ternary Search

```

1 double ternary_search(double l, double r) {
2     double eps = 1e-9;           //set the error limit here
3     while (r - l > eps) {
4         double m1 = l + (r - l) / 3;

```

```

5         double m2 = r - (r - l) / 3;
6         double f1 = f(m1);       //evaluates the function at m1
7         double f2 = f(m2);       //evaluates the function at m2
8         if (f1 < f2)
9             l = m1;
10        else
11            r = m2;
12    }
13    return f(l);                  //return the maximum of f(x) in [l,
14    r]
15 }

```

11.17 Ternary Search Int

```

1 int lo = -1, hi = n;
2 while (hi - lo > 1){
3     int mid = (hi + lo)>>1;
4     if (f(mid) > f(mid + 1))
5         hi = mid;
6     else
7         lo = mid;
8 }
9 //lo + 1 is the answer

```

11.18 XOR Convolution

```

1 void FWHT (int A[], int k, int inv) {
2     for (int j = 0; j < k; j++)
3         for (int i = 0; i < (1 << k); i++)
4             if (~i & (1 << j)) {
5                 int p0 = A[i];
6                 int p1 = A[i | (1 << j)];
7
8                 A[i] = p0 + p1;
9                 A[i | (1 << j)] = p0 - p1;
10
11                 if (inv) {
12                     A[i] /= 2;
13                     A[i | (1 << j)] /= 2;
14                 }
15             }
16 }
17
18 void XOR_conv (int A[], int B[], int C[], int k) {

```

```

28     x = min(x, x ^ basis[i]);
29 }
30 if (x != 0)
31 {
32     basis.pb(x);
33 }
34 }

```

12.1 Berlekamp Massey

```

1 template<typename T>
2 vector<T> berlekampMassey(const vector<T> &s) {
3     vector<T> c;    // the linear recurrence sequence we are building
4     vector<T> oldC; // the best previous version of c to use (the one
5                     // with the rightmost left endpoint)
6     int f = -1;    // the index at which the best previous version of c
7                     // failed on
8     for (int i=0; i<(int)s.size(); i++) {
9         // evaluate c(i)
10        // delta = s_i - \sum_{j=1}^n c_j s_{i-j}
11        // if delta == 0, c(i) is correct
12        T delta = s[i];
13        for (int j=1; j<=(int)c.size(); j++)
14            delta -= c[j-1] * s[i-j];    // c_j is one-indexed, so we
15                                         // actually need index j - 1 in the code
16        if (delta == 0)
17            continue;    // c(i) is correct, keep going
18        // now at this point, delta != 0, so we need to adjust it
19        if (f == -1) {
20            // this is the first time we're updating c
21            // s_i was the first non-zero element we encountered
22            // we make c of length i + 1 so that s_i is part of the base
23            // case
24            c.resize(i + 1);
25            mt19937 rng(chrono::steady_clock::now().time_since_epoch().
26                        count());
27            for (T &x : c)
28                x = rng();    // just to prove that the initial values don
29                               // 't matter in the first step, I will set to random
30                               // values
31            f = i;
32        }
33        // update oldC to be the best previous version of c
34        oldC = c;
35        // update c to be the new version
36        c = vector<T>();
37        for (int j=0; j<=i; j++)
38            c.push_back(s[j] - delta * (j > f ? oldC[j-f-1] : 1));
39    }
40    return c;
41 }

```

```

25     } else {
26         // we need to use a previous version of c to improve on this
           one
27         // apply the 5 steps to build d
28         // 1. set d equal to our chosen sequence
29         vector<T> d = oldC;
30         // 2. multiply the sequence by -1
31         for (T &x : d)
32             x = -x;
33         // 3. insert a 1 on the left
34         d.insert(d.begin(), 1);
35         // 4. multiply the sequence by delta / d(f + 1)
36         T df1 = 0; // d(f + 1)
37         for (int j=1; j<=(int)d.size(); j++)
38             df1 += d[j-1] * s[f+1-j];
39         assert(df1 != 0);
40         T coef = delta / df1; // storing this in outer variable so
           it's O(n^2) instead of O(n^2 log MOD)
41         for (T &x : d)
42             x *= coef;
43         // 5. insert i - f - 1 zeros on the left
44         vector<T> zeros(i - f - 1);
45         zeros.insert(zeros.end(), d.begin(), d.end());
46         d = zeros;
47         // now we have our new recurrence: c + d
48         vector<T> temp = c; // save the last version of c because it
           might have a better left endpoint
49         c.resize(max(c.size(), d.size()));
50         for (int j=0; j<(int)d.size(); j++)
51             c[j] += d[j];
52         // finally, let's consider updating oldC
53         if (i - (int) temp.size() > f - (int) oldC.size()) {
54             // better left endpoint, let's update!
55             oldC = temp;
56             f = i;
57         }
58     }
59 }
60 return c;
61 }

```

12.2 FFT

```

1  using cd = complex<double>;
2  const double PI = acos(-1);
3  //declare size of vectors used like this
4  const int MAXN=2<<19;
5
6  void fft(vector<cd> & a, bool invert) {
7      int n = (int)a.size();
8
9      for (int i = 1, j = 0; i < n; i++) {
10         int bit = n >> 1;
11         for (; j & bit; bit >>= 1)
12             j ^= bit;
13         j ^= bit;
14
15         if (i < j)
16             swap(a[i], a[j]);
17     }
18
19     for (int len = 2; len <= n; len <= 1) {
20         double ang = 2 * PI / len * (invert ? -1 : 1);
21         cd wlen(cos(ang), sin(ang));
22         for (int i = 0; i < n; i += len) {
23             cd w(1);
24             for (int j = 0; j < len / 2; j++) {
25                 cd u = a[i+j], v = a[i+j+len/2] * w;
26                 a[i+j] = u + v;
27                 a[i+j+len/2] = u - v;
28             }
29             w *= wlen;
30         }
31     }
32
33     if (invert) {
34         for (cd & x : a)
35             x /= n;
36     }
37 }
38
39 vector<int> multiply(vector<int> const& a, vector<int> const& b) {
40     vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
41     int n = 1;
42     while (n < a.size() + b.size())
43         n <= 1;

```

```

44     fa.resize(n);
45     fb.resize(n);
46
47     fft(fa, false);
48     fft(fb, false);
49     for (int i = 0; i < n; i++)
50         fa[i] *= fb[i];
51     fft(fa, true);
52
53     vector<int> result(n);
54     for (int i = 0; i < n; i++)
55         result[i] = round(fa[i].real());
56     return result;
57 }
58
59 //normalizing for when mult is between 2 big numbers and not polynomials
60 int carry = 0;
61 for (int i = 0; i < n; i++){
62     result[i] += carry;
63     carry = result[i] / 10;
64     result[i] %= 10;
65 }

```

12.3 NTT

```

1 // number theory transform
2
3 const int MOD = 998244353, ROOT = 3;
4 // const int MOD = 7340033, ROOT = 5;
5 // const int MOD = 167772161, ROOT = 3;
6 // const int MOD = 469762049, ROOT = 3;
7
8 int power(int base, int exp) {
9     int res = 1;
10    while (exp) {
11        if (exp % 2) res = 1LL * res * base % MOD;
12        base = 1LL * base * base % MOD;
13        exp /= 2;
14    }
15    return res;
16 }
17
18 void ntt(vector<int>& a, bool invert) {

```

```

19     int n = a.size();
20     for (int i = 1, j = 0; i < n; i++) {
21         int bit = n >> 1;
22         for (; j & bit; bit >>= 1) j ^= bit;
23         j ^= bit;
24         if (i < j) swap(a[i], a[j]);
25     }
26     for (int len = 2; len <= n; len <<= 1) {
27         int wlen = power(ROOT, (MOD - 1) / len);
28         if (invert) wlen = power(wlen, MOD - 2);
29         for (int i = 0; i < n; i += len) {
30             int w = 1;
31             for (int j = 0; j < len / 2; j++) {
32                 int u = a[i + j], v = 1LL * a[i + j + len / 2] * w % MOD;
33                 a[i + j] = u + v < MOD ? u + v : u + v - MOD;
34                 a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + MOD;
35                 w = 1LL * w * wlen % MOD;
36             }
37         }
38     }
39     if (invert) {
40         int n_inv = power(n, MOD - 2);
41         for (int& x : a) x = 1LL * x * n_inv % MOD;
42     }
43 }
44
45 vector<int> multiply(vector<int>& a, vector<int>& b) {
46     int n = 1;
47     while (n < a.size() + b.size()) n <<= 1;
48     a.resize(n), b.resize(n);
49     ntt(a, false), ntt(b, false);
50     for (int i = 0; i < n; i++) a[i] = 1LL * a[i] * b[i] % MOD;
51     ntt(a, true);
52     return a;
53 }
54 // usage
55 // vector<int> a = {1, 2, 3}, b = {4, 5, 6};
56 // vector<int> c = multiply(a, b);
57 // for (int x : c) cout << x << " ";

```

12.4 Roots NTT

```

1 1*2^0 + 1 = 2, 1, 1

```

```

2 1*2^1 + 1 = 3, 2, 2
3 1*2^2 + 1 = 5, 2, 3
4 2*2^3 + 1 = 17, 2, 9
5 1*2^4 + 1 = 17, 3, 6
6 3*2^5 + 1 = 97, 19, 46
7 3*2^6 + 1 = 193, 11, 158
8 2*2^7 + 1 = 257, 9, 200
9 1*2^8 + 1 = 257, 3, 86
10 15*2^9 + 1 = 7681, 62, 1115
11 12*2^10 + 1 = 15361, 49, 1254
12 6*2^11 + 1 = 12289, 7, 8778
13 3*2^12 + 1 = 12289, 41, 4496
14 5*2^13 + 1 = 40961, 12, 23894
15 4*2^14 + 1 = 65537, 15, 30584
16 2*2^15 + 1 = 65537, 9, 7282
17 1*2^16 + 1 = 65537, 3, 21846
18 6*2^17 + 1 = 786433, 8, 688129
19 3*2^18 + 1 = 786433, 5, 471860
20 11*2^19 + 1 = 5767169, 12, 3364182
21 7*2^20 + 1 = 7340033, 5, 4404020
22 11*2^21 + 1 = 23068673, 38, 21247462
23 25*2^22 + 1 = 104857601, 21, 49932191
24 20*2^23 + 1 = 167772161, 4, 125829121
25 10*2^24 + 1 = 167772161, 2, 83886081
26 5*2^25 + 1 = 167772161, 17, 29606852
27 7*2^26 + 1 = 469762049, 30, 15658735
28 15*2^27 + 1 = 2013265921, 137, 749463956
29 12*2^28 + 1 = 3221225473, 8, 2818572289
30 6*2^29 + 1 = 3221225473, 14, 1150437669
31 3*2^30 + 1 = 3221225473, 13, 1734506024
32 35*2^31 + 1 = 75161927681, 93, 44450602392
33 18*2^32 + 1 = 77309411329, 106, 5105338484

```

13 Scripts

13.1 build.sh

This file should be called before stress.sh or validate.sh. build.sh name.cpp

```

1 g++ -static -DLOCAL -lm -s -x c++ -Wall -Wextra -O2 -std=c++17 -o $1 $1.
  cpp

```

13.2 stress.sh

Format is stress.sh Awrong Aslow Agen Numtests

```

1 #!/usr/bin/env bash
2
3 for ((testNum=0;testNum<$4;testNum++))
4 do
5     ./ $3 > input
6     ./ $2 < input > outSlow
7     ./ $1 < input > outWrong
8     H1='md5sum outWrong'
9     H2='md5sum outSlow'
10    if !(cmp -s "outWrong" "outSlow")
11    then
12        echo "Error_found!"
13        echo "Input:"
14        cat input
15        echo "Wrong_Output:"
16        cat outWrong
17        echo "Slow_Output:"
18        cat outSlow
19        exit
20    fi
21 done
22 echo Passed $4 tests

```

13.3 validate.sh

Format is validate.sh Awrong Avalidator Agen NumTests

```

1 #!/usr/bin/env bash
2
3 for ((testNum=0;testNum<$4;testNum++))
4 do
5     ./ $3 > input
6     ./ $1 < input > out
7     cat input out > data
8     ./ $2 < data > res
9     result=$(cat res)
10    if [ "${result:0:2}" != "OK" ];
11    then
12        echo "Error_found!"
13        echo "Input:"

```

```

14     cat input
15     echo "Output:"
16     cat out
17     echo "Validator_Result:"
18     cat res
19     exit
20 fi
21 done
22 echo Passed $4 tests

```

14 Strings

14.1 Hashed String

```

1  /*
2
3      Hashed string
4  -----
5  Class for hashing string. Allows retrieval of hashes of any substring
6  in the string.
7
8  Double hash or use big mod values to avoid problems with collisions
9
10 Time Complexity(Construction): O(n)
11 Space Complexity: O(n)
12 */
13 const ll MOD = 212345678987654321LL;
14 const ll base = 33;
15
16 class HashedString {
17     private:
18         // change M and B if you want
19         static const long long M = 1e9 + 9;
20         static const long long B = 9973;
21
22         // pow[i] contains B^i % M
23         static vector<long long> pow;
24
25         // p_hash[i] is the hash of the first i characters of the given string
26         vector<long long> p_hash;
27

```

```

28     public:
29     HashedString(const string &s) : p_hash(s.size() + 1) {
30         while (pow.size() < s.size()) { pow.push_back((pow.back() * B) % M);
31         }
32
33         p_hash[0] = 0;
34         for (int i = 0; i < s.size(); i++) {
35             p_hash[i + 1] = ((p_hash[i] * B) % M + s[i]) % M;
36         }
37     }
38
39     // Returns hash of substring [start, end]
40     long long get_hash(int start, int end) {
41         long long raw_val =
42             (p_hash[end + 1] - (p_hash[start] * pow[end - start + 1]));
43         return (raw_val % M + M) % M;
44     };
45
46     // you cant skip this
47     vector<long long> HashedString::pow = {1};

```

14.2 KMP

```

1  /*
2
3      KMP
4  -----
5  Computes the prefix function for a string.
6
7  Maximum length of substring that ends at position i and is proper
8  prefix (not equal to string itself) of string
9  pf[i] is the length of the longest proper prefix of the substrings
10 [0.....i]$ which is also a suffix of this substring.
11
12 For matching, one can append the string with a delimiter like $
13 between them
14
15 Time Complexity: O(n)
16 Space Complexity: O(n)
17 */
18
19 vector<int> KMP(string s){
20     int n=(int)s.length();
21     vector<int> pf(n, 0);

```

```

18     for(int i=1;i<n;i++){
19         int j=pf[i-1];
20         while(j>0 && s[i]!=s[j]){
21             j=pf[j-1];
22         }
23         if(s[i]==s[j]){
24             pf[i]=j+1;
25         }
26     }
27     return pf;
28 }
29
30 // Counts how many times each prefix occurs
31 // Same thing can be done for two strings but only considering indices
   of second string
32 vector<int> count_occurrences_of_prefixes(vector<int> pf){
33     int n=(int)pf.size();
34     vector<int> ans(n + 1);
35     for (int i = 0; i < n; i++)
36         ans[pi[i]]++;
37     for (int i = n-1; i > 0; i--)
38         ans[pi[i-1]] += ans[i];
39     for (int i = 0; i <= n; i++)
40         ans[i]++;
41 }
42
43 // Computes automaton for string
44 // useful for not having to recalculate KMP of string s
45 // can be utilized when the second string (the one in which we are
   trying to count occurrences)
46 // is very large
47 void compute_automaton(string s, vector<vector<int>>& aut) {
48     s += '#';
49     int n = s.size();
50     vector<int> pi = KMP(s);
51     aut.assign(n, vector<int>(26));
52     for (int i = 0; i < n; i++) {
53         for (int c = 0; c < 26; c++) {
54             if (i > 0 && 'a' + c != s[i])
55                 aut[i][c] = aut[pi[i-1]][c];
56             else
57                 aut[i][c] = i + ('a' + c == s[i]);
58         }

```

```

59     }
60 }

```

14.3 Least Rotation String

```

1  /*
2      Min cyclic shift
3      -----
4      Finds the lexicographically minimum cyclic shift of a string
5
6      Time Complexity: O(n)
7      Space Complexity: O(n)
8  */
9
10 string least_rotation(string s)
11 {
12     s += s;
13     vector<int> f(s.size(), -1);
14     int k = 0;
15     for(int j = 1; j < s.size(); j++)
16     {
17         char sj = s[j];
18         int i = f[j - k - 1];
19         while(i != -1 && sj != s[k + i + 1])
20         {
21             if(sj < s[k + i + 1]){
22                 k = j - i - 1;
23             }
24             i = f[i];
25         }
26         if(sj != s[k + i + 1])
27         {
28             if(sj < s[k]){
29                 k = j;
30             }
31             f[j - k] = -1;
32         }
33         else
34             f[j - k] = i + 1;
35     }
36     return s.substr(k, s.size() / 2);
37 }

```

14.4 Manacher

```

1  /*
2      Manacher
3      -----
4      Computes the length of the longest palindrome centered at position i.
5
6      p[i] is length of biggest palindrome centered in this position.
7      Be careful with characters that are inserted to account for odd and
8          even palindromes
9
10     Time Complexity: O(n)
11     Space Complexity: O(n)
12 */
13
14 // Number of palindromes centered at each position
15
16 vector<int> manacher_odd(string s)
17 {
18     int n = s.size();
19     s = "$" + s + "~";
20     vector<int> p(n + 2);
21     int l = 1, r = 1;
22     for (int i = 1; i <= n; i++)
23     {
24         p[i] = max(0, min(r - i, p[l + (r - i)]));
25         while (s[i - p[i]] == s[i + p[i]])
26         {
27             p[i]++;
28         }
29         if (i + p[i] > r)
30         {
31             l = i - p[i], r = i + p[i];
32         }
33     }
34     return vector<int>(begin(p) + 1, end(p) - 1);
35 }
36 vector<int> manacher(string s)
37 {
38     string t;
39     for (auto c : s)
40     {

```

```

41         t += string("#") + c;
42     }
43     auto res = manacher_odd(t + "#");
44     return vector<int>(begin(res) + 1, end(res) - 1);
45 }
46
47 // usage
48 // vector<int> p = manacher("abacaba");
49 // this will return {2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2}
50 // vector<int> p = manacher("abaaba");
51 // this will return {2, 1, 4, 1, 2, 7, 2, 1, 4, 1, 2}

```

14.5 Suffix Array

```

1  /*
2      Suffix Array
3      -----
4      Computes the suffix array of a string in O(n log n).
5      Sorted array of all cyclic shifts of a string.
6
7      If you want sorted suffixes append $ to the end of the string.
8      lc is longest common prefix. Lcp of two substrings j > i is min(lc[i],
9          ....., lc[j - 1]).
10
11     To compute Largest common substring of multiple strings
12     Join all strings separating them with special character like $ (it has
13         to be different for each string)
14     Sliding window on lcp array (all string have to appear on the sliding
15         window and
16         the lcp of the interval will give the length of the substring that
17         appears on all strings)
18
19     Time Complexity: O(n log n)
20     Space Complexity: O(n)
21 */
22
23 struct SuffixArray
24 {
25     int n;
26     string t;
27     vector<int> sa, rk, lc;
28     SuffixArray(const std::string &s)

```

```

26 {
27     n = s.length();
28     t = s;
29     sa.resize(n);
30     lc.resize(n - 1);
31     rk.resize(n);
32     std::iota(sa.begin(), sa.end(), 0);
33     std::sort(sa.begin(), sa.end(), [&](int a, int b)
34         { return s[a] < s[b]; });
35     rk[sa[0]] = 0;
36     for (int i = 1; i < n; ++i)
37         rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
38     int k = 1;
39     std::vector<int> tmp, cnt(n);
40     tmp.reserve(n);
41     while (rk[sa[n - 1]] < n - 1)
42     {
43         tmp.clear();
44         for (int i = 0; i < k; ++i)
45             tmp.push_back(n - k + i);
46         for (auto i : sa)
47             if (i >= k)
48                 tmp.push_back(i - k);
49         std::fill(cnt.begin(), cnt.end(), 0);
50         for (int i = 0; i < n; ++i)
51             ++cnt[rk[i]];
52         for (int i = 1; i < n; ++i)
53             cnt[i] += cnt[i - 1];
54         for (int i = n - 1; i >= 0; --i)
55             sa[--cnt[rk[tmp[i]]]] = tmp[i];
56         std::swap(rk, tmp);
57         rk[sa[0]] = 0;
58         for (int i = 1; i < n; ++i)
59             rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i]
60                 - 1 + k == n || tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
61         k *= 2;
62     }
63     for (int i = 0, j = 0; i < n; ++i)
64     {
65         if (rk[i] == 0)
66         {
67             j = 0;

```

```

68     else
69     {
70         for (j -= j > 0; i + j < n && sa[rk[i] - 1] + j < n && s[i + j]
71             == s[sa[rk[i] - 1] + j];)
72             ++j;
73         lc[rk[i] - 1] = j;
74     }
75 }
76
77 // Finds if string p appears as substring in the string
78 // might now work perfectly
79 int search(string &p){
80     int tam = p.size();
81     int l = 0, r = n;
82
83     string tmp = "";
84     while(r > l) {
85         int m = l + (r-l)/2;
86         tmp = t.substr(sa[m], min(n-sa[m], tam));
87         if(tmp >= p){
88             r = m;
89         } else {
90             l = m + 1;
91         }
92     }
93     if(l < n) {
94         tmp = t.substr(sa[l], min(n-sa[l], tam));
95     } else{
96         return -1;
97     }
98     if(tmp == p){
99         return l;
100     } else {
101         return -1;
102     }
103 }
104
105 // Counts number of times a string p appears as substring in string
106 int count(string &p) {
107     int x = search(p);
108     if(x == -1) return 0;
109

```

```

110     int cnt = 0;
111     int tam = p.size();
112     int maxx = 0;
113     while((1 << maxx) + x < n) maxx++;
114     int y = x;
115     for(int i = maxx-1; i >= 0; i--) {
116         if(x + (1 << i) >= n) continue;
117         string tmp = t.substr(sa[x + (1 << i)], min(n-sa[x + (1 << i)
118             ], tam));
119         if(tmp == p) x += (1 << i);
120     }
121     return x-y+1;
122 }
123 int main() {
124     cin.tie(0)->sync_with_stdio(0);
125     string s; cin >> s;
126     SuffixArray SA(s);
127
128     int q; cin >> q;
129     for(int t = 0; t < q; t++) {
130         string tmp; cin >> tmp;
131         cout << SA.count(tmp) << endl;
132     }
133
134     return 0;
135 }

```

14.6 Suffix Automaton

```

1  /*
2
3      Suffix Automaton
4
5  -----
6
7  Constructs suffix automaton for a given string.
8  Be careful with overlapping substrings.
9
10 Firstposition if first position string ends in.
11 If you want starting index you need to substract length of the string
12    being searched.
13
14 len is length of longest string of state
15
16 Time Complexity(Construction): O(n)

```

```

13     Space Complexity: O(n)
14 */
15
16
17 struct state {
18     int len, link, firstposition;
19     vector<int> inv_link; // can skip for almost everything
20     map<char, int> next;
21 };
22
23 const int MAXN = 100000;
24 state st[MAXN * 2];
25 ll cnt[MAXN*2], cntPaths[MAXN*2], cntSum[MAXN*2], cnt1[2 * MAXN];
26 int sz, last;
27
28 // call this first
29 void initSuffixAutomaton() {
30     st[0].len = 0;
31     st[0].link = -1;
32     sz++;
33     last = 0;
34 }
35
36 // construction is O(n)
37 void insertChar(char c) {
38     int cur = sz++;
39     st[cur].len = st[last].len + 1;
40     st[cur].firstposition = st[last].len;
41     int p = last;
42     while (p != -1 && !st[p].next.count(c)) {
43         st[p].next[c] = cur;
44         p = st[p].link;
45     }
46     if (p == -1) {
47         st[cur].link = 0;
48     } else {
49         int q = st[p].next[c];
50         if (st[p].len + 1 == st[q].len) {
51             st[cur].link = q;
52         } else {
53             int clone = sz++;
54             st[clone].len = st[p].len + 1;
55             st[clone].next = st[q].next;

```

```

56     st[clone].link = st[q].link;
57     st[clone].firstposition=st[q].firstposition;
58     while (p != -1 && st[p].next[c] == q) {
59         st[p].next[c] = clone;
60         p = st[p].link;
61     }
62     st[q].link = st[cur].link = clone;
63 }
64 }
65 last = cur;
66 cnt[last]=1;
67 }
68
69 // searches for the starting position in O(len(s)). Returns starting
    index of first ocurrence or -1 if it does not appear.
70 int search(string s){
71     int cur=0, i=0, n=(int)s.length();
72     while(i<n){
73         if(!st[cur].next.count(s[i])) return -1;
74         cur=st[cur].next[s[i]];
75         i++;
76     }
77     //sumar 2 si se quiere 1 indexado
78     return st[cur].firstposition-n+1;
79 }
80
81 void dfs(int cur){
82     cntPaths[cur]=1;
83     for(auto [x, y]:st[cur].next){
84         if(cntPaths[y]==0) dfs(y);
85         cntPaths[cur]+=cntPaths[y];
86     }
87 }
88
89 // Counts how many paths exist from state. How many substrings exist
    from a specific state.
90 // Stored in cntPaths
91 void countPaths(){
92     dfs(0);
93 }
94
95 // Computes the number of times each state appears
96 void countOcurrances(){

```

```

97     vector<pair<int, int>> a;
98     for(int i=sz-1;i>0;i--){
99         a.push_back({st[i].len, i});
100    }
101    sort(a.begin(), a.end());
102    for(int i=sz-2;i>=0;i--){
103        cnt[st[a[i].second].link]+=cnt[a[i].second];
104    }
105 }
106
107 void dfs1(int cur){
108     for(auto [x, y]:st[cur].next){
109         if(cntSum[y]==cnt[y]) dfs1(y);
110         cntSum[cur]+=cntSum[y];
111     }
112 }
113
114 // Computes the number of times each state or any of its children appear
    in the string.
115 void countSumOcurrances(){
116     for(int i=0;i<sz;i++){
117         cntSum[i]=cnt[i];
118     }
119     dfs1(0);
120 }
121
122
123 // Counts number of paths that can reach specific state.
124 void countPathsReverse(){
125     cnt1[0]=1;
126     queue<int> q;
127     q.push(0);
128     vector<int> in(2*MAXN, 0);
129     for(int i=0;i<sz;i++){
130         for(auto [x, y]:st[i].next){
131             in[y]++;
132         }
133     }
134     while((int)q.size()){
135         int cur=q.front();
136         q.pop();
137         for(auto [x, y]:st[cur].next){
138             cnt1[y]+=cnt1[cur];

```

```

139     in[y]--;
140     if(in[y]==0){
141         q.push(y);
142     }
143 }
144 }
145 }
146
147 // Computes the kth smallest string that appears on the string (counting
    repetitions)
148 string kthSmallest(ll k){
149     string s="";
150     int cur=0;
151     while(k>0){
152         for(auto [c, y]:st[cur].next){
153             if(k>cntSum[y]) k-=cntSum[y];
154             else{
155                 k-=cnt[y];
156                 s+=c;
157                 cur=y;
158                 break;
159             }
160         }
161     }
162     return s;
163 }
164
165 // Computes the kth smallest string that appears on the string (without
    counting repetitions)
166 string kthSmallestDistinct(ll k){
167     string s="";
168     int cur=0;
169     while(k>0){
170         for(auto [c, y]:st[cur].next){
171             if(k>cntPaths[y]) k-=cntPaths[y];
172             else{
173                 k--;
174                 s+=c;
175                 cur=y;
176                 break;
177             }
178         }
179     }

```

```

180     }
181     return s;
182 }
183
184 // Precomputation to find all occurrences of a substring
185 void precumppte_for_all_ocurrences(){
186     for (int v = 1; v < sz; v++) {
187         st[st[v].link].inv_link.push_back(v);
188     }
189 }
190
191 // Finding all ocurrences of substring in string
192 // P_length is length of substring
193 // v is state where first occurrence happens
194 // be careful as indices can appear multiple times due to clone states
195 // if you want to avoid duplicate positions utilize set or have a flag
    for each state to know if it is clone or not
196 void output_all_ocurrences(int v, int P_length) {
197     cout << st[v].firstposition - P_length + 1 << endl;
198     for (int u : st[v].inv_link)
199         output_all_ocurrences(u, P_length);
200 }
201
202 //longest common substring
203 //build automaton for s first
204 string lcs (string S, string T) {
205     int v = 0, l = 0, best = 0, bestpos = 0;
206     for (int i = 0; i < T.size(); i++) {
207         while (v && !st[v].next.count(T[i])) {
208             v = st[v].link ;
209             l = st[v].len;
210         }
211         if (st[v].next.count(T[i])) {
212             v = st [v].next[T[i]];
213             l++;
214         }
215         if (l > best) {
216             best = l;
217             bestpos = i;
218         }
219     }
220 }
221

```

```

222     return T.substr(bestpos - best + 1, best);
223 }
224
225
226 int main(){
227     ios_base::sync_with_stdio(false); cin.tie(NULL);
228     string s; cin >> s;
229     initSuffixAutomaton();
230     for(char c:s){
231         insertChar(c);
232     }
233 }

```

14.7 Trie Ahocorasick

```

1  /*
2      Trie - AhoCorasick
3      -----
4      Builds a trie for subset of strings and computes suffix links.
5      KATCL implementation is cleaner.
6
7      Time Complexity(Construction): O(m) where m is sum
8      of lengths of strings
9      Space Complexity: O(m)
10 */
11
12
13
14 const int K = 26;
15
16 struct Vertex {
17     int next[K];
18     bool output = false;
19     int p = -1;
20     char pch;
21     int link = -1;
22     int go[K];
23
24     Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
25         fill(begin(next), end(next), -1);
26         fill(begin(go), end(go), -1);
27     }
28 };

```

```

29
30 vector<Vertex> t(1);
31
32 void add_string(string const& s) {
33     int v = 0;
34     for (char ch : s) {
35         int c = ch - 'a';
36         if (t[v].next[c] == -1) {
37             t[v].next[c] = t.size();
38             t.emplace_back(v, ch);
39         }
40         v = t[v].next[c];
41     }
42     t[v].output = true;
43 }
44
45 int go(int v, char ch);
46
47 int get_link(int v) {
48     if (t[v].link == -1) {
49         if (v == 0 || t[v].p == 0)
50             t[v].link = 0;
51         else
52             t[v].link = go(get_link(t[v].p), t[v].pch);
53     }
54     return t[v].link;
55 }
56
57 int go(int v, char ch) {
58     int c = ch - 'a';
59     if (t[v].go[c] == -1) {
60         if (t[v].next[c] != -1)
61             t[v].go[c] = t[v].next[c];
62         else
63             t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
64     }
65     return t[v].go[c];
66 }

```

14.8 Z Function

```

1  /*
2      Z_function

```

```

3 -----
4 Computes the z_function for any string.
5 ith element is equal to the greatest number of characters starting
6   from the position i that coincide with the first characters of s
7
8 z[i] length of the longest string that is, at the same time, a prefix
9   of s and a prefix of the suffix of $$$ starting at i.
10
11 to compress string, one can run z_function and then find the smallest
12   i that divides n such that i + z[i] = n
13
14
15 Time Complexity: O(n)
16 Space Complexity: O(n)
17 */
18
19 vector<int> z_function(string s) {
20     int n = s.size();
21     vector<int> z(n);
22     int l = 0, r = 0;
23     for(int i = 1; i < n; i++) {
24         if(i < r) {
25             z[i] = min(r - i, z[i - l]);
26         }
27         while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
28             z[i]++;
29         }
30         if(i + z[i] > r) {
31             l = i;
32             r = i + z[i];
33         }
34     }
35     return z;
36 }
37
38 // usage
39 // vector<int> z = z_function("abacaba");
40 // this will return {0, 0, 1, 0, 3, 0, 1}
41 // vector<int> z = z_function("aaaaa");
42 // this will return {0, 4, 3, 2, 1}
43 // vector<int> z = z_function("aaabaab");
44 // this will return {0, 2, 1, 0, 2, 1, 0}

```

15 Trees

15.1 Centroid Decomposition

```

1 /*
2
3 -----
4 Finds the centroid decomposition of a given tree.
5 Any vertex can have at most log n centroid ancestors
6
7 The code below is the solution to Xenia and tree.
8 Given tree, queries of two types:
9 1) u - color vertex u
10 2) v - print minimum distance of vertex v to any colored vertex before
11
12
13 Time Complexity: O(n log n)
14 Space Complexity: O(n log n)
15 */
16 const int MAXN=200005;
17
18 vector<int> adj[MAXN];
19 vector<bool> is_removed(MAXN, false);
20 vector<int> subtree_size(MAXN, 0);
21 vector<int> dis(MAXN, 1e9);
22 vector<vector<pair<int, int>>> ancestor(MAXN);
23
24 int get_subtree_size(int node, int parent = -1) {
25     subtree_size[node] = 1;
26     for (int child : adj[node]) {
27         if (child == parent || is_removed[child]) { continue; }
28         subtree_size[node] += get_subtree_size(child, node);
29     }
30     return subtree_size[node];
31 }
32
33 int get_centroid(int node, int tree_size, int parent = -1) {
34     for (int child : adj[node]) {
35         if (child == parent || is_removed[child]) { continue; }
36         if (subtree_size[child] * 2 > tree_size) {
37             return get_centroid(child, tree_size, node);
38         }
39     }

```

```

40     return node;
41 }
42
43 void getDist(int cur, int centroid, int p=-1, int dist=1){
44     for (int child:adj[cur]){
45         if(child==p || is_removed[child])
46             continue;
47         dist++;
48         getDist(child, centroid, cur, dist);
49         dist--;
50     }
51     ancestor[cur].push_back(make_pair(centroid, dist));
52 }
53
54 void update(int cur){
55     for (int i=0;i<ancestor[cur].size();i++){
56         dis[ancestor[cur][i].first]=min(dis[ancestor[cur][i].first],
57             ancestor[cur][i].second);
58     }
59     dis[cur]=0;
60 }
61
62 int query(int cur){
63     int mini=dis[cur];
64     for (int i=0;i<ancestor[cur].size();i++){
65         mini=min(mini, ancestor[cur][i].second+dis[ancestor[cur][i].first]);
66     }
67     return mini;
68 }
69
70 void build_centroid_decomp(int node = 1) {
71     int centroid = get_centroid(node, get_subtree_size(node));
72
73     for (int child : adj[centroid]) {
74         if (is_removed[child]) { continue; }
75         getDist(child, centroid, centroid);
76     }
77
78     is_removed[centroid] = true;
79
80     for (int child : adj[centroid]) {
81         if (is_removed[child]) { continue; }

```

```

81     build_centroid_decomp(child);
82 }
83 }

```

15.2 Heavy Light Decomposition

```

1  /*
2      Heavy Light Decomposition(HLD)
3      -----
4      Constructs the heavy light decomposition of a tree
5
6      Splits the tree into several paths so that we can reach the root
7      vertex from any v by traversing at most log n paths.
8      In addition, none of these paths intersect with another.
9
10     Time Complexity(Creation): O(n log n)
11     Time Complexity(Query): O((log n) ^ 2) usually, depending on the query
12     itself
13     Space Complexity: O(n)
14 */
15
16 //call dfs1 first
17 struct SegmentTree {
18     vector<ll> a;
19     int n;
20
21     SegmentTree(int _n) : a(2 * _n, 0), n(_n) {}
22
23     void update(int pos, ll val) {
24         for (a[pos += n] = val; pos > 1; pos >>= 1) {
25             a[pos / 2] = (a[pos] ^ a[pos ^ 1]);
26         }
27     }
28
29     ll get(int l, int r) {
30         ll res = 0;
31         for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
32             if (l & 1) {
33                 res ^= a[l++];
34             }
35             if (r & 1) {
36                 res ^= a[--r];
37             }
38         }
39     }

```

```

36     }
37     return res;
38 }
39 };
40
41
42 const int MAXN=500005;
43 vector<int> adj[MAXN];
44 SegmentTree st(MAXN);
45 int a[MAXN], sz[MAXN], to[MAXN], dpth[MAXN], s[MAXN], par[MAXN];
46 int cnt=0;
47
48 void dfs1(int cur, int p){
49     sz[cur]=1;
50     for(int x:adj[cur]){
51         if(x==p) continue;
52         dpth[x]=dpth[cur]+1;
53         par[x]=cur;
54         dfs1(x, cur);
55         sz[cur]+=sz[x];
56     }
57 }
58
59 void dfs(int cur, int p, int l){
60     st.update(cnt, a[cur]);
61     s[cur]=cnt++;
62     to[cur]=1;
63     int g=-1;
64     for(int x:adj[cur]){
65         if(x==p) continue;
66         if(g==-1 || sz[g]<sz[x]){
67             g=x;
68         }
69     }
70     if(g==-1) return;
71     dfs(g, cur, l);
72     for(int x:adj[cur]){
73         if(x==p || x==g) continue;
74         dfs(x, cur, x);
75     }
76 }
77
78 int query(int u, int v){

```

```

79     int res=0;
80     while(to[u]!=to[v]){
81         if(dpth[to[u]]<dpth[to[v]]) swap(u, v);
82         res^=st.get(s[to[u]], s[u]+1);
83         u=par[to[u]];
84     }
85     if(dpth[u]>dpth[v]) swap(u, v);
86     res^=st.get(s[u], s[v]+1);
87     return res;
88 }
89
90
91
92
93 //alternate implementation
94 vector<int> parent, depth, heavy, head, pos;
95 int cur_pos;
96
97 int dfs(int v, vector<vector<int>> const& adj) {
98     int size = 1;
99     int max_c_size = 0;
100     for (int c : adj[v]) {
101         if (c != parent[v]) {
102             parent[c] = v, depth[c] = depth[v] + 1;
103             int c_size = dfs(c, adj);
104             size += c_size;
105             if (c_size > max_c_size)
106                 max_c_size = c_size, heavy[v] = c;
107         }
108     }
109     return size;
110 }
111
112 void decompose(int v, int h, vector<vector<int>> const& adj) {
113     head[v] = h, pos[v] = cur_pos++;
114     if (heavy[v] != -1)
115         decompose(heavy[v], h, adj);
116     for (int c : adj[v]) {
117         if (c != parent[v] && c != heavy[v])
118             decompose(c, c, adj);
119     }
120 }
121

```

```

122 void init(vector<vector<int>> const& adj) {
123     int n = adj.size();
124     parent = vector<int>(n);
125     depth = vector<int>(n);
126     heavy = vector<int>(n, -1);
127     head = vector<int>(n);
128     pos = vector<int>(n);
129     cur_pos = 0;
130
131     dfs(0, adj);
132     decompose(0, 0, adj);
133 }
134
135 int query(int a, int b) {
136     int res = 0;
137     for (; head[a] != head[b]; b = parent[head[b]]) {
138         if (depth[head[a]] > depth[head[b]])
139             swap(a, b);
140         int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
141         res = max(res, cur_heavy_path_max);
142     }
143     if (depth[a] > depth[b])
144         swap(a, b);
145     int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
146     res = max(res, last_heavy_path_max);
147     return res;
148 }

```

15.3 Lowest Common Ancestor (LCA)

```

1  /*
2      LCA(Lowest Common Ancestor)
3      -----
4      Computes the lowest common ancestor of two vertices in a tree.
5
6      Be careful as implementation is indexed starting with 1
7
8      Time Complexity(Creation): O(n log n)
9      Time Complexity(Query): O(log n)
10     Space Complexity: O(n log n)
11  */
12

```

```

13 const int N=200005;
14 vector<int> adj[N];
15 vector<int> start(N), end1(N), depth(N);
16 vector<vector<int>> t(N, vi(32));
17 int timer=0;
18 int n, l;
19 // l=(int)ceil(log2(n))
20 // call dfs(1, 1, 0)
21 // 1 indexed, dont use 0 indexing
22
23
24 void dfs(int cur, int p, int cnt){
25     depth[cur]=cnt;
26     t[cur][0]=p;
27     start[cur]=timer++;
28     for(int i=1;i<=l;i++){
29         t[cur][i]=t[t[cur][i-1]][i-1];
30     }
31     for(int x:adj[cur]){
32         if(x==p) continue;
33         dfs(x, cur, cnt+1);
34     }
35     end1[cur]=++timer;
36 }
37
38 bool ancestor(int u, int v){
39     return start[u]<=start[v] && end1[u]>=end1[v];
40 }
41
42 int lca(int u, int v){
43     if(ancestor(u, v))
44         return u;
45     if (ancestor(v, u)){
46         return v;
47     }
48     for(int i=l;i>=0;i--){
49         if(!ancestor(t[u][i], v)){
50             u=t[u][i];
51         }
52     }
53     return t[u][0];
54 }

```

15.4 Tree Diameter

```
1  /*
2      Tree Diameter
3      -----
4  Finds the vertex most distant to vertex on which function is called.
5
6  The first value is the vertex itself and the second value is the
   distance.
7
8  To find diameter run algorithm twice, first on random vertex and then
   on the vertex that is farthest away.
9
10 The vertex that is the farthest away from any vertex in tree must be
   an endpoint of the diameter.
11
12 Time Complexity: O(n)
13 Space Complexity: O(n)
14 */
15
16 pair<int, int> dfs(const vector<vector<int>> &tree, int node = 1,
17     int previous = 0, int length = 0) {
18     pair<int, int> max_path = {node, length};
19     for (const int &i : tree[node]) {
20         if (i == previous) { continue; }
21         pair<int, int> other = dfs(tree, i, node, length + 1);
22         if (other.second > max_path.second) { max_path = other; }
23     }
24     return max_path;
25 }
```