

ICPC MX 2025 Reference (Date 3 version)

Falsificamos el INE de Wicho

Last Updated: August 22, 2025

Table of Contents

1	C++ Functions	4	4.7	BFS Iterative Flood Fill	25
1.1	Common STL Algorithms.	4	4.8	DFS Flood Fill.	25
2	Binary search in the answer	5	4.9	Lava Flow (Multi-source BFS)	25
3	Data Structures	6	4.10	Dijkstra	26
3.1	Fenwick Tree	6	4.11	Bellman Ford (With path restoring)	26
3.2	Fenwick Minimum	6	4.12	SPFA Bellman Ford	27
3.3	1-Indexed Fenwick Tree	7	4.13	Floyd-Warshall.	27
3.4	Fenwick 2D (Sum query)	7	4.14	Prim's Algorithm (MST)	28
3.5	Fenwick 2D (Counting in range)	7	4.15	Kruskal's Algorithm (MST)	28
3.6	Fenwick Tree Range Update - Point Query	8	4.16	Another Kruskal	28
3.7	Fenwick Tree - Range update and query	8	4.17	Kosaraju Algorithm (SCC)	28
3.8	Segment Tree (Iterative)	8	4.18	SCC	29
3.9	Segment Tree (Sum query)	9	4.19	Tarjan algorithm (SCC)	29
3.10	Segment Tree (Minimum query)	9	4.20	Finding Articulation Points	30
3.11	Segment Tree Lazy Propagation	10	4.21	Finding bridges	30
3.12	Segment Tree 2D	11	4.22	Finding Bridges Online.	31
3.13	Segment tree with Index Compression	12	4.23	Bridge Tree	32
3.14	Segment Tree Prefix-Suffix-Biggest	13	4.24	2-SAT	33
3.15	Persistent Array	13	4.25	Hierholzer's Algorithm (Eulerian Path)	34
3.16	Path Copying - Persistent Array	14	4.26	Gale-Shapley Algorithm (Stable marriage).	34
3.17	Persistent Segment Tree	14	5	Trees	35
3.18	Policy Ordered Set	15	5.1	Succesor	35
3.19	Disjoint Set Union	15	5.2	Euler Tour	35
3.20	DSU to detect cycles	16	5.3	Lowest Common Ancestor.	36
3.21	DSU to check online bipartitness	16	5.4	Binary Lifting	37
3.22	DSU with rollback	17	5.5	Cartesian Tree	37
3.23	Dynamic connectivity	17	5.6	Heavy-Light Decomposition	37
3.24	Trie.	19	5.7	Centroid Decomposition	38
3.25	Palindromic Tree	20	5.8	Tree Distances	38
3.26	Implicit Treap	21	6	Flows	39
3.27	Treap	22	6.1	Ford-Fulkerson Maximum Flow.	39
4	Graph Theory	22	6.2	Dinic's Max Flow	39
4.1	Bipartite Check BFS	22	6.3	Min-cost Flow	40
4.2	Cycle Detection DFS	23	6.4	Hungarian Algorithm	41
4.3	Topological Sort	23	6.5	Kuhn's Algorithm	41
4.4	Kahn's Algorithm.	23			
4.5	Lexicographically Min. TopoSort	24			
4.6	BFS Flood Fill.	24			

7	Dynamic Programming.	42	10	Linear Algebra.	52
7.1	Coin Problem (Count ways)	42	10.1	Determinant of a Matrix	52
7.2	Coin Problem (Count sorted ways)	42	10.2	Rank of a Matrix	52
7.3	Coin Problem (Minimum).	42	10.3	Gauss-Jordan	52
7.4	Counting paths	42	10.4	Matrix Exponentiation	53
7.5	Longest Increasing Subsequence	43	11	Geometry	53
7.6	Length of LIS	43	11.1	Line Segment Intersection.	53
7.7	Longest Common Subsequence	43	11.2	Minimum Euclidian Distance	54
7.8	Edit Distance	43	11.3	Point in polygon	55
7.9	Bitmask DP.	44	11.4	Point Location Test	56
7.10	Digit DP	44	11.5	Polygon Area	56
7.11	Double DP	45	11.6	Convex Hull.	56
8	Math	45	11.7	Complex point	57
8.1	Miller Rabin	45	11.8	Polar sort	58
8.2	Sieve of Erathostenes	45	12	Strings	58
8.3	Sieve of Eratosthenes (count primes).	45	12.1	Marranadas de Quique	58
8.4	Segmented Sieve	46	12.2	KMP Algorithm	59
8.5	Linear sieve	46	12.3	Rolling Hash	59
8.6	Sum of divisors	46	12.4	Hash marrano	60
8.7	Finding the divisors of a number (Trial Division)	46	12.5	Suffix Array	60
8.8	Finding the divisors of a number (Fermat).	47	12.6	LCP	60
8.9	Factorials.	47	12.7	Z Function	62
8.10	Binpow	47	12.8	Longest Palindrome	62
8.11	Modulo Inverse	47	12.9	String Hashing.	63
8.12	BinPow Modulo Inv	47	12.10	Manacher Algorithm.	64
8.13	Binomial Coefficients	47	12.11	Suffix Automaton.	64
8.14	Newton Method (Sqrt and iSqrt)	48	13	Formulas	65
8.15	Integration with Simpson Method	48	13.1	Sums	65
8.16	Ternary Search.	48	13.2	Catalan numbers	65
8.17	DP Pascal triangle 1D	48	13.3	Cayley's Formula	65
8.18	DP Pascal triangle 2D	48	13.4	Geometric series	65
8.19	Euler's Totient.	49	13.5	Divisors	66
8.20	Diophantine equations	49	13.6	Number of primes between 1 and n	66
8.21	Discrete Log.	49	13.7	Pythagorean triplets.	66
9	Polynomials.	50	13.8	Derangments	66
9.1	FFT	50	14	Miscellaneous	66
9.2	NTT	51	14.1	Random number generator	66
9.3	Berlekamp Messey	51	14.2	Kadane's Algorithm	66
			14.3	Moore's Voting Algorithm.	66

15	Marranadas de C++	67
15.1	Compilation	67
15.2	Compiler optimizations	67
15.3	Decimal printing	67
15.4	Bit tricks	67

1 C++ Functions

1.1 Common STL Algorithms

Sorting Algorithms

Function	Parameters	Description
sort	begin, end, [comp]	Standard unstable sort ($O(n \log n)$)
stable_sort	begin, end, [comp]	Stable sort preserves element order
is_sorted	begin, end, [comp]	Checks if range is sorted (returns bool)
nth_element	begin, nth, end, [comp]	Partitions around nth element

Searching Functions

Function	Parameters	Description
lower_bound	begin, end, val, [comp]	First element \leq value
upper_bound	begin, end, val, [comp]	First element $>$ value
binary_search	begin, end, val, [comp]	Existence check in sorted range
find	begin, end, val	Linear search for value
find_if	begin, end, pred	Find first matching predicate

Sequence Operations

Function	Parameters	Description
reverse	begin, end	Reverse elements in-place
rotate	begin, mid, end	Rotate elements left
next_permutation	begin, end	Generate next permutation
unique	begin, end, [pred]	Remove consecutive duplicates
remove	begin, end, val	Remove elements equal to value

Numerical Functions

Function	Parameters	Description
accumulate	begin, end, init, [op]	Sum/accumulate elements
partial_sum	begin, end, dest, [op]	Compute prefix sums
_gcd	a, b	Greatest common divisor (C++17)
lcm	a, b	Least common multiple (C++17)
iota	begin, end, val	Fill with consecutive values

Memory/Array Operations

Function	Parameters	Description
memset	ptr, value, count	Fill memory with byte value
fill	begin, end, value	Fill range with value
fill_n	begin, count, value	Fill N elements with value
copy	src_b, src_e, dest	Copy range to destination
copy_if	src_b, src_e, dest, pred	Copy elements matching predicate

Utility Functions

Function	Parameters	Description
swap	a, b	Swap two values
max_element	begin, end, [comp]	Find maximum element
min_element	begin, end, [comp]	Find minimum element
count	begin, end, val	Count element occurrences
all_of	begin, end, pred	Check all elements satisfy condition

Mathematical / Bitwise Builtins

Function	Parameters	Description
__builtin_popcount	x (int)	Count number of 1-bits
__builtin_popcountll	x (long long)	Count number of 1-bits (64-bit)
__builtin_clz	x (unsigned int)	Count leading zeros
__builtin_clzll	x (unsigned long long)	Leading zeros (64-bit)
__builtin_ctz	x (unsigned int)	Count trailing zeros
__builtin_ctzll	x (unsigned long long)	Trailing zeros (64-bit)
__builtin_parity	x	Return 1 if #bits is odd
__builtin_ffs	x	Position of least significant 1-bit (1-indexed)
_lg	x	Floor of $\log_2(x)$ (index of highest bit)

Priority Queues and Heaps

Function	Parameters	Description
priority_queue	[type], [container], [comp]	Max-heap by default
make_heap	begin, end, [comp]	Turn range into heap
push_heap	begin, end, [comp]	Push element into heap
pop_heap	begin, end, [comp]	Pop max element into end
sort_heap	begin, end, [comp]	Heap sort

Set / Map Utilities

Operation	Usage	Description
s.lower_bound(x)	set/map	First element \geq x
s.upper_bound(x)	set/map	First element $>$ x
s.equal_range(x)	multiset/map	Pair of lower/upper bound
s.erase(it)	iterator	Erase element at iterator
s.find(x)	key	Iterator to key or end

String Functions

Function	Parameters	Description
stoi, stol, stoll	string, [pos], [base]	Convert string \rightarrow int/long/ll
stoul, stoull	string, [pos], [base]	Convert string \rightarrow unsigned
stod, stof, stold	string	Convert string \rightarrow double/float/-long double
to_string	value	Convert number \rightarrow string
substr	pos, len	Substring
find	str, pos	Find first occurrence
rfind	str, pos	Find last occurrence

Random Number Utilities

Type / Function	Usage	Description
mt19937 rng	chrono::steady_clock::now()	Fast random generator
uniform_int_distribution	dist(a,b)(rng)	Random int in [a,b]
shuffle	begin, end, rng	Random shuffle

Other Useful Utilities

Function	Parameters	Description
chrono::high_resolution_clock	now()	Get precise current time
_int128	value	128-bit integer (manual I/O needed)
bitset<N>	ops: &, —, ^, , &&	Fixed-size bitset manipulation
tuple	get< i >(t)	Store and access heterogeneous data
pair	first, second	Store pair of values

2 Binary search in the answer

```
1 // Standard binary search (iterative)
2 int binary_search(vector<int>& arr, int target) {
3     int left = 0, right = arr.size() - 1;
4     while (left <= right) {
5         int mid = left + (right - left) / 2;
6         if (arr[mid] == target) return mid;
7         if (arr[mid] < target) left = mid + 1;
8         else right = mid - 1;
9     }
10    return -1;
11 }
12
13 // Lower bound (first element >= target)
14 int lower_bound(vector<int>& arr, int target) {
15     int left = 0, right = arr.size();
16     while (left < right) {
17         int mid = left + (right - left) / 2;
18         arr[mid] < target ? left = mid + 1
19                           : right = mid;
20     }
21    return left;
22 }
23
24 // Upper bound (first element > target)
25 int upper_bound(vector<int>& arr, int target) {
26     int left = 0, right = arr.size();
27     while (left < right) {
28         int mid = left + (right - left) / 2;
29         arr[mid] <= target ? left = mid + 1
30                           : right = mid;
31     }
32    return left;
```

```

33 }
34
35 // Binary search on real numbers (e.g. sqrt)
36 double sqrt_precision(double n, double eps=1e-6) {
37     double left = 0, right = n;
38     for (int i = 0; i < 100; ++i) { // or while (right-left >
39         eps)
40         double mid = (left + right) / 2;
41         if (mid*mid < n) left = mid;
42         else right = mid;
43     }
44     return left;
45 }
46
47 // Binary search on answer space (monotonic condition)
48 int find_min_valid(vector<int>& nums, int k) {
49     auto is_valid = [&](int x) {
50         /* condition check */
51     };
52     int left = 0, right = 1e9; // adjust bounds
53     while (left < right) {
54         int mid = left + (right - left) / 2;
55         is_valid(mid) ? right = mid
56                     : left = mid + 1;
57     }
58     return left;
59 }z

```

3 Data Structures

3.1 Fenwick Tree

```

1 struct FenwickTree {
2     vector<int> bit; // binary indexed tree
3     int n;
4
5     FenwickTree(int n) {
6         this->n = n;
7         bit.assign(n, 0);
8     }
9
10    FenwickTree(vector<int> const &a) : FenwickTree(a.size()){
11        for (int i = 0; i < n; i++) {
12            bit[i] += a[i];
13            int r = i | (i + 1);
14            if (r < n) bit[r] += bit[i];
15        }

```

```

16 }
17
18 FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
19     for (size_t i = 0; i < a.size(); i++)
20         add(i, a[i]);
21 }
22
23 int sum(int r) {
24     int ret = 0;
25     for (; r >= 0; r = (r & (r + 1)) - 1)
26         ret += bit[r];
27     return ret;
28 }
29
30 int sum(int l, int r) {
31     return sum(r) - sum(l - 1);
32 }
33
34 void add(int idx, int delta) {
35     for (; idx < n; idx = idx | (idx + 1))
36         bit[idx] += delta;
37 }
38 };

```

3.2 Fenwick Minimum

```

1 struct FenwickTreeMin {
2     vector<int> bit;
3     int n;
4     const int INF = (int)1e9;
5
6     FenwickTreeMin(int n) {
7         this->n = n;
8         bit.assign(n, INF);
9     }
10
11    FenwickTreeMin(vector<int> a) : FenwickTreeMin(a.size()) {
12        for (size_t i = 0; i < a.size(); i++)
13            update(i, a[i]);
14    }
15
16    int getmin(int r) {
17        int ret = INF;
18        for (; r >= 0; r = (r & (r + 1)) - 1)
19            ret = min(ret, bit[r]);
20        return ret;
21    }
22
23    void update(int idx, int val) {

```

```

24         for (; idx < n; idx = idx | (idx + 1))
25             bit[idx] = min(bit[idx], val);
26     }
27 };

```

3.3 1-Indexed Fenwick Tree

```

1 struct FenwickTreeOneBasedIndexing {
2     vector<int> bit; // binary indexed tree
3     int n;
4
5     FenwickTreeOneBasedIndexing(int n) {
6         this->n = n + 1;
7         bit.assign(n + 1, 0);
8     }
9
10    FenwickTreeOneBasedIndexing(vector<int> a)
11        : FenwickTreeOneBasedIndexing(a.size()) {
12        for (size_t i = 0; i < a.size(); i++)
13            add(i, a[i]);
14    }
15
16    int sum(int idx) {
17        int ret = 0;
18        for (++idx; idx > 0; idx -= idx & -idx)
19            ret += bit[idx];
20        return ret;
21    }
22
23    int sum(int l, int r) {
24        return sum(r) - sum(l - 1);
25    }
26
27    void add(int idx, int delta) {
28        for (++idx; idx < n; idx += idx & -idx)
29            bit[idx] += delta;
30    }
31 };

```

3.4 Fenwick 2D (Sum query)

```

1 struct Fenwick2D {
2     vector<vector<int>> tree;
3     int rows, cols;
4
5     Fenwick2D(int r, int c) : rows(r), cols(c),
6         tree(r + 1, vector<int>(c + 1)) {}
7

```

```

8     // Update: add delta to (x, y) (1-based)
9     void update(int x, int y, int delta) {
10         for(int i = x; i <= rows; i += lsb(i))
11             for(int j = y; j <= cols; j += lsb(j))
12                 tree[i][j] += delta;
13     }
14
15     // Query sum from (1,1) to (x,y)
16     int query(int x, int y) {
17         int sum = 0;
18         for(int i = x; i > 0; i -= lsb(i))
19             for(int j = y; j > 0; j -= lsb(j))
20                 sum += tree[i][j];
21         return sum;
22     }
23
24     // Range sum from (x1,y1) to (x2,y2)
25     int range_query(int x1, int y1, int x2, int y2) {
26         return query(x2, y2) - query(x1-1, y2)
27             - query(x2, y1-1) + query(x1-1, y1-1);
28     }
29
30     int lsb(int i) { return i & -i; }
31 };

```

3.5 Fenwick 2D (Counting in range)

```

1 struct Fenwick2DPerType {
2     int rows, cols;
3     unordered_map<int, Fenwick2D> trees; // Map from type to 2D
4                                         // Fenwick Tree
5
6     Fenwick2DPerType(int r, int c) : rows(r), cols(c) {}
7
8     // Update: add 'delta' objects of type 't' at position (x, y)
9     void update(int t, int x, int y, int delta) {
10         if (trees.find(t) == trees.end()) {
11             trees[t] = Fenwick2D(rows, cols);
12         }
13         trees[t].update(x, y, delta);
14     }
15
16     // Query: count of type 't' in rectangle [x1,y1] to [x2,y2]
17     int query(int t, int x1, int y1, int x2, int y2) {
18         if (trees.find(t) == trees.end()) return 0;
19         return trees[t].range_query(x1, y1, x2, y2);
20     }
21 };

```



```

22 // Requires the base Fenwick2D implementation from previous answer
23 struct Fenwick2D {
24     vector<vector<int>> tree;
25     int rows, cols;
26
27     Fenwick2D(int r, int c) : rows(r), cols(c),
28         tree(r + 1, vector<int>(c + 1)) {}
29
30     void update(int x, int y, int delta) { /* same as before */ }
31
32     int query(int x, int y) { /* same as before */ }
33
34     int range_query(int x1, int y1, int x2, int y2) { /* same as
35         before */ }
36
37     int lsb(int i) { return i & -i; }
38 };

```

3.6 Fenwick Tree Range Update - Point Query

```

1 // Range Update - Point Query (1-based indexing)
2 struct FenwickRUQ {
3     int n;
4     std::vector<int> bit;
5
6     FenwickRUQ(int size) : n(size + 1), bit(size + 2) {}
7
8     // Add val to range [l, r] (1-based)
9     void range_add(int l, int r, int val) {
10         add(l, val);
11         add(r + 1, -val);
12     }
13
14     // Get value at position idx (1-based)
15     int point_query(int idx) {
16         int res = 0;
17         for(; idx > 0; idx -= idx & -idx)
18             res += bit[idx];
19         return res;
20     }
21
22 private:
23     void add(int idx, int val) {
24         for(; idx < n; idx += idx & -idx)
25             bit[idx] += val;
26     }
27 };

```

3.7 Fenwick Tree - Range update and query

```

1 // Range Update - Range Query (1-based indexing)
2 struct FenwickRURQ {
3     int n;
4     std::vector<int> B1, B2;
5
6     FenwickRURQ(int size) : n(size + 1), B1(size + 2), B2(size +
7         2) {}
8
9     // Add val to range [l, r] (1-based)
10    void range_add(int l, int r, int val) {
11        add(B1, l, val);
12        add(B1, r + 1, -val);
13        add(B2, l, val * (l - 1));
14        add(B2, r + 1, -val * r);
15    }
16
17    // Get sum of range [l, r] (1-based)
18    int range_sum(int l, int r) {
19        return prefix_sum(r) - prefix_sum(l - 1);
20    }
21
22 private:
23     void add(std::vector<int>& b, int idx, int val) {
24         for(; idx < n; idx += idx & -idx)
25             b[idx] += val;
26     }
27
28     int sum(const std::vector<int>& b, int idx) {
29         int total = 0;
30         for(; idx > 0; idx -= idx & -idx)
31             total += b[idx];
32         return total;
33     }
34
35     int prefix_sum(int idx) {
36         return sum(B1, idx) * idx - sum(B2, idx);
37     }
38 };

```

3.8 Segment Tree (Iterative)

```

1 int segtree[2*100000 + 5];
2
3 void build(vector<int> &arr, int n){
4     for(int i=0; i<n; i++)
5         segtree[n+i] = arr[i];
6

```

```

7     for(int i=n-1; i>=1; i--)
8         segtree[i] = max(segtree[2*i], segtree[2*i+1]);
9
10
11 void update(int pos, int value, int n){
12     pos+=n;
13     segtree[pos] = value;
14
15     while(pos>1){
16         pos>>=1;
17         segtree[pos] = max(segtree[2*pos], segtree[2*pos+1]);
18     }
19 }
20
21 int query(int l, int r, int n){
22     l+=n;
23     r+=n;
24
25     int mx = INT_MIN;
26
27     while(l <= r){
28         if(l % 2 == 1) mx = max(mx, segtree[l++]);
29         if(r % 2 == 0) mx = max(mx, segtree[r--]);
30         l >>= 1;
31         r >>= 1;
32     }
33
34     return mx;
35 }

```

3.9 Segment Tree (Sum query)

```

1 ll t[4*MAX];
2
3 // Shout-out to CP algo for the SegTree implementation: https://cp
4 // -algorithms.com/data_structures/segment_tree.html#memory-
5 // efficient-implementation
6
7 void buildSegTree(vector<ll> &a, int v, int tl, int tr) {
8     if (tl == tr) {
9         t[v] = a[tl];
10    } else {
11        int tm = (tl + tr) / 2;
12        buildSegTree(a, v*2, tl, tm);
13        buildSegTree(a, v*2+1, tm+1, tr);
14        t[v] = t[v*2] + t[v*2+1];
15    }
16 }

```

```

16
17 ll sum(int v, int tl, int tr, int l, int r) {
18     if (l > r)
19         return 0;
20     if (l == tl && r == tr) {
21         return t[v];
22     }
23     int tm = (tl + tr) / 2;
24     return sum(v*2, tl, tm, l, min(r, tm))
25         + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
26 }
27
28 void update(int v, int tl, int tr, int pos, ll new_val) {
29     if (tl == tr) {
30         t[v] = new_val;
31     } else {
32         int tm = (tl + tr) / 2;
33         if (pos <= tm)
34             update(v*2, tl, tm, pos, new_val);
35         else
36             update(v*2+1, tm+1, tr, pos, new_val);
37         t[v] = t[v*2] + t[v*2+1];
38     }
39 }

```

3.10 Segment Tree (Minimum query)

```

1 ll t[4*MAX];
2
3 // Shout-out to CP algo for the SegTree implementation: https://cp
4 // -algorithms.com/data_structures/segment_tree.html#memory-
5 // efficient-implementation
6
7 void buildSegTree(vector<ll> &a, int v, int tl, int tr) {
8     if (tl == tr) {
9         t[v] = a[tl];
10    } else {
11        int tm = (tl + tr) / 2;
12        buildSegTree(a, v*2, tl, tm);
13        buildSegTree(a, v*2+1, tm+1, tr);
14        t[v] = min(t[v*2], t[v*2+1]); // Change to minimum
15    }
16 }
17
18 ll query(int v, int tl, int tr, int l, int r) {
19     if (l > r)
20         return LLONG_MAX; // Return maximum possible value for
21         empty range

```

```

20     if (l == tl && r == tr) {
21         return t[v];
22     }
23     int tm = (tl + tr) / 2;
24     return min(query(v*2, tl, tm, l, min(r, tm)),
25               query(v*2+1, tm+1, tr, max(l, tm+1), r));
26 }
27
28 void update(int v, int tl, int tr, int pos, ll new_val) {
29     if (tl == tr) {
30         t[v] = new_val;
31     } else {
32         int tm = (tl + tr) / 2;
33         if (pos <= tm)
34             update(v*2, tl, tm, pos, new_val);
35         else
36             update(v*2+1, tm+1, tr, pos, new_val);
37         t[v] = min(t[v*2], t[v*2+1]); // Change to minimum
38     }
39 }

```

3.11 Segment Tree Lazy Propagation

```

1  typedef long long ll;
2  typedef vector<int> vec;
3  typedef vector<pair<int,int>> vpai;
4  const ll mod=1e9+7;
5  const int MAX=1e5+3;
6  const int limit=2e5+3;
7  const int TAM=2e5+1;
8  ll t[4*TAM];
9  ll op[4*TAM];
10 int type[4*TAM];
11 //ascii https://elcodigoascii.com.ar/
12
13 void propagate(int root,int l,int r)
14 {
15     if(type[root]==1)
16     {
17         t[root]+=op[root]*(r+1-l);
18         if(l!=r){
19             op[2*root]+=op[root];
20             op[2*root+1]+=op[root];
21             type[2*root+1]=max(1,type[2*root+1]);
22             type[2*root]=max(1,type[2*root]);
23         }
24     }
25     else
26     {

```

```

27         if(type[root]==2){
28             t[root]=op[root]*(r+1-l);
29             if(l!=r){
30                 op[2*root]=op[root];
31                 op[2*root+1]=op[root];
32                 type[2*root+1]=2;
33                 type[2*root]=2;
34             }
35         }
36     }
37     op[root]=0;
38     type[root]=0;
39 }
40
41 void build(int root,int l,int r,vector<ll> &arr)
42 {
43     if(l==r)
44     {
45         t[root]=arr[l];
46         op[root]=0;
47         type[root]=0;
48         return;
49     }
50     int mid=(l+r)/2;
51     build(2*root,l,mid,arr);
52     build(2*root+1,mid+1,r,arr);
53     t[root]=t[2*root]+t[2*root+1];
54     op[root]=0;
55     type[root]=0;
56 }
57
58 void sum(int root,int l,int r,int a,int b,ll val)
59 {
60     propagate(root,l,r);
61     if(a>b) return;
62     if(l==a && r==b)
63     {
64         op[root]=val;
65         type[root]=1;
66         propagate(root,l,r);
67         return;
68     }
69     int mid=(l+r)/2;
70     sum(2*root,l,mid,a,min(b,mid),val);
71     sum(2*root+1,mid+1,r,max(mid+1,a),b,val);
72     t[root]=t[2*root]+t[2*root+1];
73 }
74
75 void setR(int root,int l,int r,int a,int b,ll val)
76 {
77     propagate(root,l,r);

```

```

78     if(a>b) return;
79     if(l==a && r==b)
80     {
81         op[root]=val;
82         type[root]=2;
83         propagate(root,l,r);
84         return;
85     }
86     int mid=(l+r)/2;
87     setR(2*root,l,mid,a,min(b,mid),val);
88     setR(2*root+1,mid+1,r,max(mid+1,a),b,val);
89     t[root]=t[2*root]+t[2*root+1];
90 }
91
92 ll consult(int root,int l,int r, int a,int b)
93 {
94     propagate(root,l,r);
95     if(a>b) return 0;
96     if(l==a && r==b){
97         return t[root];
98     }
99     int mid=(l+r)/2;
100    return consult(2*root,l,mid,a,min(b,mid))+
101    consult(2*root+1,mid+1,r,max(mid+1,a),b);
102 }

```

3.12 Segment Tree 2D

```

1  typedef long long ll;
2  typedef vector<int> vec;
3  const ll mod=1e9+7;
4  const int TAM=1e3+1;
5  //ascii https://elcodigoascii.com.ar/
6  vector<vector<int>> forest(TAM,vector<int> (TAM));
7  ll t[4*TAM][4*TAM];
8  int n;
9
10 void buildNode(int root,int l,int r,int node,vector<int> &arr){
11     if(l==r)
12     {
13         t[node][root]=arr[l];
14         return;
15     }
16     int mid=(l+r)/2;
17     buildNode(2*root,l,mid,node,arr);
18     buildNode(2*root+1,mid+1,r,node,arr);
19     t[node][root]=t[node][2*root]+t[node][2*root+1];
20 }
21

```

```

22 void build(int root,int l,int r,vector<vector<int>> &arr)
23 {
24     if(l==r)
25     {
26         buildNode(1,0,n-1,root,arr[l]);
27         return;
28     }
29     int mid=(l+r)/2;
30     build(2*root,l,mid,arr);
31     build(2*root+1,mid+1,r,arr);
32     FO(i,4*TAM) t[root][i]=t[2*root][i]+t[2*root+1][i];
33 }
34
35 void updateNode(int root,int l,int r,int y,int node,int val)
36 {
37     if(l==r)
38     {
39         t[node][root]=val;
40         return;
41     }
42     int mid=(l+r)/2;
43     if(y>mid)
44     {
45         updateNode(2*root+1,mid+1,r,y,node,val);
46     }
47     else{
48         updateNode(2*root,l,mid,y,node,val);
49     }
50     t[node][root]=t[node][2*root]+t[node][2*root+1];
51 }
52
53 void update(int root,int l,int r,int x,int y,int val)
54 {
55     if(l==r)
56     {
57         updateNode(1,0,n-1,y,root,val);
58         return;
59     }
60     int mid=(l+r)/2;
61     if(x>mid)
62     {
63         update(2*root+1,mid+1,r,x,y,val);
64     }
65     else{
66         update(2*root,l,mid,x,y,val);
67     }
68     int i=0,j=n-1,Ndt=1,mid_aux;
69     while(i!=j)
70     {
71         mid_aux=(i+j)/2;
72

```

```

73     t[root][Ndt]=t[2*root][Ndt]+t[2*root+1][Ndt];
74     if(y>mid_aux){
75         i=mid_aux+1;
76         Ndt=2*Ndt+1;
77     }
78     else{
79         j=mid_aux;
80         Ndt*=2;
81     }
82 }
83 t[root][Ndt]=t[2*root][Ndt]+t[2*root+1][Ndt];
84 }
85
86 ll consultNode(int root,int l,int r,int node,int y1,int y2)
87 {
88     if(y1>y2) return 0;
89     if(l==y1 && r==y2) return t[node][root];
90     int mid=(l+r)/2;
91     return consultNode(2*root,l,mid,node,y1,min(y2,mid))+
92     consultNode(2*root+1,mid+1,r,node,max(mid+1,y1),y2);
93 }
94
95 ll consult(int root,int l,int r, int x1,int x2,int y1,int y2)
96 {
97     if(x1>x2) return 0;
98     if(l==x1 && r==x2) return consultNode(1,0,n-1,root,y1,y2);
99     int mid=(l+r)/2;
100    return consult(2*root,l,mid,x1,min(x2,mid),y1,y2)+
101    consult(2*root+1,mid+1,r,max(mid+1,x1),x2,y1,y2);
102 }

```

3.13 Segment tree with Index Compression

```

1  typedef long long ll;
2  typedef vector<int> vec;
3  typedef vector<pair<int,int>> vpii;
4  const ll mod=1e9+7;
5  const int MAX=4e5+3;
6  const int limit=2e5+3;
7  const int TAM=2e5+1;
8  ll t[4*MAX];
9  //ascii https://elcodigoascii.com.ar/
10
11
12 void update(int root,int l,int r,int pos,int val)
13 {
14     if(l==r)
15     {
16         t[root]+=val;

```

```

17         return;
18     }
19     int mid=(l+r)/2;
20     if(pos>mid)
21     {
22         update(2*root+1,mid+1,r,pos,val);
23     }
24     else{
25         update(2*root,l,mid,pos,val);
26     }
27     t[root]=t[2*root]+t[2*root+1];
28 }
29
30 ll consult(int root,int l,int r, int a,int b)
31 {
32     if(a>b) return 0;
33     if(l==a && r==b) return t[root];
34     int mid=(l+r)/2;
35     return consult(2*root,l,mid,a,min(b,mid))+
36     consult(2*root+1,mid+1,r,max(mid+1,a),b);
37 }
38
39 inline void solve()
40 {
41     int n,m,index;
42     cin>>n>>m;
43     vector<ll> arr(n);
44     vector<tuple<char,ll,ll>> queries(m);
45     set<ll> salary;
46     memset(t,0,sizeof(t));
47     FO(i,n){
48         ll aux; cin>>aux;
49         arr[i]=aux;
50         salary.insert(aux);
51     }
52     FO(i,m)
53     {
54         char a;
55         ll b,c;
56         cin>>a>>b>>c;
57         queries[i]=make_tuple(a,b,c);
58         if(a=='!') salary.insert(c);
59     }
60
61     vector<ll> coord(all(salary));
62     int tn=coord.size();
63     //FO(i,tn) cout<<coord[i]<<" ";
64     //cout<<endl;
65     FO(i,n)
66     {
67         index=lower_bound(all(coord),arr[i])-coord.begin();

```

```

68     update(1,0,tn-1,index,1);
69 }
70 F0(i,m)
71 {
72     char a=get<0>(queries[i]);
73     ll b=get<1>(queries[i]);
74     ll c=get<2>(queries[i]);
75     if(a=='?'){
76         b=lower_bound(all(coord),b)-coord.begin();
77         c=(upper_bound(all(coord),c)-coord.begin())-1;
78         if(b==tn || c==tn ){
79             cout<<0<<endl;
80         }
81         else cout<<consult(1,0,tn-1,b,c)<<endl;
82     }
83     else{
84         index=lower_bound(all(coord),arr[b-1])-coord.begin();
85         update(1,0,tn-1,index,-1);
86         arr[b-1]=c;
87         index=lower_bound(all(coord),arr[b-1])-coord.begin();
88         update(1,0,tn-1,index,1);
89     }
90 }
91 }
92 }
93 }

```

3.14 Segment Tree Prefix-Suffix-Biggest

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 typedef vector<pair<int,int>> vpii;
4 const ll mod=1e9+7;
5 const int MAX=1e5+3;
6 const int limit=2e5+3;
7 const int TAM=2e5+1;
8 ll t[4*TAM];
9 ll prefix[4*TAM],suffix[4*TAM],biggest[4*TAM];
10 //ascii https://elcodigoascii.com.ar/
11 ll cero=0;
12 void build(int root,int l,int r,vector<ll> &arr)
13 {
14     if(l==r)
15     {
16         t[root]=arr[l];
17         suffix[root]=max(t[root],cero);
18         prefix[root]=max(t[root],cero);
19         biggest[root]=max(t[root],cero);
20         return;

```

```

21     }
22     int mid=(l+r)/2;
23     build(2*root,l,mid,arr);
24     build(2*root+1,mid+1,r,arr);
25     t[root]=t[2*root]+t[2*root+1];
26     biggest[root]=max(biggest[2*root],
27         max(biggest[2*root+1],suffix[2*root]+prefix[2*root+1]));
28     prefix[root]=max(prefix[2*root],t[2*root]+prefix[2*root+1]);
29     suffix[root]=max(suffix[2*root+1],t[2*root+1]+suffix[2*root]);
30 }
31 }
32 void update(int root,int l,int r,int pos,ll val)
33 {
34     if(l==r)
35     {
36         t[root]=val;
37         suffix[root]=max(cero,t[root]);
38         prefix[root]=max(cero,t[root]);
39         biggest[root]=max(t[root],cero);
40         return;
41     }
42     int mid=(l+r)/2;
43     if(pos>mid)
44     {
45         update(2*root+1,mid+1,r,pos,val);
46     }
47     else{
48         update(2*root,l,mid,pos,val);
49     }
50     t[root]=t[2*root]+t[2*root+1];
51     biggest[root]=max(biggest[2*root],
52         max(biggest[2*root+1],suffix[2*root]+prefix[2*root+1]));
53     prefix[root]=max(prefix[2*root],t[2*root]+prefix[2*root+1]);
54     suffix[root]=max(suffix[2*root+1],t[2*root+1]+suffix[2*root]);
55 }
56 }
57 ll consult(int root,int l,int r, int a,int b)
58 {
59     if(a>b) return 0;
60     if(l==a && r==b) return t[root];
61     int mid=(l+r)/2;
62     return consult(2*root,l,mid,a,min(b,mid))+
63         consult(2*root+1,mid+1,r,max(mid+1,a),b);
64 }
65 }

```

3.15 Persistent Array

```

1 vector<pair<int, int>> arr[100001]; // The persistent array

```

```

2
3 int get_item(int index, int time) {
4     // Gets the array item at a given index and time
5     auto ub =
6         upper_bound(arr[index].begin(), arr[index].end(),
7             make_pair(time, INT_MAX));
8     return prev(ub)->second;
9 }
10 void update_item(int index, int value, int time) {
11     // Updates the array item at a given index and time
12     // Note that this only works if the time is later than all
13     // previous
14     // update times
15     assert(arr[index].back().first < time);
16     arr[index].push_back({time, value});
17 }
18 void init_arr(int n, int *init) {
19     // Initializes the persistent array, given an input array
20     for (int i = 0; i < n; i++) arr[i].push_back({0, init[i]});
21 }

```

3.16 Path Copying - Persistent Array

```

1     struct Node {
2         int val;
3         Node *l, *r;
4
5         Node(ll x) : val(x), l(nullptr), r(nullptr) {}
6         Node(Node *ll, Node *rr) : val(0), l(ll), r(rr) {}
7     };
8
9     int n, a[100001];    // The initial array and its size
10    Node *roots[100001]; // The persistent array's roots
11
12    Node *build(int l = 0, int r = n - 1) {
13        if (l == r) return new Node(a[l]);
14        int mid = (l + r) / 2;
15        return new Node(build(l, mid), build(mid + 1, r));
16    }
17
18    Node *update(Node *node, int val, int pos, int l = 0, int r = n -
19        1) {
20        if (l == r) return new Node(val);
21        int mid = (l + r) / 2;
22        if (pos > mid) return new Node(node->l, update(node->r, val,
23            pos, mid + 1, r));

```

```

22         else return new Node(update(node->l, val, pos, l, mid), node->
23             r);
24     }
25     int query(Node *node, int pos, int l = 0, int r = n - 1) {
26         if (l == r) return node->val;
27         int mid = (l + r) / 2;
28         if (pos > mid) return query(node->r, pos, mid + 1, r);
29         return query(node->l, pos, l, mid);
30     }
31
32     int get_item(int index, int time) {
33         // Gets the array item at a given index and time
34         return query(roots[time], index);
35     }
36
37     void update_item(int index, int value, int prev_time, int
38         curr_time) {
39         // Updates the array item at a given index and time
40         roots[curr_time] = update(roots[prev_time], index, value);
41     }
42
43     void init_arr(int nn, int *init) {
44         // Initializes the persistent array, given an input array
45         n = nn;
46         for (int i = 0; i < n; i++) a[i] = init[i];
47         roots[0] = build();

```

3.17 Persistent Segment Tree

```

1 using ll = long long;
2
3 class PersistentSegtree {
4     private:
5         struct Node {
6             ll sum = 0;
7             int l = 0, r = 0;
8         };
9
10        const int n;
11        vector<Node> tree;
12        int timer = 1;
13
14        Node join(int l, int r) { return Node{tree[l].sum + tree[r].
15            sum, l, r}; }
16
17        int build(int tl, int tr, const vector<int> &arr) {
18            if (tl == tr) {

```

```

18         tree[timer] = {arr[tl], 0, 0};
19         return timer++;
20     }
21
22     int mid = (tl + tr) / 2;
23     tree[timer] = join(build(tl, mid, arr), build(mid + 1, tr,
24                         arr));
25
26     return timer++;
27 }
28
29 int set(int v, int pos, int val, int tl, int tr) {
30     if (tl == tr) {
31         tree[timer] = {val, 0, 0};
32         return timer++;
33     }
34
35     int mid = (tl + tr) / 2;
36     if (pos <= mid) {
37         tree[timer] = join(set(tree[v].l, pos, val, tl, mid),
38                             tree[v].r);
39     } else {
40         tree[timer] = join(tree[v].l, set(tree[v].r, pos, val,
41                                         mid + 1, tr));
42     }
43
44     return timer++;
45 }
46
47 ll range_sum(int v, int ql, int qr, int tl, int tr) {
48     if (qr < tl || tr < ql) { return 0ll; }
49     if (ql <= tl && tr <= qr) { return tree[v].sum; }
50
51     int mid = (tl + tr) / 2;
52     return range_sum(tree[v].l, ql, qr, tl, mid) +
53            range_sum(tree[v].r, ql, qr, mid + 1, tr);
54 }
55
56 public:
57 PersistentSegtree(int n, int MX_NODES) : n(n), tree(MX_NODES)
58 {}
59
60 int build(const vector<int> &arr) { return build(0, n - 1, arr
61 ); }
62
63 int set(int root, int pos, int val) { return set(root, pos,
64 val, 0, n - 1); }
65
66 ll range_sum(int root, int l, int r) { return range_sum(root,
67 l, r, 0, n - 1); }

```

```

62     int add_copy(int root) {
63         tree[timer] = tree[root];
64         return timer++;
65     }
66 };

```

3.18 Policy Ordered Set

```

1 #include <ext/pb_ds/assoc_container.hpp> // Common file
2 #include <ext/pb_ds/tree_policy.hpp>
3 #include <functional> // for less
4 using namespace __gnu_pbds;
5
6 // To allow repetitions
7 typedef tree<int, null_type, less<int>, rb_tree_tag,
8             tree_order_statistics_node_update>
9     ordered_set;
10
11 // To not allow repetitions
12 typedef tree<pair<int, int>, null_type,
13             less<pair<int, int>>, rb_tree_tag,
14             tree_order_statistics_node_update>
15     ordered_multiset;
16
17 ordered_set pt; // Definition
18
19 pt.order_of_key(x); // Number of items strictly smaller than x
20 pt.find_by_order(k); // Iterator to the kth element

```

3.19 Disjoint Set Union

```

1 // Shout-out to Usaco Guide for DSU implementation: https://usaco.guide/gold/dsu?lang=cpp
2
3 class DisjointSets{
4     private:
5         vector<int> parents;
6         vector<int> sizes;
7         int components;
8     public:
9         DisjointSets(int size) : parents(size), sizes(size,1),
10                                components(size){
11             for(int i=0; i<size; i++){parents[i] = i;}
12         }
13
14         int find(int x) {return parents[x] == x ? x : (parents[x]
15 = find(parents[x]));}

```



```

15     bool unite(int x, int y){
16         int x_root = find(x);
17         int y_root = find(y);
18
19         if(x_root == y_root) {return false;}
20
21         if(sizes[x_root] < sizes[y_root]) {swap(x_root,y_root)}
22         ;}
23         sizes[x_root] += sizes[y_root];
24         parents[y_root] = x_root;
25         components--;
26         return true;
27     }
28
29     vector<int> getAllComponentSizes(){
30         map<int, int> component_sizes;
31         for (int i = 0; i < parents.size(); ++i){
32             int root = find(i);
33             if (component_sizes.find(root) == component_sizes.
34                 end()){
35                 component_sizes[root] = sizes[root];
36             }
37         }
38
39         vector<int> result;
40         for (auto& [root, size] : component_sizes) {
41             result.push_back(size);
42         }
43
44         return result;
45     }
46
47     bool connected(int x, int y) { return find(x) == find(y);}
48     int getSize(int x) {return sizes[find(x)];}
49     int getComponents() const {return components;}
50 };

```

3.20 DSU to detect cycles

```

1 class CycleDetectionDSU {
2     vector<int> parent;
3     vector<int> size;
4
5 public:
6     CycleDetectionDSU(int n) : parent(n), size(n, 1) {
7         iota(parent.begin(), parent.end(), 0);
8     }
9

```

```

10     int find(int x) {
11         return parent[x] == x ? x : parent[x] = find(parent[x]);
12     }
13
14     // Returns true if adding edge u-v creates a cycle
15     bool add_edge(int u, int v) {
16         int u_root = find(u);
17         int v_root = find(v);
18         if (u_root == v_root) return true;
19
20         if (size[u_root] < size[v_root]) swap(u_root, v_root);
21         parent[v_root] = u_root;
22         size[u_root] += size[v_root];
23         return false;
24     }
25 };

```

3.21 DSU to check online bipartiteness

```

1 class BipartiteDSU {
2     vector<int> parent;
3     vector<int> size;
4
5 public:
6     BipartiteDSU(int n) : parent(2*n), size(2*n, 1) {
7         iota(parent.begin(), parent.end(), 0);
8     }
9
10    int find(int x) {
11        return parent[x] == x ? x : parent[x] = find(parent[x]);
12    }
13
14    // Returns true if graph remains bipartite after adding u-v
15    bool add_edge(int u, int v) {
16        int u_orig = 2*u;           // Original node
17        int u_mirror = 2*u+1;       // Mirror node
18        int v_orig = 2*v;
19        int v_mirror = 2*v+1;
20
21        // Union u_orig <-> v_mirror and v_orig <-> u_mirror
22        for(int i = 0; i < 2; i++) {
23            int x = i ? v_orig : u_orig;
24            int y = i ? u_mirror : v_mirror;
25
26            int x_root = find(x);
27            int y_root = find(y);
28            if (x_root != y_root) {
29                if (size[x_root] < size[y_root]) swap(x_root,
30                    y_root);

```

```

30         parent[y_root] = x_root;
31         size[x_root] += size[y_root];
32     }
33 }
34
35 // Check if u is in both partitions
36 return find(u_orig) != find(u_mirror);
37 }
38 };
39
40 // -- Other implementation --
41
42 void make_set(int v) {
43     parent[v] = make_pair(v, 0);
44     rank[v] = 0;
45     bipartite[v] = true;
46 }
47
48 pair<int, int> find_set(int v) {
49     if (v != parent[v].first) {
50         int parity = parent[v].second;
51         parent[v] = find_set(parent[v].first);
52         parent[v].second ^= parity;
53     }
54     return parent[v];
55 }
56
57 void add_edge(int a, int b) {
58     pair<int, int> pa = find_set(a);
59     a = pa.first;
60     int x = pa.second;
61
62     pair<int, int> pb = find_set(b);
63     b = pb.first;
64     int y = pb.second;
65
66     if (a == b) {
67         if (x == y)
68             bipartite[a] = false;
69     } else {
70         if (rank[a] < rank[b])
71             swap(a, b);
72         parent[b] = make_pair(a, x^y^1);
73         bipartite[a] ^= bipartite[b];
74         if (rank[a] == rank[b])
75             ++rank[a];
76     }
77 }
78
79 bool is_bipartite(int v) {

```

```

81     return bipartite[find_set(v).first];
82 }

```

3.22 DSU with rollback

```

1 class DSU {
2     private:
3         vector<int> p, sz;
4         // stores previous unites
5         vector<pair<int &, int>> history;
6
7     public:
8         DSU(int n) : p(n), sz(n, 1) { iota(p.begin(), p.end(), 0); }
9
10        int get(int x) { return x == p[x] ? x : get(p[x]); }
11
12        void unite(int a, int b) {
13            a = get(a);
14            b = get(b);
15            if (a == b) { return; }
16            if (sz[a] < sz[b]) { swap(a, b); }
17
18            // save this unite operation
19            history.push_back({sz[a], sz[a]});
20            history.push_back({p[b], p[b]});
21
22            p[b] = a;
23            sz[a] += sz[b];
24        }
25
26        int snapshot() { return history.size(); }
27
28        void rollback(int until) {
29            while (snapshot() > until) {
30                history.back().first = history.back().second;
31                history.pop_back();
32            }
33        }
34 };

```

3.23 Dynamic connectivity

```

1 struct dsu_save {
2     int v, rnkv, u, rnku;
3
4     dsu_save() {}
5
6     dsu_save(int _v, int _rnkv, int _u, int _rnku)

```

```

7         : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
8     };
9
10 struct dsu_with_rollbacks {
11     vector<int> p, rnk;
12     int comps;
13     stack<dsu_save> op;
14
15     dsu_with_rollbacks() {}
16
17     dsu_with_rollbacks(int n) {
18         p.resize(n);
19         rnk.resize(n);
20         for (int i = 0; i < n; i++) {
21             p[i] = i;
22             rnk[i] = 0;
23         }
24         comps = n;
25     }
26
27     int find_set(int v) {
28         return (v == p[v]) ? v : find_set(p[v]);
29     }
30
31     bool unite(int v, int u) {
32         v = find_set(v);
33         u = find_set(u);
34         if (v == u)
35             return false;
36         comps--;
37         if (rnk[v] > rnk[u])
38             swap(v, u);
39         op.push(dsu_save(v, rnk[v], u, rnk[u]));
40         p[v] = u;
41         if (rnk[u] == rnk[v])
42             rnk[u]++;
43         return true;
44     }
45
46     void rollback() {
47         if (op.empty())
48             return;
49         dsu_save x = op.top();
50         op.pop();
51         comps++;
52         p[x.v] = x.v;
53         rnk[x.v] = x.rnk;
54         p[x.u] = x.u;
55         rnk[x.u] = x.rnk;
56     }
57 };

```

```

58
59 struct query {
60     int v, u;
61     bool united;
62     query(int _v, int _u) : v(_v), u(_u) {}
63 };
64
65
66 struct QueryTree {
67     vector<vector<query>> t;
68     dsu_with_rollbacks dsu;
69     int T;
70
71     QueryTree() {}
72
73     QueryTree(int _T, int n) : T(_T) {
74         dsu = dsu_with_rollbacks(n);
75         t.resize(4 * T + 4);
76     }
77
78     void add_to_tree(int v, int l, int r, int ul, int ur, query& q)
79     {
80         if (ul > ur)
81             return;
82         if (l == ul && r == ur) {
83             t[v].push_back(q);
84             return;
85         }
86         int mid = (l + r) / 2;
87         add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
88         add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
89     }
90
91     void add_query(query q, int l, int r) {
92         add_to_tree(1, 0, T - 1, l, r, q);
93     }
94
95     void dfs(int v, int l, int r, vector<int>& ans) {
96         for (query& q : t[v]) {
97             q.united = dsu.unite(q.v, q.u);
98         }
99         if (l == r)
100             ans[l] = dsu.comps;
101         else {
102             int mid = (l + r) / 2;
103             dfs(2 * v, l, mid, ans);
104             dfs(2 * v + 1, mid + 1, r, ans);
105         }
106         for (query q : t[v]) {
107             if (q.united)

```

```

107         dsu.rollback();
108     }
109 }
110
111 vector<int> solve() {
112     vector<int> ans(T);
113     dfs(1, 0, T - 1, ans);
114     return ans;
115 }
116 }

```

3.24 Trie

```

1 class TrieNode
2 {
3     public:
4         // Array for children nodes of each node
5         TrieNode *children[26];
6
7         // for end of word
8         bool isLeaf;
9
10        TrieNode()
11        {
12            isLeaf = false;
13            for (int i = 0; i < 26; i++)
14            {
15                children[i] = nullptr;
16            }
17        }
18    };
19
20    // Method to insert a key into the Trie
21    void insert(TrieNode *root, const string &key)
22    {
23
24        // Initialize the curr pointer with the root node
25        TrieNode *curr = root;
26
27        // Iterate across the length of the string
28        for (char c : key)
29        {
30
31            // Check if the node exists for the
32            // current character in the Trie
33            if (curr->children[c - 'a'] == nullptr)
34            {
35
36                // If node for current character does

```

```

37            // not exist then make a new node
38            TrieNode *newNode = new TrieNode();
39
40            // Keep the reference for the newly
41            // created node
42            curr->children[c - 'a'] = newNode;
43        }
44
45        // Move the curr pointer to the
46        // newly created node
47        curr = curr->children[c - 'a'];
48    }
49
50    // Mark the end of the word
51    curr->isLeaf = true;
52 }
53
54 // Method to search a key in the Trie
55 bool search(TrieNode *root, const string &key)
56 {
57
58     if (root == nullptr)
59     {
60         return false;
61     }
62
63     // Initialize the curr pointer with the root node
64     TrieNode *curr = root;
65
66     // Iterate across the length of the string
67     for (char c : key)
68     {
69
70         // Check if the node exists for the
71         // current character in the Trie
72         if (curr->children[c - 'a'] == nullptr)
73             return false;
74
75         // Move the curr pointer to the
76         // already existing node for the
77         // current character
78         curr = curr->children[c - 'a'];
79     }
80
81     // Return true if the word exists
82     // and is marked as ending
83     return curr->isLeaf;
84 }
85
86 // Method to check if a prefix exists in the Trie
87 bool isPrefix(TrieNode *root, const string &prefix)

```

```

88 {
89     // Initialize the curr pointer with the root node
90     TrieNode *curr = root;
91
92     // Iterate across the length of the prefix string
93     for (char c : prefix)
94     {
95         // Check if the node exists for the current character in
96         // the Trie
97         if (curr->children[c - 'a'] == nullptr)
98             return false;
99
100        // Move the curr pointer to the already existing node
101        // for the current character
102        curr = curr->children[c - 'a'];
103    }
104
105    // If we reach here, the prefix exists in the Trie
106    return true;
107 }

```

3.25 Palindromic Tree

```

1  const int MAXN = 105000;
2
3  struct node {
4      int next[26];
5      int len;
6      int sufflink;
7      int num;
8  };
9
10 int len;
11 char s[MAXN];
12 node tree[MAXN];
13 int num;          // node 1 - root with len -1, node 2 - root
14                   // with len 0
15 int suff;         // max suffix palindrome
16 long long ans;
17
18 bool addLetter(int pos) {
19     int cur = suff, curlen = 0;
20     int let = s[pos] - 'a';
21
22     while (true) {
23         curlen = tree[cur].len;
24         if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos])
25             break;

```

```

25         cur = tree[cur].sufflink;
26     }
27     if (tree[cur].next[let]) {
28         suff = tree[cur].next[let];
29         return false;
30     }
31
32     num++;
33     suff = num;
34     tree[num].len = tree[cur].len + 2;
35     tree[cur].next[let] = num;
36
37     if (tree[num].len == 1) {
38         tree[num].sufflink = 2;
39         tree[num].num = 1;
40         return true;
41     }
42
43     while (true) {
44         cur = tree[cur].sufflink;
45         curlen = tree[cur].len;
46         if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
47             tree[num].sufflink = tree[cur].next[let];
48             break;
49         }
50     }
51
52     tree[num].num = 1 + tree[tree[num].sufflink].num;
53
54     return true;
55 }
56
57 void initTree() {
58     num = 2; suff = 2;
59     tree[1].len = -1; tree[1].sufflink = 1;
60     tree[2].len = 0; tree[2].sufflink = 1;
61 }
62
63 // -- Other implementation --
64
65 const int maxn = 1e5, sigma = 26;
66
67 int s[maxn], len[maxn], link[maxn], to[maxn][sigma];
68
69 int n, last, sz;
70
71 void init()
72 {
73     s[n++] = -1;
74     link[0] = 1;

```

```

75     len[1] = -1;
76     sz = 2;
77 }
78
79 int get_link(int v)
80 {
81     while(s[n - len[v] - 2] != s[n - 1]) v = link[v];
82     return v;
83 }
84
85 void add_letter(int c)
86 {
87     s[n++] = c;
88     last = get_link(last);
89     if(!to[last][c])
90     {
91         len[sz] = len[last] + 2;
92         link[sz] = to[get_link(link[last])][c];
93         to[last][c] = sz++;
94     }
95     last = to[last][c];
96 }

```

3.26 Implicit Treap

```

1 using namespace std;
2
3 #include<random>
4 #include<chrono>
5
6 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count()
7 );
8 #define ll long long
9
10 struct TreapNode{
11     ll key, pr, sz;
12     TreapNode *l, *r;
13 };
14
15 typedef TreapNode* Treap;
16
17 int getSize(Treap &t){
18     return t ? t->sz : 0;
19 }
20
21 void updateSize(Treap &t){
22     if (t) t->sz = 1 + getSize(t->l) + getSize(t->r);
23 }

```

```

24
25 void split(Treap& t, ll k, Treap &l, Treap &r){
26     if(not t) l = r = nullptr;
27
28     else if(k < t->key){
29         split(t->l, k, l, t->l);
30         r = t;
31         updateSize(r);
32     }else{
33         split(t->r, k, t->r, r);
34         l = t;
35         updateSize(l);
36     }
37 }
38
39 void insert(Treap& t, Treap a){
40     if(not t) t=a;
41     else if(a->pr > t->pr){
42         split(t, a->key, a->l, a->r);
43         t = a;
44     }else{
45         if(a->key < t-> key) insert(t->l, a);
46         else insert(t->r, a);
47     }
48     updateSize(t);
49 }
50
51 void merge(Treap &t, Treap l, Treap r){
52     if(not l) t = r;
53     else if(not r) t = l;
54
55     else if(l->pr > r->pr){
56         merge(l->r, l->r, r);
57         t=l;
58         updateSize(t);
59     }else{
60         merge(r->l, l, r->l);
61         t=r;
62         updateSize(t);
63     }
64 }
65
66 void erase(Treap &t, ll k){
67     if(not t) return;
68     if(t->key == k) merge(t, t->l, t->r);
69
70     else{
71         if(k<t->key) erase(t->l, k);
72         else erase(t->r, k);
73     }
74     updateSize(t);

```

```

75 }
76
77 bool find(Treap& t, ll k){
78     if (not t) return false;
79     if(t->key == k) return true;
80     if(k<t->key) return find(t->l,k);
81     return find(t->r,k);
82 }
83
84 void insertValue(Treap &t, ll k){
85     if(not find(t,k)){
86         Treap new_node = new TreapNode {k,rng(), 0,nullptr,
87             nullptr};
88         insert(t, new_node);
89     }
90 }
91 ll getKth(Treap &t, int k){
92     if(!t || k<=0 || k>getSize(t)) return 0;
93     int leftSize = getSize(t->l);
94     if(k == leftSize+1) return t->key;
95     if(k <= leftSize) return getKth(t->l,k);
96     return getKth(t->r, k-leftSize-1);
97 }

```

3.27 Treap

```

1 typedef struct item * pitem;
2 struct item {
3     int prior, value, cnt;
4     bool rev;
5     pitem l, r;
6 };
7
8 int cnt (pitem it) {
9     return it ? it->cnt : 0;
10 }
11
12 void upd_cnt (pitem it) {
13     if (it)
14         it->cnt = cnt(it->l) + cnt(it->r) + 1;
15 }
16
17 void push (pitem it) {
18     if (it && it->rev) {
19         it->rev = false;
20         swap (it->l, it->r);
21         if (it->l) it->l->rev ^= true;
22         if (it->r) it->r->rev ^= true;

```

```

23     }
24 }
25
26 void merge (pitem &t, pitem l, pitem r) {
27     push (l);
28     push (r);
29     if (!l || !r)
30         t = l ? l : r;
31     else if (l->prior > r->prior)
32         merge (l->r, l->r, r), t = l;
33     else
34         merge (r->l, l, r->l), t = r;
35     upd_cnt (t);
36 }
37
38 void split (pitem t, pitem &l, pitem &r, int key, int add = 0) {
39     if (!t)
40         return void( l = r = 0 );
41     push (t);
42     int cur_key = add + cnt(t->l);
43     if (key <= cur_key)
44         split (t->l, l, t->l, key, add), r = t;
45     else
46         split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
47     upd_cnt (t);
48 }
49
50 void reverse (pitem t, int l, int r) {
51     pitem t1, t2, t3;
52     split (t, t1, t2, l);
53     split (t2, t2, t3, r-l+1);
54     t2->rev ^= true;
55     merge (t, t1, t2);
56     merge (t, t, t3);
57 }
58
59 void output (pitem t) {
60     if (!t) return;
61     push (t);
62     output (t->l);
63     printf ("%d ", t->value);
64     output (t->r);
65 }

```

4 Graph Theory

4.1 Bipartite Check BFS

```

1 bool bfs(int s){
2     queue<int> q;
3     q.push(s);
4     color[s] = 1; // Assign the initial color
5
6     while(!q.empty()){
7         int u = q.front();
8         q.pop();
9
10        // Check all adjacent vertices of u
11        for(auto v : adj[u]){
12            // If v is not colored yet
13            if(color[v] == 0){
14                color[v] = (color[u] == 1) ? 2 : 1;
15                q.push(v);
16            }
17            else if (color[v] == color[u]){
18                return false;
19            }
20        }
21    }
22    return true;
23 }

```

4.2 Cycle Detection DFS

```

1 // Thanks CP- Algo for Cycle finding implementation: https://cp-
  algorithms.com/graph/finding-cycle.html
2
3 bool dfs(int v, int par) { // passing vertex and its parent vertex
4     visited[v] = true;
5     for (int u : adj[v]) {
6         if(u == par) continue; // skipping edge to parent vertex
7         if (visited[u]) {
8             cycle_end = v;
9             cycle_start = u;
10            return true;
11        }
12        parent[u] = v;
13        if (dfs(u, parent[u]))
14            return true;
15    }
16    return false;
17 }
18
19 void find_cycle() {
20     visited.assign(n+1, false);
21     parent.assign(n+1, -1);
22     cycle_start = -1;

```

```

23
24     for (int v = 0; v < n; v++) {
25         if (!visited[v] && dfs(v, parent[v]))
26             break;
27     }
28
29     if (cycle_start == -1) {
30         cout << "IMPOSSIBLE" << endl;
31     } else {
32         vector<int> cycle;
33         cycle.push_back(cycle_start);
34         for (int v = cycle_end; v != cycle_start; v = parent[v])
35             cycle.push_back(v);
36         cycle.push_back(cycle_start);
37
38         cout << cycle.size() << endl;
39         for (int v : cycle)
40             cout << v << " ";
41         cout << endl;
42     }
43 }

```

4.3 Topological Sort

```

1 vector<int> ans;
2
3 void dfs(int v) {
4     visited[v] = true;
5     for (int u : adj[v]) {
6         if (!visited[u])
7             dfs(u);
8     }
9     ans.push_back(v);
10 }
11
12 void topological_sort() {
13     visited.assign(n+1, false);
14     ans.clear();
15     for (int i = 1; i <= n; ++i) {
16         if (!visited[i]) {
17             dfs(i);
18         }
19     }
20     reverse(ans.begin(), ans.end());
21 }

```

4.4 Kahn's Algorithm


```

1 def kahnTopoSort(self, adj: List[List[int]]) -> List[int]:
2     #print(adj)
3     in_deg = [0] * len(adj)
4     for i in range(len(adj)):
5         for u in adj[i]:
6             in_deg[u] += 1
7
8     q = []
9     for i in range(len(in_deg)):
10        if in_deg[i] == 0:
11            q.append(i)
12
13    arns = []
14    while len(q) > 0:
15        u = q[0]
16        q.pop(0)
17        arns.append(u)
18
19        for v in adj[u]:
20            in_deg[v] -= 1
21            if in_deg[v] == 0:
22                q.append(v)
23
24    print(str(len(arns)) + " " + str(len(adj)))
25    if len(arns) != len(adj):
26        return []
27
28    return arns

```

4.5 Lexicographically Min. TopoSort

```

1 int n;
2 vector<vector<int>> adj(MAX);
3 vector<int> in_degree(MAX);
4 vector<int> group_ids(MAX);
5 vector<int> ans;
6
7 //topological sort implementation: https://cp-algorithms.com/graph
8 //topological-sort.html
9
10 void topological_sort() {
11     priority_queue<pair<int, int>, vector<pair<int, int>>, greater
12     <pair<int, int>>> pq;
13
14     for(int i = 1; i <= n; i++) {
15         if(in_degree[i] == 0) {
16             pq.emplace(group_ids[i], i);
17         }
18     }
19 }

```

```

17 while(!pq.empty()) {
18     int u = pq.top().second;
19     pq.pop();
20     ans.push_back(u);
21
22     for(int v : adj[u]) {
23         in_degree[v]--;
24         if(in_degree[v] == 0) {
25             pq.emplace(group_ids[v], v);
26         }
27     }
28 }
29
30 }
31

```

4.6 BFS Flood Fill

```

1 bool validate(int x, int y){
2     if(vis[x][y]) return false;
3     if(maze[x][y] == '#') return false;
4     if(x<0 or x>=n or y<0 or y>=m) return false;
5     return true;
6 }
7
8 bool solveMaze(int x, int y){
9     queue<pii> q;
10    q.push(mp(x,y));
11    vis[x][y] = true;
12
13    int dx[] = {1, -1, 0, 0};
14    int dy[] = {0, 0, 1, -1};
15    char move_dir[] = {'D', 'U', 'R', 'L'};
16
17    while(!q.empty()){
18        int u = q.front().fs;
19        int v = q.front().sc;
20        q.pop();
21
22        if(maze[u][v] == 'B'){
23            while(true){
24                res.push_back(path[u][v]);
25
26                if(res.back() == 'U' && u + 1 < n) u++;
27                if(res.back() == 'D' && u - 1 >= 0) u--;
28                if(res.back() == 'L' && v + 1 < m) v++;
29                if(res.back() == 'R' && v - 1 >= 0) v--;
30
31                if(u == x and v == y) break;

```

```

32     }
33     return true;
34 }
35 for (int i = 0; i < 4; ++i) {
36     int new_u = u + dx[i];
37     int new_v = v + dy[i];
38     if (validate(new_u, new_v)) {
39         path[new_u][new_v] = move_dir[i];
40         vis[new_u][new_v] = true;
41         q.push(mp(new_u, new_v));
42     }
43 }
44 }
45 return false;
46 }

```

4.7 BFS Iterative Flood Fill

```

1 void floodFill(int x, int y, char color, int r, int c) {
2     if (maze[x][y] == color) return;
3     queue<pii> q;
4     q.push(pii(x, y));
5     while (!q.empty()) {
6         pii currentCoor = q.front();
7         q.pop();
8         x = currentCoor.fi;
9         y = currentCoor.sc;
10        if (x >= 0 && x < r && y >= 0 && y < c && maze[x][y] !=
11            color) {
12            maze[x][y] = color;
13            q.push(pii(x + 1, y));
14            q.push(pii(x - 1, y));
15            q.push(pii(x, y + 1));
16            q.push(pii(x, y - 1));
17        }
18    }
19 }

```

4.8 DFS Flood Fill

```

1 void floodFill(int x, int y, char color, vector<vector<char>>&
2     board){
3     if(x<0 or y<0 or x>=board.size() or y>=board[x].size() or
4         board[x][y] != '0') return;
5     board[x][y] = color;
6     floodFill(x+1,y,color,board);
7     floodFill(x-1,y,color,board);
8     floodFill(x,y+1,color,board);
9     floodFill(x,y-1,color,board);

```

```

7     floodFill(x,y-1,color,board);
8 }

```

4.9 Lava Flow (Multi-source BFS)

```

1 struct Cell{
2     int x,y,t;
3 };
4
5 const int MAX = 1005;
6 int n,m;
7
8 char maze[MAX][MAX];
9 int vis[MAX][MAX];
10 int player[MAX][MAX];
11 char path[MAX][MAX];
12 set<pii> isExit;
13 queue<Cell> q;
14 string res;
15
16 bool isValid(int x, int y){
17     if(x < 0 || x >= n || y < 0 || y >= m) return false;
18     if(maze[x][y] == '#') return false;
19     return true;
20 }
21
22 bool isSafe(int x, int y, int u, int v){
23     return player[x][y] == -1 and maze[x][y] != 'M' and (vis[x][y]
24         == -1 or player[u][v] + 1 < vis[x][y]);
25 }
26
27 void restorePath(int u, int v, int x, int y){
28     while (x != u || y != v) {
29         res.push_back(path[u][v]);
30
31         if (res.back() == 'U') u++;
32         if (res.back() == 'D') u--;
33         if (res.back() == 'L') v++;
34         if (res.back() == 'R') v--;
35     }
36 }
37
38 bool lavaFlow(int x,int y){
39     q.push({x,y,1});
40     player[x][y] = 0;
41
42     while(!q.empty()){
43         int u = q.front().x;

```

```

44     int v = q.front().y;
45     int t = q.front().t;
46
47     q.pop();
48
49     vector<pii> dir = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
50
51     for(auto it: dir){
52         int i = u+it.fs;
53         int j = v+it.sc;
54
55         if(isValid(i,j)){
56             if(t == 0){
57                 if(vis[i][j] == -1){
58                     vis[i][j] = vis[u][v]+1;
59                     q.push(Cell{i,j,0});
60                 }
61             }else{
62                 if(isSafe(i,j,u,v)){
63                     path[i][j] = (it.fs == 1) ? 'D' : (it.
64                         fs == -1) ? 'U' : (it.sc == 1) ? '
65                         R' : 'L';
66                     player[i][j] = player[u][v]+1;
67                     q.push(Cell{i,j,1});
68                     if (isExit.find({i,j}) != isExit.end()
69                         ) {
70                         if (player[i][j] < vis[i][j] ||
71                             vis[i][j] == -1) {
72                             restorePath(i, j, x, y);
73                             return true;
74                         }
75                     }
76                 }
77             }
78         }
79     }
80     return false;

```

4.10 Dijkstra

```

1 typedef pair<ll, ll> pll;
2
3 vector<ll> dijkstra(int n, int source, vector<vector<pll>> &adj) {
4     vector<ll> dist(n, INF);
5     priority_queue<pll, vector<pll>, greater<pll>> pq;

```

```

6     dist[source] = 0;
7     pq.push({0, source});
8
9     while (!pq.empty()) {
10         ll d = pq.top().first;
11         ll u = pq.top().second;
12         pq.pop();
13
14         if (d > dist[u]) continue;
15
16         for (auto &edge : adj[u]) {
17             ll v = edge.first;
18             ll weight = edge.second;
19
20             if (dist[u] + weight < dist[v]) {
21                 dist[v] = dist[u] + weight;
22                 pq.push({dist[v], v});
23             }
24         }
25     }
26
27     return dist;
28 }

```

4.11 Bellman Ford (With path restoring)

```

1 struct Edge {
2     int src, dest, weight;
3 };
4
5 void bellmanFord(int V, int E, vector<Edge>& edges, int start) {
6     vector<int> dist(V+1, INT_MAX);
7     dist[start] = 0;
8
9     for (int i = 1; i < V; i++) {
10         for (int j = 0; j < E; j++) {
11             int u = edges[j].src;
12             int v = edges[j].dest;
13             int weight = edges[j].weight;
14             if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
15                 {
16                     dist[v] = dist[u] + weight;
17                 }
18         }
19     }
20
21     for (int j = 0; j < E; j++) {
22         int u = edges[j].src;
23         int v = edges[j].dest;

```

```

23     int weight = edges[j].weight;
24     if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
25         //cout << "Graph contains a negative weight cycle\n";
26         return;
27     }
28 }
29
30 for(int i=1; i<=V; i++){
31     if(dist[i]!=INT_MAX){
32         cout<<dist[i]<<" ";
33     }else{
34         cout<<"30000 ";
35     }
36 }
37 cout<<endl;
38 }
39
40 void solve()
41 {
42     vector<int> d(n, INF);
43     d[v] = 0;
44     vector<int> p(n, -1);
45
46     for (;;) {
47         bool any = false;
48         for (Edge e : edges)
49             if (d[e.a] < INF)
50                 if (d[e.b] > d[e.a] + e.cost) {
51                     d[e.b] = d[e.a] + e.cost;
52                     p[e.b] = e.a;
53                     any = true;
54                 }
55         if (!any)
56             break;
57     }
58
59     if (d[t] == INF)
60         cout << "No path from " << v << " to " << t << ".";
61     else {
62         vector<int> path;
63         for (int cur = t; cur != -1; cur = p[cur])
64             path.push_back(cur);
65         reverse(path.begin(), path.end());
66
67         cout << "Path from " << v << " to " << t << ": ";
68         for (int u : path)
69             cout << u << ' ';
70     }
71 }
72 }

```

4.12 SPFA Bellman Ford

```

1  const int INF = 1000000000;
2  vector<vector<pair<int, int>>> adj;
3
4  bool spfa(int s, vector<int>& d) {
5      int n = adj.size();
6      d.assign(n, INF);
7      vector<int> cnt(n, 0);
8      vector<bool> inqueue(n, false);
9      queue<int> q;
10
11      d[s] = 0;
12      q.push(s);
13      inqueue[s] = true;
14      while (!q.empty()) {
15          int v = q.front();
16          q.pop();
17          inqueue[v] = false;
18
19          for (auto edge : adj[v]) {
20              int to = edge.first;
21              int len = edge.second;
22
23              if (d[v] + len < d[to]) {
24                  d[to] = d[v] + len;
25                  if (!inqueue[to]) {
26                      q.push(to);
27                      inqueue[to] = true;
28                      cnt[to]++;
29                      if (cnt[to] > n)
30                          return false; // negative cycle
31                  }
32              }
33          }
34      }
35      return true;
36 }

```

4.13 Floyd-Warshall

```

1  void floydWarshall(vector<vector<ll>> &d, int n){
2      for (int k = 0; k < n; ++k) {
3          for (int i = 0; i < n; ++i) {
4              for (int j = 0; j < n; ++j) {
5                  d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
6              }
7          }
8      }

```

```
9 }
```

4.14 Prim's Algorithm (MST)

```
1 ll prim(int V, int E, vector<vector<pll>> &adj) {
2
3     priority_queue<pll, vector<pll>, greater<pll>> pq;
4
5     vector<bool> visited(V, false);
6
7     ll res = 0;
8
9     pq.push({0, 0});
10
11     while(!pq.empty()){
12         auto p = pq.top();
13         pq.pop();
14
15         int wt = p.first;
16         int u = p.second;
17
18         if(visited[u] == true){
19             continue;
20         }
21
22         res += wt;
23         visited[u] = true;
24
25         for(auto v : adj[u]){
26             if(visited[v.first] == false){
27                 pq.push({v.second, v.first});
28             }
29         }
30     }
31
32     for(int i=0; i<V; i++){
33         if(!visited[i])
34             return -1;
35     }
36
37     return res;
38 }
```

4.15 Kruskal's Algorithm (MST)

```
1 struct Edge { int u, v, weight; };
2
3 int kruskal(vector<Edge>& edges, int n) {
```

```
4     sort(edges.begin(), edges.end(),
5           [](Edge& a, Edge& b) { return a.weight < b.weight; });
6
7     DisjointSets dsu(n);
8     int total_weight = 0;
9
10    for (Edge& e : edges) {
11        if (!dsu.connected(e.u, e.v)) {
12            dsu.unite(e.u, e.v);
13            total_weight += e.weight;
14        }
15    }
16    return total_weight;
17 }
```

4.16 Another Kruskal

```
1 struct Edge {
2     int u, v, w;
3     bool operator<(Edge const& other) {
4         return w < other.w;
5     }
6 };
7
8 int kruskal(int n, vector<Edge> &edges, DisjointSets &dsu, vector<
9     Edge> &ans) {
10     int cost = 0;
11     sort(edges.begin(), edges.end());
12     for (Edge e : edges) {
13         if (ans.size() == n - 1) break;
14         if (dsu.unite(e.u, e.v)){
15             cost += e.w;
16             ans.push_back(e);
17         }
18     }
19     if(ans.size()!=n-1) return -1;
20     return cost;
21 }
```

4.17 Kosaraju Algorithm (SCC)

```
1 vector<bool> visited; // keeps track of which vertices are already
2     visited
3 // runs depth first search starting at vertex v.
4 // each visited vertex is appended to the output vector when dfs
5     leaves it.
```

```

5 void dfs(int v, vector<vector<int>> const& adj, vector<int> &
  output) {
6   visited[v] = true;
7   for (auto u : adj[v])
8     if (!visited[u])
9       dfs(u, adj, output);
10  output.push_back(v);
11 }
12
13 // input: adj -- adjacency list of G
14 // output: components -- the strongly connected components in G
15 // output: adj_cond -- adjacency list of G^SCC (by root vertices)
16 void strongly_connected_components(vector<vector<int>> const& adj,
17                                   vector<vector<int>> &components,
18                                   vector<vector<int>> &adj_cond) {
19
20   int n = adj.size();
21   components.clear(), adj_cond.clear();
22
23   vector<int> order; // will be a sorted list of G's vertices by
24                       // exit time
25
26   visited.assign(n, false);
27
28   // first series of depth first searches
29   for (int i = 0; i < n; i++)
30     if (!visited[i])
31       dfs(i, adj, order);
32
33   // create adjacency list of G^T
34   vector<vector<int>> adj_rev(n);
35   for (int v = 0; v < n; v++)
36     for (int u : adj[v])
37       adj_rev[u].push_back(v);
38
39   visited.assign(n, false);
40   reverse(order.begin(), order.end());
41
42   vector<int> roots(n, 0); // gives the root vertex of a vertex'
43                             // s SCC
44
45   // second series of depth first searches
46   for (auto v : order)
47     if (!visited[v]) {
48       std::vector<int> component;
49       dfs(v, adj_rev, component);
50       components.push_back(component);
51       int root = *min_element(begin(component), end(
52         component));
53       for (auto u : component)
54         roots[u] = root;
55     }

```

```

52
53 // add edges to condensation graph
54 adj_cond.assign(n, {});
55 for (int v = 0; v < n; v++)
56   for (auto u : adj[v])
57     if (roots[v] != roots[u])
58       adj_cond[roots[v]].push_back(roots[u]);
59 }

```

4.18 SCC

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 const ll mod=1e9+7;
4 const int MAX=1e5+3;
5 vector<vector<int>> g(MAX);
6 vector<vector<int>> r(MAX);
7 vector<int> id(MAX);
8 bool visitados[MAX]={false};
9 vector<int> l;
10
11 void dfs(int s){
12   visitados[s]=true;
13   for(int c:g[s]){
14     if(!visitados[c]) dfs(c);
15   }
16   l.push_back(s);
17 }
18
19 void rdfs(int s,int d)
20 {
21   visitados[s]=true;
22   id[s]=d;
23   for(int c:r[s])
24   {
25     if(!visitados[c]) rdfs(c,d);
26   }
27 }

```

4.19 Tarjan algorithm (SCC)

```

1 /** Takes in an adjacency list and calculates the SCCs of the
2     graph. */
3 class TarjanSolver {
4   private:
5     vector<vector<int>> rev_adj;
6     vector<int> post;
7     vector<int> comp;

```

```

7
8 vector<bool> visited;
9 int timer = 0;
10 int id = 0;
11
12 void fill_post(int at) {
13     visited[at] = true;
14     for (int n : rev_adj[at]) {
15         if (!visited[n]) { fill_post(n); }
16     }
17     post[at] = timer++;
18 }
19
20 void find_comp(int at) {
21     visited[at] = true;
22     comp[at] = id;
23     for (int n : adj[at]) {
24         if (!visited[n]) { find_comp(n); }
25     }
26 }
27
28 public:
29     const vector<vector<int>>> &adj;
30
31     TarjanSolver(const vector<vector<int>>> &adj)
32         : adj(adj), rev_adj(adj.size()), post(adj.size()), comp(
33             adj.size()),
34             visited(adj.size()) {
35         vector<int> nodes(adj.size());
36         for (int n = 0; n < adj.size(); n++) {
37             nodes[n] = n;
38             for (int next : adj[n]) { rev_adj[next].push_back(n); }
39         }
40
41         for (int n = 0; n < adj.size(); n++) {
42             if (!visited[n]) { fill_post(n); }
43         }
44         std::sort(nodes.begin(), nodes.end(),
45             [&](int n1, int n2) { return post[n1] > post[n2]; });
46
47         visited.assign(adj.size(), false);
48         for (int n : nodes) {
49             if (!visited[n]) {
50                 find_comp(n);
51                 id++;
52             }
53         }
54 }

```

```

55 int comp_num() const { return id; }
56
57 int get_comp(int n) const { return comp[n]; }
58 };

```

4.20 Finding Articulation Points

```

1 // adj[u] = adjacent nodes of u
2 // ap = AP = articulation points
3 // p = parent
4 // disc[u] = discovery time of u
5 // low[u] = 'low' node of u
6
7 int dfsAP(int u, int p) {
8     int children = 0;
9     low[u] = disc[u] = ++Time;
10    for (int& v : adj[u]) {
11        if (v == p) continue; // we don't want to go back through the
12                               // same path.
13        if (!disc[v]) { // if v has not been discovered before
14            children++;
15            dfsAP(v, u); // recursive DFS call
16            if (disc[u] <= low[v]) // condition #1
17                ap[u] = 1;
18            low[u] = min(low[u], low[v]); // low[v] might be an ancestor
19                                         // of u
20        } else // if v was already discovered means that we found an
21               // ancestor
22            low[u] = min(low[u], disc[v]); // finds the ancestor with
23                                         // the least discovery time
24    }
25    return children;
26 }
27
28 void AP() {
29     ap = low = disc = vector<int>(adj.size());
30     Time = 0;
31     for (int u = 0; u < adj.size(); u++)
32         if (!disc[u])
33             ap[u] = dfsAP(u, u) > 1; // condition #2
34 }

```

4.21 Finding bridges

```

1 // br = bridges, p = parent
2

```

```

3 vector<pair<int, int>> br;
4
5 int dfsBR(int u, int p) {
6     low[u] = disc[u] = ++Time;
7     for (int& v : adj[u]) {
8         if (v == p) continue; // we don't want to go back through the
9                                 // same path.
10                                // if we go back is because we found
11                                // another way back
12         if (!disc[v]) { // if V has not been discovered before
13             dfsBR(v, u); // recursive DFS call
14             if (disc[u] < low[v]) // condition to find a bridge
15                 br.push_back({u, v});
16             low[u] = min(low[u], low[v]); // low[v] might be an ancestor
17                                     // of u
18         } else // if v was already discovered means that we found an
19             ancestor
20             low[u] = min(low[u], disc[v]); // finds the ancestor with
21                                     // the least discovery time
22     }
23 }
24
25 void BR() {
26     low = disc = vector<int>(adj.size());
27     Time = 0;
28     for (int u = 0; u < adj.size(); u++)
29         if (!disc[u])
30             dfsBR(u, u)
31 }

```

4.22 Finding Bridges Online

```

1 vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
2 int bridges;
3 int lca_iteration;
4 vector<int> last_visit;
5
6 void init(int n) {
7     par.resize(n);
8     dsu_2ecc.resize(n);
9     dsu_cc.resize(n);
10    dsu_cc_size.resize(n);
11    lca_iteration = 0;
12    last_visit.assign(n, 0);
13    for (int i=0; i<n; ++i) {
14        dsu_2ecc[i] = i;
15        dsu_cc[i] = i;
16        dsu_cc_size[i] = 1;
17        par[i] = -1;

```

```

18    }
19    bridges = 0;
20 }
21
22 int find_2ecc(int v) {
23     if (v == -1)
24         return -1;
25     return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(dsu_2ecc[v]);
26 }
27
28 int find_cc(int v) {
29     v = find_2ecc(v);
30     return dsu_cc[v] == v ? v : dsu_cc[v] = find_cc(dsu_cc[v]);
31 }
32
33 void make_root(int v) {
34     int root = v;
35     int child = -1;
36     while (v != -1) {
37         int p = find_2ecc(par[v]);
38         par[v] = child;
39         dsu_cc[v] = root;
40         child = v;
41         v = p;
42     }
43     dsu_cc_size[root] = dsu_cc_size[child];
44 }
45
46 void merge_path (int a, int b) {
47     ++lca_iteration;
48     vector<int> path_a, path_b;
49     int lca = -1;
50     while (lca == -1) {
51         if (a != -1) {
52             a = find_2ecc(a);
53             path_a.push_back(a);
54             if (last_visit[a] == lca_iteration){
55                 lca = a;
56                 break;
57             }
58             last_visit[a] = lca_iteration;
59             a = par[a];
60         }
61         if (b != -1) {
62             b = find_2ecc(b);
63             path_b.push_back(b);
64             if (last_visit[b] == lca_iteration){
65                 lca = b;
66                 break;
67             }

```



```

68         last_visit[b] = lca_iteration;
69         b = par[b];
70     }
71 }
72
73 for (int v : path_a) {
74     dsu_2ecc[v] = lca;
75     if (v == lca)
76         break;
77     --bridges;
78 }
79 for (int v : path_b) {
80     dsu_2ecc[v] = lca;
81     if (v == lca)
82         break;
83     --bridges;
84 }
85 }
86 }
87
88 void add_edge(int a, int b) {
89     a = find_2ecc(a);
90     b = find_2ecc(b);
91     if (a == b)
92         return;
93
94     int ca = find_cc(a);
95     int cb = find_cc(b);
96
97     if (ca != cb) {
98         ++bridges;
99         if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
100             swap(a, b);
101             swap(ca, cb);
102         }
103         make_root(a);
104         par[a] = dsu_cc[a] = b;
105         dsu_cc_size[cb] += dsu_cc_size[a];
106     } else {
107         merge_path(a, b);
108     }
109 }

```

4.23 Bridge Tree

```

1 vector<pair<int, int>> g[MAXN];
2 bool used[MAXN], isBridge[MAXM];
3 int comp[MAXN], tin[MAXN], minAncestor[MAXN];
4

```

```

5 vector<int> tree[MAXN]; // Store 2-edge-connected component tree.(
6     Bridge tree).
7
8 void dfs(int v, int p) {
9     tin[v] = minAncestor[v] = ++timer;
10    used[v] = 1;
11    for(auto &e: g[v]) {
12        int to, id;
13        tie(to, id) = e;
14        if(to == p) continue;
15        if(used[to]) {
16            minAncestor[v] = min(minAncestor[v], tin[to]);
17        } else {
18            dfs(to, v);
19            minAncestor[v] = min(minAncestor[v], minAncestor[to]);
20            if(minAncestor[to] > tin[v]) {
21                isBridge[id] = true;
22            }
23        }
24    }
25
26 void dfs1(int v, int p) {
27     used[v] = 1;
28     comp[v] = compid;
29     for(auto &e: g[v]) {
30         int to, id;
31         tie(to, id) = e;
32
33         if(isBridge[id]) { // avoid traversing from this edge. so
34             // we get full component.
35             continue;
36         }
37         if(used[to]) {
38             continue;
39         }
40         dfs1(to, v);
41     }
42
43 vector<pair<int, int>> edges;
44
45 void addEdge(int from, int to, int id) {
46     g[from].push_back({to, id});
47     g[to].push_back({from, id});
48     edges[id] = {from, to};
49 }
50
51 void initB() {
52
53     for(int i = 0; i <= compid; ++i)

```

```

54     tree[i].clear();
55     for(int i = 1; i <= N; ++i)
56         used[i] = false;
57     for(int i = 1; i <= M; ++i)
58         isBridge[i] = false;
59
60     timer = 0;
61     compid = 0;
62 }
63
64 void bridge_tree() {
65     initB();
66
67     dfs(1, -1); //Assuming graph is connected.
68
69     for(int i = 1; i <= N; ++i)
70         used[i] = 0;
71
72     for(int i = 1; i <= N; ++i) {
73         if(!used[i]) {
74             dfs1(i, -1);
75             ++compid;
76         }
77     }
78
79     for(int i = 1; i <= M; ++i) {
80         if(isBridge[i]) {
81             int u, v;
82             tie(u, v) = edges[i];
83             // connect two componets using edge.
84             tree[comp[u]].push_back(comp[v]);
85             tree[comp[v]].push_back(comp[u]);
86         }
87     }
88 }
89
90 void init() {
91     edges.clear(); edges.resize(M + 1);
92     for(int i = 1; i <= N; ++i)
93         g[i].clear();
94 }
95

```

4.24 2-SAT

```

1 struct TwoSatSolver {
2     int n_vars;
3     int n_vertices;
4     vector<vector<int>> adj, adj_t;

```

```

5     vector<bool> used;
6     vector<int> order, comp;
7     vector<bool> assignment;
8
9     TwoSatSolver(int _n_vars) : n_vars(_n_vars), n_vertices(2 *
10         n_vars), adj(n_vertices), adj_t(n_vertices), used(
11         n_vertices), order(), comp(n_vertices, -1), assignment(
12         n_vars) {
13         order.reserve(n_vertices);
14     }
15
16     void dfs1(int v) {
17         used[v] = true;
18         for (int u : adj[v]) {
19             if (!used[u])
20                 dfs1(u);
21         }
22         order.push_back(v);
23     }
24
25     void dfs2(int v, int c1) {
26         comp[v] = c1;
27         for (int u : adj_t[v]) {
28             if (comp[u] == -1)
29                 dfs2(u, c1);
30         }
31     }
32
33     bool solve_2SAT() {
34         order.clear();
35         used.assign(n_vertices, false);
36         for (int i = 0; i < n_vertices; ++i) {
37             if (!used[i])
38                 dfs1(i);
39         }
40
41         comp.assign(n_vertices, -1);
42         for (int i = 0, j = 0; i < n_vertices; ++i) {
43             int v = order[n_vertices - i - 1];
44             if (comp[v] == -1)
45                 dfs2(v, j++);
46         }
47
48         assignment.assign(n_vars, false);
49         for (int i = 0; i < n_vertices; i += 2) {
50             if (comp[i] == comp[i + 1])
51                 return false;
52             assignment[i / 2] = comp[i] > comp[i + 1];
53         }
54         return true;
55     }
56 }

```

```

53 void add_disjunction(int a, bool na, int b, bool nb) {
54     // na and nb signify whether a and b are to be negated
55     a = 2 * a ^ na;
56     b = 2 * b ^ nb;
57     int neg_a = a ^ 1;
58     int neg_b = b ^ 1;
59     adj[neg_a].push_back(b);
60     adj[neg_b].push_back(a);
61     adj_t[b].push_back(neg_a);
62     adj_t[a].push_back(neg_b);
63 }
64
65 static void example_usage() {
66     TwoSatSolver solver(3); // a, b, c
67     solver.add_disjunction(0, false, 1, true); // a v
68         not b
69     solver.add_disjunction(0, true, 1, true); // not a v
70         not b
71     solver.add_disjunction(1, false, 2, false); // b v
72         c
73     solver.add_disjunction(0, false, 0, false); // a v
74         a
75     assert(solver.solve_2SAT() == true);
76     auto expected = vector<bool>(True, False, True);
77     assert(solver.assignment == expected);
78 }
79 };

```

4.25 Hierholzer's Algorithm (Eulerian Path)

```

1 int n, m;
2 vector<vector<int>> g;
3 vector<int> in, out, path;
4
5 // Undirected
6
7 int n, m;
8 vector<vector<pair<int, int>>> g;
9 vector<int> path;
10 vector<bool> seen;
11
12 void dfs(int node) {
13     while (!g[node].empty()) {
14         auto [son, idx] = g[node].back();
15         g[node].pop_back();
16         if (seen[idx]) { continue; }
17         seen[idx] = true;
18         dfs(son);
19     }

```

```

20     path.push_back(node);
21 }
22
23 // Directed
24 void dfs(int node) {
25     while (!g[node].empty()) {
26         int son = g[node].back();
27         g[node].pop_back();
28         dfs(son);
29     }
30     path.push_back(node);
31 }

```

4.26 Gale-Shapley Algorithm (Stable marriage)

```

1 // Checks if woman 'w' prefers 'm1' over 'm'
2 bool wPrefersM1OverM(vector<vector<int>> &prefer, int w, int m,
3     int m1)
4 {
5     int N = prefer[0].size();
6     for (int i = 0; i < N; i++)
7     {
8         // If m1 comes before m, w prefers
9         // her current engagement
10        if (prefer[w][i] == m1)
11            return true;
12
13        // If m comes before m1, w prefers m
14        if (prefer[w][i] == m)
15            return false;
16    }
17 }
18
19 // Implements the stable marriage algorithm
20 vector<int> stableMarriage(vector<vector<int>> &prefer)
21 {
22     int N = prefer[0].size();
23
24     // Stores women's partners
25     vector<int> wPartner(N, -1);
26
27     // Tracks free men
28     vector<bool> mFree(N, false);
29     int freeCount = N;
30
31     while (freeCount > 0)
32     {
33         int m;
34         for (m = 0; m < N; m++)

```

```

34         if (!mFree[m])
35             break;
36
37     // Process each woman in m's preference list
38     for (int i = 0; i < N && !mFree[m]; i++)
39     {
40         int w = prefer[m][i];
41         if (wPartner[w - N] == -1)
42         {
43             // Engage m and w if w is free
44             wPartner[w - N] = m;
45             mFree[m] = true;
46             freeCount--;
47         }
48         else
49         {
50             int m1 = wPartner[w - N];
51             // If w prefers m over her current partner,
52             // reassign
53             if (!wPrefersM1OverM(prefer, w, m, m1))
54             {
55                 wPartner[w - N] = m;
56                 mFree[m] = true;
57                 mFree[m1] = false;
58             }
59         }
60     }
61     return wPartner;
62 }

```

5 Trees

5.1 Succesor

```

1  const ll mod=1e9+7;
2  const ll MAX=1e9+1;
3  const int limit=2e5+1;
4  const int m=30;
5  int sucesorM[limit][m];
6  //ascii https://elcodigoascii.com.ar/
7
8  inline void solve()
9  {
10     int n,q; cin>>n>>q;
11     int res,aux;
12     ll k;
13     lFOR(i,n){

```

```

14         cin>>sucesorM[i][0];
15     }
16     FOR(j,1,m)
17     {
18         lFOR(i,n)
19         {
20             sucesorM[i][j]=sucesorM[sucesorM[i][j-1]][j-1];
21         }
22     }
23     FO(i,q)
24     {
25         cin>>res>>k;
26         aux=0;
27         while(k)
28         {
29             if(k%2){
30                 res=sucesorM[res][aux];
31             }
32             k/=2;
33             aux++;
34         }
35         cout<<res<<endl;
36     }
37 }

```

5.2 Euler Tour

```

1  const int MAXN = 1e5 + 5;
2
3  vector<int> adj[MAXN];
4  int in_time[MAXN], out_time[MAXN];
5  int timer = 0;
6
7  struct FenwickTree {
8      vector<int> bit;
9      int n;
10
11      FenwickTree(int n) {
12          this->n = n;
13          bit.assign(n + 1, 0);
14      }
15
16      void update(int idx, int delta) {
17          for (; idx <= n; idx += idx & -idx)
18              bit[idx] += delta;
19      }
20
21      int query(int idx) {
22          int sum = 0;

```

```

23     for (; idx > 0; idx -= idx & -idx)
24         sum += bit[idx];
25     return sum;
26 }
27
28 int range_query(int l, int r) {
29     return query(r) - query(l - 1);
30 }
31 };
32
33 void euler_tour(int root) {
34     stack<tuple<int, int, bool>> st;
35     st.push({root, -1, false});
36
37     while (!st.empty()) {
38         auto [u, parent, visited] = st.top();
39         st.pop();
40
41         if (!visited) {
42             in_time[u] = ++timer;
43             st.push({u, parent, true});
44
45             for (auto it = adj[u].rbegin(); it != adj[u].rend();
46                  ++it) {
47                 if (*it != parent) {
48                     st.push({*it, u, false});
49                 }
50             }
51             out_time[u] = ++timer;
52         }
53     }
54 }

```

5.3 Lowest Common Ancestor

```

1 struct LCA {
2     vector<int> height, euler, first, segtree;
3     vector<bool> visited;
4     int n;
5
6     LCA(vector<vector<int>> &adj, int root = 0) {
7         n = adj.size();
8         height.resize(n);
9         first.resize(n);
10        euler.reserve(n * 2);
11        visited.assign(n, false);
12        dfs(adj, root);
13        int m = euler.size();

```

```

14        segtree.resize(m * 4);
15        build(1, 0, m - 1);
16    }
17
18    void dfs(vector<vector<int>> &adj, int node, int h = 0) {
19        visited[node] = true;
20        height[node] = h;
21        first[node] = euler.size();
22        euler.push_back(node);
23        for (auto to : adj[node]) {
24            if (!visited[to]) {
25                dfs(adj, to, h + 1);
26                euler.push_back(node);
27            }
28        }
29    }
30
31    void build(int node, int b, int e) {
32        if (b == e) {
33            segtree[node] = euler[b];
34        } else {
35            int mid = (b + e) / 2;
36            build(node << 1, b, mid);
37            build(node << 1 | 1, mid + 1, e);
38            int l = segtree[node << 1], r = segtree[node << 1 | 1];
39            segtree[node] = (height[l] < height[r]) ? l : r;
40        }
41    }
42
43    int query(int node, int b, int e, int L, int R) {
44        if (b > R || e < L)
45            return -1;
46        if (b >= L && e <= R)
47            return segtree[node];
48        int mid = (b + e) >> 1;
49
50        int left = query(node << 1, b, mid, L, R);
51        int right = query(node << 1 | 1, mid + 1, e, L, R);
52        if (left == -1) return right;
53        if (right == -1) return left;
54        return height[left] < height[right] ? left : right;
55    }
56
57    int lca(int u, int v) {
58        int left = first[u], right = first[v];
59        if (left > right)
60            swap(left, right);
61        return query(1, 0, euler.size() - 1, left, right);
62    }
63 };

```

5.4 Binary Lifting

```

1 int n, l;
2 vector<vector<int>> adj;
3
4 int timer;
5 vector<int> tin, tout;
6 vector<vector<int>> up;
7
8 void dfs(int v, int p)
9 {
10     tin[v] = ++timer;
11     up[v][0] = p;
12     for (int i = 1; i <= l; ++i)
13         up[v][i] = up[up[v][i-1]][i-1];
14
15     for (int u : adj[v]) {
16         if (u != p)
17             dfs(u, v);
18     }
19
20     tout[v] = ++timer;
21 }
22
23 bool is_ancestor(int u, int v)
24 {
25     return tin[u] <= tin[v] && tout[u] >= tout[v];
26 }
27
28 int lca(int u, int v)
29 {
30     if (is_ancestor(u, v))
31         return u;
32     if (is_ancestor(v, u))
33         return v;
34     for (int i = l; i >= 0; --i) {
35         if (!is_ancestor(up[u][i], v))
36             u = up[u][i];
37     }
38     return up[u][0];
39 }
40
41 void preprocess(int root) {
42     tin.resize(n);
43     tout.resize(n);
44     timer = 0;
45     l = ceil(log2(n));
46     up.assign(n, vector<int>(l + 1));

```

```

47     dfs(root, root);
48 }

```

5.5 Cartesian Tree

```

1 vector<int> parent(n, -1);
2 stack<int> s;
3 for (int i = 0; i < n; i++) {
4     int last = -1;
5     while (!s.empty() && A[s.top()] >= A[i]) {
6         last = s.top();
7         s.pop();
8     }
9     if (!s.empty())
10         parent[i] = s.top();
11     if (last >= 0)
12         parent[last] = i;
13     s.push(i);
14 }

```

5.6 Heavy-Light Decomposition

```

1 vector<int> parent, depth, heavy, head, pos;
2 int cur_pos;
3
4 int dfs(int v, vector<vector<int>> const& adj) {
5     int size = 1;
6     int max_c_size = 0;
7     for (int c : adj[v]) {
8         if (c != parent[v]) {
9             parent[c] = v, depth[c] = depth[v] + 1;
10            int c_size = dfs(c, adj);
11            size += c_size;
12            if (c_size > max_c_size)
13                max_c_size = c_size, heavy[v] = c;
14        }
15    }
16    return size;
17 }
18
19 void decompose(int v, int h, vector<vector<int>> const& adj) {
20     head[v] = h, pos[v] = cur_pos++;
21     if (heavy[v] != -1)
22         decompose(heavy[v], h, adj);
23     for (int c : adj[v]) {
24         if (c != parent[v] && c != heavy[v])
25             decompose(c, c, adj);
26    }

```

```

27 }
28
29 void init(vector<vector<int>> const& adj) {
30     int n = adj.size();
31     parent = vector<int>(n);
32     depth = vector<int>(n);
33     heavy = vector<int>(n, -1);
34     head = vector<int>(n);
35     pos = vector<int>(n);
36     cur_pos = 0;
37
38     dfs(0, adj);
39     decompose(0, 0, adj);
40 }
41
42 int query(int a, int b) {
43     int res = 0;
44     for (; head[a] != head[b]; b = parent[head[b]]) {
45         if (depth[head[a]] > depth[head[b]])
46             swap(a, b);
47         int cur_heavy_path_max = segment_tree_query(pos[head[b]],
48             pos[b]);
49         res = max(res, cur_heavy_path_max);
50     }
51     if (depth[a] > depth[b])
52         swap(a, b);
53     int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
54     res = max(res, last_heavy_path_max);
55     return res;
56 }

```

5.7 Centroid Decomposition

```

1     vector<vector<int>> adj;
2     vector<bool> is_removed;
3     vector<int> subtree_size;
4
5     /** DFS to calculate the size of the subtree rooted at 'node' */
6     int get_subtree_size(int node, int parent = -1) {
7         subtree_size[node] = 1;
8         for (int child : adj[node]) {
9             if (child == parent || is_removed[child]) { continue; }
10            subtree_size[node] += get_subtree_size(child, node);
11        }
12        return subtree_size[node];
13    }
14
15    /**

```

```

16     * Returns a centroid (a tree may have two centroids) of the
17     * subtree
18     * @param node current node
19     * @param tree_size size of current subtree after node removals
20     * @param parent parent of u
21     * @return first centroid found
22     */
23     int get_centroid(int node, int tree_size, int parent = -1) {
24         for (int child : adj[node]) {
25             if (child == parent || is_removed[child]) { continue; }
26             if (subtree_size[child] * 2 > tree_size) {
27                 return get_centroid(child, tree_size, node);
28             }
29         }
30         return node;
31     }
32
33     /** Build up the centroid decomposition recursively */
34     void build_centroid_decomp(int node = 0) {
35         int centroid = get_centroid(node, get_subtree_size(node));
36
37         // do something
38
39         is_removed[centroid] = true;
40
41         for (int child : adj[centroid]) {
42             if (is_removed[child]) { continue; }
43             build_centroid_decomp(child);
44         }
45     }

```

5.8 Tree Distances

```

1     vector<int> graph[200001];
2     int fir[200001], sec[200001], ans[200001];
3
4     void dfs1(int node = 1, int parent = 0) {
5         for (int i : graph[node])
6             if (i != parent) {
7                 dfs1(i, node);
8                 if (fir[i] + 1 > fir[node]) {
9                     sec[node] = fir[node];
10                    fir[node] = fir[i] + 1;
11                } else if (fir[i] + 1 > sec[node]) {
12                    sec[node] = fir[i] + 1;
13                }
14            }
15    }

```

```

16 }
17
18 void dfs2(int node = 1, int parent = 0, int to_p = 0) {
19     ans[node] = max(to_p, fir[node]);
20     for (int i : graph[node])
21         if (i != parent) {
22             if (fir[i] + 1 == fir[node]) dfs2(i, node, max(to_p,
23                 sec[node] + 1);
24             else dfs2(i, node, ans[node] + 1);
25 }

```

6 Flows

6.1 Ford-Fulkerson Maximum Flow

```

1 int n;
2 vector<vector<int>> capacity;
3 vector<vector<int>> adj;
4
5 int bfs(int s, int t, vector<int>& parent) {
6     fill(parent.begin(), parent.end(), -1);
7     parent[s] = -2;
8     queue<pair<int, int>> q;
9     q.push({s, INF});
10
11     while (!q.empty()) {
12         int cur = q.front().first;
13         int flow = q.front().second;
14         q.pop();
15
16         for (int next : adj[cur]) {
17             if (parent[next] == -1 && capacity[cur][next]) {
18                 parent[next] = cur;
19                 int new_flow = min(flow, capacity[cur][next]);
20                 if (next == t)
21                     return new_flow;
22                 q.push({next, new_flow});
23             }
24         }
25     }
26
27     return 0;
28 }
29
30 int maxflow(int s, int t) {
31     int flow = 0;
32     vector<int> parent(n);

```

```

33     int new_flow;
34
35     while (new_flow = bfs(s, t, parent)) {
36         flow += new_flow;
37         int cur = t;
38         while (cur != s) {
39             int prev = parent[cur];
40             capacity[prev][cur] -= new_flow;
41             capacity[cur][prev] += new_flow;
42             cur = prev;
43         }
44     }
45
46     return flow;
47 }

```

6.2 Dinic's Max Flow

```

1 struct FlowEdge {
2     int v, u;
3     long long cap, flow = 0;
4     FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap)
5     {}
6 };
7
8 struct Dinic {
9     const long long flow_inf = 1e18;
10    vector<FlowEdge> edges;
11    vector<vector<int>> adj;
12    int n, m = 0;
13    int s, t;
14    vector<int> level, ptr;
15    queue<int> q;
16
17    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
18        adj.resize(n);
19        level.resize(n);
20        ptr.resize(n);
21    }
22
23    void add_edge(int v, int u, long long cap) {
24        edges.emplace_back(v, u, cap);
25        edges.emplace_back(u, v, 0);
26        adj[v].push_back(m);
27        adj[u].push_back(m + 1);
28        m += 2;
29    }
30
31    bool bfs() {

```



```

31 while (!q.empty()) {
32     int v = q.front();
33     q.pop();
34     for (int id : adj[v]) {
35         if (edges[id].cap == edges[id].flow)
36             continue;
37         if (level[edges[id].u] != -1)
38             continue;
39         level[edges[id].u] = level[v] + 1;
40         q.push(edges[id].u);
41     }
42 }
43 return level[t] != -1;
44 }
45
46 long long dfs(int v, long long pushed) {
47     if (pushed == 0)
48         return 0;
49     if (v == t)
50         return pushed;
51     for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
52         int id = adj[v][cid];
53         int u = edges[id].u;
54         if (level[v] + 1 != level[u])
55             continue;
56         long long tr = dfs(u, min(pushed, edges[id].cap -
57                                 edges[id].flow));
58         if (tr == 0)
59             continue;
60         edges[id].flow += tr;
61         edges[id ^ 1].flow -= tr;
62         return tr;
63     }
64     return 0;
65 }
66
67 long long flow() {
68     long long f = 0;
69     while (true) {
70         fill(level.begin(), level.end(), -1);
71         level[s] = 0;
72         q.push(s);
73         if (!bfs())
74             break;
75         fill(ptr.begin(), ptr.end(), 0);
76         while (long long pushed = dfs(s, flow_inf)) {
77             f += pushed;
78         }
79     }
80     return f;
}

```

```
81 };
```

6.3 Min-cost Flow

```

1 struct Edge
2 {
3     int from, to, capacity, cost;
4 };
5
6 vector<vector<int>> adj, cost, capacity;
7
8 const int INF = 1e9;
9
10 void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p)
11 {
12     d.assign(n, INF);
13     d[v0] = 0;
14     vector<bool> inq(n, false);
15     queue<int> q;
16     q.push(v0);
17     p.assign(n, -1);
18
19     while (!q.empty()) {
20         int u = q.front();
21         q.pop();
22         inq[u] = false;
23         for (int v : adj[u]) {
24             if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
25                 d[v] = d[u] + cost[u][v];
26                 p[v] = u;
27                 if (!inq[v]) {
28                     inq[v] = true;
29                     q.push(v);
30                 }
31             }
32         }
33     }
34 }
35
36 int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t)
37 {
38     adj.assign(N, vector<int>());
39     cost.assign(N, vector<int>(N, 0));
40     capacity.assign(N, vector<int>(N, 0));
41     for (Edge e : edges) {
42         adj[e.from].push_back(e.to);
43         adj[e.to].push_back(e.from);
44         cost[e.from][e.to] = e.cost;
45         cost[e.to][e.from] = -e.cost;
46     }
47 }

```

```

44     capacity[e.from][e.to] = e.capacity;
45 }
46
47 int flow = 0;
48 int cost = 0;
49 vector<int> d, p;
50 while (flow < K) {
51     shortest_paths(N, s, d, p);
52     if (d[t] == INF)
53         break;
54
55     // find max flow on that path
56     int f = K - flow;
57     int cur = t;
58     while (cur != s) {
59         f = min(f, capacity[p[cur]][cur]);
60         cur = p[cur];
61     }
62
63     // apply flow
64     flow += f;
65     cost += f * d[t];
66     cur = t;
67     while (cur != s) {
68         capacity[p[cur]][cur] -= f;
69         capacity[cur][p[cur]] += f;
70         cur = p[cur];
71     }
72 }
73
74 if (flow < K)
75     return -1;
76 else
77     return cost;
78 }

```

6.4 Hungarian Algorithm

```

1 vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
2 for (int i=1; i<=n; ++i) {
3     p[0] = i;
4     int j0 = 0;
5     vector<int> minv (m+1, INF);
6     vector<bool> used (m+1, false);
7     do {
8         used[j0] = true;
9         int i0 = p[j0], delta = INF, j1;
10        for (int j=1; j<=m; ++j)
11            if (!used[j]) {

```

```

12                int cur = A[i0][j]-u[i0]-v[j];
13                if (cur < minv[j])
14                    minv[j] = cur, way[j] = j0;
15                if (minv[j] < delta)
16                    delta = minv[j], j1 = j;
17            }
18        for (int j=0; j<=m; ++j)
19            if (used[j])
20                u[p[j]] += delta, v[j] -= delta;
21        else
22            minv[j] -= delta;
23        j0 = j1;
24    } while (p[j0] != 0);
25    do {
26        int j1 = way[j0];
27        p[j0] = p[j1];
28        j0 = j1;
29    } while (j0);
30 }
31
32 vector<int> ans (n+1);
33 for (int j=1; j<=m; ++j)
34     ans[p[j]] = j;
35
36 int cost = -v[0];

```

6.5 Kuhn's Algorithm

```

1 int n, k;
2 vector<vector<int>> g;
3 vector<int> mt;
4 vector<bool> used;
5
6 bool try_kuhn(int v) {
7     if (used[v])
8         return false;
9     used[v] = true;
10    for (int to : g[v]) {
11        if (mt[to] == -1 || try_kuhn(mt[to])) {
12            mt[to] = v;
13            return true;
14        }
15    }
16    return false;
17 }
18
19 int main() {
20     //... reading the graph ...
21 }

```

```

22     mt.assign(k, -1);
23     for (int v = 0; v < n; ++v) {
24         used.assign(n, false);
25         try_kuhn(v);
26     }
27
28     for (int i = 0; i < k; ++i)
29         if (mt[i] != -1)
30             printf("%d %d\n", mt[i] + 1, i + 1);
31 }

```

7 Dynamic Programming

7.1 Coin Problem (Count ways)

```

1 vector<ll> coins(n);
2 for(int i=0; i<n; i++){
3     cin>>coins[i];
4 }
5
6 vector<ll> dp(x+1,0);
7 dp[0] = 1;
8 for(int i=0; i<=x; i++){
9     for(int j=0; j<n; j++){
10        if(i-coins[j]>=0){
11            dp[i] = (dp[i] + dp[i-coins[j]]);
12            dp[i]%=MOD;
13        }
14    }
15 }
16
17
18 cout<<dp[x]<<endl;

```

7.2 Coin Problem (Count sorted ways)

```

1 vector<ll> coins(n);
2 for(int i=0; i<n; i++){
3     cin>>coins[i];
4 }
5
6 int dp[102][1000005];
7 dp[0][0] = 1;
8 for(int i=1; i<=n; i++){
9     for(int j=0; j<=x; j++){
10        dp[i][j] = dp[i-1][j];
11        int l = j-coins[i-1];

```

```

12        if(l>=0){
13            dp[i][j] += (dp[i][l])%MOD;
14            dp[i][j]%=MOD;
15        }
16    }
17 }
18
19
20 cout<<dp[n][x]%MOD<<endl;

```

7.3 Coin Problem (Minimum)

```

1 vector<ll> coins(n);
2 for(int i=0; i<n; i++){
3     cin>>coins[i];
4 }
5
6 vector<ll> dp(x+1,INT_MAX);
7 dp[0] = 0;
8 for(int i=0; i<=x; i++){
9     for(int j=0; j<n; j++){
10        if(i-coins[j]>=0){
11            dp[i] = min(dp[i], dp[i-coins[j]]+1);
12        }
13    }
14 }
15
16 if(dp[x] != INT_MAX){
17     cout<<dp[x]<<endl;
18 }else{
19     cout<<"-1"<<endl;
20 }

```

7.4 Counting paths

```

1 int n; cin>>n;
2 char grid[n][n];
3 int dp[n][n];
4
5 for(int i=0; i<n; i++){
6     for(int j=0; j<n; j++){
7         cin>>grid[i][j];
8         dp[i][j] = 0;
9     }
10 }
11 if(grid[0][0] != '*') dp[0][0] = 1;
12 else dp[0][0] = 0;
13 for(int i=0; i<n; i++){

```

```

14     for(int j=0; j<n; j++){
15         if(grid[i+1][j] == '.' and i+1 < n){
16             dp[i+1][j] += dp[i][j]%MOD;
17         }
18         if(grid[i][j+1] == '.' and j+1 < n){
19             dp[i][j+1] += dp[i][j]%MOD;
20         }
21
22         if(grid[i][j] == '*'){
23             dp[i][j] = 0;
24         }
25     }
26 }
27 cout<<dp[n-1][n-1]%MOD<<endl;

```

7.5 Longest Increasing Subsequence

```

1 vector<int> lis(vector<int> const& a) {
2     int n = a.size();
3     vector<int> d(n, 1), p(n, -1);
4     for (int i = 0; i < n; i++) {
5         for (int j = 0; j < i; j++) {
6             if (a[j] < a[i] && d[i] < d[j] + 1) {
7                 d[i] = d[j] + 1;
8                 p[i] = j;
9             }
10        }
11    }
12
13    int ans = d[0], pos = 0;
14    for (int i = 1; i < n; i++) {
15        if (d[i] > ans) {
16            ans = d[i];
17            pos = i;
18        }
19    }
20
21    vector<int> subseq;
22    while (pos != -1) {
23        subseq.push_back(a[pos]);
24        pos = p[pos];
25    }
26    reverse(subseq.begin(), subseq.end());
27    return subseq;
28 }

```

7.6 Length of LIS

```

1 int lis(vector<ll> const& a) {
2     int n = a.size();
3     const int INF = 1e9;
4     vector<int> d(n+1, INF);
5     d[0] = -INF;
6
7     for (int i = 0; i < n; i++) {
8         int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
9         if (d[l-1] < a[i] && a[i] < d[l])
10             d[l] = a[i];
11     }
12
13     int ans = 0;
14     for (int l = 0; l <= n; l++) {
15         if (d[l] < INF)
16             ans = l;
17     }
18     return ans;
19 }

```

7.7 Longest Common Subsequence

```

1 // Returns length of LCS for s1[0..m-1], s2[0..n-1]
2 int lcs(string &s1, string &s2) {
3     int m = s1.size();
4     int n = s2.size();
5
6     // Initializing a matrix of size (m+1)*(n+1)
7     vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
8
9     // Building dp[m+1][n+1] in bottom-up fashion
10    for (int i = 1; i <= m; ++i) {
11        for (int j = 1; j <= n; ++j) {
12            if (s1[i - 1] == s2[j - 1])
13                dp[i][j] = dp[i - 1][j - 1] + 1;
14            else
15                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
16        }
17    }
18
19    // dp[m][n] contains length of LCS for s1[0..m-1]
20    // and s2[0..n-1]
21    return dp[m][n];
22 }

```

7.8 Edit Distance

```

1 int editDistance(string &s1, string &s2) {
2
3     int m = s1.length();
4     int n = s2.length();
5
6     // Create a table to store results of subproblems
7     vector<vector<int>> dp(m + 1, vector<int>(n + 1));
8
9     // Fill the known entries in dp[][]
10    // If one string is empty, then answer
11    // is length of the other string
12    for (int i = 0; i <= m; i++)
13        dp[i][0] = i;
14    for (int j = 0; j <= n; j++)
15        dp[0][j] = j;
16
17    // Fill the rest of dp[][]
18    for (int i = 1; i <= m; i++) {
19        for (int j = 1; j <= n; j++) {
20            if (s1[i - 1] == s2[j - 1])
21                dp[i][j] = dp[i - 1][j - 1];
22            else
23                dp[i][j] = 1 + min({dp[i][j - 1],
24                                dp[i - 1][j],
25                                dp[i - 1][j - 1]});
26        }
27    }
28
29    return dp[m][n];
30 }

```

7.9 Bitmask DP

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 const ll mod=1e9+7;
4 const int limit=20;
5 vector<pair<ll,ll>> dp((1<<limit));
6 //ascii https://elcodigoascii.com.ar/
7
8 inline void solve()
9 {
10     int n; cin>>n;
11     ll x; cin>>x;
12     vector<ll> weight(n);
13     dp[0]={1,0};
14     F0(i,n) cin>>weight[i];
15     for(ll i=1;i<(1<<n);i++)
16     {

```

```

17         dp[i]={n+1,0};
18         for(int j=0;j<n;j++)
19         {
20             if(i&(1<<j))
21             {
22                 pair<ll,ll> aux=dp[i^(1<<j)];
23                 if(aux.second+weight[j]<=x){
24                     aux.second+=weight[j];
25                 }
26                 else{
27                     aux.first++;
28                     aux.second=weight[j];
29                 }
30                 dp[i]=min(dp[i],aux);
31             }
32         }
33     }
34     cout<<dp[(1<<n)-1].first<<endl;
35 }

```

7.10 Digit DP

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 const ll mod=1e9+7;
4 ll dp[20][10][2][2];
5 //ascii https://elcodigoascii.com.ar/
6
7 ll mem(int idx,int tight,int prev,int ld,string s)
8 {
9     if(idx==0)
10     {
11         return 1;
12     }
13     if(dp[idx][prev][ld][tight]!=-1){
14         return dp[idx][prev][ld][tight];
15     }
16     int k=9;
17     if(tight) k=s[s.size()-idx]-'0';
18     ll sum=0;
19     for(int i=0;i<=k;i++)
20     {
21         if(ld || prev!=i)
22         {
23             int new_ld,new_tight;
24             if(i==0 && ld) new_ld=1;
25             else new_ld=0;
26             if(tight && k==i) new_tight=1;
27             else new_tight=0;

```

```

28         sum+=mem(idx-1,new_tight,i,new_ld,s);
29     }
30 }
31 dp[idx][prev][ld][tight]=sum;
32 return sum;
33 }

```

7.11 Double DP

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 const ll mod=1e9+7;
4 const ll MAX=1e6+3;
5 ll dp[MAX][2];
6 //ascii https://elcodigoascii.com.ar/
7
8 inline void solve()
9 {
10     int n; cin>>n;
11     dp[n][0]=1;
12     dp[n][1]=1;
13     for(int i=n-1;i>0;i--)
14     {
15         dp[i][1]=4*dp[i+1][1]+dp[i+1][0];
16         dp[i][0]=2*dp[i+1][0]+dp[i+1][1];
17         dp[i][1]%=mod;
18         dp[i][0]%=mod;
19     }
20     cout<<(dp[1][1]+dp[1][0])%mod<<endl;
21 }

```

8 Math

8.1 Miller Rabin

```

1 bool MillerRabin(u64 n) { // returns true if n is prime, else
2     returns false.
3     if (n < 2)
4         return false;
5
6     int r = 0;
7     u64 d = n - 1;
8     while ((d & 1) == 0) {
9         d >>= 1;
10        r++;
11    }

```

```

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

```

8.2 Sieve of Erathostenes

```

1 int n;
2 vector<bool> is_prime(n+1, true);
3 is_prime[0] = is_prime[1] = false;
4 for (int i = 2; i * i <= n; i++) {
5     if (is_prime[i]) {
6         for (int j = i * i; j <= n; j += i)
7             is_prime[j] = false;
8     }
9 }

```

8.3 Sieve of Eratosthenes (count primes)

```

1 int count_primes(int n) {
2     const int S = 10000;
3
4     vector<int> primes;
5     int nsqrt = sqrt(n);
6     vector<char> is_prime(nsqrt + 2, true);
7     for (int i = 2; i <= nsqrt; i++) {
8         if (is_prime[i]) {
9             primes.push_back(i);
10            for (int j = i * i; j <= nsqrt; j += i)
11                is_prime[j] = false;
12        }
13    }
14
15    int result = 0;
16    vector<char> block(S);
17    for (int k = 0; k * S <= n; k++) {
18        fill(block.begin(), block.end(), true);
19        int start = k * S;
20        for (int p : primes) {
21            int start_idx = (start + p - 1) / p;
22            int j = max(start_idx, p) * p - start;
23            for (; j < S; j += p)
24                block[j] = false;
25        }

```

```

26     if (k == 0)
27         block[0] = block[1] = false;
28     for (int i = 0; i < S && start + i <= n; i++) {
29         if (block[i])
30             result++;
31     }
32 }
33 return result;
34 }

```

8.4 Segmented Sieve

```

1 vector<char> segmentedSieve(long long L, long long R) {
2     // generate all primes up to sqrt(R)
3     long long lim = sqrt(R);
4     vector<char> mark(lim + 1, false);
5     vector<long long> primes;
6     for (long long i = 2; i <= lim; ++i) {
7         if (!mark[i]) {
8             primes.emplace_back(i);
9             for (long long j = i * i; j <= lim; j += i)
10                 mark[j] = true;
11         }
12     }
13
14     vector<char> isPrime(R - L + 1, true);
15     for (long long i : primes)
16         for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
17             isPrime[j - L] = false;
18     if (L == 1)
19         isPrime[0] = false;
20     return isPrime;
21 }

```

8.5 Linear sieve

```

1 const int N = 10000000;
2 vector<int> lp(N+1);
3 vector<int> pr;
4
5 for (int i=2; i <= N; ++i) {
6     if (lp[i] == 0) {
7         lp[i] = i;
8         pr.push_back(i);
9     }
10    for (int j = 0; i * pr[j] <= N; ++j) {
11        lp[i * pr[j]] = pr[j];

```

```

12        if (pr[j] == lp[i]) {
13            break;
14        }
15    }
16 }

```

8.6 Sum of divisors

```

1 long long SumOfDivisors(long long num) {
2     long long total = 1;
3
4     for (int i = 2; (long long)i * i <= num; i++) {
5         if (num % i == 0) {
6             int e = 0;
7             do {
8                 e++;
9                 num /= i;
10            } while (num % i == 0);
11
12            long long sum = 0, pow = 1;
13            do {
14                sum += pow;
15                pow *= i;
16            } while (e-- > 0);
17            total *= sum;
18        }
19    }
20    if (num > 1) {
21        total *= (1 + num);
22    }
23    return total;
24 }

```

8.7 Finding the divisors of a number (Trial Division)

```

1 vector<long long> trial_division2(long long n) {
2     vector<long long> factorization;
3     while (n % 2 == 0) {
4         factorization.push_back(2);
5         n /= 2;
6     }
7     for (long long d = 3; d * d <= n; d += 2) {
8         while (n % d == 0) {
9             factorization.push_back(d);
10            n /= d;
11        }
12    }
13    if (n > 1)

```

```

14     factorization.push_back(n);
15     return factorization;
16 }

```

8.8 Finding the divisors of a number (Fermat)

```

1 int fermat(int n) {
2     int a = ceil(sqrt(n));
3     int b2 = a*a - n;
4     int b = round(sqrt(b2));
5     while (b * b != b2) {
6         a = a + 1;
7         b2 = a*a - n;
8         b = round(sqrt(b2));
9     }
10    return a - b;
11 }

```

8.9 Factorials

```

1 // Precompute factorials and inverse factorials
2 void precompute(ll n = MAXN - 1) {
3     factorial[0] = factorial[1] = 1;
4
5     // Compute factorials
6     for (ll i = 2; i <= n; i++) {
7         factorial[i] = (factorial[i - 1] * i) % MOD;
8     }
9
10    // Compute inverse factorials efficiently
11    inv_factorial[n] = modInv(factorial[n]);
12    for (ll i = n - 1; i >= 0; i--) {
13        inv_factorial[i] =
14            (inv_factorial[i + 1] * (i + 1)) % MOD;
15    }
16 }

```

8.10 Binpow

```

1 long long binpow(long long a, long long b) {
2     long long res = 1;
3     while (b > 0) {
4         if (b & 1)
5             res = res * a;
6         a = a * a;
7         b >>= 1;
8     }

```

```

9     return res;
10 }

```

8.11 Modulo Inverse

```

1 int modInverse(int A, int M) {
2     int m0 = M;
3     int y = 0, x = 1;
4
5     if (M == 1)
6         return 0;
7
8     while (A > 1) {
9         // q is quotient
10        int q = A / M;
11        int t = M;
12
13        // m is remainder now, process same as
14        // Euclid's algo
15        M = A % M, A = t;
16        t = y;
17
18        // Update y and x
19        y = x - q * y;
20        x = t;
21    }
22
23    // Make x positive
24    if (x < 0)
25        x += m0;
26
27    return x;
28 }

```

8.12 BinPow Modulo Inv

```

1 ll modInv(ll a, ll mod = MOD) {
2     return power(a, mod - 2, mod);
3 }

```

8.13 Binomial Coefficients

```

1 long long binomial_coefficient(int n, int k) {
2     return factorial[n] * inverse_factorial[k] % m *
3         inverse_factorial[n - k] % m;

```


8.14 Newton Method (Sqrt and iSqrt)

```

1 double sqrt_newton(double n) {
2     const double eps = 1E-15;
3     double x = 1;
4     for (;;) {
5         double nx = (x + n / x) / 2;
6         if (abs(x - nx) < eps)
7             break;
8         x = nx;
9     }
10    return x;
11 }
12
13 int isqrt_newton(int n) {
14     int x = 1;
15     bool decreased = false;
16     for (;;) {
17         int nx = (x + n / x) >> 1;
18         if (x == nx || nx > x && decreased)
19             break;
20         decreased = nx < x;
21         x = nx;
22     }
23     return x;
24 }

```

8.15 Integration with Simpson Method

```

1 const int N = 1000 * 1000; // number of steps (already multiplied
   by 2)
2
3 double simpson_integration(double a, double b){
4     double h = (b - a) / N;
5     double s = f(a) + f(b); // a = x_0 and b = x_2n
6     for (int i = 1; i <= N - 1; ++i) { // Refer to final Simpson's
       formula
7         double x = a + h * i;
8         s += f(x) * ((i & 1) ? 4 : 2);
9     }
10    s *= h / 3;
11    return s;
12 }

```

8.16 Ternary Search

```

1 double ternary_search(double l, double r) {
2     double eps = 1e-9; //set the error limit here

```

```

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

```

8.17 DP Pascal triangle 1D

```

1 int binomialCoeff(int n, int k) {
2     vector<int> dp(k + 1);
3
4     // nC0 is 1
5     dp[0] = 1;
6
7     for (int i = 1; i <= n; i++) {
8
9         // Compute next row of pascal triangle using
10        // the previous row
11        for (int j = min(i, k); j > 0; j--)
12            dp[j] = dp[j] + dp[j - 1];
13    }
14    return dp[k];
15 }

```

8.18 DP Pascal triangle 2D

```

1 // Returns value of Binomial Coefficient C(n, k)
2 int binomialCoeff(int n, int k) {
3     vector<vector<int>> dp(n + 1, vector<int> (k + 1));
4
5     // Calculate value of Binomial Coefficient
6     // in bottom up manner
7     for (int i = 0; i <= n; i++) {
8         for (int j = 0; j <= min(i, k); j++) {
9
10            // Base Cases
11            if (j == 0 || j == i)
12                dp[i][j] = 1;
13
14            // Calculate value using previously

```

```

15         // stored values
16         else
17             dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
18     }
19 }
20
21 return dp[n][k];
22 }

```

8.19 Euler's Totient

```

1 void phi_1_to_n(int n) {
2     vector<int> phi(n + 1);
3     for (int i = 0; i <= n; i++)
4         phi[i] = i;
5
6     for (int i = 2; i <= n; i++) {
7         if (phi[i] == i) {
8             for (int j = i; j <= n; j += i)
9                 phi[j] -= phi[j] / i;
10        }
11    }
12 }
13
14 void phi_1_to_n(int n) {
15     vector<int> phi(n + 1);
16     phi[0] = 0;
17     phi[1] = 1;
18     for (int i = 2; i <= n; i++)
19         phi[i] = i - 1;
20
21     for (int i = 2; i <= n; i++)
22         for (int j = 2 * i; j <= n; j += i)
23             phi[j] -= phi[i];
24 }

```

8.20 Diophantine equations

```

1 void shift_solution(int & x, int & y, int a, int b, int cnt) {
2     x += cnt * b;
3     y -= cnt * a;
4 }
5
6 int find_all_solutions(int a, int b, int c, int minx, int maxx,
7 int miny, int maxy) {
8     int x, y, g;
9     if (!find_any_solution(a, b, c, x, y, g))
10         return 0;

```

```

10     a /= g;
11     b /= g;
12
13     int sign_a = a > 0 ? +1 : -1;
14     int sign_b = b > 0 ? +1 : -1;
15
16     shift_solution(x, y, a, b, (minx - x) / b);
17     if (x < minx)
18         shift_solution(x, y, a, b, sign_b);
19     if (x > maxx)
20         return 0;
21     int lx1 = x;
22
23     shift_solution(x, y, a, b, (maxx - x) / b);
24     if (x > maxx)
25         shift_solution(x, y, a, b, -sign_b);
26     int rx1 = x;
27
28     shift_solution(x, y, a, b, -(miny - y) / a);
29     if (y < miny)
30         shift_solution(x, y, a, b, -sign_a);
31     if (y > maxy)
32         return 0;
33     int lx2 = x;
34
35     shift_solution(x, y, a, b, -(maxy - y) / a);
36     if (y > maxy)
37         shift_solution(x, y, a, b, sign_a);
38     int rx2 = x;
39
40     if (lx2 > rx2)
41         swap(lx2, rx2);
42     int lx = max(lx1, lx2);
43     int rx = min(rx1, rx2);
44
45     if (lx > rx)
46         return 0;
47     return (rx - lx) / abs(b) + 1;
48 }

```

8.21 Discrete Log

```

1 // Returns minimum x for which a ^ x % m = b % m.
2 int solve(int a, int b, int m) {
3     a %= m, b %= m;
4     int k = 1, add = 0, g;
5     while ((g = gcd(a, m)) > 1) {
6         if (b == k)
7             return add;

```

```

8     if (b % g)
9         return -1;
10    b /= g, m /= g, ++add;
11    k = (k * 1ll * a / g) % m;
12 }
13
14 int n = sqrt(m) + 1;
15 int an = 1;
16 for (int i = 0; i < n; ++i)
17     an = (an * 1ll * a) % m;
18
19 unordered_map<int, int> vals;
20 for (int q = 0, cur = b; q <= n; ++q) {
21     vals[cur] = q;
22     cur = (cur * 1ll * a) % m;
23 }
24
25 for (int p = 1, cur = k; p <= n; ++p) {
26     cur = (cur * 1ll * an) % m;
27     if (vals.count(cur)) {
28         int ans = n * p - vals[cur] + add;
29         return ans;
30     }
31 }
32 return -1;
33 }

```

9 Polynomials

9.1 FFT

```

1 using cd = complex<double>;
2 const double PI = acos(-1);
3
4 int reverse(int num, int lg_n) {
5     int res = 0;
6     for (int i = 0; i < lg_n; i++) {
7         if (num & (1 << i))
8             res |= 1 << (lg_n - 1 - i);
9     }
10    return res;
11 }
12
13 void fft(vector<cd> & a, bool invert) {
14     int n = a.size();
15     int lg_n = 0;
16     while ((1 << lg_n) < n)
17         lg_n++;

```

```

18
19     for (int i = 0; i < n; i++) {
20         if (i < reverse(i, lg_n))
21             swap(a[i], a[reverse(i, lg_n)]);
22     }
23
24     for (int len = 2; len <= n; len <= 1) {
25         double ang = 2 * PI / len * (invert ? -1 : 1);
26         cd wlen(cos(ang), sin(ang));
27         for (int i = 0; i < n; i += len) {
28             cd w(1);
29             for (int j = 0; j < len / 2; j++) {
30                 cd u = a[i+j], v = a[i+j+len/2] * w;
31                 a[i+j] = u + v;
32                 a[i+j+len/2] = u - v;
33                 w *= wlen;
34             }
35         }
36     }
37
38     if (invert) {
39         for (cd & x : a)
40             x /= n;
41     }
42 }
43
44 vector<int> multiply(vector<int> const& a, vector<int> const& b) {
45     vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
46     int n = 1;
47     while (n < a.size() + b.size())
48         n <= 1;
49     fa.resize(n);
50     fb.resize(n);
51
52     fft(fa, false);
53     fft(fb, false);
54     for (int i = 0; i < n; i++)
55         fa[i] *= fb[i];
56     fft(fa, true);
57
58     vector<int> result(n);
59     for (int i = 0; i < n; i++)
60         result[i] = round(fa[i].real());
61     return result;
62 }
63
64 // Normalization
65
66 int carry = 0;
67 for (int i = 0; i < n; i++){
68     result[i] += carry;

```

```

69     carry = result[i] / 10;
70     result[i] %= 10;
71 }

```

9.2 NTT

```

1  const int mod = 7340033;
2  const int root = 5;
3  const int root_1 = 4404020;
4  const int root_pw = 1 << 20;
5
6  void fft(vector<int> &a, bool invert) {
7      int n = 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         int wlen = invert ? root_1 : root;
21         for (int i = len; i < root_pw; i <= 1)
22             wlen = (int)(1LL * wlen * wlen % mod);
23
24         for (int i = 0; i < n; i += len) {
25             int w = 1;
26             for (int j = 0; j < len / 2; j++) {
27                 int u = a[i+j], v = (int)(1LL * a[i+j+len/2] * w %
28                     mod);
29                 a[i+j] = u + v < mod ? u + v : u + v - mod;
30                 a[i+j+len/2] = u - v >= 0 ? u - v : u - v + mod;
31                 w = (int)(1LL * w * wlen % mod);
32             }
33         }
34
35         if (invert) {
36             int n_1 = inverse(n, mod);
37             for (int &x : a)
38                 x = (int)(1LL * x * n_1 % mod);
39         }
40     }

```

9.3 Berlekamp Messey

```

1  vector<T> berlekampMassey(const vector<T> &s) {
2      vector<T> c; // the linear recurrence sequence we are
3          building
4      vector<T> oldC; // the best previous version of c to use (the
5          one with the rightmost left endpoint)
6      int f = -1; // the index at which the best previous
7          version of c 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
15             we 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
23             the base case
24             c.resize(i + 1);
25             mt19937 rng(chrono::steady_clock::now().
26                 time_since_epoch().count());
27             for (T &x : c)
28                 x = rng(); // just to prove that the initial
29                 values don't matter in the first step, I will
30                 set to random values
31             f = i;
32         } else {
33             // we need to use a previous version of c to improve
34             on this 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             // 3. insert a 1 on the left
42             d.insert(d.begin(), 1);
43             // 4. multiply the sequence by delta / d(f + 1)
44             T df1 = 0; // d(f + 1)
45             for (int j=1; j<=(int)d.size(); j++)
46                 df1 += d[j-1] * s[f+1-j];
47             assert(df1 != 0);
48             T coef = delta / df1; // storing this in outer
49                 variable so it's O(n^2) instead of O(n^2 log MOD)

```

```

40     for (T &x : d)
41         x *= coef;
42     // 5. insert i - f - 1 zeros on the left
43     vector<T> zeros(i - f - 1);
44     zeros.insert(zeros.end(), d.begin(), d.end());
45     d = zeros;
46     // now we have our new recurrence: c + d
47     vector<T> temp = c; // save the last version of c
48     // because it might have a better left endpoint
49     c.resize(max(c.size(), d.size()));
50     for (int j=0; j<(int)d.size(); j++)
51         c[j] += d[j];
52     // finally, let's consider updating oldC
53     if (i - (int) temp.size() > f - (int) oldC.size()) {
54         // better left endpoint, let's update!
55         oldC = temp;
56         f = i;
57     }
58 }
59 return c;
60 }

```

10 Linear Algebra

10.1 Determinant of a Matrix

```

1 const double EPS = 1E-9;
2 int n;
3 vector < vector<double> > a (n, vector<double> (n));
4
5 double det = 1;
6 for (int i=0; i<n; ++i) {
7     int k = i;
8     for (int j=i+1; j<n; ++j)
9         if (abs (a[j][i]) > abs (a[k][i]))
10             k = j;
11     if (abs (a[k][i]) < EPS) {
12         det = 0;
13         break;
14     }
15     swap (a[i], a[k]);
16     if (i != k)
17         det = -det;
18     det *= a[i][i];
19     for (int j=i+1; j<n; ++j)
20         a[i][j] /= a[i][i];
21     for (int j=0; j<n; ++j)

```

```

22         if (j != i && abs (a[j][i]) > EPS)
23             for (int k=i+1; k<n; ++k)
24                 a[j][k] -= a[i][k] * a[j][i];
25     }
26
27     cout << det;

```

10.2 Rank of a Matrix

```

1 const double EPS = 1E-9;
2
3 int compute_rank(vector<vector<double>> A) {
4     int n = A.size();
5     int m = A[0].size();
6
7     int rank = 0;
8     vector<bool> row_selected(n, false);
9     for (int i = 0; i < m; ++i) {
10         int j;
11         for (j = 0; j < n; ++j) {
12             if (!row_selected[j] && abs(A[j][i]) > EPS)
13                 break;
14         }
15
16         if (j != n) {
17             ++rank;
18             row_selected[j] = true;
19             for (int p = i + 1; p < m; ++p)
20                 A[j][p] /= A[j][i];
21             for (int k = 0; k < n; ++k) {
22                 if (k != j && abs(A[k][i]) > EPS) {
23                     for (int p = i + 1; p < m; ++p)
24                         A[k][p] -= A[j][p] * A[k][i];
25                 }
26             }
27         }
28     }
29     return rank;
30 }

```

10.3 Gauss-Jordan

```

1 const double EPS = 1e-9;
2 const int INF = 2; // it doesn't actually have to be infinity or a
3                     // big number
4 int gauss (vector < vector<double> > a, vector<double> & ans) {
5     int n = (int) a.size();

```

```

6   int m = (int) a[0].size() - 1;
7
8   vector<int> where (m, -1);
9   for (int col=0; row=0; col<m && row<n; ++col) {
10      int sel = row;
11      for (int i=row; i<n; ++i)
12         if (abs (a[i][col]) > abs (a[sel][col]))
13            sel = i;
14      if (abs (a[sel][col]) < EPS)
15         continue;
16      for (int i=col; i<=m; ++i)
17         swap (a[sel][i], a[row][i]);
18      where[col] = row;
19
20      for (int i=0; i<n; ++i)
21         if (i != row) {
22            double c = a[i][col] / a[row][col];
23            for (int j=col; j<=m; ++j)
24               a[i][j] -= a[row][j] * c;
25         }
26      ++row;
27   }
28
29   ans.assign (m, 0);
30   for (int i=0; i<m; ++i)
31      if (where[i] != -1)
32         ans[i] = a[where[i]][m] / a[where[i]][i];
33   for (int i=0; i<n; ++i) {
34      double sum = 0;
35      for (int j=0; j<m; ++j)
36         sum += ans[j] * a[i][j];
37      if (abs (sum - a[i][m]) > EPS)
38         return 0;
39   }
40
41   for (int i=0; i<m; ++i)
42      if (where[i] == -1)
43         return INF;
44   return 1;
45 }

```

10.4 Matrix Exponentiation

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 using ll = long long;
5
6 const ll MOD = 1e9 + 7;

```

```

7
8 using Matrix = array<array<ll, 2>, 2>;
9
10 Matrix mul(Matrix a, Matrix b) {
11     Matrix res = {{0, 0}, {0, 0}};
12     for (int i = 0; i < 2; i++) {
13         for (int j = 0; j < 2; j++) {
14             for (int k = 0; k < 2; k++) {
15                 res[i][j] += a[i][k] * b[k][j];
16                 res[i][j] %= MOD;
17             }
18         }
19     }
20
21     return res;
22 }
23
24 int main() {
25     ll n;
26     cin >> n;
27
28     Matrix base = {{1, 0}, {0, 1}};
29     Matrix m = {{1, 1}, {1, 0}};
30
31     for (; n > 0; n /= 2, m = mul(m, m)) {
32         if (n & 1) base = mul(base, m);
33     }
34
35     cout << base[0][1];
36 }

```

11 Geometry

11.1 Line Segment Intersection

```

1 // BeginCodeSnip{Point Class}
2 struct Point {
3     int x, y;
4     Point(int a = 0, int b = 0) : x(a), y(b) {}
5
6     friend istream &operator>>(istream &in, Point &p) {
7         int x, y;
8         in >> p.x >> p.y;
9         return in;
10    }
11 };
12 // EndCodeSnip
13

```

```

14 int sign(long long num) {
15     if (num < 0) {
16         return -1;
17     } else if (num == 0) {
18         return 0;
19     } else {
20         return 1;
21     }
22 }
23
24 long long trigonometric_sense(Point p, Point p1, Point p2) {
25     return sign(1LL * (p1.x - p.x) * (p2.y - p.y) -
26               1LL * (p2.x - p.x) * (p1.y - p.y));
27 }
28
29 // Check if the rectangles with [P1, P2] and [P3, P4] as diagonals
30 // intersect
31 bool quick_check(Point p1, Point p2, Point p3, Point p4) {
32     int x1, x2, x3, x4, y1, y2, y3, y4;
33     x1 = min(p1.x, p2.x), x2 = max(p1.x, p2.x);
34     y1 = min(p1.y, p2.y), y2 = max(p1.y, p2.y);
35     x3 = min(p3.x, p4.x), x4 = max(p3.x, p4.x);
36     y3 = min(p3.y, p4.y), y4 = max(p3.y, p4.y);
37     return x2 < x3 || x4 < x1 || y2 < y3 || y4 < y1;
38 }
39
40 bool check(Point p1, Point p2, Point p3, Point p4) {
41     if (trigonometric_sense(p1, p2, p3) * trigonometric_sense(p1,
42     p2, p4) > 0) {
43         return false;
44     }
45     if (trigonometric_sense(p3, p4, p1) * trigonometric_sense(p3,
46     p4, p2) > 0) {
47         return false;
48     }
49     return true;
50 }
51
52 int main() {
53     int test_num;
54     cin >> test_num;
55     for (int t = 0; t < test_num; t++) {
56         Point p1, p2, p3, p4;
57         cin >> p1 >> p2 >> p3 >> p4;
58
59         if (quick_check(p1, p2, p3, p4)) {
60             cout << "NO" << endl;
61         } else if (check(p1, p2, p3, p4)) {
62             cout << "YES" << endl;
63         } else {
64             cout << "NO" << endl;
65         }
66     }
67 }

```

```

62     }
63 }
64 }

```

11.2 Minimum Euclidian Distance

```

1 const ll mod=1e9+7;
2 const ll MAX=8e18;
3 const ll limit=1e9+1;
4 //ascii https://elcodigoascii.com.ar/
5
6 ll distance(point a, point b) {
7     return (a.X-b.X)*(a.X-b.X)+(a.Y-b.Y)*(a.Y-b.Y);
8 }
9
10 inline void solve()
11 {
12     int n; cin >> n;
13     vector<point> sortedX(n);
14     set<point> sortedY;
15     FO(i, n)
16     {
17         ll x, y; cin >> x >> y;
18         sortedX[i] = make_pair(x, y);
19     }
20     sort(all(sortedX));
21     sortedY.insert(make_pair(sortedX[0].Y, sortedX[0].X));
22     ll d, minSquare = MAX;
23     int j = 0;
24     FOR(i, 1, n)
25     {
26         d = ceil(sqrt(minSquare));
27         while(sortedX[i].X - sortedX[j].X > d)
28         {
29             sortedY.erase(make_pair(sortedX[j].Y, sortedX[j].X));
30             j++;
31         }
32         auto lower = sortedY.lower_bound(make_pair(sortedX[i].Y - d, 0));
33         auto upper = sortedY.upper_bound(make_pair(sortedX[i].Y + d, 0));
34         for(auto pointer = lower; pointer != upper; pointer++)
35         {
36             minSquare = min(minSquare, distance(*pointer, make_pair(
37             sortedX[i].Y, sortedX[i].X)));
38         }
39         sortedY.insert(make_pair(sortedX[i].Y, sortedX[i].X));
40     }
41     cout << minSquare << endl;
42 }

```

41 }

11.3 Point in polygon

```

1 struct point{
2     ll x,y;
3     void show(){
4         cout<<x<<" "<<y<<endl;
5     }
6 };
7
8 int sign(ll a){
9     if(a<0) return -1;
10    if(a==0) return 0;
11    if(a>0) return 1;
12 }
13
14 int signCP(point p,point p1,point p2)
15 {
16     return sign(1LL*((p1.x-p.x)*(p2.y-p.y)-(p1.y-p.y)*(p2.x-p.x)))
17     ;
18 }
19
20 bool intersect(point n, point m,point a,point b)
21 {
22     if(signCP(n,a,b)*signCP(m,a,b)>0) return false;
23     if(signCP(a,n,m)*signCP(b,n,m)>0) return false;
24     return true;
25 }
26
27 bool inside(point a,point b,point c){
28     return a.x>=min(b.x,c.x) && a.x<=max(b.x,c.x) && a.y>=min(b.y,
29         c.y)
30     && a.y<=max(b.y,c.y);
31 }
32
33 inline void solve()
34 {
35     int n,m; cin>>n>>m;
36     vector<point> vertices(n);
37     FO(i,n)
38     {
39         cin>>vertices[i].x>>vertices[i].y;
40     }
41     point query,par,init,first,second;
42     int counter;
43     int resta=0;
44     FO(i,m)
45     {

```

```

44     resta=0;
45     counter=0;
46     cin>>query.x>>query.y;
47     par.x=query.x;
48     par.y=-MAX-1;
49     init.x=vertices[0].x;
50     init.y=vertices[0].y;
51     first.x=init.x;
52     first.y=init.y;
53     bool ver=false;
54     for(int j=1;j<=n;j++)
55     {
56         second.x=vertices[j%n].x;
57         second.y=vertices[j%n].y;
58         point AB,u;
59         AB.x=second.x-first.x;
60         AB.y=second.y-first.y;
61         u.x=second.x-query.x;
62         u.y=second.y-query.y;
63         if((AB.x*u.y-AB.y*u.x)==0 && inside(query,first,second)
64             ){
65             cout<<"BOUNDARY"<<endl;
66             ver=true;
67             break;
68         }
69         if(intersect(query,par,first,second) && first.x<=query
70             .x && query.x<second.x)
71         {
72             counter++;
73         }
74         if(intersect(query,par,first,second) && second.x<=
75             query.x && query.x<first.x){
76             counter++;
77         }
78         first.x=second.x;
79         first.y=second.y;
80     }
81     point AB,u;
82     AB.x=init.x-first.x;
83     AB.y=init.y-first.y;
84     u.x=init.x-query.x;
85     u.y=init.y-query.y;
86     if(!ver){
87         //if(intersect(query,par,first,init)) counter++;
88         if((counter)&1) cout<<"INSIDE";
89         else cout<<"OUTSIDE";
90         cout<<endl;
91     }
92 }
93 }

```


11.4 Point Location Test

```

1 struct point{
2     double x,y;
3 };
4
5 struct Vector{
6     double a=0,b=0;
7     void getVector(point p1,point p2){
8         a=p2.x-p1.x;
9         b=p2.y-p1.y;
10    }
11
12    double getModulo(){
13        return pow(a*a+b*b,0.5);
14    }
15
16    Vector getUnitarian(){
17        Vector x;
18        x.a=a/getModulo();
19        x.b=b/getModulo();
20        //cout<<x.a<<" "<<x.b<<endl;
21        return x;
22    }
23 };
24
25 double dotProduct(Vector x,Vector y)
26 {
27     return x.a*y.a+x.b*y.b;
28 }
29
30 double CrossProduct(Vector x,Vector y)
31 {
32     return x.a*y.b-x.b*y.a;
33 }
34
35 inline void solve()
36 {
37
38     point p1,p2,p3,p4;
39     cin>>p1.x>>p1.y>>p2.x>>p2.y>>p3.x>>p3.y;
40     Vector u,v,t;
41     u.getVector(p1,p3);
42     //cout<<u.a<<" "<<u.b<<endl;
43     v.getVector(p2,p3);
44     if(CrossProduct(u,v)>0) cout<<"LEFT"<<endl;
45     else if(CrossProduct(u,v)<0) cout<<"RIGHT"<<endl;
46     else cout<<"TOUCH"<<endl;
47 }
48
49 }
```

11.5 Polygon Area

```

1 struct point{
2     ll x,y;
3 };
4
5 ll CrossP(point a,point b){
6     return a.x*b.y-a.y*b.x;
7 }
8
9 inline void solve()
10 {
11     int n; cin>>n;
12     ll res=0;
13     point p1,p2,p3;
14     cin>>p3.x>>p3.y;
15     p1.x=p3.x;
16     p1.y=p3.y;
17     F0(i,n-1)
18     {
19         cin>>p2.x>>p2.y;
20         res+=CrossP(p1,p2);
21         p1.x=p2.x;
22         p1.y=p2.y;
23     }
24     res+=CrossP(p1,p3);
25     cout<<abs(res)<<endl;
26 }
```

11.6 Convex Hull

```

1 const ll mod=1e9+7;
2 const ll limit=4e9;
3 //ascii https://elcodigoascii.com.ar/
4
5 int orientation(point a,point b,point c){
6     ll ori=(b.y-c.y)*(b.x-a.x)-(b.y-a.y)*(b.x-c.x);
7     if(ori==0) return 0;
8     if(ori>0) return 1;
9     return 2;
10 }
11
12 void getLastTwo(point &a,point &b,stack<point> &s)
13 {
14     a=s.top();
15     s.pop();
16 }
```

```

16     b=s.top();
17     s.pop();
18 }
19
20 void show(point a){
21     cout<<a.x<<" "<<a.y<<endl;
22 }
23
24 //Graham scan
25
26 void solve(){
27     int n; cin>>n;
28     vector<point> puntos(n);
29     FO(i,n){
30         ll a,b; cin>>a>>b;
31         puntos[i]=make_pair(a,b);
32     }
33     sort(all(puntos));
34     //Lower Part
35     stack<point> lower;
36     FO(i,n)
37     {
38         if(lower.size()<2){
39             lower.push(puntos[i]);
40             continue;
41         }
42         point a,b;
43         getLastTwo(a,b,lower);
44         if(orientation(a,b,puntos[i])<2)
45         {
46             lower.push(b);
47             lower.push(a);
48             lower.push(puntos[i]);
49         }
50         else{
51             lower.push(b);
52             i--;
53         }
54     }
55     stack<point> upper;
56     for(int i=n-1;i>=0;i--)
57     {
58         if(upper.size()<2){
59             upper.push(puntos[i]);
60             continue;
61         }
62         point a,b;
63         getLastTwo(a,b,upper);
64         if(orientation(a,b,puntos[i])<2)
65         {
66             upper.push(b);

```

```

67             upper.push(a);
68             upper.push(puntos[i]);
69         }
70         else{
71             upper.push(b);
72             i++;
73         }
74     }
75
76     set<point> res;
77
78     while(!lower.empty()){
79         res.insert(lower.top());
80         lower.pop();
81     }
82     while(!upper.empty()){
83         res.insert(upper.top());
84         upper.pop();
85     }
86     cout<<res.size()<<endl;
87     for(auto c:res) show(c);
88 }

```

11.7 Complex point

```

1 typedef double T;
2 typedef complex<T> pt;
3 #define x real()
4 #define y imag()
5
6 typedef long long ll;
7 typedef vector<int> vec;
8 const ll mod=1e9+7;
9 const int MAX=2e5+3;
10
11 //ascii https://elcodigoascii.com.ar/
12
13 T norma(pt a){return a.x*a.x+a.y*a.y;}
14
15
16 int sgn(T X){
17     return (T(0)<X)-(T(0)>X);
18 }
19
20
21 pt translate(pt a,pt v){return a+v;}
22 pt scale(pt p,pt c,T factor){return c+(p-c)*factor;}
23 pt rot(pt p,T a){return p*polar(1.0,a);}
24 pt perp(pt p){return pt({-p.y,p.x});}

```

```

25 pt linearFunc(pt p,pt q,pt r,pt fp,pt fq){
26     return fp+(r-p)*(fq-fp)/(q-p);
27 }
28 T dot(pt v,pt w){ return v.x*w.x+v.y*w.y;}
29 T cross(pt v,pt w){ return v.x*w.y-v.y*w.x;}
30
31 bool isperp(pt a,pt b){return dot(a,b)==0;}
32
33 double angle(pt v,pt w){
34     return acos(clamp(dot(v,w)/abs(v)/abs(w),-1.0,-1.0));
35 }
36
37 T orientation(pt a,pt b,pt c){return cross(b-a,c-a);}
38
39 bool inAngle(pt a,pt b,pt c,pt p){
40     if(orientation(a,b,c)<0) swap(b,c);
41     return sgn(orientation(a,b,p))*sgn(orientation(a,c,p))<=0;
42 }
43
44 bool isconvex(vector<pt> p){
45     bool hasPos=false,hasNeg=false;
46     for(int i=0,n=p.size();i<n;i++){
47         int o=orientation(p[i],p[(i+1)%n],p[(i+2)%n]);
48         if(o>0) hasPos=true;
49         if(o<0) hasNeg=true;
50     }
51     return !(hasPos && hasNeg);
52 }
53
54 inline void solve()
55 {
56     pt p{3,-4};
57     p+=pt({1,2});
58     cout<<p<<endl;
59     cout<<norma(p)<<endl;
60 }

```

11.8 Polar sort

```

1 #define x real()
2 #define y imag()
3
4 typedef long long ll;
5 typedef double T;
6 typedef complex<T> pt;
7 typedef vector<int> vec;
8 const ll mod=1e9+7;
9 const int MAX=2e5+3;
10

```

```

11 T cross(pt v,pt w){ return v.x*w.y-v.y*w.x;}
12 T norma(pt a){return a.x*a.x+a.y*a.y;}
13 //ascii https://elcodigoascii.com.ar/
14
15 bool half(pt p){
16     assert(p.x!=0 || p.y!=0);
17     return p.y>0 || (p.y==0 && p.x<0);
18 }
19
20 void polarSort(vector<pt> &v){
21     sort(all(v),[](pt v,pt w){
22         return make_tuple(half(v),0)<make_tuple(half(w),cross(v,w));
23     });
24 }
25
26 void polarSortNorm(vector<pt> &v){
27     sort(all(v),[](pt v,pt w){
28         return make_tuple(half(v),0,norma(v))<make_tuple(half(w),cross(v,w),norma(w));
29     });
30 }
31 inline void solve()
32 {
33
34 }

```

12 Strings

12.1 Marranadas de Quique

```

1
2 //To Upper and Lower
3 transform(s.begin(), s.end(), s.begin(), ::toupper);
4 transform(s.begin(), s.end(), s.begin(), ::tolower);
5
6 // From i to the end
7 string a = s.substr(i);
8 // From i to j
9 string a = s.substr(i,j);
10
11 int a;
12 int b;
13 int c;
14 char comma;
15 char colon;
16
17 // Create a stringstream object

```

```

18 stringstream ss(fullString);
19 // Extract the strings
20 ss >> a >> colon >> b >> comma >> c;

```

12.2 KMP Algorithm

```

1 // LPS for s, lps[i] could also be defined as the longest prefix
  // which is also a proper suffix
2 vi computeLPS(string s){
3     size_t len = 0;
4     size_t M = s.size();
5     vi lps(M, 0);
6
7     size_t i = 1;
8     while(i < M) {
9         if( s[i] == s[len]){
10             len++;
11             lps[i] = len;
12             i++;
13         } else {
14             if(len != 0){
15                 len = lps[len-1];
16             } else {
17                 lps[i] = 0;
18                 i++;
19             }
20         }
21     }
22
23     return lps;
24 }
25
26 // Get number of occurrences of a pattern in a text using KMP
27 // O(N+M)
28 size_t KMPOccurrences(string pattern, string text){
29     vi lps = computeLPS(pattern); // LPS array
30
31     size_t M = pattern.size();
32     size_t N = text.size();
33
34     size_t i = 0; // Index for text
35     size_t j = 0; // Index for pattern
36
37     size_t cnt = 0; // Counter
38
39     while ((N - i) >= (M - j)) {
40         // Watch for the pattern
41         if (pattern[j] == text[i]) {
42             j++;

```

```

43             i++;
44         }
45
46         // If the full match found
47         if (j == M) {
48             cnt++;
49             j = lps[j - 1];
50         }
51
52         // Mismatch after j matches
53         else if (i < N && pattern[j] != text[i]) {
54             // Do not match lps[0..lps[j-1]] characters,
55             // they will match anyway
56             if (j != 0)
57                 j = lps[j - 1];
58             else
59                 i++;
60         }
61     }
62
63     return cnt;
64 }

```

12.3 Rolling Hash

```

1 // Rolling hash
2 struct Hash {
3     // Prime number and modulo
4     long long p = 31, m = 1e9 + 7;
5     long long hash_value;
6     Hash(const string& s)
7     {
8         long long hash_so_far = 0;
9         long long p_pow = 1;
10        const long long n = s.length();
11        for (long long i = 0; i < n; ++i) {
12            hash_so_far
13                = (hash_so_far + (s[i] - 'a' + 1) * p_pow)
14                  % m;
15            p_pow = (p_pow * p) % m;
16        }
17        hash_value = hash_so_far;
18    }
19    bool operator==(const Hash& other)
20    {
21        return (hash_value == other.hash_value);
22    }
23 };
24

```

```

25 // Usage
26 int main(){
27     string s = "hello";
28
29     return 0;
30 }

```

12.4 Hash marrano

```

1 vector<vector<int>> group_identical_strings(vector<string> const&
  s) {
2     int n = s.size();
3     vector<pair<long long, int>> hashes(n);
4     for (int i = 0; i < n; i++)
5         hashes[i] = {compute_hash(s[i]), i};
6
7     sort(hashes.begin(), hashes.end());
8
9     vector<vector<int>> groups;
10    for (int i = 0; i < n; i++) {
11        if (i == 0 || hashes[i].first != hashes[i-1].first)
12            groups.emplace_back();
13        groups.back().push_back(hashes[i].second);
14    }
15    return groups;
16 }

```

12.5 Suffix Array

```

1 // Structure to store information of a suffix
2 struct suffix
3 {
4     int index;
5     char *suff;
6 };
7
8 // A comparison function used by sort() to compare two suffixes
9 int cmp(struct suffix a, struct suffix b)
10 {
11     return strcmp(a.suff, b.suff) < 0? 1 : 0;
12 }
13
14 // This is the main function that takes a string 'txt' of size n
15 // as an
16 // argument, builds and return the suffix array for the given
17 // string
18 int *buildSuffixArray(char *txt, int n)
19 {

```

```

18 // A structure to store suffixes and their indexes
19 struct suffix suffixes[n];
20
21 // Store suffixes and their indexes in an array of structures.
22 // The structure is needed to sort the suffixes alphabetically
23 // and maintain their old indexes while sorting
24 for (int i = 0; i < n; i++)
25 {
26     suffixes[i].index = i;
27     suffixes[i].suff = (txt+i);
28 }
29
30 // Sort the suffixes using the comparison function
31 // defined above.
32 sort(suffixes, suffixes+n, cmp);
33
34 // Store indexes of all sorted suffixes in the suffix array
35 int *suffixArr = new int[n];
36 for (int i = 0; i < n; i++)
37     suffixArr[i] = suffixes[i].index;
38
39 // Return the suffix array
40 return suffixArr;
41 }
42
43 // A utility function to print an array of given size
44 void printArr(int arr[], int n)
45 {
46     for(int i = 0; i < n; i++)
47         cout << arr[i] << " ";
48     cout << endl;
49 }

```

12.6 LCP

```

1 // Structure to store information of a suffix
2 struct suffix
3 {
4     int index; // To store original index
5     int rank[2]; // To store ranks and next rank pair
6 };
7
8 // A comparison function used by sort() to compare two suffixes
9 // Compares two pairs, returns 1 if first pair is smaller
10 int cmp(struct suffix a, struct suffix b)
11 {
12     return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0)
13         :
14         (a.rank[0] < b.rank[0] ?1: 0);

```

```

14 }
15
16 // This is the main function that takes a string 'txt' of size n
17 // as an
18 // argument, builds and return the suffix array for the given
19 // string
20 vector<int> buildSuffixArray(string txt, int n)
21 {
22     // A structure to store suffixes and their indexes
23     struct suffix suffixes[n];
24
25     // Store suffixes and their indexes in an array of structures.
26     // The structure is needed to sort the suffixes alphabetically
27     // and maintain their old indexes while sorting
28     for (int i = 0; i < n; i++)
29     {
30         suffixes[i].index = i;
31         suffixes[i].rank[0] = txt[i] - 'a';
32         suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
33     }
34
35     // Sort the suffixes using the comparison function
36     // defined above.
37     sort(suffixes, suffixes+n, cmp);
38
39     // At this point, all suffixes are sorted according to first
40     // 2 characters. Let us sort suffixes according to first 4
41     // characters, then first 8 and so on
42     int ind[n]; // This array is needed to get the index in
43     // suffixes[]
44     // from original index. This mapping is needed to get
45     // next suffix.
46     for (int k = 4; k < 2*n; k = k*2)
47     {
48         // Assigning rank and index values to first suffix
49         int rank = 0;
50         int prev_rank = suffixes[0].rank[0];
51         suffixes[0].rank[0] = rank;
52         ind[suffixes[0].index] = 0;
53
54         // Assigning rank to suffixes
55         for (int i = 1; i < n; i++)
56         {
57             // If first rank and next ranks are same as that of
58             // previous
59             // suffix in array, assign the same new rank to this
60             // suffix
61             if (suffixes[i].rank[0] == prev_rank &&
62                 suffixes[i].rank[1] == suffixes[i-1].rank[1])
63             {
64                 prev_rank = suffixes[i].rank[0];

```

```

60         suffixes[i].rank[0] = rank;
61     }
62     else // Otherwise increment rank and assign
63     {
64         prev_rank = suffixes[i].rank[0];
65         suffixes[i].rank[0] = ++rank;
66     }
67     ind[suffixes[i].index] = i;
68 }
69
70 // Assign next rank to every suffix
71 for (int i = 0; i < n; i++)
72 {
73     int nextindex = suffixes[i].index + k/2;
74     suffixes[i].rank[1] = (nextindex < n)?
75         suffixes[ind[nextindex]].rank
76         [0]: -1;
77 }
78
79 // Sort the suffixes according to first k characters
80 sort(suffixes, suffixes+n, cmp);
81 }
82
83 // Store indexes of all sorted suffixes in the suffix array
84 vector<int> suffixArr;
85 for (int i = 0; i < n; i++)
86     suffixArr.push_back(suffixes[i].index);
87
88 // Return the suffix array
89 return suffixArr;
90 }
91
92 /* To construct and return LCP */
93 vector<int> kasai(string txt, vector<int> suffixArr)
94 {
95     int n = suffixArr.size();
96
97     // To store LCP array
98     vector<int> lcp(n, 0);
99
100     // An auxiliary array to store inverse of suffix array
101     // elements. For example if suffixArr[0] is 5, the
102     // invSuff[5] would store 0. This is used to get next
103     // suffix string from suffix array.
104     vector<int> invSuff(n, 0);
105
106     // Fill values in invSuff[]
107     for (int i=0; i < n; i++)
108         invSuff[suffixArr[i]] = i;
109
110     // Initialize length of previous LCP

```

```

110     int k = 0;
111
112     // Process all suffixes one by one starting from
113     // first suffix in txt[]
114     for (int i=0; i<n; i++)
115     {
116         /* If the current suffix is at n-1, then we dont
117         have next substring to consider. So lcp is not
118         defined for this substring, we put zero. */
119         if (invSuff[i] == n-1)
120         {
121             k = 0;
122             continue;
123         }
124
125         /* j contains index of the next substring to
126         be considered to compare with the present
127         substring, i.e., next string in suffix array */
128         int j = suffixArr[invSuff[i]+1];
129
130         // Directly start matching from k'th index as
131         // at-least k-1 characters will match
132         while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
133             k++;
134
135         lcp[invSuff[i]] = k; // lcp for the present suffix.
136
137         // Deleting the starting character from the string.
138         if (k>0)
139             k--;
140     }
141
142     // return the constructed lcp array
143     return lcp;
144 }
145
146 // Utility function to print an array
147 void printArr(vector<int>arr, int n)
148 {
149     for (int i = 0; i < n; i++)
150         cout << arr[i] << " ";
151     cout << endl;
152 }

```

12.7 Z Function

```

1 vector<int> z_function(string s) {
2     int n = s.size();
3     vector<int> z(n);

```

```

4     int l = 0, r = 0;
5     for(int i = 1; i < n; i++) {
6         if(i < r) {
7             z[i] = min(r - i, z[i - l]);
8         }
9         while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
10             z[i]++;
11         }
12         if(i + z[i] > r) {
13             l = i;
14             r = i + z[i];
15         }
16     }
17     return z;
18 }

```

12.8 Longest Palindrome

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 const ll mod=1e9+7;
4 const int MAX=1e6+3;
5 vector<int> lps(2*MAX);
6 int n;
7 string s;
8
9 //ascii https://elcodigoascii.com.ar/
10
11 void show(int idx)
12 {
13     int start=(idx-lps[idx])/2;
14     int end=start+lps[idx];
15     for(int i=start;i<end;i++){
16         cout<<s[i];
17     }
18 }
19
20
21 inline void solve()
22 {
23     cin>>s;
24     n=s.size();
25     lps[0]=0;
26     lps[1]=1;
27     int rightCenter,leftCenter=center,curRightCenter,curLeftCenter;
28     ;
29     center=1;
30     rightCenter=center+lps[center];
31     leftCenter=center-lps[center];

```

```

31 int maxLPSCenter=1;
32 int diff=-1;
33 bool exp;
34 for(curRightCenter=2; curRightCenter<2*n+1; curRightCenter++)
35 {
36     //Condicion de cambio de centro
37     curLeftCenter=2*center-curRightCenter;
38     diff=rightCenter-curRightCenter;
39     exp=false;
40     if(diff>=0){
41         if(lps[curLeftCenter]<diff){
42             lps[curRightCenter]=lps[curLeftCenter];
43         }
44         else if(lps[curLeftCenter]==diff && rightCenter==2*n){
45             lps[curRightCenter]=lps[curLeftCenter];
46         }
47         else if(lps[curLeftCenter]==diff && rightCenter<2*n){
48             lps[curRightCenter]=lps[curLeftCenter];
49             exp=true;
50         }
51         else if(lps[curLeftCenter]>diff){
52             lps[curRightCenter]=diff;
53             exp=true;
54         }
55     }
56 }
57 else{
58     lps[curRightCenter]=0;
59     exp=true;
60 }
61 if(exp)
62 {
63     while(((curRightCenter+lps[curRightCenter])<2*n &&
64         curRightCenter-lps[curRightCenter]>0)
65         && ((curRightCenter+lps[curRightCenter]+1)%2==0 || s[(
66         curRightCenter+lps[curRightCenter]+1)/2]==s[(
67         curRightCenter-lps[curRightCenter]-1)/2])){
68         lps[curRightCenter]++;
69     }
70 }
71 if(lps[curRightCenter]>lps[maxLPSCenter])
72 {
73     maxLPSCenter=curRightCenter;
74 }
75 if(curRightCenter+lps[curRightCenter]>rightCenter){
76     center=curRightCenter;
77     rightCenter=curRightCenter+lps[curRightCenter];
78 }
79 show(maxLPSCenter);

```

```

79 }

```

12.9 String Hashing

```

1 typedef long long ll;
2 typedef vector<int> vec;
3 const ll mod=1e9+7;
4 const int MAX=1e6+3;
5 const ll A=911382323;
6 const ll B=972663749;
7 ll str[MAX];
8 ll pk[MAX];
9 bool prefix[MAX]={false};
10
11
12
13 ll subs(int i,int j)
14 {
15     if(i)
16         return ((str[j]-pk[j-i+1]*str[i-1])%B+B)%B;
17     else
18         return str[j];
19 }
20
21 //ascii https://elcodigoascii.com.ar/
22
23 inline void solve()
24 {
25     string s; cin>>s;
26     memset(prefix,true,sizeof(prefix));
27     str[0]=s[0];
28     pk[0]=1;
29     int n=s.size();
30     for(int i=1;i<n;i++)
31     {
32         str[i]=A*str[i-1]+s[i];
33         pk[i]=pk[i-1]*A;
34         pk[i]%=B;
35         str[i]%=B;
36     }
37     ll aux;
38     bool ver;
39     for(int i=1;i<=n;i++)
40     {
41         aux=subs(0,i-1);
42         for(int j=0;j+i<=n;j+=i)
43         {
44             if(aux!=subs(j,j+i-1))
45                 {

```



```

46         //cout<<aux<<" "<<subs(j,j+i-1)<<" "<<i<<" "<<j<<
47         endl;
48         prefix[i]=false;
49         break;
50     }
51     if(!prefix[i]) continue;
52     if(n%i && (subs(n-n%i,n-1)!=subs(0,n%i-1)))
53     {
54         continue;
55     }
56     cout<<i<<" ";
57 }
58 }

```

12.10 Manacher Algorithm

```

1 vector<int> manacher(string s) {
2     string t;
3     for(auto c: s) {
4         t += string("#") + c;
5     }
6     auto res = manacher_odd(t + "#");
7     return vector<int>(begin(res) + 1, end(res) - 1);
8 }

```

12.11 Suffix Automaton

```

1 struct state {
2     int len, link;
3     map<char, int> next;
4 };
5
6 const int MAXLEN = 100000;
7 state st[MAXLEN * 2];
8 int sz, last;
9
10 void sa_init() {
11     st[0].len = 0;
12     st[0].link = -1;
13     sz++;
14     last = 0;
15 }
16
17 void sa_extend(char c) {
18     int cur = sz++;
19     st[cur].len = st[last].len + 1;
20     int p = last;

```

```

21     while (p != -1 && !st[p].next.count(c)) {
22         st[p].next[c] = cur;
23         p = st[p].link;
24     }
25     if (p == -1) {
26         st[cur].link = 0;
27     } else {
28         int q = st[p].next[c];
29         if (st[p].len + 1 == st[q].len) {
30             st[cur].link = q;
31         } else {
32             int clone = sz++;
33             st[clone].len = st[p].len + 1;
34             st[clone].next = st[q].next;
35             st[clone].link = st[q].link;
36             while (p != -1 && st[p].next[c] == q) {
37                 st[p].next[c] = clone;
38                 p = st[p].link;
39             }
40             st[q].link = st[cur].link = clone;
41         }
42     }
43     last = cur;
44 }
45
46 long long get_diff_strings(){
47     long long tot = 0;
48     for(int i = 1; i < sz; i++) {
49         tot += st[i].len - st[st[i].link].len;
50     }
51     return tot;
52 }
53
54 long long get_tot_len_diff_substings() {
55     long long tot = 0;
56     for(int i = 1; i < sz; i++) {
57         long long shortest = st[st[i].link].len + 1;
58         long long longest = st[i].len;
59
60         long long num_strings = longest - shortest + 1;
61         long long cur = num_strings * (longest + shortest) / 2;
62         tot += cur;
63     }
64     return tot;
65 }
66
67 string lcs (string S, string T) {
68     sa_init();
69     for (int i = 0; i < S.size(); i++)
70         sa_extend(S[i]);
71 }

```

```

72  int v = 0, l = 0, best = 0, bestpos = 0;
73  for (int i = 0; i < T.size(); i++) {
74      while (v && !st[v].next.count(T[i])) {
75          v = st[v].link;
76          l = st[v].len;
77      }
78      if (st[v].next.count(T[i])) {
79          v = st[v].next[T[i]];
80          l++;
81      }
82      if (l > best) {
83          best = l;
84          bestpos = i;
85      }
86  }
87  return T.substr(bestpos - best + 1, best);
88 }

```

13 Formulas

13.1 Sums

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

Gauss

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

Gauss squares

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

Cubes

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

Powers of 4

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

13.2 Catalan numbers

$$C_0 = 1, \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad (\text{Recursive})$$

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

- **Valid Parentheses:** Count of balanced parentheses expressions with n pairs.
- **Full Binary Trees:** Structurally unique full binary trees with $n + 1$ leaves.
- **Polygon Triangulation:** Ways to triangulate a convex $(n + 2)$ -gon.
- **Dyck Paths:** Paths from $(0, 0)$ to $(2n, 0)$ that never dip below the x-axis.
- **Non-Crossing Partitions:** Ways to connect $2n$ points on a circle without crossing chords.
- **Stack Permutations:** Valid stack-sortable permutations of length n .
- **Mountain Ranges:** Sequences of $2n$ up/down steps forming valid mountain ranges.
- **Unique BSTs:** Number of distinct binary search trees with n keys.
- **Diagonal-Avoiding Paths:** Paths in a grid from $(0, 0)$ to (n, n) without crossing the diagonal.

13.3 Cayley's Formula

Number of labeled trees of n vertices: n^{n-2} .

Number of rooted forest of n vertices is: $(n+1)^{n-1}$

13.4 Geometric series

Finite:

$$\sum_{k=0}^n ar^k = \begin{cases} a \frac{1-r^{n+1}}{1-r} & \text{if } r \neq 1, \\ a(n+1) & \text{if } r = 1. \end{cases}$$

Infinite:

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r} \quad (\text{converges iff } |r| < 1)$$

13.5 Divisors

The number of divisors of any number n is:

$$\begin{cases} \approx 100 & n < 5 \times 10^4 \\ \approx 500 & n < 1 \times 10^7 \\ \approx 2000 & n < 1 \times 10^{10} \\ \approx 200000 & n < 1 \times 10^{19} \end{cases}$$

13.6 Number of primes between 1 and n

$$\frac{n}{\ln(n)}$$

13.7 Pythagorean triplets

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

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

13.8 Derangments

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

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

14 Miscellaneous

Gus, this is a reminder to add more stuff here

14.1 Random number generator

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

14.2 Kadane's Algorithm

```
1 inline void solve()
2 {
3     int n; cin>>n;
4     vector<int> normal(n);
5     vector<int> rever(n);
6     F0(i,n){
7         cin>>normal[i];
8         rever[i]=-normal[i];
9     }
10    ll sum = 0, max_sum = -1e9;
11    ll sumr=0;
12    for (int i = 0; i < n; i++) {
13        sum += normal[i];
14        max_sum = max(max_sum, sum);
15        sumr+= rever[i];
16        max_sum=max(max_sum,sumr);
17        if(i%2==1){
18            sum=max(sum,sumr);
19            sumr=max(sum,sumr);
20        }
21        if (sum < 0) sum = 0;
22        if (sumr<0) sumr=0;
23    }
24    cout<<max_sum<<endl;
25    //Geeks for geeks
26    //https://www.geeksforgeeks.org/cses-solutions-maximum-
27    subarray-sum/
28 }
```

14.3 Moore's Voting Algorithm

```
1 int majorityElement(vector<int>& nums) {
2     int vote = 0, r = 0;
3     for(int i=0; i<nums.size();i++){
4         if(nums[i] == nums[r])
5             vote++;
6         else
7             vote--;
8         if(vote == 0){
9             r = i;
10            vote = 1;
11        }
12    }
13
14    int cnt = 0;
15    int goal = (nums.size())/2;
16    for(int i=0; i<nums.size(); i++){
```

```
17         if(nums[i] == nums[r]){
18             cnt++;
19             if(cnt > goal){
20                 break;
21             }
22         }
23     }
24
25     return nums[r];
26 }
```

15 Marranadas de C++

15.1 Compilation

g++-13 -std=c++20 name.cpp

15.2 Compiler optimizations

```
1 // Makes bit operations faster
2 #pragma GCC target("popcnt")
3
4 //Auto vectorize for-loops and optimizes floating points (assumes
5 // associativity and turns off denormals)
6 #pragma GCC optimize("Ofast")
7
8 // Doubles performance of vectorized code, crashes in old
9 // computers
10 #pragma GCC target("avx2")
11
12 #pragma GCC optimize("O3,unroll-loops")
13 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```

15.3 Decimal printing

Friendly reminder to use `printf()` with decimals

```
1 cout << fixed << setprecision(n) << endl;
```

15.4 Bit tricks

`x & -x` is the least bit in `x`

`c = x & -x, r = x + c, (((bin_pow(r, x)) >> 2) / c)` OR `r` next number bigger than `x` same number of bits set.