

Contents**1 Funciones C++**

4

3.35 Sparse Table	41
3.36 Sparse Table 2D	41
3.37 Square Root Decomposition	43
3.38 Treap	44
3.39 Treap 2	45
3.40 Treap With Inversion	46
3.41 Treap With Inversions and Range Updates	47

2 Compile

4

2.1 Compile	4
2.2 Template	4

3 Data Structures

5

3.1 BIT	5
3.2 Bitset	5
3.3 Bit Trie	5
3.4 CHMIN/CHMAX Rangesum GCD Queries	6
3.5 Disjoint Sparse Table	9
3.6 Disjoint Set Union Bipartite	10
3.7 Disjoint Set Union	11
3.8 Disjoint Set Union Queue Like Rollback	12
3.9 Dynamic Conectivity	15
3.10 Fenwick Tree	17
3.11 Fenwick Tree 2D	17
3.12 Li Chao Tree	18
3.13 Link Cut Tree	19
3.14 Linked List	22
3.15 Merge Sort Tree	23
3.16 Minimum Cartesian Tree	24
3.17 Multi Ordered Set	25
3.18 Ordered Set	25
3.19 Palindromic Tree	25
3.20 Persistent Array	27
3.21 Persistent Segment Tree	28
3.22 Polynomial Queries	29
3.23 Segment Tree	30
3.24 Segment Tree 2D	31
3.25 Segment Tree Dynamic	32
3.26 Segment Tree Lazy Types	32
3.27 Segment Tree Lazy	33
3.28 Segment Tree Lazy Range Set	34
3.29 Segment Tree Lazy Range Set 2	35
3.30 Segment Tree Max Subarray Sum	37
3.31 Segment Tree Range Update	38
3.32 Segment Tree Struct Types	38
3.33 Segment Tree Struct	39
3.34 Segment Tree Walk	40

4 Dynamic Programming

49

4.1 CHT Deque	49
4.2 Digit DP	50
4.3 Divide and Conquer DP	50
4.4 Edit Distance	51
4.5 Knuth's Algorithm	51
4.6 LCS	52
4.7 Line Container	53
4.8 Longest Increasing Subsequence	53
4.9 SOS DP	54

5 Flow

54

5.1 Dinic	54
5.2 Hopcroft-Karp	56
5.3 Hungarian	57
5.4 Max Flow Min Cost	58
5.5 Max Flow	60
5.6 Min Cost Max Flow	60
5.7 Push Relabel	61

6 Geometry

62

6.1 Point Struct	62
6.2 Sort Points	63
6.3 Point Struct2	63
6.4 Find Triangle With Given Area	64
6.5 Antipodal Pairs	65
6.6 Area and Perimeter	66
6.7 Area Union Circles	66
6.8 Centroid	67
6.9 Circle Inside Circle	67
6.10 Circle Outside Circle	67
6.11 Closest Pair of Points	67
6.12 Common Tangents	67
6.13 Convex Hull	68
6.14 Segment Intersection with Ray	68

6.15 Cut Polygon	68	7.16 SCC	91
6.16 Delaunay Triangulation	68	7.17 Stoer-Wagner	92
6.17 Diameter and Width	70		
6.18 Distance Between Point and Circle	71		
6.19 Distance Between Point and Line	71		
6.20 Example of Geometry	71		
6.21 Half Plane Intersection	72		
6.22 Incircle	73		
6.23 Intersection of Two Circles	73		
6.24 Intersection Line and Circle	73		
6.25 Intersection Polygon and Circle	73		
6.26 Intersection Segment and Circle	74		
6.27 Line Intersection	74		
6.28 Minkowski Sum	74		
6.29 Point in Circle	74		
6.30 Point in Convex Hull	74		
6.31 Point in Line	75		
6.32 Point in Perimeter	75		
6.33 Point in Polygon	75		
6.34 Point in Segment	75		
6.35 Points Tangency	75		
6.36 Projection Point Circle	75		
6.37 Segment Intersection	75		
6.38 Smallest Enclosing Circle	76		
6.39 Smallest Enclosing Rectangle	76		
6.40 Vantage Point Tree	77		
7 Graphs	78		
7.1 2Sat	78	9.1 BinPow	97
7.2 Articulation Points	79	9.2 Combination Rank	97
7.3 Bellman-Ford	79	9.3 Chinese Remainder Theorem	97
7.4 Bipartite Checker	80	9.4 Diophantine	97
7.5 Bipartite Maximum Matching	81	9.5 Discrete Logarithm	98
7.6 Bipartite Minimum Maximum Matching ²	81	9.6 Divisors	99
7.7 Block Cut Tree	83	9.7 Euler Totient (Phi)	100
7.8 Blossom	84	9.8 Fibonacci	100
7.9 Bridges	86	9.9 Matrix Exponentiation	100
7.10 Bridges Online	87	9.10 Miller Rabin Deterministic	101
7.11 Dijkstra	89	9.11 Mobius	101
7.12 Eulerian Path	89	9.12 Permutation Rank	102
7.13 Floyd-Warshall	90	9.13 Prefix Sum Phi	102
7.14 Kruskal	90	9.14 Sieve	103
7.15 Marriage	91	9.15 Identities	103
		9.16 Burnside's Lemma	103
		9.17 Recursion	104
		9.18 Theorems	104
		9.19 Sums	104
		9.20 Catalan numbers	104
		9.21 Cayley's formula	104
		9.22 Geometric series	104
		9.23 Estimates For Divisors	104
		9.24 Sum of divisors	104
		9.25 Pythagorean Triplets	105
		9.26 Derangements	105
		10 Game Theory	105
		10.1 Sprague-Grundy theorem	105
		11 Fórmulas y notas	105
		11.1 Números de Stirling del primer tipo	105
		11.2 Números de Stirling del segundo tipo	105
		11.3 Números de Euler	105

11.4 Números de Catalan	105	13 Polynomials	124
11.5 Números de Bell	106	13.1 Berlekamp Massey	124
11.6 Números de Bernoulli	106	13.2 Binary Exp FFT	125
11.7 Fórmula de Faulhaber	106	13.3 FFT	126
11.8 Función Beta	106	13.4 NTT	127
11.9 Función zeta de Riemann	106	13.5 Roots NTT	128
11.10 Funciones generadoras	106		
11.11 Números armónicos	106		
11.12 Aproximación de Stirling	106	14 Strings	128
11.13 Ternas pitagóricas	107	14.1 Hashed String	128
11.14 Árbol de Stern-Brocot	107	14.2 KMP	129
11.15 Combinatoria	107	14.3 Least Rotation String	129
11.16 Grafos	107	14.4 Manacher	130
11.17 Teoría de números	108	14.5 Suffix Array	131
11.18 Primos	109	14.6 Suffix Automaton	132
11.19 Números primos de Mersenne	109	14.7 Trie Ahocorasick	135
11.20 Números primos de Fermat	109	14.8 Z Function	136
12 More Topics	109		
12.1 2D Prefix Sum	109	15 Trees	136
12.2 CDQ Divide and Conquer	109	15.1 Centroid Decomposition	136
12.3 Count Segments Around Index	110	15.2 Heavy Light Decomposition	137
12.4 Custom Comparators	111	15.3 Lowest Common Ancestor (LCA)	139
12.5 Day of the Week	111	15.4 Tree Diameter	140
12.6 Directed MST	111		
12.7 GCD Convolution	112		
12.8 int128	113		
12.9 Iterating Over All Subsets	114		
12.10 LCM Convolution	114		
12.11 Manhattan MST	114		
12.12 Max Manhattan Distance	115		
12.13 Mo	115		
12.14 MOD INT	116		
12.15 Next Permutation	117		
12.16 Next and Previous Smaller/Greater Element	117		
12.17 Parallel Binary Search	117		
12.18 Random Number Generators	118		
12.19 setprecision	118		
12.20 Ternary Search	118		
12.21 Ternary Search Int	118		
12.22 XOR Convolution	119		
12.23 XOR Basis	119		
12.24 XOR Basis Online	120		

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]j[f2,l2]
next/prev_permutation	f,l	deja en [f,l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f,l), hace un heap de [f,l)
is_heap	f,l	bool es [f,l) un heap
accumulate	f,l,i,[op]	$T = \sum$ /oper de [f,l)
inner_product	f1, l1, f2, i	$T = i + [f1, l1] . [f2, \dots)$
partial_sum	f, l, r, [op]	$r+i = \sum$ /oper de [f,f+i] $\forall i \in [f,l)$
_builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
_builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
_builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
_builtin_popcount	unsigned int	Cant. de 1's en x.
_builtin_parity	unsigned int	1 si x es par, 0 si es impar.
_builtin_XXXXXXll	unsigned ll	= pero para long long's.

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

2 Compile

2.1 Compile

```
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 1000000000000000000
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

```

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

```

```

11 const int MAXN = 10000;
12 int bit[MAXN];
13
14 // Add 'val' to index 'x'
15 void update(int x, int val) {
16     for (; x < MAXN; x += x & -x) {
17         bit[x] += val;
18     }
19 }
20
21 // Get prefix sum from 1 to 'x'
22 int get(int x) {
23     int ans = 0;
24     for (; x > 0; x -= x & -x) {
25         ans += bit[x];
26     }
27     return ans;
28 }
```

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

```

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

```

```

11 const int K = 2; // Each node has 2 branches (bit 0 or 1)
12 const int MAX_BIT = 30; // Max bit position (for 32-bit integers)
13
14 struct Vertex {
15     int next[K]; // next[0] = child for bit 0, next[1] = child for bit 1
16
17     Vertex() {
18         fill(begin(next), end(next), -1); // -1 means no child
19     }
20 }
```

```
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 CHMIN/CHMAX Rangesum GCD Queries

```

1 // Segment tree for chmin, chmax, add and query sum.
2 // Complexity is O (n (log n) ^ 2)
3
4 // Range chmin (l, r, v) -> a[l], ..., a[r] = min(a[i], v) call update
5 // with 1
6 // Range chmax (l, r, v) -> a[l], ..., a[r] = max(a[i], v) call update
7 // with 2
8 // Range set (l, r, v) -> a[l], ..., a[r] = v call update with 3 (this
9 // calls chmax and chmin on the interval)
10 // Range add (l, r, v) -> a[l], ..., a[r] += v call update with 4
11 // You can obtain sum, max, min, gcd in segment
12
13 typedef long long ll;
14
15 const ll inf = 1e17;
16
17 ll gcd(ll a, ll b){
18     a = abs(a);
19     b = abs(b);
20     if(a == 0) return b;
21     return gcd(b % a, a);
22 }
23
24 struct sum_t {
25
26     ll sum;
27     ll max;
28     ll min;
29     ll gcd;
30
31     sum_t() : sum(0), max(0), min(0), gcd(1) {}
32
33     void update(ll val) {
34         sum += val;
35         max = max(max, val);
36         min = min(min, val);
37         gcd = gcd(gcd, val);
38     }
39
40     ll query_max() { return max; }
41     ll query_min() { return min; }
42     ll query_gcd() { return gcd; }
43
44     void print() {
45         cout << "sum: " << sum << endl;
46         cout << "max: " << max << endl;
47         cout << "min: " << min << endl;
48         cout << "gcd: " << gcd << endl;
49     }
50
51 };

```

```

60     res.mn = other.mn;
61     res.cnt_mn = other.cnt_mn;
62     res.mn2 = min(mn, other.mn2);
63 }
64
65 if(mx2 != -inf && mx2 != mn && other.mx2 != -inf && other.mx2 !=
66     other.mn){
67     res.gcd_no_max_no_min = gcd(res.gcd_no_max_no_min, mx2 - other.mx2
68         );
69 }
70
71 ll any = -1;
72
73 if(mx2 != -inf && mx2 != mn) any = mx2;
74 if(other.mx2 != -inf && other.mx2 != other.mn) any = other.mx2;
75
76 vector<ll> diff_max_min = {mx, mn, other.mx, other.mn};
77 for(ll val: diff_max_min){
78     if(val != res.mn && val != res.mx){
79         if(any != -1) res.gcd_no_max_no_min = gcd(res.gcd_no_max_no_min,
80             val - any);
81         else any = val;
82     }
83 }
84
85     return res;
86 }
87
88 template <typename num_t>
89 struct segtree {
90     int n;
91     vector<num_t> tree, lazy;
92
93     void init(int s, const vector<ll> &A) {
94         n = s;
95         tree.assign(4 * n, num_t());
96         lazy.assign(4 * n, num_t());
97         init(0, 0, n - 1, A);
98     }
99
100    num_t init(int i, int l, int r, const vector<ll> &A) {
101        if (l == r) return tree[i] = num_t(A[l]);
102
103        int mid = (l + r) / 2;
104        num_t left = init(2 * i + 1, l, mid, A);
105        num_t right = init(2 * i + 2, mid + 1, r, A);
106        return tree[i] = left.op(right);
107    }
108
109    // update 3 is setting values to v
110    void update(int l, int r, ll v, int op) {
111        if (l > r) return;
112        if(op == 1){
113            update_chmin(0, 0, n - 1, l, r, v);
114        }
115        if(op == 2){
116            update_chmax(0, 0, n - 1, l, r, v);
117        }
118        if(op == 3){
119            update_chmin(0, 0, n - 1, l, r, v);
120            update_chmax(0, 0, n - 1, l, r, v);
121        }
122        if(op == 4){
123            update_add(0, 0, n - 1, l, r, v);
124        }
125    }
126
127    num_t update_chmin(int i, int tl, int tr, int ql, int qr, ll v) {
128        eval_lazy(i, tl, tr);
129
130        if (tr < ql || qr < tl || tree[i].mx <= v) return tree[i];
131        if (ql <= tl && tr <= qr && tree[i].mx2 < v) {
132            push_min(i, v);
133            eval_lazy(i, tl, tr);
134            return tree[i];
135        }
136
137        int mid = (tl + tr) / 2;
138        num_t a = update_chmin(2 * i + 1, tl, mid, ql, qr, v);
139        num_t b = update_chmin(2 * i + 2, mid + 1, tr, ql, qr, v);
140        return tree[i] = a.op(b);
141
142    num_t update_chmax(int i, int tl, int tr, int ql, int qr, ll v) {
143        eval_lazy(i, tl, tr);

```

```

143
144     if (tr < ql || qr < tl || tree[i].mn >= v) return tree[i];
145     if (ql <= tl && tr <= qr && tree[i].mn2 > v) {
146         push_max(i, v);
147         eval_lazy(i, tl, tr);
148         return tree[i];
149     }
150
151     int mid = (tl + tr) / 2;
152     num_t a = update_chmax(2 * i + 1, tl, mid, ql, qr, v);
153     num_t b = update_chmax(2 * i + 2, mid + 1, tr, ql, qr, v);
154     return tree[i] = a.op(b);
155 }
156
157 num_t update_add(int i, int tl, int tr, int ql, int qr, ll v) {
158     eval_lazy(i, tl, tr);
159
160     if (tr < ql || qr < tl) return tree[i];
161     if (ql <= tl && tr <= qr) {
162         push_add(i, tl, tr, v);
163         eval_lazy(i, tl, tr);
164         return tree[i];
165     }
166
167     int mid = (tl + tr) / 2;
168     num_t a = update_add(2 * i + 1, tl, mid, ql, qr, v);
169     num_t b = update_add(2 * i + 2, mid + 1, tr, ql, qr, v);
170     return tree[i] = a.op(b);
171 }
172
173 num_t query(int l, int r){
174     if (l > r) return num_t();
175     return query(0, 0, n - 1, l , r);
176 }
177
178 num_t query(int i, int tl, int tr, int ql, int qr) {
179     eval_lazy(i, tl, tr);
180
181     if (ql <= tl && tr <= qr) return tree[i];
182     if (tr < ql || qr < tl) return num_t();
183
184     int mid = (tl + tr) / 2;
185     num_t a = query(2 * i + 1, tl, mid, ql, qr);
186
187     num_t b = query(2 * i + 2, mid + 1, tr, ql, qr);
188     return a.op(b);
189 }
190
191 ll query_gcd(int l, int r){
192     if(l > r) return 0;
193     return query_gcd(0, 0, n - 1, l, r);
194 }
195
196 ll query_gcd(int i, int tl, int tr, int ql, int qr){
197     eval_lazy(i, tl, tr);
198
199     if (ql <= tl && tr <= qr){
200         ll res = gcd(tree[i].mx, tree[i].gcd_no_max_no_min);
201         if(tree[i].mn2 != inf) res = gcd(res, tree[i].mn2 - tree[i].mn);
202         if(tree[i].mx2 != -inf) res = gcd(res, tree[i].mx - tree[i].mx2);
203         return res;
204     }
205     if (tr < ql || qr < tl) return 0;
206
207     int mid = (tl + tr) / 2;
208     ll a = query_gcd(2 * i + 1, tl, mid, ql, qr);
209     ll b = query_gcd(2 * i + 2, mid + 1, tr, ql, qr);
210     return gcd(a, b);
211 }
212
213 void eval_lazy(int i, int l, int r) {
214     if (l != r) {
215         int mid = (l + r) / 2;
216
217         push_add(2 * i + 1, l, mid, lazy[i].sum);
218         push_add(2 * i + 2, mid + 1, r, lazy[i].sum);
219
220         push_min(2 * i + 1, tree[i].mx);
221         push_min(2 * i + 2, tree[i].mx);
222
223         push_max(2 * i + 1, tree[i].mn);
224         push_max(2 * i + 2, tree[i].mn);
225
226         lazy[i] = num_t();
227     }
228 }
```

```

229 void push_add(int i, int l, int r, ll v){
230     if(v == 0) return;
231     tree[i].mx += v;
232     tree[i].sum += v * (r - l + 1);
233     tree[i].mx2 += (tree[i].mx2 == -inf ? 0 : v);
234     tree[i].mn += v;
235     tree[i].mn2 += (tree[i].mn2 == inf ? 0 : v);
236     lazy[i].sum += v;
237 }
238
239 void push_min(int i, ll v){
240     if (tree[i].mx <= v) return;
241     if(tree[i].mn == tree[i].mx){
242         tree[i].mn = v;
243     }
244     else if(tree[i].mn2 == tree[i].mx){
245         tree[i].mn2 = v;
246     }
247     tree[i].sum -= tree[i].mx * tree[i].cnt_mx;
248     tree[i].mx = v;
249     tree[i].sum += tree[i].mx * tree[i].cnt_mx;
250 }
251
252 void push_max(int i, ll v){
253     if (tree[i].mn >= v) return;
254     if(tree[i].mx == tree[i].mn){
255         tree[i].mx = v;
256     }
257     else if(tree[i].mx2 == tree[i].mn){
258         tree[i].mx2 = v;
259     }
260     tree[i].sum -= tree[i].mn * tree[i].cnt_mn;
261     tree[i].mn = v;
262     tree[i].sum += tree[i].mn * tree[i].cnt_mn;
263 }
264 };
265
266 int main(){
267     ios_base::sync_with_stdio(false); cin.tie(NULL);
268     int n; cin >> n;
269     vector<ll> A(n);
270     for(int i = 0; i < n; i++) cin >> A[i];
271     segtree<sum_t> st;

```

```

272     st.init(n, A);
273     int q; cin >> q;
274     while(q--){
275         int t, l, r; cin >> t >> l >> r;
276         l--; r--;
277         if(t < 5){
278             ll x; cin >> x;
279             st.update(l, r, x, t);
280         }
281         else{
282             if(t == 8){
283                 cout << st.query_gcd(l, r) << endl;
284             }
285             else{
286                 auto ans = st.query(l, r);
287                 if(t == 5) cout << ans.sum << endl;
288                 else if (t == 6) cout << ans.mn << endl;
289                 else if(t == 7) cout << ans.mx << endl;
290             }
291         }
292     }
293 }

```

3.5 Disjoint Sparse Table

```

1 // queries are given weird because it is a lot of them but it is segment
   multiplication
2 #include<bits/stdc++.h>
3 using namespace std;
4
5 #define pii pair<int,int>
6 #define pll pair<ll,ll>
7 #define eb emplace_back
8 #define ll long long
9 #define nl '\n'
10 #define deb(x) cerr<<"#x" = "<<x<<nl"
11 #define in() ( { int a ; scanf("%d",&a); a; } )
12
13 const int N = 3e5 + 9;
14 int mod = 1e9 + 7;
15
16 struct DST {
17     vector<vector<int>> left, right;

```

```

18 int k, n;
19 DST(vector<int> & a) {
20     n = (int)a.size();
21     k = log2(n) + 2;
22     left.assign(k + 1, vector<int>(n));
23     right.assign(k + 1, vector<int>(n));
24     for(int j = 0; (1 << j) <= n; ++j) {
25         int mask = (1 << j) - 1;
26         int nw = 1; //neutral
27         for(int i = 0; i < n; ++i) {
28             nw = 1LL * nw * a[i] % mod; //prefix value
29             left[j][i] = nw;
30             if((i & mask) == mask) nw = 1; //neutral
31         }
32         nw = 1; //neutral
33         for(int i = n - 1; i >= 0; --i) {
34             nw = 1LL * nw * a[i] % mod; //prefix value
35             right[j][i] = nw;
36             if((i & mask) == 0) nw = 1; //neutral
37         }
38     }
39 }
40 int query(int l, int r) {
41     if(l == r) return left[0][l];
42     int i = 31 - __builtin_clz(l ^ r);
43     int uno = left[i][r];
44     int dos = right[i][l];
45     return 1LL * uno * dos % mod;
46 }
47 };
48
49 int32_t main() {
50     int tc = in();
51     while(tc--) {
52         int n = in(), p = in(), q = in();
53         mod = p;
54         vector<int> a(n);
55         for(int i = 0; i < n; i++) a[i] = in();
56         DST t(a);
57         vector<int> b((q >> 6) + 2);
58         for(int i = 0; i < (int)b.size(); i++) b[i] = in();
59         int x = 0, l = 1, r = 1;
60         for(int i = 0; i < q; i++) {
61

```

```

61             if(i % 64 == 0) {
62                 l = (b[i / 64] + x) % n;
63                 r = (b[(i / 64) + 1] + x) % n;
64             } else {
65                 l = (l + x) % n;
66                 r = (r + x) % n;
67             }
68             if(l > r) swap(l, r);
69             int ans = t.query(l, r);
70             x = ans;
71             x++;
72             x %= mod;
73         }
74         printf("%d\n", x);
75     }
76     return 0;
77 }

```

3.6 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
16     violation
17 - is_bipartite(v): returns whether component containing v is
18     bipartite
19 */
20 const int MAXN = 100005; // Set according to constraints
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 // Create a new set for node v
25 void make_set(int v) {
26     parent[v] = {v, 0};    // Self-rooted, even parity to self
27     rank[v] = 0;
28     bipartite[v] = true;
29 }
30

31 // Find the root of v and track parity along the path (0 = even, 1 = odd
   )
32 pair<int, int> find_set(int v) {
33     if (v != parent[v].first) {
34         auto [par, parity] = parent[v];
35         auto root = find_set(par);
36         parent[v] = {root.first, parity ^ root.second}; // Path compression
                           with parity update
37     }
38     return parent[v];
39 }
40

41 // Adds an edge between a and b, merges components and checks for odd
   cycles
42 void add_edge(int a, int b) {
43     auto [ra, pa] = find_set(a); // ra = root of a, pa = parity from root
                           to a
44     auto [rb, pb] = find_set(b); // rb = root of b, pb = parity from root
                           to b
45

46     if (ra == rb) {
47         // Same component: edge (a, b) adds a cycle -> check parity
48         if ((pa ^ pb) == 0) {
49             bipartite[ra] = false; // Found odd-length cycle
50         }
51     } else {
52         // Merge smaller rank under larger
53         if (rank[ra] < rank[rb]) swap(ra, rb), swap(pa, pb);
54
55         // Make rb child of ra; update parity of root
56         parent[rb] = {ra, pa ^ pb ^ 1};
57

```

```

58
59     bipartite[ra] &= bipartite[rb]; // Component is only bipartite if
                           both were
60     if (rank[ra] == rank[rb]) rank[ra]++;
61 }
62 }

63 // Check if the component containing v is bipartite
64 bool is_bipartite(int v) {
65     return bipartite[find_set(v).first];
66 }
67

```

3.7 Disjoint Set Union

```

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

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

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

```

```

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
47     // negative)
48     st.push_back({x, e[x]});
49     st.push_back({y, e[y]});
50     e[x] += e[y];
51     e[y] = x;
52     return true;
53 }
54
55 // Return current rollback timestamp
56 int time() {
57     return (int)st.size();
58 }
59
60 // Roll back to previous state at time t
61 void rollback(int t) {
62     for (int i = time(); i-- > t;) {
63         e[st[i].first] = st[i].second;
64     }
65     st.resize(t);
66 }

```

3.8 Disjoint Set Union Queue Like Rollback

```

1 #include <bits/stdc++.h>
2 #define finish(x) return cout << x << endl, 0
3
4 typedef long long ll;
5 typedef long double ldb;
6 const int md = 1e9 + 7;
7 const ll inf = 2e18;
8 const int OO = 1;
9 const int OOO = 1;
10 using namespace std;
11
12 // used in the original data structure to represent a change in
13 // memory cell
14 struct change {
15     int t, value;
16     int *cell;
17     change() {}
18     change(int _time, int _value, int *_cell) {
19         t = _time;
20         value = _value;
21         cell = _cell;
22     }
23     void undo() {
24         *cell = value;
25     }
26 };
27
28 // dsu with normal undo, allows to query bipartite-ness.
29 struct dsu {
30     int n;
31     vector<int> p, r;
32     vector<int> up;
33     int nxtupd;
34     vector<change> stack;
35     int first_violate;
36     dsu() {}
37     dsu(int sz) {
38         n = sz;
39         p.resize(n);
40         r.resize(n, 0);
41         up.resize(n, 0);
42         nxtupd = 0;
43     }
44     void add(int u, int v) {
45         if (find(u) != find(v)) {
46             p[find(v)] = find(u);
47             r[find(u)] += r[find(v)];
48             up[find(u)] = up[find(v)] + 1;
49         }
50     }
51     int find(int u) {
52         if (p[u] == u) return u;
53         return p[u] = find(p[u]);
54     }
55     void update(int u, int v) {
56         if (find(u) == find(v)) return;
57         add(u, v);
58         if (up[find(u)] > md) {
59             cout << "Bipartite" << endl;
60             exit(0);
61         }
62     }
63     void rollback(int t) {
64         if (t < 0) return;
65         if (t >= stack.size()) return;
66         auto &c = stack[t];
67         if (c.t >= first_violate) {
68             cout << "Violate" << endl;
69             exit(0);
70         }
71         if (c.t <= first_violate) {
72             if (c.t <= t) {
73                 *c.cell = c.value;
74             } else {
75                 if (c.t <= first_violate) {
76                     cout << "Violate" << endl;
77                     exit(0);
78                 }
79                 if (c.t <= t) {
80                     *c.cell = c.value;
81                 } else {
82                     cout << "Violate" << endl;
83                     exit(0);
84                 }
85             }
86         }
87     }
88     void print() {
89         for (int i = 0; i < n; ++i) {
90             cout << "p[" << i << "] = " << find(i) << ", ";
91             cout << "r[" << i << "] = " << r[i] << ", ";
92             cout << "up[" << i << "] = " << up[i] << endl;
93         }
94     }
95 };

```

```

41     first_violate = -1;
42     for (int i = 0; i < n; i++) p[i] = i;
43   }
44   void upd(int *cell, int value) {
45     stack.push_back(change(nxtupd, *cell, cell));
46     *cell = value;
47   }
48   pair<int, int> find(int x) {
49     if (x == p[x]) return{ x, 0 };
50     pair<int, int> tmp = find(p[x]);
51     tmp.second ^= up[x];
52     return tmp;
53   }
54   bool is_bipartite() {
55     return first_violate == -1; // no violation
56   }
57   void unite(int x, int y) {
58     nxtupd++;
59     int col = 1;
60     pair<int, int> tmp = find(x);
61     x = tmp.first, col ^= tmp.second;
62     tmp = find(y);
63     y = tmp.first, col ^= tmp.second;
64     if (x == y) {
65       if (col == 1 && first_violate == -1) first_violate = nxtupd;
66       upd(&p[0], p[0]); // mark this update's existence.
67       return;
68     }
69     if (r[x] < r[y]) {
70       upd(&p[x], y);
71       upd(&up[x], col);
72     }
73     else {
74       upd(&p[y], x);
75       upd(&up[y], col);
76       if (r[x] == r[y]) {
77         upd(&r[x], r[x] + 1);
78       }
79     }
80     return;
81   }
82   void undo() {
83     if (!stack.size()) return;
84     int t = stack.back().t;
85     if (first_violate == t) first_violate = -1;
86     while (stack.size() && stack.back().t == t) {
87       stack.back().undo();
88       stack.pop_back();
89     }
90   }
91 };
92
93 // I'd want this 'struct update' to be more generic, and support
94 // maintaining different kinds of updates...
95 // but for now I'm not sure how. So this is a specific impl for this
96 // problem ('update' only relates to 'unite').
97 struct update {
98   char type; // 'A' or 'B'
99   int x, y;
100  update() {}
101  update(int xx, int yy) {
102    x = xx;
103    y = yy;
104    type = 'B';
105  }
106
107 struct dsuqueue {
108   dsu D;
109   vector<update> S;
110   int bottom; // bottom of the stack: S[0..bottom-1] is entirely B's
111   .
112   dsuqueue() {}
113   dsuqueue(int sz) {
114     D = dsu(sz);
115     bottom = 0;
116   }
117   // utility:
118   void advance_bottom() {
119     while (bottom < S.size() && S[bottom].type == 'B') bottom++;
120   }
121   void fix() {
122     if (!S.size() || S.back().type == 'A') return;
123     advance_bottom();
124     vector<update> saveB, saveA;
125     saveB.push_back(S.back());

```

```

124     S.pop_back(), D.undo();
125     while (saveA.size() != saveB.size() && S.size() > bottom) {
126         if (S.back().type == 'A')
127             saveA.push_back(S.back());
128         else
129             saveB.push_back(S.back());
130         S.pop_back(), D.undo();
131     }
132     // reverse saveA and saveB so their relative order is maintained
133     reverse(saveA.begin(), saveA.end());
134     reverse(saveB.begin(), saveB.end());
135     for (const update &u : saveB) {
136         S.push_back(u);
137         D.unite(u.x, u.y);
138     }
139     for (const update &u : saveA) {
140         S.push_back(u);
141         D.unite(u.x, u.y);
142     }
143     advance_bottom();
144 }
145 void reverse_updates() {
146     for (int i = 0; i < S.size(); i++)
147         D.undo();
148     for (int i = (int)S.size() - 1; i >= 0; i--) {
149         D.unite(S[i].x, S[i].y);
150         S[i].type = 'A';
151     }
152     reverse(S.begin(), S.end());
153     bottom = 0;
154 }
155 void undo() {
156     advance_bottom();
157     if (bottom == S.size()) {
158         // no more A's, let's reverse and begin again.
159         reverse_updates();
160     }
161     fix();
162     D.undo();
163     S.pop_back();
164 }
165 // begin copying all functions from the original dsu.
166
167     bool is_bipartite() {
168         return D.is_bipartite();
169     }
170     void unite(int x, int y) {
171         D.unite(x, y);
172         S.push_back(update(x, y));
173     }
174 };
175
176 int n, m, q;
177 vector<pair<int, int>> e;
178 vector<int> R; // for each i, R[i] would be the minimum such that
179     // query (i, R[i]) is answered with NO (the graph is bipartite)
180
181 int main() {
182     ios::sync_with_stdio(0), cin.tie(0);
183     cin >> n >> m >> q;
184     dsuqueue D(n);
185     e.resize(m);
186     for (auto &i : e) {
187         cin >> i.first >> i.second;
188         --i.first, --i.second;
189     }
190
191     R.resize(m);
192     for (int i = 0; i < m; i++)
193         D.unite(e[i].first, e[i].second);
194     int nxt = 0;
195     for (int i = 0; i < m; i++) {
196         while (!D.is_bipartite() && nxt < m) {
197             D.undo();
198             nxt++;
199         }
200         if (D.is_bipartite())
201             R[i] = nxt - 1;
202         else
203             R[i] = md;
204
205         D.unite(e[i].first, e[i].second);
206     }
207
208     while (q--) {
209         int l, r;

```

```

209     cin >> l >> r;
210     --l, --r;
211     if (R[l] <= r) cout << "NO\n";
212     else cout << "YES\n";
213 }
214 }
```

3.9 Dynamic Conectivity

```

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

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

Components:
- DSU with rollback (stores a stack of previous states)
- Segment tree over time to store active edge intervals
*/
// Rollbackable Disjoint Set Union (Union-Find)
struct DSU {
    vector<int> e; // e[x] < 0 means x is a root, and size is -e[x]
    vector<pair<int, int>> st; // Stack for rollback (stores changed
        values)
    int cnt; // Current number of connected components
    DSU() {}
    DSU(int N) : e(N, -1), cnt(N) {}

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

    // Check if x and y are connected
    bool connected(int a, int b) {

```

```

33         return get(a) == get(b);
34     }
35
36     // Size of component containing x
37     int size(int x) {
38         return -e[get(x)];
39     }
40
41     // 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);
85         t.resize(4 * T + 4);
86     }
87
88     // Internal segment tree add function
89     void add(int v, int l, int r, int ul, int ur, query& q) {
90         if (ul > ur) return;
91         if (l == ul && r == ur) {
92             t[v].push_back(q);
93             return;
94         }
95         int mid = (l + r) / 2;
96         add(2 * v, l, mid, ul, min(ur, mid), q);
97         add(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
98     }
99
100    // Public wrapper: add a query in interval [l, r]
101    void add_query(query q, int l, int r) {
102        add(1, 0, T - 1, l, r, q);
103    }
104
105    // Traverse the segment tree and simulate unions with rollback
106    void dfs(int v, int l, int r, vector<int>& ans) {
107        // Apply all union operations in this segment node
108        for (query& q : t[v]) {
109            q.united = dsu.unite(q.v, q.u);
110        }
111
112        if (l == r) {
113            ans[l] = dsu.cnt; // Save answer for time l
114        } else {
115            int mid = (l + r) / 2;
116            dfs(2 * v, l, mid, ans);
117            dfs(2 * v + 1, mid + 1, r, ans);
118        }
119    }
120
121    // Rollback all operations applied in this node
122    for (query& q : t[v]) {
123        if (q.united)
124            dsu.rollback();
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
136        // Parse all k operations
137        for (int i = 0; i < k; i++) {
138            char c;
139            cin >> c;
140            if (c == '?') {
141                qmarks.push_back(i); // Save query index
142                continue;
143            }
144            int u, v;
145            cin >> u >> v;
146            u--; v--;
147            if (u > v) swap(u, v); // Normalize edge direction
148
149            if (c == '+') {
150                mp[{u, v}] = i; // Mark time edge is added
151            } else {
152                // Edge removed: store active interval
153                st.add_query(query(u, v), mp[{u, v}], i);
154                mp[{u, v}] = -1;
155            }
156
157            // Any edge still active is added until end of timeline
158            for (auto [edge, start] : mp) {
159                if (start != -1) {
160                    st.add_query(query(edge.first, edge.second), start, k - 1);
161                }
162            }
163        }
164    }

```

```

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 }
```

3.10 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);
```

3.11 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  -Can be adapted for rectangular grids by using n, m separately
12 */
13
14 struct Fenwick2D {
15     vector<vector<ll>> b; // 2D BIT array
16     int n;                // Grid size (1-based)
17
18     Fenwick2D(int _n) : b(_n + 5, vector<ll>(_n + 5, 0)), n(_n) {}
19
20     // Add 'val' to cell (x, y)
21     void update(int x, int y, int val) {
22         for (; x <= n; x += (x & -x)) {
23             for (int j = y; j <= n; j += (j & -j)) {
24                 b[x][j] += val;
25             }
26         }
27     }
28
29     // Get sum of rectangle [(1,1) to (x,y)]
30     ll get(int x, int y) {
31         ll ans = 0;
32         for (; x > 0; x -= x & -x) {
33             for (int j = y; j > 0; j -= j & -j) {
34                 ans += b[x][j];
35             }
36         }
37         return ans;
38     }
39
40     // Get sum over subrectangle [(x1,y1) to (x2,y2)]
41     ll get1(int x1, int y1, int x2, int y2) {
42         return get(x2, y2)-get(x1-1, y2)-get(x2, y1-1)+get(x1-1, y1-1);
43     }
44 };
45 // Usage example:
46 Fenwick2D fw(n);
47 fw.update(3, 4, 5);           // add 5 to (3, 4)
48 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)]
```

3.12 Li Chao Tree

```

1  // min of a * x^3 + b * x ^2 + c * x + d
2
3 #include <bits/stdc++.h>
4 using namespace std;
5 typedef long long ll;
6
7 const int N = (int)1e5 + 5;
8 const int C = (int)1e5 + 5;
9 const int T = 10;
10 const int M = 350; // sqrt(10^5)
11 const ll inf = (ll)1e19;
12
13 namespace segtree {
14     struct Poly {
15         ll a, b, c, d = inf;
16         ll operator()(ll x) { return a * x * x * x + b * x * x + c * x + d; }
17     } a[T][C * 4];
18     int t = -1;
19
20     void insert(int l, int r, Poly poly, int o=0) {
21         if(l + 1 == r) {
22             if(poly(l) < a[t][o](l)) a[t][o] = poly;
23             return;
24         }
25         int mid = (l + r) >> 1, lson = o * 2 + 1, rson = o * 2 + 2;
26         bool b1 = poly(mid) < a[t][o](mid), b2 = poly(l) < a[t][o](l);
27         if(b1) swap(poly, a[t][o]);
28         if(b1 != b2) insert(l, mid, poly, lson);
29         else insert(mid, r, poly, rson);
30     }
31     ll query(int l, int r, int x, int o=0) {
32         if(l + 1 == r) return a[t][o](x);
33         int mid = (l + r) >> 1, lson = o * 2 + 1, rson = o * 2 + 2;
34         if(x < mid) return min(a[t][o](x), query(l, mid, x, lson));
35         else return min(a[t][o](x), query(mid, r, x, rson));
36     }
37 }
38

```

```

39 ll ans[M];
40
41 void init() {
42     int n; cin >> n;
43     fill(ans, ans + M, inf);
44     segtree::t++;
45     while(n--) {
46         int a, b, c, d; cin >> d >> c >> b >> a;
47         segtree::Poly p = {a, b, c, d};
48         segtree::insert(M, C, p);
49         for(int i = 0 ; i < M ; ++i) ans[i] = min(ans[i], p(i));
50     }
51 }
52 void solve() {
53     int q; cin >> q;
54     while(q--) {
55         int x; cin >> x;
56         if(x < M) cout << ans[x] << '\n';
57         else cout << segtree::query(M, C, x) << '\n';
58     }
59 }
60
61 int main() {
62     ios_base::sync_with_stdio(0), cin.tie(0);
63     int t; cin >> t;
64     while(t--) {
65         init();
66         solve();
67     }
68 }
```

3.13 Link Cut Tree

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 const int N = 1e5 + 9;
5
6 struct node {
7     int p = 0, c[2] = {0, 0}, pp = 0;
8     bool flip = 0;
9     int sz = 0, ssz = 0, vsz = 0; // sz -> aux tree size, ssz = subtree
        size in rep tree, vsz = virtual tree size
```

```

10 long long val = 0, sum = 0, lazy = 0, subsum = 0, vsum = 0;
11 node() {}
12 node(int x) {
13     val = x; sum = x;
14     sz = 1; lazy = 0;
15     ssz = 1; vsz = 0;
16     subsum = x; vsum = 0;
17 }
18 };
19 struct LCT {
20     vector<node> t;
21     LCT() {}
22     LCT(int n) : t(n + 1) {}

23 // <independant splay tree code>
24 int dir(int x, int y) { return t[x].c[1] == y; }
25 void set(int x, int d, int y) {
26     if (x) t[x].c[d] = y, pull(x);
27     if (y) t[y].p = x;
28 }
29 void pull(int x) {
30     if (!x) return;
31     int &l = t[x].c[0], &r = t[x].c[1];
32     push(l); push(r);
33     t[x].sum = t[l].sum + t[r].sum + t[x].val;
34     t[x].sz = t[l].sz + t[r].sz + 1;
35     t[x].ssz = t[l].ssz + t[r].ssz + t[x].vsz + 1;
36     t[x].subsum = t[l].subsum + t[r].subsum + t[x].vsum + t[x].val;
37 }
38 void push(int x) {
39     if (!x) return;
40     int &l = t[x].c[0], &r = t[x].c[1];
41     if (t[x].flip) {
42         swap(l, r);
43         if (l) t[l].flip ^= 1;
44         if (r) t[r].flip ^= 1;
45         t[x].flip = 0;
46     }
47     if (t[x].lazy) {
48         t[x].val += t[x].lazy;
49         t[x].sum += t[x].lazy * t[x].sz;
50         t[x].subsum += t[x].lazy * t[x].ssz;
51         t[x].vsum += t[x].lazy * t[x].vsz;
52     }
```

```

53     if (l) t[l].lazy += t[x].lazy;
54     if (r) t[r].lazy += t[x].lazy;
55     t[x].lazy = 0;
56   }
57 }
58 void rotate(int x, int d) {
59   int y = t[x].p, z = t[y].p, w = t[x].c[d];
60   swap(t[x].pp, t[y].pp);
61   set(y, !d, w);
62   set(x, d, y);
63   set(z, dir(z, y), x);
64 }
65 void splay(int x) {
66   for (push(x); t[x].p;) {
67     int y = t[x].p, z = t[y].p;
68     push(z); push(y); push(x);
69     int dx = dir(y, x), dy = dir(z, y);
70     if (!z) rotate(x, !dx);
71     else if (dx == dy) rotate(y, !dx), rotate(x, !dx);
72     else rotate(x, dy), rotate(x, dx);
73   }
74 }
75 // </independant splay tree code>
76
77 // making it a root in the rep. tree
78 void make_root(int u) {
79   access(u);
80   int l = t[u].c[0];
81   t[l].flip ^= 1;
82   swap(t[l].p, t[l].pp);
83   t[u].vsz += t[l].ssz;
84   t[u].vsum += t[l].subsum;
85   set(u, 0, 0);
86 }
87 // make the path from root to u a preferred path
88 // returns last path-parent of a node as it moves up the tree
89 int access(int _u) {
90   int last = _u;
91   for (int v = 0, u = _u; u; u = t[v = u].pp) {
92     splay(u); splay(v);
93     t[u].vsz -= t[v].ssz;
94     t[u].vsum -= t[v].subsum;
95     int r = t[u].c[1];
96     t[u].vsz += t[r].ssz;
97     t[u].vsum += t[r].subsum;
98     t[v].pp = 0;
99     swap(t[r].p, t[r].pp);
100    set(u, 1, v);
101    last = u;
102  }
103  splay(_u);
104  return last;
105 }
106 void link(int u, int v) { // u -> v
107   // assert(!connected(u, v));
108   make_root(v);
109   access(u); splay(u);
110   t[v].pp = u;
111   t[u].vsz += t[v].ssz;
112   t[u].vsum += t[v].subsum;
113 }
114 void cut(int u) { // cut par[u] -> u, u is non root vertex
115   access(u);
116   assert(t[u].c[0] != 0);
117   t[t[u].c[0]].p = 0;
118   t[u].c[0] = 0;
119   pull(u);
120 }
121 // parent of u in the rep. tree
122 int get_parent(int u) {
123   access(u); splay(u); push(u);
124   u = t[u].c[0]; push(u);
125   while (t[u].c[1]) {
126     u = t[u].c[1]; push(u);
127   }
128   splay(u);
129   return u;
130 }
131 // root of the rep. tree containing this node
132 int find_root(int u) {
133   access(u); splay(u); push(u);
134   while (t[u].c[0]) {
135     u = t[u].c[0]; push(u);
136   }
137   splay(u);
138   return u;

```

```

139 }
140 bool connected(int u, int v) {
141     return find_root(u) == find_root(v);
142 }
143 // depth in the rep. tree
144 int depth(int u) {
145     access(u); splay(u);
146     return t[u].sz;
147 }
148 int lca(int u, int v) {
149     // assert(connected(u, v));
150     if (u == v) return u;
151     if (depth(u) > depth(v)) swap(u, v);
152     access(v);
153     return access(u);
154 }
155 int is_root(int u) {
156     return get_parent(u) == 0;
157 }
158 int component_size(int u) {
159     return t[find_root(u)].ssz;
160 }
161 int subtree_size(int u) {
162     int p = get_parent(u);
163     if (p == 0) {
164         return component_size(u);
165     }
166     cut(u);
167     int ans = component_size(u);
168     link(p, u);
169     return ans;
170 }
171 long long component_sum(int u) {
172     return t[find_root(u)].subsum;
173 }
174 long long subtree_sum(int u) {
175     int p = get_parent(u);
176     if (p == 0) {
177         return component_sum(u);
178     }
179     cut(u);
180     long long ans = component_sum(u);
181     link(p, u);
182     return ans;
183 }
184 // sum of the subtree of u when root is specified
185 long long subtree_query(int u, int root) {
186     int cur = find_root(u);
187     make_root(root);
188     long long ans = subtree_sum(u);
189     make_root(cur);
190     return ans;
191 }
192 // path sum
193 long long query(int u, int v) {
194     int cur = find_root(u);
195     make_root(u); access(v);
196     long long ans = t[v].sum;
197     make_root(cur);
198     return ans;
199 }
200 void upd(int u, int x) {
201     access(u); splay(u);
202     t[u].val += x;
203 }
204 // add x to the nodes on the path from u to v
205 void upd(int u, int v, int x) {
206     int cur = find_root(u);
207     make_root(u); access(v);
208     t[v].lazy += x;
209     make_root(cur);
210 }
211 }t[2];
212
213 vector<int> g[N];
214 int par[N];
215 void dfs(int u, int p = 0) {
216     par[u] = p;
217     for (auto v: g[u]) {
218         if (v ^ p) {
219             dfs(v, u);
220         }
221     }
222 }
223 int col[N];
224 int32_t main() {

```

```

225 ios_base::sync_with_stdio(0);
226 cin.tie(0);
227 int n; cin >> n;
228 for (int i = 1; i < n; i++) {
229     int u, v; cin >> u >> v;
230     g[u].push_back(v);
231     g[v].push_back(u);
232 }
233 dfs(1, n + 1);
234 t[0] = LCT(n + 1);
235 t[1] = LCT(n + 1);
236 for (int i = 1; i <= n + 1; i++) {
237     t[0].t[i] = node(i);
238     t[1].t[i] = node(i);
239 }
240 for (int i = 1; i <= n; i++) {
241     t[0].link(par[i], i);
242 }
243 int q; cin >> q;
244 while (q--) {
245     int ty, u; cin >> ty >> u;
246     if (ty == 0) {
247         int z = t[col[u]].find_root(u);
248         int c = t[col[u]].t[z].c[1];
249         cout << t[col[u]].t[c].ssz << '\n';
250     }
251     else {
252         t[col[u]].cut(u);
253         col[u] ^= 1;
254         t[col[u]].link(par[u], u);
255     }
256 }
257 return 0;
258 }
```

3.14 Linked List

```

1 /*
2  * Linked list
3  -----
4  Time Complexity:
5      -insert: O(1)
6  Space Complexity: O(n)
```

```

7
8     -Can be adapted for rectangular grids by using n, m separately
9 */
10 // === Constructors and Destructor ===
11 // list()                  // Default constructor, creates an empty list.
12 // list(n, val)            // Creates a list with 'n' elements, all
13 // initialized to 'val'.
14 // list(first, last)       // Creates a list from another range (e.g.,
15 // another list or array).
16 // list(const list& other) // Copy constructor.
17 // ~list()                 // Destructor, destroys the list.
18 // === Iterators ===
19 // begin()                // Returns an iterator to the first element of
20 // the list.
21 // end()                   // Returns an iterator to the position just
22 // past the last element.
23 // rbegin()                // Returns a reverse iterator to the last
24 // element.
25 // rend()                  // Returns a reverse iterator to the position
26 // just before the first element.
27 // insert(it, val)          // Inserts 'val' before the position of the
28 // iterator 'it'.
29 // erase(it)                // Removes the element at the position of the
30 // iterator 'it'.
31 // erase(first, last)       // Removes a range of elements from 'first' to
32 // 'last'.
33 // push_back(val)           // Adds 'val' to the end of the list.
34 // push_front(val)          // Adds 'val' to the front of the list.
35 // pop_back()                // Removes the last element of the list.
36 // pop_front()               // Removes the first element of the list.
37 // clear()                  // Removes all elements from the list.
38 // === Capacity ===
39 // empty()                  // Returns true if the list is empty, otherwise
40 // false.
41 // size()                   // Returns the number of elements in the list.
42 // max_size()                // Returns the maximum number of elements the
43 // list can hold.
44
45 // === Modifiers ===
46 // insert(it, n, val)        // Inserts 'n' copies of 'val' before the
```

```

    position of 'it'.
39 // insert(it, first, last) // Inserts a range of elements before the
   position of 'it'.
40 // remove(val)           // Removes all occurrences of 'val' from the
   list.
41 // remove_if(pred)       // Removes all elements that satisfy the
   predicate 'pred'.
42 // unique()              // Removes consecutive duplicate elements from
   the list.
43 // merge(other)          // Merges two sorted lists into one sorted
   list.
44 // merge(other, pred)    // Merges two sorted lists into one sorted
   list using custom comparison 'pred'.
45 // splice(it, other)     // Transfers elements from another list 'other'
   ' to the list at position 'it'.
46 // splice(it, other, it2) // Transfers an element from another list '
   other' at position 'it2' to the list at position 'it'.
47 // splice(it, other, first, last) // Transfers a range from another list
   'other' from 'first' to 'last' to the list at position 'it'.
48 // reverse()             // Reverses the order of elements in the list.
49 // sort()                // Sorts the elements in the list in ascending
   order.
50 // sort(comp)            // Sorts the elements in the list using custom
   comparison 'comp'.
51 // swap(other)           // Swaps the content of the list with another
   list 'other'.
52
53 // === Element Access ===
54 // front()               // Returns a reference to the first element in
   the list.
55 // back()                // Returns a reference to the last element in
   the list.
56
57 // === Search ===
58 // find(val)              // Returns an iterator to the first occurrence
   of 'val'.
59 // find_if(pred)          // Returns an iterator to the first element
   satisfying the predicate 'pred'.
60 // count(val)              // Returns the number of occurrences of 'val'
   in the list.
61 // count_if(pred)          // Returns the number of elements satisfying
   the predicate 'pred'.
62 // equal_range(val)        // Returns a pair of iterators representing the

```

```

range of elements equal to 'val'.
63
64 // === Other Functions ===
65 // swap(other)           // Swaps the contents of the list with another
   list 'other'.
66 // emplace(it, args...)  // Constructs and inserts an element at the
   position 'it' using 'args...' (perfect forwarding).
67 // emplace_front(args...) // Constructs and inserts an element at the
   front using 'args...' (perfect forwarding).
68 // emplace_back(args...) // Constructs and inserts an element at the
   back using 'args...' (perfect forwarding).

```

3.15 Merge Sort Tree

```

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

Features:
  - Supports frequency/count queries: "how many values <= x in range [
    l, r]?"
  - Static array (no point updates unless rebuilt)
*/
const int MAXN = 100005;           // size of original array
const int MAXT = 2 * MAXN;         // size of segment tree (2n)

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

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

```

```

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 // Query how many elements <= 'x' in range [l, r]
38 int q(int l, int r, int x) {
39     int res = 0;
40     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
41         if (l & 1) {
42             res += upper_bound(t[l].begin(), t[l].end(), x) - t[l].begin();
43             l++;
44         }
45         if (r & 1) {
46             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()

```

3.16 Minimum Cartesian Tree

```

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

Use cases:
    - RMQ construction

```

```

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';

```

3.17 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})

```

3.18 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 // check if upperbound or lowerbound does what you want
23 // because they give better time.
24
25 // to allow repetitions
26 #define ordered_set tree<int, null_type,less_equal<int>, rb_tree_tag,
27     tree_order_statistics_node_update>
28
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

```

3.19 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 }
```

3.20 Persistent Array

```

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

Features:
    - Supports point updates with full version history
    - Allows querying any version at any index
*/
struct Node {
    int val;
    Node *l, *r;

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

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

int n;
```

```

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    // Initializes version 0 from given array
68    void init_arr(int nn, int *init) {
69        n = nn;
```

```

69 |     for (int i = 0; i < n; i++) a[i] = init[i];
70 |     roots[0] = build();
71 |

```

3.21 Persistent Segment Tree

```

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

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

```

```

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     int main() {
68         ios_base::sync_with_stdio(false); cin.tie(NULL);
69
70         int q;
71         cin >> n >> q;
72         for (int i = 1; i <= n; i++) {
73             cin >> a[i];
74         }
75
76         // Build version 0
77         t[0] = build();
78         sz = 1;

```

```

79
80 while (q--) {
81     int ty;
82     cin >> ty;
83
84     if (ty == 1) {
85         // Point update: create new version from t[k] with a[pos] = x
86         int k, pos, x;
87         cin >> k >> pos >> x;
88         t[k] = update(t[k], pos, x);
89
90     } else if (ty == 2) {
91         // Range query on version k over [l, r]
92         int k, l, r;
93         cin >> k >> l >> r;
94         cout << query(t[k], l, r) << '\n';
95
96     } 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}
103return 0;
104}

```

3.22 Polynomial Queries

```

1 // adds d * (i - l + 1) + a to range [l, r]
2 // d * (i - l + 1) = d*(i-l) + d*i
3 // Sum = A x segment_length + B x sum_of_indices_in_segment
4
5 struct sum_t {
6     ll val;
7     static const ll null_v = 0;
8     sum_t(): val(null_v) {}
9     sum_t(ll v): val(v) {}
10    sum_t op(sum_t& other) {
11        return sum_t(val + other.val);
12    }
13    sum_t lazy_op(sum_t& v, int size) {
14        return val + v.val * size;

```

```

15    }
16    sum_t lazy_op(sum_t& v, ll l, ll r) {
17        return val + v.val * ((l + r) * (r - l + 1)) / 2;
18    }
19};
20
21 template <typename num_t>
22 struct segtree {
23     int n;
24     vector<num_t> tree, lazy, lazy1;
25
26     void init(int s) {
27         n = s;
28         tree.assign(4 * n, num_t(0));
29         lazy.assign(4 * n, num_t(0));
30         lazy1.assign(4 * n, num_t(0));
31         init(0, 0, n - 1);
32     }
33
34     num_t init(int i, int l, int r) {
35         if (l == r) return tree[i] = num_t(0);
36
37         int mid = (l + r) / 2;
38         num_t left = init(2 * i + 1, l, mid);
39         num_t right = init(2 * i + 2, mid + 1, r);
40         return tree[i] = left.op(right);
41     }
42
43     // do -l if you want it to add 0 to l. adds d * (i - l + 1) + a to
44     // range [l, r]
45     void update(int l, int r, ll d, ll a) {
46         if (l > r) return;
47         ll dd = d;
48         d = (l - 1) * d + a;
49         a = dd;
50         update(0, 0, n - 1, l, r, d, a);
51     }
52
53     num_t update(int i, int tl, int tr, int ql, int qr, ll D, ll A) {
54         eval_lazy(i, tl, tr);
55
56         if (tr < ql || qr < tl) return tree[i];
57         if (ql <= tl && tr <= qr) {

```

```

57     lazy[i].val += D;
58     lazy1[i].val += A;
59     eval_lazy(i, tl, tr);
60     return tree[i];
61 }
62
63     int mid = (tl + tr) / 2;
64     num_t a = update(2 * i + 1, tl, mid, ql, qr, D, A);
65     num_t b = update(2 * i + 2, mid + 1, tr, ql, qr, D, A);
66     return tree[i] = a.op(b);
67 }
68
69 num_t query(int l, int r) {
70     if (l > r) return num_t::null_v;
71     return query(0, 0, n - 1, l, r);
72 }
73
74 num_t query(int i, int tl, int tr, int ql, int qr) {
75     eval_lazy(i, tl, tr);
76
77     if (ql <= tl && tr <= qr) return tree[i];
78     if (tr < ql || qr < tl) return num_t::null_v;
79
80     int mid = (tl + tr) / 2;
81     num_t a = query(2 * i + 1, tl, mid, ql, qr);
82     num_t b = query(2 * i + 2, mid + 1, tr, ql, qr);
83     return a.op(b);
84 }
85
86 void eval_lazy(int i, int l, int r) {
87     tree[i] = tree[i].lazy_op(lazy1[i], l, r);
88     tree[i] = tree[i].lazy_op(lazy[i], r - l + 1);
89     if (l != r) {
90         lazy[2 * i + 1].val += lazy[i].val;
91         lazy[2 * i + 2].val += lazy[i].val;
92         lazy1[2 * i + 1].val += lazy1[i].val;
93         lazy1[2 * i + 2].val += lazy1[i].val;
94     }
95     lazy[i] = lazy1[i] = num_t(0);
96 }
97 }
98 int main(){

```

```

100    ios_base::sync_with_stdio(false); cin.tie(NULL);
101    int n, q; cin >> n >> q;
102    segtree<sum_t> st;
103    st.init(n);
104    while(q--){
105        int t; cin >> t;
106        if(t == 1){
107            int l, r, a, d; cin >> l >> r >> a >> d;
108            l--; r--;
109            st.update(l, r, d, a);
110        }
111        else{
112            int l; cin >> l;
113            l--;
114            cout << st.query(l, l).val << endl;
115        }
116    }
117 }

```

3.23 Segment Tree

```

/*
Segment Tree (Iterative, Range Minimum Query)
-----
Indexing: 0-based
Bounds: [0, n-1] inclusive
Time Complexity:
- update(pos, val): O(log n)
- get(l, r): O(log n) -> query min in range [l, r]
Space Complexity: O(2n)
*/
struct SegmentTree {
    vector<ll> a; // segment tree array
    int n; // number of elements in original array
    SegmentTree(int _n) : a(2 * _n, 1e18), n(_n) {}
    // Update position 'pos' to value 'val'
    void update(int pos, ll val) {
        pos += n; // move to leaf
        a[pos] = val; // set value
        for (pos /= 2; pos > 0; pos /= 2) {
            a[pos] = min(a[2 * pos], a[2 * pos + 1]); // update parent
        }
    }
}

```

```

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 };

```

3.24 Segment Tree 2D

```

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

```

const int MAXN = 505;

int n, m; // grid dimensions

int a[MAXN][MAXN]; // input grid

int t[4 * MAXN][4 * MAXN]; // segment tree

// Build the tree along y-axis (internal to each x-interval)

```

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

```

```

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, new_val);
73         update_y(vx, lx, rx, vy * 2 + 1, my + 1, ry, x, new_val);
74     }
}

```

```

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 }

```

3.25 Segment Tree Dynamic

```

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

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

```

```

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

```

3.26 Segment Tree Lazy Types

```

1 | struct max_t {
2 |     ll val;

```

```

3 static const ll null_v = -1LL << 61;
4 max_t(): val(0) {}
5 max_t(ll v): val(v) {}
6 max_t op(max_t& other) {
7     return max_t(max(val, other.val));
8 }
9 max_t lazy_op(max_t& v, int size) {
10    return max_t(val + v.val);
11 }
12 };
13
14 struct min_t {
15     ll val;
16     static const ll null_v = 1LL << 61;
17     min_t(): val(0) {}
18     min_t(ll v): val(v) {}
19     min_t op(min_t& other) {
20         return min_t(min(val, other.val));
21     }
22     min_t lazy_op(min_t& v, int size) {
23         return min_t(val + v.val);
24     }
25 };
26
27 struct sum_t {
28     ll val;
29     static const ll null_v = 0;
30     sum_t(): val(0) {}
31     sum_t(ll v): val(v) {}
32     sum_t op(sum_t& other) {
33         return sum_t(val + other.val);
34     }
35     sum_t lazy_op(sum_t& v, int size) {
36         return sum_t(val + v.val * size);
37     }
38 };

```

3.27 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
18 template <typename num_t>
19 struct segtree {
20     int n;
21     vector<num_t> tree, lazy;
22
23     // Initialize segment tree with array of size s
24     void init(int s, long long* arr) {
25         n = s;
26         tree.assign(4 * n, num_t());
27         lazy.assign(4 * n, num_t());
28         init(0, 0, n - 1, arr);
29     }
30
31     // Build segment tree from array
32     num_t init(int i, int l, int r, long long* arr) {
33         if (l == r) return tree[i] = num_t(arr[l]);
34
35         int mid = (l + r) / 2;
36         num_t left = init(2 * i + 1, l, mid, arr);
37         num_t right = init(2 * i + 2, mid + 1, r, arr);
38         return tree[i] = left.op(right);
39     }
40
41     // Public wrapper: update range [l, r] with value v
42     void update(int l, int r, num_t v) {
43         if (l > r) return;
44         update(0, 0, n - 1, l, r, v);
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     lazy[i] = num_t(); // reset lazy at this node
91 }
92 }
```

3.28 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     }
39 }
```

```

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 }

// Build tree from array a[0..N-1]
59 void build(int p, int l, int r) {
60     tree[p].lzAdd = tree[p].lzSet = 0;
61     if (l == r) {
62         tree[p].val = a[l];
63         return;
64     }
65     int mid = (l + r) >> 1;
66     build(lc, l, mid);
67     build(rc, mid + 1, r);
68     pushup(p);
69 }
70 }

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

80 }

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

```

3.29 Segment Tree Lazy Range Set 2

```
2 using namespace std;
3
4 typedef long long ll;
5
6 struct sum_t {
7     ll val;
8     static const ll null_v = 0;
9     sum_t(): val(null_v) {}
10    sum_t(ll v): val(v) {}
11    sum_t op(sum_t& other) {
12        return sum_t(val + other.val);
13    }
14    sum_t lazy_op(sum_t& v, int size) {
15        return v.val * size;
16    }
17    sum_t lazy_op1(sum_t& v, int size) {
18        return val + v.val * size;
19    }
20};
21
22 template <typename num_t>
23 struct segtree {
24     int n;
25     vector<num_t> tree, lazy, lazy1;
26
27     void init(int s) {
28         n = s;
29         tree.assign(4 * n, num_t(0));
30         lazy.assign(4 * n, num_t(-1));
31         lazy1.assign(4 * n, num_t(0));
32         init(0, 0, n - 1);
33     }
34
35     num_t init(int i, int l, int r) {
36         if (l == r) return tree[i] = num_t(0);
37
38         int mid = (l + r) / 2;
39         num_t left = init(2 * i + 1, l, mid);
40         num_t right = init(2 * i + 2, mid + 1, r);
41         return tree[i] = left.op(right);
42     }
43
44     void update(int l, int r, num_t v) {
45         if (l > r) return;
46         update(0, 0, n - 1, l, r, v);
47     }
48
49     num_t update(int i, int tl, int tr, int ql, int qr, num_t v) {
50         eval_lazy(i, tl, tr);
51
52         if (tr < ql || qr < tl) return tree[i];
53         if (ql <= tl && tr <= qr) {
54             lazy[i].val = v.val;
55             lazy1[i].val = 0;
56             eval_lazy(i, tl, tr);
57             return tree[i];
58         }
59
60         int mid = (tl + tr) / 2;
61         num_t a = update(2 * i + 1, tl, mid, ql, qr, v);
62         num_t b = update(2 * i + 2, mid + 1, tr, ql, qr, v);
63         return tree[i] = a.op(b);
64     }
65
66     void update1(int l, int r, num_t v) {
67         if (l > r) return;
68         update1(0, 0, n - 1, l, r, v);
69     }
70
71     num_t update1(int i, int tl, int tr, int ql, int qr, num_t v) {
72
73         if (tr < ql || qr < tl){
74             eval_lazy(i, tl, tr);
75             return tree[i];
76         }
77         if (ql <= tl && tr <= qr) {
78             lazy1[i].val += v.val;
79             if(lazy[i].val != -1){
80                 lazy[i].val += lazy1[i].val;
81                 lazy1[i].val = 0;
82             }
83             eval_lazy(i, tl, tr);
84             return tree[i];
85         }
86
87         eval_lazy(i, tl, tr);
88     }
89 }
```

```

88     int mid = (tl + tr) / 2;
89     num_t a = update1(2 * i + 1, tl, mid, ql, qr, v);
90     num_t b = update1(2 * i + 2, mid + 1, tr, ql, qr, v);
91     return tree[i] = a.op(b);
92 }
93
94 num_t query(int l, int r) {
95     if (l > r) return num_t::null_v;
96     return query(0, 0, n - 1, l, r);
97 }
98
99 num_t query(int i, int tl, int tr, int ql, int qr) {
100    eval_lazy(i, tl, tr);
101
102    if (ql <= tl && tr <= qr) return tree[i];
103    if (tr < ql || qr < tl) return num_t::null_v;
104
105    int mid = (tl + tr) / 2;
106    num_t a = query(2 * i + 1, tl, mid, ql, qr);
107    num_t b = query(2 * i + 2, mid + 1, tr, ql, qr);
108    return a.op(b);
109 }
110
111 void eval_lazy(int i, int l, int r) {
112     if(lazy[i].val != -1){
113         tree[i] = tree[i].lazy_op(lazy[i], r - l + 1);
114         if (l != r) {
115             lazy[2 * i + 1].val = lazy[i].val;
116             lazy[2 * i + 2].val = lazy[i].val;
117             lazy1[2 * i + 1].val = lazy1[2 * i + 2].val = 0;
118         }
119         lazy[i] = num_t(-1);
120         lazy1[i] = num_t(0);
121     }
122     else{
123         tree[i] = tree[i].lazy_op1(lazy1[i], r - l + 1);
124         if(l != r){
125             if(lazy[2 * i + 1].val == -1) lazy1[2 * i + 1].val += lazy1[i].
126                 val;
127             else{
128                 lazy[2 * i + 1].val += lazy1[i].val + lazy1[2 * i + 1].val;
129                 lazy1[2 * i + 1].val = 0;
130             }
131         }
132         if(lazy[2 * i + 2].val == -1) lazy1[2 * i + 2].val += lazy1[i].
133             val;
134     }
135 }
136     lazy1[i] = num_t(0);
137 }
138 }
139 };
140
141 int main(){
142     ios_base::sync_with_stdio(false); cin.tie(NULL);
143     int n, q; cin >> n >> q;
144     segtree<sum_t> st;
145     st.init(n);
146     while(q--){
147         int t; cin >> t;
148         if(t == 1){
149             int l, r, x; cin >> l >> r >> x;
150             r--;
151             st.update(l, r, x);
152         }
153         else if(t == 2){
154             int l, r, x; cin >> l >> r >> x;
155             r--;
156             st.update1(l, r, x);
157         }
158         else{
159             int l, r; cin >> l >> r;
160             r--;
161             cout << st.query(l, r).val << endl;
162         }
163     }
164 }

```

3.30 Segment Tree Max Subarray Sum

```

1 const ll inf=1e18;
2
3 struct Node {
4     ll maxi, l_max, r_max, sum;

```

```

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

```

3.31 Segment Tree Range Update

```

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 };

```

3.32 Segment Tree Struct Types

```

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};

```

3.33 Segment Tree Struct

```

1 // works as a 0-indexed segtree (not lazy)
2 template <typename num_t>
3 struct segtree
4 {
5     int n, k;
6     vector<num_t> tree;
7     void init(int s, vector<ll> arr)
8     {
9         n = s;
10        k = 0;
11        while ((1 << k) < n)
12            k++;
13        tree = vector<num_t>(2 * (1 << k) + 1);
14        for (int i = 0; i < n; i++)
15        {
16            tree[(1 << k) + i] = arr[i];
17        }
18        for (int i = (1 << k) - 1; i > 0; i--)
19        {
20            tree[i] = tree[i * 2] + tree[i * 2 + 1];
21        }
22    }
23    void update(int a, ll b)
24    {
25        a += (1 << k);
26        tree[a] = b;
27        for (a /= 2; a >= 1; a /= 2)
28        {

```

```

29     tree[a] = tree[a * 2] + tree[a * 2 + 1];
30 }
31 }
32 num_t find(int a, int b)
33 {
34     a += (1 << k);
35     b += (1 << k);
36     num_t s;
37     while (a <= b)
38     {
39         if (a % 2 == 1)
40             s = s + tree[a++];
41         if (b % 2 == 0)
42             s = s + tree[b--];
43         a /= 2;
44         b /= 2;
45     }
46     return s;
47 }
48 };

```

3.34 Segment Tree Walk

```

/*
Segment Tree Walk - Find First Position >= val
-----
Indexing: 0-based
Bounds: [0, n-1]
Time Complexity:
    - build: O(n)
    - update(pos, val): O(log n)
    - get(L, R): O(log n) -> min value in [L, R]
    - query(L, R, val): O(log n) -> find first index in [L, R] where a[i]
        ] >= val

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

```

```

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         // Update value at original index 'pos' to 'val'
39         void update(int pos, ll val) {
40             pos = final_pos[pos]; // leaf position
41             a[pos] = val;
42             for (pos /= 2; pos > 0; pos /= 2)
43                 a[pos] = min(a[pos * 2], a[pos * 2 + 1]);
44         }
45
46         // Get min value in [L, R], with current node interval [l, r] and root
47         // 'node'
48         ll get(int l, int r, int L, int R, int node) {
49             if (L > R) return 1e18;
50             if (l == L && r == R) return a[node];
51             int mid = (l + r) / 2;
52             return min(
53                 get(l, mid, L, min(R, mid), node * 2),
54                 get(mid + 1, r, max(L, mid + 1), R, node * 2 + 1)
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) {

```

```

59     if (l > R || r < L) return {-1, 0};           // out of query
60     bounds
61     if (a[node] < val) return {-1, 0};           // all values < val
62     if (l == r) return {a[node], 1};              // leaf node that
63     satisfies
64
65     int mid = (l + r) / 2;
66     auto left = query(l, mid, L, R, node * 2, val);
67     if (left.first != -1) return left;
68     return query(mid + 1, r, L, R, node * 2 + 1, val);
69 }
70
71 // Example usage:
72 int n = 8;
73 vector<ll> vals = {4, 2, 7, 1, 9, 5, 6, 3};
74 SegmentTreeWalk st(n);
75 st.build(0, n - 1, 1, vals);

```

3.35 Sparse Table

```

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

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

```

```

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;

```

```

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 }

```

3.36 Sparse Table 2D

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 const int N = 505, LG = 10;
5

```

```

6 int st[N][N][LG][LG];
7 int a[N][N], lg2[N];
8
9 int yo(int x1, int y1, int x2, int y2) {
10    x2++;
11    y2++;
12    int a = lg2[x2 - x1], b = lg2[y2 - y1];
13    return max(
14        max(st[x1][y1][a][b], st[x2 - (1 << a)][y1][a][b]),
15        max(st[x1][y2 - (1 << b)][a][b], st[x2 - (1 << a)][y2 - (1 << b)][a][b])
16    );
17}
18
19 void build(int n, int m) { // 0 indexed
20    for (int i = 2; i < N; i++) lg2[i] = lg2[i >> 1] + 1;
21    for (int i = 0; i < n; i++) {
22        for (int j = 0; j < m; j++) {
23            st[i][j][0][0] = a[i][j];
24        }
25    }
26    for (int a = 0; a < LG; a++) {
27        for (int b = 0; b < LG; b++) {
28            if (a + b == 0) continue;
29            for (int i = 0; i + (1 << a) <= n; i++) {
30                for (int j = 0; j + (1 << b) <= m; j++) {
31                    if (!a) {
32                        st[i][j][a][b] = max(st[i][j][a][b - 1], st[i][j + (1 << (b - 1))][a][b - 1]);
33                    } else {
34                        st[i][j][a][b] = max(st[i][j][a - 1][b], st[i + (1 << (a - 1))][j][a - 1][b]);
35                    }
36                }
37            }
38        }
39    }
40}
41
42 string s[N];
43 int l[N][N], u[N][N];
44
45 int32_t main() {
46    ios_base::sync_with_stdio(0);
47    cin.tie(0);
48
49    int n, m;
50    cin >> n >> m;
51    for (int i = 0; i < n; i++) {
52        cin >> s[i];
53    }
54    for (int i = 0; i < n; i++) {
55        for (int j = 0; j < m; j++) {
56            if (!j) l[i][j] = 1;
57            else l[i][j] = 1 + (s[i][j - 1] <= s[i][j] ? l[i][j - 1] : 0);
58        }
59    }
60    for (int j = 0; j < m; j++) {
61        for (int i = 0; i < n; i++) {
62            if (!i) u[i][j] = 1;
63            else u[i][j] = 1 + (s[i - 1][j] <= s[i][j] ? u[i - 1][j] : 0);
64        }
65    }
66    for (int i = 0; i < n; i++) {
67        for (int j = 0; j < m; j++) {
68            int nw = 1, mnx = u[i][j], mny = l[i][j];
69            for (int len = 1; len <= min(i, j); len++) {
70                mnx = min(mnx, u[i][j - len]);
71                mny = min(mny, l[i - len][j]);
72                if (min(mnx, mny) >= len + 1) nw++;
73                else break;
74            }
75            a[i][j] = nw;
76        }
77    }
78    build(n, m);
79    int q;
80    cin >> q;
81    while (q--) {
82        int x1, y1, x2, y2;
83        cin >> x1 >> y1 >> x2 >> y2;
84        x1--, y1--;
85        x2--;
86        y2--;
87        int l = 1, r = min(x2 - x1 + 1, y2 - y1 + 1), ans = 0;
88        while (l <= r) {

```

```

89     int mid = l + r >> 1;
90     if (yo(x1 + mid - 1, y1 + mid - 1, x2, y2) >= mid) ans = mid, l =
91         mid + 1;
92     else r = mid - 1;
93 }
94 cout << ans << '\n';
95 }
96 return 0;
97 }
// https://www.codechef.com/problems/CENS20B

```

3.37 Square Root Decomposition

```

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

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

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

```

```

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         if (containsA && containsB) {
58             newBlocks[curBlock++] = {B.l, cutA};
59             newBlocks[curBlock++] = {cutA, cutB};
60             newBlocks[curBlock++] = {cutB, B.r};
61         } else if (containsA) {
62             newBlocks[curBlock++] = {B.l, cutA};
63             newBlocks[curBlock++] = {cutA, B.r};
64         } else if (containsB) {
65             newBlocks[curBlock++] = {B.l, cutB};
66             newBlocks[curBlock++] = {cutB, B.r};
67         } else {
68             newBlocks[curBlock++] = B;
69         }
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 }

```

3.38 Tuple

```

1 #include <tuple>
2 #include <iostream>
3 #include <string>
4
5 int main() {
6     // -----
7     // DECLARING A TUPLE
8     // -----
9     std::tuple<int, double, std::string> t1;
10
11    // -----
12    // INITIALIZING A TUPLE
13    // -----
14    std::tuple<int, double, std::string> t2(42, 3.14, "hello");
15
16    // -----
17    // USING std::make_tuple
18    // (automatic type deduction)
19    // -----
20    auto t3 = std::make_tuple(7, 2.71, "world");
21
22    // -----
23    // ACCESSING ELEMENTS USING std::get
24    // -----
25    int a = std::get<0>(t2);           // a = 42
26    double b = std::get<1>(t2);        // b = 3.14
27    std::string c = std::get<2>(t2);   // c = "hello"
28
29    // -----
30    // MODIFYING ELEMENTS
31    // -----
32    std::get<1>(t2) = 5.67; // change second element
33
34    // -----
35    // STRUCTURED BINDINGS (C++17)
36    // Unpacks the tuple into variables
37    // -----
38    auto [x, y, z] = t3; // x=7, y=2.71, z="world"
39
40    // -----
41    // tuple_size and tuple_element
42    // (using decltype to reference the type)
43    // -----

```

```

44     std::size_t n = std::tuple_size<decltype(t3)>::value; // n = 3
45     using FirstType = std::tuple_element<0, decltype(t3)>::type;
46     // FirstType is int
47
48     // -----
49     // SMALL EXAMPLE OF USE
50     // -----
51     auto person = std::make_tuple("Alice", 25, 1.70);
52
53     // Access values
54     std::cout << "Name:" << std::get<0>(person) << "\n";
55     std::cout << "Age:" << std::get<1>(person) << "\n";
56     std::cout << "Height:" << std::get<2>(person) << "\n";
57
58     // Modify a value
59     std::get<1>(person) = 26;
60
61     // Print updated value
62     std::cout << "Updated age:" << std::get<1>(person) << "\n";
63
64     return 0;
65 }
```

3.39 Treap

```

1 std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().
2   count());
3
3 struct Node {
4   Node *l = 0, *r = 0;
5   int val, y, c = 1;
6   Node(int val) : val(val), y(rng()) {}
7   void recalc();
8 };
9
10 int cnt(Node* n) { return n ? n->c : 0; }
11 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
12
13 template<class F> void each(Node* n, F f) {
14   if (n) { each(n->l, f); f(n->val); each(n->r, f); }
15 }
16
17 pair<Node*, Node*> split(Node* n, int k) {
18   if (!n) return {};
19   if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
20     auto pa = split(n->l, k);
21     n->l = pa.second;
22     n->recalc();
23     return {pa.first, n};
24   } else {
25     auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
26     n->r = pa.first;
27     n->recalc();
28     return {n, pa.second};
29   }
30 }
31
32 Node* merge(Node* l, Node* r) {
33   if (!l) return r;
34   if (!r) return l;
35   if (l->y > r->y) {
36     l->r = merge(l->r, r);
37     l->recalc();
38     return l;
39   } else {
40     r->l = merge(l, r->l);
41     r->recalc();
42     return r;
43   }
44 }
45
46 Node* ins(Node* t, Node* n, int pos) {
47   auto pa = split(t, pos);
48   return merge(merge(pa.first, n), pa.second);
49 }
50
51 // Example application: move the range [l, r) to index k
52 void move(Node*& t, int l, int r, int k) {
53   Node *a, *b, *c;
54   tie(a,b) = split(t, l); tie(b,c) = split(b, r - 1);
55   if (k <= l) t = merge(ins(a, b, k), c);
56   else t = merge(a, ins(c, b, k - r));
57 }
58
59 // Usage
60 // create treap
```

```

61 // Node* name=nullptr;
62 // insert element
63 // name=ins(name, new Node(val), pos);
64 // Node* x = new Node(val);
65 // name = ins(name, x, pos);
66 // merge two treaps (name before x)
67 // name=merge(name, x);
68 // split treap (this will split treap in two treaps,
69 // first with elements [0, pos) and second with elements [pos, n])
70 // pa will be pair of two treaps
71 // auto pa = split(name, pos);
72 // move range [l, r) to index k
73 // move(name, l, r, k);
74 // iterate over treap
75 // each(name, [&](int val) {
76 //     cout << val << ' ';
77 // });

```

3.40 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 }

```

3.41 Treap With Inversion

```

1 std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch() .
2   count());
3
4 struct Node {
5   Node *l = 0, *r = 0;
6   int val, y, c = 1;
7   bool rev = 0;
8   Node(int val) : val(val), y(rng()) {}
9   void recalc();
10  void push();
11 };
12
13 int cnt(Node* n) { return n ? n->c : 0; }
14 void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
15 void Node::push() {
16   if (rev) {
17     rev = 0;
18     swap(l, r);
19     if (l) l->rev ^= 1;
20     if (r) r->rev ^= 1;
21   }
22 }
23
24 template<class F> void each(Node* n, F f) {
25   if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
26 }
27
28 pair<Node*, Node*> split(Node* n, int k) {
29   if (!n) return {};
30   n->push();
31   if (cnt(n->l) >= k) {
32     auto pa = split(n->l, k);
33     n->l = pa.second;
34     n->recalc();
35     return {pa.first, n};
36   } else {
37     auto pa = split(n->r, k - cnt(n->l) - 1);
38     n->r = pa.first;
39     n->recalc();
40     return {n, pa.second};
41   }
42 }
43 Node* merge(Node* l, Node* r) {
44   if (!l) return r;
45   if (!r) return l;
46   l->push();
47   r->push();
48   if (l->y > r->y) {
49     l->r = merge(l->r, r);
50     l->recalc();
51     return l;
52   } else {
53     r->l = merge(l, r->l);
54     r->recalc();
55     return r;
56   }
57 }
58
59 Node* ins(Node* t, Node* n, int pos) {
60   auto pa = split(t, pos);
61   return merge(merge(pa.first, n), pa.second);
62 }
63
64 // Example application: reverse the range [l, r]
65 void reverse(Node*& t, int l, int r) {
66   Node *a, *b, *c;
67   tie(a,b) = split(t, l);
68   tie(b,c) = split(b, r - l + 1);
69   b->rev ^= 1;
70   t = merge(merge(a, b), c);
71 }
72
73 void move(Node*& t, int l, int r, int k) {
74   Node *a, *b, *c;
75   tie(a,b) = split(t, l);
76   tie(b,c) = split(b, r - l);
77   if (k <= l) t = merge(ins(a, b, k), c);
78   else t = merge(a, ins(c, b, k - r));
79 }

```

3.42 Treap With Inversions and Range Updates

```

2  using namespace std;
3
4  typedef long long ll;
5
6  const int mod = 998244353;
7
8  template<int MOD>
9  struct ModInt {
10    ll v;
11    ModInt(ll _v = 0) {v = (-MOD < _v && _v < MOD) ? _v : _v % MOD; if (v
12      < 0) v += MOD;}
13    ModInt& operator += (const ModInt &other) {v += other.v; if (v >= MOD)
14      v -= MOD; return *this;}
15    ModInt& operator -= (const ModInt &other) {v -= other.v; if (v < 0) v
16      += MOD; return *this;}
17    ModInt& operator *= (const ModInt &other) {v = v * other.v % MOD;
18      return *this;}
19    ModInt& operator /= (const ModInt &other) {return *this *= inverse(
20      other);}
21    bool operator == (const ModInt &other) const {return v == other.v;}
22    bool operator != (const ModInt &other) const {return v != other.v;}
23    friend ModInt operator + (ModInt a, const ModInt &b) {return a += b;}
24    friend ModInt operator - (ModInt a, const ModInt &b) {return a -= b;}
25    friend ModInt operator * (ModInt a, const ModInt &b) {return a *= b;}
26    friend ModInt operator / (ModInt a, const ModInt &b) {return a /= b;}
27    friend ModInt operator - (const ModInt &a) {return 0 - a;}
28    friend ModInt power(ModInt a, ll b) {ModInt ret(1); while (b > 0) {if
29      (b & 1) ret *= a; a *= a; b >>= 1;} return ret;}
30    friend ModInt inverse(ModInt a) {return power(a, MOD - 2);}
31    friend istream& operator >> (istream &is, ModInt &m) {is >> m.v; m.v =
32      (-MOD < m.v && m.v < MOD) ? m.v : m.v % MOD; if (m.v < 0) m.v +=
33      MOD; return is;}
34    friend ostream& operator << (ostream &os, const ModInt &m) {return os
35      << m.v;}
36
37  struct Line{
38    ModInt<mod> b = 1, c = 0;
39    ModInt<mod> operator()(ModInt<mod> x){ return b * x + c;}
40    Line operator()(Line other) {return Line{b * other.b, other.c * b + c
41      };}
42    operator bool() const{
43      return b != 1 || c != 0;
44    }
45
46  };
47
48  template<class F> void each(Node* n, F f) {
49    if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
50
51  int cnt(Node* n) { return n ? n->c : 0; }
52  ModInt<mod> sum1(Node* n) { return n ? n->sum : ModInt<mod>(0); }
53  void Node::recalc() { if(l) l->push(); if(r) r->push(); c = cnt(l) +
54    cnt(r) + 1; sum = sum1(l) + sum1(r) + val;}
55  void Node::push(){
56    if(rev){
57      rev = 0;
58      swap(l, r);
59      if(l) l->rev ^= 1;
60      if(r) r->rev ^= 1;
61    }
62    if(line){
63      val = line(val);
64      sum = sum * line.b + line.c * c;
65      if(l){
66        l->line = line(l->line);
67      }
68      if(r){
69        r->line = line(r->line);
70      }
71      line = {1, 0};
72    }
73
74  template<class F> void each(Node* n, F f) {
75    if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }

```

```

76 }
77
78 pair<Node*, Node*> split(Node* n, int k) {
79     if (!n) return {};
80     n->push();
81     if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
82         auto [L,R] = split(n->l, k);
83         n->l = R;
84         n->recalc();
85         return {L, n};
86     } else {
87         auto [L,R] = split(n->r, k - cnt(n->l) - 1); // and just "k"
88         n->r = L;
89         n->recalc();
90         return {n, R};
91     }
92 }
93
94 Node* merge(Node* l, Node* r) {
95     if (!l) return r;
96     if (!r) return l;
97     l->push();
98     r->push();
99     if (l->y > r->y) {
100         l->r = merge(l->r, r);
101         return l->recalc(), l;
102     } else {
103         r->l = merge(l, r->l);
104         return r->recalc(), r;
105     }
106 }
107
108 Node* ins(Node* t, Node* n, int pos) {
109     auto [l,r] = split(t, pos);
110     return merge(merge(l, n), r);
111 }
112
113 void move(Node*& t, int l, int r, int k){
114     Node *a, *b, *c;
115     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
116     if(k <= l) t = merge(ins(a, b, k), c);
117     else t = merge(a, ins(c, b, k - r));
118 }

119 void revv(Node*& t, int l, int r){
120     Node *a, *b, *c;
121     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
122     b->rev ^= 1;
123     t = merge(merge(a, b), c);
124 }
125
126 ModInt<mod> query(Node*& t, int l, int r){
127     Node *a, *b, *c;
128     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
129     ModInt<mod> res = sum1(b);
130     t = merge(merge(a, b), c);
131     return res;
132 }
133
134 void remove(Node*& t, int x){
135     Node *a, *b, *c;
136     tie(a, b) = split(t, x); tie(b, c) = split(b, 1);
137     t = merge(a, c);
138 }
139
140 void update(Node*& t, int l, int r, ll X, ll Y){
141     Node *a, *b, *c;
142     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
143     b->line = Line{X, Y}(b->line);
144     t = merge(a, merge(b, c));
145 }
146
147 int main(){
148     ios_base::sync_with_stdio(false); cin.tie(NULL);
149     int n, q; cin >> n >> q;
150     vector<int> a(n);
151     for(int i = 0; i < n; i++){
152         cin >> a[i];
153     }
154     Node* treap = nullptr;
155     for(int i = 0; i < n; i++){
156         treap = ins(treap, new Node(a[i]), i);
157     }
158     while(q--){
159         int t; cin >> t;
160         if(t == 0){
161

```

```

162     int i, x; cin >> i >> x;
163     treap = ins(treap, new Node(x), i);
164 }
165 else if(t == 1){
166     int x; cin >> x;
167     remove(treap, x);
168 }
169 else if(t == 2){
170     int l, r; cin >> l >> r;
171     revv(treap, l, r);
172 }
173 else if(t == 3){
174     int l, r, b, c; cin >> l >> r >> b >> c;
175     update(treap, l, r, b, c);
176 }
177 else{
178     int l, r; cin >> l >> r;
179     cout << query(treap, l, r) << endl;
180 }
181 }
182 }
```

4 Dynamic Programming

4.1 CHT Deque

```

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

We reformulate:
```

```

17     y = m * x + c
18     line: m = -k - 1, c = dp[k][j-1] - s[k+1]
19     eval: m * a[i] + c + a[i] * i + s[i]
20 */
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 }
```

```
59 }
60 }
```

4.2 Digit DP

```
/*
Digit Dynamic Programming (Digit DP)
-----
Goal: Count numbers in range [0, x] that do not have two adjacent
equal digits.

State:
- pos: current digit position
- last: digit placed at previous position (0 to 9)
- f: tight flag (0 = must match prefix of x, 1 = already below x)
- z: leading zero flag (1 = still in leading zero zone)

Notes:
- Solve up to x using 'solve(x)'
- Can be modified to count palindromes, digits divisible by 3, etc.
*/
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();
```

```
37     while (x > 0) {
38         num.push_back(x % 10);
39         x /= 10;
40     }
41     reverse(num.begin(), num.end());
42     memset(DP, -1, sizeof(DP));
43     return g(0, 0, 0, 1);
44 }
```

4.3 Divide and Conquer DP

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

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

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

```

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

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

```

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 Knuth's Algorithm

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 const int N = 1010;
5 using ll = long long;
6 /*
7 Knuths optimization works for optimization over sub arrays
8 for which optimal middle point depends monotonously on the end points.
9 Let mid[l,r] be the first middle point for (l,r) sub array which gives
10 optimal result.
11 It can be proven that mid[l,r-1] <= mid[l,r] <= mid[l+1,r]
12 - this means monotonicity of mid by l and r.
13 Applying this optimization reduces time complexity from O(k^3) to O(k^2)
14 because with fixed s (sub array length) we have m_right(l) = mid[l+1][r]
15 = m_left(l+1).
16 That's why nested l and m loops require not more than 2k iterations
17 overall.
18 */
19
20 int n, k;
21 int a[N], mid[N][N];
22 ll res[N][N];
23 ll solve() {
24     for (int s = 0; s <= k; s++) {           // s - length of the subarray
25         for (int l = 0; l + s <= k; l++) { // l - left point
26             int r = l + s;                  // r - right point
27             if (s < 2) {
28                 res[l][r] = 0;            // base case- nothing to break
29             } else {
```

```

26     mid[l][r] = 1;           // mid is equal to left border
27     continue;
28 }
29 int mleft = mid[l][r - 1];
30 int mright = mid[l + 1][r];
31 res[l][r] = 2e18;
32 for (int m = mleft; m <= mright; m++) {    // iterating for m in
33     the bounds only
34     ll tmp = res[l][m] + res[m][r] + (a[r] - a[l]);
35     if (res[l][r] > tmp) {           // relax current solution
36         res[l][r] = tmp;
37         mid[l][r] = m;
38     }
39 }
40 ll ans = res[0][k];
41 return ans;
42 }
43 int main() {
44     int i, j, m;
45     while(cin >> n >> k) {
46         for(i = 1; i <= k; i++) cin >> a[i];
47         a[0] = 0;
48         a[k + 1] = n;
49         k++;
50         cout << solve() << endl;
51     }
52     return 0;
53 }
54 // https://vjudge.net/problem/ZOJ-2860

```

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

```

4.7 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
13      query
14  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
21         slope
22     bool operator<(ll x) const { return p < x; }           // Query
23         comparator
24 };
25
26 struct LineContainer : multiset<Line, less<> {
27     // (for doubles, use inf = 1/.0, div(a,b) = a/b)
28     static const ll inf = LLONG_MAX;
29     ll div(ll a, ll b) { // Floored division
30         return a / b - ((a ^ b) < 0 && a % b);
31     }
32     // Update intersection point x->p with y
33     bool isect(iterator x, iterator y) {
34         if (y == end()) return x->p = inf, false;
35         if (x->k == y->k)
36             x->p = (x->m > y->m ? inf : -inf); // higher line wins
37         else
38             x->p = div(y->m - x->m, x->k - y->k);
39         return x->p >= y->p;
40     }
41     // Add new line: y = k * x + m
42     void add(ll k, ll m) {
43         auto z = insert({k, m, 0}), y = z++, x = y;
44         // Remove dominated lines after y
45         while (isect(y, z)) z = erase(z);
46         // Remove dominated lines before y
47         if (x != begin() && isect(--x, y))
48             isect(x, y = erase(y));
49         // Further cleanup to preserve order
50         while ((y = x) != begin() && (--x)->p >= y->p)
51             isect(x, erase(y));
52     }
53     // Query max y at given x
54     ll query(ll x) {
55         assert(!empty());
56         auto l = *lower_bound(x);
57     }

```

```

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

```

4.8 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
11 vector<int> last_pos(n + 1); // last_pos[len] = index in v[] ending LIS
12     of length len
13 for (int i = 0; i < n; i++) {
14     auto it = lower_bound(dp.begin(), dp.end(), v[i]);
15     int len = it - dp.begin();
16     if (it == dp.end()) {
17         dp.push_back(v[i]);
18         dp_index.push_back(i); // Ignore if no recovery
19     } else {
20         *it = v[i];
21         dp_index[len] = i; // Ignore if no recovery
22     }
23     if (len > 0) parent[i] = dp_index[len - 1]; // Ignore if no recovery
24 }
25 // Reconstruct LIS
26 vector<int> lis;
27 int pos = dp_index.back();
28 while (pos != -1) {
29     lis.push_back(v[pos]);
30     pos = parent[pos];
31 }

```

```
31 | reverse(lis.begin(), lis.end());
```

4.9 SOS DP

```
1 vector<int> SOS_DP(vector<int> A, int k) {
2     vector<int> Ap = A;
3     for (int i = k - 1; i >= 0; i--)
4         for (int mk = 0; mk < (1 << k); mk++)
5             if (mk & (1 << i))
6                 Ap[mk] += Ap[mk - (1 << i)];
7     return Ap;
8 }
9
10 vector<int> SOS_DP_inv (vector<int> Ap, int k) {
11     vector<int> A = Ap;
12     for (int i = 0; i < k; i++)
13         for (int mk = 0; mk < (1 << k); mk++)
14             if (mk & (1 << i))
15                 A[mk] -= A[mk - (1 << i)];
16     return A;
17 }
18
19 vector<int> SUPER_SOS_DP(vector<int> A, int k) {
20     vector<int> Ap = A;
21     for (int i = k - 1; i >= 0; i--)
22         for (int mk = 0; mk < (1 << k); mk++)
23             if (~mk & (1 << i))
24                 Ap[mk] += Ap[mk + (1 << i)];
25     return Ap;
26 }
27
28 vector<int> SUPER_SOS_DP_inv (vector<int> Ap, int k) {
29     vector<int> A = Ap;
30     for (int i = 0; i < k; i++)
31         for (int mk = (1 << k) - 1; mk >= 0; mk--)
32             if (~mk & (1 << i))
33                 A[mk] -= A[mk + (1 << i)];
34     return A;
35 }
36
37 // or convolution
38 Ap = SOS_DP(A, k);
39 Bp = SOS_DP(B, k);
```

```
40
41 for (int i = 0; i < (1 << k); i++)
42     Cp[i] = Ap[i] * Bp[i];
43
44 C = SOS_DP_inv(Cp, k);
45
46 // and convolution
47 Ap = SUPER_SOS_DP(A, k);
48 Bp = SUPER_SOS_DP(B, k);
49
50 for (int i = 0; i < (1 << k); i++)
51     Cp[i] = Ap[i] * Bp[i];
52
53 C = SUPER_SOS_DP_inv(Cp, k);
```

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 // to find min cut run bfs from source and find all vertices that can be
11 // reached
12 // edges between vertices that can be reached and the ones that cant are
13 // the min cut
14 struct FlowEdge {
15     int v, u;
16     long long cap, flow = 0;
17     FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
18 };
19
20 struct Dinic {
21     const long long flow_inf = 1e18;
22     vector<FlowEdge> edges;
23     vector<vector<int>> adj;
24     vector<int> sz;
```

```

23     int n, m = 0;
24     int s, t;
25     vector<int> level, ptr;
26     queue<int> q;
27
28     Dinic(int n, int s, int t) : n(n), s(s), t(t) {
29         adj.resize(n);
30         level.resize(n);
31         ptr.resize(n);
32         sz.resize(n);
33     }
34
35     void add_edge(int v, int u, long long cap) {
36         edges.emplace_back(v, u, cap);
37         edges.emplace_back(u, v, 0);
38         adj[v].push_back(m);
39         adj[u].push_back(m + 1);
40         m += 2;
41     }
42
43     bool bfs() {
44         while (!q.empty()) {
45             int v = q.front();
46             q.pop();
47             for (int id : adj[v]) {
48                 if (edges[id].cap - edges[id].flow < 1)
49                     continue;
50                 if (level[edges[id].u] != -1)
51                     continue;
52                 level[edges[id].u] = level[v] + 1;
53                 q.push(edges[id].u);
54             }
55         }
56         return level[t] != -1;
57     }
58
59     long long dfs(int v, long long pushed) {
60         if (pushed == 0)
61             return 0;
62         if (v == t)
63             return pushed;
64         for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
65             int id = adj[v][cid];
66             int u = edges[id].u;
67             if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow
68                 < 1)
69                 continue;
70             long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
71             if (tr == 0)
72                 continue;
73             edges[id].flow += tr;
74             edges[id ^ 1].flow -= tr;
75             return tr;
76         }
77         return 0;
78     }
79     long long flow() {
80         long long f = 0;
81         while (true) {
82             fill(level.begin(), level.end(), -1);
83             level[s] = 0;
84             q.push(s);
85             if (!bfs())
86                 break;
87             fill(ptr.begin(), ptr.end(), 0);
88             while (long long pushed = dfs(s, flow_inf)) {
89                 f += pushed;
90             }
91         }
92         return f;
93     }
94
95     vector<pair<int, int>> minCut(){
96         vector<pair<int, int>> ans;
97         queue<int> q;
98         q.push(s);
99         vector<bool> vis(n, 0);
100        while((int)q.size()){
101            int cur = q.front();
102            q.pop();
103            vis[cur] = 1;
104            for(int id: adj[cur]){
105                if(edges[id].cap - edges[id].flow <= 0) continue;
106            }
107        }
108    }

```

```

107     if(vis[edges[id].u]) continue;
108     vis[edges[id].u] = 1;
109     q.push(edges[id].u);
110   }
111 }
112 for(auto edge: edges){
113   if(vis[edge.v] && !vis[edge.u] && edge.flow > 0) {
114     ans.push_back({edge.v, edge.u});
115   }
116 }
117 return ans;
118 }

119 bool getPath(int cur, vector<int> &path){
120   path.push_back(cur);
121   for(int &i = sz[cur]; i >= 0; i--){
122     int edge_index = adj[cur][i];
123     if(edge_index & 1) continue;
124     if(edges[edge_index].cap - edges[edge_index].flow == 0){
125       i--;
126       getPath(edges[edge_index].u, path);
127       return true;
128     }
129   }
130   return false;
131 }

132 vector<vector<int>> getPaths(){
133   vector<vector<int>> ans;
134   vector<int> path;
135   for(int i = 0; i < n; i++) sz[i] = (int)adj[i].size() - 1;
136   while(getPath(s, path)){
137     ans.push_back(path);
138     path.clear();
139   }
140   return ans;
141 }
142 }
143 }
144 };

```

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   dist[0] = INF;
22
23   while (!q.empty())
24   {
25     int node = q.front();
26     q.pop();
27     if (dist[node] >= dist[0])
28     {
29       continue;
30     }
31     for (int son : g[node])
32     {
33       // If the match of son is matched
34       if (dist[match[son]] == INF)
35       {
36         dist[match[son]] = dist[node] + 1;
37         q.push(match[son]);
38       }
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
67 int hopcroft_karp()
68 {
69     int cnt = 0;
70     // While there is an alternating path
71     while (bfs())
72     {
73         for (int node = 1; node <= n; node++)
74         {
75             // If node is unmatched but we can match it using an augmenting
76             // path
77             if (!match[node] && dfs(node))
78             {
79                 cnt++;
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

```

```

87 // los puntos estan 1 indexados
88 // el punto 1 de m es el punto n+1 de g
89 // hopcroft_karp() devuelve el tamano del maximo matching
90 // match contiene el match de cada punto
91 // si match de i es 0, entonces i no esta matcheado
92 //
93 // 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         int x=0, y=-1;
44         while (y== -1) {
45             while (!q.empty() && y== -1) {
46                 x = q.front(); q.pop();
47                 forn(j, n) if (mt[x][j] == lx[x] + ly[j] && !T[j]) {
48                     if (yx[j] == -1) { y = j; break; }
49                     T[j] = 1;
50                     bpone(yx[j], x);
51                 }
52             }
53             if (y!= -1) break;
54             int dlt = INFTO;
55             forn(j, n) if (!T[j]) dlt = min(dlt, slk[j]);
56             forn(k, n) {
57                 if (S[k]) lx[k] -= dlt;
58                 if (T[k]) ly[k] += dlt;
59                 if (!T[k]) slk[k] -= dlt;
60             }
61             // q = queue<int>();
62             forn(j, n) if (!T[j] && !slk[j]) {
63                 if (yx[j] == -1) {
64                     x = slkx[j]; y = j; break;
65                 } else {
66                     T[j] = 1;
67                     if (!S[yx[j]]) bpone(yx[j], slkx[j]);
68                 }
69             }
70         }
71         if (y!= -1) {
72             for(int p = x; p != -2; p = prv[p]) {
73                 yx[y] = p;
74                 int ty = xy[p]; xy[p] = y; y = ty;
75             }
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 }

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     int flow = 0;
64     int cost = 0;
65     vector<int> d, p;
66     while (flow < K)
67     {
68         shortest_paths(N, s, d, p);
69         if (d[t] == INF)
70             break;

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         }

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     }

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>>
2                             capacity,
3                                     int source, int sink)
4 {
5     int n = adj.size();
6     vector<int> parent(n, -1);
7     // Find a way from the source to sink on a path with non-negative
8     // capacities
9     auto reachable = [&]() -> bool
10    {
11        queue<int> q;
12        q.push(source);
13        while (!q.empty())
14        {
15            int node = q.front();
16            q.pop();
17            for (int son : adj[node])
18            {
19                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 long long flow = 0;
27 // While there is a way from source to sink with non-negative
28 // 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
56
57 //vector<vector<long long>> capacity(n, vector<long long>(n));
58 //vector<vector<int>> adj(n);
59 //adj[a].push_back(b);

```

```

60 //adj[b].push_back(a);
61 //capacity[a][b] += c;

```

5.6 Min Cost Max Flow

```

1 /**
2 * If costs can be negative, call setpi before maxflow, but note that
3 * negative cost cycles are not supported.
4 * To obtain the actual flow, look at positive values only
5 * Time: $O(F E \log(V))$ where F is max flow. $O(VE)$ for setpi.
6 */
7 #include <bits/stdc++.h>
8 using namespace std;
9
10 #include <ext/pb_ds/priority_queue.hpp>
11 using namespace __gnu_pbds;
12
13 #define rep(i, a, b) for(int i = a; i < (b); ++i)
14 #define all(x) begin(x), end(x)
15 #define sz(x) (int)(x).size()
16
17 typedef long long ll;
18 typedef pair<int, int> pii;
19 typedef vector<int> vi;
20
21 #pragma once
22
23 // #include <bits/extc++.h> // include-line, keep-include
24
25 const ll INF = numeric_limits<ll>::max() / 4;
26
27 struct MCMF {
28     struct edge {
29         int from, to, rev;
30         ll cap, cost, flow;
31     };
32     int N;
33     vector<vector<edge>> ed;
34     vi seen;
35     vector<ll> dist, pi;
36     vector<edge*> par;
37
38     MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}

```

```

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 void discharge(int u) {
28     while (excess[u] > 0) {
29         if (seen[u] < n) {
30             int v = seen[u];
31             if (capacity[u][v] - flow[u][v] > 0 && height[u] > height[v])
32                 push(u, v);
33             else
34                 seen[u]++;
35         } else {
36             relabel(u);
37             seen[u] = 0;
38         }
39     }
40 }
41 }

42 int max_flow(int s, int t) {
43     height.assign(n, 0);
44     height[s] = n;
45     flow.assign(n, vector<int>(n, 0));
46     excess.assign(n, 0);
47     excess[s] = inf;
48     for (int i = 0; i < n; i++) {
49         if (i != s)
50             push(s, i);
51     }
52     seen.assign(n, 0);

53     while (!excess_vertices.empty()) {
54         int u = excess_vertices.front();
55         excess_vertices.pop();
56         if (u != s && u != t)
57             discharge(u);
58     }
59 }

60     int max_flow = 0;
61     for (int i = 0; i < n; i++)
62         max_flow += flow[i][t];
63

```

```

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 }

```

6.3 Point Struct2

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 using ld = long double;
4 const ld eps = 1e-9, inf = numeric_limits<ld>::max(), pi = acos(-1);
5 // For use with integers, just set eps=0 and everything remains the same
6 bool geq(ld a, ld b){return a-b >= -eps;}      //a >= b
7 bool leq(ld a, ld b){return b-a >= -eps;}      //a <= b
8 bool ge(ld a, ld b){return a-b > eps;}          //a > b
9 bool le(ld a, ld b){return b-a > eps;}          //a < b
10 bool eq(ld a, ld b){return abs(a-b) <= eps;}    //a == b
11 bool neq(ld a, ld b){return abs(a-b) > eps;}   //a != b
12
13 struct point{
14     ld x, y;
15     point(): x(0), y(0){}
16     point(ld x, ld y): x(x), y(y){}
17
18     point operator+(const point & p) const{return point(x + p.x, y + p.y)
19     ;}
20     point operator-(const point & p) const{return point(x - p.x, y - p.y)
21     ;}
22     point operator*(const ld & k) const{return point(x * k, y * k);}
23     point operator/(const ld & k) const{return point(x / k, y / k);}
24
25     point operator+=(const point & p){*this = *this + p; return *this;}
26     point operator-=(const point & p){*this = *this - p; return *this;}
27     point operator*=(const ld & p){*this = *this * p; return *this;}
28     point operator/=(const ld & p){*this = *this / p; return *this;}
29
30     point rotate(const ld & a) const{return point(x*cos(a) - y*sin(a), x*
31         sin(a) + y*cos(a));}

```

```

29     point perp() const{return point(-y, x);}
30     ld ang() const{
31         ld a = atan2l(y, x); a += le(a, 0) ? 2*pi : 0; return a;
32     }
33     ld dot(const point & p) const{return x * p.x + y * p.y;}
34     ld cross(const point & p) const{return x * p.y - y * p.x;}
35     ld norm() const{return x * x + y * y;}
36     ld length() const{return sqrtl(x * x + y * y);}
37     point unit() const{return (*this) / length();}
38
39     bool operator==(const point & p) const{return eq(x, p.x) && eq(y, p.y)
40     ;}
41     bool operator!=(const point & p) const{return !(*this == p);}
42     bool operator<(const point & p) const{return le(x, p.x) || (eq(x, p.x)
43         && le(y, p.y));}
44     bool operator>(const point & p) const{return ge(x, p.x) || (eq(x, p.x)
45         && ge(y, p.y));}
46     bool half(const point & p) const{return le(p.cross(*this), 0) || (eq(p
47         .cross(*this), 0) && le(p.dot(*this), 0));}
48 };
49 istream &operator>>(istream &is, point & p){return is >> p.x >> p.y;}
50 ostream &operator<<(ostream &os, const point & p){return os << "(" << p.
51     x << ", " << p.y << ")";}
52
53 int sgn(ld x){
54     if(ge(x, 0)) return 1;
55     if(le(x, 0)) return -1;
56     return 0;
57 }
58
59 void polarSort(vector<point> & P, const point & o, const point & v){
60     //sort points in P around o, taking the direction of v as first angle
61     sort(P.begin(), P.end(), [&](const point & a, const point & b){
62         return point((a - o).half(v), 0) < point((b - o).half(v), (a - o).
63             cross(b - o));
64     });
65 }

```

6.4 Find Triangle With Given Area

```

1 #include <algorithm>
2 #include <cassert>

```

```

3 #include <iostream>
4 #include <vector>
5 using namespace std;
6
7 struct point {
8     int x, y;
9     int index;
10
11    point(int _x = 0, int _y = 0) : x(_x), y(_y) {
12        index = -1;
13    }
14
15    point& operator+=(const point &other) {
16        x += other.x;
17        y += other.y;
18        return *this;
19    }
20
21    point& operator-=(const point &other) {
22        x -= other.x;
23        y -= other.y;
24        return *this;
25    }
26
27    point operator+(const point &other) const {
28        return point(*this) += other;
29    }
30
31    point operator-(const point &other) const {
32        return point(*this) -= other;
33    }
34
35    bool operator<(const point &other) const {
36        return make_pair(y, x) > make_pair(other.y, other.x);
37    }
38};
39
40 long long cross(const point &a, const point &b) {
41    return (long long) a.x * b.y - (long long) b.x * a.y;
42}
43
44 long long triangle_area(const point &a, const point &b, const point &c)
{
45    return abs(cross(a, b) + cross(b, c) + cross(c, a));
46}
47
48 struct segment {
49     point p;
50     int index1, index2;
51
52    segment(const point &p, int _index1, int _index2) : p(p), index1(
53        _index1), index2(_index2) {}
54
55    bool operator<(const segment &other) const {
56        return cross(p, other.p) > 0;
57    }
58
59    int N;
60    long long S;
61    vector<point> points;
62    vector<segment> segments;
63    vector<int> location;
64
65    void yes(point a, point b, point c) {
66        cout << "Yes" << '\n';
67        cout << a.x << ' ' << a.y << '\n';
68        cout << b.x << ' ' << b.y << '\n';
69        cout << c.x << ' ' << c.y << '\n';
70        assert(triangle_area(a, b, c) == 2 * S);
71    }
72
73    int main() {
74        ios::sync_with_stdio(false);
75        cin.tie(nullptr);
76
77        cin >> N >> S;
78        points.resize(N);
79
80        for (int i = 0; i < N; i++)
81            cin >> points[i].x >> points[i].y;
82
83        sort(points.begin(), points.end());
84        location.resize(N);
85
86        for (int i = 0; i < N; i++) {

```

```

87     points[i].index = i;
88     location[i] = i;
89 }
90
91 for (int i = 0; i < N; i++)
92     for (int j = 0; j < i; j++)
93         segments.emplace_back(points[j] - points[i], i, j);
94
95 sort(segments.begin(), segments.end());
96
97 for (segment &s : segments) {
98     int &a = location[s.index1];
99     int &b = location[s.index2];
100    assert(b == a - 1);
101
102    int low = 0, high = b - 1;
103
104    while (low <= high) {
105        int mid = (low + high) / 2;
106        long long area = triangle_area(points[mid], points[a]
107                                         points[b]);
108
109        if (area == 2 * S) {
110            yes(points[mid], points[a], points[b]);
111            return 0;
112        }
113
114        if (area > 2 * S)
115            low = mid + 1;
116        else
117            high = mid - 1;
118    }
119
120    low = a + 1;
121    high = N - 1;
122
123    while (low <= high) {
124        int mid = (low + high) / 2;
125        long long area = triangle_area(points[mid], points[a]
126                                         points[b]);
127
128        if (area == 2 * S) {
129            yes(points[mid], points[a], points[b]);
130        }

```

```

128         return 0;
129     }
130
131     if (area > 2 * S)
132         high = mid - 1;
133     else
134         low = mid + 1;
135     }
136
137     swap(points[a], points[b])
138     swap(a, b);
139 }
140
141 cout << "No" << '\n';
142 return 0;
143 }
```

6.5 Antipodal Pairs

```

1 vector<pair<int, int>> antipodalPairs(vector<point> & P){
2     vector<pair<int, int>> ans;
3     int n = P.size(), k = 1;
4     auto f = [&](int u, int v, int w){return abs((P[v%n]-P[u%n]).cross(P[w%n]-P[u%n]));};
5     while(ge(f(n-1, 0, k+1), f(n-1, 0, k))) ++k;
6     for(int i = 0, j = k; i <= k && j < n; ++i){
7         ans.emplace_back(i, j);
8         while(j < n-1 && ge(f(i, i+1, j+1), f(i, i+1, j)))
9             ans.emplace_back(i, ++j);
10    }
11    return ans;
12 }
```

6.6 Area and Perimeter

```
1 ld perimeter(vector<point> & P){  
2     int n = P.size();  
3     ld ans = 0;  
4     for(int i = 0; i < n; i++){  
5         ans += (P[i] - P[(i + 1) % n]).length();  
6     }  
7     return ans;  
8 }
```

```

10 ld area(vector<point> & P){
11     int n = P.size();
12     ld ans = 0;
13     for(int i = 0; i < n; i++){
14         ans += P[i].cross(P[(i + 1) % n]);
15     }
16     return abs(ans / 2);
17 }
```

6.7 Area Union Circles

```

1 struct circ{
2     point c;
3     ld r;
4     circ() {}
5     circ(const point & c, ld r): c(c), r(r) {}
6     set<pair<ld, ld>> ranges;
7
8     void disable(ld l, ld r){
9         ranges.emplace(l, r);
10    }
11
12    auto getActive() const{
13        vector<pair<ld, ld>> ans;
14        ld maxi = 0;
15        for(const auto & dis : ranges){
16            ld l, r;
17            tie(l, r) = dis;
18            if(l > maxi){
19                ans.emplace_back(maxi, l);
20            }
21            maxi = max(maxi, r);
22        }
23        if(!eq(maxi, 2*pi)){
24            ans.emplace_back(maxi, 2*pi);
25        }
26        return ans;
27    }
28};

29 ld areaUnionCircles(const vector<circ> & circs){
30     vector<circ> valid;
31     for(const circ & curr : circs){
```

```

33     if(eq(curr.r, 0)) continue;
34     circ nuevo = curr;
35     for(circ & prev : valid){
36         if(circleInsideCircle(prev.c, prev.r, nuevo.c, nuevo.r)){
37             nuevo.disable(0, 2*pi);
38         }else if(circleInsideCircle(nuevo.c, nuevo.r, prev.c, prev.r)){
39             prev.disable(0, 2*pi);
40         }else{
41             auto cruce = intersectionCircles(prev.c, prev.r, nuevo.c, nuevo.r);
42             if(cruce.size() == 2){
43                 ld a1 = (cruce[0] - prev.c).ang();
44                 ld a2 = (cruce[1] - prev.c).ang();
45                 ld b1 = (cruce[1] - nuevo.c).ang();
46                 ld b2 = (cruce[0] - nuevo.c).ang();
47                 if(a1 < a2){
48                     prev.disable(a1, a2);
49                 }else{
50                     prev.disable(a1, 2*pi);
51                     prev.disable(0, a2);
52                 }
53                 if(b1 < b2){
54                     nuevo.disable(b1, b2);
55                 }else{
56                     nuevo.disable(b1, 2*pi);
57                     nuevo.disable(0, b2);
58                 }
59             }
60         }
61     }
62     valido.push_back(nuevo);
63 }
64 ld ans = 0;
65 for(const circ & curr : valid){
66     for(const auto & range : curr.getActive()){
67         ld l, r;
68         tie(l, r) = range;
69         ans += curr.r*(curr.c.x * (sin(r) - sin(l)) - curr.c.y * (cos(r) -
70             cos(l))) + curr.r*curr.r*(r-l);
71     }
72 }
73 return ans/2;
74 };
```

6.8 Centroid

```

1 point centroid(vector<point> & P){
2     point num;
3     ld den = 0;
4     int n = P.size();
5     for(int i = 0; i < n; ++i){
6         ld cross = P[i].cross(P[(i + 1) % n]);
7         num += (P[i] + P[(i + 1) % n]) * cross;
8         den += cross;
9     }
10    return num / (3 * den);
11}

```

6.9 Circle Inside Circle

```

1 int circleInsideCircle(const point & c1, ld r1, const point & c2, ld r2)
2 {
3     //test if circle 2 is inside circle 1
4     //returns "-1" if 2 touches internally 1, "1" if 2 is inside 1, "0" if
5     //they overlap
6     ld l = r1 - r2 - (c1 - c2).length();
7     return (ge(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
8 }

```

6.10 Circle Outside Circle

```

1 int circleOutsideCircle(const point & c1, ld r1, const point & c2, ld r2)
2 {
3     //test if circle 2 is outside circle 1
4     //returns "-1" if they touch externally, "1" if 2 is outside 1, "0" if
5     //they overlap
6     ld l = (c1 - c2).length() - (r1 + r2);
7     return (ge(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
8 }

```

6.11 Closest Pair of Points

```

1 bool comp1(const point & a, const point & b){
2     return le(a.y, b.y);
3 }
4 pair<point, point> closestPairOfPoints(vector<point> P){
5     sort(P.begin(), P.end(), comp1);
6     set<point> S;

```

```

7     ld ans = inf;
8     point p, q;
9     int pos = 0;
10    for(int i = 0; i < P.size(); ++i){
11        while(pos < i && geq(P[i].y - P[pos].y, ans)){
12            S.erase(P[pos++]);
13        }
14        auto lower = S.lower_bound({P[i].x - ans - eps, -inf});
15        auto upper = S.upper_bound({P[i].x + ans + eps, -inf});
16        for(auto it = lower; it != upper; ++it){
17            ld d = (P[i] - *it).length();
18            if(le(d, ans)){
19                ans = d;
20                p = P[i];
21                q = *it;
22            }
23        }
24        S.insert(P[i]);
25    }
26    return {p, q};
27 }

```

6.12 Common Tangents

```

1 vector<vector<point>> tangents(const point & c1, ld r1, const point & c2
2 , ld r2, bool inner){
3     //returns a vector of segments or a single point
4     if(inner) r2 = -r2;
5     point d = c2 - c1;
6     ld dr = r1 - r2, d2 = d.norm(), h2 = d2 - dr*dr;
7     if(eq(d2, 0) || le(h2, 0)) return {};
8     point v = d*dr/d2;
9     if(eq(h2, 0)) return {{c1 + v*r1}};
10    else{
11        point u = d.perp()*sqrt(h2)/d2;
12        return {{c1 + (v - u)*r1, c2 + (v - u)*r2}, {c1 + (v + u)*r1, c2 +
13        v + u)*r2}};
14    }

```

6.13 Convex Hull

```

1 vector<point> convexHull(vector<point> P){
2     sort(P.begin(), P.end());

```

```

3   vector<point> L, U;
4   for(int i = 0; i < P.size(); i++){
5     while(L.size() >= 2 && leq((L[L.size() - 2] - P[i]).cross(L[L.size()
6       - 1] - P[i]), 0)){
7       L.pop_back();
8     }
9     L.push_back(P[i]);
10  }
11  for(int i = P.size() - 1; i >= 0; i--){
12    while(U.size() >= 2 && leq((U[U.size() - 2] - P[i]).cross(U[U.size()
13      - 1] - P[i]), 0)){
14      U.pop_back();
15    }
16    U.push_back(P[i]);
17  }
18  L.pop_back();
19  U.pop_back();
20  L.insert(L.end(), U.begin(), U.end());
21  return L;
}

```

6.14 Segment Intersection with Ray

```

1 bool crossesRay(const point & a, const point & b, const point & p){
2   return (geq(b.y, p.y) - geq(a.y, p.y)) * sgn((a - p).cross(b - p)) >
3     0;
}

```

6.15 Cut Polygon

```

1 vector<point> cutPolygon(const vector<point> & P, const point & a, const
2   point & v){
3   //returns the part of the convex polygon P on the left side of line a+
4   //v
5   int n = P.size();
6   vector<point> lhs;
7   for(int i = 0; i < n; ++i){
8     if(geq(v.cross(P[i] - a), 0)){
9       lhs.push_back(P[i]);
10    }
11    if(intersectLineSegmentInfo(a, v, P[i], P[(i+1)%n]) == 1){
12      point p = intersectLines(a, v, P[i], P[(i+1)%n] - P[i]);
13      if(p != P[i] && p != P[(i+1)%n]){
14        lhs.push_back(p);
15      }
16    }
17  }
18  return lhs;
}

```

```

13   }
14   }
15 }
16 return lhs;
17 }

```

6.16 Delaunay Triangulation

```

1 //Delaunay triangulation in O(n log n)
2 const point inf_pt(inf, inf);
3
4 struct QuadEdge{
5   point origin;
6   QuadEdge* rot = nullptr;
7   QuadEdge* onext = nullptr;
8   bool used = false;
9   QuadEdge* rev() const{return rot->rot;}
10  QuadEdge* lnext() const{return rot->rev()->onext->rot;}
11  QuadEdge* oprev() const{return rot->onext->rot;}
12  point dest() const{return rev()->origin;}
13 };
14
15 QuadEdge* make_edge(const point & from, const point & to){
16   QuadEdge* e1 = new QuadEdge;
17   QuadEdge* e2 = new QuadEdge;
18   QuadEdge* e3 = new QuadEdge;
19   QuadEdge* e4 = new QuadEdge;
20   e1->origin = from;
21   e2->origin = to;
22   e3->origin = e4->origin = inf_pt;
23   e1->rot = e3;
24   e2->rot = e4;
25   e3->rot = e2;
26   e4->rot = e1;
27   e1->onext = e1;
28   e2->onext = e2;
29   e3->onext = e4;
30   e4->onext = e3;
31   return e1;
32 }
33
34 void splice(QuadEdge* a, QuadEdge* b){
35   swap(a->onext->rot->onext, b->onext->rot->onext);
}

```

```

36     swap(a->onext, b->onext);
37 }
38
39 void delete_edge(QuadEdge* e){
40     splice(e, e->oprev());
41     splice(e->rev(), e->rev()->oprev());
42     delete e->rot;
43     delete e->rev()->rot;
44     delete e;
45     delete e->rev();
46 }
47
48 QuadEdge* connect(QuadEdge* a, QuadEdge* b){
49     QuadEdge* e = make_edge(a->dest(), b->origin);
50     splice(e, a->lnext());
51     splice(e->rev(), b);
52     return e;
53 }
54
55 bool left_of(const point & p, QuadEdge* e){
56     return ge((e->origin - p).cross(e->dest() - p), 0);
57 }
58
59 bool right_of(const point & p, QuadEdge* e){
60     return le((e->origin - p).cross(e->dest() - p), 0);
61 }
62
63 ld det3(ld a1, ld a2, ld a3, ld b1, ld b2, ld b3, ld c1, ld c2, ld c3) {
64     return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) + a3 * (b1
65         * c2 - c1 * b2);
66 }
67
68 bool in_circle(const point & a, const point & b, const point & c, const
69     point & d) {
70     ld det = -det3(b.x, b.y, b.norm(), c.x, c.y, c.norm(), d.x, d.y, d.
71         norm());
72     det += det3(a.x, a.y, a.norm(), c.x, c.y, c.norm(), d.x, d.y, d.norm()
73         );
74     det -= det3(a.x, a.y, a.norm(), b.x, b.y, b.norm(), d.x, d.y, d.norm()
75         );
76     det += det3(a.x, a.y, a.norm(), b.x, b.y, b.norm(), c.x, c.y, c.norm()
77         );
78     return ge(det, 0);
79 }
80
81 pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<point> & P){
82     if(r - l + 1 == 2){
83         QuadEdge* res = make_edge(P[l], P[r]);
84         return {res, res->rev()};
85     }
86     if(r - l + 1 == 3){
87         QuadEdge *a = make_edge(P[l], P[l + 1]), *b = make_edge(P[l + 1], P[
88             r]);
89         splice(a->rev(), b);
90         int sg = sgn((P[l + 1] - P[l]).cross(P[r] - P[l]));
91         if(sg == 0)
92             return {a, b->rev()};
93         QuadEdge* c = connect(b, a);
94         if(sg == 1)
95             return {a, b->rev()};
96         else
97             return {c->rev(), c};
98     }
99     int mid = (l + r) / 2;
100    QuadEdge *ldo, *ldi, *rdo, *rdi;
101    tie(ldo, ldi) = build_tr(l, mid, P);
102    tie(rdi, rdo) = build_tr(mid + 1, r, P);
103    while(true){
104        if(left_of(rdi->origin, ldi)){
105            ldi = ldi->lnext();
106            continue;
107        }
108        if(right_of(ldi->origin, rdi)){
109            rdi = rdi->rev()->onext;
110            continue;
111        }
112        break;
113    }
114    QuadEdge* basel = connect(rdi->rev(), ldi);
115    auto valid = [&basel](QuadEdge* e){return right_of(e->dest(), basel)
116       ;};
117    if(ldi->origin == ldo->origin)
118        ldo = basel->rev();
119    if(rdi->origin == rdo->origin)
120        rdo = basel;
121    while(true){
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
823
824
825
825
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
```

```

114     QuadEdge* lcand = basel->rev()->onext;
115     if(valid(lcand)){
116         while(in_circle(basel->dest(), basel->origin, lcand->dest(), lcand
117             ->onext->dest())){
118             QuadEdge* t = lcand->onext;
119             delete_edge(lcand);
120             lcand = t;
121         }
122     }
123     QuadEdge* rcand = basel->oprev();
124     if(valid(rcand)){
125         while(in_circle(basel->dest(), basel->origin, rcand->dest(), rcand
126             ->oprev()->dest())){
127             QuadEdge* t = rcand->oprev();
128             delete_edge(rcand);
129             rcand = t;
130         }
131         if(!valid(lcand) && !valid(rcand))
132             break;
133         if(!valid(lcand) || (valid(rcand) && in_circle(lcand->dest(), lcand
134             ->origin, rcand->origin, rcand->dest())))
135             basel = connect(rcand, basel->rev());
136         else
137             basel = connect(basel->rev(), lcand->rev());
138     }
139     return {ldo, rdo};
140 }
141 vector<tuple<point, point, point>> delaunay(vector<point> & P){
142     sort(P.begin(), P.end());
143     auto res = build_tr(0, (int)P.size() - 1, P);
144     QuadEdge* e = res.first;
145     vector<QuadEdge*> edges = {e};
146     while((e->dest() - e->onext->dest()).cross(e->origin - e->onext->
147         dest(), 0))
148         e = e->onext;
149     auto add = [&P, &e, &edges](){
150         QuadEdge* curr = e;
151         do{
152             curr->used = true;
153             P.push_back(curr->origin);
154             edges.push_back(curr->rev());
155         }
156     };
157 }
158 
```

```

153     curr = curr->lnext();
154     }while(curr != e);
155 };
156 add();
157 P.clear();
158 int kek = 0;
159 while(kek < (int)edges.size())
160     if(!(e = edges[kek++])->used)
161         add();
162 vector<tuple<point, point, point>> ans;
163 for(int i = 0; i < (int)P.size(); i += 3){
164     ans.emplace_back(P[i], P[i + 1], P[i + 2]);
165 }
166 return ans;
167 }
```

6.17 Diameter and Width

```

1 pair<ld, ld> diameterAndWidth(vector<point> & P){
2     int n = P.size(), k = 0;
3     auto dot = [&](int a, int b){return (P[(a+1)%n]-P[a]).dot(P[(b+1)%n]-P
4         [b]);};
5     auto cross = [&](int a, int b){return (P[(a+1)%n]-P[a]).cross(P[(b+1)%n]-P[b]);};
6     ld diameter = 0;
7     ld width = inf;
8     while(ge(dot(0, k), 0)) k = (k+1) % n;
9     for(int i = 0; i < n; ++i){
10        while(ge(cross(i, k), 0)) k = (k+1) % n;
11        //pair: (i, k)
12        diameter = max(diameter, (P[k] - P[i]).length());
13        width = min(width, distancePointLine(P[i], P[(i+1)%n] - P[i], P[k]));
14    }
15 }
```

6.18 Distance Between Point and Circle

```

1 ld distancePointCircle(const point & c, ld r, const point & p){
2     //point p, circle with center c and radius r
3     return max((ld)0, (p - c).length() - r);
4 }
```

6.19 Distance Between Point and Line

```

1 ld distancePointLine(const point & a, const point & v, const point & p){
2     //line: a + tv, point p
3     return abs(v.cross(p - a)) / v.length();
4 }
```

6.20 Example of Geometry

```

1 int main(){
2     /*vector<pair<point, point>> centers = {{point(-2, 5), point(-8, -7)},
3         {point(14, 4), point(18, 6)}, {point(9, 20), point(9, 28)},
4             {point(21, 20), point(21, 29)}, {point(8, -10),
5                 point(14, -10)}, {point(24, -6), point(34, -6)
6                     },
7             {point(34, 8), point(36, 9)}, {point(50, 20),
8                 point(56, 24.5)}};
9     vector<pair<ld, ld>> radii = {{7, 4}, {3, 5}, {4, 4}, {4, 5}, {3, 3},
10         {4, 6}, {5, 1}, {10, 2.5}};
11     int n = centers.size();
12     for(int i = 0; i < n; ++i){
13         cout << "\n" << centers[i].first << " " << radii[i].first << " "
14             << centers[i].second << " " << radii[i].second << "\n";
15         auto extLines = tangents(centers[i].first, radii[i].first, centers[i]
16             ].second, radii[i].second, false);
17         cout << "Exterior tangents:\n";
18         for(auto par : extLines){
19             for(auto p : par){
20                 cout << p << " ";
21             }
22             cout << "\n";
23         }
24         auto intLines = tangents(centers[i].first, radii[i].first, centers[i]
25             ].second, radii[i].second, true);
26         cout << "Interior tangents:\n";
27         for(auto par : intLines){
28             for(auto p : par){
29                 cout << p << " ";
30             }
31             cout << "\n";
32         }
33     }*/
34 }
```

```

27     /*int n;
28     cin >> n;
29     vector<point> P(n);
30     for(auto & p : P) cin >> p;
31     auto triangulation = delaunay(P);
32     for(auto triangle : triangulation){
33         cout << get<0>(triangle) << " " << get<1>(triangle) << " " << get
34             << 2>(triangle) << "\n";
35     }*/
36 
37     /*int n;
38     cin >> n;
39     vector<point> P(n);
40     for(auto & p : P) cin >> p;
41     auto ans = smallestEnclosingCircle(P);
42     cout << ans.first << " " << ans.second << "\n";*/
43 
44     /*vector<point> P;
45     srand(time(0));
46     for(int i = 0; i < 1000; ++i){
47         P.emplace_back(rand() % 1000000000, rand() % 1000000000);
48     }
49     point o(rand() % 1000000000, rand() % 1000000000), v(rand() %
50         1000000000, rand() % 1000000000);
51     polarSort(P, o, v);
52     auto ang = [&](point p){
53         ld th = atan2(p.y, p.x);
54         if(th < 0) th += acosl(-1)*2;
55         ld t = atan2(v.y, v.x);
56         if(t < 0) t += acosl(-1)*2;
57         if(th < t) th += acosl(-1)*2;
58         return th;
59     };
59     for(int i = 0; i < P.size()-1; ++i){
60         assert(leq(ang(P[i] - o), ang(P[i+1] - o)));
61     }*/
62 }
```

6.21 Half Plane Intersection

```

1 struct plane{
2     point a, v;
```

```

3  plane(): a(), v(){}
4  plane(const point& a, const point& v): a(a), v(v){}
5
6  point intersect(const plane& p) const{
7      ld t = (p.a - a).cross(p.v) / v.cross(p.v);
8      return a + v*t;
9  }
10
11 bool outside(const point& p) const{ // test if point p is strictly
12     outside
13     return le(v.cross(p - a), 0);
14 }
15
16 bool inside(const point& p) const{ // test if point p is inside or in
17     the boundary
18     return geq(v.cross(p - a), 0);
19 }
20
21 bool operator<(const plane& p) const{ // sort by angle
22     auto lhs = make_tuple(v.half({1, 0}), ld(0), v.cross(p.a - a));
23     auto rhs = make_tuple(p.v.half({1, 0}), v.cross(p.v), ld(0));
24     return lhs < rhs;
25 }
26
27 bool operator==(const plane& p) const{ // paralell and same directions
28     , not really equal
29     return eq(v.cross(p.v), 0) && ge(v.dot(p.v), 0);
30 }
31
32 vector<point> halfPlaneIntersection(vector<plane> planes){
33     planes.push_back({{0, -inf}, {1, 0}});
34     planes.push_back({{inf, 0}, {0, 1}});
35     planes.push_back({{0, inf}, {-1, 0}});
36     planes.push_back({{-inf, 0}, {0, -1}});
37     sort(planes.begin(), planes.end());
38     planes.erase(unique(planes.begin(), planes.end()), planes.end());
39     deque<plane> ch;
40     deque<point> poly;
41     for(const plane& p : planes){
42         while(ch.size() >= 2 && p.outside(poly.back())) ch.pop_back(), poly.
43             pop_back();
44         while(ch.size() >= 2 && p.outside(poly.front())) ch.pop_front(), poly.
45             pop_front();
46     }
47 }
```

```

42     poly.pop_front();
43     if(p.v.half({1, 0}) && poly.empty()) return {};
44     ch.push_back(p);
45     if(ch.size() >= 2) poly.push_back(ch[ch.size()-2].intersect(ch[ch.size()-1]));
46   }
47   while(ch.size() >= 3 && ch.front().outside(poly.back())) ch.pop_back()
48     , poly.pop_back();
49   while(ch.size() >= 3 && ch.back().outside(poly.front())) ch.pop_front
50     (), poly.pop_front();
51   poly.push_back(ch.back().intersect(ch.front()));
52   return vector<point>(poly.begin(), poly.end());
53 }

54 vector<point> halfPlaneIntersectionRandomized(vector<plane> planes){
55   point p = planes[0].a;
56   int n = planes.size();
57   random_shuffle(planes.begin(), planes.end());
58   for(int i = 0; i < n; ++i){
59     if(planes[i].inside(p)) continue;
60     ld lo = -inf, hi = inf;
61     for(int j = 0; j < i; ++j){
62       ld A = planes[j].v.cross(planes[i].v);
63       ld B = planes[j].v.cross(planes[j].a - planes[i].a);
64       if(ge(A, 0)){
65         lo = max(lo, B/A);
66       }else if(le(A, 0)){
67         hi = min(hi, B/A);
68       }else{
69         if(ge(B, 0)) return {};
70       }
71       if(ge(lo, hi)) return {};
72     }
73     p = planes[i].a + planes[i].v*lo;
74   }
75   return {p};
76 }
```

6.22 Incircle

```
1 | pair<point, ld> getCircle(const point & m, const point & n, const point
|   & p){
2 |   //find circle that passes through points p, q, r
```

```

3   point c = intersectLines((n + m) / 2, (n - m).perp(), (p + n) / 2, (p
4     - n).perp());
5   ld r = (c - m).length();
6   return {c, r};
}

```

6.23 Intersection of Two Circles

```

1 vector<point> intersectionCircles(const point & c1, ld r1, const point &
2   c2, ld r2){
3   //circle 1 with center c1 and radius r1
4   //circle 2 with center c2 and radius r2
5   point d = c2 - c1;
6   ld d2 = d.norm();
7   if(eq(d2, 0)) return {};//concentric circles
8   ld pd = (d2 + r1*r1 - r2*r2) / 2;
9   ld h2 = r1*r1 - pd*pd/d2;
10  point p = c1 + d*pd/d2;
11  if(eq(h2, 0)) return {p}; //circles touch at one point
12  else if(le(h2, 0)) return {};//circles don't intersect
13  else{
14    point u = d.perp() * sqrt(h2/d2);
15    return {p - u, p + u};
16  }
}

```

6.24 Intersection Line and Circle

```

1 vector<point> intersectLineCircle(const point & a, const point & v,
2   const point & c, ld r){
3   //line a+tv, circle with center c and radius r
4   ld h2 = r*r - v.cross(c - a) * v.cross(c - a) / v.norm();
5   point p = a + v * v.dot(c - a) / v.norm();
6   if(eq(h2, 0)) return {p}; //line tangent to circle
7   else if(le(h2, 0)) return {};//no intersection
8   else{
9     point u = v.unit() * sqrt(h2);
10    return {p - u, p + u}; //two points of intersection (chord)
11  }
}

```

6.25 Intersection Polygon and Circle

```

1 ld signed_angle(const point & a, const point & b){

```

```

2   return sgn(a.cross(b)) * acosl(a.dot(b) / (a.length() * b.length()));
3 }
4
5 ld intersectPolygonCircle(const vector<point> & P, const point & c, ld r
6   ){
7   //Gets the area of the intersection of the polygon with the circle
8   int n = P.size();
9   ld ans = 0;
10  for(int i = 0; i < n; ++i){
11    point p = P[i], q = P[(i+1)%n];
12    bool p_inside = (pointInCircle(c, r, p) != 0);
13    bool q_inside = (pointInCircle(c, r, q) != 0);
14    if(p_inside && q_inside){
15      ans += (p - c).cross(q - c);
16    }else if(p_inside && !q_inside){
17      point s1 = intersectSegmentCircle(p, q, c, r)[0];
18      point s2 = intersectSegmentCircle(c, q, c, r)[0];
19      ans += (p - c).cross(s1 - c) + r*r * signed_angle(s1 - c, s2 - c);
20    }else if(!p_inside && q_inside){
21      point s1 = intersectSegmentCircle(c, p, c, r)[0];
22      point s2 = intersectSegmentCircle(p, q, c, r)[0];
23      ans += (s2 - c).cross(q - c) + r*r * signed_angle(s1 - c, s2 - c);
24    }else{
25      auto info = intersectSegmentCircle(p, q, c, r);
26      if(info.size() <= 1){
27        ans += r*r * signed_angle(p - c, q - c);
28      }else{
29        point s2 = info[0], s3 = info[1];
30        point s1 = intersectSegmentCircle(c, p, c, r)[0];
31        point s4 = intersectSegmentCircle(c, q, c, r)[0];
32        ans += (s2 - c).cross(s3 - c) + r*r * (signed_angle(s1 - c, s2 -
33          c) + signed_angle(s3 - c, s4 - c));
34      }
35    }
36  }
  return abs(ans)/2;
}

```

6.26 Intersection Segment and Circle

```

1 vector<point> intersectSegmentCircle(const point & a, const point & b,
2   const point & c, ld r){
3   //segment ab, circle with center c and radius r

```

```

3   vector<point> P = intersectLineCircle(a, b - a, c, r), ans;
4   for(const point & p : P){
5     if(pointInSegment(a, b, p)) ans.push_back(p);
6   }
7   return ans;
8 }
```

6.27 Line Intersection

```

1 int intersectLinesInfo(const point & a1, const point & v1, const point &
2   a2, const point & v2){
3   //lines a1+tv1 and a2+tv2
4   ld det = v1.cross(v2);
5   if(eq(det, 0)){
6     if(eq((a2 - a1).cross(v1), 0)){
7       return -1; //infinity points
8     }else{
9       return 0; //no points
10    }
11  }else{
12    return 1; //single point
13  }
14 }
15 point intersectLines(const point & a1, const point & v1, const point &
16   a2, const point & v2){
17   //lines a1+tv1, a2+tv2
18   //assuming that they intersect
19   ld det = v1.cross(v2);
20   return a1 + v1 * ((a2 - a1).cross(v2) / det);
21 }
```

6.28 Minkowski Sum

```

1 vector<point> minkowskiSum(vector<point> A, vector<point> B){
2   int na = (int)A.size(), nb = (int)B.size();
3   if(A.empty() || B.empty()) return {};
4
5   rotate(A.begin(), min_element(A.begin(), A.end()), A.end());
6   rotate(B.begin(), min_element(B.begin(), B.end()), B.end());
7
8   int pa = 0, pb = 0;
9   vector<point> M;
```

```

11  while(pa < na && pb < nb){
12    M.push_back(A[pa] + B[pb]);
13    ld x = (A[(pa + 1) % na] - A[pa]).cross(B[(pb + 1) % nb] - B[pb]);
14    if(leq(x, 0)) pb++;
15    if(geq(x, 0)) pa++;
16  }
17
18  while(pa < na) M.push_back(A[pa++] + B[0]);
19  while(pb < nb) M.push_back(B[pb++] + A[0]);
20
21  return M;
22 }
```

6.29 Point in Circle

```

1 int pointInCircle(const point & c, ld r, const point & p){
2   //test if point p is inside the circle with center c and radius r
3   //returns "0" if it's outside, "-1" if it's in the perimeter, "1" if
4   //it's inside
5   ld l = (p - c).length() - r;
6   return (le(l, 0) ? 1 : (eq(l, 0) ? -1 : 0));
7 }
```

6.30 Point in Convex Hull

```

1 //point in convex polygon in O(log n)
2 //make sure that P is convex and in ccw
3 //before the queries, do the preprocess on P:
4 // rotate(P.begin(), min_element(P.begin(), P.end()), P.end());
5 // int right = max_element(P.begin(), P.end()) - P.begin();
6 //returns 0 if p is outside, 1 if p is inside, -1 if p is in the
7 //perimeter
8 int pointInConvexPolygon(const vector<point> & P, const point & p, int
9   right){
10  if(p < P[0] || P[right] < p) return 0;
11  int orientation = sgn((P[right] - P[0]).cross(p - P[0]));
12  if(orientation == 0){
13    if(p == P[0] || p == P[right]) return -1;
14    return (right == 1 || right + 1 == P.size()) ? -1 : 1;
15  }else if(orientation < 0){
16    auto r = lower_bound(P.begin() + 1, P.begin() + right, p);
17    int det = sgn((p - r[-1]).cross(r[0] - r[-1])) - 1;
18    if(det == -2) det = 1;
19    return det;
20 }
```

```

18 }else{
19     auto l = upper_bound(P.rbegin(), P.rend() - right - 1, p);
20     int det = sgn((p - l[0]).cross((l == P.rbegin() ? P[0] : l[-1]) - l
21         [0])) - 1;
22     if(det == -2) det = 1;
23     return det;
24 }

```

6.31 Point in Line

```

1 bool pointInLine(const point & a, const point & v, const point & p){
2     //line a+tv, point p
3     return eq((p - a).cross(v), 0);
4 }

```

6.32 Point in Perimeter

```

1 bool pointInPerimeter(const vector<point> & P, const point & p){
2     int n = P.size();
3     for(int i = 0; i < n; i++){
4         if(pointInSegment(P[i], P[(i + 1) % n], p)){
5             return true;
6         }
7     }
8     return false;
9 }

```

6.33 Point in Polygon

```

1 int pointInPolygon(const vector<point> & P, const point & p){
2     if(pointInPerimeter(P, p)){
3         return -1; //point in the perimeter
4     }
5     int n = P.size();
6     int rays = 0;
7     for(int i = 0; i < n; i++){
8         rays += crossesRay(P[i], P[(i + 1) % n], p);
9     }
10    return rays & 1; //0: point outside, 1: point inside
11 }

```

6.34 Point in Segment

```

1 bool pointInSegment(const point & a, const point & b, const point & p){
2     //segment ab, point p
3     return pointInLine(a, b - a, p) && leq((a - p).dot(b - p), 0);
4 }

```

6.35 Points Tangency

```

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

```

6.36 Projection Point Circle

```

1 point projectionPointCircle(const point & c, ld r, const point & p){
2     //point p (outside the circle), circle with center c and radius r
3     return c + (p - c).unit() * r;
4 }

```

6.37 Segment Intersection

```

1 int intersectLineSegmentInfo(const point & a, const point & v, const
2     point & c, const point & d){
3     //line a+tv, segment cd
4     point v2 = d - c;
5     ld det = v.cross(v2);
6     if(eq(det, 0)){
7         if(eq((c - a).cross(v), 0)){
8             return -1; //infinity points
9         }
10    }else{
11        return 0; //no point
12    }
13 }else{
14     return sgn(v.cross(c - a)) != sgn(v.cross(d - a)); //1: single point
15     , 0: no point
16 }
17
18 int intersectSegmentsInfo(const point & a, const point & b, const point &
19     c, const point & d){
20 }

```

```

17 //segment ab, segment cd
18 point v1 = b - a, v2 = d - c;
19 int t = sgn(v1.cross(c - a)), u = sgn(v1.cross(d - a));
20 if(t == u){
21     if(t == 0){
22         if(pointInSegment(a, b, c) || pointInSegment(a, b, d) ||
23             pointInSegment(c, d, a) || pointInSegment(c, d, b)){
24             return -1; //infinity points
25         }else{
26             return 0; //no point
27         }
28     }else{
29         return 0; //no point
30     }
31 }else{
32     return sgn(v2.cross(a - c)) != sgn(v2.cross(b - c)); //1: single
33     point, 0: no point
}

```

6.38 Smallest Enclosing Circle

```

1 pair<point, ld> mec2(vector<point> & S, const point & a, const point & b
, int n){
2     ld hi = inf, lo = -hi;
3     for(int i = 0; i < n; ++i){
4         ld si = (b - a).cross(S[i] - a);
5         if(eq(si, 0)) continue;
6         point m = getCircle(a, b, S[i]).first;
7         ld cr = (b - a).cross(m - a);
8         if(le(si, 0)) hi = min(hi, cr);
9         else lo = max(lo, cr);
10    }
11    ld v = (ge(lo, 0) ? lo : le(hi, 0) ? hi : 0);
12    point c = (a + b) / 2 + (b - a).perp() * v / (b - a).norm();
13    return {c, (a - c).norm()};
14 }
15
16 pair<point, ld> mec(vector<point> & S, const point & a, int n){
17     random_shuffle(S.begin(), S.begin() + n);
18     point b = S[0], c = (a + b) / 2;
19     ld r = (a - c).norm();
20     for(int i = 1; i < n; ++i){

```

```

21     if(ge((S[i] - c).norm(), r)){
22         tie(c, r) = (n == S.size() ? mec(S, S[i], i) : mec2(S, a, S[i], i)
23                         );
24     }
25 }
26 }
27
28 pair<point, ld> smallestEnclosingCircle(vector<point> S){
29     assert(!S.empty());
30     auto r = mec(S, S[0], S.size());
31     return {r.first, sqrt(r.second)};
32 }

```

6.39 Smallest Enclosing Rectangle

```

1 pair<ld, ld> smallestEnclosingRectangle(vector<point> & P){
2     int n = P.size();
3     auto dot = [&](int a, int b){return (P[(a+1)%n]-P[a]).dot(P[(b+1)%n]-P
[b]);};
4     auto cross = [&](int a, int b){return (P[(a+1)%n]-P[a]).cross(P[(b+1)%n]-P[b]);};
5     ld perimeter = inf, area = inf;
6     for(int i = 0, j = 0, k = 0, m = 0; i < n; ++i){
7         while(ge(dot(i, j), 0)) j = (j+1) % n;
8         if(!i) k = j;
9         while(ge(cross(i, k), 0)) k = (k+1) % n;
10        if(!i) m = k;
11        while(le(dot(i, m), 0)) m = (m+1) % n;
12        //pairs: (i, k) , (j, m)
13        point v = P[(i+1)%n] - P[i];
14        ld h = distancePointLine(P[i], v, P[k]);
15        ld w = distancePointLine(P[j], v.perp(), P[m]);
16        perimeter = min(perimeter, 2 * (h + w));
17        area = min(area, h * w);
18    }
19    return {area, perimeter};
20 }

```

6.40 Vantage Point Tree

```

1 struct vantage_point_tree{
2     struct node
3     {

```

```

4 point p;
5 ld th;
6 node *l, *r;
7 }*root;
8
9 vector<pair<ld, point>> aux;
10
11 vantage_point_tree(vector<point> &ps){
12     for(int i = 0; i < ps.size(); ++i)
13         aux.push_back({ 0, ps[i] });
14     root = build(0, ps.size());
15 }
16
17 node *build(int l, int r){
18     if(l == r)
19         return 0;
20     swap(aux[1], aux[l + rand() % (r - 1)]);
21     point p = aux[l++].second;
22     if(l == r)
23         return new node({ p });
24     for(int i = l; i < r; ++i)
25         aux[i].first = (p - aux[i].second).dot(p - aux[i].second);
26     int m = (l + r) / 2;
27     nth_element(aux.begin() + l, aux.begin() + m, aux.begin() + r);
28     return new node({ p, sqrt(aux[m].first), build(l, m), build(m, r) });
29     ;
30 }
31 priority_queue<pair<ld, node*>> que;
32
33 void k_nn(node *t, point p, int k){
34     if(!t)
35         return;
36     ld d = (p - t->p).length();
37     if(que.size() < k)
38         que.push({ d, t });
39     else if(ge(que.top().first, d)){
40         que.pop();
41         que.push({ d, t });
42     }
43     if(!t->l && !t->r)
44         return;
45     if(le(d, t->th)){

```

```
46     k_nn(t->l, p, k);
47     if(leq(t->th - d, que.top().first))
48         k_nn(t->r, p, k);
49 }else{
50     k_nn(t->r, p, k);
51     if( leq(d - t->th, que.top().first))
52         k_nn(t->l, p, k);
53 }
54 }
55
56 vector<point> k_nn(point p, int k){
57     k_nn(root, p, k);
58     vector<point> ans;
59     for(; !que.empty(); que.pop())
60         ans.push_back(que.top().second->p);
61     reverse(ans.begin(), ans.end());
62     return ans;
63 }
64 };
```

7 Graphs

7.1 2Sat

```

16     adj_t(n_vertices),
17     used(n_vertices),
18     comp(n_vertices, -1),
19     assignment(n_vars) {
20         order.reserve(n_vertices); // Pre-allocate memory for efficiency
21     }
22
23     // First DFS pass for Kosaraju's algorithm (on original graph)
24     void dfs1(int v) {
25         used[v] = true;
26         for (int u : adj[v]) {
27             if (!used[u])
28                 dfs1(u);
29         }
30         order.push_back(v); // Save the vertex post-DFS for reverse ordering
31     }
32
33     // Second DFS pass on the transposed graph to label components
34     void dfs2(int v, int cl) {
35         comp[v] = cl;
36         for (int u : adj_t[v]) {
37             if (comp[u] == -1)
38                 dfs2(u, cl);
39         }
40     }
41
42     // Solves the 2-SAT problem using Kosaraju's algorithm
43     bool solve_2SAT() {
44         // 1st pass: fill the order vector
45         order.clear();
46         used.assign(n_vertices, false);
47         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
65             the same SCC
66         assignment[i / 2] = comp[i] > comp[i + 1]; // True if var's
67             component comes after its negation
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
11   connected components.
12 - Works on undirected graphs (connected or disconnected).
13 */
14
15 int n; // Number of nodes in the graph
16 vector<vector<int>> adj; // Adjacency list of the undirected graph
17
18 vector<bool> visited; // Marks if a node was visited during DFS
19 vector<int> tin, low; // tin[v]: discovery time; low[v]: lowest
20   discovery time reachable from subtree
21 int timer; // Global time counter for DFS
22
23 // DFS traversal to identify articulation points
24 void dfs(int v, int p = -1) {
25   visited[v] = true;
26   tin[v] = low[v] = timer++;
27   int children = 0;
28   for (int to : adj[v]) {
29     if (to == p) continue; // Skip the parent edge
30     if (visited[to]) {
31       // Back edge
32       low[v] = min(low[v], tin[to]);
33     } else {
34       dfs(to, v);
35       low[v] = min(low[v], low[to]);
36       // Articulation point condition for non-root
37       if (low[to] >= tin[v] && p != -1) {
38         // v is an articulation point
39         // handle_cutpoint(v);
40       }
41       ++children;
42     }
43   }
44   // Articulation point condition for root
45   if (p == -1 && children > 1) {
46     // v is an articulation point
47     // handle_cutpoint(v);
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

```

```

29 // For path reconstruction add vector<int>& parent as parameter
30 bool spfa(int s, vector<int>& d, vector<int>& parent) {
31     int n = adj.size();
32     d.assign(n, INF);
33     vector<int> cnt(n, 0);      // Count how many times each node has
34         been relaxed
35     vector<bool> inqueue(n, false); // Tracks if a node is currently in
36         queue
37     queue<int> q;
38
39     d[s] = 0;
40     q.push(s);
41     inqueue[s] = true;
42
43     while (!q.empty()) {
44         int v = q.front();
45         q.pop();
46         inqueue[v] = false;
47
48         for (auto edge : adj[v]) {
49             int to = edge.first;
50             int len = edge.second;
51
52             if (d[v] + len < d[to]) {
53                 parent[to] = v; // For path reconstruction
54                 d[to] = d[v] + len;
55                 if (!inqueue[to]) {
56                     q.push(to);
57                     inqueue[to] = true;
58                     cnt[to]++;
59                     if (cnt[to] > n)
60                         return false; // Negative weight cycle detected
61                 }
62             }
63         }
64     }
65
66     return true; // No negative cycles; shortest paths computed
}

```

7.4 Bipartite Checker

```

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

```

7.5 Bipartite Maximum Matching

```

1 /*
2 Maximum Bipartite Matching (Kuhn's Algorithm)

```

```
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 Bipartite Minimum Maximum Matching2

```

1  /*
2   THIS CODE HAS NOT BEEN TESTED BEFOREHAND
3   Maximum Bipartite Matching - Hopcroft-Karp Algorithm
4
5   Indexing: 0-based
6   Time Complexity: O(sqrt(V) * E)
7   Space Complexity: O(V + E)
8
9   Input:
10  - n: number of nodes on the left
11  - m: number of nodes on the right
12  - g[l]: adjacency list for left node l (list of right-side neighbors
13
14   Output:
15  - match_right[r] = l means right node r is matched to left node l
16  - match_left[l] = r means left node l is matched to right node r
17  - Function returns total number of matching pairs
18 */
19
20 const int INF = 1e9;
21
22 int n, m; // Number of nodes on left and right
23 vector<vector<int>> g; // Adjacency list: g[l] contains neighbors
24 // of left node l
25 vector<int> match_left, match_right; // match_left[l], match_right[r]
26 vector<int> dist; // Distance levels for BFS
27 // BFS to build layer graph, returns true if there's at least one
28 // unmatched node on the right reachable from left

```

```

28 bool bfs() {
29     queue<int> q;
30     dist.assign(n, INF);
31
32     for (int l = 0; l < n; ++l) {
33         if (match_left[l] == -1) {
34             dist[l] = 0;
35             q.push(l);
36         }
37     }
38
39     bool found = false;
40
41     while (!q.empty()) {
42         int l = q.front(); q.pop();
43         for (int r : g[l]) {
44             int matched_l = match_right[r];
45             if (matched_l == -1) {
46                 found = true; // Free right node reachable -> potential
47                 augmenting path
48             } else if (dist[matched_l] == INF) {
49                 dist[matched_l] = dist[l] + 1;
50                 q.push(matched_l);
51             }
52         }
53
54         return found;
55     }
56
57 // DFS to find augmenting paths in the layered graph
58 bool dfs(int l) {
59     for (int r : g[l]) {
60         int matched_l = match_right[r];
61         if (matched_l == -1 || (dist[matched_l] == dist[l] + 1 && dfs(
62             matched_l))) {
63             match_left[l] = r;
64             match_right[r] = l;
65             return true;
66         }
67     }
68     dist[l] = INF;
69     return false;
70 }
71
72 // Main function to compute maximum matching
73 int hopcroft_karp() {
74     match_left.assign(n, -1);
75     match_right.assign(m, -1);
76
77     int matching = 0;
78     while (bfs()) {
79         for (int l = 0; l < n; ++l) {
80             if (match_left[l] == -1 && dfs(l)) {
81                 ++matching;
82             }
83         }
84     }
85     return matching;
86 }
87
88 int main() {
89     cin >> n >> m;
90     g.assign(n, {});
91     int e; cin >> e;
92     while (e--) {
93         int u, v;
94         cin >> u >> v;
95         g[u].push_back(v); // u in [0, n-1], v in [0, m-1]
96     }
97
98     int res = hopcroft_karp();
99     cout << "Maximum_matching_size:" << res << '\n';
100    for (int r = 0; r < m; ++r) {
101        if (match_right[r] != -1) {
102            cout << match_right[r] + 1 << " " << r + 1 << '\n'; // 1-based
103            output
104        }
105    }
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
823
824
825
825
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1
```

```

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, // Adjacency list of the undirected graph
24                                         vector<bool> &is_cutpoint, // Output vector (resized internally)
25                                         vector<int> &id) { // Output vector (resized internally)
26
27   int n = g.size();
28   vector<vector<int>> comps; // Stores all biconnected components
29   vector<int> stk; // Stack of visited nodes for current component
30   vector<int> num(n), low(n); // DFS discovery time and low-link values
31   is_cutpoint.assign(n, false);
32
33   // DFS to find BCCs and articulation points
34   function<void(int, int, int&)> dfs = [&](int node, int parent, int &timer) {
35     num[node] = low[node] = ++timer;
36     stk.push_back(node);
37     for (int son : g[node]) {
38       if (son == parent) continue;
39       if (num[son]) {

```

```

40         // Back edge
41         low[node] = min(low[node], num[son]);
42     } else {
43       dfs(son, node, timer);
44       low[node] = min(low[node], low[son]);
45       // Check articulation point condition
46       if (low[son] >= num[node]) {
47         is_cutpoint[node] = (num[node] > 1 || num[son] > 2); // For root and non-root
48         comps.push_back({node});
49       while (comps.back().back() != son) {
50         comps.back().push_back(stk.back());
51         stk.pop_back();
52       }
53     }
54   }
55 }
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 return t;
87 };
88
89 return build_tree(); // Return the block-cut tree
90 }

```

7.8 Blossom

```

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

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

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

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

```

```

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

```

```

30     int match[N];        // match[u]: matched partner of u (0 if unmatched)
31     int aux[N];          // aux[u]: visit marker for LCA
32     int t;               // global timestamp for LCA markers
33     int n;               // number of nodes
34     bool ad[N];          // ad[u]: whether u is reachable in an alternating
                           // path
35     vector<int> g[N];   // g[u]: adjacency list
36     queue<int> Q;        // BFS queue
37
38 // Constructor: initializes data for n nodes
39 Blossom() {}
40 Blossom(int _n) {
41     n = _n;
42     t = 0;
43     for (int i = 0; i <= n; ++i) {
44         g[i].clear();
45         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         }
73     }
74 }

```

```

72     v = orig[par[match[v]]]; // Move to the parent in the
    // alternating tree
73 }
74 swap(v, w);
75 }
76 }

// Contract a blossom from v and w with common ancestor a
78 void blossom(int v, int w, int a) {
79     while (orig[v] != a) {
80         par[v] = w;
81         w = match[v];
82         ad[v] = true;
83         if (vis[w] == 1) Q.push(w), vis[w] = 0;
84         orig[v] = orig[w] = a;
85         v = par[w];
86     }
87 }

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

95     while (!Q.empty()) {
96         int v = Q.front(); Q.pop();
97         ad[v] = true;
98         for (int x : g[v]) {
99             if (vis[x] == -1) {
100                 par[x] = v;
101                 vis[x] = 1;
102                 if (!match[x]) {
103                     augment(u, x);
104                     return true;
105                 }
106                 Q.push(match[x]);
107                 vis[match[x]] = 0;
108             } else if (vis[x] == 0 && orig[v] != orig[x]) {
109                 int a = lca(orig[v], orig[x]);
110                 blossom(x, v, a);
111             }
112         }
113     }
114     blossom(v, x, a);
115 }
116 }
117 }
118 return false;
119 }

// Computes maximum matching and returns the size
121 int maximum_matching() {
122     int ans = 0;
123     vector<int> p(n - 1);
124     iota(p.begin(), p.end(), 1);
125     shuffle(p.begin(), p.end(), rnd);
126     for (int i = 1; i <= n; ++i) {
127         shuffle(g[i].begin(), g[i].end(), rnd);
128     }
129 }

// Greedy matching: try to match unmatched nodes directly
131 for (int u : p) {
132     if (!match[u]) {
133         for (int v : g[u]) {
134             if (!match[v]) {
135                 match[u] = v;
136                 match[v] = u;
137                 ++ans;
138                 break;
139             }
140         }
141     }
142 }
143 }

// Augmenting path phase
145 for (int i = 1; i <= n; ++i) {
146     if (!match[i] && bfs(i)) ++ans;
147 }

149 return ans;
150 }
151 }
152 } M;

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 }
```

7.9 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 Input:
9 n - Number of nodes in the graph
10 adj - Adjacency list of the undirected graph

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

17 */

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

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

25 // DFS to detect bridges

26 void dfs(int v, int p = -1) {
27 visited[v] = true;
28 tin[v] = low[v] = timer++;
29 for (int to : adj[v]) {
30 if (to == p) continue; // Skip edge to parent
31 if (visited[to]) {
32 // Back edge
33 low[v] = min(low[v], tin[to]);
34 } else {
35 dfs(to, v);
36 low[v] = min(low[v], low[to]);
37 // Bridge condition: if no back edge connects subtree rooted at ‘
38 to‘ to ancestors of ‘v‘
39 if (low[to] > tin[v]) {
40 // (v, to) is a bridge
41 // Example: bridges.push_back({v, to});
42 }
43 }
44 }
45 // 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 }
```

7.10 Bridges Online

```

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

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

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

Output:
    'bridges' - Global variable representing the current number of
                bridges.
*/
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges; // Number of bridges in the graph
int lca_iteration;
vector<int> last_visit;
```

```

27
28 // Initializes the data structures
29 void init(int n) {
30     par.resize(n);
31     dsu_2ecc.resize(n);
32     dsu_cc.resize(n);
33     dsu_cc_size.resize(n);
34     last_visit.assign(n, 0);
35     lca_iteration = 0;
36     bridges = 0;
37
38     for (int i = 0; i < n; ++i) {
39         par[i] = -1;
40         dsu_2ecc[i] = i;
41         dsu_cc[i] = i;
42         dsu_cc_size[i] = 1;
43     }
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 }
```

```

69     dsu_cc_size[root] = dsu_cc_size[child];
70 }
71
72 // Merges paths from a and b to their lowest common ancestor in the 2ECC
// forest
73 void merge_path(int a, int b) {
74     ++lca_iteration;
75     vector<int> path_a, path_b;
76     int lca = -1;
77
78     while (lca == -1) {
79         if (a != -1) {
80             a = find_2ecc(a);
81             path_a.push_back(a);
82             if (last_visit[a] == lca_iteration) {
83                 lca = a;
84                 break;
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     --bridges;
106 }
107 for (int v : path_b) {
108     dsu_2ecc[v] = lca;
109     if (v == lca) break;
110     --bridges;
111 }
112 }
113
114 // Adds an undirected edge between a and b and updates bridge count
115 void add_edge(int a, int b) {
116     a = find_2ecc(a);
117     b = find_2ecc(b);
118     if (a == b) return; // Already in the same 2ECC
119
120     int ca = find_cc(a);
121     int cb = find_cc(b);
122
123     if (ca != cb) {
124         // Bridge found - connects two different components
125         ++bridges;
126         // Union by size
127         if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
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 }

```

7.11 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 priority_queue<pair<ll, int>> q; // Max-heap, so we push negative
4     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}

```

7.12 Eulerian Path

An Eulerian Path is a path that passes through every edge once. For an undirected graph an eulerian path exists if the degree of every node is even or the degree of exactly two nodes is odd. In the first case, the eulerian path is also an eulerian circuit or cycle. In a directed graph, an eulerian path exists if at most one 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 This will find an Eulerian Circuit.
4 For directed graph eulerian path must start on vertex with outdegree[i]
5     ] == indegree[i] + 1 and ends on indegree[i] = outdegree[i] + 1.
6 For directed graph eulerian Cycle only exists if indegree[i] =
7     outdegree[i] for all i.
8 For undirected graph eulerian path must start on vertex with odd
9     degree and will end in the other vertex with odd degree.
10 For undirected graph eulerian Cycle only exists degree is even for all
11     vertices.
12
13 -----
14 Time Complexity: O(E)
15 Space Complexity: O(V + E)
16
17 Input:

```

```

14     - g: adjacency list of the graph
15         * Directed: vector<vector<pair<int, int>>> g
16             where g[v] = list of {to, edge_index}
17         * Undirected: vector<vector<int>> g
18             where g[v] = list of neighbors
19     - seen: vector<bool> seen(E) - only needed for directed version
20     - path: vector<int> path - will be filled in reverse order of
21         traversal
22             reverse(path.begin(), path.end());
23 */
24
25 // Directed Version //
26 void dfs_directed(int node) {
27     while (!g[node].empty()) {
28         auto [son, idx] = g[node].back();
29         g[node].pop_back();
30         if (seen[idx]) continue; // Skip if edge already visited
31         seen[idx] = true;
32         dfs_directed(son);
33     }
34     path.push_back(node); // Post-order insertion (reverse of actual path)
35 }
36
37 // Undirected Version //
38 // Be careful with graph not being connected. Add set to not repeat
39 // edges(going from u to v and then from v to u)
40 void dfs_undirected(int node) {
41     while (!g[node].empty()) {
42         int son = g[node].back();
43         g[node].pop_back();
44         dfs_undirected(son);
45     }
46     path.push_back(node); // Post-order insertion

```

7.13 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 }
```

7.14 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)
```

```

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 }
```

7.15 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
15      m
16
17 Output:
18  - 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];
25     to
26
27 void stableMarriage(int n) {
28     fill_n(pv, n, -1); // All women initially unmatched
29     fill_n(pm, n, -1); // All men initially unmatched
30     fill_n(pun, n, 0); // Each man starts at his top preference
31
32     int unmatched = n; // Number of free men
33     int i = n - 1;     // Current man index (rotates over all men)
34
35     #define engage pm[j] = i; pv[i] = j;
36
37     while (unmatched) {
38         while (pm[i] == -1) {
39             int j = gv[i][pun[i]++]; // Next woman on man i's list
40
41             if (pv[j] == -1) {
42                 // Woman j is free -> engage with man i
43                 unmatched--;
44                 engage;
45             } else if (om[j][i] < om[j][pv[j]]) {
46                 // Woman j prefers i over her current partner
47                 int loser = pv[j];
48                 pm[loser] = -1;
49                 engage;
50                 i = loser; // Reconsider the rejected man
51             }
52
53             // Move to next unmatched man
54             i--;
55             if (i < 0) i = n - 1;
56         }
57
58     #undef engage
59 }

```

7.16 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 }

```

7.17 Stoer-Wagner

```

1 /*
2          Stoer - Wagner
3
4 Solves the minimum cut problem in undirected weighted graphs with non-
      negative weights.

```

```

5
6     Time Complexity: O(V ^ 3)
7     Space Complexity: O(V ^ 2)
8 */
9
10    mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
11    struct StoerWagner {
12        int n;
13        long long G[N][N], dis[N];
14        int idx[N];
15        bool vis[N];
16        const long long inf = 1e18;
17        StoerWagner() {}
18        StoerWagner(int _n) {
19            n = _n;
20            memset(G, 0, sizeof G);
21        }
22        void add_edge(int u, int v, long long w) { //undirected edge, multiple
23            //edges are merged into one edge
24            if (u != v) {
25                G[u][v] += w;
26                G[v][u] += w;
27            }
28        }
29        long long solve() {
30            long long ans = inf;
31            for (int i = 0; i < n; ++ i) idx[i] = i + 1;
32            shuffle(idx, idx + n, rnd);
33            while (n > 1) {
34                int t = 1, s = 0;
35                for (int i = 1; i < n; ++ i) {
36                    dis[idx[i]] = G[idx[0]][idx[i]];
37                    if (dis[idx[i]] > dis[idx[t]]) t = i;
38                }
39                memset(vis, 0, sizeof vis);
40                vis[idx[0]] = true;
41                for (int i = 1; i < n; ++ i) {
42                    if (i == n - 1) {
43                        if (ans > dis[idx[t]]) ans = dis[idx[t]]; //idx[s] - idx[t] is
44                        ----- in two halves of the mincut
45                        if (ans == 0) return 0;
46                        for (int j = 0; j < n; ++ j) {
47                            G[idx[s]][idx[j]] += G[idx[j]][idx[t]];
48                        }
49                    }
50                }
51            }
52        }
53    };
54
55    int main() {
56        int n, m;
57        cin >> n >> m;
58        G.assign(n, vector<long long>(n));
59        for (int i = 0; i < m; i++) {
60            int u, v;
61            cin >> u >> v;
62            G[u][v] = G[v][u] = 1;
63        }
64        StoerWagner sw(n);
65        cout << sw.solve();
66    }
67
68 }

```

```

46         G[idx[j]][idx[s]] += G[idx[j]][idx[t]];
47     }
48     idx[t] = idx[--n];
49 }
50 vis[idx[t]] = true;
51 s = t;
52 t = -1;
53 for (int j = 1; j < n; ++j) {
54     if (!vis[idx[j]]) {
55         dis[idx[j]] += G[idx[s]][idx[j]];
56         if (t == -1 || dis[idx[t]] < dis[idx[j]]) t = j;
57     }
58 }
59
60 return ans;
61 }
62 };
63
64 int32_t main() {
65     ios_base::sync_with_stdio(0);
66     cin.tie(0);
67     int t, cs = 0;
68     cin >> t;
69     while (t--) {
70         int n, m;
71         cin >> n >> m;
72         StoerWagner st(n);
73         while (m--) {
74             int u, v, w;
75             cin >> u >> v >> w;
76             st.add_edge(u, v, w);
77         }
78         cout << "Case #"
79             << ++cs << ": "
80             << st.solve() << '\n';
81     }
82     return 0;
83 }
```

8 Linear Algebra

8.1 Gaussian Elimination

1 /*

```

2 Gaussian elimination
3 -----
4 Utilized for solving linear systems of equations.
5
6 If the system is modulo 2, use GaussianEliminationModulo instead.
7
8 Time Complexity: O(n ^ 3)
9 */

10
11 const double eps = 1e-9;
12 int Gauss(vector<vector<double>> a, vector<double> &ans) {
13     int n = (int)a.size(), m = (int)a[0].size() - 1;
14     vector<int> pos(m, -1);
15     double det = 1; int rank = 0;
16     for(int col = 0, row = 0; col < m && row < n; ++col) {
17         int mx = row;
18         for(int i = row; i < n; i++) if(fabs(a[i][col]) > fabs(a[mx][col]))
19             mx = i;
20         if(fabs(a[mx][col]) < eps) {det = 0; continue;}
21         for(int i = col; i <= m; i++) swap(a[row][i], a[mx][i]);
22         if (row != mx) det = -det;
23         det *= a[row][col];
24         pos[col] = row;
25         for(int i = 0; i < n; i++) {
26             if(i != row && fabs(a[i][col]) > eps) {
27                 double c = a[i][col] / a[row][col];
28                 for(int j = col; j <= m; j++) a[i][j] -= a[row][j] * c;
29             }
30         }
31         ++row; ++rank;
32     }
33     ans.assign(m, 0);
34     for(int i = 0; i < m; i++) {
35         if(pos[i] != -1) ans[i] = a[pos[i]][m] / a[pos[i]][i];
36     }
37     for(int i = 0; i < n; i++) {
38         double sum = 0;
39         for(int j = 0; j < m; j++) sum += ans[j] * a[i][j];
40         if(fabs(sum - a[i][m]) > eps) return -1; //no solution
41     }
42     for(int i = 0; i < m; i++) if(pos[i] == -1) return 2; //infinte
43         solutions
44 }
```

```

43     return 1; //unique solution
44 }
45 int main() {
46     int n, m; cin >> n >> m;
47     vector<vector<double> > v(n);
48     for(int i = 0; i < n; i++) {
49         for(int j = 0; j <= m; j++) {
50             double x; cin >> x; v[i].push_back(x);
51         }
52     }
53     vector<double> ans;
54     int k = Gauss(v, ans);
55     if(k) for(int i = 0; i < n; i++) cout << fixed << setprecision(5) <<
56         ans[i] << ' ';
57     else cout << "no_solution\n";
58     return 0;
}

```

8.2 Gaussian Elimination Modulo

```

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

Time Complexity: O(n ^ 3)
*/
//n = number of equations, m = number of variables
int Gauss(int n, int m, vector<bitset<N>> a, bitset<N> &ans) {
    //reversing for lexicographically largest solution
    for (int i = 0; i < n; i++) {
        bitset<N> tmp;
        for (int j = 0; j < m; j++) tmp[j] = a[i][m - j - 1];
        tmp[m] = a[i][m];
        a[i] = tmp;
    }
    int rank = 0, det = 1;
    vector<int> pos(N, -1);
    for(int col = 0, row = 0; col < m && row < n; ++col) {
        int mx = row;
        for(int i = row; i < n; ++i) if(a[i][col]) { mx = i; break; }
        if(!a[mx][col]) { det = 0; continue; }

```

```

24     swap(a[mx], a[row]);
25     if(row != mx) det = det == 0 ? 0 : 1;
26     det &= a[row][col];
27     pos[col] = row;
28     //forward elimination
29     for(int i = row + 1; i < n; ++i) if (i != row && a[i][col]) a[i] ^= a[row];
30     ++row, ++rank;
}
31 ans.reset();
//backward substitution
32 for (int i = m - 1; i >= 0; i--) {
33     if (pos[i] == -1) ans[i] = true;
34     else {
35         int k = pos[i];
36         for (int j = i + 1; j < m; j++) if (a[k][j]) ans[i] = ans[i] ^
37             ans[j];
38         ans[i] = ans[i] ^ a[k][m];
39     }
40 }
41 for(int i = rank; i < n; ++i) if(a[i][m]) return -1; //no solution
42 //reversing again because we reversed earlier
43 bitset<N> tmp;
44 for (int j = 0; j < m; j++) tmp[j] = ans[m - j - 1];
45 ans = tmp;
46 int free_var = 0;
47 for(int i = 0; i < m; ++i) if(pos[i] == -1) free_var++;
48 return free_var; //has solution
}
49 string read() {
50     string t;
51     if(!(cin >> t)) return "";
52     if(t.empty() || t == "and") return "";
53     while(t[0] == '(') t.erase(t.begin());
54     while(t.back() == ')') t.pop_back();
55     return t;
}
56 bool is_var(string t) { return t.size() > 0 && t[0] == 'x'; }
57 int get_var(string t) { return atoi(t.substr(1).c_str()) - 1; }
58 int32_t main() {
59     ios_base::sync_with_stdio(0);
60     cin.tie(0);
61     int n, m; cin >> n >> m;
62 
```

```

65     vector<bitset<N>> bs(n, bitset<N>(0));
66     for(int i = 0; i < n; i++) {
67         string s;
68         bool eq = 1;
69         while((s = read()).size() > 0) {
70             if(is_var(s)) {
71                 int x = get_var(s);
72                 bs[i][x] = bs[i][x] ^ 1;
73             }
74             else if(s == "not") eq ^= 1;
75         }
76         bs[i][m] = eq;
77     }
78     bitset<N> ans;
79     int ok = Gauss(n, m, bs, ans);
80     if (ok == -1) cout << "impossible\n";
81     else {
82         for (int i = 0; i < m; i++) cout << "FT"[ans[i]]; cout << '\n';
83     }
84     return 0;
85 }
86 //https://codeforces.com/gym/101908/problem/M

```

8.3 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
16     &c, bool mini = true){
17     int n = c.size(), m = b.size();
18     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         }
61     }
62 }
```

```

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     return {vec(n), oo};
91 }
92
93
94 int main(){
95     int m, n;
96     bool mini = true;
97     cout << "Número de restricciones:" ;
98     cin >> m;
99     cout << "Número de incógnitas:" ;
100    cin >> n;
101    mat A(m, vec(n));
102    vec b(m), c(n);
103    for(int i = 0; i < m; ++i){

```

```

104        cout << "Restriccion#" << (i + 1) << ":" ;
105        for(int j = 0; j < n; ++j){
106            cin >> A[i][j];
107        }
108        cin >> b[i];
109    }
110    cout << "[0]Max_o [1]Min?:" ;
111    cin >> mini;
112    cout << "Coeficientes de:" << (mini ? "min" : "max") << "\n";
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") << "\n" << ans.second << ", cuando:" ;
119    cout << "\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 binpow(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)*binpow(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 Combination Rank

```

1 // Compute nCr (binomial coefficient)
2 long long nCr(int n, int r) {
3     if (r > n) return 0;
4     if (r == 0 || r == n) return 1;
5     long long res = 1;
6     for (int i = 1; i <= r; ++i) {
7         res = res * (n - i + 1) / i;
8     }
9     return res;
}
11
12 // Compute 0-indexed lexicographic rank of a combination
13 long long combinationRank0(const vector<int>& comb) {
14     long long rank = 0;
15     for (int i = 0; i < (int)comb.size(); ++i) {
16         rank += nCr(comb[i] - 1, i + 1);
17     }
18     return rank; // 0-indexed
}
19
20
21 // Given n, r, and rank (0-indexed), reconstruct the combination
22 vector<int> combinationUnrank0(int n, int r, long long rank) {
23     vector<int> comb;
24     int x = 1;
25
26     for (int i = r; i >= 1; --i) {
27         while (nCr(n - x, i) > rank) {
28             x++;
29         }
30         comb.push_back(x);
31         rank -= nCr(n - x, i);
32         x++;
33     }
34
35     return comb;
}

```

9.3 Chinese Remainder Theorem

1 | ll euclid(ll a, ll b, ll &x, ll &y) {

```

2     if (!b) return x = 1, y = 0, a;
3     ll d = euclid(b, a % b, y, x);
4     return y -= a/b * x, d;
5 }
6
7 ll crt(ll a, ll m, ll b, ll n) {
8     if (n > m) swap(a, b), swap(m, n);
9     ll x, y, g = euclid(m, n, x, y);
10    if ((a - b) % g != 0) return -1;
11    x = (b - a) % n * x % n / g * m + a;
12    return x < 0 ? x + m*n/g : x;
13 }

```

9.4 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 }

```

9.5 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 * 1ll * 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 * 1ll * a) % m;
14    }
15
16    for (int p = 1, cur = 1; p <= n; ++p) {
17        cur = (cur * 1ll * 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 * 1ll * 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 * 1ll * 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 * 1ll * a) % m;
48 }
49
50 for (int p = 1, cur = k; p <= n; ++p) {
51     cur = (cur * 1ll * an) % m;
52     if (vals.count(cur)) {
53         int ans = n * p - vals[cur] + add;
54         return ans;
55     }
56 }
57 return -1;
58 }
```

9.6 Divisors

```

1 ll numberOfDivisors(ll num) {
2     ll total = 1;
3     for (int i = 2; (ll)i * i <= num; i++) {
4         if (num % i == 0){
5             int e = 0;
6             do{
7                 e++;
8                 num /= i;
9             } while (num % i == 0);
10            total *= e + 1;
11        }
12    }
13    if (num > 1) total *= 2;
14    return total;
```

```

15 }
16
17 ll SumOfDivisors(ll num) {
18     ll total = 1;
19     for (int i = 2; (ll)i * i <= num; i++) {
20         if (num % i == 0) {
21             int e = 0;
22             do {
23                 e++;
24                 num /= i;
25             } while (num % i == 0);
26             ll sum = 0, pow = 1;
27             do {
28                 sum += pow;
29                 pow *= i;
30             } while (e-- > 0);
31             total *= sum;
32         }
33     }
34     if (num > 1) total *= (1 + num);
35     return total;
36 }
```

9.7 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++) phi[i] = i;
5     for (int i = 2; i <= n; i++) {
6         if (phi[i] == i) {
7             for (int j = i; j <= n; j += i)
8                 phi[j] -= phi[j] / i;
9         }
10    }
11    return phi;
12 }
```

9.8 Fibonacci

```

1 // Fibonacci in O(log n)
2 void fib(ll n, ll&x, ll&y){
3     if(n==0){
4         x = 0;
```

```

5     y = 1;
6     return ;
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 // Usage
19 ll x, y;
20 fib(10, x, y);
21 cout << x << " " << y << endl;
22 // This will output 55 89

```

9.9 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    }
14    inline Mat operator + (const Mat &b) {
15        assert(n == b.n && m == b.m);
16        Mat ans = Mat(n, m);
17        for(int i = 0; i < n; i++) {
18            for(int j = 0; j < m; j++) {
19                ans.a[i][j] = (a[i][j] + b.a[i][j]) % mod;
20            }
21        }
22        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 // Usage
63 // Mat a(n, n);

```

```

64 // Mat b(n, n);
65 // Mat c = a * b;
66 // Mat d = a + b;
67 // Mat e = a - b;
68 // Mat f = a.pow(k);
69 // a.a[i][j] = x;

```

9.10 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 }

```

9.11 Möbius

```

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 }

```

9.12 Permutation Rank

```

1 // Compute factorial (up to reasonable limits)
2 long long fact(int n) {
3     static vector<long long> f(21, -1); // cache up to 20!
4     if (n <= 1) return 1;
5     if (f[n] != -1) return f[n];
6     return f[n] = n * fact(n - 1);
7 }
8
9 // Compute 0-indexed lexicographic rank of a permutation
10 long long permutationRank0(const vector<int>& perm) {
11     int n = perm.size();
12     long long rank = 0;
13     vector<int> used(n + 1, 0);
14
15     for (int i = 0; i < n; ++i) {

```

```

16     int smaller_unused = 0;
17     for (int j = 1; j < perm[i]; ++j) {
18         if (!used[j]) smaller_unused++;
19     }
20     rank += smaller_unused * fact(n - i - 1);
21     used[perm[i]] = 1;
22 }
23 return rank;
24 }

// Given n and rank (0-indexed), reconstruct the permutation
26 vector<int> permutationUnrank0(int n, long long rank) {
27     vector<int> elems(n);
28     iota(elems.begin(), elems.end(), 1); // {1, 2, 3, ..., n}
29     vector<int> perm;

31     for (int i = n; i >= 1; --i) {
32         long long f = fact(i - 1);
33         int idx = rank / f;
34         rank %= f;
35         perm.push_back(elems[idx]);
36         elems.erase(elems.begin() + idx);
37     }
38
39     return perm;
40 }
41

```

9.13 Prefix Sum Phi

```

1 vector<ll> sieve(kMaxV + 1, 0);
2 vector<ll> phi(kMaxV + 1, 0);
3
4 void primes()
{
5     phi[1]=1;
6     vector<ll> pr;
7     for(int i=2;i<kMaxV;i++){
8         if(sieve[i]==0){
9             sieve[i]=i;
10            pr.pb(i);
11            phi[i]=i-1;
12        }
13        for(auto p:pr){
14

```

```

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

```

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

```

9.15 Identities

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

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

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

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

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

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

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

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

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

(Dirichlet Convolution) Let f, g be arithmetic functions, then

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

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

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

9.16 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 $\frac{1}{2} \sum_{i=0}^n f(i)$ 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 $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$

9.17 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 exponentiación de matrices.

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

9.18 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 .

9.19 Sums

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

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$$

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

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

9.21 Cayley's formula

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

9.22 Geometric series

$$\begin{aligned} & \text{Infinite} \\ & a + ar + ar^2 + ar^3 + \dots + \sum_{k=0}^{\infty} ar^k \\ & \text{Sum} = \frac{a}{1-r} \end{aligned}$$

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

9.23 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 $n < 1e19$.

9.24 Sum of divisors

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

9.25 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.

9.26 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 $g(r) = \text{mex}(\{g(r_1), g(r_2), \dots, g(r_k)\})$ 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 Fórmulas y notas

11.1 Números de Stirling del primer tipo

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

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

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

, $n > 0$

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

, $k > 0$

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

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

11.2 Números de Stirling del segundo tipo

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

$$\left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\} = 1$$

$$\left\{ \begin{array}{c} 0 \\ n \end{array} \right\} = \left\{ \begin{array}{c} n \\ 0 \end{array} \right\} = 0$$

, $n > 0$

$$\left\{ \begin{array}{c} n \\ k \end{array} \right\} = k \left\{ \begin{array}{c} n-1 \\ k \end{array} \right\} + \left\{ \begin{array}{c} n-1 \\ k-1 \end{array} \right\}$$

, $k > 0$

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

11.3 Números de Euler

$\langle\!\langle \begin{array}{c} n \\ k \end{array} \rangle\!\rangle$ representa el número de permutaciones de 1 a n en donde exactamente k números son mayores que el número anterior, es decir, cuántas permutaciones tienen

k “ascensos”.

$$\langle\!\langle \begin{array}{c} 1 \\ 0 \end{array} \rangle\!\rangle = 1$$

$$\langle\!\langle \begin{array}{c} n \\ k \end{array} \rangle\!\rangle = (n-k) \langle\!\langle \begin{array}{c} n-1 \\ k-1 \end{array} \rangle\!\rangle + (k+1) \langle\!\langle \begin{array}{c} n-1 \\ k \end{array} \rangle\!\rangle , \quad n \geq 2$$

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

$$\sum_{k=0}^{n-1} \langle\!\langle \begin{array}{c} n \\ k \end{array} \rangle\!\rangle = n!$$

11.4 Números de Catalan

$$C_0 = 1$$

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

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

11.5 Números de Bell

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

$$B_n = \sum_{k=0}^n \left\{ \begin{array}{c} n \\ k \end{array} \right\} = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k$$

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

11.6 Números de Bernoulli

$$B_0^+ = 1$$

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

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

11.7 Fórmula de Faulhaber

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

11.8 Función Beta

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

11.9 Función zeta de Riemann

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

$$\begin{aligned} \zeta(s) &\approx \frac{1}{d_0(1-2^{1-s})} \sum_{k=1}^n \frac{(-1)^{k-1} d_k}{k^s} \\ d_k &= \sum_{j=k}^n \frac{4^j}{n+j} \binom{n+j}{2j} \end{aligned}$$

11.10 Funciones generadoras

$$\begin{aligned} \sum_{n=0}^{\infty} \left(\sum_{k=0}^n a_k \right) x^n &= \frac{1}{1-x} \sum_{n=0}^{\infty} a_n x^n \\ \sum_{n=0}^{\infty} \binom{n+k-1}{k-1} x^n &= \frac{1}{(1-x)^k} \\ \sum_{n=0}^{\infty} p_n x^n &= \frac{1}{\prod_{k=1}^{\infty} (1-x^k)} = \frac{1}{\sum_{n=-\infty}^{\infty} (-1)^n x^{\frac{1}{2}n(3n+1)}} \\ \sum_{p=0}^{\infty} \frac{S_p(n)}{p!} x^p &= \frac{e^{x(n+1)} - e^x}{e^x - 1} \\ \sum_{n=0}^{\infty} n^k x^n &= \frac{\sum_{i=0}^{k-1} \langle\!\langle k \rangle\!\rangle_i x^{i+1}}{(1-x)^{k+1}} \quad , \quad k \geq 1 \end{aligned}$$

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

11.11 Números armónicos

$$\begin{aligned} H_n &= \sum_{k=1}^n \frac{1}{k} \approx \ln(n) + \gamma + \frac{1}{2n} - \frac{1}{12n^2} \\ \gamma &\approx 0.577215664901532860606512 \end{aligned}$$

11.12 Aproximación de Stirling

$$\begin{aligned} \ln(n!) &\approx n \ln(n) - n + \frac{1}{2} \ln(2\pi n) \\ \# \text{ de dígitos de } n! &= 1 + \left\lfloor n \log\left(\frac{n}{e}\right) + \frac{1}{2} \log(2\pi n) \right\rfloor \quad (n \geq 30) \end{aligned}$$

11.13 Ternas pitagóricas

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

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

donde $n \geq 1$, $m > n$, $\gcd(m, n) = 1$ y m, n tienen distinta paridad.

- Árbol de ternas pitagóricas primitivas: al multiplicar cualquiera de estas matrices:

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

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

11.14 Árbol de Stern–Brocot

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

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

11.15 Combinatoria

- Principio de las casillas: al colocar n objetos en k lugares hay al menos $\lceil \frac{n}{k} \rceil$ objetos en un mismo lugar.
- Número de funciones: sean A y B conjuntos con $m = |A|$ y $n = |B|$. Sea $f : A \rightarrow B$:

- Si $m \leq n$, entonces hay $m! \binom{n}{m}$ funciones inyectivas f .
- Si $m = n$, entonces hay $n!$ funciones biyectivas f .
- Si $m \geq n$, entonces hay $n! \binom{m}{n}$ funciones suprayectivas f .

- Barras y estrellas: ¿cuántas soluciones en los enteros no negativos tiene la ecuación $\sum_{i=1}^k x_i = n$? Tiene $\binom{n+k-1}{k-1}$ soluciones.
- ¿Cuántas soluciones en los enteros positivos tiene la ecuación $\sum_{i=1}^k x_i = n$? Tiene $\binom{n-1}{k-1}$ soluciones.
- Desordenamientos: $a_0 = 1$, $a_1 = 0$, $a_n = (n-1)(a_{n-1} + a_{n-2}) = na_{n-1} + (-1)^n$.
- Sea $f(x)$ una función. Sea $g_n(x) = xg'_{n-1}(x)$ con $g_0(x) = f(x)$. Entonces $g_n(x) = \sum_{k=0}^n \binom{n}{k} x^k f^{(k)}(x)$.
- Supongamos que tenemos $m+1$ puntos: $(0, y_0), (1, y_1), \dots, (m, y_m)$. Entonces el polinomio $P(x)$ de grado m que pasa por todos ellos es:

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

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

11.16 Grafos

- Sea d_n el número de grafos con n vértices etiquetados: $d_n = 2^{\binom{n}{2}}$.
- Sea c_n el número de grafos conexos con n vértices etiquetados. Tenemos la recurrencia: $c_1 = 1$ y $d_n = \sum_{k=1}^n \binom{n-1}{k-1} c_k d_{n-k}$. También se cumple, usando funciones generadoras exponenciales, que $C(x) = 1 + \ln(D(x))$.
- Sea t_n el número de torneos fuertemente conexos en n nodos etiquetados. Tenemos la recurrencia $t_1 = 1$ y $d_n = \sum_{k=1}^n \binom{n}{k} t_k d_{n-k}$. Usando funciones generadoras exponenciales, tenemos que $T(x) = 1 - \frac{1}{D(x)}$.
- Número de spanning trees en un grafo completo con n vértices etiquetados: n^{n-2} .
- Número de bosques etiquetados con n vértices y k componentes conexas: kn^{n-k-1} .
- Para un grafo no dirigido simple G con n vértices etiquetados de 1 a n , sea $Q = D - A$, donde D es la matriz diagonal de los grados de cada nodo de G y A es la matriz de adyacencia de G . Entonces el número de spanning trees de G es igual a cualquier cofactor de Q .
- Sea G un grafo. Se define al polinomio $P_G(x)$ como el polinomio cromático de G , en donde $P_G(k)$ nos dice cuántas k -coloraciones de los vértices admite G . Ejemplos comunes:
 - Grafo completo de n nodos: $P(x) = x(x-1)(x-2)\dots(x-(n-1))$
 - Grafo vacío de n nodos: $P(x) = x^n$
 - Árbol de n nodos: $P(x) = x(x-1)^{n-1}$
 - Ciclo de n nodos: $P(x) = (x-1)^n + (-1)^n(x-1)$

11.17 Teoría de números

$$(f * e)(n) = f(n)$$

$$(\varphi * \mathbf{1})(n) = n$$

$$(\mu * \mathbf{1})(n) = e(n)$$

$$\varphi(n^k) = n^{k-1} \varphi(n)$$

$$\sum_{\substack{k=1 \\ \gcd(k,n)=1}}^n k = \frac{n\varphi(n)}{2}, \quad n \geq 2$$

$$\sum_{k=1}^n \text{lcm}(k, n) = \frac{n}{2} + \frac{n}{2} \sum_{d|n} d\varphi(d) = \frac{n}{2} + \frac{n}{2} \prod_{p^a|n} \frac{p^{2a+1} + 1}{p + 1}$$

$$\sum_{k=1}^n \gcd(k, n) = \sum_{d|n} d\varphi\left(\frac{n}{d}\right) = \prod_{p^a|n} p^{a-1} (1 + (a+1)(p-1))$$

- Lifting the exponent: sea p un primo, x, y enteros y n un entero positivo tal que $p | x - y$ pero $p \nmid x$ ni $p \nmid y$. Entonces:

- Si p es impar: $v_p(x^n - y^n) = v_p(x - y) + v_p(n)$
- Si $p = 2$ y n es par: $v_p(x^n - y^n) = v_p(x - y) + v_p(n) + v_p(x + y) - 1$

donde $v_p(n)$ es el exponente de p en la factorización en primos de n .

- Suma de dos cuadrados: sea $\chi_4(n)$ una función multiplicativa igual a 1 si $n \equiv 1 \pmod{4}$, -1 si $n \equiv 3 \pmod{4}$ y cero en otro caso. Entonces, el número de soluciones enteras (a, b) de la ecuación $a^2 + b^2 = n$ es $4(\chi_4 * 1)(n) = 4 \sum_{d|n} \chi_4(d)$.

- Teorema de Lucas:

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{k_i} \pmod{p}$$

$$m = \sum_{i=0}^k m_i p^i, \quad n = \sum_{i=0}^k n_i p^i$$

$$0 \leq m_i, n_i < p$$

- Sean $a, b, c \in \mathbb{Z}$ con $a \neq 0$ y $b \neq 0$. La ecuación $ax + by = c$ tiene como soluciones:

$$x = \frac{x_0 c - b k}{d}$$

$$y = \frac{y_0 c + a k}{d}$$

para toda $k \in \mathbb{Z}$ si y solo si $d|c$, donde $ax_0 + by_0 = \gcd(a, b) = d$ (Euclides extendido). Si a y b tienen el mismo signo, hay exactamente $\max\left(\left\lfloor \frac{x_0 c}{|b|} \right\rfloor + \left\lfloor \frac{y_0 c}{|a|} \right\rfloor + 1, 0\right)$ soluciones no negativas. Si tienen el signo distinto, hay infinitas soluciones no negativas.

- Dada una función aritmética f con $f(1) \neq 0$, existe otra función aritmética g tal que $(f * g)(n) = e(n)$, dada por:

$$g(1) = \frac{1}{f(1)}$$

$$g(n) = -\frac{1}{f(1)} \sum_{d|n, d < n} f\left(\frac{n}{d}\right) g(d), \quad n > 1$$

- Sean $h(n) = \sum_{k=1}^n f\left(\left\lfloor \frac{n}{k} \right\rfloor\right) g(k)$, $G(n) = \sum_{k=1}^n g(k)$ y $m = \lfloor \sqrt{n} \rfloor$, entonces:

$$h(n) = \sum_{k=1}^{\lfloor n/m \rfloor} f\left(\left\lfloor \frac{n}{k} \right\rfloor\right) g(k) + \sum_{k=1}^{m-1} \left(G\left(\left\lfloor \frac{n}{k} \right\rfloor\right) - G\left(\left\lfloor \frac{n}{k+1} \right\rfloor\right) \right) f(k)$$

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

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

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

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

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

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

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

11.18 Primos

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

11.19 Números primos de Mersenne

Números primos de la forma $M_p = 2^p - 1$ con p primo. Todos los números perfectos pares son de la forma $2^{p-1}M_p$ y viceversa.

Los primeros 47 valores de p son: 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951, 30402457, 32582657, 37156667, 42643801, 43112609.

11.20 Números primos de Fermat

Números primos de la forma $F_p = 2^{2^p} + 1$, solo se conocen cinco: 3, 5, 17, 257, 65537.

Un polígono de n lados es construible si y solo si n es el producto de algunas potencias de dos y distintos primos de Fermat.

12 More Topics

12.1 2D Prefix Sum

```

1 int b[MAXN][MAXN];
2 int a[MAXN][MAXN];
3 for (int i = 1; i <= N; i++) {
4     for (int j = 1; j <= N; j++) {
5         b[i][j] = a[i][j] + b[i - 1][j] +
6                     b[i][j - 1] - b[i - 1][j - 1];
7     }
8 }
9 for (int q = 0; q < Q; q++) {

```

```

10     int from_row, to_row, from_col, to_col;
11     cin >> from_row >> from_col >> to_row >> to_col;
12     cout << b[to_row][to_col] - b[from_row - 1][to_col] -
13         b[to_row][from_col - 1] +
14         b[from_row - 1][from_col - 1]
15     << '\n';
16 }

```

12.2 CDQ Divide and Conquer

```

1 // given x[i], y[i], z[i]
2 // count for each i number of j such that x[j] > x[i] && y[j] > x[i] &&
3 // z[j] > z[i]
4
5 #include <bits/stdc++.h>
6 using namespace std;
7 typedef long long ll;
8
9 struct Pt {
10     int x, y, z, i;
11 };
12
13 const int N = (int)1e5 + 5;
14
15 struct BIT {
16     int b[N], n;
17
18     void init(int _n) {
19         n = _n;
20         for(int i = 0 ; i <= n ; ++i) b[i] = 0;
21     }
22
23     inline int lowbit(int x) { return x & (-x); }
24     void update(int x, int v) {
25         for(int i = x ; i <= n ; i += lowbit(i)) b[i] += v;
26     }
27
28     int query(int x) {
29         int ans = 0;
30         for(int i = x ; i > 0 ; i -= lowbit(i)) ans += b[i];
31     }
32 } bit;
33
34 vector<Pt> v;

```

```

33 int n, ans[N];
34
35 void cdq(int l, int r) {
36     if(l + 1 == r) return;
37     int m = (l + r) >> 1;
38     cdq(l, m); cdq(m, r);
39     int a = l, b = m, sum = 0;
40     // need to record the modifications on BIT in order to reset it. The
41     // complexity will be $O(N^2)$ if we resetting it brute-forcely.
42     vector<int> record;
43     // temporary array for merge sort
44     vector<Pt> tmp;
45     while(a < m && b < r) {
46         if(v[a].y > v[b].y) bit.update(v[a].z, 1), sum++, record.push_back(v
47             [a].z), tmp.push_back(v[a++]);
48         else ans[v[b].i] += sum - bit.query(v[b].z), tmp.push_back(v[b++]);
49     }
50     while(a < m) tmp.push_back(v[a++]);
51     while(b < r) ans[v[b].i] += sum - bit.query(v[b].z), tmp.push_back(v[b
52         ++]);
53     for(int i = l ; i < r ; ++i) v[i] = tmp[i - 1];
54     // reset BIT
55     for(auto i : record) bit.update(i, -1);
56     // release used memory
57     vector<int>().swap(record);
58     vector<Pt>().swap(tmp);
59 }
60
61 void init() {
62     cin >> n;
63     for(int i = 0 ; i < n ; ++i) {
64         int a, b, c; cin >> a >> b >> c;
65         v.push_back({a, b, c, i});
66     }
67     bit.init(n);
68 }
69 void solve() {
70     // As we require > not >=
71     sort(v.begin(), v.end(), [&](auto a, auto b) {
72         return (a.x == b.x ? (a.y == b.y ? a.z < b.z : a.y < b.y) : a.x >
73             b.x);
74     });
75     cdq(0, n);

```

```

72     for(int i = 0 ; i < n ; ++i) cout << ans[i] << '\n';
73 }
74
75 int main() {
76     ios_base::sync_with_stdio(0), cin.tie(0);
77     init();
78     solve();
79 }

```

12.3 Count Segments Around Index

```

1 // count of segments that include i with at most size k, at most l to
2 // the left and at most r to the right
3
4 auto calc = [&](ll val, ll mxx, ll left, ll right) -> ll{
5     left = min(left, mxx - 1);
6     right = min(right, mxx - 1);
7     ll mn = min(left, right);
8     ll mx = max(left, right);
9     mn++;
10    ll res = (mn * (mn + 1)) / 2;
11    mn--;
12    ll dif = mx - mn;
13    res += dif * (mn + 1);
14    mxx = min(mxx, left + right + 1);
15    ll dif1 = mxx - mx - 1;
16    res += (mn * (mn + 1)) / 2;
17    dif1 = mn - dif1;
18    res -= (dif1 * (dif1 + 1)) / 2;
19    return res;
}

```

12.4 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;

```

12.5 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 }
```

12.6 Directed MST

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 const int N = 3e5 + 9;
5
6 const long long inf = 1e18;
7
8 template<typename T> struct PQ {
9     long long sum = 0;
10    priority_queue<T, vector<T>, greater<T>> Q;
11    void push(T x) { x.w -= sum; Q.push(x); }
12    T pop() { auto ans = Q.top(); Q.pop(); ans.w += sum; return ans; }
13    int size() { return Q.size(); }
14    void add(long long x) { sum += x; }
15    void merge(PQ &&x) {
16        if (size() < x.size()) swap(sum, x.sum), swap(Q, x.Q);
17        while (x.size()) {
18            auto tmp = x.pop();
19            tmp.w -= sum;
20            Q.push(tmp);
21        }
22    }
23 };
24 struct edge {
25     int u, v; long long w;
26     bool operator > (const edge &rhs) const { return w > rhs.w; }
27 };
28 struct DSU {
29     vector<int> par;
30     DSU (int n) : par(n, -1) {}
31     int root(int i) { return par[i] < 0 ? i : par[i] = root(par[i]); }
```

```

32     void set_par(int c, int p) { par[c] = p; }
33 };
34 // returns parents of each vertex
35 // each edge should be distinct
36 // it assumes that a solution exists (all vertices are reachable from
37 // root)
38 // 0 indexed
39 // Takes ~300ms for n = 2e5
40 vector<int> DMST(int n, int root, const vector<edge> &edges) {
41     vector<int> u(2 * n - 1, -1), par(2 * n - 1, -1);
42     edge par_edge[2 * n - 1];
43     vector<int> child[2 * n - 1];
44
45     PQ<edge> Q[2 * n - 1];
46     for (auto e : edges) Q[e.v].push(e);
47     for (int i = 0; i < n; i++) Q[(i + 1) % n].push({i, (i + 1) % n, inf});
48
49     int super = n;
50     DSU dsu(2 * n - 1);
51     int head = 0;
52     while (Q[head].size()) {
53         auto x = Q[head].pop();
54         int nxt_root = dsu.root(x.u);
55         if (nxt_root == head) continue;
56         u[head] = nxt_root;
57         par_edge[head] = x;
58         if (u[nxt_root] == -1) head = nxt_root;
59         else {
60             int j = nxt_root;
61             do {
62                 Q[j].add(-par_edge[j].w);
63                 Q[super].merge(move(Q[j]));
64                 assert(u[j] != -1);
65                 child[super].push_back(j);
66                 j = dsu.root(u[j]);
67             } while (j != nxt_root);
68             for (auto u : child[super]) par[u] = super, dsu.set_par(u, super);
69             head = super++;
70     }
71     vector<int> res(2 * n - 1, -1);
72     queue<int> q; q.push(root);
```

```

73     while (q.size()) {
74         int u = q.front();
75         q.pop();
76         while (par[u] != -1) {
77             for (auto v : child[par[u]]) {
78                 if (v != u) {
79                     res[par_edge[v].v] = par_edge[v].u;
80                     q.push(par_edge[v].v);
81                     par[v] = -1;
82                 }
83             }
84             u = par[u];
85         }
86     }
87     res[root] = root; res.resize(n);
88     return res;
89 }
90 int32_t main() {
91     ios_base::sync_with_stdio(0);
92     cin.tie(0);
93     int n, m, root; cin >> n >> m >> root;
94     vector<edge> edges(m);
95     for (auto &e : edges) cin >> e.u >> e.v >> e.w;
96     auto res = DMST(n, root, edges);
97
98     unordered_map<int, int> W[n];
99     for (auto u : edges) W[u.v][u.u] = u.w;
100
101    long long ans = 0;
102    for (int i = 0; i < n; i++) if (i != root) ans += W[i][res[i]];
103    cout << ans << '\n';
104    for (auto x : res) cout << x << ' ' ; cout << '\n';
105    return 0;
106 }
107 // https://judge.yosupo.jp/problem/directedmst
108 // http://www.cs.tau.ac.il/~zwick/grad-algo-13/directed-mst.pdf

```

12.7 GCD Convolution

```

1 /*
2  * GCD Convolution via Multiple Zeta and Mobius Transforms
3  * -----
4  * Given two sequences A[0..n] and B[0..n], computes C where

```

```

5     C[d] = sum_{i, j: gcd(i, j) = d} A[i] * B[j]
6     Indexing: 1-based, vectors of size n+1 (ignore index 0)
7     Bounds:
8         - i, j, d in [1..n]
9     Time Complexity: O(n log log n + n log n) per convolution
10    Space Complexity: O(n)
11    Steps:
12        1. Multiple Zeta Transform: for each prime p, v[i] += v[i*p]
13        2. Pointwise multiply transformed A and B
14        3. Multiple Mobius Transform: for each prime p, v[i] -= v[i*p]
15    */
16    // Enumerate primes up to n in O(n)
17    vector<int> PrimeEnumerate(int n) {
18        vector<bool> isPrime(n+1, true);
19        vector<int> P;
20        for (int i = 2; i <= n; i++) {
21            if (isPrime[i]) P.push_back(i);
22            for (int p : P) {
23                if (i * p > n) break;
24                isPrime[i*p] = false;
25                if (i % p == 0) break;
26            }
27        }
28        return P;
29    }
30    // Multiple Zeta Transform (over divisibility poset)
31    template<typename T>
32    void MultipleZetaTransform(vector<T>& v) {
33        int n = (int)v.size() - 1;
34        for (int p : PrimeEnumerate(n)) {
35            for (int i = n/p; i >= 1; i--) {
36                v[i] += v[i * p];
37            }
38        }
39    }
40    // Multiple Mobius Transform (inverse of zeta)
41    template<typename T>
42    void MultipleMobiusTransform(vector<T>& v) {
43        int n = (int)v.size() - 1;
44        for (int p : PrimeEnumerate(n)) {
45            for (int i = 1; i * p <= n; i++) {
46                v[i] -= v[i * p];
47            }
48        }
49    }

```

```

48     }
49 }
50 // GCD convolution of A and B, both size n+1
51 template<typename T>
52 vector<T> GCDConvolution(vector<T> A, vector<T> B) {
53     int n = (int)A.size() - 1;
54     MultipleZetaTransform(A);
55     MultipleZetaTransform(B);
56     for (int i = 1; i <= n; i++) {
57         A[i] *= B[i]; // pointwise multiplication
58     }
59     MultipleMobiusTransform(A);
60     return A; // result C[0..n], where C[d] = sum_{gcd(i,j)=d} A[i]*B[j]
61 }
62 // Example usage
63 vector<long long> A(n+1), B(n+1);
64 for (int i = 1; i <= n; i++) cin >> A[i];
65 for (int i = 1; i <= n; i++) cin >> B[i];
66 auto C = GCDConvolution<long long>(A, B);
67 for (int d = 1; d <= n; d++){
68     cout << C[d] << (d==n?'\n':'_');
69 }

```

12.8 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 }

```

```

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; }

```

12.9 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) break;
6     }
7 }

```

12.10 LCM Convolution

```

1 /*
2  LCM Convolution via Divisor Zeta and Mobius Transforms
3  -----
4  Given two sequences A[1..n] and B[1..n], computes C where
5  C[d] = sum_{i, j: lcm(i, j) = d} A[i] * B[j]
6  Indexing: 1-based (vectors of size n+1, ignore index 0)
7  Bounds:
8      - i, j, d in [1..n]

```

```

9  Time Complexity: O(n log log n + n log n)
10 Space Complexity: O(n)
11 Steps:
12   1. Divisor Zeta Transform: for each prime p, v[i*p] += v[i]
13   2. Pointwise multiply transformed A and B
14   3. Divisor Mobius Transform: for each prime p, v[i*p] -= v[i]
15 */
16 // Sieve primes up to n in O(n)
17 vector<int> PrimeEnumerate(int n) {
18     vector<bool> isPrime(n+1, true);
19     vector<int> P;
20     for (int i = 2; i <= n; i++) {
21         if (isPrime[i]) P.push_back(i);
22         for (int p : P) {
23             if (i * p > n) break;
24             isPrime[i*p] = false;
25             if (i % p == 0) break;
26         }
27     }
28     return P;
29 }
30 // Divisor Zeta Transform: v[d] = sum_{d|k} original_v[k]
31 template<typename T>
32 void DivisorZetaTransform(vector<T>& v) {
33     int n = (int)v.size() - 1;
34     auto primes = PrimeEnumerate(n);
35     for (int p : primes) {
36         for (int i = 1; i * p <= n; i++) {
37             v[i * p] += v[i];
38         }
39     }
40 }
41 // Divisor Mobius Transform (inverse of zeta)
42 template<typename T>
43 void DivisorMobiusTransform(vector<T>& v) {
44     int n = (int)v.size() - 1;
45     auto primes = PrimeEnumerate(n);
46     for (int p : primes) {
47         for (int i = n / p; i >= 1; i--) {
48             v[i * p] -= v[i];
49         }
50     }
51 }
```

```

52 // LCM convolution of A and B (each size n+1)
53 template<typename T>
54 vector<T> LCMConvolution(vector<T> A, vector<T> B) {
55     int n = (int)A.size() - 1;
56     DivisorZetaTransform(A);
57     DivisorZetaTransform(B);
58     for (int d = 1; d <= n; d++) {
59         A[d] *= B[d]; // pointwise multiplication
60     }
61     DivisorMobiusTransform(A);
62     return A; // result C[1..n], C[d] = sum_{lcm(i,j)=d} A[i]*B[j]
63 }
64 // Example usage
65 vector<long long> A(n+1), B(n+1);
66 for (int i = 1; i <= n; i++) cin >> A[i];
67 for (int i = 1; i <= n; i++) cin >> B[i];
68 auto C = LCMConvolution<long long>(A, B);
69 for (int d = 1; d <= n; d++){
70     cout << C[d] << (d==n?'\\n':' ');
71 }
```

12.11 Manhattan MST

```

1 /*
2  Manhattan MST Edge Generation
3  -----
4  Given n points ps[0..n-1], returns a list of candidate edges
5   (weight, u, v) for the Manhattan MST, where
6   weight = |x[u]-x[v]| + |y[u]-y[v]|.
7  Indexing: 0-based for points and returned edges
8  Time Complexity: O(n log n)
9 */
10 struct Point {
11     ll x, y;
12 };
13
14 vector<tuple<ll,int,int>> manhattan_mst_edges(vector<Point> ps) {
15     int n = ps.size();
16     vector<int> order(n);
17     iota(order.begin(), order.end(), 0);
18     vector<tuple<ll,int,int>> edges;
19     edges.reserve(n * 4);
20     // Repeat for 4 orientations

```

```

21  for (int r = 0; r < 4; r++) {
22    // Sort by x+y ascending
23    sort(order.begin(), order.end(), [&](int i, int j) {
24      return ps[i].x + ps[i].y < ps[j].x + ps[j].y;
25    });
26    // Active map: key = x-coordinate, value = point index
27    map<ll,int,greater<ll>> active;
28    for (int i : order) {
29      // For all active points with x >= ps[i].x
30      for (auto it = active.lower_bound(ps[i].x); it != active.end();
31          active.erase(it++)) {
32        int j = it->second;
33        if (ps[i].x - ps[i].y > ps[j].x - ps[j].y) break;
34        assert(ps[i].x >= ps[j].x && ps[i].y >= ps[j].y);
35        edges.emplace_back({(ps[i].x - ps[j].x) + (ps[i].y - ps[j].y), i
36                            , j});
37      }
38      active[ps[i].x] = i;
39    }
40    // Rotate points 90 degrees: (x,y) -> (y,-x)
41    for (auto &p : ps) {
42      ll tx = p.x;
43      p.x = p.y;
44      p.y = -tx;
45    }
46  }
47  return edges;
}

```

12.12 Max Manhattan Distance

```

/*
Max Manhattan distance
-----
Finds maximum manhattan distance for any two points with d dimensions.
Generalized code for d dimensions.
Time Complexity: (n * 2 ^ d * d)
*/
long long ans = 0;
for (int msk = 0; msk < (1 << d); msk++) {
  long long mx = LLONG_MIN, mn = LLONG_MAX;

```

```

13   for (int i = 0; i < n; i++) {
14     long long cur = 0;
15     for (int j = 0; j < d; j++) {
16       if (msk & (1 << j)) cur += p[i][j];
17       else cur -= p[i][j];
18     }
19     mx = max(mx, cur);
20     mn = min(mn, cur);
21   }
22   ans = max(ans, mx - mn);
23 }

```

12.13 Mo

```

/*
Mo's Algorithm (Sqrt Decomposition for Offline Queries)
-----
Problem: Answer q range queries [L, R] over array of length n
          using add/remove operations efficiently
Indexing: 0-based
Bounds:
  - arr[0..n-1]
  - queries input as 1-based and converted to 0-based
Time Complexity: O((n + q) * sqrt(n)) per query batch
Space Complexity: O(n + q)
Usage:
  - Fill in logic for add/remove operations (update cur)
  - Fill answers[] indexed by original query order
*/
const int MAXN = 200500;
ll n, q;
ll arr[MAXN]; // input array
ll cnt[1000005]; // frequency count
ll answers[MAXN]; // output array
ll cur = 0; // current query result
ll BLOCK_SIZE;
pair< pair<ll, ll>, ll> queries[MAXN]; // {{L, R}, query_index}
// Sort by block and then by R
inline bool cmp(const pair< pair<ll, ll>, ll> &x, const pair< pair<ll,
  ll>, ll> &y) {

```

```

29    ll block_x = x.first.first / BLOCK_SIZE;
30    ll block_y = y.first.first / BLOCK_SIZE;
31    if (block_x != block_y) return block_x < block_y;
32    return x.first.second < y.first.second;
33 }
34
35 int main() {
36     cin >> n >> q;
37     BLOCK_SIZE = sqrt(n);
38     for (int i = 0; i < n; i++) cin >> arr[i];
39     for (int i = 0; i < q; i++) {
40         int l, r;
41         cin >> l >> r;
42         --l; --r; // convert to 0-based
43         queries[i] = {{l, r}, i};
44     }
45     sort(queries, queries + q, cmp);
46     ll l = 0, r = -1;
47     for (int i = 0; i < q; i++) {
48         int left = queries[i].first.first;
49         int right = queries[i].first.second;
50         // 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         }
56         // Expand to left
57         while (l < left) {
58             // Operations to remove arr[l], implement exactly here
59             l++;
60         }
61         // Shrink from left
62         while (l > left) {
63             l--;
64             // Operations to add arr[l], implement exactly here
65         }
66         answers[queries[i].second] = cur; // Current answer
67     }
68 }
```

```

72     for (int i = 0; i < q; i++) cout << answers[i] << '\n';
73 }
```

12.14 MOD INT

```

1 template<int MOD>
2 struct ModInt {
3     ll v;
4     ModInt(ll _v = 0) {v = (-MOD < _v && _v < MOD) ? _v : _v % MOD; if (v
5         < 0) v += MOD;}
6     ModInt& operator += (const ModInt &other) {v += other.v; if (v >= MOD)
7         v -= MOD; return *this;}
8     ModInt& operator -= (const ModInt &other) {v -= other.v; if (v < 0) v
9         += MOD; return *this;}
10    ModInt& operator *= (const ModInt &other) {v = v * other.v % MOD;
11        return *this;}
12    ModInt& operator /= (const ModInt &other) {return *this *= inverse(
13        other);}
14    bool operator == (const ModInt &other) const {return v == other.v;}
15    bool operator != (const ModInt &other) const {return v != other.v;}
16    friend ModInt operator + (ModInt a, const ModInt &b) {return a += b;}
17    friend ModInt operator - (ModInt a, const ModInt &b) {return a -= b;}
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 - (const ModInt &a) {return 0 - a;}
21    friend ModInt power(ModInt a, ll b) {ModInt ret(1); while (b > 0) {if
22        (b & 1) ret *= a; a *= a; b >>= 1;} return ret;}
23    friend ModInt inverse(ModInt a) {return power(a, MOD - 2);}
24    friend istream& operator >> (istream &is, ModInt &m) {is >> m.v; m.v =
25        (-MOD < m.v && m.v < MOD) ? m.v : m.v % MOD; if (m.v < 0) m.v +=
26        MOD; return is;}
27    friend ostream& operator << (ostream &os, const ModInt &m) {return os
28        << m.v;}
29 };
30 }
```

12.15 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 }

```

12.16 Next and Previous Smaller/Greater Element

```

1 /*
2  Next and Previous Smaller / Greater Elements
3 -----
4 Given: array a[0..n-1]
5 nextSmaller[i]: index of next element smaller than a[i], or n if none
6 prevSmaller[i]: index of previous element smaller than a[i], or -1 if
7     none
8 For GREATER, replace a[s.top()] < a[i] with a[s.top()] > a[i]
9 Indexing: 0-based
10 */
11 vector<int> nextSmaller(vector<int> a, int n) {
12     stack<int> s;
13     vector<int> res(n, n); // default: no smaller to right
14     for (int i = 0; i < n; i++) {
15         while (!s.empty() && a[s.top()] > a[i]) {
16             res[s.top()] = i;
17             s.pop();
18         }
19         s.push(i);
20     }
21     return res;
22 }
23 vector<int> prevSmaller(vector<int> a, int n) {
24     stack<int> s;
25     vector<int> res(n, -1); // default: no smaller to left
26     for (int i = n - 1; i >= 0; i--) {
27         while (!s.empty() && a[s.top()] > a[i]) {
28             res[s.top()] = i;
29             s.pop();
30         }
31         s.push(i);
32     }
33     return res;
34 }

```

12.17 Parallel Binary Search

```

1 /*
2  Parallel Binary Search
3 -----
4 Solves: k independent queries where each answer lies in range [lo, hi]
5 - At each iteration, test the midpoint for a batch of queries
6 Indexing: 0-based
7 Bounds:
8     - i in [0, m-1] for the structure version
9     - q in [0, k-1] for each of the k queries
10 Time Complexity: O((m + k) * log M)
11 Space Complexity: O(k + m)
12 You must:
13     - Reset data structures before each iteration
14     - Implement apply_change(i) and check_query(q)
15 */
16 int lo[MAXN], hi[MAXN]; // binary search bounds for each query
17 vector<int> tocheck[MAXN]; // tocheck[mid] = list of queries to check
18     at mid
19
20 bool done = false;
21 while (!done) {
22     done = true;
23     // Reset changes of structure to 0 before each iteration
24     // Assign queries to current midpoint
25     for (int q = 0; q < k; q++) {
26         if (lo[q] < hi[q]) {
27             tocheck[(lo[q] + hi[q]) / 2].push_back(q);
28         }
29     }
30     // Apply changes and evaluate queries
31     for (int i = 0; i < m; i++) {
32         // Apply change for ith query to your data structure
33         apply_change(i);
34         for (int q : tocheck[i]) {
35             done = false; // at least one query is not done
36             // Evaluate query q using the updated structure
37             if (operationToCheck(q)) {
38                 hi[q] = i; // condition is true, try earlier
39             } else {
40                 lo[q] = i + 1; // condition false, try later
41             }
42         }
43     }
44 }

```

```

41   }
42   }
43 }
```

12.18 Random Number Generators

```

1 //to avoid hacks
2 mt19937 mt(chrono::steady_clock::now().time_since_epoch().count());
3 mt19937_64 mt(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> rng(0, numeric_limits<int>::max());
10
11 // then for generating random values just do
12 int random_value = rng(mt);
13
14 // if you do not want to create variable
15 uniform_int_distribution<int>(0, n)(mt)
16
17 // other distributions
18 // Normal distribution: mean = 0.0, stddev = 1.0
19 normal_distribution<double> normal_dist(0.0, 1.0);
20
21 // Exponential distribution: lambda = 1.5
22 exponential_distribution<double> exp_dist(1.5);
23
24 // for shuffling array
25 shuffle(permutation.begin(), permutation.end(), rng);
```

12.19 setprecision

```

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

12.20 Ternary Search

```

1 /*
2  Ternary Search (for continuous unimodal functions)
3  -----
4  Finds the maximum (or minimum) value of a unimodal function f(x) in [l
5      , r]
6  Indexing: continuous domain (double)
```

```

6 Bounds:
7     - Search range is [l, r]
8     - Precision controlled by eps
9 Time Complexity: O(log((r - l) / eps))
10    - For minimum, reverse the condition (if f1 > f2 then l = m1)
11 */
12 double ternary_search(double l, double r) {
13     double eps = 1e-9; // precision error
14     while (r - l > eps) {
15         double m1 = l + (r - l) / 3;
16         double m2 = r - (r - l) / 3;
17         double f1 = f(m1); // evaluate function at m1
18         double f2 = f(m2); // evaluate function at m2
19         if (f1 < f2)
20             l = m1; // move left bound up if increasing
21         else
22             r = m2; // move right bound down if decreasing
23     }
24     return f(l); // return value at the best point
25 }
```

12.21 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)) hi = mid;
5     else lo = mid;
6 }
7 //lo + 1 is the answer
```

12.22 XOR Convolution

```

1 /*
2  Fast Walsh-Hadamard Transform (FWHT) for XOR Convolution
3  -----
4  Given two arrays A[0..N-1], B[0..N-1], with N = 1<<k (power of two).
5  Computes C[d] = sum_{i xor j = d} A[i] * B[j].
6  Indexing: 0-based
7  Bounds:
8      - N must be a power of two
9  Time Complexity: O(N * log(N))
10 Steps:
11     1. fwht(A, false); fwht(B, false); // forward transform
```

```

12     2. for i in [0..N]: C[i] = A[i] * B[i]
13         3. fwht(C, true);           // inverse transform
14     */
15 void FWHT (int A[], int k, int inv) {
16     for (int j = 0; j < k; j++)
17         for (int i = 0; i < (1 << k); i++)
18             if (~i & (1 << j)) {
19                 int p0 = A[i];
20                 int p1 = A[i | (1 << j)];
21                 A[i] = p0 + p1;
22                 A[i | (1 << j)] = p0 - p1;
23                 if (inv) {
24                     A[i] /= 2;
25                     A[i | (1 << j)] /= 2;
26                 }
27             }
28 }
29 void XOR_conv (int A[], int B[], int C[], int k) {
30     FWHT(A, k, false);
31     FWHT(B, k, false);
32     for (int i = 0; i < (1 << k); i++)
33         C[i] = A[i] * B[i];
34     FWHT(A, k, true);
35     FWHT(B, k, true);
36     FWHT(C, k, true);
37 }
38 // Example usage:
39 int A[1 << 20], B[1 << 20], C[1 << 20];
40 XOR_conv(A, B, C, 20);
41 for (int d = 0; d < (1<<20); d++){
42     cout << C[d] << (d==(1<<20)?'\n':'_');
43 }

```

12.23 XOR Basis

```

1 /*
2  XOR Basis (Linear Basis over GF(2))
3  -----
4  Supports insertion of numbers into a basis and querying the maximum
5  XOR of any subset with a given value.
6
7  Bit Width: up to D bits (here D = 60 for 64-bit integers)
8  Indexing: basis[0] is for bit 0 (least significant), basis[D-1] for

```

```

9     bit D-1
10    Bounds:
11        - Values x inserted must satisfy 0 <= x < 2^D
12    Time Complexity:
13        - insert(x): O(D)
14        - getMax(x): O(D)
15    Space Complexity: O(D)
16
17    Notes:
18        - We maintain one basis vector per bit position.
19        - On insert, we reduce x by existing basis vectors, then store it
20          in the highest bit it still has.
21        - To query max XOR, we greedily try to xor with basis vectors
22          from highest bit down.
23 */
24 const int D = 31; // Maximum number of bits in the numbers
25 int basis[D]; // basis[i] keeps the mask of the vector whose f value is
26           i
27 int sz; // Current size of the basis
28
29 void insertVector(int mask) {
30     //turn for around if you want max xor
31     for (int i = 0; i < D; i++) {
32         if ((mask & 1 << i) == 0) continue; // continue if i != f(mask)
33         if (!basis[i]) { // If there is no basis vector with the ith bit set
34             , then insert this vector into the basis
35             basis[i] = mask;
36             ++sz;
37             return;
38         }
39         mask ^= basis[i]; // Otherwise subtract the basis vector from this
                           vector
40     }
41 }
42
43 // V2: If you dont need the basis sorted.
44 vector<ll> basis;
45 void add(ll x)
46 {
47     for (int i = 0; i < basis.size(); i++)
48     {
49         x = min(x, x ^ basis[i]);
50     }
51 }

```

```

48 if (x != 0)
49 {
50     basis.pb(x);
51 }
52 }
```

12.24 XOR Basis Online

```

1 int rebuild_and_delete (int id) {
2     int pos = 0, mn = inf, p2 = 0;
3     for (int i = 0; i < basis.size(); i++) {
4         if (basis[i].id == id) {
5             pos = i;
6         }
7     }
8     int bits = 0;
9     for (int i = 0; i < basis.size(); i++) {
10        if (basis[i].mask & (1 << pos)) {
11            if (mn > basis[i].high) {
12                mn = basis[i].high;
13                p2 = i;
14            }
15            bits ^= 1 << basis[i].high;
16        }
17    }
18
19    if (p2 != pos) {
20        swap(basis[p2].id, basis[pos].id);
21        for (auto &i : basis) {
22            swap_bits(i.mask, pos, p2); // just swaps pos-th and p2-th bit
23            in i.mask
24        }
25        pos = p2;
26    }
27    for (int i = 0; i < basis.size(); i++) {
28        if (i != pos) {
29            if (basis[i].mask & (1 << pos)) {
30                basis[i].val ^= basis[pos].val;
31                basis[i].mask ^= basis[pos].mask;
32            }
33        }
34    }
35    int good = (1 << pos) - 1;
```

```

35     int other = ((1 << len(basis)) - 1) ^ (good | (1 << pos));
36     basis.erase(basis.begin() + pos);
37     for (auto &i : basis) {
38         i.mask = ((i.mask & other) >> 1) | (i.mask & good);
39     }
40     return bits;
41 }
42
```

C++

41 lines

1161 bytes

```

43
44 int get_the_same_high_bit (int bits, vector <int> &val) {
45     for (auto &i : basis) {
46         if (__builtin_popcount(val[i.id] & bits) & 1) {
47             return i.id;
48         }
49     }
50     return -1;
51 }
```

C++

8 lines

200 bytes

60 Thus, we have learned to `delete` in $O(m^2)$.

62 Here is full version of the code:

63 Code

```

64
65 // #pragma GCC optimize("Ofast", "unroll-loops")
66 // #pragma GCC target("sse", "sse2", "sse3", "ssse3", "sse4")
67
68 #include <bits/stdc++.h>
69
70 #define all(a) a.begin(), a.end()
71 #define len(a) (int)(a.size())
72 #define mp make_pair
73 #define pb push_back
74 #define fir first
75 #define sec second
76 #define fi first
```

```

78 #define se second
79
80 using namespace std;
81
82 typedef pair<int, int> pii;
83 typedef long long ll;
84 typedef long double ld;
85
86 template<typename T>
87 bool umin(T &a, T b) {
88     if (b < a) {
89         a = b;
90         return true;
91     }
92     return false;
93 }
94 template<typename T>
95 bool umax(T &a, T b) {
96     if (a < b) {
97         a = b;
98         return true;
99     }
100    return false;
101 }
102
103 #ifdef KIVI
104 #define DEBUG for (bool _FLAG = true; _FLAG; _FLAG = false)
105 #define LOG(...) print(#__VA_ARGS__ :: __VA_ARGS__) << endl
106 template <class ...Ts> auto &print(Ts ...ts) { return ((cerr << ts << " "
107     ", ...); }
108 #else
109 #define DEBUG while (false)
110 #define LOG(...)
111 #endif
112 const int max_n = -1, inf = 1000111222;
113
114
115 const int C = 20;
116
117 struct node {
118     int val, id, mask, high;
119 };
120
121
122 inline int get_high (int x) {
123     if (x == 0) {
124         return -1;
125     }
126     return 31 - __builtin_clz(x);
127 }
128
129 inline void swap_bits (int &x, int a, int b) {
130     int x1 = bool(x & (1 << a));
131     int x2 = bool(x & (1 << b));
132     x ^= (x1 ^ x2) << a;
133     x ^= (x1 ^ x2) << b;
134 }
135
136 struct xor_basis {
137     vector <node> basis;
138
139     inline bool add (int x, int id) {
140         int mask = 0;
141         for (auto &i : basis) {
142             if (umin(x, x ^ i.val)) {
143                 mask ^= i.mask;
144             }
145         }
146         if (x) {
147             mask |= 1 << len(basis);
148             for (auto &i : basis) {
149                 if (umin(i.val, i.val ^ x)) {
150                     i.mask ^= mask;
151                 }
152             }
153             basis.pb(node{val: x, id: id, mask: mask, high: get_high(x)});
154             return true;
155         }
156         return false;
157     }
158
159
160 inline int get_the_same_high_bit (int bits, const vector <int> &val) {
161     for (auto &i : basis) {
162

```

```

163     if (_builtin_popcount(val[i.id] & bits) & 1) {
164         return i.id;
165     }
166 }
167 return -1;
168 }

169
170
171 inline int rebuild_and_delete (int id) {
172     int pos = 0, mn = inf, p2 = 0;
173     for (int i = 0; i < len(basis); i++) {
174         if (basis[i].id == id) {
175             pos = i;
176         }
177     }
178     int bits = 0;
179     for (int i = 0; i < len(basis); i++) {
180         if (basis[i].mask & (1 << pos)) {
181             if (umin(mn, basis[i].high)) {
182                 p2 = i;
183             }
184             bits ^= 1 << basis[i].high;
185         }
186     }
187
188     if (p2 != pos) {
189         swap(basis[p2].id, basis[pos].id);
190         for (auto &i : basis) {
191             swap_bits(i.mask, pos, p2);
192         }
193         pos = p2;
194     }
195     for (int i = 0; i < len(basis); i++) {
196         if (i != pos) {
197             if (basis[i].mask & (1 << pos)) {
198                 basis[i].val ^= basis[pos].val;
199                 basis[i].mask ^= basis[pos].mask;
200             }
201         }
202     }
203     int good = (1 << pos) - 1;
204     int other = ((1 << len(basis)) - 1) ^ (good | (1 << pos));
205     basis.erase(basis.begin() + pos);

206     for (auto &i : basis) {
207         i.mask = ((i.mask & other) >> 1) | (i.mask & good);
208     }
209     return bits;
210 }
211
212 };
213
214 template<int max_bit> // not inclusive
215 struct xor_basis_online {
216     vector <xor_basis> baseses[max_bit + 1];
217
218     int mx;
219
220     vector <pii> where;
221     vector <int> val;
222
223     xor_basis_online() : mx(0), cur_id(0) {}
224
225     int cur_id;
226
227     inline int add (int x) {
228         val.pb(x);
229         where.pb(make_pair(-1, -1));
230         int id = cur_id++;
231         if (x == 0) {
232             return id;
233         }
234         for (int i = mx; i >= 0; i--) {
235             if (baseses[i].empty()) {
236                 continue;
237             }
238             if (baseses[i].back().add(x, id)) {
239                 baseses[i + 1].pb(baseses[i].back());
240                 baseses[i].pop_back();
241                 umax(mx, i + 1);
242                 for (auto &j : baseses[i + 1].back().basis) {
243                     where[j.id] = make_pair(i + 1, len(baseses[i + 1]) -
244                                         1);
245                 }
246             }
247         }
248     }
249
250     return id;
251 }
252
253 }
```

```

248     }
249     basises[1].pb(xor_basis());
250     basises[1].back().add(x, id);
251     where.back() = make_pair(1, len(basises[1]) - 1);
252     umax(mx, 1);
253     return id;
254 }
255
256 inline int move_back (int sz, int pos) {
257     int to = len(basises[sz]) - 1;
258     if (to == pos) {
259         return pos;
260     }
261     for (auto &i : basises[sz][pos].basis) {
262         where[i.id].second = to;
263     }
264     for (auto &i : basises[sz][to].basis) {
265         where[i.id].second = pos;
266     }
267     swap(basises[sz][pos], basises[sz][to]);
268     return to;
269 }
270
271 inline void del (int id) {
272     if (val[id] == 0) {
273         return;
274     }
275     int sz, pos;
276     tie(sz, pos) = where[id];
277     pos = move_back(sz, pos);
278     while (true) {
279         int bits = basises[sz].back().rebuild_and_delete(id);
280         int i = sz - 1;
281         while (i > 0 && basises[i].empty()) {
282             --i;
283         }
284         int new_id = -1;
285         if (i > 0) {
286             new_id = basises[i].back().get_the_same_high_bit(bits, val
287                 );
288         }
289         if (new_id == -1) {
290             if (sz > 1) {
291                 basises[sz - 1].pb(basises[sz].back());
292                 for (auto &j : basises[sz - 1].back().basis) {
293                     where[j.id] = make_pair(sz - 1, len(basises[sz -
294                         1]) - 1);
295                 }
296                 basises[sz].pop_back();
297                 if (mx > 0 && basises[mx].empty()) {
298                     --mx;
299                 }
300                 return;
301             }
302             int cur = val[new_id];
303             assert(basises[sz].back().add(cur, new_id));
304             int new_sz = i;
305             int new_pos = len(basises[i]) - 1;
306             where[new_id] = make_pair(sz, pos);
307             id = new_id;
308             sz = new_sz;
309             pos = new_pos;
310         }
311     }
312     inline int size () {
313         return mx;
314     }
315 };
316
317 int main() {
318 //    freopen("input.txt", "r", stdin);
319 //    freopen("output.txt", "w", stdout);
320
321     ios_base::sync_with_stdio(0);
322     cin.tie(0);
323
324
325 // solution to https://basecamp.eolymp.com/uk/problems/11732
326     int n, q, add;
327     cin >> n >> add;
328     vector <int> p(n);
329     xor_basis_online<19> t;
330     vector <int> now(n);
331     for (int i = 0; i < n; i++) {

```

```

332     cin >> p[i];
333     now[i] = t.add(i ^ p[i]);
334 }
335 cin >> q;
336 int ans = t.size();
337 cout << ans << '\n';
338 for (int i = 0, l, r; i < q; i++) {
339     cin >> l >> r;
340     l = (l + ans * add) % n;
341     r = (r + ans * add) % n;
342     if (l != r) {
343         t.del(now[l]);
344         t.del(now[r]);
345         swap(p[l], p[r]);
346         now[l] = t.add(l ^ p[l]);
347         now[r] = t.add(r ^ p[r]);
348     }
349     ans = t.size();
350     cout << ans << '\n';
351 }
352 }
```

13 Polynomials

13.1 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
5         // failed on
7     for (int i=0; i<(int)s.size(); i++) {
8         // evaluate c(i)
9         // delta = s_i - \sum_{j=1}^n c_j s_{i-j}
10        // if delta == 0, c(i) is correct
11        T delta = s[i];
12        for (int j=1; j<=(int)c.size(); j++)
13            delta -= c[j-1] * s[i-j]; // c_j is one-indexed, so we
14                // actually need index j - 1 in the code
15        if (delta == 0)
16            continue; // c(i) is correct, keep going
```

```

15 // now at this point, delta != 0, so we need to adjust it
16 if (f == -1) {
17     // this is the first time we're updating c
18     // s_i was the first non-zero element we encountered
19     // we make c of length i + 1 so that s_i is part of the base
20     case
21     c.resize(i + 1);
22     mt19937 rng(chrono::steady_clock::now().time_since_epoch().
23         count());
24     for (T &x : c)
25         x = rng(); // just to prove that the initial values don
26             // t matter in the first step, I will set to random
27             // values
28     f = i;
29 } else {
30     // we need to use a previous version of c to improve on this
31     // one
32     // apply the 5 steps to build d
33     // 1. set d equal to our chosen sequence
34     vector<T> d = oldC;
35     // 2. multiply the sequence by -1
36     for (T &x : d)
37         x = -x;
38     // 3. insert a 1 on the left
39     d.insert(d.begin(), 1);
40     // 4. multiply the sequence by delta / d(f + 1)
41     T df1 = 0; // d(f + 1)
42     for (int j=1; j<=(int)d.size(); j++)
43         df1 += d[j-1] * s[f+1-j];
44     assert(df1 != 0);
45     T coef = delta / df1; // storing this in outer variable so
46         // it's O(n^2) instead of O(n^2 log MOD)
47     for (T &x : d)
48         x *= coef;
49     // 5. insert i - f - 1 zeros on the left
50     vector<T> zeros(i - f - 1);
51     zeros.insert(zeros.end(), d.begin(), d.end());
52     d = zeros;
53     // now we have our new recurrence: c + d
54     vector<T> temp = c; // save the last version of c because it
55         // might have a better left endpoint
56     c.resize(max(c.size(), d.size()));
57     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 return c;
60 }
```

13.2 Binary Exp FFT

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define rep(i, a, b) for(int i = a; i < (b); ++i)
5 #define all(x) begin(x), end(x)
6 #define sz(x) (int)(x).size()
7 typedef long long ll;
8 typedef pair<int, int> pii;
9 typedef vector<int> vi;
10
11
12 typedef complex<double> C;
13 typedef vector<double> vd;
14 void fft(vector<C>& a) {
15     int n = sz(a), L = 31 - __builtin_clz(n);
16     static vector<complex<long double>> R(2, 1);
17     static vector<C> rt(2, 1); // (^ 10% faster if double)
18     for (static int k = 2; k < n; k *= 2) {
19         R.resize(n); rt.resize(n);
20         auto x = polar(1.0L, acos(-1.0L) / k);
21         rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
22     }
23     vi rev(n);
24     rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
25     rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
26     for (int k = 1; k < n; k *= 2)
27         for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
28             // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
29             // include-line
30         }
```

```

29         auto x = (double *)&rt[j+k], y = (double *)&a[i+j+k];
30         // exclude-line
31         C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
32         // exclude-line
33         a[i + j + k] = a[i + j] - z;
34         a[i + j] += z;
35     }
36 }
37 vd conv(const vd& a, const vd& b) {
38     if (a.empty() || b.empty()) return {};
39     vd res(sz(a) + sz(b) - 1);
40     int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
41     vector<C> in(n), out(n);
42     copy(all(a), begin(in));
43     rep(i,0,sz(b)) in[i].imag(b[i]);
44     fft(in);
45     for (C& x : in) x *= x;
46     rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
47     fft(out);
48     rep(i,0,sz(res)) res[i] = imag(out[i]) / (4 * n);
49     return res;
50 }
51
52 const int maxn = 1 << 20;
53
54 int main(){
55     ios_base::sync_with_stdio(false); cin.tie(NULL);
56     int n, k; cin >> n >> k;
57     vd a(1 << 10, 0);
58     for(int i = 0; i < n; i++){
59         int x; cin >> x;
60         a[x] = 1;
61     }
62     vd res(1 << 10, 0);
63     res[0] = 1;
64     auto fix_sizes = [&]() -> void{
65         while(__builtin_popcount(sz(a)) != 1){
66             a.push_back(0);
67         }
68         while(__builtin_popcount(sz(res)) != 1){
69             res.push_back(0);
70         }
71     }
72 }
```

```

70 }
71     while(sz(a) > sz(res)) res.push_back(0);
72     while(sz(res) > sz(a)) a.push_back(0);
73 };
74 while(k){
75     if(k % 2){
76         res = conv(res, a);
77     }
78     a = conv(a, a);
79     fix_sizes();
80     for(int i = 0; i < sz(a); i++){
81         if(llround(a[i]) > 0) a[i] = 1;
82         else a[i] = 0;
83         if(llround(res[i]) > 0) res[i] = 1;
84         else res[i] = 0;
85     }
86     k /= 2;
87 }
88 for(int i = 0; i < sz(res); i++){
89     ll x = llround(res[i]);
90     if(x > 0){
91         cout << i << "U";
92     }
93 }
94 cout << endl;
95 }
```

13.3 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
16    if (i < j)
17        swap(a[i], a[j]);
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                w *= wlen;
29            }
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
57 }
```

```

58 //normalizing for when mult is between 2 big numbers and not polynomials
59 int carry = 0;
60 for (int i = 0; i < n; i++){
61     result[i] += carry;
62     carry = result[i] / 10;
63     result[i] %= 10;
64 }
65

```

13.4 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 << " ";

```

13.5 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

```

14 Strings

14.1 Hashed String

```

1 /*
2          Hashed string
3 -----
4 Class for hashing string. Allows retrieval of hashes of any substring
5      in the string.
6
7 Double hash or use big mod values to avoid problems with collisions
8
9 Important to take into account that the values used for the letters
10     are not 0 to 26. it is ascii characters.
11
12 Time Complexity(Construction): O(n)
13 Space Complexity: O(n)
14 */
15
16 const ll MOD = 212345678987654321LL;
17 const ll base = 33;
18
19 class HashedString {

```

```

19
20     private:
21         static const long long M = 1e9 + 9;
22         static const long long B = 9973;
23
24         static vector<long long> pow;
25
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
34             p_hash[0] = 0;
35             for (int i = 0; i < (int)s.size(); i++) {
36                 p_hash[i + 1] = ((p_hash[i] * B) % M + s[i]) % M;
37             }
38
39         }
40
41         // substring [start, end]
42         long long get_hash(int start, int end) {
43             long long raw_val =
44                 (p_hash[end + 1] - (p_hash[start] * pow[end - start + 1]));
45             return (raw_val % M + M) % M;
46         }
47
48         // substring [start, end] with 1 added at the end. Remember it is
49         // ascii characters, not the number from 1 to 26 for the hash.
50         long long get_hash(int start, int end, int l){
51             if(start > end) return 1;
52             long long val = ((p_hash[end + 1] * B) + 1 - (p_hash[start] * pow[
53                 end - start + 2]));
54             val = (val % M + M) % M;
55             return val;
56         }
57
58     };
59
60     // you cant skip this
61     vector<long long> HashedString::pow = {1};

```

14.2 KMP

```

1 /*
2          KMP
3 -----
4 Computes the prefix function for a string.

```

```

5      Maximum length of substring that ends at position i and is proper
6          prefix (not equal to string itself) of string
7          pf[i]  is the length of the longest proper prefix of the substrings
8              [0.....i]$ which is also a suffix of this substring.
9
10     For matching, one can append the string with a delimiter like $ between them
11
12     Time Complexity: O(n)
13     Space Complexity: O(n)
14 */
15
16 vector<int> KMP(string s){
17     int n=(int)s.length();
18     vector<int> pf(n, 0);
19     for(int i=1;i<n;i++){
20         int j=pf[i-1];
21         while(j>0 && s[i]!=s[j]){
22             j=pf[j-1];
23         }
24         if(s[i]==s[j]){
25             pf[i]=j+1;
26         }
27     }
28     return pf;
29 }
30
31 // Counts how many times each prefix occurs
32 // Same thing can be done for two strings but only considering indices
33 // of second string
34 vector<int> count_occurrences_of_prefixes(vector<int> pf){
35     int n=(int)pf.size();
36     vector<int> ans(n + 1);
37     for (int i = 0; i < n; i++)
38         ans[pi[i]]++;
39     for (int i = n-1; i > 0; i--)
40         ans[pi[i-1]] += ans[i];
41     for (int i = 0; i <= n; i++)
42         ans[i]++;
43 }
44
45 // 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
46 // trying to count occurrences)
47 // is very large
48 void compute_automaton(string s, vector<vector<int>>& aut) {
49     s += '#';
50     int n = s.size();
51     vector<int> pi = KMP(s);
52     aut.assign(n, vector<int>(26));
53     for (int i = 0; i < n; i++) {
54         for (int c = 0; c < 26; c++) {
55             if (i > 0 && 'a' + c != s[i])
56                 aut[i][c] = aut[pi[i-1]][c];
57             else
58                 aut[i][c] = i + ('a' + c == s[i]);
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         }
25     }
26 }

```

```

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   }
26 }
```

```

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)
```

```

12     Sliding window on lcp array (all string have to appear on the sliding
13         window and
14         the lcp of the interval will give the length of the substring that
15             appears on all strings)
16
17     Time Complexity: O(n log n)
18     Space Complexity: O(n)
19
20 */
21
22 struct SuffixArray
23 {
24     int n;
25     string t;
26     vector<int> sa, rk, lc;
27     SuffixArray(const std::string &s)
28     {
29         n = s.length();
30         t = s;
31         sa.resize(n);
32         lc.resize(n - 1);
33         rk.resize(n);
34         std::iota(sa.begin(), sa.end(), 0);
35         std::sort(sa.begin(), sa.end(), [&](int a, int b)
36                   { return s[a] < s[b]; });
37         rk[sa[0]] = 0;
38         for (int i = 1; i < n; ++i)
39             rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
40         int k = 1;
41         std::vector<int> tmp, cnt(n);
42         tmp.reserve(n);
43         while (rk[sa[n - 1]] < n - 1)
44         {
45             tmp.clear();
46             for (int i = 0; i < k; ++i)
47                 tmp.push_back(n - k + i);
48             for (auto i : sa)
49                 if (i >= k)
50                     tmp.push_back(i - k);
51             std::fill(cnt.begin(), cnt.end(), 0);
52             for (int i = 0; i < n; ++i)
53                 ++cnt[rk[i]];
54             for (int i = 1; i < n; ++i)
55                 cnt[i] += cnt[i - 1];
56             for (int i = n - 1; i >= 0; --i)
57                 sa[--cnt[rk[tmp[i]]]] = tmp[i];
58             std::swap(rk, tmp);
59             rk[sa[0]] = 0;
60             for (int i = 1; i < n; ++i)
61                 rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i - 1] + k == n || tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
62             k *= 2;
63         }
64         for (int i = 0, j = 0; i < n; ++i)
65         {
66             if (rk[i] == 0)
67             {
68                 j = 0;
69             }
69             else
70             {
71                 for (j -= j > 0; i + j < n && sa[rk[i] - 1] + j < n && s[i + j]
72                     == s[sa[rk[i] - 1] + j];)
73                     ++j;
74                 lc[rk[i] - 1] = j;
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[1], min(n-sa[1], tam));
95 } else{
96     return -1;
97 }
98 if(tmp == p){
99     return 1;
100 } else {
101     return -1;
102 }
103 }

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

127     int q; cin >> q;
128     for(int t = 0; t < q; t++) {
129         string tmp; cin >> tmp;
130         cout << SA.count(tmp) << endl;
131     }
132 }

133     return 0;
134 }

```

14.6 Suffix Automaton

```

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

// searches for the starting position in O(len(s)). Returns starting
// index of first occurrence or -1 if it does not appear.
int search(string s){
    int cur=0, i=0, n=(int)s.length();
    while(i<n){
        if(!st[cur].next.count(s[i])) return -1;
        cur=st[cur].next[s[i]];
        i++;
    }
    //sumar 2 si se quiere 1 indexado
    return st[cur].firstposition-n+1;
}

void dfs(int cur){
    cntPaths[cur]=1;
    for(auto [x, y]:st[cur].next){
        if(cntPaths[y]==0) dfs(y);
        cntPaths[cur]+=cntPaths[y];
    }
}

// Counts how many paths exist from state. How many substrings exist
// from a specific state.
// Stored in cntPaths
void countPaths(){
    dfs(0);
}

// Computes the number of times each state appears
void countOccurrences(){
    vector<pair<int, int>> a;
    for(int i=sz-1;i>0;i--){
        a.push_back({st[i].len, i});
    }
    sort(a.begin(), a.end());
    for(int i=sz-2;i>=0;i--){
        cnt[st[a[i].second].link]+=cnt[a[i].second];
    }
}

void dfs1(int cur){
    for(auto [x, y]:st[cur].next){
        if(cntSum[y]==cnt[y]) dfs1(y);
        cntSum[cur]+=cntSum[y];
    }
}

// Computes the number of times each state or any of its children appear
// in the string.
void countSumOccurrences(){
    for(int i=0;i<sz;i++){
        cntSum[i]=cnt[i];
    }
    dfs1(0);
}

// 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
166 // Computes the kth smallest string that appears on the string (without
// counting repetitions)
167 string kthSmallestDistinct(ll k){
168     string s="";
169     int cur=0;
170     while(k>0){
171         for(auto [c, y]:st[cur].next){
172             if(k>cntPaths[y]) k-=cntPaths[y];
173             else{
174                 k--;
175                 s+=c;
176                 cur=y;
177                 break;
178             }
179         }
180     }
181     return s;
182 }
183
184
185 // Precomputation to find all occurrences of a substring
186 void precompute_for_all_occurrences(){
187     for (int v = 1; v < sz; v++) {
188         st[st[v].link].inv_link.push_back(v);
189     }
190 }
191
192 // Finding all occurrences of substring in string
193 // P_length is length of substring
194 // v is state where first occurrence happens
195 // be careful as indices can appear multiple times due to clone states
196 // if you want to avoid duplicate positions utilize set or have a flag
// for each state to know if it is clone or not
197 void output_all_occurrences(int v, int P_length) {
198     cout << st[v].firstposition - P_length + 1 << endl;
199     for (int u : st[v].inv_link)
200         output_all_occurrences(u, P_length);
201 }
202
203
204 //longest common substring
205 //build automaton for s first
206 string lcs (string S, string T) {

```

```

207     int v = 0, l = 0, best = 0, bestpos = 0;
208     for (int i = 0; i < T.size(); i++) {
209         while (v && !st[v].next.count(T[i])) {
210             v = st[v].link ;
211             l = st[v].len;
212         }
213         if (st[v].next.count(T[i])) {
214             v = st [v].next[T[i]];
215             l++;
216         }
217         if (l > best) {
218             best = l;
219             bestpos = i;
220         }
221     }
222     return T.substr(bestpos - best + 1, best);
223 }

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

```

/*
-----  

     Z_function  

-----  

4 Computes the z_function for any string.  

5 ith element is equal to the greatest number of characters starting  

   from the position i that coincide with the first characters of s  

6  

7 z[i] length of the longest string that is, at the same time, a prefix  

   of s and a prefix of $s$ starting at i.  

8  

9 to compress string, one can run z_function and then find the smallest  

   i that divides n such that i + z[i] = n  

10  

11 Time Complexity: O(n)  

12 Space Complexity: O(n)  

13 */
14  

15  

16 vector<int> z_function(string s) {
17     int n = s.size();
18     vector<int> z(n);
19     int l = 0, r = 0;
20     for(int i = 1; i < n; i++) {
21         if(i < r) {
22             z[i] = min(r - i, z[i - 1]);
23         }
24         while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
25             z[i]++;
26         }
27         if(i + z[i] > r) {
```

```

28             l = i;
29             r = i + z[i];
30         }
31     }
32     return z;
33 }
34
35 // usage
36 // vector<int> z = z_function("abacaba");
37 // this will return {0, 0, 1, 0, 3, 0, 1}
38 // vector<int> z = z_function("aaaaaa");
39 // this will return {0, 4, 3, 2, 1}
40 // vector<int> z = z_function("aaabaab");
41 // this will return {0, 2, 1, 0, 2, 1, 0}
```

15 Trees

15.1 Centroid Decomposition

```

/*
-----  

     Centroid Decomposition  

-----  

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     }
68 }
```

```

66     return mini;
67 }
68
69 void build_centroid_decomp(int node = 1) {
70     int centroid = get_centroid(node, get_subtree_size(node));
71
72     for (int child : adj[centroid]) {
73         if (is_removed[child]) { continue; }
74         getDist(child, centroid, centroid);
75     }
76
77     is_removed[centroid] = true;
78
79     for (int child : adj[centroid]) {
80         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) {}
```

```

void update(int pos, ll val) {
    for (a[pos += n] = val; pos > 1; pos >>= 1) {
        a[pos / 2] = (a[pos] ^ a[pos ^ 1]);
    }
}

ll get(int l, int r) {
    ll res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l & 1) {
            res ^= a[l++];
        }
        if (r & 1) {
            res ^= a[--r];
        }
    }
    return res;
}
};

const int MAXN=500005;
vector<int> adj[MAXN];
SegmentTree st(MAXN);
int a[MAXN], sz[MAXN], to[MAXN], dpth[MAXN], s[MAXN], par[MAXN];
int cnt=0;

void dfs1(int cur, int p){
    sz[cur]=1;
    for(int x:adj[cur]){
        if(x==p) continue;
        dpth[x]=dpth[cur]+1;
        par[x]=cur;
        dfs1(x, cur);
        sz[cur]+=sz[x];
    }
}

void dfs(int cur, int p, int l){
    st.update(cnt, a[cur]);
    s[cur]=cnt++;
    to[cur]=l;
    int g=-1;
    for(int x:adj[cur]){
        if(x==p) continue;
        if(g==-1 || sz[g]<sz[x]){
            g=x;
        }
    }
    if(g==-1) return;
    dfs(g, cur, l);
    for(int x:adj[cur]){
        if(x==p || x==g) continue;
        dfs(x, cur, x);
    }
}

int query(int u, int v){
    int res=0;
    while(to[u]!=to[v]){
        if(dpth[to[u]]<dpth[to[v]]) swap(u, v);
        res^=st.get(s[to[u]], s[u]+1);
        u=par[to[u]];
    }
    if(dpth[u]>dpth[v]) swap(u, v);
    res^=st.get(s[u], s[v]+1);
    return res;
}

//alternate implementation
vector<int> parent, depth, heavy, head, pos;
int cur_pos;

int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}

```

```

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             ]);
142         res = max(res, cur_heavy_path_max);
143     }
144     if (depth[a] > depth[b])
145         swap(a, b);
146     int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
147     res = max(res, last_heavy_path_max);
148     return res;
149 }
```

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 }
```

```

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=1;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
7 distance.
8
9 To find diameter run algorithm twice, first on random vertex and then
10 on the vertex that is farthest away.
11
12 The vertex that is the farthest away from any vertex in tree must be
13 an endpoint of the diameter.
14
15 Time Complexity: O(n)
16 Space Complexity: O(n)
17 */
18
19 pair<int, int> dfs(const vector<vector<int>> &tree, int node = 1,
20     int previous = 0, int length = 0) {
21     pair<int, int> max_path = {node, length};
22     for (const int &i : tree[node]) {
23         if (i == previous) { continue; }
24         pair<int, int> other = dfs(tree, i, node, length + 1);
25         if (other.second > max_path.second) { max_path = other; }
26     }
27     return max_path;
28 }
```

```
25 | }
```

16 Scripts

16.1 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.
2           cpp
```

16.2 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
```

16.3 validate.sh

Format is validate.sh Awrong Validator 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

```

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer (lower-case)	7fa
X	Unsigned hexadecimal integer (upper-case)	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
n	Nothing printed. The argument must be a pointer to a <code>signed int</code> . The number of characters written so far is stored.	
%	A % followed by another % prints a single %	%