

Contents**1 Funciones C++**

4

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 Linked List

19

3.14 Merge Sort Tree

20

3.15 Minimum Cartesian Tree

21

3.16 Multi Ordered Set

22

3.17 Ordered Set

22

3.18 Palindromic Tree

22

3.19 Persistent Array

24

3.20 Persistent Segment Tree

25

3.21 Polynomial Queries

26

3.22 Segment Tree

27

3.23 Segment Tree 2D

28

3.24 Segment Tree Dynamic

29

3.25 Segment Tree Lazy Types

29

3.26 Segment Tree Lazy

30

3.27 Segment Tree Lazy Range Set

31

3.28 Segment Tree Lazy Range Set 2

32

3.29 Segment Tree Max Subarray Sum

34

3.30 Segment Tree Range Update

35

3.31 Segment Tree Struct Types

35

3.32 Segment Tree Struct

36

3.33 Segment Tree Walk

37

3.34 Sparse Table

38

3.35 Sparse Table 2D	38
3.36 Square Root Decomposition	40
3.37 Treap	41
3.38 Treap 2	42
3.39 Treap With Inversion	43
3.40 Treap With Inversions and Range Updates	44
4 Dynamic Programming	46
4.1 CHT Deque	46
4.2 Digit DP	47
4.3 Divide and Conquer DP	47
4.4 Edit Distance	48
4.5 Knuth's Algorithm	48
4.6 LCS	49
4.7 Line Container	49
4.8 Longest Increasing Subsequence	50
4.9 SOS DP	51
5 Flow	51
5.1 Dinic	51
5.2 Hopcroft-Karp	53
5.3 Hungarian	54
5.4 Max Flow Min Cost	55
5.5 Max Flow	56
5.6 Min Cost Max Flow	57
5.7 Push Relabel	58
6 Geometry	59
6.1 Point Struct	59
6.2 Sort Points	60
6.3 Point Struct2	60
6.4 Antipodal Pairs	61
6.5 Area and Perimeter	61
6.6 Area Union Circles	61
6.7 Centroid	62
6.8 Circle Inside Circle	62
6.9 Circle Outside Circle	62
6.10 Closest Pair of Points	62
6.11 Common Tangents	63
6.12 Convex Hull	63
6.13 Segment Intersection with Ray	63
6.14 Cut Polygon	63
6.15 Delaunay Triangulation	64

6.16 Diameter and Width	66	8 Linear Algebra	88
6.17 Distance Between Point and Circle	66	8.1 Gaussian Elimination	88
6.18 Distance Between Point and Line	66	8.2 Gaussian Elimination Modulo	89
6.19 Example of Geometry	66	8.3 Simplex	90
6.20 Half Plane Intersection	67	9 Math	92
6.21 Incircle	68	9.1 BinPow	92
6.22 Intersection of Two Circles	68	9.2 Combination Rank	92
6.23 Intersection Line and Circle	68	9.3 Chinese Remainder Theorem	92
6.24 Intersection Polygon and Circle	68	9.4 Diophantine	93
6.25 Intersection Segment and Circle	69	9.5 Discrete Logarithm	94
6.26 Line Intersection	69	9.6 Divisors	94
6.27 Minkowski Sum	69	9.7 Euler Totient (Phi)	95
6.28 Point in Circle	70	9.8 Fibonacci	95
6.29 Point in Convex Hull	70	9.9 Matrix Exponentiation	95
6.30 Point in Line	70	9.10 Miller Rabin Deterministic	96
6.31 Point in Perimeter	70	9.11 Möbius	97
6.32 Point in Polygon	70	9.12 Permutation Rank	97
6.33 Point in Segment	70	9.13 Prefix Sum Phi	97
6.34 Points Tangency	70	9.14 Sieve	98
6.35 Projection Point Circle	71	9.15 Identities	98
6.36 Segment Intersection	71	9.16 Burnside's Lemma	98
6.37 Smallest Enclosing Circle	71	9.17 Recursion	99
6.38 Smallest Enclosing Rectangle	72	9.18 Theorems	99
6.39 Vantage Point Tree	72	9.19 Sums	99
7 Graphs	73	9.20 Catalan numbers	99
7.1 2Sat	73	9.21 Cayley's formula	99
7.2 Articulation Points	74	9.22 Geometric series	99
7.3 Bellman-Ford	75	9.23 Estimates For Divisors	100
7.4 Bipartite Checker	75	9.24 Sum of divisors	100
7.5 Bipartite Maximum Matching	76	9.25 Pythagorean Triplets	100
7.6 Bipartite Minimum Maximum Matching ²	77	9.26 Derangements	100
7.7 Block Cut Tree	78	10 Game Theory	100
7.8 Blossom	79	10.1 Sprague-Grundy theorem	100
7.9 Bridges	81	11 Fórmulas y notas	100
7.10 Bridges Online	82	11.1 Números de Stirling del primer tipo	100
7.11 Dijkstra	84	11.2 Números de Stirling del segundo tipo	100
7.12 Eulerian Path	84	11.3 Números de Euler	100
7.13 Floyd-Warshall	85	11.4 Números de Catalan	101
7.14 Kruskal	85	11.5 Números de Bell	101
7.15 Marriage	86	11.6 Números de Bernoulli	101
7.16 SCC	86		
7.17 Stoer-Wagner	87		

11.7 Fórmula de Faulhaber	101	13.4 Roots NTT	120
11.8 Función Beta	101		
11.9 Función zeta de Riemann	101	14 Strings	121
11.10 Funciones generadoras	101	14.1 Hashed String	121
11.11 Números armónicos	102	14.2 KMP	122
11.12 Aproximación de Stirling	102	14.3 Least Rotation String	122
11.13 Ternas pitagóricas	102	14.4 Manacher	123
11.14 Árbol de Stern-Brocot	102	14.5 Suffix Array	123
11.15 Combinatoria	102	14.6 Suffix Automaton	125
11.16 Grafos	103	14.7 Trie Ahocorasick	128
11.17 Teoría de números	103	14.8 Z Function	129
11.18 Primos	104		
11.19 Números primos de Mersenne	104	15 Trees	129
11.20 Números primos de Fermat	104	15.1 Centroid Decomposition	129
12 More Topics	104	15.2 Heavy Light Decomposition	130
12.1 2D Prefix Sum	104	15.3 Lowest Common Ancestor (LCA)	132
12.2 Custom Comparators	105	15.4 Tree Diameter	133
12.3 Day of the Week	105		
12.4 Directed MST	105		
12.5 GCD Convolution	106		
12.6 int128	107		
12.7 Iterating Over All Subsets	108		
12.8 LCM Convolution	108		
12.9 Manhattan MST	108		
12.10 Max Manhattan Distance	109		
12.11 Mo	109		
12.12 MOD INT	110		
12.13 Next Permutation	111		
12.14 Next and Previous Smaller/Greater Element	111		
12.15 Parallel Binary Search	111		
12.16 Random Number Generators	112		
12.17 setprecision	112		
12.18 Ternary Search	112		
12.19 Ternary Search Int	112		
12.20 XOR Convolution	113		
12.21 XOR Basis	113		
12.22 XOR Basis Online	114		
13 Polynomials	118		
13.1 Berlekamp Massey	118		
13.2 FFT	119		
13.3 NTT	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

```

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             int y = in();
62             if(y < l) x = query(x, l - 1);
63             if(y > r) x = query(r + 1, y);
64             if(y == l) l = query(x, l);
65             if(y == r) r = query(r, x);
66             cout << x << endl;
67         }
68     }
69 }
```

```
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
21 const int MAXN = 100005; // Set according to constraints
22
23 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(v)] > 1) {
59             first_violate = 1;
60         }
61     }
62     void rollback(int t) {
63         if (t < 0) return;
64         if (t >= first_violate) return;
65         if (nxtupd <= t) {
66             for (int i = stack[t].t; i <= stack[t].t + 1; i++) {
67                 if (stack[i].cell) {
68                     *stack[i].cell = stack[i].value;
69                 }
70             }
71             stack[t].undo();
72             stack[t].cell = 0;
73             stack[t].value = 0;
74             stack[t].t = -1;
75         }
76     }
77     void print() {
78         for (int i = 0; i < n; i++) {
79             cout << "p[" << i << "] = " << p[i] << ", ";
80             cout << "r[" << i << "] = " << r[i] << ", ";
81             cout << "up[" << i << "] = " << up[i] << endl;
82         }
83     }
84 };

```

```

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             D.unite(e[i].first, e[i].second);
205         }
206
207         while (q--) {
208             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 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(list&& other) // Move constructor.
18 // ~list()          // Destructor, destroys the list.
19
20 // === Iterators ===
21 // begin()          // Returns an iterator to the first element of
22 // the list.
23 // end()            // Returns an iterator to the position just
24 // past the last element.
25 // rbegin()          // Returns a reverse iterator to the last
26 // element.
27 // rend()            // Returns a reverse iterator to the position
28 // just before the first element.
29 // insert(it, val)  // Inserts 'val' before the position of the
30 // iterator 'it'.
31 // erase(it)         // Removes the element at the position of the
32 // iterator 'it'.
33 // erase(first, last) // Removes a range of elements from 'first' to
34 // 'last'.
35 // push_back(val)   // Adds 'val' to the end of the list.
36 // push_front(val)  // Adds 'val' to the front of the list.
37 // pop_back()        // Removes the last element of the list.
38 // pop_front()       // Removes the first element of the list.
39 // clear()           // Removes all elements from the list.
40
41 // === Capacity ===
42 // empty()           // Returns true if the list is empty, otherwise
43 // false.
44 // size()            // Returns the number of elements in the list.
45 // max_size()         // Returns the maximum number of elements the
46 // list can hold.
47
48 // === Modifiers ===
49 // insert(it, n, val) // Inserts 'n' copies of 'val' before the
50 // position of 'it'.
51 // insert(it, first, last) // Inserts a range of elements before the
52 // position of 'it'.
53 // 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).

```

3.14 Merge Sort Tree

```

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

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

```

```

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

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

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

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

```

```

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()

```

3.15 Minimum Cartesian Tree

```

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

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

```

```

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';

```

3.16 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})

```

3.17 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

```

3.18 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     } else {
65         t[last].oc++;
66         return 0;
67     }
68     // Create new node
69     sz++;
70     last = sz;
71     t[sz].oc = 1;
72     t[sz].len = t[cur].len + 2;
73     t[cur].nxt[ch] = sz;
74     t[sz].en = pos;
75     t[sz].st = pos - t[sz].len + 1;
76
77     if (t[sz].len == 1) {
78         t[sz].link = 2;
79         t[sz].cnt = 1;
80         return 1;
81     }
82     // Compute suffix link
83     while (true) {
84         cur = t[cur].link;
85         int curlen = t[cur].len;
86         if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
87             t[sz].link = t[cur].nxt[ch];
88             break;
89         }
90     }
91     t[sz].cnt = 1 + t[t[sz].link].cnt;
92     return 1;
93 }
94 // Accumulate total occurrences for each palindrome node
95 void calc_occurrences() {
96     for (int i = sz; i >= 3; i--) {
97         t[t[i].link].oc += t[i].oc;
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 }
```

```

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 }
```

3.19 Persistent Array

```

1 /*
2 Persistent Array (via Persistent Segment Tree)
3 -----
4 Indexing: 0-based
5 Bounds: [0, n-1]
6 Time Complexity:
7   - point update: O(log n)
8   - point query: O(log n)
9 Space Complexity: O(log n) per update/version
10
11 Features:
12   - Supports point updates with full version history
13   - Allows querying any version at any index
14 */
15
16 struct Node {
17     int val;
18     Node *l, *r;
19
20     // Leaf node with value
21     Node(int x) : val(x), l(nullptr), r(nullptr) {}
22
23     // Internal node with children (value is not used)
24     Node(Node *ll, Node *rr) : val(0), l(ll), r(rr) {}
25 };
26
27 int n;
28 int a[100001];      // Initial array
29 Node *roots[100001]; // Roots of all versions (0-based)
30
31 // Build version 0 from initial array
```

```

32 Node *build(int l = 0, int r = n - 1) {
33     if (l == r) return new Node(a[l]);
34     int mid = (l + r) / 2;
35     return new Node(build(l, mid), build(mid + 1, r));
36 }
37
38 // Create new version with a[pos] = val
39 Node *update(Node *node, int pos, int val, int l = 0, int r = n - 1) {
40     if (l == r) return new Node(val);
41     int mid = (l + r) / 2;
42     if (pos <= mid)
43         return new Node(update(node->l, pos, val, l, mid), node->r);
44     else
45         return new Node(node->l, update(node->r, pos, val, mid + 1, r));
46 }
47
48 // Query value at position 'pos' in a given version (node)
49 int query(Node *node, int pos, int l = 0, int r = n - 1) {
50     if (l == r) return node->val;
51     int mid = (l + r) / 2;
52     if (pos <= mid) return query(node->l, pos, l, mid);
53     else return query(node->r, pos, mid + 1, r);
54 }
55
56 // External helper: get value at index in version
57 int get_item(int index, int version) {
58     return query(roots[version], index);
59 }
60
61 // External helper: make new version based on 'prev_version', updating
62 // one index
63 void update_item(int index, int value, int prev_version, int new_version
64 ) {
65     roots[new_version] = update(roots[prev_version], index, value);
66 }
67
68 // Initializes version 0 from given array
69 void init_arr(int nn, int *init) {
70     n = nn;
71     for (int i = 0; i < n; i++) a[i] = init[i];
72     roots[0] = build();
```

3.20 Persistent Segment Tree

```

1  /*
2   * Persistent Segment Tree (Point Updates, Range Queries)
3   -----
4   * Indexing: 1-based
5   * Bounds: [1, n]
6   * Time Complexity:
7   *   - Build: O(n)
8   *   - Point update: O(log n) -> returns new version
9   *   - Range query: O(log n)
10  *   - Copy version: O(1)
11
12 Features:
13   - Each update creates a new tree version with shared unchanged nodes
14   - Supports querying over any version
15   - Useful in rollback problems, version history, and functional
16     programming
17 */
18
19 struct Node {
20     ll val;           // segment sum
21     Node *_l, *_r;
22
23     // Leaf node
24     Node(ll x) : val(x), _l(nullptr), _r(nullptr) {}
25
26     // Internal node with children
27     Node(Node *_l, Node *_r) {
28         _l = _l;
29         _r = _r;
30         val = 0;
31         if (_l) val += _l->val;
32         if (_r) val += _r->val;
33     }
34
35     // Version clone (used when copying tree version directly)
36     Node(Node *cp) : val(cp->val), _l(cp->l), _r(cp->r) {}
37 }
38
39 int n, sz = 1;
40 ll a[200001];           // Input array (1-indexed)
41 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}

```

3.21 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 = (r - 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) {
58                lazy[i].val += D;
59                lazy1[i].val += A;
60                eval_lazy(i, tl, tr);
61            }
62            return tree[i];
63        }
64    };
65
66    segtree<ll> st;
67
68    void eval_lazy(int i, int tl, int tr) {
69        tree[i] = tree[i].op(lazy[i]);
70        if (tl != tr) {
71            lazy[2 * i + 1] = lazy[2 * i + 1].op(lazy[i]);
72            lazy[2 * i + 2] = lazy[2 * i + 2].op(lazy[i]);
73        }
74    }
75
76    ll query(int i, int l, int r) {
77        eval_lazy(i, l, r);
78        return tree[i];
79    }
80
81    ll update(int i, int l, int r, ll d, ll a) {
82        eval_lazy(i, l, r);
83        update(i, l, r, d, a);
84        eval_lazy(i, l, r);
85        return tree[i];
86    }
87
88    ll update(int i, int l, int r, ll d, ll a, ll D, ll A) {
89        eval_lazy(i, l, r);
90        update(i, l, r, d, a);
91        eval_lazy(i, l, r);
92        update(i, l, r, D, A);
93        eval_lazy(i, l, r);
94        return tree[i];
95    }
96
97    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B) {
98        eval_lazy(i, l, r);
99        update(i, l, r, d, a);
100       eval_lazy(i, l, r);
101       update(i, l, r, D, A);
102       eval_lazy(i, l, r);
103       update(i, l, r, B);
104       eval_lazy(i, l, r);
105       return tree[i];
106    }
107
108    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C) {
109        eval_lazy(i, l, r);
110        update(i, l, r, d, a);
111        eval_lazy(i, l, r);
112        update(i, l, r, D, A);
113        eval_lazy(i, l, r);
114        update(i, l, r, B, C);
115        eval_lazy(i, l, r);
116        return tree[i];
117    }
118
119    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E) {
120        eval_lazy(i, l, r);
121        update(i, l, r, d, a);
122        eval_lazy(i, l, r);
123        update(i, l, r, D, A);
124        eval_lazy(i, l, r);
125        update(i, l, r, B, C);
126        eval_lazy(i, l, r);
127        update(i, l, r, E);
128        eval_lazy(i, l, r);
129        return tree[i];
130    }
131
132    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F) {
133        eval_lazy(i, l, r);
134        update(i, l, r, d, a);
135        eval_lazy(i, l, r);
136        update(i, l, r, D, A);
137        eval_lazy(i, l, r);
138        update(i, l, r, B, C);
139        eval_lazy(i, l, r);
140        update(i, l, r, E, F);
141        eval_lazy(i, l, r);
142        return tree[i];
143    }
144
145    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G) {
146        eval_lazy(i, l, r);
147        update(i, l, r, d, a);
148        eval_lazy(i, l, r);
149        update(i, l, r, D, A);
150        eval_lazy(i, l, r);
151        update(i, l, r, B, C);
152        eval_lazy(i, l, r);
153        update(i, l, r, E, F);
154        eval_lazy(i, l, r);
155        update(i, l, r, G);
156        eval_lazy(i, l, r);
157        return tree[i];
158    }
159
160    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H) {
161        eval_lazy(i, l, r);
162        update(i, l, r, d, a);
163        eval_lazy(i, l, r);
164        update(i, l, r, D, A);
165        eval_lazy(i, l, r);
166        update(i, l, r, B, C);
167        eval_lazy(i, l, r);
168        update(i, l, r, E, F);
169        eval_lazy(i, l, r);
170        update(i, l, r, G, H);
171        eval_lazy(i, l, r);
172        return tree[i];
173    }
174
175    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I) {
176        eval_lazy(i, l, r);
177        update(i, l, r, d, a);
178        eval_lazy(i, l, r);
179        update(i, l, r, D, A);
180        eval_lazy(i, l, r);
181        update(i, l, r, B, C);
182        eval_lazy(i, l, r);
183        update(i, l, r, E, F);
184        eval_lazy(i, l, r);
185        update(i, l, r, G, H);
186        eval_lazy(i, l, r);
187        update(i, l, r, I);
188        eval_lazy(i, l, r);
189        return tree[i];
190    }
191
192    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J) {
193        eval_lazy(i, l, r);
194        update(i, l, r, d, a);
195        eval_lazy(i, l, r);
196        update(i, l, r, D, A);
197        eval_lazy(i, l, r);
198        update(i, l, r, B, C);
199        eval_lazy(i, l, r);
200        update(i, l, r, E, F);
201        eval_lazy(i, l, r);
202        update(i, l, r, G, H);
203        eval_lazy(i, l, r);
204        update(i, l, r, I, J);
205        eval_lazy(i, l, r);
206        return tree[i];
207    }
208
209    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K) {
210        eval_lazy(i, l, r);
211        update(i, l, r, d, a);
212        eval_lazy(i, l, r);
213        update(i, l, r, D, A);
214        eval_lazy(i, l, r);
215        update(i, l, r, B, C);
216        eval_lazy(i, l, r);
217        update(i, l, r, E, F);
218        eval_lazy(i, l, r);
219        update(i, l, r, G, H);
220        eval_lazy(i, l, r);
221        update(i, l, r, I, J);
222        eval_lazy(i, l, r);
223        update(i, l, r, K);
224        eval_lazy(i, l, r);
225        return tree[i];
226    }
227
228    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L) {
229        eval_lazy(i, l, r);
230        update(i, l, r, d, a);
231        eval_lazy(i, l, r);
232        update(i, l, r, D, A);
233        eval_lazy(i, l, r);
234        update(i, l, r, B, C);
235        eval_lazy(i, l, r);
236        update(i, l, r, E, F);
237        eval_lazy(i, l, r);
238        update(i, l, r, G, H);
239        eval_lazy(i, l, r);
240        update(i, l, r, I, J);
241        eval_lazy(i, l, r);
242        update(i, l, r, K, L);
243        eval_lazy(i, l, r);
244        return tree[i];
245    }
246
247    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M) {
248        eval_lazy(i, l, r);
249        update(i, l, r, d, a);
250        eval_lazy(i, l, r);
251        update(i, l, r, D, A);
252        eval_lazy(i, l, r);
253        update(i, l, r, B, C);
254        eval_lazy(i, l, r);
255        update(i, l, r, E, F);
256        eval_lazy(i, l, r);
257        update(i, l, r, G, H);
258        eval_lazy(i, l, r);
259        update(i, l, r, I, J);
260        eval_lazy(i, l, r);
261        update(i, l, r, K, L);
262        eval_lazy(i, l, r);
263        update(i, l, r, M);
264        eval_lazy(i, l, r);
265        return tree[i];
266    }
267
268    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N) {
269        eval_lazy(i, l, r);
270        update(i, l, r, d, a);
271        eval_lazy(i, l, r);
272        update(i, l, r, D, A);
273        eval_lazy(i, l, r);
274        update(i, l, r, B, C);
275        eval_lazy(i, l, r);
276        update(i, l, r, E, F);
277        eval_lazy(i, l, r);
278        update(i, l, r, G, H);
279        eval_lazy(i, l, r);
280        update(i, l, r, I, J);
281        eval_lazy(i, l, r);
282        update(i, l, r, K, L);
283        eval_lazy(i, l, r);
284        update(i, l, r, M, N);
285        eval_lazy(i, l, r);
286        return tree[i];
287    }
288
289    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O) {
290        eval_lazy(i, l, r);
291        update(i, l, r, d, a);
292        eval_lazy(i, l, r);
293        update(i, l, r, D, A);
294        eval_lazy(i, l, r);
295        update(i, l, r, B, C);
296        eval_lazy(i, l, r);
297        update(i, l, r, E, F);
298        eval_lazy(i, l, r);
299        update(i, l, r, G, H);
300        eval_lazy(i, l, r);
301        update(i, l, r, I, J);
302        eval_lazy(i, l, r);
303        update(i, l, r, K, L);
304        eval_lazy(i, l, r);
305        update(i, l, r, M, N);
306        eval_lazy(i, l, r);
307        update(i, l, r, O);
308        eval_lazy(i, l, r);
309        return tree[i];
310    }
311
312    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P) {
313        eval_lazy(i, l, r);
314        update(i, l, r, d, a);
315        eval_lazy(i, l, r);
316        update(i, l, r, D, A);
317        eval_lazy(i, l, r);
318        update(i, l, r, B, C);
319        eval_lazy(i, l, r);
320        update(i, l, r, E, F);
321        eval_lazy(i, l, r);
322        update(i, l, r, G, H);
323        eval_lazy(i, l, r);
324        update(i, l, r, I, J);
325        eval_lazy(i, l, r);
326        update(i, l, r, K, L);
327        eval_lazy(i, l, r);
328        update(i, l, r, M, N);
329        eval_lazy(i, l, r);
330        update(i, l, r, O, P);
331        eval_lazy(i, l, r);
332        return tree[i];
333    }
334
335    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q) {
336        eval_lazy(i, l, r);
337        update(i, l, r, d, a);
338        eval_lazy(i, l, r);
339        update(i, l, r, D, A);
340        eval_lazy(i, l, r);
341        update(i, l, r, B, C);
342        eval_lazy(i, l, r);
343        update(i, l, r, E, F);
344        eval_lazy(i, l, r);
345        update(i, l, r, G, H);
346        eval_lazy(i, l, r);
347        update(i, l, r, I, J);
348        eval_lazy(i, l, r);
349        update(i, l, r, K, L);
350        eval_lazy(i, l, r);
351        update(i, l, r, M, N);
352        eval_lazy(i, l, r);
353        update(i, l, r, O, P);
354        eval_lazy(i, l, r);
355        update(i, l, r, Q);
356        eval_lazy(i, l, r);
357        return tree[i];
358    }
359
360    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R) {
361        eval_lazy(i, l, r);
362        update(i, l, r, d, a);
363        eval_lazy(i, l, r);
364        update(i, l, r, D, A);
365        eval_lazy(i, l, r);
366        update(i, l, r, B, C);
367        eval_lazy(i, l, r);
368        update(i, l, r, E, F);
369        eval_lazy(i, l, r);
370        update(i, l, r, G, H);
371        eval_lazy(i, l, r);
372        update(i, l, r, I, J);
373        eval_lazy(i, l, r);
374        update(i, l, r, K, L);
375        eval_lazy(i, l, r);
376        update(i, l, r, M, N);
377        eval_lazy(i, l, r);
378        update(i, l, r, O, P);
379        eval_lazy(i, l, r);
380        update(i, l, r, Q, R);
381        eval_lazy(i, l, r);
382        return tree[i];
383    }
384
385    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S) {
386        eval_lazy(i, l, r);
387        update(i, l, r, d, a);
388        eval_lazy(i, l, r);
389        update(i, l, r, D, A);
390        eval_lazy(i, l, r);
391        update(i, l, r, B, C);
392        eval_lazy(i, l, r);
393        update(i, l, r, E, F);
394        eval_lazy(i, l, r);
395        update(i, l, r, G, H);
396        eval_lazy(i, l, r);
397        update(i, l, r, I, J);
398        eval_lazy(i, l, r);
399        update(i, l, r, K, L);
400        eval_lazy(i, l, r);
401        update(i, l, r, M, N);
402        eval_lazy(i, l, r);
403        update(i, l, r, O, P);
404        eval_lazy(i, l, r);
405        update(i, l, r, Q, R);
406        eval_lazy(i, l, r);
407        update(i, l, r, S);
408        eval_lazy(i, l, r);
409        return tree[i];
410    }
411
412    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T) {
413        eval_lazy(i, l, r);
414        update(i, l, r, d, a);
415        eval_lazy(i, l, r);
416        update(i, l, r, D, A);
417        eval_lazy(i, l, r);
418        update(i, l, r, B, C);
419        eval_lazy(i, l, r);
420        update(i, l, r, E, F);
421        eval_lazy(i, l, r);
422        update(i, l, r, G, H);
423        eval_lazy(i, l, r);
424        update(i, l, r, I, J);
425        eval_lazy(i, l, r);
426        update(i, l, r, K, L);
427        eval_lazy(i, l, r);
428        update(i, l, r, M, N);
429        eval_lazy(i, l, r);
430        update(i, l, r, O, P);
431        eval_lazy(i, l, r);
432        update(i, l, r, Q, R);
433        eval_lazy(i, l, r);
434        update(i, l, r, S, T);
435        eval_lazy(i, l, r);
436        return tree[i];
437    }
438
439    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U) {
440        eval_lazy(i, l, r);
441        update(i, l, r, d, a);
442        eval_lazy(i, l, r);
443        update(i, l, r, D, A);
444        eval_lazy(i, l, r);
445        update(i, l, r, B, C);
446        eval_lazy(i, l, r);
447        update(i, l, r, E, F);
448        eval_lazy(i, l, r);
449        update(i, l, r, G, H);
450        eval_lazy(i, l, r);
451        update(i, l, r, I, J);
452        eval_lazy(i, l, r);
453        update(i, l, r, K, L);
454        eval_lazy(i, l, r);
455        update(i, l, r, M, N);
456        eval_lazy(i, l, r);
457        update(i, l, r, O, P);
458        eval_lazy(i, l, r);
459        update(i, l, r, Q, R);
460        eval_lazy(i, l, r);
461        update(i, l, r, S, T);
462        eval_lazy(i, l, r);
463        update(i, l, r, U);
464        eval_lazy(i, l, r);
465        return tree[i];
466    }
467
468    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V) {
469        eval_lazy(i, l, r);
470        update(i, l, r, d, a);
471        eval_lazy(i, l, r);
472        update(i, l, r, D, A);
473        eval_lazy(i, l, r);
474        update(i, l, r, B, C);
475        eval_lazy(i, l, r);
476        update(i, l, r, E, F);
477        eval_lazy(i, l, r);
478        update(i, l, r, G, H);
479        eval_lazy(i, l, r);
480        update(i, l, r, I, J);
481        eval_lazy(i, l, r);
482        update(i, l, r, K, L);
483        eval_lazy(i, l, r);
484        update(i, l, r, M, N);
485        eval_lazy(i, l, r);
486        update(i, l, r, O, P);
487        eval_lazy(i, l, r);
488        update(i, l, r, Q, R);
489        eval_lazy(i, l, r);
490        update(i, l, r, S, T);
491        eval_lazy(i, l, r);
492        update(i, l, r, U, V);
493        eval_lazy(i, l, r);
494        return tree[i];
495    }
496
497    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W) {
498        eval_lazy(i, l, r);
499        update(i, l, r, d, a);
500        eval_lazy(i, l, r);
501        update(i, l, r, D, A);
502        eval_lazy(i, l, r);
503        update(i, l, r, B, C);
504        eval_lazy(i, l, r);
505        update(i, l, r, E, F);
506        eval_lazy(i, l, r);
507        update(i, l, r, G, H);
508        eval_lazy(i, l, r);
509        update(i, l, r, I, J);
510        eval_lazy(i, l, r);
511        update(i, l, r, K, L);
512        eval_lazy(i, l, r);
513        update(i, l, r, M, N);
514        eval_lazy(i, l, r);
515        update(i, l, r, O, P);
516        eval_lazy(i, l, r);
517        update(i, l, r, Q, R);
518        eval_lazy(i, l, r);
519        update(i, l, r, S, T);
520        eval_lazy(i, l, r);
521        update(i, l, r, U, V);
522        eval_lazy(i, l, r);
523        update(i, l, r, W);
524        eval_lazy(i, l, r);
525        return tree[i];
526    }
527
528    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X) {
529        eval_lazy(i, l, r);
530        update(i, l, r, d, a);
531        eval_lazy(i, l, r);
532        update(i, l, r, D, A);
533        eval_lazy(i, l, r);
534        update(i, l, r, B, C);
535        eval_lazy(i, l, r);
536        update(i, l, r, E, F);
537        eval_lazy(i, l, r);
538        update(i, l, r, G, H);
539        eval_lazy(i, l, r);
540        update(i, l, r, I, J);
541        eval_lazy(i, l, r);
542        update(i, l, r, K, L);
543        eval_lazy(i, l, r);
544        update(i, l, r, M, N);
545        eval_lazy(i, l, r);
546        update(i, l, r, O, P);
547        eval_lazy(i, l, r);
548        update(i, l, r, Q, R);
549        eval_lazy(i, l, r);
550        update(i, l, r, S, T);
551        eval_lazy(i, l, r);
552        update(i, l, r, U, V);
553        eval_lazy(i, l, r);
554        update(i, l, r, W, X);
555        eval_lazy(i, l, r);
556        return tree[i];
557    }
558
559    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X, ll Y) {
560        eval_lazy(i, l, r);
561        update(i, l, r, d, a);
562        eval_lazy(i, l, r);
563        update(i, l, r, D, A);
564        eval_lazy(i, l, r);
565        update(i, l, r, B, C);
566        eval_lazy(i, l, r);
567        update(i, l, r, E, F);
568        eval_lazy(i, l, r);
569        update(i, l, r, G, H);
570        eval_lazy(i, l, r);
571        update(i, l, r, I, J);
572        eval_lazy(i, l, r);
573        update(i, l, r, K, L);
574        eval_lazy(i, l, r);
575        update(i, l, r, M, N);
576        eval_lazy(i, l, r);
577        update(i, l, r, O, P);
578        eval_lazy(i, l, r);
579        update(i, l, r, Q, R);
580        eval_lazy(i, l, r);
581        update(i, l, r, S, T);
582        eval_lazy(i, l, r);
583        update(i, l, r, U, V);
584        eval_lazy(i, l, r);
585        update(i, l, r, W, X);
586        eval_lazy(i, l, r);
587        update(i, l, r, Y);
588        eval_lazy(i, l, r);
589        return tree[i];
590    }
591
592    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X, ll Y, ll Z) {
593        eval_lazy(i, l, r);
594        update(i, l, r, d, a);
595        eval_lazy(i, l, r);
596        update(i, l, r, D, A);
597        eval_lazy(i, l, r);
598        update(i, l, r, B, C);
599        eval_lazy(i, l, r);
600        update(i, l, r, E, F);
601        eval_lazy(i, l, r);
602        update(i, l, r, G, H);
603        eval_lazy(i, l, r);
604        update(i, l, r, I, J);
605        eval_lazy(i, l, r);
606        update(i, l, r, K, L);
607        eval_lazy(i, l, r);
608        update(i, l, r, M, N);
609        eval_lazy(i, l, r);
610        update(i, l, r, O, P);
611        eval_lazy(i, l, r);
612        update(i, l, r, Q, R);
613        eval_lazy(i, l, r);
614        update(i, l, r, S, T);
615        eval_lazy(i, l, r);
616        update(i, l, r, U, V);
617        eval_lazy(i, l, r);
618        update(i, l, r, W, X);
619        eval_lazy(i, l, r);
620        update(i, l, r, Y, Z);
621        eval_lazy(i, l, r);
622        return tree[i];
623    }
624
625    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X, ll Y, ll Z, ll AA) {
626        eval_lazy(i, l, r);
627        update(i, l, r, d, a);
628        eval_lazy(i, l, r);
629        update(i, l, r, D, A);
630        eval_lazy(i, l, r);
631        update(i, l, r, B, C);
632        eval_lazy(i, l, r);
633        update(i, l, r, E, F);
634        eval_lazy(i, l, r);
635        update(i, l, r, G, H);
636        eval_lazy(i, l, r);
637        update(i, l, r, I, J);
638        eval_lazy(i, l, r);
639        update(i, l, r, K, L);
640        eval_lazy(i, l, r);
641        update(i, l, r, M, N);
642        eval_lazy(i, l, r);
643        update(i, l, r, O, P);
644        eval_lazy(i, l, r);
645        update(i, l, r, Q, R);
646        eval_lazy(i, l, r);
647        update(i, l, r, S, T);
648        eval_lazy(i, l, r);
649        update(i, l, r, U, V);
650        eval_lazy(i, l, r);
651        update(i, l, r, W, X);
652        eval_lazy(i, l, r);
653        update(i, l, r, Y, Z);
654        eval_lazy(i, l, r);
655        update(i, l, r, AA);
656        eval_lazy(i, l, r);
657        return tree[i];
658    }
659
660    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X, ll Y, ll Z, ll AA, ll BB) {
661        eval_lazy(i, l, r);
662        update(i, l, r, d, a);
663        eval_lazy(i, l, r);
664        update(i, l, r, D, A);
665        eval_lazy(i, l, r);
666        update(i, l, r, B, C);
667        eval_lazy(i, l, r);
668        update(i, l, r, E, F);
669        eval_lazy(i, l, r);
670        update(i, l, r, G, H);
671        eval_lazy(i, l, r);
672        update(i, l, r, I, J);
673        eval_lazy(i, l, r);
674        update(i, l, r, K, L);
675        eval_lazy(i, l, r);
676        update(i, l, r, M, N);
677        eval_lazy(i, l, r);
678        update(i, l, r, O, P);
679        eval_lazy(i, l, r);
680        update(i, l, r, Q, R);
681        eval_lazy(i, l, r);
682        update(i, l, r, S, T);
683        eval_lazy(i, l, r);
684        update(i, l, r, U, V);
685        eval_lazy(i, l, r);
686        update(i, l, r, W, X);
687        eval_lazy(i, l, r);
688        update(i, l, r, Y, Z);
689        eval_lazy(i, l, r);
690        update(i, l, r, AA, BB);
691        eval_lazy(i, l, r);
692        return tree[i];
693    }
694
695    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X, ll Y, ll Z, ll AA, ll BB, ll CC) {
696        eval_lazy(i, l, r);
697        update(i, l, r, d, a);
698        eval_lazy(i, l, r);
699        update(i, l, r, D, A);
700        eval_lazy(i, l, r);
701        update(i, l, r, B, C);
702        eval_lazy(i, l, r);
703        update(i, l, r, E, F);
704        eval_lazy(i, l, r);
705        update(i, l, r, G, H);
706        eval_lazy(i, l, r);
707        update(i, l, r, I, J);
708        eval_lazy(i, l, r);
709        update(i, l, r, K, L);
710        eval_lazy(i, l, r);
711        update(i, l, r, M, N);
712        eval_lazy(i, l, r);
713        update(i, l, r, O, P);
714        eval_lazy(i, l, r);
715        update(i, l, r, Q, R);
716        eval_lazy(i, l, r);
717        update(i, l, r, S, T);
718        eval_lazy(i, l, r);
719        update(i, l, r, U, V);
720        eval_lazy(i, l, r);
721        update(i, l, r, W, X);
722        eval_lazy(i, l, r);
723        update(i, l, r, Y, Z);
724        eval_lazy(i, l, r);
725        update(i, l, r, AA, BB, CC);
726        eval_lazy(i, l, r);
727        return tree[i];
728    }
729
730    ll update(int i, int l, int r, ll d, ll a, ll D, ll A, ll B, ll C, ll E, ll F, ll G, ll H, ll I, ll J, ll K, ll L, ll M, ll N, ll O, ll P, ll Q, ll R, ll S, ll T, ll U, ll V, ll W, ll X, ll Y, ll Z, ll AA, ll BB, ll CC, ll DD) {
731        eval_lazy(i, l, r);
732        update(i, l, r, d, a);
733        eval_lazy(i, l, r);
734        update(i, l, r, D, A);
735        eval_lazy(i, l, r);
736        update(i, l, r, B, C);
737        eval_lazy(i, l, r);
738        update(i, l, r, E, F);
739        eval_lazy(i, l
```

```

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(){
99     ios_base::sync_with_stdio(false); cin.tie(NULL);
100    int n, q; cin >> n >> q;
101    segtree<sum_t> st;
102    st.init(n);
103

```

```

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 }


```

3.22 Segment Tree

```

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

```

```

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 };

```

3.23 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)
*/

```

```

13 const int MAXN = 505;
14 int n, m;           // grid dimensions
15 int a[MAXN][MAXN]; // input grid
16 int t[4 * MAXN][4 * MAXN]; // segment tree
17
18 // Build the tree along y-axis (internal to each x-interval)
19 void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
20     if (ly == ry) {
21         if (lx == rx)
22             t[vx][vy] = a[lx][ly];
23         else
24             t[vx][vy] = t[vx * 2][vy] + t[vx * 2 + 1][vy];
25     } else {
26         int my = (ly + ry) / 2;
27         build_y(vx, lx, rx, vy * 2, ly, my);
28         build_y(vx, lx, rx, vy * 2 + 1, my + 1, ry);
29         t[vx][vy] = t[vx][vy * 2] + t[vx][vy * 2 + 1];
30     }
31 }
32 // 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
64   y, int new_val) {
65     if (ly == ry) {
66         if (lx == rx)
67             t[vx][vy] = new_val;
68         else
69             t[vx][vy] = t[vx * 2][vy] + t[vx * 2 + 1][vy];
70     } else {
71         int my = (ly + ry) / 2;
72         if (y <= my)
73             update_y(vx, lx, rx, vy * 2, ly, my, x, y, new_val);
74         else
75             update_y(vx, lx, rx, vy * 2 + 1, my + 1, ry, x, y, new_val);
76     }
77     t[vx][vy] = t[vx][vy * 2] + t[vx][vy * 2 + 1];
78 }

```

```

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 }
```

3.24 Segment Tree Dynamic

```

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

Space Complexity: O(nodes visited or created) -> worst O(log U) per
operation
*/
struct Vertex {
    int left, right;          // interval [left, right)
    int sum = 0;               // sum of elements in this interval
    Vertex *left_child = nullptr, *right_child = nullptr;

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

    // Create children lazily only when needed
    void extend() {
        if (!left_child && left + 1 < right) {
```

```

26     int mid = (left + right) / 2;
27     left_child = new Vertex(left, mid);
28     right_child = new Vertex(mid, right);
29 }
30
31 // Add 'x' to position 'k'
32 void add(int k, int x) {
33     extend();
34     sum += x;
35     if (left_child) {
36         if (k < left_child->right)
37             left_child->add(k, x);
38         else
39             right_child->add(k, x);
40     }
41 }
42
43 // Get sum over interval [lq, rq)
44 int get_sum(int lq, int rq) {
45     if (lq <= left && right <= rq)
46         return sum;
47     if (rq <= left || right <= lq)
48         return 0;
49     extend();
50     return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq);
51 }
52 };
53
54 Vertex *root = new Vertex(0, 1e9); // Range [0, 1e9)
55 root->add(5, 10);           // a[5] += 10
56 root->add(1000, 20);        // a[1000] += 20
57 cout << root->get_sum(0, 10) << '\n';      // sum of [0, 10) = 10
58 cout << root->get_sum(0, 2000) << '\n';    // sum of [0, 2000) = 30
```

3.25 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 };

```

3.26 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 };

```

3.27 Segment Tree Lazy Range Set

```

1 /*
2  Lazy Segment Tree (Range Set + Range Add + Range Sum)
3  -----
4  Indexing: 0-based
5  Bounds: [0, N-1]
6
7  Features:
8      - Supports range set (assign value), range add (increment), and
9          range sum queries
10     - Properly prioritizes lazy set > lazy add
11 */
12 const int maxN = 1e5 + 5;
13 int N, Q;
14 int a[maxN];
15
16 struct node {
17     ll val = 0;           // range sum
18     ll lzAdd = 0;         // pending addition
19     ll lzSet = 0;         // pending set (non-zero means active)
20 };
21
22 node tree[maxN << 2];
23
24 #define lc (p << 1)
25 #define rc ((p << 1) | 1)
26
27 // Update current node based on its children
28 inline void pushup(int p) {
29     tree[p].val = tree[lc].val + tree[rc].val;
30 }
31
32 // Push lazy values down to children
33 void pushdown(int p, int l, int mid, int r) {
34     // Range set overrides any pending add
35     if (tree[p].lzSet != 0) {
36         tree[lc].lzSet = tree[rc].lzSet = tree[p].lzSet;
37         tree[lc].val = (mid - l + 1) * tree[p].lzSet;
38         tree[rc].val = (r - mid) * tree[p].lzSet;
39         tree[lc].lzAdd = tree[rc].lzAdd = 0;
40         tree[p].lzSet = 0;

```

```

41 }
42 // Otherwise propagate add
43 else if (tree[p].lzAdd != 0) {
44     if (tree[lc].lzSet == 0) tree[lc].lzAdd += tree[p].lzAdd;
45     else {
46         tree[lc].lzSet += tree[p].lzAdd;
47         tree[lc].lzAdd = 0;
48     }
49     if (tree[rc].lzSet == 0) tree[rc].lzAdd += tree[p].lzAdd;
50     else {
51         tree[rc].lzSet += tree[p].lzAdd;
52         tree[rc].lzAdd = 0;
53     }
54     tree[lc].val += (mid - l + 1) * tree[p].lzAdd;
55     tree[rc].val += (r - mid) * tree[p].lzAdd;
56     tree[p].lzAdd = 0;
57 }
58 }

59 // Build tree from array a[0..N-1]
60 void build(int p, int l, int r) {
61     tree[p].lzAdd = tree[p].lzSet = 0;
62     if (l == r) {
63         tree[p].val = a[l];
64         return;
65     }
66     int mid = (l + r) >> 1;
67     build(lc, l, mid);
68     build(rc, mid + 1, r);
69     pushup(p);
70 }
71

72 // Add 'val' to all elements in [a, b]
73 void add(int p, int l, int r, int a, int b, ll val) {
74     if (a > r || b < l) return;
75     if (a <= l && r <= b) {
76         tree[p].val += (r - l + 1) * val;
77         if (tree[p].lzSet == 0) tree[p].lzAdd += val;
78         else tree[p].lzSet += val;
79         return;
80     }
81     int mid = (l + r) >> 1;
82     pushdown(p, l, mid, r);
83 }

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
114 // Example usage
115 N = 5;
116 a[0] = 2, a[1] = 4, a[2] = 3, a[3] = 1, a[4] = 5;
117 build(1, 0, N - 1);
118 set(1, 0, N - 1, 1, 3, 7);      // a[1..3] = 7
119 add(1, 0, N - 1, 2, 4, 2);      // a[2..4] += 2
120 cout << query(1, 0, N - 1, 0, 4) << '\n'; // total sum

```

3.28 Segment Tree Lazy Range Set 2

```

1 #include <bits/stdc++.h>
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         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             if(lazy[2 * i + 2].val == -1) lazy1[2 * i + 2].val += lazy1[i].
132                 val;
133             else{
134                 lazy[2 * i + 2].val += lazy1[i].val + lazy1[2 * i + 2].val;
135                 lazy1[2 * i + 2].val = 0;
136             }
137             lazy1[i] = num_t(0);
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 }
```

3.29 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         return node_l+node_r;
42     }
43 };

```

3.30 Segment Tree Range Update

```

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    };

```

3.31 Segment Tree Struct Types

```

1 // Sum segment tree
2 struct sum_t{
3     ll val;
4     static const long long null_v = 0;

```

```

1 /*
2  Segment Tree (Range Min Update, Point Query)
3  -----
4  Indexing: 0-based

```

```

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

7   sum_t operator + (const sum_t &a) const {
8     sum_t ans;
9     ans.val = val + a.val;
10    return ans;
11  }
12}
13};

// Min segment tree
14 struct min_t{
15   ll val;
16   static const long long null_v = 1e18;

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

19   min_t operator + (const min_t &a) const {
20     min_t ans;
21     ans.val = min(val, a.val);
22     return ans;
23   }
24};
25};

// Max segment tree
26 struct max_t{
27   ll val;
28   static const long long null_v = -1e18;

29   max_t(): val(null_v) {}
30   max_t(long long v): val(v) {}

31   max_t operator + (const max_t &a) const {
32     max_t ans;
33     ans.val = max(val, a.val);
34     return ans;
35   }
36};
37};

// GCD segment tree
38 struct gcd_t{
39   ll val;
40   static const long long null_v = 0;
41 };
42};


```

```

48   gcd_t(): val(null_v) {}
49   gcd_t(long long v): val(v) {}

50   gcd_t operator + (const gcd_t &a) const {
51     gcd_t ans;
52     ans.val = gcd(val, a.val);
53     return ans;
54   }
55};


```

3.32 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 };

```

3.33 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)
*/

```

```

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

```

```

22
23 SegmentTreeWalk(int _n) : a(4 * _n, 1e18), final_pos(_n), n(_n) {}
24
25 // Build segment tree from array 'vals[0..n-1]', start with node=1, l
26 // =0, r=n-1
27 void build(int l, int r, int node, const vector<ll> &vals) {
28     if (l == r) {
29         final_pos[l] = node;
30         a[node] = vals[l];
31     } else {
32         int mid = (l + r) / 2;
33         build(l, mid, node * 2, vals);
34         build(mid + 1, r, node * 2 + 1, vals);
35         a[node] = min(a[node * 2], a[node * 2 + 1]);
36     }
37
38 // 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) {
60     if (l > R || r < L) return {-1, 0}; // out of query
61     if (a[node] < val) return {-1, 0}; // all values < val

```

```

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

```

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

22 // Returns min in range [L, R]
23 int mini(int L, int R) {
24     int len = R - L + 1;

```

```

27     int i = lg[len];
28     return min(st[i][L], st[i][R - (1 << i) + 1]);
29 }

30 int main() {
31     cin >> n;
32     a.resize(n);
33     for (int i = 0; i < n; i++) cin >> a[i];
34     // Precompute binary logs
35     lg[1] = 0;
36     for (int i = 2; i <= n; i++) {
37         lg[i] = lg[i / 2] + 1;
38     }
39     // Initialize 2^0 intervals
40     for (int i = 0; i < n; i++) {
41         st[0][i] = a[i];
42     }
43     // Build sparse table
44     for (int k = 1; k <= K; k++) {
45         for (int i = 0; i + (1 << k) <= n; i++) {
46             st[k][i] = min(st[k - 1][i], st[k - 1][i + (1 << (k - 1))]);
47         }
48     }
49     // Example usage
50     int q; cin >> q;
51     while (q--) {
52         int l, r;
53         cin >> l >> r;
54         cout << mini(l, r) << '\n';
55     }
56     return 0;
57 }

```

3.35 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);

```

```

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     else r = mid - 1;
92 }
93 cout << ans << '\n';
94 }
95 return 0;
96 }
97 // https://www.codechef.com/problems/CENS20B

```

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

56         if (containsA && containsB) {
57             newBlocks[curBlock++] = {B.l, cutA};
58             newBlocks[curBlock++] = {cutA, cutB};
59             newBlocks[curBlock++] = {cutB, B.r};
60         } else if (containsA) {
61             newBlocks[curBlock++] = {B.l, cutA};
62             newBlocks[curBlock++] = {cutA, B.r};
63         } else if (containsB) {
64             newBlocks[curBlock++] = {B.l, cutB};
65             newBlocks[curBlock++] = {cutB, B.r};
66         } else {
67             newBlocks[curBlock++] = B;
68         }
69     }
70     pos += B.sz();
71 }
72
73 // Pass 2: Reorder - move [a, b) to the end
74
75 // Rebuilds the entire decomposition into 1 block (or balanced ones)
76 void rebuildDecomp() {

```

```

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 }

```

3.37 Treap

1 std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().
2 count());

```

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 // });

```

3.38 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\u2193", t->value);
64     output (t->r);
65 }
```

3.39 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 template<class F> void each(Node* n, F f) {
24     if (n) { n->push(); each(n->l, f); f(n->val); each(n->r, f); }
25 }
```

```

26
27 pair<Node*, Node*> split(Node* n, int k) {
28     if (!n) return {};
29     n->push();
30     if (cnt(n->l) >= k) {
31         auto pa = split(n->l, k);
32         n->l = pa.second;
33         n->recalc();
34         return {pa.first, n};
35     } else {
36         auto pa = split(n->r, k - cnt(n->l) - 1);
37         n->r = pa.first;
38         n->recalc();
39         return {n, pa.second};
40     }
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 - 1);
77     if (k <= l) t = merge(ins(a, b, k), c);
78     else t = merge(a, ins(c, b, k - r));
79 }

```

3.40 Treap With Inversions and Range Updates

```

1 #include <bits/stdc++.h>
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 < 0) v += MOD;}
12     ModInt& operator += (const ModInt &other) {v += other.v; if (v >= MOD) v -= MOD; return *this;}
13     ModInt& operator -= (const ModInt &other) {v -= other.v; if (v < 0) v += MOD; return *this;}
14     ModInt& operator *= (const ModInt &other) {v = v * other.v % MOD;
15         return *this;}
16     ModInt& operator /= (const ModInt &other) {return *this *= inverse(
17         other);}
18     bool operator == (const ModInt &other) const {return v == other.v;}
19     bool operator != (const ModInt &other) const {return v != other.v;}
20     friend ModInt operator + (ModInt a, const ModInt &b) {return a += b;}
21     friend ModInt operator - (ModInt a, const ModInt &b) {return a -= b;}
22     friend ModInt operator * (ModInt a, const ModInt &b) {return a *= b;}
23     friend ModInt operator / (ModInt a, const ModInt &b) {return a /= b;}
24     friend ModInt operator - (const ModInt &a) {return 0 - a;}
25     friend ModInt power(ModInt a, ll b) {ModInt ret(1); while (b > 0) {if
26         (b & 1) ret *= a; a *= a; b >>= 1;} return ret;}

```

```

24     friend ModInt inverse(ModInt a) {return power(a, MOD - 2);}
25     friend istream& operator >> (istream &is, ModInt &m) {is >> m.v; m.v =
26         (-MOD < m.v && m.v < MOD) ? m.v : m.v % MOD; if (m.v < 0) m.v +=
27         MOD; return is;}
28     friend ostream& operator << (ostream &os, const ModInt &m) {return os
29         << m.v;}
30
31     struct Line{
32         ModInt<mod> b = 1, c = 0;
33         ModInt<mod> operator()(ModInt<mod> x){ return b * x + c;}
34         Line operator()(Line other) {return Line{b * other.b, other.c * b + c
35             };}
36         operator bool() const{
37             return b != 1 || c != 0;
38         }
39     };
40
41     std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().
42         count());
43
44     struct Node {
45         Node *l = 0, *r = 0;
46         int y, c = 1;
47         ModInt<mod> val, sum;
48         Line line;
49         bool rev = 0;
50         Node(int val) : val(val), sum(val), y(rng()) {}
51         void recalc();
52         void push();
53     };
54
55     int cnt(Node* n) { return n ? n->c : 0; }
56     ModInt<mod> sum1(Node* n) { return n ? n->sum : ModInt<mod>(0); }
57     void Node::recalc() { if(l) l->push(); if(r) r->push(); c = cnt(l) +
58         cnt(r) + 1; sum = sum1(l) + sum1(r) + val; }
59     void Node::push(){
60         if(rev){
61             rev = 0;
62             swap(l, r);
63             if(l) l->rev ^= 1;
64             if(r) r->rev ^= 1;
65         }

```

```

61 if(line){
62     val = line(val);
63     sum = sum * line.b + line.c * c;
64     if(l){
65         l -> line = line(l -> line);
66     }
67     if(r){
68         r -> line = line(r -> line);
69     }
70     line = {1, 0};
71 }
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     }
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
120 void revv(Node*& t, int l, int r){
121     Node *a, *b, *c;
122     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
123     b -> rev ^= 1;
124     t = merge(merge(a, b), c);
125 }
126
127 ModInt<mod> query(Node*& t, int l, int r){
128     Node *a, *b, *c;
129     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
130     ModInt<mod> res = sum1(b);
131     t = merge(merge(a, b), c);
132     return res;
133 }
134
135 void remove(Node*& t, int x){
136     Node *a, *b, *c;
137     tie(a, b) = split(t, x); tie(b, c) = split(b, 1);
138     t = merge(a, c);
139 }
140
141 void update(Node*& t, int l, int r, ll X, ll Y){
142     Node *a, *b, *c;
143     tie(a, b) = split(t, l); tie(b, c) = split(b, r - 1);
144     b -> line = Line{X, Y}(b -> line);
145     t = merge(a, merge(b, c));
146 }

```

```

147
148 int main(){
149     ios_base::sync_with_stdio(false); cin.tie(NULL);
150     int n, q; cin >> n >> q;
151     vector<int> a(n);
152     for(int i = 0; i < n; i++){
153         cin >> a[i];
154     }
155     Node* treap = nullptr;
156     for(int i = 0; i < n; i++){
157         treap = ins(treap, new Node(a[i]), i);
158     }
159     while(q--){
160         int t; cin >> t;
161         if(t == 0){
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

```

3 -----
4 Indexing: 1-based for 'a', 'dp', 's'
5 Bounds:
6     - i from 1 to m // number of elements in the array
7     - j from 1 to p // number of transitions
8 Time Complexity: O(m * p)
9 Requires:
10    - Lines inserted in increasing slope order for min query
11    - Queries made with increasing x values
12
13 dp[i][j] = min over k < i of { dp[k][j-1] + a[i] * (i - k) + s[i] - s[
14     k+1] }
14 dp[i][j] = min over k < i of { dp[k][j-1] + cost(k, i) }
15
16 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 }

```

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.

*/

```

```

15 vector<int> num;
16 ll DP[20][20][2][2]; // pos, last digit, f (tight), z (leading zero)
17
18 ll g(int pos, int last, int f, int z) {
19     if (pos == num.size()) return 1; // reached end, valid number
20     if (DP[pos][last][f][z] != -1) return DP[pos][last][f][z];
21     ll res = 0;
22     int limit = f ? 9 : num[pos]; // upper digit bound based on tight flag

```

```

23     for (int dgt = 0; dgt <= limit; dgt++) {
24         // Skip if digit equals last (unless it's a leading zero)
25         if (dgt == last && !(dgt == 0 && z == 1)) continue;
26         int nf = f;
27         if (!f && dgt < limit) nf = 1;
28         if (z && dgt == 0) res += g(pos + 1, dgt, nf, 1); // still leading
29             zeros
30         else res += g(pos + 1, dgt, nf, 0); // now in significant digits
31     }
32     return DP[pos][last][f][z] = res;
33 }
34
35 ll solve(ll x) {
36     if (x == -1) return 0;
37     num.clear();
38     while (x > 0) {
39         num.push_back(x % 10);
40         x /= 10;
41     }
42     reverse(num.begin(), num.end());
43     memset(DP, -1, sizeof(DP));
44     return g(0, 0, 0, 1);
45 }

```

4.3 Divide and Conquer DP

```

/*
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
*/

```

```

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])
*/

```

```

9     - delete: dp[i-1][j] + 1
10    - replace/match: dp[i-1][j-1] + (s[i-1] != t[j-1])
11 */
12 const int MAXN = 5005;
13 int dp[MAXN][MAXN];
14
15 string s, t; cin >> s >> t;
16 int n = s.length(), m = t.length();
17 // Initialize all to a large number
18 for (int i = 0; i <= n; i++) {
19     fill(dp[i], dp[i] + m + 1, 1e9);
20 }
21 dp[0][0] = 0;
22 for (int i = 0; i <= n; i++) {
23     for (int j = 0; j <= m; j++) {
24         if (j) { // insert
25             dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
26         }
27         if (i) { // delete
28             dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
29         }
30         if (i && j) { // replace or match
31             int cost = (s[i - 1] != t[j - 1]) ? 1 : 0;
32             dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + cost);
33         }
34     }
35 }
```

4.5 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)
```

```

13 because with fixed s (sub array length) we have m_right(l) = mid[l+1][r]
   = m_left(l+1).
14 That's why nested l and m loops require not more than 2k iterations
   overall.
15 */
16
17 int n, k;
18 int a[N], mid[N][N];
19 ll res[N][N];
20 ll solve() {
21     for (int s = 0; s <= k; s++) {           // s - length of the subarray
22         for (int l = 0; l + s <= k; l++) {    // l - left point
23             int r = l + s;                      // r - right point
24             if (s < 2) {
25                 res[l][r] = 0;                  // base case- nothing to break
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
               the bounds only
33                 ll tmp = res[l][m] + res[m][r] + (a[r] - a[l]);
34                 if (res[l][r] > tmp) {            // relax current solution
35                     res[l][r] = tmp;
36                     mid[l][r] = m;
37                 }
38             }
39         }
40     }
41     ll ans = res[0][k];
42     return ans;
43 }
44 int main() {
45     int i, j, m;
46     while(cin >> n >> k) {
47         for(i = 1; i <= k; i++) cin >> a[i];
48         a[0] = 0;
49         a[k + 1] = n;
50         k++;
51         cout << solve() << endl;
52     }

```

```

53     return 0;
54 }
55 // 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 }
13 int i=n, j=m;
14 string ans="";
15 while(i && j){
16     if(s[i-1]==t[j-1]){
17         ans+=s[i-1];
18         i--; j--;
19     }
20     else if(dp[i][j-1]>=dp[i-1][j]){
21         j--;
22     }
23     else{
24         i--;
25     }
26 }
27 reverse(all(ans));
28 cout << ans << endl;
29
30 // For two permutations one can create new array that will map each
   element from the first permutation to the second.
31 // For each element a[i] in the first permutatio, you find which j is a[
   i] == b[j].
32 // After creating this new array, run LIS (Longest Increasing
   subsequence).

```

4.7 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:
15    - Stores lines in slope-sorted order ( $k$ )
16    - Each line keeps its intersection point ' $p$ ' with the next line
17    - Binary search on ' $p$ ' to answer queries
18 */
19 struct Line {
20     mutable ll k, m, p;
21     bool operator<(const Line& o) const { return k < o.k; } // Sort by
22         slope
23     bool operator<(ll x) const { return p < x; }           // Query
24         comparator
25 };
26
27 struct LineContainer : multiset<Line, less<> {
28     // (for doubles, use inf = 1/.0, div(a,b) = a/b)
29     static const ll inf = LLONG_MAX;
30     ll div(ll a, ll b) { // Floored division
31         return a / b - ((a ^ b) < 0 && a % b);
32     }
33     // Update intersection point  $x->p$  with  $y$ 
34     bool isect(iterator x, iterator y) {
35         if (y == end()) return x->p = inf, false;
36         if (x->k == y->k)
37             x->p = (x->m > y->m ? inf : -inf); // higher line wins
38         else
39             x->p = div(y->m - x->m, x->k - y->k);
40         return x->p >= y->p;
41     }
42     // Add new line:  $y = k * x + m$ 
43     void add(ll k, ll m) {
44

```

```

45         auto z = insert({k, m, 0}), y = z++, x = y;
46         // Remove dominated lines after  $y$ 
47         while (isect(y, z)) z = erase(z);
48         // Remove dominated lines before  $y$ 
49         if (x != begin() && isect(--x, y))
50             isect(x, y = erase(y));
51         // Further cleanup to preserve order
52         while ((y = x) != begin() && (--x)->p >= y->p)
53             isect(x, erase(y));
54     }
55     // Query max  $y$  at given  $x$ 
56     ll query(ll x) {
57         assert(!empty());
58         auto l = *lower_bound(x);
59         return l.k * x + l.m;
60     }
61     // Example usage:
62     LineContainer cht;
63     cht.add(3, 5);      //  $y = 3x + 5$ 
64     cht.add(2, 7);      //  $y = 2x + 7$ 
65     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  in LIS
12  vector<int> last_pos(n + 1); // last_pos[len] = index in v[] ending LIS
13  of length len
14  for (int i = 0; i < n; i++) {
15      auto it = lower_bound(dp.begin(), dp.end(), v[i]);
16      int len = it - dp.begin();

```

```

17     dp_index.push_back(i); // Ignore if no recovery
18 } else {
19     *it = v[i];
20     dp_index[len] = i; // Ignore if no recovery
21 }
22 if (len > 0) parent[i] = dp_index[len - 1]; // Ignore if no recovery
23 }
24 // Reconstruct LIS
25 vector<int> lis;
26 int pos = dp_index.back();
27 while (pos != -1) {
28     lis.push_back(v[pos]);
29     pos = parent[pos];
30 }
31 reverse(lis.begin(), lis.end());

```

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

```

```

reached
// edges between vertices that can be reached and the ones that cant are
the min cut
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

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

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

    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

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

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

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

```

```

93 }
94
95
96 vector<pair<int, int>> minCut(){
97     vector<pair<int, int>> ans;
98     queue<int> q;
99     q.push(s);
100    vector<bool> vis(n, 0);
101    while((int)q.size()){
102        int cur = q.front();
103        q.pop();
104        vis[cur] = 1;
105        for(int id: adj[cur]){
106            if(edges[id].cap - edges[id].flow <= 0) continue;
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
120 bool getPath(int cur, vector<int> &path){
121     path.push_back(cur);
122     for(int &i = sz[cur]; i >= 0; i--){
123         int edge_index = adj[cur][i];
124         if(edge_index & 1) continue;
125         if(edges[edge_index].cap - edges[edge_index].flow == 0){
126             i--;
127             getPath(edges[edge_index].u, path);
128             return true;
129         }
130     }
131     return false;
132 }
133
134 vector<vector<int>> getPaths(){
135     vector<vector<int>> ans;

```

```

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

```

```

32     for (int son : g[node])
33     {
34         // If the match of son is matched
35         if (dist[match[son]] == INF)
36         {
37             dist[match[son]] = dist[node] + 1;
38             q.push(match[son]);
39         }
40     }
41 }
42 // Returns true if an alternating path has been found
43 return dist[0] != INF;
44 }

45 // Returns true if an augmenting path has been found starting from
46 // vertex node
47 bool bfs(int node)
48 {
49     if (node == 0)
50     {
51         return true;
52     }
53     for (int son : g[node])
54     {
55         if (dist[match[son]] == dist[node] + 1 && bfs(match[son]))
56         {
57             match[node] = son;
58             match[son] = node;
59             return true;
60         }
61     }
62     dist[node] = INF;
63     return false;
64 }

65 int hopcroft_karp()
66 {
67     int cnt = 0;
68     // While there is an alternating path
69     while (bfs())
70     {
71         for (int node = 1; node <= n; node++)
72         {

```

```

74         // If node is unmatched but we can match it using an augmenting
75         // path
76         if (!match[node] && dfs(node))
77         {
78             cnt++;
79         }
80     }
81     return cnt;
82 }

83 // usage
84 // n numero de puntos en la izquierda
85 // m numero de puntos en la derecha
86 // 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
44         int x=0, y=-1;
45         while (y== -1) {
46             while (!q.empty() && y== -1) {
47                 x = q.front(); q.pop();
48                 forn(j, n) if (mt[x][j] == lx[x] + ly[j] && !T[j]) {
49                     if (yx[j] == -1) { y = j; break; }
50                     T[j] = 1;
51                     bpone(yx[j], x);
52                 }
53             }
54             if (y!= -1) break;
55             int dlt = INFT0;
56             forn(j, n) if (!T[j]) dlt = min(dlt, slk[j]);
57             forn(k, n) {
58                 if (S[k]) lx[k] -= dlt;
59                 if (T[k]) ly[k] += dlt;
60                 if (!T[k]) slk[k] -= dlt;
61             }
62 //             q = queue<int>();
63
64         }
65     }
66 }
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82

```

```

63     forn(j, n) if (!T[j] && !slk[j]) {
64         if (yx[j] == -1) {
65             x = slkx[j]; y = j; break;
66         } else {
67             T[j] = 1;
68             if (!S[yx[j]]) bpone(yx[j], slkx[j]);
69         }
70     }
71     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         } else break;
77     }
78     int res = 0;
79     forn(i, n) res += mt[i][xy[i]];
80     return res;
81 }
82

```

5.4 Max Flow Min Cost

```

1 // dado un acomodo de flujos con costos
2 // devuelve el costo minimo para un flujo especificado
3
4 struct Edge
5 {
6     int from, to, capacity, cost;
7     Edge(int _from, int _to, int _capacity, int _cost)
8     {
9         from = _from;
10        to = _to;
11        capacity = _capacity;
12        cost = _cost;
13    }
14 };
15
16 vector<vector<int>> adj, cost, capacity;
17
18 const int INF = 1e9;
19
20 void shortest_paths(int n, int v0, vector<int> &d, vector<int> &p)

```

```

21  {
22      d.assign(n, INF);
23      d[v0] = 0;
24      vector<bool> inq(n, false);
25      queue<int> q;
26      q.push(v0);
27      p.assign(n, -1);
28
29      while (!q.empty())
30      {
31          int u = q.front();
32          q.pop();
33          inq[u] = false;
34          for (int v : adj[u])
35          {
36              if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v])
37              {
38                  d[v] = d[u] + cost[u][v];
39                  p[v] = u;
40                  if (!inq[v])
41                  {
42                      inq[v] = true;
43                      q.push(v);
44                  }
45              }
46          }
47      }
48  }

49
50  int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t)
51  {
52      adj.assign(N, vector<int>());
53      cost.assign(N, vector<int>(N, 0));
54      capacity.assign(N, vector<int>(N, 0));
55      for (Edge e : edges)
56      {
57          adj[e.from].push_back(e.to);
58          adj[e.to].push_back(e.from);
59          cost[e.from][e.to] = e.cost;
60          cost[e.to][e.from] = -e.cost;
61          capacity[e.from][e.to] = e.capacity;
62      }
63  }

```

```

64  int flow = 0;
65  int cost = 0;
66  vector<int> d, p;
67  while (flow < K)
68  {
69      shortest_paths(N, s, d, p);
70      if (d[t] == INF)
71          break;
72
73      // find max flow on that path
74      int f = K - flow;
75      int cur = t;
76      while (cur != s)
77      {
78          f = min(f, capacity[p[cur]][cur]);
79          cur = p[cur];
80      }
81
82      // apply flow
83      flow += f;
84      cost += f * d[t];
85      cur = t;
86      while (cur != s)
87      {
88          capacity[p[cur]][cur] -= f;
89          capacity[cur][p[cur]] += f;
90          cur = p[cur];
91      }
92
93      if (flow < K)
94          return -1;
95      else
96          return cost;
97  }
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  {
4      int n = adj.size();

```

```

5   vector<int> parent(n, -1);
6   // Find a way from the source to sink on a path with non-negative
7   // capacities
8   auto reachable = [&]() -> bool
9   {
10     queue<int> q;
11     q.push(source);
12     while (!q.empty())
13     {
14       int node = q.front();
15       q.pop();
16       for (int son : adj[node])
17       {
18         long long w = capacity[node][son];
19         if (w <= 0 || parent[son] != -1)
20           continue;
21         parent[son] = node;
22         q.push(son);
23       }
24     }
25     return parent[sink] != -1;
26   };
27
28   long long flow = 0;
29   // While there is a way from source to sink with non-negative
30   // capacities
31   while (reachable())
32   {
33     int node = sink;
34     // The minimum capacity on the path from source to sink
35     long long curr_flow = LLONG_MAX;
36     while (node != source)
37     {
38       curr_flow = min(curr_flow, capacity[parent[node]][node]);
39       node = parent[node];
40     }
41     node = sink;
42     while (node != source)
43     {
44       // Subtract the capacity from capacity edges
45       capacity[parent[node]][node] -= curr_flow;
46       // Add the current flow to flow backedges
47       capacity[node][parent[node]] += curr_flow;
48     }
49   }
50 }
```

```

46     node = parent[node];
47   }
48   flow += curr_flow;
49   fill(parent.begin(), parent.end(), -1);
50 }
51
52 return flow;
53 }

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 #define long long ll;
17 #define pair<int, int> pii;
18 #define vector<int> vi;
19
20 #pragma once
21 // #include <bits/extc++.h> // include-line, keep-include
22
23 const ll INF = numeric_limits<ll>::max() / 4;
```

```

24
25 struct MCMF {
26     struct edge {
27         int from, to, rev;
28         ll cap, cost, flow;
29     };
30     int N;
31     vector<vector<edge>> ed;
32     vi seen;
33     vector<ll> dist, pi;
34     vector<edge*> par;
35
36     MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}
37
38     void addEdge(int from, int to, ll cap, ll cost) {
39         if (from == to) return;
40         ed[from].push_back(edge{ from,to,sz(ed[to]),cap,cost,0 });
41         ed[to].push_back(edge{ to,from,sz(ed[from])-1,0,-cost,0 });
42     }
43
44     void path(int s) {
45         fill(all(seen), 0);
46         fill(all(dist), INF);
47         dist[s] = 0; ll di;
48
49         __gnu_pbds::priority_queue<pair<ll, int>> q;
50         vector<decltype(q)::point_iterator> its(N);
51         q.push({ 0, s });
52
53         while (!q.empty()) {
54             s = q.top().second; q.pop();
55             seen[s] = 1; di = dist[s] + pi[s];
56             for (edge& e : ed[s]) if (!seen[e.to]) {
57                 ll val = di - pi[e.to] + e.cost;
58                 if (e.cap - e.flow > 0 && val < dist[e.to]) {
59                     dist[e.to] = val;
60                     par[e.to] = &e;
61                     if (its[e.to] == q.end())
62                         its[e.to] = q.push({ -dist[e.to], e.to });
63                     else
64                         q.modify(its[e.to], { -dist[e.to], e.to });
65                 }
66             }
67         }
68         rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
69     }
70
71     pair<ll, ll> maxflow(int s, int t) {
72         ll totflow = 0, totcost = 0;
73         while (path(s), seen[t]) {
74             ll fl = INF;
75             for (edge* x = par[t]; x; x = par[x->from])
76                 fl = min(fl, x->cap - x->flow);
77
78             totflow += fl;
79             for (edge* x = par[t]; x; x = par[x->from]) {
80                 x->flow += fl;
81                 ed[x->to][x->rev].flow -= fl;
82             }
83         }
84         rep(i,0,N) for(edge& e : ed[i]) totcost += e.cost * e.flow;
85         return {totflow, totcost/2};
86     }
87
88     // If some costs can be negative, call this before maxflow:
89     void setpi(int s) { // (otherwise, leave this out)
90         fill(all(pi), INF); pi[s] = 0;
91         int it = N, ch = 1; ll v;
92         while (ch-- && it--)
93             rep(i,0,N) if (pi[i] != INF)
94                 for (edge& e : ed[i]) if (e.cap)
95                     if ((v = pi[i] + e.cost) < pi[e.to])
96                         pi[e.to] = v, ch = 1;
97         assert(it >= 0); // negative cost cycle
98     }
99 }

```

5.7 Push Relabel

```

1 const int inf = 1000000000;
2
3 int n;
4 vector<vector<int>> capacity, flow;
5 vector<int> height, excess, seen;
6 queue<int> excess_vertices;
7

```

```

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

```

```

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

```

6 Geometry

6.1 Point Struct

```

1 typedef long long T;
2 struct pt {
3     T x,y;
4     pt operator+(pt p) {return {x+p.x, y+p.y};}
5     pt operator-(pt p) {return {x-p.x, y-p.y};}
6     pt operator*(T d) {return {x*d, y*d};}
7     pt operator/(T d) {return {x/d, y/d};}
8 };
9
10 // cross product
11 // positivo si el segundo esta en sentido antihorario
12 // 0 si el angulo es 180
13 // negativo si el segundo esta en sentido horario
14 T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}
15
16 // dot product
17 // positivo si el angulo entre los vectores es agudo
18 // 0 si son perpendiculares
19 // negativo si el angulo es obtuso
20 T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}
21
22 T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}

```

```

23 T dist(pt a,pt b){
24     pt aux=b-a;
25     return sqrtl(aux.x*aux.x+aux.y*aux.y);
26 }

```

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));}
32     point perp() const{return point(-y, x);}
33     ld ang() const{
34         ld a = atan2l(y, x); a += le(a, 0) ? 2*pi : 0; return a;
35     }
36     ld dot(const point & p) const{return x * p.x + y * p.y;}
37     ld cross(const point & p) const{return x * p.y - y * p.x;}
38     ld norm() const{return x * x + y * y;}
39     ld length() const{return sqrtl(x * x + y * y);}
40     point unit() const{return (*this) / length();}
41
42     bool operator==(const point & p) const{return eq(x, p.x) && eq(y, p.y)
43         ;}
44     bool operator!=(const point & p) const{return !(*this == p);}
45     bool operator<(const point & p) const{return le(x, p.x) || (eq(x, p.x)
46         && le(y, p.y));}
47     bool operator>(const point & p) const{return ge(x, p.x) || (eq(x, p.x)
48         && ge(y, p.y));}
49     bool half(const point & p) const{return le(p.cross(*this), 0) || (eq(p
50         .cross(*this), 0) && le(p.dot(*this), 0));}
51
52     istream &operator>>(istream &is, point & p){return is >> p.x >> p.y;}
53     ostream &operator<<(ostream &os, const point & p){return os << "(" << p.
54         x << ", " << p.y << ")";}
55
56     int sgn(ld x){
57         if(ge(x, 0)) return 1;
58         if(le(x, 0)) return -1;
59         return 0;
60     }
61
62 };

```

```

53 }
54
55 void polarSort(vector<point> & P, const point & o, const point & v){
56     //sort points in P around o, taking the direction of v as first angle
57     sort(P.begin(), P.end(), [&](const point & a, const point & b){
58         return point((a - o).half(v), 0) < point((b - o).half(v), (a - o).
59             cross(b - o));
60     });
}

```

6.4 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;
}

```

6.5 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
9
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 }

```

6.6 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
30 ld areaUnionCircles(const vector<circ> & circs){
31     vector<circ> valid;
32     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.
42         r);
43     if(cruce.size() == 2){
44         ld a1 = (cruce[0] - prev.c).ang();
45         ld a2 = (cruce[1] - prev.c).ang();
46         ld b1 = (cruce[1] - nuevo.c).ang();
47         ld b2 = (cruce[0] - nuevo.c).ang();
48         if(a1 < a2){
49             prev.disable(a1, a2);
50         }else{
51             prev.disable(a1, 2*pi);
52             prev.disable(0, a2);
53         }
54         if(b1 < b2){
55             nuevo.disable(b1, b2);
56         }else{
57             nuevo.disable(b1, 2*pi);
58             nuevo.disable(0, b2);
59         }
60     }
61     valid.push_back(nuevo);
62 }
63 ld ans = 0;
64 for(const circ & curr : valid){
65     for(const auto & range : curr.getActive()){
66         ld l, r;
67         tie(l, r) = range;
68         ans += curr.r*(curr.c.x * (sin(r) - sin(l)) - curr.c.y * (cos(r) -
69             cos(l))) + curr.r*curr.r*(r-l);
70     }
71 }
72 return ans/2;
73 };

```

6.7 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 }

```

6.8 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 }

```

6.9 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 }

```

6.10 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    }
15    return {S.begin(), S.end()};
16 }

```

```

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 }
```

6.11 Common Tangents

```

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

6.12 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() - 1] - P[i]), 0)){
6             L.pop_back();
7         }
8     }
9 }
```

```

8     L.push_back(P[i]);
9 }
10 for(int i = P.size() - 1; i >= 0; i--){
11     while(U.size() >= 2 && leq((U[U.size() - 2] - P[i]).cross(U[U.size() - 1] - P[i]), 0)){
12         U.pop_back();
13     }
14     U.push_back(P[i]);
15 }
16 L.pop_back();
17 U.pop_back();
18 L.insert(L.end(), U.begin(), U.end());
19 }
20 }
```

6.13 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;
4 }
```

6.14 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     tv
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;
19 }
```

6.15 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 pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<point> & P){
81     if(r - l + 1 == 2){
82         QuadEdge* res = make_edge(P[l], P[r]);
83         return {res, res->rev()};
84 }

```

```

79
80 } if(r - l + 1 == 3){
81     QuadEdge *a = make_edge(P[l], P[l + 1]), *b = make_edge(P[l + 1], P[
82         r]);
83     splice(a->rev(), b);
84     int sg = sgn((P[l + 1] - P[l]).cross(P[r] - P[l]));
85     if(sg == 0)
86         return {a, b->rev()};
87     QuadEdge* c = connect(b, a);
88     if(sg == 1)
89         return {a, b->rev()};
90     else
91         return {c->rev(), c};
92 }
93 int mid = (l + r) / 2;
94 QuadEdge *ldo, *ldi, *rdo, *rdi;
95 tie(ldo, ldi) = build_tr(l, mid, P);
96 tie(rdi, rdo) = build_tr(mid + 1, r, P);
97 while(true){
98     if(left_of(rdi->origin, ldi)){
99         ldi = ldi->lnext();
100        continue;
101    }
102    if(right_of(ldi->origin, rdi)){
103        rdi = rdi->rev()->onext;
104        continue;
105    }
106    break;
107 }
108 QuadEdge* basel = connect(rdi->rev(), ldi);
109 auto valid = [&basel](QuadEdge* e){return right_of(e->dest(), basel)
110    ;};
111 if(ldi->origin == ldo->origin)
112     ldo = basel->rev();
113 if(rdi->origin == rdo->origin)
114     rdo = basel;
115 while(true){
116     QuadEdge* lcand = basel->rev()->onext;
117     if(valid(lcand)){
118         while(in_circle(basel->dest(), basel->origin, lcand->dest(),
119             lcand->onext->dest())){
120             lcand = lcand->onext;
121         }
122         QuadEdge* rcand = basel->oprev();
123         if(valid(rcand)){
124             while(in_circle(basel->dest(), basel->origin, rcand->dest(),
125                 rcand->oprev()->dest())){
126                 QuadEdge* t = rcand->oprev();
127                 delete_edge(rcand);
128                 rcand = t;
129             }
130             if(!valid(lcand) && !valid(rcand))
131                 break;
132             if(!valid(lcand) || (valid(rcand) && in_circle(lcand->dest(),
133                 lcand->origin, rcand->origin, rcand->dest())))
134                 basel = connect(rcand, basel->rev());
135             else
136                 basel = connect(basel->rev(), lcand->rev());
137         }
138         return {ldo, rdo};
139     }
140     vector<tuple<point, point, point>> delaunay(vector<point> & P){
141         sort(P.begin(), P.end());
142         auto res = build_tr(0, (int)P.size() - 1, P);
143         QuadEdge* e = res.first;
144         vector<QuadEdge*> edges = {e};
145         while(left_of(e->dest() - e->onext->dest()).cross(e->origin -
146             e->onext->dest()), 0)
147             e = e->onext;
148         auto add = [&P, &e, &edges](){
149             QuadEdge* curr = e;
150             do{
151                 curr->used = true;
152                 P.push_back(curr->origin);
153                 edges.push_back(curr->rev());
154                 curr = curr->lnext();
155             }while(curr != e);
156         };
157         add();
158         P.clear();
159         int kek = 0;
160     }
161 }
162

```

```

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 }
```

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

6.17 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 }
```

6.18 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 }
```

6.19 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         {point(34, 8), point(36, 9)}, {point(50, 20),
7             point(56, 24.5)}};
8     vector<pair<ld, ld>> radii = {{7, 4}, {3, 5}, {4, 4}, {4, 5}, {3, 3},
9         {4, 6}, {5, 1}, {10, 2.5}};
10    int n = centers.size();
11    for(int i = 0; i < n; ++i){
12        cout << "\n" << centers[i].first << " " << radii[i].first << " " <<
13            centers[i].second << " " << radii[i].second << "\n";
14        auto extLines = tangents(centers[i].first, radii[i].first, centers[i].
15            second, radii[i].second, false);
16        cout << "Exterior tangents:\n";
17        for(auto par : extLines){
18            for(auto p : par){
19                cout << p << " ";
20            }
21            cout << "\n";
22        }
23        auto intLines = tangents(centers[i].first, radii[i].first, centers[i].
24            second, radii[i].second, true);
25        cout << "Interior tangents:\n";
26        for(auto par : intLines){
27            for(auto p : par){
28                cout << p << " ";
29            }
30            cout << "\n";
31        }
32    }*/
33
34    /*int n;
35    cin >> n;
36    vector<point> P(n);
37    for(auto & p : P) cin >> p;
38    auto triangulation = delaunay(P);
39    for(auto triangle : triangulation){
40        cout << get<0>(triangle) << " " << get<1>(triangle) << " " << get
41            */
42 }
```

```

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

```

6.20 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(),
45                 poly.pop_front();
46             if(p.v.half({1, 0}) && poly.empty()) return {};
47             ch.push_back(p);
48             if(ch.size() >= 2) poly.push_back(ch[ch.size()-2].intersect(ch[ch.
49                 size()-1]));
50         }
51         while(ch.size() >= 3 && ch.front().outside(poly.back())) ch.pop_back()
52     }

```

```

    , poly.pop_back();
47 while(ch.size() >= 3 && ch.back().outside(poly.front())) ch.pop_front
    (), poly.pop_front();
48 poly.push_back(ch.back().intersect(ch.front())));
49 return vector<point>(poly.begin(), poly.end());
50 }
51
52 vector<point> halfPlaneIntersectionRandomized(vector<plane> planes){
53     point p = planes[0].a;
54     int n = planes.size();
55     random_shuffle(planes.begin(), planes.end());
56     for(int i = 0; i < n; ++i){
57         if(planes[i].inside(p)) continue;
58         ld lo = -inf, hi = inf;
59         for(int j = 0; j < i; ++j){
60             ld A = planes[j].v.cross(planes[i].v);
61             ld B = planes[j].v.cross(planes[j].a - planes[i].a);
62             if(ge(A, 0)){
63                 lo = max(lo, B/A);
64             }else if(le(A, 0)){
65                 hi = min(hi, B/A);
66             }else{
67                 if(ge(B, 0)) return {};
68             }
69             if(ge(lo, hi)) return {};
70         }
71         p = planes[i].a + planes[i].v*lo;
72     }
73     return {p};
74 }
```

6.21 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
        - n).perp());
4     ld r = (c - m).length();
5     return {c, r};
6 }
```

6.22 Intersection of Two Circles

```

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

6.23 Intersection Line and Circle

```

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

6.24 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     //Gets the area of the intersection of the polygon with the circle
7     int n = P.size();
8     ld ans = 0;
```

```

9  for(int i = 0; i < n; ++i){
10    point p = P[i], q = P[(i+1)%n];
11    bool p_inside = (pointInCircle(c, r, p) != 0);
12    bool q_inside = (pointInCircle(c, r, q) != 0);
13    if(p_inside && q_inside){
14      ans += (p - c).cross(q - c);
15    }else if(p_inside && !q_inside){
16      point s1 = intersectSegmentCircle(p, q, c, r)[0];
17      point s2 = intersectSegmentCircle(c, q, c, r)[0];
18      ans += (p - c).cross(s1 - c) + r*r * signed_angle(s1 - c, s2 - c);
19    }else if(!p_inside && q_inside){
20      point s1 = intersectSegmentCircle(c, p, c, r)[0];
21      point s2 = intersectSegmentCircle(p, q, c, r)[0];
22      ans += (s2 - c).cross(q - c) + r*r * signed_angle(s1 - c, s2 - c);
23    }else{
24      auto info = intersectSegmentCircle(p, q, c, r);
25      if(info.size() <= 1){
26        ans += r*r * signed_angle(p - c, q - c);
27      }else{
28        point s2 = info[0], s3 = info[1];
29        point s1 = intersectSegmentCircle(c, p, c, r)[0];
30        point s4 = intersectSegmentCircle(c, q, c, r)[0];
31        ans += (s2 - c).cross(s3 - c) + r*r * (signed_angle(s1 - c, s2 - c) +
32          signed_angle(s3 - c, s4 - c));
33      }
34    }
35  }
36  return abs(ans)/2;
}

```

6.25 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
4   vector<point> P = intersectLineCircle(a, b - a, c, r), ans;
5   for(const point & p : P){
6     if(pointInSegment(a, b, p)) ans.push_back(p);
7   }
8   return ans;
}

```

6.26 Line Intersection

```

1 int intersectLinesInfo(const point & a1, const point & v1, const point &
2   a2, const point & v2){
3   ld det = v1.cross(v2);
4   if(eq(det, 0)){
5     if(eq((a2 - a1).cross(v1), 0)){
6       return -1; //infinity points
7     }else{
8       return 0; //no points
9     }
10  }else{
11    return 1; //single point
12  }
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);
}

```

6.27 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;
10
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 }
```

6.28 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 }
```

6.29 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    }else{
21        auto l = upper_bound(P.rbegin(), P.rend() - right - 1, p);
22        int det = sgn((p - l[0]).cross((l == P.rbegin() ? P[0] : l[-1]) - 1
23            [0])) - 1;
24        if(det == -2) det = 1;
25        return det;
26    }
27 }
```

6.30 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 }
```

6.31 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 }
```

6.32 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 }
```

6.33 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 }
```

6.34 Points Tangency

```

1 pair<point, point> pointsOfTangency(const point & c, ld r, const point &
```

```

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

6.35 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 | }
```

6.36 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     }else{
10      return 0; //no point
11    }
12  }else{
13    return sgn(v.cross(c - a)) != sgn(v.cross(d - a)); //1: single point
14    , 0: no point
15  }
16}
17
18 int intersectSegmentsInfo(const point & a, const point & b, const point
19   & c, const point & d){
20   //segment ab, segment cd
21   point v1 = b - a, v2 = d - c;
22   int t = sgn(v1.cross(c - a)), u = sgn(v1.cross(d - a));
23   if(t == u){
24     if(t == 0){
25       if(pointInSegment(a, b, c) || pointInSegment(a, b, d) ||
26         pointInSegment(c, d, a) || pointInSegment(c, d, b)){
27         return -1; //infinity points
28       }else{
29     }
30   }

```

```
25     return 0; //no point
26 }
27 }else{
28     return 0; //no point
29 }
30 }else{
31     return sgn(v2.cross(a - c)) != sgn(v2.cross(b - c)); //1: single
32         point, 0: no point
33 }
```

6.37 Smallest Enclosing Circle

```

1 pair<point, ld> mec2(vector<point> & S, const point & a, const point & b
2   , int n){
3     ld hi = inf, lo = -hi;
4     for(int i = 0; i < n; ++i){
5       ld si = (b - a).cross(S[i] - a);
6       if(eq(si, 0)) continue;
7       point m = getCircle(a, b, S[i]).first;
8       ld cr = (b - a).cross(m - a);
9       if(le(si, 0)) hi = min(hi, cr);
10      else lo = max(lo, cr);
11    }
12    ld v = (ge(lo, 0) ? lo : le(hi, 0) ? hi : 0);
13    point c = (a + b) / 2 + (b - a).perp() * v / (b - a).norm();
14    return {c, (a - c).norm()};
15  }
16
17 pair<point, ld> mec(vector<point> & S, const point & a, int n){
18   random_shuffle(S.begin(), S.begin() + n);
19   point b = S[0], c = (a + b) / 2;
20   ld r = (a - c).norm();
21   for(int i = 1; i < n; ++i){
22     if(ge((S[i] - c).norm(), r)){
23       tie(c, r) = (n == S.size() ? mec(S, S[i], i) : mec2(S, a, S[i], i)
24         );
25     }
26   }
27   return {c, r};
28 }
29
30 pair<point, ld> smallestEnclosingCircle(vector<point> S){
31   point a, b;
32   ld r;
33   for(int i = 0; i < S.size(); ++i){
34     if(i == 0) a = S[i];
35     else if(i == 1) b = S[i];
36     else{
37       tie(a, b) = minEnclosingCircle(a, b, S[i]);
38     }
39   }
40   return {a, r};
41 }
```

```

29     assert(!S.empty());
30     auto r = mec(S, S[0], S.size());
31     return {r.first, sqrt(r.second)};
32 }
```

6.38 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 }
```

6.39 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[l], aux[l + rand() % (r - l)]);
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() + 1, 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(ge(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

```

1 struct TwoSatSolver {
2     int n_vars;                                // Number of boolean variables
3     int n_vertices;                            // Total vertices in the implication
4     graph (2 per variable)
5     vector<vector<int>> adj;                  // Implication graph: adj[i] contains
6         edges from node i
7     vector<vector<int>> adj_t;                // Transposed graph for Kosaraju's
8         algorithm
9     vector<bool> used;                        // Visited marker for DFS
10    vector<int> order;                      // Finishing order of vertices (DFS1)
11    vector<int> comp;                        // Component ID for each node (DFS2)
12    vector<bool> assignment;                 // Final truth assignment for each
13        variable
14
15    // Constructor initializes all data structures
16    TwoSatSolver(int _n_vars)
17        : n_vars(_n_vars),
18            n_vertices(2 * _n_vars),
19            adj(n_vertices),
20            adj_t(n_vertices),
21            used(n_vertices),
22            comp(n_vertices, -1),
23            assignment(n_vars) {
24                order.reserve(n_vertices); // Pre-allocate memory for efficiency
25            }
26
27    // First DFS pass for Kosaraju's algorithm (on original graph)
28    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

```

```

66     }
67     return true;
68 }

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

78     adj[neg_a].push_back(b);
79     adj[neg_b].push_back(a);
80     adj_t[b].push_back(neg_a);
81     adj_t[a].push_back(neg_b);
82 }
83 };

```

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 Use Case:
9   - Identifies vertices whose removal increases the number of
10    connected components.
11   - Works on undirected graphs (connected or disconnected).
12 */
13
14 int n; // Number of nodes in the graph
15 vector<vector<int>> adj; // Adjacency list of the undirected graph
16

```

```

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

21 // DFS traversal to identify articulation points
22 void dfs(int v, int p = -1) {
23     visited[v] = true;
24     tin[v] = low[v] = timer++;
25     int children = 0;
26     for (int to : adj[v]) {
27         if (to == p) continue; // Skip the parent edge
28         if (visited[to]) {
29             // Back edge
30             low[v] = min(low[v], tin[to]);
31         } else {
32             dfs(to, v);
33             low[v] = min(low[v], low[to]);
34             // Articulation point condition for non-root
35             if (low[to] >= tin[v] && p != -1) {
36                 // v is an articulation point
37                 // handle_cutpoint(v);
38             }
39             ++children;
40         }
41     }
42     // Articulation point condition for root
43     if (p == -1 && children > 1) {
44         // v is an articulation point
45         // handle_cutpoint(v);
46     }
47 }

48 // Initializes structures and launches DFS
49 void find_cutpoints() {
50     timer = 0;
51     visited.assign(n, false);
52     tin.assign(n, -1);
53     low.assign(n, -1);

54     for (int i = 0; i < n; ++i) {
55         if (!visited[i])
56             dfs(i);
57     }
58 }

```

```
59 }
60 }
```

7.3 Bellman-Ford

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

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

Path Reconstruction:
- To recover the path from source 's' to any node 'u':
vector<int> path;
for (int v = u; v != -1; v = parent[v])
    path.push_back(v);
reverse(path.begin(), path.end());
*/
const int INF = 1<<30; // Large value to represent "infinity"
vector<vector<pair<int, int>>> adj; // adj[v] = list of (neighbor,
// weight) pairs
vector<int> parent; // parent(n, -1) for path reconstruction

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

```
36     d[s] = 0;
37     q.push(s);
38     inqueue[s] = true;
39
40
41     while (!q.empty()) {
42         int v = q.front();
43         q.pop();
44         inqueue[v] = false;
45
46         for (auto edge : adj[v]) {
47             int to = edge.first;
48             int len = edge.second;
49
50             if (d[v] + len < d[to]) {
51                 parent[to] = v; // For path reconstruction
52                 d[to] = d[v] + len;
53                 if (!inqueue[to]) {
54                     q.push(to);
55                     inqueue[to] = true;
56                     cnt[to]++;
57                     if (cnt[to] > n)
58                         return false; // Negative weight cycle detected
59                 }
60             }
61         }
62     }
63
64     return true; // No negative cycles; shortest paths computed
65 }
```

7.4 Bipartite Checker

```
/*
Bipartite Graph Checker (BFS-based)
-----
Indexing: 0-based
Time Complexity: O(V + E)
Space Complexity: O(V)

Handles disconnected graphs
*/

```

```

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

```

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

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

```

```

        left node v

12
13 Output:
14     - Prints the pairs (left, right) in the matching
15     - mt[r] = 1 means right node r is matched to left node l
16 */
17
18 int n, k; // n: number of left nodes, k: number of right nodes
19 vector<vector<int>> g; // g[l]: list of right-side neighbors of left
20           node l
21 vector<int> mt; // mt[r]: matched left node for right node r (or
22           -1 if unmatched)
23 vector<bool> used; // used[l]: visited status for left node l during
24           DFS
25
26 // Try to find an augmenting path from left node v
27 bool try_kuhn(int v) {
28     if (used[v])
29         return false;
30     used[v] = true;
31     for (int to : g[v]) {
32         if (mt[to] == -1 || try_kuhn(mt[to])) {
33             mt[to] = v;
34             return true;
35         }
36     }
37     return false;
38 }
39
40 int main() {
41     //... reading the graph ...
42
43     mt.assign(k, -1); // Right-side nodes initially unmatched
44     for (int v = 0; v < n; ++v) {
45         used.assign(n, false); // Reset visited for each left node
46         try_kuhn(v);
47     }
48     // Output matched pairs (left+1, right+1 for 1-based output)
49     for (int i = 0; i < k; ++i) {
50         if (mt[i] != -1)
51             printf("%d %d\n", mt[i] + 1, i + 1);
52     }
53     return 0;
54 }

```

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[1]: adjacency list for left node 1 (list of right-side neighbors
13
14  Output:
15  - match_right[r] = 1 means right node r is matched to left node 1
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
28 // BFS to build layer graph, returns true if there's at least one
29 // unmatched node on the right reachable from left
30 bool bfs() {
31     queue<int> q;
32     dist.assign(n, INF);
33
34     for (int l = 0; l < n; ++l) {
35         if (match_left[l] == -1) {
36             dist[l] = 0;
37             q.push(l);
38         }
39     }
40
41     while (!q.empty()) {
42         int u = q.front();
43         q.pop();
44
45         for (int v : g[u]) {
46             if (match_right[v] == -1) {
47                 match_right[v] = u;
48                 match_left[u] = v;
49                 dist[v] = dist[u] + 1;
50                 q.push(v);
51             }
52         }
53     }
54
55     return !match_left.empty();
56 }

```

```

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
55     return found;
56 }
57
58 // DFS to find augmenting paths in the layered graph
59 bool dfs(int l) {
60     for (int r : g[l]) {
61         int matched_l = match_right[r];
62         if (matched_l == -1 || (dist[matched_l] == dist[l] + 1 && dfs(
63             matched_l))) {
64             match_left[l] = r;
65             match_right[r] = l;
66             return true;
67         }
68     }
69     dist[l] = INF;
70     return false;
71 }
72
73 // Main function to compute maximum matching
74 int hopcroft_karp() {
75     match_left.assign(n, -1);
76     match_right.assign(m, -1);
77
78     int matching = 0;
79     while (bfs()) {
80
81         for (int l : range(n)) {
82             if (match_left[l] == -1) {
83                 for (int r : g[l]) {
84                     if (match_right[r] == -1) {
85                         match_left[l] = r;
86                         match_right[r] = l;
87                         matching++;
88                     }
89                 }
90             }
91         }
92
93         for (int l : range(n)) {
94             if (match_left[l] == -1) {
95                 for (int r : g[l]) {
96                     if (match_right[r] == -1) {
97                         match_left[l] = r;
98                         match_right[r] = l;
99                         matching++;
100                    }
101                }
102            }
103        }
104    }
105
106    return matching;
107 }
```

```

78     for (int l = 0; l < n; ++l) {
79         if (match_left[l] == -1 && dfs(l)) {
80             ++matching;
81         }
82     }
83     return matching;
84 }
85 }

86 int main() {
87     cin >> n >> m;
88     g.assign(n, {});
89     int e; cin >> e;
90     while (e--) {
91         int u, v;
92         cin >> u >> v;
93         g[u].push_back(v); // u in [0, n-1], v in [0, m-1]
94     }
95 }

96 int res = hopcroft_karp();
97 cout << "Maximum_matching_size:" << res << '\n';
98 for (int r = 0; r < m; ++r) {
99     if (match_right[r] != -1) {
100         cout << match_right[r] + 1 << " " << r + 1 << '\n'; // 1-based
101         output
102     }
103 }
104 }
```

7.7 Block Cut Tree

```

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

Features:
- Identifies articulation points (cut vertices)
- Extracts all biconnected components (BCCs)
- Constructs the Block-Cut Tree:
```

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 vector<vector<int>> biconnected_components(vector<vector<int>> &g, //
 23 Adjacency list of the undirected graph
 24 vector<bool> &is_cutpoint, //
 25 Output vector (resized
 26 internally)
 27 vector<int> &id) { // Output
 28 vector (resized
 29 internally)

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

36 // DFS to find BCCs and articulation points
 37 function<void(int, int, int &)> dfs = [&](int node, int parent, int &
 38 timer) {
 39 num[node] = low[node] = ++timer;
 40 stk.push_back(node);
 41 for (int son : g[node]) {
 42 if (son == parent) continue;
 43 if (num[son]) {
 44 // Back edge
 45 low[node] = min(low[node], num[son]);
 46 } else {
 47 dfs(son, node, timer);
 48 low[node] = min(low[node], low[son]);
 49 // Check articulation point condition
 50 if (low[son] >= num[node]) {
 51 is_cutpoint[node] = (num[node] > 1 || num[son] > 2); // For
 52 root and non-root
 53 }
 54 }
 55 }
 56 }

```

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     }
87     return t;
88 };
89
90 return build_tree(); // Return the block-cut tree

```

```

91 | }

```

7.8 Blossom

```

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

```

```

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             v = orig[par[match[v]]]; // Move to the parent in the
73             // alternating tree
74         }
75         swap(v, w);
76     }
77
78 // Contract a blossom from v and w with common ancestor a
79 void blossom(int v, int w, int a) {
80     while (orig[v] != a) {
81         par[v] = w;
82         w = match[v];
83         ad[v] = true;
84         if (vis[w] == 1) Q.push(w), vis[w] = 0;
85         orig[v] = orig[w] = a;
86         v = par[w];
87     }
88 }
89
90 // Find augmenting path starting from unmatched node u
91 bool bfs(int u) {
92     fill(vis + 1, vis + n + 1, -1);
93     iota(orig + 1, orig + n + 1, 1);
94     Q = queue<int>();
95     Q.push(u);
96     vis[u] = 0;
97
98     while (!Q.empty()) {
99         int v = Q.front(); Q.pop();
100        ad[v] = true;
101        for (int x : g[v]) {
102            if (vis[x] == -1) {
103                par[x] = v;
104                vis[x] = 1;
105                if (!match[x]) {
106                    augment(u, x);
107                    return true;
108                }
109                Q.push(match[x]);
110                vis[match[x]] = 0;
111            } else if (vis[x] == 0 && orig[v] != orig[x]) {
112                int a = lca(orig[v], orig[x]);
113                blossom(x, v, a);
114                blossom(v, x, a);
115            }
116        }
117    }
118    return false;
119 }
120
121 // Computes maximum matching and returns the size
122 int maximum_matching() {

```

```

123     int ans = 0;
124     vector<int> p(n - 1);
125     iota(p.begin(), p.end(), 1);
126     shuffle(p.begin(), p.end(), rnd);
127     for (int i = 1; i <= n; ++i) {
128         shuffle(g[i].begin(), g[i].end(), rnd);
129     }
130
131     // Greedy matching: try to match unmatched nodes directly
132     for (int u : p) {
133         if (!match[u]) {
134             for (int v : g[u]) {
135                 if (!match[v]) {
136                     match[u] = v;
137                     match[v] = u;
138                     ++ans;
139                     break;
140                 }
141             }
142         }
143     }
144
145     // Augmenting path phase
146     for (int i = 1; i <= n; ++i) {
147         if (!match[i] && bfs(i)) ++ans;
148     }
149
150     return ans;
151 }
152 } M;
153
154 int main() {
155     ios_base::sync_with_stdio(0);
156     cin.tie(0);
157
158     int t;
159     cin >> t;
160     while (t--) {
161         int n, m;
162         cin >> n >> m;
163         M = Blossom(n);
164         // Read all edges
165         for (int i = 0; i < m; i++) {
166             int u, v;
167             cin >> u >> v;
168             M.add_edge(u, v);
169         }
170         // Compute max matching
171         int matched = M.maximum_matching();
172         if (matched * 2 == n) {
173             // Perfect matching
174             cout << 0 << '\n';
175         } else {
176             // Find reachable unmatched nodes in alternating trees
177             memset(M.ad, 0, sizeof M.ad);
178             for (int i = 1; i <= n; i++) {
179                 if (M.match[i] == 0) M.bfs(i);
180             }
181             int unmatched_reachable = 0;
182             for (int i = 1; i <= n; i++) {
183                 unmatched_reachable += M.ad[i];
184             }
185             cout << unmatched_reachable << '\n';
186         }
187     }
188     return 0;
189 }
```

7.9 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
9   Input:
10    n    - Number of nodes in the graph
11    adj - Adjacency list of the undirected graph
12
13  Output:
14    - Call 'find_bridges()' to populate bridge information.
15    - Modify the DFS 'Bridge' section to store or print the bridges.
16      A bridge is an edge (v, to) such that removing it increases the
```

```

        number of connected components.

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

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

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

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

```

```
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;

// Initializes the data structures
void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    last_visit.assign(n, 0);
    lca_iteration = 0;
}

```

```

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     dsu_cc_size[root] = dsu_cc_size[child];
71 }
72
73 // Merges paths from a and b to their lowest common ancestor in the 2ECC
74 // forest
75 void merge_path(int a, int b) {
76     ++lca_iteration;
77     vector<int> path_a, path_b;
78     int lca = -1;
79
80     while (lca == -1) {
81         if (a != -1) {
82             a = find_2ecc(a);
83             path_a.push_back(a);
84             if (last_visit[a] == lca_iteration) {
85                 lca = a;
86                 break;
87             }
88         }
89         if (b != -1) {
90             b = find_2ecc(b);
91             path_b.push_back(b);
92             if (last_visit[b] == lca_iteration) {
93                 lca = b;
94                 break;
95             }
96         }
97         last_visit[a] = lca_iteration;
98         a = par[a];
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 // Example usage
141 int main() {
142     init(n);
143     for (auto [u, v] : edges) {
144         add_edge(u, v);
145         cout << "Current_bridge_count:" << bridges << '\n';
146     }
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
4 priority_queue<pair<ll, int>> q; // Max-heap, so we push negative
5     weights to simulate min-heap
6 dist[0] = 0; // Starting node distance
7 q.push({0, 0}); // (distance, vertex)
8
9 while (!q.empty()) {
10     auto [w, v] = q.top(); q.pop();
11     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 Time Complexity: O(E)
14 Space Complexity: O(V + E)
15
16 Input:
17     - g: adjacency list of the graph
18         * Directed: vector<vector<pair<int, int>>> g
19             where g[v] = list of {to, edge_index}
20         * Undirected: vector<vector<int>> g
21             where g[v] = list of neighbors
22     - seen: vector<bool> seen(E) - only needed for directed version
23     - path: vector<int> path - will be filled in reverse order of
24         traversal
25             reverse(path.begin(), path.end());

```

```

22 */
23
24 // Directed Version //
25 void dfs_directed(int node) {
26     while (!g[node].empty()) {
27         auto [son, idx] = g[node].back();
28         g[node].pop_back();
29         if (seen[idx]) continue; // Skip if edge already visited
30         seen[idx] = true;
31         dfs_directed(son);
32     }
33     path.push_back(node); // Post-order insertion (reverse of actual path)
34 }
35
36 // Undirected Version //
37 // Be careful with graph not being connected. Add set to not repeat
38 // edges(going from u to v and then from v to u)
39 void dfs_undirected(int node) {
40     while (!g[node].empty()) {
41         int son = g[node].back();
42         g[node].pop_back();
43         dfs_undirected(son);
44     }
45     path.push_back(node); // Post-order insertion
}

```

7.13 Floyd-Warshall

```

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

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

```

```

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             long long new_dist = d[i][k] + d[k][j];
23             if (new_dist < d[i][j]) {
24                 d[i][j] = d[j][i] = new_dist; // update both directions for
25                                         undirected graph
26             }
27         }
28     }
}

```

7.14 Kruskal

```

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

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

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

Note:
- Requires a Disjoint Set Union (DSU) / Union-Find data structure
  with:
    - unite(a, b): merges components, returns true if successful
    - size(v): returns size of component containing v
*/
template <class T>
T kruskal(int N, vector<pair<T, pair<int, int>>> edges) {

```

```

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)
19    - pv[i]: Man matched to woman i
20 */
21 #define MAXN 1000
22 int gv[MAXN][MAXN], om[MAXN][MAXN]; // Male and female preference lists
23 int pv[MAXN], pm[MAXN];           // pv[woman] = man, pm[man] = woman
24 int pun[MAXN];                  // pun[man] = next woman to propose
25 to

```

```

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

```

7.16 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
51   // Read edges and build both original and reversed graphs
52   for (int i = 0; i < m; i++) {
53     int a, b;
54     cin >> a >> b;
55     a--; b--;
56     adj[a].push_back(b);
57     adjr[b].push_back(a);
58   }
59   // First pass: DFS on original graph to get order
60   vis.assign(n, false);
61   order.clear();
62   for (int i = 0; i < n; i++) {
63     if (!vis[i]) dfs(i);
64   }
65   // Second pass: DFS on reversed graph using reverse post-order
66   vis.assign(n, false);
67   int nc = 0;
68   for (int i = n - 1; i >= 0; i--) {
69     int v = order[i];
70     if (!vis[v]) {
71       dfsr(v, nc++);
72     }
73   }
74   // comp[i] now holds the component ID for node i (0-based)
75   // nc = number of strongly connected components
76 }

```

7.17 Stoer-Wagner

```

1 /*
2                                     Stoer - Wagner
3 -----
4 Solves the minimum cut problem in undirected weighted graphs with non-
5 negative weights.
6
7 Time Complexity: O(V ^ 3)
8 Space Complexity: O(V ^ 2)
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                     G[idx[j]][idx[s]] += G[idx[j]][idx[t]];
49                 }
50                 idx[t] = idx[-- n];
51             }
52             vis[idx[t]] = true;
53             s = t;
54             t = -1;
55             for (int j = 1; j < n; ++ j) {
56                 if (!vis[idx[j]]) {
57                     dis[idx[j]] += G[idx[s]][idx[j]];
58                     if (t == -1 || dis[idx[t]] < dis[idx[j]]) t = j;
59                 }
60             }
61         }
62     }
63 }
64
65 int32_t main() {
66     ios_base::sync_with_stdio(0);
67     cin.tie(0);
68     int t, cs = 0;
69     cin >> t;
70     while (t--) {
71         int n, m;
72         cin >> n >> m;
73         StoerWagner st(n);
74         while (m--) {
75             int u, v, w;
76             cin >> u >> v >> w;
77             st.add_edge(u, v, w);
78         }
79         cout << "Case " << ++cs << ":" << st.solve() << '\n';
80     }
81     return 0;
82 }
```

```

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 }
61 return ans;
62 }
63 };
64
65 int32_t main() {
66     ios_base::sync_with_stdio(0);
67     cin.tie(0);
68     int t, cs = 0;
69     cin >> t;
70     while (t--) {
71         int n, m;
72         cin >> n >> m;
73         StoerWagner st(n);
74         while (m--) {
75             int u, v, w;
76             cin >> u >> v >> w;
77             st.add_edge(u, v, w);
78         }
79         cout << "Case " << ++cs << ":" << st.solve() << '\n';
80     }
81     return 0;
82 }
```

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
12 const double eps = 1e-9;
13 int Gauss(vector<vector<double>> a, vector<double> &ans) {
14     int n = (int)a.size(), m = (int)a[0].size() - 1;
15     vector<int> pos(m, -1);
16     double det = 1; int rank = 0;
17     for(int col = 0, row = 0; col < m && row < n; ++col) {
18         int mx = row;
19         for(int i = row; i < n; i++) if(fabs(a[i][col]) > fabs(a[mx][col]))
20             mx = i;
21         if(fabs(a[mx][col]) < eps) {det = 0; continue;}
22         for(int i = col; i <= m; i++) swap(a[row][i], a[mx][i]);
23         if (row != mx) det = -det;
24         det *= a[row][col];
25         pos[col] = row;
26         for(int i = 0; i < n; i++) {
27             if(i != row && fabs(a[i][col]) > eps) {
28                 double c = a[i][col] / a[row][col];
29                 for(int j = col; j <= m; j++) a[i][j] -= a[row][j] * c;
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; //infinite
43             solutions
44         return 1; //unique solution
45     }
46     int main() {
47         int n, m; cin >> n >> m;
48         vector<vector<double>> v(n);
49         for(int i = 0; i < n; i++) {
50             for(int j = 0; j <= m; j++) {
51                 double x; cin >> x; v[i].push_back(x);
52             }
53         }
54     }
55 }
```

```

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] << '\u2022';
57 else cout << "no\u2022solution\n";
58 }
```

8.2 Gaussian Elimination Modulo

```

1 /*
2 Gaussian elimination Modulo 2
3 -----
4 Utilized for solving linear systems of equations modulo2.
5
6 Time Complexity: O(n ^ 3)
7 */
8
9 //n = number of equations, m = number of variables
10 int Gauss(int n, int m, vector<bitset<N>> a, bitset<N> &ans) {
11     //reversing for lexicographically largest solution
12     for (int i = 0; i < n; i++) {
13         bitset<N> tmp;
14         for (int j = 0; j < m; j++) tmp[j] = a[i][m - j - 1];
15         tmp[m] = a[i][m];
16         a[i] = tmp;
17     }
18     int rank = 0, det = 1;
19     vector<int> pos(N, -1);
20     for(int col = 0, row = 0; col < m && row < n; ++col) {
21         int mx = row;
22         for(int i = row; i < n; ++i) if(a[i][col]) { mx = i; break; }
23         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] ^
30             a[row];
31         ++row, ++rank;
32     }
33 }
```

```

31 }
32 ans.reset();
33 //backward substituition
34 for (int i = m - 1; i >= 0; i--) {
35     if (pos[i] == -1) ans[i] = true;
36     else {
37         int k = pos[i];
38         for (int j = i + 1; j < m; j++) if (a[k][j]) ans[i] = ans[i] ^
39             ans[j];
40         ans[i] = ans[i] ^ a[k][m];
41     }
42 }
43 for(int i = rank; i < n; ++i) if(a[i][m]) return -1; //no solution
44 //reversing again beacuse we reversed earlier
45 bitset<N> tmp;
46 for (int j = 0; j < m; j++) tmp[j] = ans[m - j - 1];
47 ans = tmp;
48 int free_var = 0;
49 for(int i = 0; i < m; ++i) if(pos[i] == -1) free_var++;
50 return free_var; //has solution
51 }
52 string read() {
53     string t;
54     if(!(cin >> t)) return "";
55     if(t.empty() || t == "and") return "";
56     while(t[0] == '(') t.erase(t.begin());
57     while(t.back() == ')') t.pop_back();
58     return t;
59 }
60 bool is_var(string t) { return t.size() > 0 && t[0] == 'x'; }
61 int get_var(string t) { return atoi(t.substr(1).c_str()) - 1; }
62 int32_t main() {
63     ios_base::sync_with_stdio(0);
64     cin.tie(0);
65     int n, m; cin >> n >> m;
66     vector<bitset<N>> bs(n, bitset<N>(0));
67     for(int i = 0; i < n; i++) {
68         string s;
69         bool eq = 1;
70         while((s = read()).size() > 0) {
71             if(is_var(s)) {
72                 int x = get_var(s);
73                 bs[i][x] = bs[i][x] ^ 1;
74             }
75         }
76     }
77 }
78
79
80
81
82
83
84
85
86

```

```

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

```

/*
Parametric Self-Dual Simplex method
Solve a canonical LP:
min or max. c x
s.t. A x <= b
    x >= 0
*/
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-9, oo = numeric_limits<double>::infinity();
typedef vector<double> vec;
typedef vector<vec> mat;
pair<vec, double> simplexMethodPD(const mat &A, const vec &b, const vec
    &c, bool mini = true){
    int n = c.size(), m = b.size();
    mat T(m + 1, vec(n + m + 1));
    vector<int> base(n + m), row(m);
    for(int j = 0; j < m; ++j){
        for(int i = 0; i < n; ++i)
            T[j][i] = A[j][i];
        row[j] = n + j;
        T[j][n + j] = 1;
        base[n + j] = 1;
    }
}

```

```

26     T[j][n + m] = b[j];
27 }
28
29 for(int i = 0; i < n; ++i)
30   T[m][i] = c[i] * (mini ? 1 : -1);
31
32 while(true){
33   int p = 0, q = 0;
34   for(int i = 0; i < n + m; ++i)
35     if(T[m][i] <= T[m][p])
36       p = i;
37
38   for(int j = 0; j < m; ++j)
39     if(T[j][n + m] <= T[q][n + m])
40       q = j;
41
42   double t = min(T[m][p], T[q][n + m]);
43
44   if(t >= -eps){
45     vec x(n);
46     for(int i = 0; i < m; ++i)
47       if(row[i] < n) x[row[i]] = T[i][n + m];
48     return {x, T[m][n + m] * (mini ? -1 : 1)}; // optimal
49   }
50
51   if(t < T[q][n + m]){
52     // tight on c -> primal update
53     for(int j = 0; j < m; ++j)
54       if(T[j][p] >= eps)
55         if(T[j][p] * (T[q][n + m] - t) >= T[q][p] * (T[j][n + m] - t))
56           q = j;
57
58     if(T[q][p] <= eps)
59       return {vec(n), oo * (mini ? 1 : -1)}; // primal infeasible
60   }else{
61     // tight on b -> dual update
62     for(int i = 0; i < n + m + 1; ++i)
63       T[q][i] = -T[q][i];
64
65     for(int i = 0; i < n + m; ++i)
66       if(T[q][i] >= eps)
67         if(T[q][i] * (T[m][p] - t) >= T[q][p] * (T[m][i] - t))
68           p = i;
69
70     if(T[q][p] <= eps)
71       return {vec(n), oo * (mini ? -1 : 1)}; // dual infeasible
72   }
73
74   for(int i = 0; i < m + n + 1; ++i)
75     if(i != p) T[q][i] /= T[q][p];
76
77   T[q][p] = 1; // pivot(q, p)
78   base[p] = 1;
79   base[row[q]] = 0;
80   row[q] = p;
81
82   for(int j = 0; j < m + 1; ++j){
83     if(j != q){
84       double alpha = T[j][p];
85       for(int i = 0; i < n + m + 1; ++i)
86         T[j][i] -= T[q][i] * alpha;
87     }
88   }
89
90   return {vec(n), oo};
91 }
92
93
94 int main(){
95   int m, n;
96   bool mini = true;
97   cout << "Numero_de_restricciones:" << endl;
98   cin >> m;
99   cout << "Numero_de_incognitas:" << endl;
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) << ":" << endl;
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?:" << endl;
111  cin >> mini;

```

```

112 cout << "Coeficientes_de_u" << (mini ? "min" : "max") << "uz:u";
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") << "uzu" << ans.second << ", cuando: u
119 \n";
120 for(int i = 0; i < ans.first.size(); ++i){
121     cout << "x_" << (i + 1) << "uzu" << 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;
}
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;
}
36

```

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;
}
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;
}

```

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;
23    }
24    inline Mat operator - (const Mat &b) {
25        assert(n == b.n && m == b.m);
26        Mat ans = Mat(n, m);
27        for(int i = 0; i < n; i++) {
28            for(int j = 0; j < m; j++) {

```

```

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

```

// Compute factorial (up to reasonable limits)
1 long long fact(int n) {
2     static vector<long long> f(21, -1); // cache up to 20!
3     if (n <= 1) return 1;
4     if (f[n] != -1) return f[n];
5     return f[n] = n * fact(n - 1);
6 }
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 }
25
26 // Given n and rank (0-indexed), reconstruct the permutation
27 vector<int> permutationUnrank0(int n, long long rank) {
28     vector<int> elems(n);
29     iota(elems.begin(), elems.end(), 1); // {1, 2, 3, ..., n}
30     vector<int> perm;
31
32     for (int i = n; i >= 1; --i) {
33         long long f = fact(i - 1);
34         int idx = rank / f;
35         rank %= f;
36         perm.push_back(elems[idx]);
37         elems.erase(elems.begin() + idx);
38     }
39
40     return perm;
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 {
6     phi[1]=1;
7     vector<ll> pr;
8     for(int i=2;i<kMaxV;i++){
9         if(sieve[i]==0){
10             sieve[i]=i;
11             pr.pb(i);
12             phi[i]=i-1;
13         }
14         for(auto p:pr){
15             if(p>sieve[i] || i*p>=kMaxV)break;
16             sieve[i*p]=p;
17             phi[i*p]=(p==sieve[i]?p:p-1)*phi[i];
18         }
19     }
20     for(int i=1;i<kMaxV;i++){
21         phi[i]+=phi[i-1];
22     }
23 }
```

```

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

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

9.15 Identities

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\frac{1}{n} \sum_{i=0}^n f(i)$ donde $f(i)$ es el número de collares que quedan fijos bajo una rotación de i posiciones.

Para contar el número de collares que quedan fijos bajo una rotación de i posiciones, se puede usar la fórmula $f(i) = 2^{\gcd(i,n)}$.

Para un collar de n cuentas y k colores, el número de collares diferentes que se pueden formar es $\frac{1}{n} \sum_{i=0}^n k^{\gcd(i,n)}$

Ejemplo: Dado un cubo con 6 caras y k colores, el número de cubos diferentes que se pueden formar es $\frac{1}{24} \sum_{i=0}^{24} f(i)$ donde $f(i)$ es el número de cubos que quedan fijos bajo una rotación de i posiciones. Esta formula es igual a $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$

9.17 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

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

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

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 $\binom{n}{\text{remaining}_\text{closed}} - \binom{n}{\text{remaining}_\text{closed}+1}$.
- binary trees with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

9.21 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

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

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 número de la pila p , entonces el número del juego es $g(p_1) \oplus g(p_2) \oplus \dots \oplus g(p_n)$. Para calcular el número 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.

$$\begin{aligned} \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} &= 1 \\ \begin{Bmatrix} 0 \\ n \end{Bmatrix} &= \begin{Bmatrix} n \\ 0 \end{Bmatrix} = 0 & , \quad n > 0 \\ \begin{Bmatrix} n \\ k \end{Bmatrix} &= (n-1) \begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} & , \quad k > 0 \\ \sum_{k=0}^n \begin{Bmatrix} n \\ k \end{Bmatrix} &= n! \\ \sum_{k=0}^{\infty} \begin{Bmatrix} n \\ k \end{Bmatrix} x^k &= \prod_{k=0}^{n-1} (x+k) \end{aligned}$$

11.2 Números de Stirling del segundo tipo

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

$$\begin{aligned} \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} &= 1 \\ \begin{Bmatrix} 0 \\ n \end{Bmatrix} &= \begin{Bmatrix} n \\ 0 \end{Bmatrix} = 0 & , \quad n > 0 \\ \begin{Bmatrix} n \\ k \end{Bmatrix} &= k \begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} & , \quad k > 0 \\ &= \sum_{j=0}^k \frac{j^n}{j!} \cdot \frac{(-1)^{k-j}}{(k-j)!} \end{aligned}$$

11.3 Números de Euler

$\langle\!\langle n \rangle\!\rangle$ 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".

$$\begin{aligned}\left\langle \begin{array}{c} 1 \\ 0 \end{array} \right\rangle &= 1 \\ \left\langle \begin{array}{c} n \\ k \end{array} \right\rangle &= (n-k) \left\langle \begin{array}{c} n-1 \\ k-1 \end{array} \right\rangle + (k+1) \left\langle \begin{array}{c} n-1 \\ k \end{array} \right\rangle \quad , \quad n \geq 2 \\ &= \sum_{j=0}^k (-1)^j \binom{n+1}{j} (k+1-j)^n \\ \sum_{k=0}^{n-1} \left\langle \begin{array}{c} n \\ k \end{array} \right\rangle &= n!\end{aligned}$$

11.4 Números de Catalan

$$C_0 = 1$$

$$\begin{aligned}C_n &= \frac{1}{n+1} \binom{2n}{n} = \sum_{j=0}^{n-1} C_j C_{n-1-j} \\ \sum_{n=0}^{\infty} C_n x^n &= \frac{1 - \sqrt{1 - 4x}}{2x}\end{aligned}$$

11.5 Números de Bell

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

$$\begin{aligned}B_n &= \sum_{k=0}^n \left\{ \begin{array}{c} n \\ k \end{array} \right\} = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k \\ \sum_{n=0}^{\infty} \frac{B_n}{n!} x^n &= e^{e^x - 1}\end{aligned}$$

11.6 Números de Bernoulli

$$\begin{aligned}B_0^+ &= 1 \\ B_n^+ &= 1 - \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k^+}{n-k+1} \\ \sum_{n=0}^{\infty} \frac{B_n^+ x^n}{n!} &= \frac{x}{1-e^{-x}} = \frac{1}{\frac{1}{1!} - \frac{x}{2!} + \frac{x^2}{3!} - \frac{x^3}{4!} + \dots}\end{aligned}$$

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

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

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 $xS'(x) + P(x)S(x) = 0$,

$$\text{donde } P(x) = \sum_{m=1}^{\infty} p_m x^m \text{ y } S(x) = \prod_{i=1}^n (1 - a_i x) = \sum_{m=0}^n (-1)^m s_m x^m.$$

11.11 Números armónicos

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

$$\gamma \approx 0.577215664901532860606512$$

11.12 Aproximación de Stirling

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

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

11.13 Ternas pitagóricas

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

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

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

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

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

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 $p \mid x - y$ pero $p \nmid x$ ni $p \nmid y$. Entonces:

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

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

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

- Suma de dos cuadrados: sea $\chi_4(n)$ una función multiplicativa igual a 1 si $n \equiv 1 \pmod{4}$, -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 $4(\chi_4 * \mathbf{1})(n) = 4 \sum_{d|n} \chi_4(d)$.

- 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:

$$x = \frac{x_0c - bk}{d}$$

$$y = \frac{y_0c + ak}{d}$$

para toda $k \in \mathbb{Z}$ si y solo si $d|c$, donde $ax_0 + by_0 = \gcd(a, b) = d$ (Euclides extendido). Si a y b tienen el mismo signo, hay exactamente $\max\left(\left\lfloor \frac{x_0c}{|b|} \right\rfloor + \left\lfloor \frac{y_0c}{|a|} \right\rfloor + 1, 0\right)$ 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:

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

- 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):

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

- En general, si queremos hallar $F(n)$ y existe una función mágica $g(n)$ tal que $G(n) = H(n)$ 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

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

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 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;
```

12.3 Day of the Week

```

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

12.4 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    struct edge {
24        int u, v; long long w;
25        bool operator > (const edge &rhs) const { return w > rhs.w; }
26    };
27    struct DSU {
28        vector<int> par;
29        DSU (int n) : par(n, -1) {}
30        int root(int i) { return par[i] < 0 ? i : par[i] = root(par[i]); }
31        void set_par(int c, int p) { par[c] = p; }
32    };
33    // returns parents of each vertex
34    // each edge should be distinct
35    // it assumes that a solution exists (all vertices are reachable from
36    // root)
37    // 0 indexed
38    // Takes ~300ms for n = 2e5
39    vector<int> DMST(int n, int root, const vector<edge> &edges) {
40        vector<int> u(2 * n - 1, -1), par(2 * n - 1, -1);
41        edge par_edge[2 * n - 1];
42        vector<int> child[2 * n - 1];
43
44        PQ<edge> Q[2 * n - 1];
45        for (auto e : edges) Q[e.v].push(e);
46        for (int i = 0; i < n; i++) Q[(i + 1) % n].push({i, (i + 1) % n, inf});
47
48        int super = n;
49        DSU dsu(2 * n - 1);
50        int head = 0;
51        while (Q[head].size()) {
52            auto x = Q[head].pop();
53            int nxt_root = dsu.root(x.u);
```

```

54     if (nxt_root == head) continue;
55     u[head] = nxt_root;
56     par_edge[head] = x;
57     if (u[nxt_root] == -1) head = nxt_root;
58     else {
59         int j = nxt_root;
60         do {
61             Q[j].add(-par_edge[j].w);
62             Q[super].merge(move(Q[j]));
63             assert(u[j] != -1);
64             child[super].push_back(j);
65             j = dsu.root(u[j]);
66         } while (j != nxt_root);
67         for (auto u : child[super]) par[u] = super, dsu.set_par(u, super);
68         head = super++;
69     }
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     res[root] = root; res.resize(n);
87     return res;
88 }
89 int32_t main() {
90     ios_base::sync_with_stdio(0);
91     cin.tie(0);
92     int n, m, root; cin >> n >> m >> root;
93     vector<edge> edges(m);
94     for (auto &e : edges) cin >> e.u >> e.v >> e.w;
95     auto res = DMST(n, root, edges);
96 }
```

```

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

```

12.5 GCD Convolution

```

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

```

```

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 }
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 }

```

```

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

```

12.7 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 }
```

12.8 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 }
```

12.9 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;
48 }

```

12.10 Max Manhattan Distance

```

1  /*
2   Max Manhattan distance
3   -----
4   Finds maximum manhattan distance for any two points with d dimensions.
5   Generalized code for d dimensions.
6   Time Complexity: (n * 2 ^ d * d)
7  */
8
9
10 long long ans = 0;
11 for (int msk = 0; msk < (1 << d); msk++) {
12     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 }

```

12.11 Mo

```

1  /*
2   Mo's Algorithm (Sqrt Decomposition for Offline Queries)
3   -----
4   Problem: Answer q range queries [L, R] over array of length n
5   using add/remove operations efficiently
6   Indexing: 0-based
7   Bounds:
8       - arr[0..n-1]
9       - queries input as 1-based and converted to 0-based
10  Time Complexity: O((n + q) * sqrt(n)) per query batch

```

```

11 Space Complexity: O(n + q)
12 Usage:
13   - Fill in logic for add/remove operations (update cur)
14   - Fill answers[] indexed by original query order
15 */
16
17 const int MAXN = 200500;
18 ll n, q;
19 ll arr[MAXN];      // input array
20 ll cnt[1000005];   // frequency count
21 ll answers[MAXN];  // output array
22 ll cur = 0;         // current query result
23 ll BLOCK_SIZE;
24
25 pair< pair<ll, ll>, ll> queries[MAXN]; // {{L, R}, query_index}
26
27 // Sort by block and then by R
28 inline bool cmp(const pair< pair<ll, ll>, ll> &x, const pair< pair<ll,
29   ll>, ll> &y) {
30   ll block_x = x.first.first / BLOCK_SIZE;
31   ll block_y = y.first.first / BLOCK_SIZE;
32   if (block_x != block_y) return block_x < block_y;
33   return x.first.second < y.first.second;
34 }
35
36 int main() {
37   cin >> n >> q;
38   BLOCK_SIZE = sqrt(n);
39   for (int i = 0; i < n; i++) cin >> arr[i];
40   for (int i = 0; i < q; i++) {
41     int l, r;
42     cin >> l >> r;
43     --l; --r; // convert to 0-based
44     queries[i] = {{l, r}, i};
45   }
46   sort(queries, queries + q, cmp);
47   ll l = 0, r = -1;
48   for (int i = 0; i < q; i++) {
49     int left = queries[i].first.first;
50     int right = queries[i].first.second;
51     // Expand to right
52     while (r < right) {
53       r++;
54     }
55     // Fill in logic for add/remove operations (update cur)
56     // Fill answers[] indexed by original query order
57   }
58 }
```

```

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

12.12 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;}
```

```

15 friend ModInt operator - (const ModInt &a) {return 0 - a;}
16 friend ModInt power(ModInt a, ll b) {ModInt ret(1); while (b > 0) {if
17     (b & 1) ret *= a; a *= a; b >>= 1;} return ret;}
18 friend ModInt inverse(ModInt a) {return power(a, MOD - 2);}
19 friend istream& operator >> (istream &is, ModInt &m) {is >> m.v; m.v =
20     (-MOD < m.v && m.v < MOD) ? m.v : m.v % MOD; if (m.v < 0) m.v +=
21     MOD; return is;}
22 friend ostream& operator << (ostream &os, const ModInt &m) {return os
23     << m.v;}
24 };

```

12.13 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 }

```

12.14 Next and Previous Smaller/Greater Element

```

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

```

```

15     res[s.top()] = i;
16     s.pop();
17 }
18 s.push(i);
19 }
20 return res;
}
22 vector<int> prevSmaller(vector<int> a, int n) {
23     stack<int> s;
24     vector<int> res(n, -1); // default: no smaller to left
25     for (int i = n - 1; i >= 0; i--) {
26         while (!s.empty() && a[s.top()] > a[i]) {
27             res[s.top()] = i;
28             s.pop();
29         }
30         s.push(i);
31     }
32     return res;
}

```

12.15 Parallel Binary Search

```

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

```

```

1 done = true;
2 // Reset changes of structure to 0 before each iteration
3 // Assign queries to current midpoint
4 for (int q = 0; q < k; q++) {
5     if (lo[q] < hi[q]) {
6         tocheck[(lo[q] + hi[q]) / 2].push_back(q);
7     }
8 }
9 // Apply changes and evaluate queries
10 for (int i = 0; i < m; i++) {
11     // Apply change for ith query to your data structure
12     apply_change(i);
13     for (int q : tocheck[i]) {
14         done = false; // at least one query is not done
15         // Evaluate query q using the updated structure
16         if (operationToCheck(q)) {
17             hi[q] = i; // condition is true, try earlier
18         } else {
19             lo[q] = i + 1; // condition false, try later
20         }
21     }
22 }
23 }
```

12.16 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(permuation.begin(), permuation.end(), rng);
```

12.17 setprecision

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

12.18 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)
7  * Bounds:
8  *   - Search range is [l, r]
9  *   - Precision controlled by eps
10 * Time Complexity: O(log((r - l) / eps))
11 *   - For minimum, reverse the condition (if f1 > f2 then l = m1)
12 */
13 double ternary_search(double l, double r) {
14     double eps = 1e-9; // precision error
15     while (r - l > eps) {
16         double m1 = l + (r - l) / 3;
17         double m2 = r - (r - l) / 3;
18         double f1 = f(m1); // evaluate function at m1
19         double f2 = f(m2); // evaluate function at m2
20         if (f1 < f2)
21             l = m1; // move left bound up if increasing
22         else
23             r = m2; // move right bound down if decreasing
24     }
25     return f(l); // return value at the best point
}
```

12.19 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

```

12.20 XOR Convolution

```

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

```

```

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 }

```

12.21 XOR Basis

```

/*
XOR Basis (Linear Basis over GF(2))
-----
Supports insertion of numbers into a basis and querying the maximum
XOR of any subset with a given value.
Bit Width: up to D bits (here D = 60 for 64-bit integers)
Indexing: basis[0] is for bit 0 (least significant), basis[D-1] for
bit D-1
Bounds:
    - Values x inserted must satisfy 0 <= x < 2^D
Time Complexity:
    - insert(x): O(D)
    - getMax(x): O(D)
Space Complexity: O(D)

Notes:
    - We maintain one basis vector per bit position.
    - On insert, we reduce x by existing basis vectors, then store it
    in the highest bit it still has.
    - To query max XOR, we greedily try to xor with basis vectors
    from highest bit down.
*/
const int D = 31; // Maximum number of bits in the numbers
int basis[D]; // basis[i] keeps the mask of the vector whose f value is
    i
int sz; // Current size of the basis
void insertVector(int mask) {
    //turn for around if you want max xor
}

```

```

29     for (int i = 0; i < D; i++) {
30         if ((mask & 1 << i) == 0) continue; // continue if i != f(mask)
31         if (!basis[i]) { // If there is no basis vector with the ith bit set
32             , then insert this vector into the basis
33             basis[i] = mask;
34             ++sz;
35             return;
36         }
37         mask ^= basis[i]; // Otherwise subtract the basis vector from this
38         // vector
39     }
40
41 // V2: If you dont need the basis sorted.
42 vector<ll> basis;
43 void add(ll x)
44 {
45     for (int i = 0; i < basis.size(); i++)
46     {
47         x = min(x, x ^ basis[i]);
48     }
49     if (x != 0)
50     {
51         basis.pb(x);
52     }
53 }
```

12.22 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        }
16    }
17 }
```

```

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     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
43 C++
44 41 lines
45 1161 bytes
```

```

47
48 int get_the_same_high_bit (int bits, vector <int> &val) {
49     for (auto &i : basis) {
50         if (__builtin_popcount(val[i.id] & bits) & 1) {
51             return i.id;
52         }
53     }
54     return -1;
55 }
```

```

57 C++
58 8 lines
59 200 bytes
60
61 Thus, we have learned to delete in O(m2).
62
63 Here is full version of the code:
64 Code
65
66 //#pragma GCC optimize("Ofast", "unroll-loops")
67 //#pragma GCC target("sse", "sse2", "sse3", "ssse3", "sse4")
68
69 #include <bits/stdc++.h>
70
71 #define all(a) a.begin(), a.end()
72 #define len(a) (int)(a.size())
73 #define mp make_pair
74 #define pb push_back
75 #define fir first
76 #define sec second
77 #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 const int C = 20;
115
116 struct node {
117     int val, id, mask, high;
118 };
119
120
121 inline int get_high (int x) {
122     if (x == 0) {
123         return -1;
124     }
125     return 31 - __builtin_clz(x);
126 }
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 }

159
160
161 inline int get_the_same_high_bit (int bits, const vector <int> &val) {
162     for (auto &i : basis) {
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> basises[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

```

```
228 inline int add (int x) {
229     val.pb(x);
230     where.pb(make_pair(-1, -1));
231     int id = cur_id++;
232     if (x == 0) {
233         return id;
234     }
235     for (int i = mx; i >= 0; i--) {
236         if (baseses[i].empty()) {
237             continue;
238         }
239         if (baseses[i].back().add(x, id)) {
240             baseses[i + 1].pb(baseses[i].back());
241             baseses[i].pop_back();
242             umax(mx, i + 1);
243             for (auto &j : baseses[i + 1].back().basis) {
244                 where[j.id] = make_pair(i + 1, len(baseses[i + 1]) -
245                                         1);
246             }
247             return id;
248         }
249         baseses[1].pb(xor_basis());
250         baseses[1].back().add(x, id);
251         where.back() = make_pair(1, len(baseses[1]) - 1);
252         umax(mx, 1);
253         return id;
254     }
255
256 inline int move_back (int sz, int pos) {
257     int to = len(baseses[sz]) - 1;
258     if (to == pos) {
259         return pos;
260     }
261     for (auto &i : baseses[sz][pos].basis) {
262         where[i.id].second = to;
263     }
264     for (auto &i : baseses[sz][to].basis) {
265         where[i.id].second = pos;
266     }
267     swap(baseses[sz][pos], baseses[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 = baseses[sz].back().rebuild_and_delete(id);
280         int i = sz - 1;
281         while (i > 0 && baseses[i].empty()) {
282             --i;
283         }
284         int new_id = -1;
285         if (i > 0) {
286             new_id = baseses[i].back().get_the_same_high_bit(val);
287         }
288         if (new_id == -1) {
289             if (sz > 1) {
290                 baseses[sz - 1].pb(baseses[sz].back());
291                 for (auto &j : baseses[sz - 1].back().basis) {
292                     where[j.id] = make_pair(sz - 1, len(baseses[sz - 1]) -
293                                         1);
294                 }
295             }
296             baseses[sz].pop_back();
297             if (mx > 0 && baseses[mx].empty()) {
298                 --mx;
299             }
300             return;
301         }
302         int cur = val[new_id];
303         assert(baseses[sz].back().add(cur, new_id));
304         int new_sz = i;
305         int new_pos = len(baseses[i]) - 1;
306         where[new_id] = make_pair(sz, pos);
307         id = new_id;
308         sz = new_sz;
309         pos = new_pos;
310     }
}
```

```

311     inline int size () {
312         return mx;
313     }
314 };
315
316 int main() {
317     //    freopen("input.txt", "r", stdin);
318     //    freopen("output.txt", "w", stdout);
319
320     ios_base::sync_with_stdio(0);
321     cin.tie(0);
322
323
324     // solution to https://basecamp.eolymp.com/uk/problems/11732
325     int n, q, add;
326     cin >> n >> add;
327     vector <int> p(n);
328     xor_basis_online<19> t;
329     vector <int> now(n);
330     for (int i = 0; i < n; i++) {
331         cin >> p[i];
332         now[i] = t.add(i ^ p[i]);
333     }
334     cin >> q;
335     int ans = t.size();
336     cout << ans << '\n';
337     for (int i = 0, l, r; i < q; i++) {
338         cin >> l >> r;
339         l = (l + ans * add) % n;
340         r = (r + ans * add) % n;
341         if (l != r) {
342             t.del(now[l]);
343             t.del(now[r]);
344             swap(p[l], p[r]);
345             now[l] = t.add(l ^ p[l]);
346             now[r] = t.add(r ^ p[r]);
347         }
348         ans = t.size();
349         cout << ans << '\n';
350     }
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
7                             // failed on
8     for (int i=0; i<(int)s.size(); i++) {
9         // evaluate c(i)
10        // delta = s_i - \sum_{j=1}^n c_j s_{i-j}
11        // if delta == 0, c(i) is correct
12        T delta = s[i];
13        for (int j=1; j<=(int)c.size(); j++)
14            delta -= c[j-1] * s[i-j];   // c_j is one-indexed, so we
15                             // actually need index j - 1 in the code
16        if (delta == 0)
17            continue; // c(i) is correct, keep going
18        // now at this point, delta != 0, so we need to adjust it
19        if (f == -1) {
20            // this is the first time we're updating c
21            // s_i was the first non-zero element we encountered
22            // we make c of length i + 1 so that s_i is part of the base
23            // case
24            c.resize(i + 1);
25            mt19937 rng(chrono::steady_clock::now().time_since_epoch().
26                         count());
27            for (T &x : c)
28                x = rng(); // just to prove that the initial values don
29                             // t matter in the first step, I will set to random
30                             // values
31            f = i;
32        } else {
33            // we need to use a previous version of c to improve on this
34            // one
35            // apply the 5 steps to build d
36            // 1. set d equal to our chosen sequence
37            vector<T> d = oldC;
38            // 2. multiply the sequence by -1
39            for (T &x : d)
40                x = -x;
41            oldC = c;
42            c = d;
43        }
44    }
45    return c;
46 }
```

```

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

13.2 FFT

```

1 using cd = complex<double>;
2 const double PI = acos(-1);
3 //declare size of vectors used like this
4 const int MAXN=2<<19;
5
6 void fft(vector<cd> & a, bool invert) {
7     int n = (int)a.size();
8 }
```

```

9     for (int i = 1, j = 0; i < n; i++) {
10         int bit = n >> 1;
11         for (; j & bit; bit >>= 1)
12             j ^= bit;
13         j ^= bit;
14
15         if (i < j)
16             swap(a[i], a[j]);
17     }
18
19     for (int len = 2; len <= n; len <= 1) {
20         double ang = 2 * PI / len * (invert ? -1 : 1);
21         cd wlen(cos(ang), sin(ang));
22         for (int i = 0; i < n; i += len) {
23             cd w(1);
24             for (int j = 0; j < len / 2; j++) {
25                 cd u = a[i+j], v = a[i+j+len/2] * w;
26                 a[i+j] = u + v;
27                 a[i+j+len/2] = u - v;
28                 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 }
```

```

52
53     vector<int> result(n);
54     for (int i = 0; i < n; i++)
55         result[i] = round(fa[i].real());
56     return result;
57 }
58
59 //normalizing for when mult is between 2 big numbers and not polynomials
60 int carry = 0;
61 for (int i = 0; i < n; i++){
62     result[i] += carry;
63     carry = result[i] / 10;
64     result[i] %= 10;
65 }

```

13.3 NTT

```

1 // number theory transform
2
3 const int MOD = 998244353, ROOT = 3;
4 // const int MOD = 7340033, ROOT = 5;
5 // const int MOD = 167772161, ROOT = 3;
6 // const int MOD = 469762049, ROOT = 3;
7
8 int power(int base, int exp) {
9     int res = 1;
10    while (exp) {
11        if (exp % 2) res = 1LL * res * base % MOD;
12        base = 1LL * base * base % MOD;
13        exp /= 2;
14    }
15    return res;
16 }
17
18 void ntt(vector<int>& a, bool invert) {
19     int n = a.size();
20     for (int i = 1, j = 0; i < n; i++) {
21         int bit = n >> 1;
22         for (; j & bit; bit >>= 1) j ^= bit;
23         j ^= bit;
24         if (i < j) swap(a[i], a[j]);
25     }
26     for (int len = 2; len <= n; len <= 1) {

```

```

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

```

13.4 Roots NTT

```

1 1*2^0 + 1 = 2, 1, 1
2 1*2^1 + 1 = 3, 2, 2
3 1*2^2 + 1 = 5, 2, 3
4 2*2^3 + 1 = 17, 2, 9
5 1*2^4 + 1 = 17, 3, 6
6 3*2^5 + 1 = 97, 19, 46
7 3*2^6 + 1 = 193, 11, 158
8 2*2^7 + 1 = 257, 9, 200
9 1*2^8 + 1 = 257, 3, 86

```

```

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

```

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)

```

```

13 */
14
15 const ll MOD = 212345678987654321LL;
16 const ll base = 33;
17
18 class HashedString {
19     private:
20         static const long long M = 1e9 + 9;
21         static const long long B = 9973;
22
23         static vector<long long> pow;
24
25         vector<long long> p_hash;
26
27     public:
28         HashedString(const string &s) : p_hash(s.size() + 1) {
29             while (pow.size() <= s.size()) { pow.push_back((pow.back() * B) % M);
30                 ;
31             }
32
33             p_hash[0] = 0;
34             for (int i = 0; i < (int)s.size(); i++) {
35                 p_hash[i + 1] = ((p_hash[i] * B) % M + s[i]) % M;
36             }
37
38         }
39
40         // substring [start, end]
41         long long get_hash(int start, int end) {
42             long long raw_val =
43                 (p_hash[end + 1] - (p_hash[start] * pow[end - start + 1]));
44             return (raw_val % M + M) % M;
45         }
46
47         // substring [start, end] with 1 added at the end. Remember it is
48         // ascii characters, not the number from 1 to 26 for the hash.
49         long long get_hash(int start, int end, int l){
50             if(start > end) return 1;
51             long long val = ((p_hash[end + 1] * B) + 1 - (p_hash[start] * pow[
52                 end - start + 2]));
53             val = (val % M + M) % M;
54             return val;
55         }
56
57         // you cant skip this
58         vector<long long> HashedString::pow = {1};

```

14.2 KMP

```

1  /*
2   *          KMP
3   -----
4   Computes the prefix function for a string.
5
6   Maximum length of substring that ends at position i and is proper
7   prefix (not equal to string itself) of string
8   pf[i]  is the length of the longest proper prefix of the substrings
9   [0.....i]$ which is also a suffix of this substring.
10
11  For matching, one can append the string with a delimiter like $
12      between them
13
14
15 vector<int> KMP(string s){
16     int n=(int)s.length();
17     vector<int> pf(n, 0);
18     for(int i=1;i<n;i++){
19         int j=pf[i-1];
20         while(j>0 && s[i]!=s[j]){
21             j=pf[j-1];
22         }
23         if(s[i]==s[j]){
24             pf[i]=j+1;
25         }
26     }
27     return pf;
28 }
29
30 // Counts how many times each prefix occurs
31 // Same thing can be done for two strings but only considering indices
32 // of second string
33 vector<int> count_occurrences_of_prefixes(vector<int> pf){
34     int n=(int)pf.size();
35     vector<int> ans(n + 1);
36     for (int i = 0; i < n; i++)
37         ans[pi[i]]++;
38     for (int i = n-1; i > 0; i--)
39         ans[pi[i-1]] += ans[i];
40     for (int i = 0; i <= n; i++)
41         ans[i]++;
42 }
43
44 // Computes automaton for string
45 // useful for not having to recalculate KMP of string s
46 // can be utilized when the second string (the one in which we are
47 // trying to count occurrences)
48 // is very large
49 void compute_automaton(string s, vector<vector<int>>& aut) {
50     s += '#';
51     int n = s.size();
52     vector<int> pi = KMP(s);
53     aut.assign(n, vector<int>(26));
54     for (int i = 0; i < n; i++) {
55         for (int c = 0; c < 26; c++) {
56             if (i > 0 && 'a' + c != s[i])
57                 aut[i][c] = aut[pi[i-1]][c];
58             else
59                 aut[i][c] = i + ('a' + c == s[i]);
60     }
61 }
```

```

38     ans[pi[i-1]] += ans[i];
39     for (int i = 0; i <= n; i++)
40         ans[i]++;
41 }
42
43 // Computes automaton for string
44 // useful for not having to recalculate KMP of string s
45 // can be utilized when the second string (the one in which we are
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         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);
}

```

14.4 Manacher

```

/*
    Manacher
-----
Computes the length of the longest palindrome centered at position i.

p[i] is length of biggest palindrome centered in this position.
Be careful with characters that are inserted to account for odd and
even palindromes

Time Complexity: O(n)
Space Complexity: O(n)

*/
// Number of palindromes centered at each position

vector<int> manacher_odd(string s)
{
    int n = s.size();

```

```

19     s = "$" + s + "^";
20     vector<int> p(n + 2);
21     int l = 1, r = 1;
22     for (int i = 1; i <= n; i++)
23     {
24         p[i] = max(0, min(r - i, p[l + (r - i)]));
25         while (s[i - p[i]] == s[i + p[i]])
26         {
27             p[i]++;
28         }
29         if (i + p[i] > r)
30         {
31             l = i - p[i], r = i + p[i];
32         }
33     }
34     return vector<int>(begin(p) + 1, end(p) - 1);
}
vector<int> manacher(string s)
{
    string t;
    for (auto c : s)
    {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}

// usage
// vector<int> p = manacher("abacaba");
// this will return {2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2}
// vector<int> p = manacher("abaaba");
// this will return {2, 1, 4, 1, 2, 7, 2, 1, 4, 1, 2}

```

14.5 Suffix Array

```

/*
    Suffix Array
-----
Computes the suffix array of a string in O(n log n).
Sorted array of all cyclic shifts of a string.

If you want sorted suffixes append $ to the end of the string.

```

```

8   lc is longest common prefix. Lcp of two substrings j > i is min(lc[i],
....., lc[j - 1]).  

9  

10 To compute Largest common substring of multiple strings  

11 Join all strings separating them with special character like $ (it has  

    to be different for each string)  

12 Sliding window on lcp array (all string have to appear on the sliding  

    window and  

13 the lcp of the interval will give the length of the substring that  

    appears on all strings)  

14  

15 Time Complexity: O(n log n)  

16 Space Complexity: O(n)  

17  

18 */
19  

20 struct SuffixArray
21 {
22     int n;
23     string t;
24     vector<int> sa, rk, lc;
25     SuffixArray(const std::string &s)
26     {
27         n = s.length();
28         t = s;
29         sa.resize(n);
30         lc.resize(n - 1);
31         rk.resize(n);
32         std::iota(sa.begin(), sa.end(), 0);
33         std::sort(sa.begin(), sa.end(), [&](int a, int b)
34             { return s[a] < s[b]; });
35         rk[sa[0]] = 0;
36         for (int i = 1; i < n; ++i)
37             rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
38         int k = 1;
39         std::vector<int> tmp, cnt(n);
40         tmp.reserve(n);
41         while (rk[sa[n - 1]] < n - 1)
42         {
43             tmp.clear();
44             for (int i = 0; i < k; ++i)
45                 tmp.push_back(n - k + i);
46             for (auto i : sa)

```

```

47                 if (i >= k)
48                     tmp.push_back(i - k);
49                 std::fill(cnt.begin(), cnt.end(), 0);
50                 for (int i = 0; i < n; ++i)
51                     ++cnt[rk[i]];
52                 for (int i = 1; i < n; ++i)
53                     cnt[i] += cnt[i - 1];
54                 for (int i = n - 1; i >= 0; --i)
55                     sa[--cnt[rk[tmp[i]]]] = tmp[i];
56                 std::swap(rk, tmp);
57                 rk[sa[0]] = 0;
58                 for (int i = 1; i < n; ++i)
59                     rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i - 1] + k == n || tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
60                     k *= 2;
61                 }
62                 for (int i = 0, j = 0; i < n; ++i)
63                 {
64                     if (rk[i] == 0)
65                     {
66                         j = 0;
67                     }
68                     else
69                     {
70                         for (j -= j > 0; i + j < n && sa[rk[i] - 1] + j < n && s[i + j]
71                             == s[sa[rk[i] - 1] + j]);
72                             ++j;
73                         lc[rk[i] - 1] = j;
74                     }
75                 }
76
77 // Finds if string p appears as substring in the string
78 // might now work perfectly
79 int search(string &p){
80     int tam = p.size();
81     int l = 0, r = n;
82
83     string tmp = "";
84     while(r > 1) {
85         int m = l + (r-1)/2;
86         tmp = t.substr(sa[m], min(n-sa[m], tam));
87         if(tmp >= p){

```

```

88         r = m;
89     } else {
90         l = m + 1;
91     }
92 }
93 if(l < n) {
94     tmp = t.substr(sa[l], min(n-sa[l], tam));
95 } else{
96     return -1;
97 }
98 if(tmp == p){
99     return 1;
100 } else {
101     return -1;
102 }
103 }

// Counts number of times a string p appears as substring in string
104 int count(string &p) {
105     int x = search(p);
106     if(x == -1) return 0;
107     int cnt = 0;
108     int tam = p.size();
109     int maxx = 0;
110     while((1 << maxx) + x < n) maxx++;
111     int y = x;
112     for(int i = maxx-1; i >= 0; i--) {
113         if(x + (1 << i) >= n) continue;
114         string tmp = t.substr(sa[x + (1 << i)], min(n-sa[x + (1 << i)]
115             ], tam));
116         if(tmp == p) x += (1 << i);
117     }
118     return x-y+1;
119 }
120 }
121 }
122 }
123 int main() {
124     cin.tie(0)->sync_with_stdio(0);
125     string s; cin >> s;
126     SuffixArray SA(s);
127
128     int q; cin >> q;
129     for(int t = 0; t < q; t++) {

```

```

130         string tmp; cin >> tmp;
131         cout << SA.count(tmp) << endl;
132     }
133
134     return 0;
135 }

```

14.6 Suffix Automaton

```

1 /*
2
3
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;

```

```

} }

// construction is O(n)
void insertChar(char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    st[cur].firstposition=st[last].len;
    int p = last;
    while (p != -1 && !st[p].next.count(c)) {
        st[p].next[c] = cur;
        p = st[p].link;
    }
    if (p == -1) {
        st[cur].link = 0;
    } else {
        int q = st[p].next[c];
        if (st[p].len + 1 == st[q].len) {
            st[cur].link = q;
        } else {
            int clone = sz++;
            st[clone].len = st[p].len + 1;
            st[clone].next = st[q].next;
            st[clone].link = st[q].link;
            st[clone].firstposition=st[q].firstposition;
            while (p != -1 && st[p].next[c] == q) {
                st[p].next[c] = clone;
                p = st[p].link;
            }
            st[q].link = st[cur].link = clone;
        }
    }
    last = cur;
    cnt[last]=1;
}

// searches for the starting position in O(len(s)). Returns starting
// index of first 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++){
        if(cntSum[i]==cnt[i])

```

```

117     cntSum[i]=cnt[i];
118 }
119 dfs1(0);
120 }
121
122
123 // Counts number of paths that can reach specific state.
124 void countPathsReverse(){
125     cnt1[0]=1;
126     queue<int> q;
127     q.push(0);
128     vector<int> in(2*MAXN, 0);
129     for(int i=0;i<sz;i++){
130         for(auto [x, y]:st[i].next){
131             in[y]++;
132         }
133     }
134     while((int)q.size()){
135         int cur=q.front();
136         q.pop();
137         for(auto [x, y]:st[cur].next){
138             cnt1[y]+=cnt1[cur];
139             in[y]--;
140             if(in[y]==0){
141                 q.push(y);
142             }
143         }
144     }
145 }
146
147 // Computes the kth smallest string that appears on the string (counting
   repetitions)
148 string kthSmallest(ll k){
149     string s="";
150     int cur=0;
151     while(k>0){
152         for(auto [c, y]:st[cur].next){
153             if(k>cntSum[y]) k-=cntSum[y];
154             else{
155                 k-=cnt[y];
156                 s+=c;
157                 cur=y;
158                 break;
159             }
160         }
161     }
162     return s;
163 }
164
165
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 }
224
225
226 int main(){
227     ios_base::sync_with_stdio(false); cin.tie(NULL);
228     string s; cin >> s;
229     initSuffixAutomaton();
230     for(char c:s){
231         insertChar(c);
232     }
233 }
```

14.7 Trie Ahocorasick

```

1 /*
2          Trie - AhoCorasick
3 -----
4 Builds a trie for subset of strings and computes suffix links.
5 KATCL implementation is cleaner.
6 
```

```

7 Time Complexity(Construction): O(m) where m is sum
8 of lengths of strings
9 Space Complexity: O(m)
10 */
11
12
13
14 const int K = 26;
15
16 struct Vertex {
17     int next[K];
18     bool output = false;
19     int p = -1;
20     char pch;
21     int link = -1;
22     int go[K];
23
24     Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
25         fill(begin(next), end(next), -1);
26         fill(begin(go), end(go), -1);
27     }
28 };
29
30 vector<Vertex> t(1);
31
32 void add_string(string const& s) {
33     int v = 0;
34     for (char ch : s) {
35         int c = ch - 'a';
36         if (t[v].next[c] == -1) {
37             t[v].next[c] = t.size();
38             t.emplace_back(v, ch);
39         }
40         v = t[v].next[c];
41     }
42     t[v].output = true;
43 }
44
45 int go(int v, char ch);
46
47 int get_link(int v) {
48     if (t[v].link == -1) {
49         if (v == 0 || t[v].p == 0)
```

```

50     t[v].link = 0;
51     else
52         t[v].link = go(get_link(t[v].p), t[v].pch);
53     }
54     return t[v].link;
55 }
56
57 int go(int v, char ch) {
58     int c = ch - 'a';
59     if (t[v].go[c] == -1) {
60         if (t[v].next[c] != -1)
61             t[v].go[c] = t[v].next[c];
62         else
63             t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
64     }
65     return t[v].go[c];
66 }
```

14.8 Z Function

```

1 /*
2      Z_function
3 -----
4 Computes the z_function for any string.
5 ith element is equal to the greatest number of characters starting
6 from the position i that coincide with the first characters of s
7
8 z[i] length of the longest string that is, at the same time, a prefix
9 of s and a prefix of $$s starting at i.
10
11 to compress string, one can run z_function and then find the smallest
12 i that divides n such that i + z[i] = n
13
14 Time Complexity: O(n)
15 Space Complexity: O(n)
16 */
17
18 vector<int> z_function(string s) {
19     int n = s.size();
20     vector<int> z(n);
21     int l = 0, r = 0;
22     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

```

1 /*
2      Centroid Decomposition
3 -----
4 Finds the centroid decomposition of a given tree.
5 Any vertex can have at most log n centroid ancestors
6
7 The code below is the solution to Xenia and tree.
8 Given tree, queries of two types:
9 1) u - color vertex u
10 2) v - print minimum distance of vertex v to any colored vertex before
11
12
13 Time Complexity: O(n log n)
14 Space Complexity: O(n log n)
15 */
16 const int MAXN=200005;
17
```

```

18 vector<int> adj [MAXN];
19 vector<bool> is_removed(MAXN, false);
20 vector<int> subtree_size(MAXN, 0);
21 vector<int> dis(MAXN, 1e9);
22 vector<vector<pair<int, int>>> ancestor(MAXN);
23
24 int get_subtree_size(int node, int parent = -1) {
25     subtree_size[node] = 1;
26     for (int child : adj[node]) {
27         if (child == parent || is_removed[child]) { continue; }
28         subtree_size[node] += get_subtree_size(child, node);
29     }
30     return subtree_size[node];
31 }
32
33 int get_centroid(int node, int tree_size, int parent = -1) {
34     for (int child : adj[node]) {
35         if (child == parent || is_removed[child]) { continue; }
36         if (subtree_size[child] * 2 > tree_size) {
37             return get_centroid(child, tree_size, node);
38         }
39     }
40     return node;
41 }
42
43 void getDist(int cur, int centroid, int p=-1, int dist=1){
44     for (int child:adj[cur]){
45         if(child==p || is_removed[child])
46             continue;
47         dist++;
48         getDist(child, centroid, cur, dist);
49         dist--;
50     }
51     ancestor[cur].push_back(make_pair(centroid, dist));
52 }
53
54 void update(int cur){
55     for (int i=0;i<ancestor[cur].size();i++){
56         dis[ancestor[cur][i].first]=min(dis[ancestor[cur][i].first],
57                                         ancestor[cur][i].second);
58     }
59     dis[cur]=0;
60 }

```

```
60
61 int query(int cur){
62     int mini=dis[cur];
63     for (int i=0;i<ancestor[cur].size();i++){
64         mini=min(mini, ancestor[cur][i].second+dis[ancestor[cur][i].first
65             ]);
66     }
67     return mini;
68 }
69
70 void build_centroid_decomp(int node = 1) {
71     int centroid = get_centroid(node, get_subtree_size(node));
72
73     for (int child : adj[centroid]) {
74         if (is_removed[child]) { continue; }
75         getDist(child, centroid, centroid);
76     }
77
78     is_removed[centroid] = true;
79
80     for (int child : adj[centroid]) {
81         if (is_removed[child]) { continue; }
82         build_centroid_decomp(child);
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  
```

```

14 //call dfs1 first
15 struct SegmentTree {
16     vector<ll> a;
17     int n;
18
19     SegmentTree(int _n) : a(2 * _n, 0), n(_n) {}
20
21     void update(int pos, ll val) {
22         for (a[pos += n] = val; pos > 1; pos >>= 1) {
23             a[pos / 2] = (a[pos] ^ a[pos ^ 1]);
24         }
25     }
26
27     ll get(int l, int r) {
28         ll res = 0;
29         for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
30             if (l & 1) {
31                 res ^= a[l++];
32             }
33             if (r & 1) {
34                 res ^= a[--r];
35             }
36         }
37         return res;
38     }
39 };
40
41 const int MAXN=500005;
42 vector<int> adj[MAXN];
43 SegmentTree st(MAXN);
44 int a[MAXN], sz[MAXN], to[MAXN], dpth[MAXN], s[MAXN], par[MAXN];
45 int cnt=0;
46
47 void dfs1(int cur, int p){
48     sz[cur]=1;
49     for(int x:adj[cur]){
50         if(x==p) continue;
51         dpth[x]=dpth[cur]+1;
52         par[x]=cur;
53         dfs1(x, cur);
54         sz[cur]+=sz[x];
55     }
56 }
57 }
58
59 void dfs(int cur, int p, int l){
60     st.update(cnt, a[cur]);
61     s[cur]=cnt++;
62     to[cur]=l;
63     int g=-1;
64     for(int x:adj[cur]){
65         if(x==p) continue;
66         if(g==-1 || sz[g]<sz[x]){
67             g=x;
68         }
69     }
70     if(g==-1) return;
71     dfs(g, cur, l);
72     for(int x:adj[cur]){
73         if(x==p || x==g) continue;
74         dfs(x, cur, x);
75     }
76 }
77
78 int query(int u, int v){
79     int res=0;
80     while(to[u]!=to[v]){
81         if(dpth[to[u]]<dpth[to[v]]) swap(u, v);
82         res^=st.get(s[to[u]], s[u]+1);
83         u=par[to[u]];
84     }
85     if(dpth[u]>dpth[v]) swap(u, v);
86     res^=st.get(s[u], s[v]+1);
87     return res;
88 }
89
90
91
92 //alternate implementation
93 vector<int> parent, depth, heavy, head, pos;
94 int cur_pos;
95
96
97 int dfs(int v, vector<vector<int>> const& adj) {
98     int size = 1;
99     int max_c_size = 0;

```

```

100    for (int c : adj[v]) {
101        if (c != parent[v]) {
102            parent[c] = v, depth[c] = depth[v] + 1;
103            int c_size = dfs(c, adj);
104            size += c_size;
105            if (c_size > max_c_size)
106                max_c_size = c_size, heavy[v] = c;
107        }
108    }
109    return size;
110 }
111
112 void decompose(int v, int h, vector<vector<int>> const& adj) {
113     head[v] = h, pos[v] = cur_pos++;
114     if (heavy[v] != -1)
115         decompose(heavy[v], h, adj);
116     for (int c : adj[v]) {
117         if (c != parent[v] && c != heavy[v])
118             decompose(c, c, adj);
119     }
120 }
121
122 void init(vector<vector<int>> const& adj) {
123     int n = adj.size();
124     parent = vector<int>(n);
125     depth = vector<int>(n);
126     heavy = vector<int>(n, -1);
127     head = vector<int>(n);
128     pos = vector<int>(n);
129     cur_pos = 0;
130
131     dfs(0, adj);
132     decompose(0, 0, adj);
133 }
134
135 int query(int a, int b) {
136     int res = 0;
137     for (; head[a] != head[b]; b = parent[head[b]]) {
138         if (depth[head[a]] > depth[head[b]])
139             swap(a, b);
140         int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
141         res = max(res, cur_heavy_path_max);

```

```

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

```

15.3 Lowest Common Ancestor (LCA)

```

1  /*
2      LCA(Lowest Common Ancestor)
3
4      Computes the lowest common ancestor of two vertices in a tree.
5
6      Be careful as implementation is indexed starting with 1
7
8      Time Complexity(Creation): O(n log n)
9      Time Complexity(Query): O(log n)
10     Space Complexity: O(n log n)
11 */
12
13 const int N=200005;
14 vector<int> adj[N];
15 vector<int> start(N), end1(N), depth(N);
16 vector<vector<int>> t(N, vi(32));
17 int timer=0;
18 int n, l;
19 // l=(int)ceil(log2(n))
20 // call dfs(1, 1, 0)
21 // 1 indexed, dont use 0 indexing
22
23
24 void dfs(int cur, int p, int cnt){
25     depth[cur]=cnt;
26     t[cur][0]=p;
27     start[cur]=timer++;
28     for(int i=1;i<=l;i++){
29         t[cur][i]=t[t[cur][i-1]][i-1];
30     }
31     for(int x:adj[cur]){
32         if(x==p) continue;
33         dfs(x, cur, cnt+1);

```

```

34     }
35     end1[cur] = ++timer;
36 }
37
38 bool ancestor(int u, int v){
39     return start[u] <= start[v] && end1[u] >= end1[v];
40 }
41
42 int lca(int u, int v){
43     if(ancestor(u, v))
44         return u;
45     if (ancestor(v, u)){
46         return v;
47     }
48     for(int i=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,
```

```

17     int previous = 0, int length = 0) {
18     pair<int, int> max_path = {node, length};
19     for (const int &i : tree[node]) {
20         if (i == previous) { continue; }
21         pair<int, int> other = dfs(tree, i, node, length + 1);
22         if (other.second > max_path.second) { max_path = other; }
23     }
24     return max_path;
25 }
```

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.
      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
```