

C++ Competitive Programming Library

DO NOT DISCLOSE OR DISTRIBUTE

bfs.07 - Bernardo Flores Salmeron

1	Template	3	3.12	Divide And Conquer Optimization	21
2	Data Structures	3	3.13	Edit Distance	22
	2.1 Bit2D	3	3.14	Knapsack	22
	2.2 Merge Sort Tree (K-Esimo Maior Elemento Num Intervalo, Valores Maiores Que K Num Intervalo,	4	3.15	Knuth Optimization	22
	2.3 Mos Algorithm	4	3.16	Lis	22
	2.4 Sqrt Decomposition	5	4	Geometry	23
	2.5 Bit	5	4.1	Centro De Massa De Um Poligono	23
	2.6 Bit (Range Update)	6	4.2	Closest Pair Of Points	23
	2.7 Counting Inversions (Minimum Number Of Adjacent Swaps To Sort Array)	6	4.3	Condicao De Existencia De Um Triangulo	23
	2.8 Min Queue	6	4.4	Convex Hull	23
	2.9 Ordered Set	7	4.5	Cross Product	24
	2.10 Persistent Segment Tree	7	4.6	Distance Point Segment	24
	2.11 Segment Tree	8	4.7	Line-Line Intersection	24
	2.12 Segment Tree 2D	10	4.8	Line-Point Distance	24
	2.13 Segment Tree Beats	11	4.9	Point Inside Convex Polygon - Log(N)	25
	2.14 Segment Tree Polynomial	13	4.10	Point Inside Polygon	26
	2.15 Sparse Table	14	4.11	Points Inside And In Boundary Polygon	26
	2.16 Treap	15	4.12	Polygon Area (3D)	27
	2.17 Treap Maximum Contiguous Segment	18	4.13	Polygon Area	27
3	Dp	18	4.14	Segment-Segment Intersection	27
	3.1 Achar Maior Palindromo	18	4.15	Upper And Lower Hull	28
	3.2 Digit Dp	18	4.16	Circle Circle Intersection	28
	3.3 Longest Common Subsequence	19	4.17	Circle Circle Intersection	28
	3.4 Longest Common Substring	19	4.18	Struct Point And Line	29
	3.5 Longest Increasing Subsequence 2D (Not Sorted)	19	5	Graphs	29
	3.6 Longest Increasing Subsequence 2D (Sorted)	20	5.1	All Eulerian Path Or Tour	29
	3.7 Subset Sum Com Bitset	20	5.2	Articulation Points	31
	3.8 Catalan	20	5.3	Bellman Ford	32
	3.9 Catalan 1 1 2 5 14 42 132	20	5.4	Bipartite Check	32
	3.10 Cht Optimization	21	5.5	Block Cut Tree	32
	3.11 Coin Change Problem	21	5.6	Bridges	34
			5.7	Centroid	34

5.8	Centroid Decomposition	34	6.4	Counting Bits	54
5.9	Compress ScCs In Dag	35	6.5	Gen Random Numbers (Rng)	55
5.10	Count (3-4) Cycles	35	6.6	Int To Binary String	55
5.11	Cycle Detection	36	6.7	Int To String	55
5.12	De Bruijn Sequence	36	6.8	Permutation	55
5.13	Diameter In Tree	36	6.9	Print Int128 T	55
5.14	Dijkstra + Dij Graph	36	6.10	Read And Write From File	55
5.15	Dinic	37	6.11	Readint	55
5.16	Dsu	41	6.12	Rotate Left	55
5.17	Dsu On Tree	41	6.13	Rotate Right	55
5.18	Floyd Warshall	42	6.14	Scanf From String	55
5.19	Functional Graph	42	6.15	Split Function	55
5.20	Girth (Shortest Cycle In A Graph)	44	6.16	String To Long Long	55
5.21	Hld	44	6.17	Substring	56
5.22	Hungarian	45	6.18	Time Measure	56
5.23	Kuhn	45	6.19	Unique Vector	56
5.24	Lca	46	6.20	Width	56
5.25	Longest Path In Dag	48	7 Math	56	
5.26	Maximum Independent Set (Set Of Vertices That Arent Directly Connected)	48	7.1	Bell Numbers	56
5.27	Maximum Path Unweighted Graph	48	7.2	Binary Exponentiation	56
5.28	Min Cost Flow Gpresso	48	7.3	Chinese Remainder Theorem	56
5.29	Min Cost Flow Katcl	49	7.4	Combinatorics	57
5.30	Minimum Edge Cover (Set Of Edges That Are Adjacent To All Vertices)	50	7.5	Diophantine Equation	57
5.31	Minimum Path Cover In Dag	50	7.6	Divide Fraction	57
5.32	Minimum Path Cover In Dag	50	7.7	Divisors	58
5.33	Mst	50	7.8	Euler Totient	58
5.34	Number Of Different Spanning Trees In A Complete Graph	51	7.9	Extended Euclidean	58
5.35	Number Of Ways To Make A Graph Connected	51	7.10	Factorization	58
5.36	Pruffer Decode	51	7.11	Fft	58
5.37	Pruffer Encode	51	7.12	Inclusion Exclusion	59
5.38	Pruffer Properties	51	7.13	Inclusion Exclusion	59
5.39	Remove All Bridges From Graph	52	7.14	Karatsuba	59
5.40	Scc (Kosaraju)	52	7.15	Markov Chains	60
5.41	Small To Large (Merge Sets)	52	7.16	Matrix Exponentiation	60
5.42	Topological Sort	53	7.17	Matrix Exponentiation	60
5.43	Tree Diameter	53	7.18	Pollard Rho (Factorize)	60
5.44	Tree Distance	53	7.19	Pollard Rho (Find A Divisor)	62
5.45	Tree Isomorphism	54	7.20	Polynomial Convolution	62
6 Language Stuff	54		7.21	Primality Check	62
6.1	Binary String To Int	54	7.22	Primes	62
6.2	Check Char Type	54	7.23	Sieve + Segmented Sieve	62
6.3	Check Overflow	54	7.24	Stars And Bars	63

8 Miscellaneous

8.1	2-Sat	63
8.2	Interval Scheduling	64
8.3	Interval Scheduling	64
8.4	Sliding Window Minimum	64
8.5	Torre De Hanoi	64
8.6	Counting Frequency Of Digits From 1 To K	64
8.7	Counting Number Of Digits Up To N	64
8.8	Infix To Postfix	65
8.9	Iterate Over Subsets Of Mask	65
8.10	Kadane	65
8.11	Kadane (Segment Tree)	65
8.12	Kadane 2D	66
8.13	Largest Area In Histogram	67
8.14	Modular Integer	67
8.15	Point Compression	68
8.16	Ternary Search	68

9 Stress Testing

9.1	Check	68
9.2	Gen	68
9.3	Run	70

10 Strings

10.1	Trie - Maximum Xor Sum	70
10.2	Trie - Maximum Xor Two Elements	70
10.3	Z-Function	70
10.4	Aho Corasick	70
10.5	Hashing	71
10.6	Kmp	72
10.7	Lcs K Strings	72
10.8	Lexicographically Smallest Rotation	73
10.9	Manacher (Longest Palindrome)	73
10.10	Suffix Array	74
10.11	Suffix Array Mine	75
10.12	Suffix Automaton	76
10.13	Trie	77

1. Template

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define INF (1ll << 62)
6 #define pb push_back
7 #define ii pair<int,int>
8 #define OK cerr <<"OK"<< endl
9 #define debug(x) cerr << #x " = " << (x) << endl
10 #define ff first
11 #define ss second
12 #define int long long
13 #define tt tuple<int, int, int>
14 #define endl '\n'
15
16 signed main () {
17
18     ios_base::sync_with_stdio(false);
19     cin.tie(NULL);
20
21 }

```

2. Data Structures**2.1. Bit2D**

```

1 // INDEX BY ONE ALWAYS!!!
2 class BIT_2D {
3 private:
4     // row, column
5     const int n, m;
6     vector<vector<int>>> tree;
7
8 // Returns an integer which constains only the least significant bit.
9 int low(const int i) { return i & (-i); }
10
11 void _update(const int x, const int y, const int delta) {
12     for (int i = x; i < n; i += low(i))
13         for (int j = y; j < m; j += low(j))
14             tree[i][j] += delta;
15 }
16
17 int _query(const int x, const int y) {
18     int ans = 0;
19     for (int i = x; i > 0; i -= low(i))
20         for (int j = y; j > 0; j -= low(j))
21             ans += tree[i][j];
22     return ans;
23 }
24
25 public:
26 // put the size of the array without 1 indexing.
27 /// Time Complexity: O(n * m)
28 BIT_2D(const int n, const int m) : n(n + 1), m(m + 1) {
29     tree.resize(this->n, vector<int>(this->m, 0));
30 }
31
32 /// Time Complexity: O(n * m * (log(n) + log(m)))
33 BIT_2D(const vector<vector<int>>> &mat)
34     : n(mat.size()), m(mat.front().size()) {
35     // Check if it is 1 indexed.
36 }

```

```

37     assert(mat[0][0] == 0);
38     tree.resize(n, vector<int>(m, 0));
39     for (int i = 1; i < n; i++)
40         for (int j = 1; j < m; j++)
41             _update(i, j, mat[i][j]);
42 }
43
44 /// Query from (1, 1) to (x, y).
45 ///
46 /// Time Complexity: O(log(n) + log(m))
47 int prefix_query(const int x, const int y) {
48     assert(0 < x), assert(x < n);
49     assert(0 < y), assert(y < m);
50     return _query(x, y);
51 }
52
53 /// Query from (x1, y1) to (x2, y2).
54 ///
55 /// Time Complexity: O(log(n) + log(m))
56 int query(const int x1, const int y1, const int x2, const int y2) {
57     assert(0 < x1), assert(x1 <= x2), assert(x2 < n);
58     assert(0 < y1), assert(y1 <= y2), assert(y2 < m);
59     return _query(x2, y2) - _query(x1 - 1, y2) - _query(x2, y1 - 1) +
60         _query(x1 - 1, y1 - 1);
61 }
62
63 /// Updates point (x, y).
64 ///
65 /// Time Complexity: O(log(n) + log(m))
66 void update(const int x, const int y, const int delta) {
67     assert(0 < x), assert(x < n);
68     assert(0 < y), assert(y < m);
69     _update(x, y, delta);
70 }
71 };

```

2.2. Merge Sort Tree (K-Esimo Maior Elemento Num Intervalo, Valores Maiores Que K Num Intervalo,

```

1 // retornar a qtd de números maiores q um numero k numa array de i...j
2 struct Tree {
3     vector<int> vet;
4 };
5 Tree tree[4*(int)3e4];
6 int arr[(int)5e4];
7
8 int query(int l, int r, int i, int j, int k, int pos) {
9     if(l > j || r < i)
10         return 0;
11
12     if(i <= l && r <= j) {
13         auto it = upper_bound(tree[pos].vet.begin(), tree[pos].vet.end(), k);
14         return tree[pos].vet.end() - it;
15     }
16
17     int mid = (l+r)>>1;
18     return query(l, mid, i, j, k, 2*pos+1) + query(mid+1, r, i, j, k, 2*pos+2);
19 }
20
21 void build(int l, int r, int pos) {
22
23     if(l == r) {
24         tree[pos].vet.pb(arr[l]);
25         return;

```

```

26     }
27
28     int mid = (l+r)>>1;
29     build(l, mid, 2*pos+1);
30     build(mid + 1, r, 2*pos+2);
31
32     merge(tree[2*pos+1].vet.begin(), tree[2*pos+1].vet.end(),
33           tree[2*pos+2].vet.begin(), tree[2*pos+2].vet.end(),
34           back_inserter(tree[pos].vet));
35 }

```

2.3. Mos Algorithm

```

1 struct Tree {
2     int l, r, ind;
3 };
4 Tree query[311111];
5 int arr[311111];
6 int freq[111111];
7 int ans[311111];
8 int block = sqrt(n), cont = 0;
9
10 bool cmp(Tree a, Tree b) {
11     if(a.l/block == b.l/block)
12         return a.r < b.r;
13     return a.l/block < b.l/block;
14 }
15
16 void add(int pos) {
17     freq[arr[pos]]++;
18     if(freq[arr[pos]] == 1) {
19         cont++;
20     }
21 }
22
23 void del(int pos) {
24     freq[arr[pos]]--;
25     if(freq[arr[pos]] == 0)
26         cont--;
27 }
28
29 int main () {
30     int n; cin >> n;
31     block = sqrt(n);
32
33     for(int i = 0; i < n; i++) {
34         cin >> arr[i];
35         freq[arr[i]] = 0;
36     }
37
38     int m; cin >> m;
39
40     for(int i = 0; i < m; i++) {
41         cin >> query[i].l >> query[i].r;
42         query[i].l--, query[i].r--;
43         query[i].ind = i;
44     }
45     sort(query, query + m, cmp);
46
47     int s, e;
48     s = e = query[0].l;
49     add(s);
50     for(int i = 0; i < m; i++) {
51         while(s > query[i].l)
52             add(--s);
53         while(s < query[i].l)

```

```

52     del(s++);
53     while(e < query[i].r)
54         add(++e);
55     while(e > query[i].r)
56         del(e--);
57     ans[query[i].ind] = cont;
58
59 }
60 for(int i = 0; i < m; i++)
61     cout << ans[i] << endl;
62 }

```

2.4. Sqrt Decomposition

```

1 // Problem: Sum from l to r
2 // Ver MO'S ALGORITHM
3 // -----
4 int getId(int indx,int blockSZ) {
5     return indx/blockSZ;
6 }
7 void init(int sz) {
8     for(int i=0; i<=sz; i++)
9         BLOCK[i]=inf;
10 }
11 int query(int left, int right) {
12     int startBlockIndex=left/sqrt;
13     int endIBlockIndex = right / sqrt;
14     int sum = 0;
15     for (int i = startBlockIndex + 1; i < endIBlockIndex; i++) {
16         sum += blockSums[i];
17     }
18     for(i=left...(startBlockIndex*BLOCK_SIZE-1))
19         sum += a[i];
20     for(j = endIBlockIndex*BLOCK_SIZE ... right)
21         sum += a[i];
22 }

```

2.5. Bit

```

1 // #define RANGE_SUM
2 // #define RANGE_UPDATE
3 // Uncomment ONLY ONE above!
4 // clang-format off
5 class BIT {
6 private:
7     vector<int> bit;
8     const int n, offset;
9
10 private:
11     int low(const int i) { return i & (-i); }
12
13     // Point update
14     void _update(int i, const int delta) {
15         while (i <= n) {
16             bit[i] += delta;
17             i += low(i);
18         }
19     }
20
21     // Prefix query
22     int _query(int i) {
23         int sum = 0;
24         while (i > 0) {

```

```

25         sum += bit[i];
26         i -= low(i);
27     }
28     return sum;
29 }
30
31 void build(const vector<int> &arr) {
32     bit.resize(arr.size() + offset, 0);
33     for (int i = 1; i <= n; i++)
34         #ifdef RANGE_UPDATE
35             update(i - offset, i - offset, arr[i - offset]);
36         #endif
37         #ifdef RANGE_SUM
38             update(i - offset, arr[i - offset]);
39         #endif
40     }
41
42 public:
43     /// Constructor responsible for initializing the tree with 0's.
44     ///
45     /// Time Complexity: O(n log n)
46     BIT(const vector<int> &arr, const int indexed_from)
47         : n(arr.size() - indexed_from), offset(indexed_from ^ 1) {
48         assert(indexed_from == 0 || indexed_from == 1);
49         build(arr);
50     }
51
52     /// Constructor responsible for building the tree based on a vector.
53     ///
54     /// Time Complexity O(n)
55     BIT(const int n, const int indexed_from) : n(n), offset(indexed_from ^ 1) {
56         bit.resize(n + 1, 0);
57     }
58
59     #ifdef RANGE_UPDATE
60     void update(int l, int r, const int val) {
61         l += offset, r += offset;
62         assert(1 <= l), assert(1 <= r), assert(r <= n);
63         _update(l, val);
64         _update(r + 1, -val);
65     }
66     #endif
67
68     #ifdef RANGE_SUM
69     /// Update at a single index.
70     ///
71     /// Time Complexity O(log n)
72     void update(int idx, const int delta) {
73         idx += offset;
74         assert(1 <= idx), assert(idx <= n);
75         _update(idx, delta);
76     }
77     #endif
78
79     #ifdef RANGE_UPDATE
80     /// Query at a single index.
81     ///
82     /// Time Complexity O(log n)
83     int query(int idx) {
84         idx += offset;
85         assert(1 <= idx), assert(idx <= n);
86         return _query(idx);
87     }
88     #endif
89 }

```

```

90  #ifdef RANGE_SUM
91  /// Range query from l to r.
92  ///
93  /// Time Complexity O(log n)
94  int query(int l, int r) {
95      l += offset, r += offset;
96      assert(l <= l), assert(l <= r), assert(r <= n);
97      return _query(r) - _query(l - 1);
98  }
99  #endif
100 };
101 // clang-format on

```

2.6. Bit (Range Update)

```

1  /// INDEX THE ARRAY BY 1!!!
2  class BIT {
3  private:
4      vector<int> bit1, bit2;
5      int n;
6
7  private:
8      int low(int i) { return i & (-i); }
9
10     // Point update
11     void update(int i, const int delta, vector<int> &bit) {
12         while (i <= n) {
13             bit[i] += delta;
14             i += low(i);
15         }
16     }
17
18     // Prefix query
19     int query(int i, const vector<int> &bit) {
20         int sum = 0;
21         while (i > 0) {
22             sum += bit[i];
23             i -= low(i);
24         }
25         return sum;
26     }
27
28     // Builds the bit
29     void build(const vector<int> &arr) {
30         // OBS: BIT IS INDEXED FROM 1
31         // THE USAGE OF 1-BASED ARRAY IS MANDATORY
32         assert(arr.front() == 0);
33         this->n = (int)arr.size() - 1;
34         bit1.resize(arr.size(), 0);
35         bit2.resize(arr.size(), 0);
36
37         for (int i = 1; i <= n; i++)
38             update(i, arr[i]);
39     }
40
41 public:
42     /// Constructor responsible for initializing the tree with 0's.
43     ///
44     /// Time Complexity: O(n log n)
45     BIT(const vector<int> &arr) { build(arr); }
46
47     /// Constructor responsible for building the tree based on a vector.
48     ///
49     /// Time Complexity O(n)

```

```

50  BIT(const int n) {
51      // OBS: BIT IS INDEXED FROM 1
52      // THE USAGE OF 1-INDEXED ARRAY IS MANDATORY
53      this->n = n;
54      bit1.resize(n + 1, 0);
55      bit2.resize(n + 1, 0);
56  }
57
58  /// Range update from l to r.
59  ///
60  /// Time Complexity O(log n)
61  void update(const int l, const int r, const int delta) {
62      assert(l <= l), assert(l <= r), assert(r <= n);
63      update(l, delta, bit1);
64      update(r + 1, -delta, bit1);
65      update(l, delta * (l - 1), bit2);
66      update(r + 1, -delta * r, bit2);
67  }
68
69  /// Update at a single index.
70  ///
71  /// Time Complexity O(log n)
72  void update(const int i, const int delta) {
73      assert(l <= i), assert(i <= n);
74      update(i, i, delta);
75  }
76
77  /// Range query from l to r.
78  ///
79  /// Time Complexity O(log n)
80  int query(const int l, const int r) {
81      assert(l <= l), assert(l <= r), assert(r <= n);
82      return query(r) - query(l - 1);
83  }
84
85  /// Prefix query from l to i.
86  ///
87  /// Time Complexity O(log n)
88  int query(const int i) {
89      assert(i <= n);
90      return (query(i, bit1) * i) - query(i, bit2);
91  }
92 };

```

2.7. Counting Inversions (Minimum Number Of Adjacent Swaps To Sort Array)

```

1  // REQUIRES bit.cpp!!
2  // REQUIRES point_compression.cpp!!
3  int count_inversions(vector<int> &arr) {
4      arr = compress(arr);
5      int ans = 0;
6      BIT bit(arr.size());
7      for (int i = arr.size() - 1; i > 0; --i) {
8          ans += bit.query(arr[i] - 1);
9          bit.update(arr[i], 1);
10     }
11     return ans;
12 }

```

2.8. Min Queue

```

1  class Min_Queue {

```

```

2 private:
3     /// Contains a pair (value, index), strictly decreasing.
4     deque<pair<int, int>> d;
5
6 public:
7     Min_Queue() {}
8
9     int size() { return d.size(); }
10
11    /// Removes all elements with index <= idx
12    void pop(const int idx) {
13        while (!d.empty() && d.front().second <= idx)
14            d.pop_front();
15    }
16
17    /// Adds an element with value (val) and index (idx).
18    void push(const int val, const int idx) {
19        while (!d.empty() && d.back().first >= val)
20            d.pop_back();
21        d.emplace_back(val, idx);
22    }
23
24    int min_element() { return d.front().first; }
25 };

```

2.9. Ordered Set

```

1 #include <bits/stdc++.h>
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/trie_policy.hpp>
4
5 using namespace std;
6 using namespace __gnu_pbds;
7
8 template <typename T>
9 using ordered_set =
10     tree<T, null_type, less<T>, rb_tree_tag,
11         tree_order_statistics_node_update>;
12
13 ordered_set<int> X;
14 X.insert(1);
15 X.insert(2);
16 X.insert(4);
17 X.insert(8);
18 X.insert(16);
19
20 // 1, 2, 4, 8, 16
21 // returns the k-th greatest element from 0
22 cout << *X.find_by_order(1) << endl; // 2
23 cout << *X.find_by_order(2) << endl; // 4
24 cout << *X.find_by_order(4) << endl; // 16
25 cout << (end(X) == X.find_by_order(6)) << endl; // true
26
27 // returns the number of items strictly less than a number
28 cout << X.order_of_key(-5) << endl; // 0
29 cout << X.order_of_key(1) << endl; // 0
30 cout << X.order_of_key(3) << endl; // 2
31 cout << X.order_of_key(4) << endl; // 2
32 cout << X.order_of_key(400) << endl; // 5

```

2.10. Persistent Segment Tree

```

1 class Persistent_Seg_Tree {

```

```

2 struct Node {
3     int val;
4     Node *left, *right;
5     Node(const int v) : val(v), left(nullptr), right(nullptr) {}
6 };
7
8 private:
9     const Node NEUTRAL_NODE = Node(0);
10    int merge_nodes(const int x, const int y) { return x + y; }
11
12 private:
13     const int n;
14     vector<Node*> version = {nullptr};
15
16 public:
17     /// Builds version[0] with the values in the array.
18     ///
19     /// Time complexity: O(n)
20     Node *build(Node *node, const int l, const int r, const vector<int> &arr) {
21         node = new Node(NEUTRAL_NODE);
22         if (l == r) {
23             node->val = arr[l];
24             return node;
25         }
26
27         const int mid = (l + r) / 2;
28         node->left = build(node->left, l, mid, arr);
29         node->right = build(node->right, mid + 1, r, arr);
30         node->val = merge_nodes(node->left->val, node->right->val);
31         return node;
32     }
33
34     Node *update(Node *cur_tree, Node *prev_tree, const int l, const int r,
35                 const int idx, const int delta) {
36         if (l > idx || r < idx)
37             return cur_tree != nullptr ? cur_tree : prev_tree;
38
39         if (cur_tree == nullptr && prev_tree == nullptr)
40             cur_tree = new Node(NEUTRAL_NODE);
41         else
42             cur_tree = new Node(cur_tree == nullptr ? *prev_tree : *cur_tree);
43
44         if (l == r) {
45             cur_tree->val += delta;
46             return cur_tree;
47         }
48
49         const int mid = (l + r) / 2;
50         cur_tree->left =
51             update(cur_tree->left, prev_tree ? prev_tree->left : nullptr, l,
52                 mid,
53                 idx, delta);
54         cur_tree->right =
55             update(cur_tree->right, prev_tree ? prev_tree->right : nullptr,
56                 mid + 1, r, idx, delta);
57         cur_tree->val =
58             merge_nodes(cur_tree->left ? cur_tree->left->val : NEUTRAL_NODE.val,
59                 cur_tree->right ? cur_tree->right->val :
60                 NEUTRAL_NODE.val);
61         return cur_tree;
62     }
63
64     int _query(Node *node, const int l, const int r, const int i, const int j)
65     {
66         if (node == nullptr || l > j || r < i)

```

```

64     return NEUTRAL_NODE.val;
65
66     if (i <= l && r <= j)
67         return node->val;
68
69     int mid = (l + r) / 2;
70     return merge_nodes(_query(node->left, l, mid, i, j),
71                       _query(node->right, mid + 1, r, i, j));
72 }
73
74 void create_version(const int v) {
75     if (v >= this->version.size())
76         version.resize(v + 1);
77 }
78
79 public:
80 Persistent_Seg_Tree() : n(-1) {}
81
82 /// Constructor that initializes the segment tree empty. It's allowed to
83 /// query
84 /// from 0 to MAXN - 1.
85 ///
86 /// Time Complexity: O(1)
87 Persistent_Seg_Tree(const int MAXN) : n(MAXN) {}
88
89 /// Constructor that allows to pass initial values to the leafs. It's
90 /// allowed
91 /// to query from 0 to n - 1.
92 ///
93 /// Time Complexity: O(n)
94 Persistent_Seg_Tree(const vector<int> &arr) : n(arr.size()) {
95     this->version[0] = this->build(this->version[0], 0, this->n - 1, arr);
96 }
97
98 /// Links the root of a version to a previous version.
99 ///
100 /// Time Complexity: O(1)
101 void link(const int version, const int prev_version) {
102     assert(this->n > -1);
103     assert(0 <= prev_version), assert(prev_version <= version);
104     this->create_version(version);
105     this->version[version] = this->version[prev_version];
106 }
107
108 /// Updates an index in cur_tree based on prev_tree with a delta.
109 ///
110 /// Time Complexity: O(log(n))
111 void update(const int cur_version, const int prev_version, const int idx,
112            const int delta) {
113     assert(this->n > -1);
114     assert(0 <= prev_version), assert(prev_version <= cur_version);
115     this->create_version(cur_version);
116     this->version[cur_version] =
117         this->update(this->version[cur_version],
118                   this->version[prev_version],
119                   0, this->n - 1, idx, delta);
120 }
121
122 /// Query from l to r.
123 ///
124 /// Time Complexity: O(log(n))
125 int query(const int version, const int l, const int r) {
126     assert(this->n > -1);
127     assert(0 <= l), assert(l <= r), assert(r < this->n);
128     return this->_query(this->version[version], 0, this->n - 1, l, r);

```

```

126     }
127 };

```

2.11. Segment Tree

```

1 class Seg_Tree {
2 public:
3     struct Node {
4         int val, lazy;
5
6         Node() {}
7         Node(const int val) : val(val), lazy(0) {}
8     };
9
10 private:
11     /// // Range Sum
12     /// Node NEUTRAL_NODE = Node(0);
13     /// Node merge_nodes(const Node &x, const Node &y) {
14     ///     return Node(x.val + y.val);
15     /// };
16     ///
17     /// void apply_lazy(const int l, const int r, const int pos) {
18     ///     // for set change this to =
19     ///     tree[pos].val += (r - l + 1) * tree[pos].lazy;
20     /// }
21
22     /// // RMQ Max
23     /// Node NEUTRAL_NODE = Node(-INF);
24     /// Node merge_nodes(const Node &x, const Node &y) {
25     ///     return Node(max(x.val, y.val));
26     /// }
27     /// void apply_lazy(const int l, const int r, const int pos) {
28     ///     tree[pos].val += tree[pos].lazy;
29     /// }
30
31     /// // RMQ Min
32     /// Node NEUTRAL_NODE = Node(INF);
33     /// Node merge_nodes(const Node &x, const Node &y) {
34     ///     return Node(min(x.val, y.val));
35     /// }
36     /// void apply_lazy(const int l, const int r, const int pos) {
37     ///     tree[pos].val += tree[pos].lazy;
38     /// }
39
40     /// // XOR
41     /// // Only works with point updates
42     /// Node NEUTRAL_NODE = Node(0);
43     /// Node merge_nodes(const Node &x, const Node &y) {
44     ///     return Node(x.val ^ y.val);
45     /// };
46     ///
47     /// void apply_lazy(const int l, const int r, const int pos) {}
48
49 private:
50     int n;
51
52 public:
53     vector<Node> tree;
54
55 private:
56     void propagate(const int l, const int r, const int pos) {
57         if (tree[pos].lazy != 0) {
58             apply_lazy(l, r, pos);
59             if (l != r) {

```



```

60     // for set change this to =
61     tree[2 * pos + 1].lazy += tree[pos].lazy;
62     tree[2 * pos + 2].lazy += tree[pos].lazy;
63 }
64 tree[pos].lazy = 0;
65 }
66 }
67
68 Node _build(const int l, const int r, const vector<int> &arr, const int
pos) {
69     if (l == r)
70         return tree[pos] = Node(arr[l]);
71
72     int mid = (l + r) / 2;
73     return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
74                                   _build(mid + 1, r, arr, 2 * pos + 2));
75 }
76
77 int _get_first(const int l, const int r, const int i, const int j,
78               const int v, const int pos) {
79     propagate(l, r, pos);
80
81     if (l > r || l > j || r < i)
82         return -1;
83     // Needs RMQ MAX
84     // Replace to <= for greater or (with RMQ MIN) > for smaller or
85     // equal or >= for smaller
86     if (tree[pos].val < v)
87         return -1;
88
89     if (l == r)
90         return l;
91
92     int mid = (l + r) / 2;
93     int aux = _get_first(l, mid, i, j, v, 2 * pos + 1);
94     if (aux != -1)
95         return aux;
96     return _get_first(mid + 1, r, i, j, v, 2 * pos + 2);
97 }
98
99 Node _query(const int l, const int r, const int i, const int j,
100            const int pos) {
101     propagate(l, r, pos);
102
103     if (l > r || l > j || r < i)
104         return NEUTRAL_NODE;
105
106     if (i <= l && r <= j)
107         return tree[pos];
108
109     int mid = (l + r) / 2;
110     return merge_nodes(_query(l, mid, i, j, 2 * pos + 1),
111                       _query(mid + 1, r, i, j, 2 * pos + 2));
112 }
113
114 // It adds a number delta to the range from i to j
115 Node _update(const int l, const int r, const int i, const int j,
116             const int delta, const int pos) {
117     propagate(l, r, pos);
118
119     if (l > r || l > j || r < i)
120         return tree[pos];
121
122     if (i <= l && r <= j) {
123         tree[pos].lazy = delta;

```

```

124     propagate(l, r, pos);
125     return tree[pos];
126 }
127
128 int mid = (l + r) / 2;
129 return tree[pos] =
130     merge_nodes(_update(l, mid, i, j, delta, 2 * pos + 1),
131               _update(mid + 1, r, i, j, delta, 2 * pos + 2));
132 }
133
134 void build(const vector<int> &arr) {
135     this->tree.resize(4 * this->n);
136     this->_build(0, this->n - 1, arr, 0);
137 }
138
139 public:
140     /// N equals to -1 means the Segment Tree hasn't been created yet.
141     Seg_Tree() : n(-1) {}
142
143     /// Constructor responsible for initializing the tree with val.
144     ///
145     /// Time Complexity O(n)
146     Seg_Tree(const int n, const int val = 0) : n(n) {
147         this->tree.resize(4 * this->n, Node(val));
148     }
149
150     /// Constructor responsible for building the tree based on a vector.
151     ///
152     /// Time Complexity O(n)
153     Seg_Tree(const vector<int> &arr) : n(arr.size()) { this->build(arr); }
154
155     /// Returns the first index from i to j compared to v.
156     /// Uncomment the line in the original function to get the proper element
157     /// that
158     /// may be: GREATER OR EQUAL, GREATER, SMALLER OR EQUAL, SMALLER.
159     ///
160     /// Time Complexity O(log n)
161     int get_first(const int i, const int j, const int v) {
162         assert(this->n >= 0);
163         return this->_get_first(0, this->n - 1, i, j, v, 0);
164     }
165
166     /// Update at a single index.
167     ///
168     /// Time Complexity O(log n)
169     void update(const int idx, const int delta) {
170         assert(this->n >= 0);
171         assert(0 <= idx), assert(idx < this->n);
172         this->_update(0, this->n - 1, idx, idx, delta, 0);
173     }
174
175     /// Range update from l to r.
176     ///
177     /// Time Complexity O(log n)
178     void update(const int l, const int r, const int delta) {
179         assert(this->n >= 0);
180         assert(0 <= l), assert(l <= r), assert(r < this->n);
181         this->_update(0, this->n - 1, l, r, delta, 0);
182     }
183
184     /// Query at a single index.
185     ///
186     /// Time Complexity O(log n)
187     int query(const int idx) {
188         assert(this->n >= 0);

```

```

188     assert(0 <= idx), assert(idx < this->n);
189     return this->_query(0, this->n - 1, idx, idx, 0).val;
190 }
191
192 /// Range query from l to r.
193 ///
194 /// Time Complexity O(log n)
195 int query(const int l, const int r) {
196     assert(this->n >= 0);
197     assert(0 <= l), assert(l <= r), assert(r < this->n);
198     return this->_query(0, this->n - 1, l, r, 0).val;
199 }
200 };

```

2.12. Segment Tree 2D

```

1 // REQUIRES segment_tree.cpp!!
2 class Seg_Tree_2d {
3 private:
4     /// // range sum
5     /// int NEUTRAL_VALUE = 0;
6     /// int merge_nodes(const int &x, const int &y) {
7     ///     return x + y;
8     /// }
9
10    /// // RMQ max
11    /// int NEUTRAL_VALUE = -INF;
12    /// int merge_nodes(const int &x, const int &y) {
13    ///     return max(x, y);
14    /// }
15
16    /// // RMQ min
17    /// int NEUTRAL_VALUE = INF;
18    /// int merge_nodes(const int &x, const int &y) {
19    ///     return min(x, y);
20    /// }
21
22 private:
23     int n, m;
24
25 public:
26     vector<Seg_Tree> tree;
27
28 private:
29     void st_build(const int l, const int r, const int pos, const
30         vector<vector<int>> &mat) {
31         if(l == r)
32             tree[pos] = Seg_Tree(mat[l]);
33         else {
34             int mid = (l + r) / 2;
35             st_build(l, mid, 2*pos + 1, mat);
36             st_build(mid + 1, r, 2*pos + 2, mat);
37             for(int i = 0; i < tree[2*pos + 1].tree.size(); i++)
38                 tree[pos].tree[i].val = merge_nodes(tree[2*pos + 1].tree[i].val,
39                                                         tree[2*pos + 2].tree[i].val);
40         }
41     }
42
43     int st_query(const int l, const int r, const int x1, const int y1, const
44         int x2, const int y2, const int pos) {
45         if(l > x2 || r < x1)
46             return NEUTRAL_VALUE;
47
48         if(x1 <= l && r <= x2)

```

```

47         return tree[pos].query(y1, y2);
48
49         int mid = (l + r) / 2;
50         return merge_nodes(st_query(l, mid, x1, y1, x2, y2, 2*pos + 1),
51                             st_query(mid + 1, r, x1, y1, x2, y2, 2*pos + 2));
52     }
53
54 void st_update(const int l, const int r, const int x, const int y, const
55     int delta, const int pos) {
56     if(l > x || r < x)
57         return;
58
59     // Only supports point updates.
60     if(l == r) {
61         tree[pos].update(y, delta);
62         return;
63     }
64
65     int mid = (l + r) / 2;
66     st_update(l, mid, x, y, delta, 2*pos + 1);
67     st_update(mid + 1, r, x, y, delta, 2*pos + 2);
68     tree[pos].update(y, delta);
69 }
70
71 public:
72     Seg_Tree_2d() {
73         this->n = -1;
74         this->m = -1;
75     }
76
77     Seg_Tree_2d(const int n, const int m) {
78         this->n = n;
79         this->m = m;
80         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
81         assert(m < 10000);
82         tree.resize(4 * n, Seg_Tree(m));
83     }
84
85     Seg_Tree_2d(const int n, const int m, const vector<vector<int>> &mat) {
86         this->n = n;
87         this->m = m;
88         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
89         assert(m < 10000);
90         tree.resize(4 * n, Seg_Tree(m));
91         st_build(0, n - 1, 0, mat);
92     }
93
94     // Query from (x1, y1) to (x2, y2).
95     //
96     // Time complexity: O((log n) * (log m))
97     int query(const int x1, const int y1, const int x2, const int y2) {
98         assert(this->n > -1);
99         assert(0 <= x1); assert(x1 <= x2); assert(x2 < this->n);
100        assert(0 <= y1); assert(y1 <= y2); assert(y2 < this->n);
101        return st_query(0, this->n - 1, x1, y1, x2, y2, 0);
102    }
103
104    // Point updates on position (x, y).
105    //
106    // Time complexity: O((log n) * (log m))
107    void update(const int x, const int y, const int delta) {
108        assert(0 <= x); assert(x < this->n);
109        assert(0 <= y); assert(y < this->n);
110        st_update(0, this->n - 1, x, y, delta, 0);

```

111

};

2.13. Segment Tree Beats

```

1  #define MIN_UPDATE // supports for i in [l, r] do a[i] = min(a[i], x)
2  #define MAX_UPDATE // supports for i in [l, r] do a[i] = max(a[i], x)
3  #define ADD_UPDATE // supports for i in [l, r] a[i] += x
4
5  // clang-format off
6  class Seg_Tree_Beats {
7      const static int INF = (sizeof(int) == 4 ? 1e9 : 2e18) + 1e5;
8
9  public:
10     struct Node {
11         int sum;
12         #ifdef ADD_UPDATE
13         int lazy = 0;
14         #endif
15         #ifdef MIN_UPDATE
16         // Stores the maximum value, its frequency, and 2nd max value.
17         int maxx, cnt_maxx, smaxx;
18         #endif
19         #ifdef MAX_UPDATE
20         // Stores the minimum value, its frequency, and 2nd min value.
21         int minn, cnt_minn, sminn;
22         #endif
23         Node() {}
24         Node(const int val) : sum(val) {
25             #ifdef MIN_UPDATE
26             maxx = val, cnt_maxx = 1, smaxx = -INF;
27             #endif
28             #ifdef MAX_UPDATE
29             minn = val, cnt_minn = 1, sminn = INF;
30             #endif
31         }
32     };
33
34 private:
35     // Range Sum
36     Node merge_nodes(const Node &x, const Node &y) {
37         Node node;
38         node.sum = x.sum + y.sum;
39
40         #ifdef MIN_UPDATE
41         node.maxx = max(x.maxx, y.maxx);
42         node.smaxx = max(x.smaxx, y.smaxx);
43         node.cnt_maxx = 0;
44         if (node.maxx == x.maxx)
45             node.cnt_maxx += x.cnt_maxx;
46         else
47             node.smaxx = max(node.smaxx, x.maxx);
48         if (node.maxx == y.maxx)
49             node.cnt_maxx += y.cnt_maxx;
50         else
51             node.smaxx = max(node.smaxx, y.maxx);
52         #endif
53
54         #ifdef MAX_UPDATE
55         node.minn = min(x.minn, y.minn);
56         node.sminn = min(x.sminn, y.sminn);
57         node.cnt_minn = 0;
58         if (node.minn == x.minn)
59             node.cnt_minn += x.cnt_minn;
60         else

```

```

61         node.sminn = min(node.sminn, x.minn);
62         if (node.minn == y.minn)
63             node.cnt_minn += y.cnt_minn;
64         else
65             node.sminn = min(node.sminn, y.minn);
66         #endif
67         return node;
68     }
69
70 private:
71     int n;
72
73 public:
74     vector<Node> tree;
75
76 private:
77     #ifdef MIN_UPDATE
78     // in queries a[i] = min(a[i], x)
79     void apply_update_min(const int pos, const int x) {
80         Node &node = tree[pos];
81         node.sum -= (node.maxx - x) * node.cnt_maxx;
82         #ifdef MAX_UPDATE
83         if (node.maxx == node.minn)
84             node.minn = x;
85         else if (node.maxx == node.sminn)
86             node.sminn = x;
87         #endif
88         node.maxx = x;
89     }
90 #endif
91
92     #ifdef MAX_UPDATE
93     void apply_update_max(const int pos, const int x) {
94         Node &node = tree[pos];
95         node.sum += (x - node.minn) * node.cnt_minn;
96         #ifdef MIN_UPDATE
97         if (node.minn == node.maxx)
98             node.maxx = x;
99         else if (node.minn == node.smaxx)
100             node.smaxx = x;
101         #endif
102         node.minn = x;
103     }
104 #endif
105
106     #ifdef ADD_UPDATE
107     void apply_update_sum(const int l, const int r, const int pos, const int
108         v) {
109         tree[pos].sum += (r - l + 1) * v;
110         #ifdef ADD_UPDATE
111         tree[pos].lazy += v;
112         #endif
113         #ifdef MIN_UPDATE
114         tree[pos].maxx += v;
115         tree[pos].smaxx += v;
116         #endif
117         #ifdef MAX_UPDATE
118         tree[pos].minn += v;
119         tree[pos].sminn += v;
120         #endif
121     }
122 #endif
123
124     void propagate(const int l, const int r, const int pos) {
125         if (l == r)

```

```

125     return;
126     Node &node = tree[pos];
127     const int c1 = 2 * pos + 1, c2 = 2 * pos + 2;
128
129     #ifdef ADD_UPDATE
130     if (node.lazy != 0) {
131         const int mid = (l + r) / 2;
132         apply_update_sum(l, mid, c1, node.lazy);
133         apply_update_sum(mid + 1, r, c2, node.lazy);
134         node.lazy = 0;
135     }
136     #endif
137
138     #ifdef MIN_UPDATE
139     // min update
140     if (tree[c1].maxx > node.maxx)
141         apply_update_min(c1, node.maxx);
142     if (tree[c2].maxx > node.maxx)
143         apply_update_min(c2, node.maxx);
144     #endif
145
146     #ifdef MAX_UPDATE
147     // max update
148     if (tree[c1].minn < node.minn)
149         apply_update_max(c1, node.minn);
150     if (tree[c2].minn < node.minn)
151         apply_update_max(c2, node.minn);
152     #endif
153 }
154
155 Node _build(const int l, const int r, const vector<int> &arr, const int
pos) {
156     if (l == r)
157         return tree[pos] = Node(arr[l]);
158
159     const int mid = (l + r) / 2;
160     return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
161                                   _build(mid + 1, r, arr, 2 * pos + 2));
162 }
163
164 Node _query(const int l, const int r, const int i, const int j, const int
pos,
165            const Node &NEUTRAL_NODE) {
166     propagate(l, r, pos);
167
168     if (l > r || l > j || r < i)
169         return NEUTRAL_NODE;
170
171     if (i <= l && r <= j)
172         return tree[pos];
173
174     const int mid = (l + r) / 2;
175     return merge_nodes(_query(l, mid, i, j, 2 * pos + 1, NEUTRAL_NODE),
176                       _query(mid + 1, r, i, j, 2 * pos + 2, NEUTRAL_NODE));
177 }
178
179 #ifdef ADD_UPDATE
180 Node _update_sum(const int l, const int r, const int i, const int j,
181                 const int v, const int pos) {
182     propagate(l, r, pos);
183
184     if (l > r || l > j || r < i)
185         return tree[pos];
186
187     if (i <= l && r <= j) {

```

```

188         apply_update_sum(l, r, pos, v);
189         return tree[pos];
190     }
191
192     int mid = (l + r) / 2;
193     return tree[pos] =
194         merge_nodes(_update_sum(l, mid, i, j, v, 2 * pos + 1),
195                   _update_sum(mid + 1, r, i, j, v, 2 * pos + 2));
196 }
197 #endif
198
199 #ifdef MIN_UPDATE
200 Node _update_min(const int l, const int r, const int i, const int j,
201                 const int x, const int pos) {
202     propagate(l, r, pos);
203
204     if (l > r || l > j || r < i || tree[pos].maxx <= x)
205         return tree[pos];
206
207     if (i <= l && r <= j && tree[pos].smaxx < x) {
208         apply_update_min(pos, x);
209         return tree[pos];
210     }
211
212     const int mid = (l + r) / 2;
213     return tree[pos] =
214         merge_nodes(_update_min(l, mid, i, j, x, 2 * pos + 1),
215                   _update_min(mid + 1, r, i, j, x, 2 * pos + 2));
216 }
217 #endif
218
219 #ifdef MAX_UPDATE
220 Node _update_max(const int l, const int r, const int i, const int j,
221                 const int x, const int pos) {
222     propagate(l, r, pos);
223
224     if (l > r || l > j || r < i || tree[pos].minn >= x)
225         return tree[pos];
226
227     if (i <= l && r <= j && tree[pos].sminn > x) {
228         apply_update_max(pos, x);
229         return tree[pos];
230     }
231
232     const int mid = (l + r) / 2;
233     return tree[pos] =
234         merge_nodes(_update_max(l, mid, i, j, x, 2 * pos + 1),
235                   _update_max(mid + 1, r, i, j, x, 2 * pos + 2));
236 }
237 #endif
238
239 void build(const vector<int> &arr) {
240     this->tree.resize(4 * this->n);
241     this->_build(0, this->n - 1, arr, 0);
242 }
243
244 public:
245     /// N equals to -1 means the Segment Tree hasn't been created yet.
246     Seg_Tree Beats() : n(-1) {}
247
248     /// Constructor responsible for initializing the tree with 0's.
249     ///
250     /// Time Complexity O(n)
251     Seg_Tree Beats(const int n) : n(n) {
252         this->tree.resize(4 * this->n, Node(0));

```

```

253 }
254
255 /// Constructor responsible for building the tree based on a vector.
256 ///
257 /// Time Complexity O(n)
258 Seg_Tree Beats(const vector<int> &arr) : n(arr.size()) { this->build(arr);
    }
259
260 #ifdef ADD_UPDATE
261 /// Range update from l to r.
262 /// Type: for i in range [l, r] do a[i] += x
263 void update_sum(const int l, const int r, const int x) {
264     assert(this->n >= 0);
265     assert(0 <= l), assert(l <= r), assert(r < this->n);
266     this->_update_sum(0, this->n - 1, l, r, x, 0);
267 }
268 #endif
269
270 #ifdef MIN_UPDATE
271 /// Range update from l to r.
272 /// Type: for i in range [l, r] do a[i] = min(a[i], x)
273 void update_min(const int l, const int r, const int x) {
274     assert(this->n >= 0);
275     assert(0 <= l), assert(l <= r), assert(r < this->n);
276     this->_update_min(0, this->n - 1, l, r, x, 0);
277 }
278 #endif
279
280 #ifdef MAX_UPDATE
281 /// Range update from l to r.
282 /// Type: for i in range [l, r] do a[i] = max(a[i], x)
283 void update_max(const int l, const int r, const int x) {
284     assert(this->n >= 0);
285     assert(0 <= l), assert(l <= r), assert(r < this->n);
286     this->_update_max(0, this->n - 1, l, r, x, 0);
287 }
288 #endif
289
290 /// Range Sum query from l to r.
291 ///
292 /// Time Complexity O(log n)
293 int query_sum(const int l, const int r) {
294     assert(this->n >= 0);
295     assert(0 <= l), assert(l <= r), assert(r < this->n);
296     return this->_query(0, this->n - 1, l, r, 0, Node(0)).sum;
297 }
298
299 #ifdef MAX_UPDATE
300 /// Range Min query from l to r.
301 ///
302 /// Time Complexity O(log n)
303 int query_min(const int l, const int r) {
304     assert(this->n >= 0);
305     assert(0 <= l), assert(l <= r), assert(r < this->n);
306     return this->_query(0, this->n - 1, l, r, 0, Node(INF)).minn;
307 }
308 #endif
309
310 #ifdef MIN_UPDATE
311 /// Range Max query from l to r.
312 ///
313 /// Time Complexity O(log n)
314 int query_max(const int l, const int r) {
315     assert(this->n >= 0);
316     assert(0 <= l), assert(l <= r), assert(r < this->n);

```

```

317     return this->_query(0, this->n - 1, l, r, 0, Node(-INF)).maxx;
318 }
319 #endif
320 };
321 // clang-format on
322 // OBS: Q updates of the type a[i] = (min/max)(a[i], x) have the amortized
323 // complexity of O(q * (log(n) ^ 2)).

```

2.14. Segment Tree Polynomial

```

1  /// Works for the polynomial f(x) = z1*x + z0
2  class Seg_Tree {
3  public:
4      struct Node {
5          int val, z1, z0;
6
7          Node() {}
8          Node(const int val, const int z1, const int z0)
9              : val(val), z1(z1), z0(z0) {}
10     };
11
12 private:
13     // range sum
14     Node NEUTRAL_NODE = Node(0, 0, 0);
15     Node merge_nodes(const Node &x, const Node &y) {
16         return Node(x.val + y.val, 0, 0);
17     }
18     void apply_lazy(const int l, const int r, const int pos) {
19         tree[pos].val += (r - l + 1) * tree[pos].z0;
20         tree[pos].val += (r - l) * (r - l + 1) / 2 * tree[pos].z1;
21     }
22
23 private:
24     int n;
25
26 public:
27     vector<Node> tree;
28
29 private:
30     void st_propagate(const int l, const int r, const int pos) {
31         if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {
32             apply_lazy(l, r, pos);
33             int mid = (l + r) / 2;
34             int sz_left = mid - l + 1;
35             if (l != r) {
36                 tree[2 * pos + 1].z0 += tree[pos].z0;
37                 tree[2 * pos + 1].z1 += tree[pos].z1;
38
39                 tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
40                 tree[2 * pos + 2].z1 += tree[pos].z1;
41             }
42             tree[pos].z0 = 0;
43             tree[pos].z1 = 0;
44         }
45     }
46
47     Node st_build(const int l, const int r, const vector<int> &arr,
48                  const int pos) {
49         if (l == r)
50             return tree[pos] = Node(arr[l], 0, 0);
51
52         int mid = (l + r) / 2;
53         return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
54                                       st_build(mid + 1, r, arr, 2 * pos + 2));
55     }

```

```

55 }
56
57 Node st_query(const int l, const int r, const int i, const int j,
58             const int pos) {
59     st_propagate(l, r, pos);
60
61     if (l > r || l > j || r < i)
62         return NEUTRAL_NODE;
63
64     if (i <= l && r <= j)
65         return tree[pos];
66
67     int mid = (l + r) / 2;
68     return merge_nodes(st_query(l, mid, i, j, 2 * pos + 1),
69                       st_query(mid + 1, r, i, j, 2 * pos + 2));
70 }
71
72 // it adds a number delta to the range from i to j
73 Node st_update(const int l, const int r, const int i, const int j,
74              const int z1, const int z0, const int pos) {
75     st_propagate(l, r, pos);
76
77     if (l > r || l > j || r < i)
78         return tree[pos];
79
80     if (i <= l && r <= j) {
81         tree[pos].z0 = (l - i + 1) * z0;
82         tree[pos].z1 = z1;
83         st_propagate(l, r, pos);
84         return tree[pos];
85     }
86
87     int mid = (l + r) / 2;
88     return tree[pos] =
89         merge_nodes(st_update(l, mid, i, j, z1, z0, 2 * pos + 1),
90                   st_update(mid + 1, r, i, j, z1, z0, 2 * pos + 2));
91 }
92
93 public:
94 Seg_Tree() : n(-1) {}
95
96 Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0, 0)); }
97
98 Seg_Tree(const vector<int> &arr) { this->build(arr); }
99
100 void build(const vector<int> &arr) {
101     this->n = arr.size();
102     this->tree.resize(4 * this->n);
103     this->st_build(0, this->n - 1, arr, 0);
104 }
105
106 /// Index update of a polynomial  $f(x) = z1 \cdot x + z0$ 
107 ///
108 /// Time Complexity  $O(\log n)$ 
109 void update(const int i, const int z1, const int z0) {
110     assert(this->n >= 0);
111     assert(0 <= i), assert(i < this->n);
112     this->st_update(0, this->n - 1, i, i, z1, z0, 0);
113 }
114
115 /// Range update of a polynomial  $f(x) = z1 \cdot x + z0$  from l to r
116 ///
117 /// Time Complexity  $O(\log n)$ 
118 void update(const int l, const int r, const int z1, const int z0) {

```

```

119     assert(this->n >= 0);
120     assert(0 <= l), assert(l <= r), assert(r < this->n);
121     this->st_update(0, this->n - 1, l, r, z1, z0, 0);
122 }
123
124 /// Range sum query from l to r
125 ///
126 /// Time Complexity  $O(\log n)$ 
127 int query(const int l, const int r) {
128     assert(this->n >= 0);
129     assert(0 <= l), assert(l <= r), assert(r < this->n);
130     return this->st_query(0, this->n - 1, l, r, 0).val;
131 }
132 };

```

2.15. Sparse Table

```

1 class Sparse_Table {
2 private:
3     /// Sparse table min
4     // int merge(const int l, const int r) { return min(l, r); }
5     /// Sparse table max
6     // int merge(const int l, const int r) { return max(l, r); }
7
8 private:
9     int n;
10    vector<vector<int>> table;
11    vector<int> lg;
12
13 private:
14     /// lg[i] represents the log2(i)
15     void build_log_array() {
16         lg.resize(this->n + 1);
17         for (int i = 2; i <= this->n; i++)
18             lg[i] = lg[i / 2] + 1;
19     }
20
21     /// Time Complexity:  $O(n \cdot \log(n))$ 
22     void build_sparse_table(const vector<int> &arr) {
23         table.resize(lg[this->n] + 1, vector<int>(this->n));
24
25         table[0] = arr;
26         int pow2 = 1;
27         for (int i = 1; i < table.size(); i++) {
28             const int lastsz = this->n - pow2 + 1;
29             for (int j = 0; j + pow2 < lastsz; j++)
30                 table[i][j] = merge(table[i - 1][j], table[i - 1][j + pow2]);
31             pow2 <= 1;
32         }
33     }
34
35 public:
36     /// Constructor that builds the log array and the sparse table.
37     ///
38     /// Time Complexity:  $O(n \cdot \log(n))$ 
39     Sparse_Table(const vector<int> &arr) : n(arr.size()) {
40         this->build_log_array();
41         this->build_sparse_table(arr);
42     }
43
44     void print() {
45         int pow2 = 1;
46         for (int i = 0; i < table.size(); i++) {
47             const int sz = (int)(table.front().size()) - pow2 + 1;

```

```

48     for (int j = 0; j < sz; j++)
49         cout << table[i][j] << " \n"[(j + 1) == sz];
50     pow2 <= 1;
51 }
52 }
53
54 /// Range query from l to r.
55 ///
56 /// Time Complexity: O(1)
57 int query(const int l, const int r) {
58     assert(0 <= l), assert(l <= r), assert(r < this->n);
59     int lgg = lg[r - l + 1];
60     return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
61 }
62 };

```

2.16. Treap

```

1  // PLEASE DO NOT COPY!
2
3  // clang-format off
4  mtl19937 rng(chrono::steady_clock::now().time_since_epoch().count());
5  // #define REVERSE
6  // #define LAZY
7  class Treap {
8  public:
9      struct Node {
10         Node *left = nullptr, *right = nullptr, *par = nullptr;
11         // Priority to be used in the treap
12         const int rank;
13         int size = 1, val;
14         // Contains the result of the range query between the node and its
15         // children.
16         int ans;
17         #ifdef LAZY
18         int lazy = 0;
19         #endif
20         #ifdef REVERSE
21         bool rev = false;
22         #endif
23
24         Node(const int val) : val(val), ans(val), rank(rng()) {}
25         Node(const int val, const int rank) : val(val), ans(val), rank(rank) {}
26     };
27 private:
28     vector<Node*> nodes;
29     int _size = 0;
30     Node *root = nullptr;
31
32 private:
33     // // Range Sum
34     // void merge_nodes(Node *node) {
35     //     node->ans = node->val;
36     //     if (node->left)
37     //         node->ans += node->left->ans;
38     //     if (node->right)
39     //         node->ans += node->right->ans;
40     // }
41
42     // #ifdef LAZY
43     // void apply_lazy(Node *node) {
44     //     node->val += node->lazy;
45     //     node->ans += node->lazy * get_size(node);

```

```

46     // }
47     // #endif
48
49     // // RMQ Min
50     void merge_nodes(Node *node) {
51         node->ans = node->val;
52         if (node->left)
53             node->ans = min(node->ans, node->left->ans);
54         if (node->right)
55             node->ans = min(node->ans, node->right->ans);
56     }
57
58     #ifdef LAZY
59     void apply_lazy(Node *node) {
60         node->val += node->lazy;
61         node->ans += node->lazy;
62     }
63     #endif
64
65     // // RMQ Max
66     void merge_nodes(Node *node) {
67         node->ans = node->val;
68         if (node->left)
69             node->ans = max(node->ans, node->left->ans);
70         if (node->right)
71             node->ans = max(node->ans, node->right->ans);
72     }
73
74     #ifdef LAZY
75     void apply_lazy(Node *node) {
76         node->val += node->lazy;
77         node->ans += node->lazy;
78     }
79     #endif
80
81     #ifdef REVERSE
82     void apply_reverse(Node *node) {
83         swap(node->left, node->right);
84         // write other operations here
85     }
86     #endif
87
88     int get_size(const Node *node) { return node ? node->size : 0; }
89
90     void update_size(Node *node) {
91         if (node)
92             node->size = 1 + get_size(node->left) + get_size(node->right);
93     }
94
95     void print(Node *node) {
96         if (!node)
97             return;
98         if (node->left) {
99             cerr << "left" << endl;
100             print(node->left);
101         }
102         cerr << node->val << endl;
103         cerr << endl;
104         if (node->right) {
105             cerr << "right" << endl;
106             print(node->right);
107         }
108     }
109
110     #ifdef REVERSE

```

```

111 void propagate_reverse(Node *node) {
112     if (node && node->rev) {
113         apply_reverse(node);
114         if (node->left)
115             node->left->rev ^= 1;
116         if (node->right)
117             node->right->rev ^= 1;
118         node->rev = 0;
119     }
120 }
121 #endif
122
123 #ifdef LAZY
124 void propagate_lazy(Node *node) {
125     if (node && node->lazy != 0) {
126         apply_lazy(node);
127         if (node->left)
128             node->left->lazy += node->lazy;
129         if (node->right)
130             node->right->lazy += node->lazy;
131         node->lazy = 0;
132     }
133 }
134 #endif
135
136 void update_node(Node *node) {
137     if (node) {
138         update_size(node);
139         #ifdef LAZY
140             propagate_lazy(node->left);
141             propagate_lazy(node->right);
142         #endif
143         #ifdef REVERSE
144             propagate_reverse(node->left);
145             propagate_reverse(node->right);
146         #endif
147         merge_nodes(node);
148     }
149 }
150
151 /// Splits the treap into to different treaps that contains nodes with
152 /// indexes
153 /// <= pos ans indexes > pos. The nodes l and r contains, in the end, these
154 /// two different treaps.
155 void split(Node *node, Node *&l, Node *&r, const int pos, Node *pl =
156     nullptr, Node *pr = nullptr) {
157     if (!node)
158         l = r = nullptr;
159     else {
160         #ifdef LAZY
161             propagate_lazy(node);
162         #endif
163         #ifdef REVERSE
164             propagate_reverse(node);
165         #endif
166         if (get_size(node->left) <= pos) {
167             node->par = pr;
168             split(node->right, node->right, r, pos - get_size(node->left) - 1,
169                 pl, node);
170             l = node;
171         } else {
172             node->par = pl;
173             split(node->left, l, node->left, pos, node, pr);

```

```

173         r = node;
174     }
175 }
176 update_node(node);
177 }
178
179 /// Merges to treaps (l and r) into a single one based on the rank of each
180 /// node.
181 void merge(Node *&node, Node *l, Node *r, Node *par = nullptr) {
182     #ifdef LAZY
183         propagate_lazy(l), propagate_lazy(r);
184     #endif
185     #ifdef REVERSE
186         propagate_reverse(l), propagate_reverse(r);
187     #endif
188     if (l == nullptr || r == nullptr)
189         node = (l == nullptr ? r : l);
190     else if (l->rank > r->rank) {
191         merge(l->right, l->right, r, l);
192         node = l;
193     } else {
194         merge(r->left, l, r->left, r);
195         node = r;
196     }
197     if (node)
198         node->par = par;
199     update_node(node);
200 }
201
202 Node *build(const int l, const int r, const vector<int> &arr,
203     vector<int> &rand) {
204     if (l > r)
205         return nullptr;
206
207     const int mid = (l + r) / 2;
208     Node *node = new Node(arr[mid], rand.back());
209     rand.pop_back();
210     node->right = build(mid + 1, r, arr, rand);
211     node->left = build(l, mid - 1, arr, rand);
212     update_node(node);
213
214     return node;
215 }
216
217 int _get_ith(const int idx) {
218     int ans = 0;
219     Node *cur = nodes[idx], *prev = nullptr;
220     while (cur) {
221         if (cur == nodes[idx] || prev == cur->right)
222             ans += 1 + get_size(cur->left);
223         prev = cur;
224         cur = cur->par;
225     }
226     return ans - 1;
227 }
228
229 vector<int> gen_rand(const int n) {
230     vector<int> ans(n);
231     for (int &x : ans)
232         x = rng();
233     sort(ans.begin(), ans.end());
234     return ans;
235 }
236
237 Node *_query(const int l, const int r) {

```



```

238     Node *L, *M, *R;
239     split(this->root, L, M, l - 1);
240     split(M, M, R, r - 1);
241     Node *ret = new Node(*M);
242     merge(L, L, M);
243     merge(root, L, R);
244     return ret;
245 }
246
247 void update(const int l, const int r, const int delta) {
248     Node *L, *M, *R;
249     split(this->root, L, M, l - 1);
250     split(M, M, R, r - 1);
251
252     Node *node = M;
253     #ifdef LAZY
254     node->lazy = delta;
255     propagate_lazy(node);
256     #else
257     node->val += delta;
258     #endif
259
260     merge(L, L, M);
261     merge(root, L, R);
262 }
263
264 void insert(const int pos, Node *node) {
265     this->_size += node->size;
266     Node *L, *R;
267     split(this->root, L, R, pos - 1);
268     merge(L, L, node);
269     merge(this->root, L, R);
270 }
271
272 Node *_erase(const int l, const int r) {
273     Node *L, *M, *R;
274     split(this->root, L, M, l - 1);
275     split(M, M, R, r - 1);
276     merge(root, L, R);
277     this->_size -= r - l + 1;
278     return M;
279 }
280
281 void _move(const int l, const int r, const int new_pos) {
282     Node *node = _erase(l, r);
283     _insert(new_pos, node);
284 }
285
286 #ifdef REVERSE
287 void _reverse(const int l, const int r) {
288     Node *L, *M, *R;
289     split(this->root, L, M, l - 1);
290     split(M, M, R, r - 1);
291
292     Node *node = M;
293     node->rev ^= true;
294
295     merge(L, L, M);
296     merge(root, L, R);
297 }
298 #endif
299
300 public:
301     Treap() {}
302

```

```

303     /// Constructor that initializes the treap based on an array.
304     ///
305     /// Time Complexity: O(n)
306     Treap(const vector<int> &arr) : _size(arr.size()) {
307         vector<int> r = gen_rand(arr.size());
308         this->root = build(0, (int)arr.size() - 1, arr, r);
309     }
310
311     int size() { return _size; }
312
313     /// Moves the subarray [l, r] to the position starting at new_pos.
314     /// new_pos represents the position BEFORE the subarray is deleted!!!
315     ///
316     /// Time Complexity: O(log n)
317     void move(const int l, const int r, int new_pos) {
318         assert(0 <= new_pos), assert(new_pos <= _size);
319         if(new_pos > l)
320             // after erase the index will be different if new_pos > l
321             new_pos -= r - l + 1;
322         _move(l, r, new_pos);
323     }
324
325     /// Moves the subarray [l, r] to the back of the array.
326     ///
327     /// Time Complexity: O(log n)
328     void move_back(const int l, const int r) {
329         assert(0 <= l), assert(l <= r), assert(r < _size);
330         move(l, r, _size);
331     }
332
333     /// Moves the subarray [l, r] to the front of the array.
334     ///
335     /// Time Complexity: O(log n)
336     void move_front(const int l, const int r) {
337         assert(0 <= l), assert(l <= r), assert(r < _size);
338         move(l, r, 0);
339     }
340
341     #ifdef REVERSE
342     /// Reverses the subarray [l, r].
343     ///
344     /// Time Complexity: O(log n)
345     void reverse(const int l, const int r) {
346         assert(0 <= l), assert(l <= r), assert(r < _size);
347         _reverse(l, r);
348     }
349     #endif
350
351     /// Erases the subarray [l, r].
352     ///
353     /// Time Complexity: O(log n)
354     void erase(const int l, const int r) {
355         assert(0 <= l), assert(l <= r), assert(r < _size);
356         _erase(l, r);
357     }
358
359     /// Inserts the value val at the position pos.
360     ///
361     /// Time Complexity: O(log n)
362     void insert(const int pos, const int val) {
363         assert(pos <= _size);
364         nodes.emplace_back(new Node(val));
365         _insert(pos, nodes.back());
366     }
367

```

```

368 /// Returns the index of the i-th added node.
369 ///
370 /// Time Complexity: O(log n)
371 int get_ith(const int idx) {
372     assert(0 <= idx), assert(idx < nodes.size());
373     return _get_ith(idx);
374 }
375
376 /// Sums the delta value to the position pos.
377 ///
378 /// Time Complexity: O(log n)
379 void update(const int pos, const int delta) {
380     assert(0 <= pos), assert(pos < _size);
381     _update(pos, pos, delta);
382 }
383
384 #ifdef LAZY
385 /// Sums the delta value to the subarray [l, r].
386 ///
387 /// Time Complexity: O(log n)
388 void update(const int l, const int r, const int delta) {
389     assert(0 <= l), assert(l <= r), assert(r < _size);
390     _update(l, r, delta);
391 }
392 #endif
393
394 /// Query at a single index.
395 ///
396 /// Time Complexity: O(log n)
397 int query(const int pos) {
398     assert(0 <= pos), assert(pos < _size);
399     return _query(pos, pos)->ans;
400 }
401
402 /// Range query from l to r.
403 ///
404 /// Time Complexity: O(log n)
405 int query(const int l, const int r) {
406     assert(0 <= l), assert(l <= r), assert(r < _size);
407     return _query(l, r)->ans;
408 }
409 };
410 // clang-format on

```

2.17. Treap Maximum Contiguous Segment

```

1 bool full(Node *node) { return node->cntl == get_size(node); }
2
3 // Range Sum
4 void merge_nodes(Node *node) {
5     node->ans = 1;
6     node->l = node->val;
7     node->cntl = 1;
8     node->r = node->val;
9     node->cntr = 1;
10
11     if (node->left) {
12         node->ans = max(node->ans, node->left->ans);
13         node->cntl = node->left->cntl;
14         node->l = node->left->l;
15         if (node->left->r == node->val) {
16             node->ans = max(node->ans, node->left->cntr + 1);
17             if (full(node->left))
18                 node->cntl = node->left->cntr + 1;

```

```

19         if (!node->right) {
20             node->r = node->val;
21             node->cntr = node->left->cntr + 1;
22         }
23     }
24 }
25
26 if (node->right) {
27     node->ans = max(node->ans, node->right->ans);
28     node->cntr = node->right->cntr;
29     node->r = node->right->r;
30     if (node->right->l == node->val) {
31         node->ans = max(node->ans, node->right->cntl + 1);
32         if (full(node->right))
33             node->cntr = node->right->cntl + 1;
34         if (!node->left) {
35             node->l = node->val;
36             node->cntl = node->right->cntl + 1;
37         }
38     }
39 }
40
41 if (node->left && node->right) {
42     node->ans = max({node->ans, node->left->ans, node->right->ans});
43     if (node->left->r == node->val && node->right->l == node->val) {
44         node->ans = max(node->ans, node->left->cntr + 1 + node->right->cntl);
45         if (full(node->left))
46             node->cntl = node->left->cntl + 1 + node->right->cntl;
47         if (full(node->right))
48             node->cntr = node->left->cntr + 1 + node->right->cntr;
49     }
50 }
51
52 node->ans = max({node->ans, node->cntl, node->cntr});
53 }

```

3. Dp

3.1. Achar Maior Palindromo

1 Fazer LCS da string com o reverso

3.2. Digit Dp

```

1 /// How many numbers x are there in the range a to b, where the digit d
2   occurs exactly k times in x?
3 vector<int> num;
4 int a, b, d, k;
5 int DP[12][12][2];
6 /// DP[p][c][f] = Number of valid numbers <= b from this state
7 /// p = current position from left side (zero based)
8 /// c = number of times we have placed the digit d so far
9 /// f = the number we are building has already become smaller than b? [0 =
10    no, 1 = yes]
11
12 int call(int pos, int cnt, int f){
13     if(cnt > k) return 0;
14
15     if(pos == num.size()){
16         if(cnt == k) return 1;
17         return 0;

```

```

18 if(DP[pos][cnt][f] != -1) return DP[pos][cnt][f];
19 int res = 0;
20 int lim = (f ? 9 : num[pos]);
21
22 /// Try to place all the valid digits such that the number doesn't exceed b
23 for(int dgt = 0; dgt<=LMT; dgt++){
24     int nf = f;
25     int ncnt = cnt;
26     if(f == 0 && dgt < LMT) nf = 1; /// The number is getting smaller at
        this position
27     if(dgt == d) ncnt++;
28     if(ncnt <= k) res += call(pos+1, ncnt, nf);
29 }
30
31 return DP[pos][cnt][f] = res;
32 }
33
34 int solve(int b){
35     num.clear();
36     while(b>0){
37         num.push_back(b%10);
38         b/=10;
39     }
40     reverse(num.begin(), num.end());
41     /// Stored all the digits of b in num for simplicity
42
43     memset(DP, -1, sizeof(DP));
44     int res = call(0, 0, 0);
45     return res;
46 }
47
48 int main () {
49
50     cin >> a >> b >> d >> k;
51     int res = solve(b) - solve(a-1);
52     cout << res << endl;
53
54     return 0;
55 }

```

3.3. Longest Common Subsequence

```

1 string lcs(string &s, string &t) {
2
3     int n = s.size(), m = t.size();
4
5     s.insert(s.begin(), '#');
6     t.insert(t.begin(), '$');
7
8     vector<vector<int>>> mat(n + 1, vector<int>(m + 1, 0));
9
10    for(int i = 1; i <= n; i++) {
11        for(int j = 1; j <= m; j++) {
12            if(s[i] == t[j])
13                mat[i][j] = mat[i - 1][j - 1] + 1;
14            else
15                mat[i][j] = max(mat[i - 1][j], mat[i][j - 1]);
16        }
17    }
18
19    string ans;
20    int i = n, j = m;
21    while(i > 0 && j > 0) {
22        if(s[i] == t[j])

```

```

23         ans += s[i], i--, j--;
24     else if(mat[i][j - 1] > mat[i - 1][j])
25         j--;
26     else
27         i--;
28 }
29
30 reverse(ans.begin(), ans.end());
31 return ans;
32 }

```

3.4. Longest Common Substring

```

1 int LCSuffStr(char *X, char *Y, int m, int n) {
2     /// Create a table to store lengths of longest common suffixes of
3     /// substrings. Notethat LCSuff[i][j] contains length of longest
4     /// common suffix of X[0..i-1] and Y[0..j-1]. The first row and
5     /// first column entries have no logical meaning, they are used only
6     /// for simplicity of program
7     int LCSuff[m+1][n+1];
8     int result = 0; /// To store length of the longest common substring
9
10    /* Following steps build LCSuff[m+1][n+1] in bottom up fashion. */
11    for (int i=0; i<=m; i++) {
12        for (int j=0; j<=n; j++) {
13            if (i == 0 || j == 0)
14                LCSuff[i][j] = 0;
15
16            else if (X[i-1] == Y[j-1]) {
17                LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
18                result = max(result, LCSuff[i][j]);
19            }
20            else LCSuff[i][j] = 0;
21        }
22    }
23    return result;
24 }

```

3.5. Longest Increasing Subsequence 2D (Not Sorted)

```

1 set<ii> s[(int)2e6];
2 bool check(ii par, int ind) {
3
4     auto it = s[ind].lower_bound(ii(par.ff, -INF));
5     if(it == s[ind].begin())
6         return false;
7
8     it--;
9
10    if(it->ss < par.ss)
11        return true;
12    return false;
13 }
14
15 int lis2d(vector<ii> &arr) {
16
17     int n = arr.size();
18     s[1].insert(arr[0]);
19
20     int maior = 1;
21     for(int i = 1; i < n; i++) {
22
23         ii x = arr[i];

```

```

24
25     int l = 1, r = maior;
26     int ansbb = 0;
27     while(l <= r) {
28         int mid = (l+r)/2;
29         if(check(x, mid)) {
30             l = mid + 1;
31             ansbb = mid;
32         } else {
33             r = mid - 1;
34         }
35     }
36
37     // inserting in list
38     auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
39     while(it != s[ansbb+1].end() && it->ss >= x.ss)
40         it = s[ansbb+1].erase(it);
41
42     it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
43     if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
44        it->ss <= x.ss)
45         continue;
46     s[ansbb+1].insert(arr[i]);
47
48     maior = max(maior, ansbb + 1);
49 }
50 return maior;
51
52 }

```

3.6. Longest Increasing Subsequence 2D (Sorted)

```

1  set<ii> s[(int)2e6];
2  bool check(ii par, int ind) {
3
4      auto it = s[ind].lower_bound(ii(par.ff, -INF));
5      if(it == s[ind].begin())
6          return false;
7
8      it--;
9
10     if(it->ss < par.ss)
11         return true;
12     return false;
13 }
14
15 int lis2d(vector<ii> &arr) {
16
17     int n = arr.size();
18     s[1].insert(arr[0]);
19
20     int maior = 1;
21     for(int i = 1; i < n; i++) {
22
23         ii x = arr[i];
24
25         int l = 1, r = maior;
26         int ansbb = 0;
27         while(l <= r) {
28             int mid = (l+r)/2;
29             if(check(x, mid)) {
30                 l = mid + 1;
31                 ansbb = mid;

```

```

32     } else {
33         r = mid - 1;
34     }
35 }
36
37 // inserting in list
38 auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
39 while(it != s[ansbb+1].end() && it->ss >= x.ss)
40     it = s[ansbb+1].erase(it);
41
42 it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
43 if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
44    it->ss <= x.ss)
45     continue;
46 s[ansbb+1].insert(arr[i]);
47
48 maior = max(maior, ansbb + 1);
49 }
50 return maior;
51
52 }

```

3.7. Subset Sum Com Bitset

```

1  bitset<312345> bit;
2  int arr[112345];
3  void subsetSum(int n) {
4      bit.reset();
5      bit.set(0);
6      for(int i = 0; i < n; i++) {
7          bit |= (bit << arr[i]);
8      }
9  }

```

3.8. Catalan

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^n \frac{n+k}{k} \quad \text{para } n \geq 0.$$

3.9. Catalan 1 1 2 5 14 42 132

```

1  // The first few Catalan numbers for n = 0, 1, 2, 3, ...
2  // are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
3  // Formula Recursiva:
4  // cat(0) = 0
5  // cat(n+1) = sum(i from 0 to n) (cat(i)*cat(n-i))
6  //
7  // Using Binomial Coefficient
8  // We can also use the below formula to find nth catalan number in O(n) time.
9
10 // Returns value of Binomial Coefficient C(n, k)
11
12 // REQUIRES combinatorics.cpp
13 int catalan(int n) {
14     return comb.fat(2 * n) * comb.inv(comb.fat(n + 1)) % MOD *
15            comb.inv(comb.fat(n)) % MOD;

```

16 }

3.10. Cht Optimization

```

1  /// Copied from:
2  ///
3  https://github.com/kth-competitive-programming/kactl/content/data-structures/LineContainer.h
4  // clang-format off
5  /// Uncomment the line below to get the minimum answer, otherwise it will
6  /// try to
7  /// get the maximum answer.
8  /// #define MINIMUM
9  struct Line {
10     // f(x) = aX + b
11     mutable int a, b, p;
12     bool operator<(const Line &o) const { return a < o.a; }
13     bool operator<(int x) const { return p < x; }
14     // Use the methods below to get the real value of the attributes!!!
15     int get_a() {
16         #ifdef MINIMUM
17             return -a;
18         #else
19             return a;
20         #endif
21     }
22     int get_b() {
23         #ifdef MINIMUM
24             return -b;
25         #else
26             return b;
27         #endif
28     }
29 }
30 struct LineContainer : multiset<Line, less<>> {
31     // (for doubles, use inf = 1/.0, div(a,b) = a/b)
32     static const int inf = LLONG_MAX;
33     int div(int a, int b) { // floored division
34         return a / b - ((a ^ b) < 0 && a % b);
35     }
36     bool isect(iterator x, iterator y) {
37         if (y == end())
38             return x->p = inf, 0;
39         if (x->a == y->a)
40             x->p = x->b > y->b ? inf : -inf;
41         else
42             x->p = div(y->b - x->b, x->a - y->a);
43         return x->p >= y->p;
44     }
45     /// Inserts the line a * x + b.
46     ///
47     /// Time Complexity: O(log n)
48     void add(int a, int b) {
49         #ifdef MINIMUM
50             a = -a, b = -b;
51         #endif
52         auto z = insert({a, b, 0}), y = z++, x = y;
53         while (isect(y, z))
54             z = erase(z);
55         if (x != begin() && isect(--x, y))
56             isect(x, y = erase(y));
57         while ((y = x) != begin() && (--x)->p >= y->p)
58             isect(x, erase(y));

```

```

59     }
60     /// Query the best line such that a * x + b is maximum/minimum.
61     ///
62     /// Time Complexity: O(log n)
63     int query(int x) {
64         assert(!empty());
65         auto l = *lower_bound(x);
66         #ifdef MINIMUM
67             return -(l.a * x + l.b);
68         #else
69             return l.a * x + l.b;
70         #endif
71     }
72 };
73 // clang-format on

```

3.11. Coin Change Problem

```

1  // função que recebe o valor de troco N, o número de moedas disponíveis M,
2  // e um vetor com as moedas disponíveis arr
3  // essa função deve retornar o número mínimo de moedas,
4  // de acordo com a solução com Programação Dinâmica.
5  int num_moedas(int N, int M, int arr[]) {
6      int dp[N+1];
7      // caso base
8      dp[0] = 0;
9      // sub-problemas
10     for(int i=1; i<=N; i++) {
11         // é comum atribuir um valor alto, que concerteza
12         // é maior que qualquer uma das próximas possibilidades,
13         // sendo assim substituído
14         dp[i] = 1000000;
15         for(int j=0; j<M; j++) {
16             if(i-arr[j] >= 0) {
17                 dp[i] = min(dp[i], dp[i-arr[j]]+1);
18             }
19         }
20     }
21     // solução
22     return dp[N];
23 }

```

3.12. Divide And Conquer Optimization

```

1  /// Problem: Split the array into k buckets such that the cost of each
2  /// bucket is
3  /// the square sum of each subarray.
4  ///
5  /// resize below
6  vector<int> last(MAXN, INF), dp(MAXN, INF);
7  /// Cost for the subarray (l, r).
8  int C(int l, int r) {
9      int val = 0;
10     val += sq(sum(l, r));
11     return val;
12 }
13 /// dp[i] represents the cost of splitting the array into k buckets (after k
14 /// iterations), with the index i as the last index.
15 ///
16 /// Time Complexity: O(n*k*(log n))
17 void f(int l, int r, int optl, int optpr) {
18     if (l > r)

```

```

19     return;
20
21     int mid = (l + r) / 2;
22     auto best = make_pair(INF, INF); // change to (-INF, -INF) to maximize
23     // change mid - 1 to mid if buckets can intercept.
24     for (int i = optl; i <= min(mid, optl); ++i)
25         best = min(best, {(i == 1 ? 0 : last[i - 1]) + C(i, mid), i});
26
27     dp[mid] = best.first;
28     const int opt = best.second;
29
30     f(l, mid - 1, optl, opt);
31     f(mid + 1, r, opt, optl);
32 }
33
34 // 1-indexed, change to 0-index if necessary.
35 int compute(const int k) {
36     for (int i = 0; i < k; ++i) {
37         f(1, n, 1, n);
38         last.swap(dp);
39     }
40
41     return last[n];
42 }

```

3.13. Edit Distance

```

1 // Returns the minimum number of operations (insert, remove and delete) to
2 // convert a into b.
3 //
4 // Time Complexity: O(a.size() * b.size())
5 int edit_distance(const string &a, const string &b) {
6     int n = a.size(), m = b.size();
7     int dp[2][n + 1];
8     memset(dp, 0, sizeof dp);
9     for (int i = 0; i <= n; i++)
10         dp[0][i] = i;
11     for (int i = 1; i <= m; i++)
12         for (int j = 0; j <= n; j++) {
13             if (j == 0)
14                 dp[i & 1][j] = i;
15             else if (a[j - 1] == b[i - 1])
16                 dp[i & 1][j] = dp[(i & 1) ^ 1][j - 1];
17             else
18                 dp[i & 1][j] = 1 + min({dp[(i & 1) ^ 1][j], dp[i & 1][j - 1],
19                                     dp[(i & 1) ^ 1][j - 1]});
20         }
21     return dp[m & 1][n];
22 }

```

3.14. Knapsack

```

1 int dp[2001][2001];
2 int moc(int q, int p, vector<ii> vec) {
3     for (int i = 1; i <= q; i++)
4     {
5         for (int j = 1; j <= p; j++) {
6             if (j >= vec[i-1].ff)
7                 dp[i][j] = max(dp[i-1][j], vec[i-1].ss + dp[i-1][j-vec[i-1].ff]);
8             else
9                 dp[i][j] = dp[i-1][j];
10        }
11    }

```

```

12     return dp[q][p];
13 }
14 int main(int argc, char *argv[])
15 {
16     int p, q;
17     vector<ii> vec;
18     cin >> p >> q;
19     int x, y;
20     for (int i = 0; i < q; i++) {
21         cin >> x >> y;
22         vec.push_back(make_pair(x, y));
23     }
24     for (int i = 0; i <= p; i++)
25         dp[0][i] = 0;
26     for (int i = 1; i <= q; i++)
27         dp[i][0] = 0;
28     sort(vec.begin(), vec.end());
29     cout << moc(q, p, vec) << endl;
30 }

```

3.15. Knuth Optimization

```

1 // Problem: Given an array of n numbers split the array until get n
2 // subarrays
3 // of one element. The cost of each split is the sum of values in the
4 // subarray.
5 // Time Complexity: O(n^2)
6 void knuth() {
7     // length of the cut
8     for (int i = 0; i < n; ++i)
9         // cutting from j to j + i
10         for (int j = 0; j + i < n; ++j) {
11             if (i == 0) {
12                 dp[j][i + j] = 0;
13                 idx[j][i + j] = j;
14             } else {
15                 dp[j][j + i] = INF;
16                 // searching for the optimal place to cut
17                 for (int k = idx[j][j + i - 1]; k <= min(j + i - 1, idx[j + 1][i +
18                     j]);
19                     ++k) {
20                         int val = dp[j][k] + dp[k + 1][j + i] + sum(j, j + i);
21                         if (val < dp[j][j + i]) {
22                             dp[j][j + i] = val;
23                             idx[j][j + i] = k;
24                         }
25                     }
26             }
27         }
28     }
29 }

```

3.16. Lis

```

1 int lis(vector<int> &arr) {
2     int n = arr.size();
3     vector<int> lis;
4     for (int i = 0; i < n; i++) {
5         int l = 0, r = (int)lis.size() - 1;
6         int ans = -1;
7         while (l <= r) {
8             int mid = (l + r) / 2;
9             // OBS: - To >= LIS change to the operation below to >

```

```

10 // - Put <= or >= for strictly!!
11 if (arr[i] < lis[mid]) {
12     r = mid - 1;
13     ans = mid;
14 } else
15     l = mid + 1;
16 }
17 if (ans == -1)
18     lis.emplace_back(arr[i]);
19 else
20     lis[ans] = arr[i];
21 }
22
23 return lis.size();
24 }

```

4. Geometry

4.1. Centro De Massa De Um Poligono

```

1 double area = 0;
2 pto c;
3
4 c.x = c.y = 0;
5 for(int i = 0; i < n; i++) {
6     double aux = (arr[i].x * arr[i+1].y) - (arr[i].y * arr[i+1].x); // shoelace
7     area += aux;
8     c.x += aux*(arr[i].x + arr[i+1].x);
9     c.y += aux*(arr[i].y + arr[i+1].y);
10 }
11
12 c.x /= (3.0*area);
13 c.y /= (3.0*area);
14
15 cout << c.x << ' ' << c.y << endl;

```

4.2. Closest Pair Of Points

```

1 struct Point {
2     int x, y;
3 };
4 int compareX(const void *a, const void *b) {
5     Point *p1 = (Point *)a, *p2 = (Point *)b;
6     return (p1->x - p2->x);
7 }
8 int compareY(const void *a, const void *b) {
9     Point *p1 = (Point *)a, *p2 = (Point *)b;
10    return (p1->y - p2->y);
11 }
12 float dist(Point p1, Point p2) {
13     return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
14 }
15 float bruteForce(Point P[], int n) {
16     float min = FLT_MAX;
17     for (int i = 0; i < n; ++i)
18         for (int j = i+1; j < n; ++j)
19             if (dist(P[i], P[j]) < min)
20                 min = dist(P[i], P[j]);
21     return min;
22 }
23 float min(float x, float y) {
24     return (x < y) ? x : y;
25 }

```

```

26 float stripClosest(Point strip[], int size, float d) {
27     float min = d;
28     for (int i = 0; i < size; ++i)
29         for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
30             if (dist(strip[i], strip[j]) < min)
31                 min = dist(strip[i], strip[j]);
32     return min;
33 }
34 float closestUtil(Point Px[], Point Py[], int n) {
35     if (n <= 3)
36         return bruteForce(Px, n);
37     int mid = n/2;
38     Point midPoint = Px[mid];
39     Point Pyl[mid+1];
40     Point Pyr[n-mid-1];
41     int li = 0, ri = 0;
42     for (int i = 0; i < n; i++)
43         if (Py[i].x <= midPoint.x)
44             Pyl[li++] = Py[i];
45         else
46             Pyr[ri++] = Py[i];
47
48     float dl = closestUtil(Px, Pyl, mid);
49     float dr = closestUtil(Px + mid, Pyr, n-mid);
50     float d = min(dl, dr);
51     Point strip[n];
52     int j = 0;
53     for (int i = 0; i < n; i++)
54         if (abs(Py[i].x - midPoint.x) < d)
55             strip[j] = Py[i], j++;
56     return min(d, stripClosest(strip, j, d));
57 }
58
59 float closest(Point P[], int n) {
60     Point Px[n];
61     Point Py[n];
62     for (int i = 0; i < n; i++) {
63         Px[i] = P[i];
64         Py[i] = P[i];
65     }
66     qsort(Px, n, sizeof(Point), compareX);
67     qsort(Py, n, sizeof(Point), compareY);
68     return closestUtil(Px, Py, n);
69 }

```

4.3. Condicao De Existencia De Um Triangulo

```

1
2 | b - c | < a < b + c
3 | a - c | < b < a + c
4 | a - b | < c < a + b
5
6 Para a < b < c, basta checar
7   a + b > c
8
9 OBS: Para um conjunto n >= 100 sempre existe um triângulo válido, pois a
    sequência de triângulos não válidos segue a sequência de Fibonacci e
    Fib(100) > 2^64

```

4.4. Convex Hull

```

1 // Asymptotic complexity: O(n log n).
2 struct pto {

```

```

3 double x, y;
4 bool operator <(const pto &p) const {
5     return x < p.x || (x == p.x && y < p.y);
6     /* a impressao será em prioridade por mais a esquerda, mais
7        abaixo, e antihorário pelo cross abaixo */
8 }
9 };
10
11 double cross(const pto &O, const pto &A, const pto &B) {
12     return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
13 }
14
15 vector<pto> convex_hull(vector<pto> P) {
16     int n = P.size(), k = 0;
17     vector<pto> H(2 * n);
18     // Sort points lexicographically
19     sort(P.begin(), P.end());
20     // Build lower hull
21     for (int i = 0; i < n; ++i) {
22         // esse <= 0 representa sentido anti-horario, caso deseje mudar
23         // trocar por >= 0
24         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
25             k--;
26         H[k++] = P[i];
27     }
28     // Build upper hull
29     for (int i = n - 2, t = k + 1; i >= 0; i--) {
30         // esse <= 0 representa sentido anti-horario, caso deseje mudar
31         // trocar por >= 0
32         while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
33             k--;
34         H[k++] = P[i];
35     }
36     H.resize(k);
37     /* o último ponto do vetor é igual ao primeiro, atente para isso
38        as vezes é necessário mudar */
39     return H;
40 }

```

4.5. Cross Product

```

1 // Outra forma de produto vetorial
2 // reta ab,ac se for zero e colinear
3 // se for < 0 entao antiHorario, > 0 horario
4 bool ehcol(pto a,pto b,pto c) {
5     return ((b.y-a.y)*(c.x-a.x) - (b.x-a.x)*(c.y-a.y));
6 }
7
8 //Produto vetorial AB x AC, se for zero e colinear
9 int cross(pto A, pto B, pto C){
10     pto AB, AC;
11     AB.x = B.x-A.x;
12     AB.y = B.y-A.y;
13     AC.x = C.x-A.x;
14     AC.y = C.y-A.y;
15     int cross = AB.x*AC.y-AB.y * AC.x;
16     return cross;
17 }
18
19 // OBS: DEFINE ÁREA DE QUADRILÁTERO FORMADO PELAS RETAS, A ÁREA DO TRIÂNGULO
    É A METADE

```

4.6. Distance Point Segment

```

1 // use struct point and line
2 double dist_point_segment(const Point p, const Point s, const Point t) {
3     if (sgn(dot(p-s, t-s)) < 0)
4         return (p-s).norm();
5     if (sgn(dot(p-t, s-t)) < 0)
6         return (p-t).norm();
7     return abs(det(s-p, t-p) / dist(s, t));
8 }

```

4.7. Line-Line Intersection

```

1 // Intersecção de retas Ax + By = C    dados pontos (x1,y1) e (x2,y2)
2 A = y2-y1
3 B = x1-x2
4 C = A*x1+B*y1
5 //Retas definidas pelas equações:
6 A1x + B1y = C1
7 A2x + B2y = C2
8 //Encontrar x e y resolvendo o sistema
9 double det = A1*B2 - A2*B1;
10 if (det == 0) {
11     //Lines are parallel
12 } else {
13     double x = (B2*C1 - B1*C2)/det;
14     double y = (A1*C2 - A2*C1)/det;
15 }

```

4.8. Line-Point Distance

```

1 double ptoReta(double x1, double y1, double x2, double y2, double pointX,
2               double pointY, double *ptox, double *ptoy) {
3     double diffX = x2 - x1;
4     double diffY = y2 - y1;
5     if ((diffX == 0) && (diffY == 0)) {
6         diffX = pointX - x1;
7         diffY = pointY - y1;
8         //se os dois sao pontos
9         return hypot(pointX - x1, pointY - y1);
10    }
11    double t = ((pointX - x1) * diffX + (pointY - y1) * diffY) /
12              (diffX * diffX + diffY * diffY);
13    if (t < 0) {
14        //point is nearest to the first point i.e x1 and y1
15        // Ex:
16        // cord do pto na reta = pto inicial(x1,y1);
17        *ptox = x1, *ptoy = y1;
18        diffX = pointX - x1;
19        diffY = pointY - y1;
20    } else if (t > 1) {
21        //point is nearest to the end point i.e x2 and y2
22        // Ex :
23        // cord do pto na reta = pto final(x2,y2);
24        *ptox = x2, *ptoy = y2;
25        diffX = pointX - x2;
26        diffY = pointY - y2;
27    } else {
28        //if perpendicular line intersect the line segment.
29        // pto nao esta mais proximo de uma das bordas do segmento
30        // Ex:
31        //
32        //

```



```

33 // cord x do pto na reta = (x1 + t * diffX)
34 // cord y do pto na reta = (y1 + t * diffY)
35 *ptox = (x1 + t * diffX), *ptoy = (y1 + t * diffY);
36 diffX = pointX - (x1 + t * diffX);
37 diffY = pointY - (y1 + t * diffY);
38 }
39 //returning shortest distance
40 return sqrt(diffX * diffX + diffY * diffY);
41 }

```

4.9. Point Inside Convex Polygon - Log(N)

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define INF 1e18
6 #define pb push_back
7 #define ii pair<int,int>
8 #define OK cout<<"OK"<<endl
9 #define debug(x) cout << #x " = " << (x) << endl
10 #define ff first
11 #define ss second
12 #define int long long
13
14 struct pto {
15     double x, y;
16     bool operator <(const pto &p) const {
17         return x < p.x || (x == p.x && y < p.y);
18         /* a impressao será em prioridade por mais a esquerda, mais
19         abaixo, e antihorário pelo cross abaixo */
20     }
21 };
22 double cross(const pto &O, const pto &A, const pto &B) {
23     return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
24 }
25
26 vector<pto> lower, upper;
27
28 vector<pto> convex_hull(vector<pto> &P) {
29     int n = P.size(), k = 0;
30     vector<pto> H(2 * n);
31     // Sort points lexicographically
32     sort(P.begin(), P.end());
33     // Build lower hull
34     for (int i = 0; i < n; ++i) {
35         // esse <= 0 representa sentido anti-horario, caso deseje mudar
36         // trocar por >= 0
37         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
38             k--;
39         H[k++] = P[i];
40     }
41     // Build upper hull
42     for (int i = n - 2, t = k + 1; i >= 0; i--) {
43         // esse <= 0 representa sentido anti-horario, caso deseje mudar
44         // trocar por >= 0
45         while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
46             k--;
47         H[k++] = P[i];
48     }
49     H.resize(k);
50     /* o último ponto do vetor é igual ao primeiro, atente para isso
51     as vezes é necessário mudar */
52

```

```

53 int j = 1;
54 lower.pb(H.front());
55 while(H[j].x >= H[j-1].x) {
56     lower.pb(H[j++]);
57 }
58
59 int l = H.size()-1;
60 while(l >= j) {
61     upper.pb(H[l--]);
62 }
63 upper.pb(H[l--]);
64
65 return H;
66 }
67
68 bool insidePolygon(pto p, vector<pto> &arr) {
69
70     if(pair<double,double>(p.x, p.y) == pair<double,double>(lower[0].x,
71         lower[0].y))
72         return true;
73
74     pto lo = {p.x, -(double)INF};
75     pto hi = {p.x, (double)INF};
76     auto itl = lower_bound(lower.begin(), lower.end(), lo);
77     auto itu = lower_bound(upper.begin(), upper.end(), lo);
78
79     if(itl == lower.begin() || itu == upper.begin()) {
80         auto it = lower_bound(arr.begin(), arr.end(), lo);
81         auto it2 = lower_bound(arr.begin(), arr.end(), hi);
82         it2--;
83         if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y
84             <= it2->y)
85             return true;
86         return false;
87     }
88     if(itl == lower.end() || itu == upper.end()) {
89         return false;
90     }
91     auto ol = itl, ou = itu;
92     ol--, ou--;
93     if(cross(*ol, *itl, p) >= 0 && cross(*ou, *itu, p) <= 0)
94         return true;
95
96     auto it = lower_bound(arr.begin(), arr.end(), lo);
97     auto it2 = lower_bound(arr.begin(), arr.end(), hi);
98     it2--;
99     if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y <=
100         it2->y)
101         return true;
102     return false;
103 }
104
105 signed main () {
106     ios_base::sync_with_stdio(false);
107     cin.tie(NULL);
108
109     double n, m, k;
110
111     cin >> n >> m >> k;
112
113     vector<pto> arr(n);
114

```

```

115
116 for(pto &x: arr) {
117     cin >> x.x >> x.y;
118 }
119
120 convex_hull(arr);
121
122 pto p;
123
124 int c = 0;
125 while(m--) {
126     cin >> p.x >> p.y;
127     cout << (insidePolygon(p, arr) ? "dentro" : "fora") << endl;
128 }
129
130 }

```

4.10. Point Inside Polygon

```

1
2 /* Traça-se uma reta do ponto até um outro ponto qualquer fora do triangulo
   e checa o número de interseção com a borda do polígono se este for impar
   então está dentro se não está fora */
3
4 // Define Infinite (Using INT_MAX caused overflow problems)
5 #define INF 10000
6
7 struct pto {
8     int x, y;
9     pto() {}
10    pto(int x, int y) : x(x), y(y) {}
11 };
12
13 // Given three colinear ptos p, q, r, the function checks if
14 // pto q lies on line segment 'pr'
15 bool onSegment(pto p, pto q, pto r) {
16     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
17         q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
18         return true;
19     return false;
20 }
21
22 // To find orientation of ordered triplet (p, q, r).
23 // The function returns following values
24 // 0 --> p, q and r are colinear
25 // 1 --> Clockwise
26 // 2 --> Counterclockwise
27 int orientation(pto p, pto q, pto r) {
28     int val = (q.y - p.y) * (r.x - q.x) -
29             (q.x - p.x) * (r.y - q.y);
30
31     if (val == 0) return 0; // colinear
32     return (val > 0) ? 1 : 2; // clock or counterclock wise
33 }
34
35 // The function that returns true if line segment 'p1q1'
36 // and 'p2q2' intersect.
37 bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
38     // Find the four orientations needed for general and
39     // special cases
40     int o1 = orientation(p1, q1, p2);
41     int o2 = orientation(p1, q1, q2);
42     int o3 = orientation(p2, q2, p1);
43     int o4 = orientation(p2, q2, q1);

```

```

44
45 // General case
46 if (o1 != o2 && o3 != o4)
47     return true;
48
49 // Special Cases
50 // p1, q1 and p2 are colinear and p2 lies on segment p1q1
51 if (o1 == 0 && onSegment(p1, p2, q1)) return true;
52
53 // p1, q1 and p2 are colinear and q2 lies on segment p1q1
54 if (o2 == 0 && onSegment(p1, q2, q1)) return true;
55
56 // p2, q2 and p1 are colinear and p1 lies on segment p2q2
57 if (o3 == 0 && onSegment(p2, p1, q2)) return true;
58
59 // p2, q2 and q1 are colinear and q1 lies on segment p2q2
60 if (o4 == 0 && onSegment(p2, q1, q2)) return true;
61
62 return false; // Doesn't fall in any of the above cases
63 }
64
65 // Returns true if the pto p lies inside the polygon[] with n vertices
66 bool isInside(pto polygon[], int n, pto p) {
67     // There must be at least 3 vertices in polygon[]
68     if (n < 3) return false;
69
70     // Create a pto for line segment from p to infinite
71     pto extreme = pto(INF, p.y);
72
73     // Count intersections of the above line with sides of polygon
74     int count = 0, i = 0;
75     do {
76         int next = (i+1)%n;
77
78         // Check if the line segment from 'p' to 'extreme' intersects
79         // with the line segment from 'polygon[i]' to 'polygon[next]'
80         if (doIntersect(polygon[i], polygon[next], p, extreme)) {
81             // If the pto 'p' is colinear with line segment 'i-next',
82             // then check if it lies on segment. If it lies, return true,
83             // otherwise false
84             if (orientation(polygon[i], p, polygon[next]) == 0)
85                 return onSegment(polygon[i], p, polygon[next]);
86
87             count++;
88         }
89         i = next;
90     } while (i != 0);
91
92     // Return true if count is odd, false otherwise
93     return count%2 == 1; // Same as (count%2 == 1)
94 }

```

4.11. Points Inside And In Boundary Polygon

```

1 int cross(pto a, pto b) {
2     return a.x * b.y - b.x * a.y;
3 }
4
5 int boundaryCount(pto a, pto b) {
6     if(a.x == b.x)
7         return abs(a.y-b.y)-1;
8     if(a.y == b.y)
9         return abs(a.x-b.x)-1;
10    return _gcd(abs(a.x-b.x), abs(a.y-b.y))-1;

```

```

11 }
12
13 int totalBoundaryPolygon(vector<pto> &arr, int n) {
14
15     int boundPoint = n;
16     for(int i = 0; i < n; i++) {
17         boundPoint += boundaryCount(arr[i], arr[(i+1)%n]);
18     }
19     return boundPoint;
20 }
21
22 int polygonArea2(vector<pto> &arr, int n) {
23     int area = 0;
24     // N = quantidade de pontos no polígono e armazenados em p;
25     // OBS: VALE PARA CONVEXO E NÃO CONVEXO
26     for(int i = 0; i < n; i++){
27         area += cross(arr[i], arr[(i+1)%n]);
28     }
29     return abs(area);
30 }
31
32 int internalCount(vector<pto> &arr, int n) {
33
34     int area_2 = polygonArea2(arr, n);
35     int boundPoints = totalBoundaryPolygon(arr,n);
36     return (area_2 - boundPoints + 2)/2;
37 }

```

4.12. Polygon Area (3D)

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 struct point{
6     double x,y,z;
7     void operator=(const point & b){
8         x = b.x;
9         y = b.y;
10        z = b.z;
11    }
12 };
13
14 point cross(point a, point b){
15     point ret;
16     ret.x = a.y*b.z - b.y*a.z;
17     ret.y = a.z*b.x - a.x*b.z;
18     ret.z = a.x*b.y - a.y*b.x;
19     return ret;
20 }
21
22 int main(){
23     int num;
24     cin >> num;
25     point v[num];
26     for(int i=0; i<num; i++) cin >> v[i].x >> v[i].y >> v[i].z;
27
28     point cur;
29     cur.x = 0, cur.y = 0, cur.z = 0;
30
31     for(int i=0; i<num; i++){
32         point res = cross(v[i], v[(i+1)%num]);
33         cur.x += res.x;
34         cur.y += res.y;

```

```

35         cur.z += res.z;
36     }
37
38     double ans = sqrt(cur.x*cur.x + cur.y*cur.y + cur.z*cur.z);
39
40     double area = abs(ans);
41
42     cout << fixed << setprecision(9) << area/2. << endl;
43 }

```

4.13. Polygon Area

```

1 double polygonArea(vector<int> &X, vector<int> &Y, int n) {
2     int area = 0;
3     int j = n - 1;
4     for (int i = 0; i < n; i++) {
5         area += (X[j] + X[i]) * (Y[j] - Y[i]);
6         j = i;
7     }
8     return abs(area / 2.0);
9 }

```

4.14. Segment-Segment Intersection

```

1 // Given three colinear points p, q, r, the function checks if
2 // point q lies on line segment 'pr'
3 int onSegment(Point p, Point q, Point r) {
4     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y <= max(p.y, r.y)
5         && q.y >= min(p.y, r.y))
6         return true;
7     return false;
8 }
9 /* PODE SER RETIRADO
10 int onSegmentNotBorda(Point p, Point q, Point r) {
11     if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) && q.y <= max(p.y, r.y)
12         && q.y >= min(p.y, r.y))
13         return true;
14     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y < max(p.y, r.y)
15         && q.y > min(p.y, r.y))
16         return true;
17     return false;
18 }
19 */
20 // To find orientation of ordered triplet (p, q, r).
21 // The function returns following values
22 // 0 --> p, q and r are colinear
23 // 1 --> Clockwise
24 // 2 --> Counterclockwise
25 int orientation(Point p, Point q, Point r) {
26     int val = (q.y - p.y) * (r.x - q.x) -
27             (q.x - p.x) * (r.y - q.y);
28     if (val == 0) return 0; // colinear
29     return (val > 0)? 1: 2; // clock or counterclock wise
30 }
31 // The main function that returns true if line segment 'p1p2'
32 // and 'q1q2' intersect.
33 int doIntersect(Point p1, Point p2, Point q1, Point q2) {
34     // Find the four orientations needed for general and
35     // special cases
36     int o1 = orientation(p1, p2, q1);
37     int o2 = orientation(p1, p2, q2);
38     int o3 = orientation(q1, q2, p1);
39     int o4 = orientation(q1, q2, p2);

```

```

37 // General case
38 if (o1 != o2 && o3 != o4) return 2;
39
40 /* PODE SER RETIRADO
41 if(o1 == o2 && o2 == o3 && o3 == o4 && o4 == 0) {
42 //INTERCEPTAM EM RETA
43 if(onSegmentNotBorda(p1,q1,p2) || onSegmentNotBorda(p1,q2,p2)) return 1;
44 if(onSegmentNotBorda(q1,p1,q2) || onSegmentNotBorda(q1,p2,q2)) return 1;
45 }
46 */
47 // Special Cases (INTERCEPTAM EM PONTO)
48 // p1, p2 and q1 are colinear and q1 lies on segment p1p2
49 if (o1 == 0 && onSegment(p1, q1, p2)) return 2;
50 // p1, p2 and q1 are colinear and q2 lies on segment p1p2
51 if (o2 == 0 && onSegment(p1, q2, p2)) return 2;
52 // q1, q2 and p1 are colinear and p1 lies on segment q1q2
53 if (o3 == 0 && onSegment(q1, p1, q2)) return 2;
54 // q1, q2 and p2 are colinear and p2 lies on segment q1q2
55 if (o4 == 0 && onSegment(q1, p2, q2)) return 2;
56 return false; // Doesn't fall in any of the above cases
57 }
58 // OBS: SE (C2/A2 == C1/A1) SÃO COLINEARES
59

```

4.15. Upper And Lower Hull

```

1 struct pto {
2     double x, y;
3     bool operator <(const pto &p) const {
4         return x < p.x || (x == p.x && y < p.y);
5         /* a impressao será em prioridade por mais a esquerda, mais
6            abaixo, e antihorário pelo cross abaixo */
7     }
8 };
9 double cross(const pto &O, const pto &A, const pto &B) {
10     return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
11 }
12 vector<pto> lower, upper;
13
14 vector<pto> convex_hull(vector<pto> &P) {
15     int n = P.size(), k = 0;
16     vector<pto> H(2 * n);
17     // Sort points lexicographically
18     sort(P.begin(), P.end());
19     // Build lower hull
20     for (int i = 0; i < n; ++i) {
21         // esse <= 0 representa sentido anti-horario, caso deseje mudar
22         // trocar por >= 0
23         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
24             k--;
25         H[k++] = P[i];
26     }
27     // Build upper hull
28     for (int i = n - 2, t = k + 1; i >= 0; i--) {
29         // esse <= 0 representa sentido anti-horario, caso deseje mudar
30         // trocar por >= 0
31         while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
32             k--;
33         H[k++] = P[i];
34     }
35     H.resize(k);
36     /* o último ponto do vetor é igual ao primeiro, atente para isso
37        as vezes é necessário mudar */
38

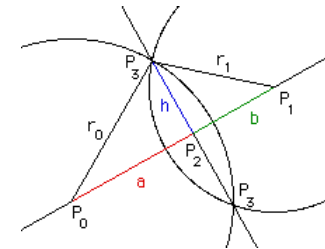
```

```

39 int j = 1;
40 lower.pb(H.front());
41 while(H[j].x >= H[j-1].x) {
42     lower.pb(H[j++]);
43 }
44
45 int l = H.size()-1;
46 while(l >= j) {
47     upper.pb(H[l--]);
48 }
49 upper.pb(H[l--]);
50
51 return H;
52
53 }

```

4.16. Circle Circle Intersection



4.17. Circle Circle Intersection

```

1 /* circle_circle_intersection() *
2  * Determine the points where 2 circles in a common plane intersect.
3  *
4  * int circle_circle_intersection(
5  *     // center and radius of 1st circle
6  *     double x0, double y0, double r0,
7  *     // center and radius of 2nd circle
8  *     double x1, double y1, double r1,
9  *     // 1st intersection point
10 *     double *xi, double *yi,
11 *     // 2nd intersection point
12 *     double *xi_prime, double *yi_prime)
13 *
14 * This is a public domain work. 3/26/2005 Tim Voght
15 *
16 */
17
18 int circle_circle_intersection(double x0, double y0, double r0, double x1,
19                               double y1, double r1, double *xi, double *yi,
20                               double *xi_prime, double *yi_prime) {
21     double a, dx, dy, d, h, rx, ry;
22     double x2, y2;
23
24     /* dx and dy are the vertical and horizontal distances between
25        * the circle centers.
26        */
27     dx = x1 - x0;
28     dy = y1 - y0;
29

```

```

30  /* Determine the straight-line distance between the centers. */
31  // d = sqrt((dy*dy) + (dx*dx));
32  d = hypot(dx, dy); // Suggested by Keith Briggs
33
34  /* Check for solvability. */
35  if (d > (r0 + r1)) {
36      /* no solution. circles do not intersect. */
37      return 0;
38  }
39  if (d < fabs(r0 - r1)) {
40      /* no solution. one circle is contained in the other */
41      return 0;
42  }
43
44  /* 'point 2' is the point where the line through the circle
45  * intersection points crosses the line between the circle
46  * centers.
47  */
48
49  /* Determine the distance from point 0 to point 2. */
50  a = ((r0 * r0) - (r1 * r1) + (d * d)) / (2.0 * d);
51
52  /* Determine the coordinates of point 2. */
53  x2 = x0 + (dx * a / d);
54  y2 = y0 + (dy * a / d);
55
56  /* Determine the distance from point 2 to either of the
57  * intersection points.
58  */
59  h = sqrt((r0 * r0) - (a * a));
60
61  /* Now determine the offsets of the intersection points from
62  * point 2.
63  */
64  rx = -dy * (h / d);
65  ry = dx * (h / d);
66
67  /* Determine the absolute intersection points. */
68  *xi = x2 + rx;
69  *xi_prime = x2 - rx;
70  *yi = y2 + ry;
71  *yi_prime = y2 - ry;
72
73  return 1;
74  }

```

4.18. Struct Point And Line

```

1  int sgn(double x) {
2      if(abs(x) < 1e-8) return 0;
3      return x > 0 ? 1 : -1;
4  }
5  inline double sqr(double x) { return x * x; }
6
7  struct Point {
8      double x, y, z;
9      Point() {};
10     Point(double a, double b): x(a), y(b) {};
11     Point (double x, double y, double z): x(x), y(y), z(z) {}
12
13     void input() { scanf("%lf %lf", &x, &y); };
14     friend Point operator+(const Point &a, const Point &b) {
15         return Point(a.x + b.x, a.y + b.y);
16     }

```

```

17     friend Point operator-(const Point &a, const Point &b) {
18         return Point(a.x - b.x, a.y - b.y);
19     }
20
21     bool operator !=(const Point& a) const {
22         return (x != a.x || y != a.y);
23     }
24
25     bool operator <(const Point &a) const{
26         if(x == a.x)
27             return y < a.y;
28         return x < a.x;
29     }
30
31     double norm() {
32         return sqrt(sqr(x) + sqr(y));
33     }
34 };
35 double det(const Point &a, const Point &b) {
36     return a.x * b.y - a.y * b.x;
37 }
38 double dot(const Point &a, const Point &b) {
39     return a.x * b.x + a.y * b.y;
40 }
41 double dist(const Point &a, const Point &b) {
42     return (a-b).norm();
43 }
44
45 struct Line {
46     Point a, b;
47     Line() {}
48     Line(Point x, Point y): a(x), b(y) {};
49 };
50
51 double dis_point_segment(const Point p, const Point s, const Point t) {
52     if(sgn(dot(p-s, t-s)) < 0)
53         return (p-s).norm();
54     if(sgn(dot(p-t, s-t)) < 0)
55         return (p-t).norm();
56     return abs(det(s-p, t-p) / dist(s, t));
57 }
58 }

```

5. Graphs

5.1. All Eulerian Path Or Tour

```

1  struct edge {
2      int v, id;
3      edge() {}
4      edge(int v, int id) : v(v), id(id) {}
5  };
6
7  // The undirected + path and directed + tour wasn't tested in a problem.
8  // TEST AGAIN BEFORE SUBMITTING IT!
9  namespace graph {
10     // Namespace which auxiliary functions are defined.
11     namespace detail {
12         pair<bool, pair<int, int>> check_both_directed(const
13             vector<vector<edge>> &adj, const vector<int> &in_degree) {
14             // source and destination
15             int src = -1, dest = -1;
16             // adj[i].size() represents the out degree of an vertex
17             for(int i = 0; i < adj.size(); i++) {

```

```

17     if((int)adj[i].size() - in_degree[i] == 1) {
18         if(src != -1)
19             return make_pair(false, pair<int, int>());
20         src = i;
21     } else if((int)adj[i].size() - in_degree[i] == -1) {
22         if(dest != -1)
23             return make_pair(false, pair<int, int>());
24         dest = i;
25     } else if(abs((int)adj[i].size() - in_degree[i]) > 1)
26         return make_pair(false, pair<int, int>());
27     }
28
29     if(src == -1 && dest == -1)
30         return make_pair(true, pair<int, int>(src, dest));
31     else if(src != -1 && dest != -1)
32         return make_pair(true, pair<int, int>(src, dest));
33
34     return make_pair(false, pair<int, int>());
35 }
36
37 /// Builds the path/tour for directed graphs.
38 void build(const int u, vector<int> &tour, vector<vector<edge>> &adj,
39 vector<bool> &used) {
40     while(!adj[u].empty()) {
41         const edge e = adj[u].back();
42         if(!used[e.id]) {
43             used[e.id] = true;
44             adj[u].pop_back();
45             build(e.v, tour, adj, used);
46         } else
47             adj[u].pop_back();
48     }
49     tour.push_back(u);
50 }
51
52 /// Auxiliary function to build the eulerian tour/path.
53 vector<int> set_build(vector<vector<edge>> &adj, const int E, const int
54 first) {
55     vector<int> path;
56     vector<bool> used(E + 3);
57
58     build(first, path, adj, used);
59
60     for(int i = 0; i < adj.size(); i++)
61         // if there are some remaining edges, it's not possible to build the
62         // tour.
63         if(adj[i].size())
64             return vector<int>();
65         reverse(path.begin(), path.end());
66         return path;
67     }
68 }
69
70 /// All vertices v should have in_degree[v] == out_degree[v]. It must not
71 /// contain a specific
72 /// start and end vertices.
73 ///
74 /// Time complexity: O(V * (log V) + E)
75 bool has_euler_tour_directed(const vector<vector<edge>> &adj, const
76 vector<int> &in_degree) {
77     const pair<bool, pair<int, int>> aux = detail::check_both_directed(adj,
78 in_degree);
79     const bool valid = aux.first;

```

```

76     const int src = aux.second.first;
77     const int dest = aux.second.second;
78     return (valid && src == -1 && dest == -1);
79 }
80
81 /// A directed graph has an eulerian path/tour if has:
82 /// - One vertex v such that out_degree[v] - in_degree[v] == 1
83 /// - One vertex v such that in_degree[v] - out_degree[v] == 1
84 /// - The remaining vertices v such that in_degree[v] == out_degree[v]
85 /// or
86 /// - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR
87 ///
88 /// Returns a boolean value that indicates whether there's a path or not.
89 /// If there's a valid path it also returns two numbers: the source and
90 /// the destination.
91 /// If the source and destination can be an arbitrary vertex it will
92 /// return the pair (-1, -1)
93 /// for the source and destination (it means the contains an eulerian
94 /// tour).
95 ///
96 /// Time complexity: O(V + E)
97 pair<bool, pair<int, int>> has_euler_path_directed(const
98 vector<vector<edge>> &adj, const vector<int> &in_degree) {
99     return detail::check_both_directed(adj, in_degree);
100 }
101
102 /// Returns the euler path. If the graph doesn't have an euler path it
103 /// returns an empty vector.
104 ///
105 /// Time Complexity: O(V + E) for directed, O(V * log(V) + E) for
106 /// undirected.
107 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
108 vector<int> get_euler_path_directed(const int E, vector<vector<edge>>
109 &adj, const vector<int> &in_degree) {
110     const pair<bool, pair<int, int>> aux = has_euler_path_directed(adj,
111 in_degree);
112     const bool valid = aux.first;
113     const int src = aux.second.first;
114     const int dest = aux.second.second;
115
116     if(!valid)
117         return vector<int>();
118
119     int first;
120     if(src != -1)
121         first = src;
122     else {
123         first = 0;
124         while(adj[first].empty())
125             first++;
126     }
127
128     return detail::set_build(adj, E, first);
129 }
130
131 /// Returns the euler tour. If the graph doesn't have an euler tour it
132 /// returns an empty vector.
133 ///
134 /// Time Complexity: O(V + E)
135 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
136 vector<int> get_euler_tour_directed(const int E, vector<vector<edge>>
137 &adj, const vector<int> &in_degree) {
138     const bool valid = has_euler_tour_directed(adj, in_degree);
139
140     if(!valid)

```

```

131     return vector<int>();
132
133     int first = 0;
134     while(adj[first].empty())
135         first++;
136
137     return detail::set_build(adj, E, first);
138 }
139
140 // The graph has a tour that passes to every edge exactly once and gets
141 // back to the first edge on the tour.
142 //
143 // A graph with an euler path has zero odd degree vertex.
144 //
145 // Time Complexity: O(V)
146 bool has_euler_tour_undirected(const vector<int> &degree) {
147     for(int i = 0; i < degree.size(); i++)
148         if(degree[i] & 1)
149             return false;
150     return true;
151 }
152
153 // The graph has a path that passes to every edge exactly once.
154 // It doesn't necessarily gets back to the beginning.
155 //
156 // A graph with an euler path has two or zero (tour) odd degree vertices.
157 //
158 // Returns a pair with the startpoint/endpoint of the path.
159 //
160 // Time Complexity: O(V)
161 pair<bool, pair<int, int>> has_euler_path_undirected(const vector<int>
&degree) {
162     vector<int> odd_degree;
163     for(int i = 0; i < degree.size(); i++)
164         if(degree[i] & 1)
165             odd_degree.pb(i);
166
167     if(odd_degree.size() == 0)
168         return make_pair(true, make_pair(-1, -1));
169     else if (odd_degree.size() == 2)
170         return make_pair(true, make_pair(odd_degree.front(),
odd_degree.back()));
171     else
172         return make_pair(false, pair<int, int>());
173 }
174
175 vector<int> get_euler_tour_undirected(const int E, const vector<int>
&degree, vector<vector<edge>> &adj) {
176     if(!has_euler_tour_undirected(degree))
177         return vector<int>();
178
179     int first = 0;
180     while(adj[first].empty())
181         first++;
182
183     return detail::set_build(adj, E, first);
184 }
185
186 /// Returns the euler tour. If the graph doesn't have an euler tour it
187 /// returns an empty vector.
188 ///
189 /// Time Complexity: O(V + E)
190 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
191 vector<int> get_euler_path_undirected(const int E, const vector<int>
&degree, vector<vector<edge>> &adj) {

```

```

191     auto aux = has_euler_path_undirected(degree);
192     const bool valid = aux.first;
193     const int x = aux.second.first;
194     const int y = aux.second.second;
195
196     if(!valid)
197         return vector<int>();
198
199     int first;
200     if(x != -1) {
201         first = x;
202         adj[x].emplace_back(y, E + 1);
203         adj[y].emplace_back(x, E + 1);
204     } else {
205         first = 0;
206         while(adj[first].empty())
207             first++;
208     }
209
210     vector<int> ans = detail::set_build(adj, E, first);
211     reverse(ans.begin(), ans.end());
212     if(x != -1)
213         ans.pop_back();
214     return ans;
215 }
216 };

```

5.2. Articulation Points

```

1 namespace graph {
2 unordered_set<int> ap;
3 vector<int> low, disc;
4 int cur_time = 1;
5
6 void dfs_ap(const int u, const int p, const vector<vector<int>> &adj) {
7     low[u] = disc[u] = cur_time++;
8     int children = 0;
9
10    for (const int v : adj[u]) {
11        // DO NOT ADD PARALLEL EDGES
12        if (disc[v] == 0) {
13            ++children;
14            dfs_ap(v, u, adj);
15
16            low[u] = min(low[v], low[u]);
17            if (p == -1 && children > 1)
18                ap.emplace(u);
19            if (p != -1 && low[v] >= disc[u])
20                ap.emplace(u);
21        } else if (v != p)
22            low[u] = min(low[u], disc[v]);
23    }
24 }
25
26 void init_ap(const int n) {
27     cur_time = 1;
28     ap = unordered_set<int>();
29     low = vector<int>(n, 0);
30     disc = vector<int>(n, 0);
31 }
32
33 /// THE GRAPH MUST BE UNDIRECTED!
34 ///
35 /// Returns the vertices in which their removal disconnects the graph.

```

```

36 ///
37 /// Time Complexity: O(V + E)
38 vector<int> articulation_points(const int indexed_from,
39                               const vector<vector<int>> &adj) {
40     init_ap(adj.size());
41     vector<int> ans;
42     for (int u = indexed_from; u < adj.size(); ++u) {
43         if (disc[u] == 0)
44             dfs_ap(u, -1, adj);
45         if (ap.count(u))
46             ans.emplace_back(u);
47     }
48     return ans;
49 }
50 }; // namespace graph

```

5.3. Bellman Ford

```

1 struct edge {
2     int src, dest, weight;
3     edge() {}
4     edge(int src, int dest, int weight) : src(src), dest(dest), weight(weight)
5     {}
6     bool operator<(const edge &a) const {
7         return weight < a.weight;
8     }
9 };
10
11 /// Works to find the shortest path with negative edges.
12 /// Also detects cycles.
13 ///
14 /// Time Complexity: O(n * e)
15 /// Space Complexity: O(n)
16 bool bellman_ford(vector<edge> &edges, int src, int n) {
17     // n = qtd of vertices, E = qtd de arestas
18
19     // To calculate the shortest path uncomment the line below
20     // vector<int> dist(n, INF);
21
22     // To check cycles uncomment the line below
23     // vector<int> dist(n, 0);
24
25     vector<int> pai(n, -1);
26     int E = edges.size();
27
28     dist[src] = 0;
29     // Relax all edges n - 1 times.
30     // A simple shortest path from src to any other vertex can have at-most n
31     // - 1 edges.
32     for (int i = 1; i <= n - 1; i++) {
33         for (int j = 0; j < E; j++) {
34             int u = edges[j].src;
35             int v = edges[j].dest;
36             int weight = edges[j].weight;
37             if (dist[u] != INF && dist[u] + weight < dist[v]) {
38                 dist[v] = dist[u] + weight;
39                 pai[v] = u;
40             }
41         }
42     }
43     // Check for NEGATIVE-WEIGHT CYCLES.

```

```

44 // The above step guarantees shortest distances if graph doesn't contain
45 // negative weight cycle.
46 // If we get a shorter path, then there is a cycle.
47 bool is_cycle = false;
48 int vert_in_cycle;
49 for (int i = 0; i < E; i++) {
50     int u = edges[i].src;
51     int v = edges[i].dest;
52     int weight = edges[i].weight;
53     if (dist[u] != INF && dist[u] + weight < dist[v]) {
54         is_cycle = true;
55         pai[v] = u;
56         vert_in_cycle = v;
57     }
58 }
59 if(is_cycle) {
60     for(int i = 0; i < n; i++)
61         vert_in_cycle = pai[vert_in_cycle];
62
63     vector<int> cycle;
64     for(int v = vert_in_cycle; (v != vert_in_cycle || cycle.size() <= 1) ; v
65         = pai[v])
66         cycle.pb(v);
67     reverse(cycle.begin(), cycle.end());
68
69     for(int x: cycle) {
70         cout << x + 1 << ' ';
71     }
72     cout << cycle.front() + 1 << endl;
73     return true;
74 } else
75     return false;
76 }

```

5.4. Bipartite Check

```

1 /// Time Complexity: O(V + E)
2 bool is_bipartite(const int src, const vector<vector<int>> &adj) {
3     vector<int> color(adj.size(), -1);
4     queue<int> q;
5
6     color[src] = 1;
7     q.emplace(src);
8     while (!q.empty()) {
9         const int u = q.front();
10        q.pop();
11
12        for (const int v : adj[u]) {
13            if (color[v] == color[u])
14                return false;
15            else if (color[v] == -1) {
16                color[v] = !color[u];
17                q.emplace(v);
18            }
19        }
20    }
21    return true;
22 }

```

5.5. Block Cut Tree


```

1 // based on kokosha's implementation.
2 /// INDEXED FROM ZERO!!!!
3 class BCT {
4     vector<vector<pair<int, int>>> adj;
5     vector<pair<int, int>> edges;
6     /// Stores the edges in the i-th component.
7     vector<vector<int>> comps;
8     /// Stores the vertices in the i-th component.
9     vector<vector<int>> vert_in_comp;
10    int cur_time = 0;
11    vector<int> disc, conv;
12    vector<vector<int>> adj_bct;
13    const int n;
14
15    /// Finds the biconnected components.
16    int dfs(const int x, const int p, stack<int> &st) {
17        int low = disc[x] = ++cur_time;
18        for (const pair<int, int> &e : adj[x]) {
19            const int v = e.first, idx = e.second;
20            if (idx != p) {
21                if (!disc[v]) { // if haven't passed
22                    st.emplace(idx); // disc[x] < low -> bridge
23                    const int low_at = dfs(v, idx, st);
24                    low = min(low, low_at);
25                    if (disc[x] <= low_at) {
26                        comps.emplace_back();
27                        vector<int> &tmp = comps.back();
28                        for (int y = -1; y != idx; st.pop())
29                            tmp.emplace_back(y = st.top());
30                    }
31                } else if (disc[v] < disc[x]) // back_edge
32                    low = min(low, disc[v]), st.emplace(idx);
33            }
34        }
35        return low;
36    }
37
38    /// Splits the graph into biconnected components.
39    void split() {
40        adj_bct.resize(n + edges.size() + 1);
41        stack<int> st;
42        for (int i = 0; i < n; ++i)
43            if (!disc[i])
44                dfs(i, -1, st);
45
46        vector<bool> in(n);
47        for (const vector<int> &comp : comps) {
48            vert_in_comp.emplace_back();
49            for (const int e : comp) {
50                const int u = edges[e].first, v = edges[e].second;
51                if (!in[u])
52                    in[u] = 1, vert_in_comp.back().emplace_back(u);
53                if (!in[v])
54                    in[v] = 1, vert_in_comp.back().emplace_back(v);
55            }
56            for (const int e : comp)
57                in[edges[e].first] = in[edges[e].second] = 0;
58        }
59    }
60
61    /// Algorithm: It compresses the biconnected components into one vertex.
62    /// Then
63    /// it creates a bipartite graph with the original vertices on the left and
64    /// the bcc's on the right. After that, it connects with an edge the i-th

```

```

64    /// vertex on the left to the j-th on the right if the vertex i is present
65    in
66    /// the j-th bcc. Note that articulation points will be present in more
67    /// than
68    /// one component.
69    void build() {
70        // next new node to be used in bct
71        int nxt = n;
72        for (const vector<int> &vic : vert_in_comp) {
73            for (const int u : vic) {
74                adj_bct[u].emplace_back(nxt);
75                adj_bct[nxt].emplace_back(u);
76                conv[u] = nxt;
77            }
78            nxt++;
79        }
80
81        // if it's not an articulation point we can remove it from the bct.
82        for (int i = 0; i < n; ++i)
83            if (adj_bct[i].size() == 1)
84                adj_bct[i].clear();
85
86    }
87
88    void init() {
89        disc.resize(n);
90        conv.resize(n);
91        adj.resize(n);
92    }
93
94    public:
95    /// Pass the number of vertices to the constructor.
96    BCT(const int n) : n(n) { init(); }
97
98    /// Adds an bidirectional edge.
99    void add_edge(const int u, const int v) {
100        assert(0 <= min(u, v)), assert(max(u, v) < n), assert(u != v);
101        adj[u].emplace_back(v, edges.size());
102        adj[v].emplace_back(u, edges.size());
103        edges.emplace_back(u, v);
104    }
105
106    /// Returns the bct tree. It builds the tree if it's not computed.
107    ///
108    /// Time Complexity: O(n + m)
109    vector<vector<int>> tree() {
110        if (adj_bct.empty()) // if it's not calculated.
111            split(), build();
112        return adj_bct;
113    }
114
115    /// Returns whether the vertex u is an articulation point or not.
116    bool is_art_point(const int u) {
117        assert(0 <= u), assert(u < n);
118        assert(!adj_bct.empty()); // the tree method should've called before.
119        return !adj_bct[u].empty();
120    }
121
122    /// Returns the corresponding vertex of the u-th vertex in the bct.
123    int convert(const int u) {
124        assert(0 <= u), assert(u < n);
125        assert(!adj_bct.empty()); // the tree method should've called before.
126        return adj_bct[u].empty() ? conv[u] : u;
127    }
128
129    };

```

5.6. Bridges

```

1 namespace graph {
2 int cur_time = 1;
3 vector<pair<int, int>> bg;
4 vector<int> disc;
5 vector<int> low;
6 vector<int> cycle;
7
8 void dfs_bg(const int u, int p, const vector<vector<int>> &adj) {
9     low[u] = disc[u] = cur_time++;
10    for (const int v : adj[u]) {
11        if (v == p) {
12            // checks parallel edges
13            // IT'S BETTER TO REMOVE THEM!
14            p = -1;
15            continue;
16        } else if (disc[v] == 0) {
17            dfs_bg(v, u, adj);
18            low[u] = min(low[u], low[v]);
19            if (low[v] > disc[u])
20                bg.emplace_back(u, v);
21        } else
22            low[u] = min(low[u], disc[v]);
23        // checks if the vertex u belongs to a cycle
24        cycle[u] |= (disc[u] >= low[v]);
25    }
26 }
27
28 void init_bg(const int n) {
29     cur_time = 1;
30     bg = vector<pair<int, int>>();
31     disc = vector<int>(n, 0);
32     low = vector<int>(n, 0);
33     cycle = vector<int>(n, 0);
34 }
35
36 /// THE GRAPH MUST BE UNDIRECTED!
37 ///
38 /// Returns the edges in which their removal disconnects the graph.
39 ///
40 /// Time Complexity: O(V + E)
41 vector<pair<int, int>> bridges(const int indexed_from,
42                             const vector<vector<int>> &adj) {
43     init_bg(adj.size());
44     for (int u = indexed_from; u < adj.size(); ++u)
45         if (disc[u] == 0)
46             dfs_bg(u, -1, adj);
47
48     return bg;
49 }
50 } // namespace graph

```

5.7. Centroid

```

1 /// Returns the centroids of the tree which can contains at most 2.
2 ///
3 /// Time complexity: O(n)
4 vector<int> centroid(const int n, const int indexed_from,
5                    const vector<vector<int>> &adj) {
6     vector<int> centers, sz(n + indexed_from);
7     function<void(int, int)> dfs = [&](const int u, const int p) {
8         sz[u] = 1;
9         bool is_centroid = true;

```

```

10    for (const int v : adj[u]) {
11        if (v == p)
12            continue;
13        dfs(v, u);
14        sz[u] += sz[v];
15        if (sz[v] > n / 2)
16            is_centroid = false;
17    }
18    if (n - sz[u] > n / 2)
19        is_centroid = false;
20    if (is_centroid)
21        centers.emplace_back(u);
22 };
23 dfs(indexed_from, -1);
24 return centers;
25 }

```

5.8. Centroid Decomposition

```

1 class Centroid {
2 private:
3     int it = 1, _vertex;
4     vector<int> vis, used, sub, _parent;
5     vector<vector<int>> _tree;
6
7     int dfs(const int u, int &cnt, const vector<vector<int>> &adj) {
8         vis[u] = it;
9         ++cnt;
10        sub[u] = 1;
11        for (const int v : adj[u])
12            if (vis[v] != it && !used[v])
13                sub[u] += dfs(v, cnt, adj);
14        return sub[u];
15    }
16
17     int find_centroid(const int u, const int cnt,
18                     const vector<vector<int>> &adj) {
19         vis[u] = it;
20
21         bool valid = true;
22         int max_sub = -1;
23         for (const int v : adj[u]) {
24             if (vis[v] == it || used[v])
25                 continue;
26             if (sub[v] > cnt / 2)
27                 valid = false;
28             if (max_sub == -1 || sub[v] > sub[max_sub])
29                 max_sub = v;
30         }
31
32         if (valid && cnt - sub[u] <= cnt / 2)
33             return u;
34         return find_centroid(max_sub, cnt, adj);
35     }
36
37     int find_centroid(const int u, const vector<vector<int>> &adj) {
38         // counts the number of vertices
39         int cnt = 0;
40
41         // set up sizes and nodes in current subtree
42         dfs(u, cnt, adj);
43         ++it;
44
45         const int ctd = find_centroid(u, cnt, adj);

```

```

46     ++it;
47     used[ctd] = true;
48     return ctd;
49 }
50
51 int build_tree(const int u, const vector<vector<int>> &adj) {
52     const int ctd = find_centroid(u, adj);
53
54     for (const int v : adj[ctd]) {
55         if (used[v])
56             continue;
57         const int ctd_v = build_tree(v, adj);
58         _tree[ctd].emplace_back(ctd_v);
59         _tree[ctd_v].emplace_back(ctd);
60         _parent[ctd_v] = ctd;
61     }
62
63     return ctd;
64 }
65
66 void allocate(const int n) {
67     vis.resize(n);
68     _parent.resize(n, -1);
69     sub.resize(n);
70     used.resize(n);
71     _tree.resize(n);
72 }
73
74 public:
75     /// Constructor that creates the centroid tree.
76     ///
77     /// Time Complexity:  $O(n \cdot \log(n))$ 
78     Centroid(const int root_idx, const vector<vector<int>> &adj) {
79         allocate(adj.size());
80         _vertex = build_tree(root_idx, adj);
81     }
82
83     /// Returns the centroid of the whole tree.
84     int vertex() { return _vertex; }
85
86     int parent(const int u) { return _parent[u]; }
87
88     vector<vector<int>> tree() { return _tree; }
89 };

```

5.9. Compress ScCs In Dag

```

1 DSU dsu(MAXN);
2 /// Compress SCC's in a directed graph.
3 ///
4 /// Time Complexity:  $O(V)$ 
5 vector<vector<int>> compress(const int indexed_from,
6                             const vector<vector<int>> &adj) {
7     const int n = adj.size();
8     SCC scc(n, indexed_from, adj);
9     vector<unordered_set<int>> g(n);
10
11     for (int i = 0; i < scc.number_of_comp; ++i)
12         for (int v : scc.scc[i])
13             dsu.Union(v, scc.scc[i].front());
14
15     for (int u = indexed_from; u < n; ++u)
16         for (int v : adj[u])
17             if (dsu.Find(u) != dsu.Find(v))

```

```

18         g[dsu.Find(u)].emplace(dsu.Find(v));
19
20     vector<vector<int>> ret(n);
21     for (int u = indexed_from; u < n; ++u)
22         ret[u] = vector<int>(g[u].begin(), g[u].end());
23     return ret;
24 }

```

5.10. Count (3-4) Cycles

```

1 /// INDEXED FROM 0!!!!
2 /// Counts the number of cycles of length 3 and 4 in the graph.
3 /// The vector cycles contains some cycles of length for and I think (not
4 /// sure)
5 /// all cycles of length 3.
6 ///
7 /// Time complexity:  $O(n \cdot \sqrt{n})$ 
8 int count_cycles(vector<vector<int>> &adj) {
9     const int n = adj.size();
10    vector<int> rep(n);
11
12    auto comp = [&](int u, int v) {
13        return adj[u].size() == adj[v].size() ? u < v
14            : adj[u].size() > adj[v].size();
15    };
16
17    // Contains edges (u, v) in the original graph such that comp is true.
18    vector<vector<int>> g(n);
19    for (int u = 0; u < n; ++u)
20        for (const int v : adj[u])
21            if (comp(u, v))
22                g[u].emplace_back(v);
23
24    vector<int> cnt(n), vis(n);
25    // Contains some cycles of length 4 and 3 from the graph
26    vector<vector<int>> cycles;
27
28    int ans = 0;
29    for (int u = 0; u < n; ++u) {
30        // Counting Squares:
31        for (int to1 : g[u]) {
32            cnt[to1] = 0;
33            rep[to1] = -1;
34            for (int to2 : adj[to1]) {
35                rep[to2] = -1;
36                cnt[to2] = 0;
37            }
38            for (int to1 : g[u])
39                for (int to2 : adj[to1]) {
40                    if (comp(u, to2)) {
41                        ans += cnt[to2];
42                        ++cnt[to2];
43                    }
44                    if (rep[to2] != -1)
45                        cycles.push_back({u, to1, to2, rep[to2]});
46                    rep[to2] = to1;
47                }
48            }
49
50    // Finding Triangles:
51    for (int to : adj[u])
52        vis[to] = 1;
53    for (int to1 : g[u])

```

```

54     for (int to2 : g[to1])
55         if (vis[to2])
56             cycles.push_back({u, to1, to2});
57     for (int to : adj[u])
58         vis[to] = 0;
59 }
60
61 return ans;
62 }

```

5.11. Cycle Detection

```

1  /// Returns an arbitrary cycle in the graph.
2  ///
3  /// Time Complexity: O(n)
4  vector<int> cycle(const int root_idx, const int n,
5                  const vector<vector<int>> &adj) {
6      vector<bool> vis(n + 1);
7      vector<int> ans;
8      function<int(int, int)> dfs = [&](const int u, const int p) {
9          vis[u] = true;
10         int val = -1;
11         for (const int v : adj[u]) {
12             if (v == p)
13                 continue;
14             if (!vis[v]) {
15                 const int x = dfs(v, u);
16                 if (x != -1) {
17                     val = x;
18                     break;
19                 }
20             } else {
21                 val = v;
22                 break;
23             }
24         }
25         if (val != -1)
26             ans.emplace_back(u);
27         return (val == u ? -1 : val);
28     };
29     dfs(root_idx, -1);
30     return ans;
31 }

```

5.12. De Bruijn Sequence

```

1  // We can solve this problem by constructing a directed graph with
2  //  $k^{(n-1)}$  nodes with each node having k outgoing edges_order. Each node
3  // corresponds to a string of size n-1. Every edge corresponds to one of the
4  // characters in A and adds that character to the starting string. For
5  // example,
6  // if n=3 and k=2, then we construct the following graph:
7
8  //      - 1 -> (01) - 1 ->
9  //      /      ^ |      \
10 // 0 -> (00)  1 0      (11) <- 1
11 //      \      | v      /
12 //      <- 0 - (10) <- 0 -
13
14 // The node '01' is connected to node '11' through edge '1', as adding '1' to
15 // '01' (and removing the first character) gives us '11'.

```

```

16 // We can observe that every node in this graph has equal in-degree and
17 // out-degree, which means that a Eulerian circuit exists in this graph.
18
19 namespace graph {
20 namespace detail {
21 // Finding an valid eulerian path
22 void dfs(const string &node, const string &alphabet, set<string> &vis,
23         string &edges_order) {
24     for (char c : alphabet) {
25         string nxt = node + c;
26         if (vis.count(nxt))
27             continue;
28
29         vis.insert(nxt);
30         nxt.erase(nxt.begin());
31         dfs(nxt, alphabet, vis, edges_order);
32         edges_order += c;
33     }
34 }
35 }; // namespace detail
36
37 // Returns a string in which every string of the alphabet of size n appears
38 // in
39 // the resulting string exactly once.
40 // Time Complexity: O(alphabet.size() ^ n * log2(alphabet.size() ^ n))
41 string de_bruijn(const int n, const string &alphabet) {
42     set<string> vis;
43     string edges_order;
44
45     string starting_node = string(n - 1, alphabet.front());
46     detail::dfs(starting_node, alphabet, vis, edges_order);
47
48     return edges_order + starting_node;
49 }
50 }; // namespace graph

```

5.13. Diameter In Tree

1 From any vertex, X find the furthestmost vertex A from X. After that, **return** the distance from vertex A from the furthestmost vertex B from A.

5.14. Dijkstra + Dij Graph

```

1  // Works also with 1-indexed graphs.
2  class Dijkstra {
3  private:
4      static constexpr int INF = 2e18;
5      bool CREATE_GRAPH = false;
6      int src;
7      int n;
8      vector<int> _dist;
9      vector<vector<int>> parent;
10
11  private:
12      /// Time Complexity: O(E log V)
13      void _compute(const int src, const vector<vector<pair<int, int>>> &adj) {
14          _dist.resize(this->n, INF);
15          vector<bool> vis(this->n, false);
16
17          if (CREATE_GRAPH) {
18              parent.resize(this->n);
19          }

```

```

20     for (int i = 0; i < this->n; i++)
21         parent[i].emplace_back(i);
22     }
23
24     priority_queue<pair<int, int>, vector<pair<int, int>>,
25         greater<pair<int, int>>>
26         pq;
27     pq.emplace(0, src);
28     _dist[src] = 0;
29
30     while (!pq.empty()) {
31         int u = pq.top().second;
32         pq.pop();
33         if (vis[u])
34             continue;
35         vis[u] = true;
36
37         for (const pair<int, int> &x : adj[u]) {
38             int v = x.first, w = x.second;
39
40             if (_dist[u] + w < _dist[v]) {
41                 _dist[v] = _dist[u] + w;
42                 pq.emplace(_dist[v], v);
43                 if (CREATE_GRAPH) {
44                     parent[v].clear();
45                     parent[v].emplace_back(u);
46                 }
47             } else if (CREATE_GRAPH && _dist[u] + w == _dist[v]) {
48                 parent[v].emplace_back(u);
49             }
50         }
51     }
52 }
53
54 vector<vector<int>> gen_dij_graph(const int dest) {
55     vector<vector<int>> dijkstra_graph(this->n);
56     vector<bool> vis(this->n, false);
57     queue<int> q;
58
59     q.emplace(dest);
60     while (!q.empty()) {
61         int v = q.front();
62         q.pop();
63
64         for (const int u : parent[v]) {
65             if (u == v)
66                 continue;
67             dijkstra_graph[u].emplace_back(v);
68             if (!vis[u]) {
69                 q.emplace(u);
70                 vis[u] = true;
71             }
72         }
73     }
74     return dijkstra_graph;
75 }
76
77 vector<int> gen_min_path(const int dest) {
78     vector<int> path, prev(this->n, -1), d(this->n, INF);
79     queue<int> q;
80
81     q.emplace(dest);
82     d[dest] = 0;
83
84     while (!q.empty()) {

```

```

85         int v = q.front();
86         q.pop();
87
88         for (const int u : parent[v]) {
89             if (u == v)
90                 continue;
91             if (d[v] + 1 < d[u]) {
92                 d[u] = d[v] + 1;
93                 prev[u] = v;
94                 q.emplace(u);
95             }
96         }
97     }
98
99     int cur = this->src;
100     while (cur != -1) {
101         path.emplace_back(cur);
102         cur = prev[cur];
103     }
104
105     return path;
106 }
107
108 public:
109     /// Allows creation of dijkstra graph and getting the minimum path.
110     Dijkstra(const int src, const bool create_graph,
111         const vector<vector<pair<int, int>>> &adj)
112         : n(adj.size()), src(src), CREATE_GRAPH(create_graph) {
113         this->_compute(src, adj);
114     }
115
116     /// Constructor that computes only the Dijkstra minimum path from src.
117     ///
118     /// Time Complexity: O(E log V)
119     Dijkstra(const int src, const vector<vector<pair<int, int>>> &adj)
120         : n(adj.size()), src(src) {
121         this->_compute(src, adj);
122     }
123
124     /// Returns the Dijkstra graph of the graph.
125     ///
126     /// Time Complexity: O(V)
127     vector<vector<int>> dij_graph(const int dest) {
128         assert(CREATE_GRAPH);
129         return gen_dij_graph(dest);
130     }
131
132     /// Returns the vertices present in a path from src to dest with
133     /// minimum cost and a minimum length.
134     ///
135     /// Time Complexity: O(V)
136     vector<int> min_path(const int dest) {
137         assert(CREATE_GRAPH);
138         return gen_min_path(dest);
139     }
140
141     /// Returns the distance from src to dest.
142     int dist(const int dest) {
143         assert(0 <= dest), assert(dest < n);
144         return _dist[dest];
145     }
146 };

```

```

1 class Dinic {
2     struct Edge {
3         const int v;
4         // capacity (maximum flow) of the edge
5         // if it is a reverse edge then its capacity should be equal to 0
6         const int cap;
7         // current flow of the edge
8         int flow = 0;
9         Edge(const int v, const int cap) : v(v), cap(cap) {}
10    };
11
12 private:
13     static constexpr int INF = (sizeof(int) == 4 ? 1e9 : 2e18) + 1e5;
14     bool COMPUTED = false;
15     int _max_flow;
16     vector<Edge> edges;
17     // holds the indexes of each edge present in each vertex.
18     vector<vector<int>>> adj;
19     const int n;
20     // src will be always 0 and sink n+1.
21     const int src, sink;
22     vector<int> level, ptr;
23
24 private:
25     vector<vector<int>>> _flow_table() {
26         vector<vector<int>>> table(n, vector<int>(n, 0));
27         for (int u = 0; u <= sink; ++u)
28             for (const int idx : adj[u])
29                 // checks if it's not a reverse edge
30                 if (!(idx & 1))
31                     table[u][edges[idx].v] += edges[idx].flow;
32         return table;
33     }
34
35     /// Algorithm: Greedily all vertices from the matching will be added and,
36     /// after that, edges in which one of the vertices is not covered will
37     /// also be
38     /// added to the answer.
39     vector<pair<int, int>> _min_edge_cover() {
40         vector<bool> covered(n, false);
41         vector<pair<int, int>> ans;
42         for (int u = 1; u < sink; ++u) {
43             for (const int idx : adj[u]) {
44                 const Edge &e = edges[idx];
45                 // ignore if it is a reverse edge or an edge linked to the sink
46                 if (idx & 1 || e.v == sink)
47                     continue;
48                 if (e.flow == e.cap) {
49                     ans.emplace_back(u, e.v);
50                     covered[u] = covered[e.v] = true;
51                     break;
52                 }
53             }
54         }
55         for (int u = 1; u < sink; ++u) {
56             for (const int idx : adj[u]) {
57                 const Edge &e = edges[idx];
58                 if (idx & 1 || e.v == sink)
59                     continue;
60                 if (e.flow < e.cap && (!covered[u] || !covered[e.v])) {
61                     ans.emplace_back(u, e.v);
62                     covered[u] = covered[e.v] = true;
63                 }
64             }
65         }
66         return ans;
67     }
68
69     /// Algorithm: Takes the complement of the vertex cover.
70     vector<int> _max_ind_set(const int max_left) {
71         const vector<int> mvc = _min_vertex_cover(max_left);
72         vector<bool> contains(n);
73         for (const int v : mvc)
74             contains[v] = true;
75         vector<int> ans;
76         for (int i = 1; i < sink; ++i)
77             if (!contains[i])
78                 ans.emplace_back(i);
79         return ans;
80     }
81
82 void dfs_vc(const int u, vector<bool> &vis, const bool left,
83             const vector<vector<int>>> &paths) {
84     vis[u] = true;
85     for (const int idx : adj[u]) {
86         const Edge &e = edges[idx];
87         if (vis[e.v])
88             continue;
89         // saturated edges goes from right to left
90         if (left && paths[u][e.v] == 0)
91             dfs_vc(e.v, vis, left ^ 1, paths);
92         // non-saturated edges goes from left to right
93         else if (!left && paths[e.v][u] == 1)
94             dfs_vc(e.v, vis, left ^ 1, paths);
95     }
96 }
97
98 /// Algorithm: The edges that belong to the Matching M will go from right
99 to
100 /// left, all other edges will go from left to right. A DFS will be run
101 /// starting at all left vertices that are not incident to edges in M. Some
102 /// vertices of the graph will become visited during this DFS and some
103 /// not-visited. To get minimum vertex cover all visited right
104 /// vertices of M will be taken, and all not-visited left vertices of M.
105 /// Source: codeforces.com/blog/entry/17534?comment=223759
106 vector<int> _min_vertex_cover(const int max_left) {
107     vector<bool> vis(n, false), saturated(n, false);
108     const auto paths = flow_table();
109
110     for (int i = 1; i <= max_left; ++i) {
111         for (int j = max_left + 1; j < sink; ++j)
112             if (paths[i][j] > 0) {
113                 saturated[i] = saturated[j] = true;
114                 break;
115             }
116         if (!saturated[i] && !vis[i])
117             dfs_vc(i, vis, 1, paths);
118     }
119
120     vector<int> ans;
121     for (int i = 1; i <= max_left; ++i)
122         if (saturated[i] && !vis[i])
123             ans.emplace_back(i);
124
125     for (int i = max_left + 1; i < sink; ++i)
126         if (saturated[i] && vis[i])
127             ans.emplace_back(i);

```

```

128     return ans;
129 }
130
131 void dfs_build_path(const int u, vector<int> &path,
132                   vector<vector<int>> &table, vector<vector<int>> &ans,
133                   const vector<vector<int>> &adj) {
134     path.emplace_back(u);
135
136     if (u == sink) {
137         ans.emplace_back(path);
138         return;
139     }
140
141     for (const int v : adj[u]) {
142         if (table[u][v]) {
143             --table[u][v];
144             dfs_build_path(v, path, table, ans, adj);
145             return;
146         }
147     }
148 }
149
150 /// Algorithm: Run DFS's from the source and gets the paths when possible.
151 vector<vector<int>> _compute_all_paths(const vector<vector<int>> &adj) {
152     vector<vector<int>> table = flow_table();
153     vector<vector<int>> ans;
154     ans.reserve(_max_flow);
155
156     for (int i = 0; i < _max_flow; i++) {
157         vector<int> path;
158         path.reserve(n);
159         dfs_build_path(src, path, table, ans, adj);
160     }
161
162     return ans;
163 }
164
165 /// Algorithm: Find the set of vertices that are reachable from the source
166 in
167 /// the residual graph. All edges which are from a reachable vertex to
168 /// non-reachable vertex are minimum cut edges.
169 /// Source: geeksforgeeks.org/minimum-cut-in-a-directed-graph
170 pair<int, vector<pair<int, int>>> _min_cut() {
171     // checks if there's an edge from i to j.
172     vector<vector<int>> mat_adj(n, vector<int>(n, 0));
173     // checks if the residual capacity is greater than 0
174     vector<vector<bool>> residual(n, vector<bool>(n, 0));
175     for (int u = 0; u <= sink; ++u)
176         for (const int idx : adj[u])
177             // checks if it's not a reverse edge
178             if (!(idx & 1)) {
179                 mat_adj[u][edges[idx].v] = edges[idx].cap;
180                 // checks if its residual capacity is greater than zero.
181                 if (edges[idx].flow < edges[idx].cap)
182                     residual[u][edges[idx].v] = true;
183             }
184
185     vector<bool> vis(n);
186     queue<int> q;
187
188     q.emplace(src);
189     vis[src] = true;
190     while (!q.empty()) {
191         int u = q.front();
192         q.pop();

```

```

192         for (int v = 0; v < n; ++v)
193             if (residual[u][v] && !vis[v]) {
194                 q.emplace(v);
195                 vis[v] = true;
196             }
197     }
198
199     int weight = 0;
200     vector<pair<int, int>> cut;
201     for (int i = 0; i < n; ++i)
202         for (int j = 0; j < n; ++j)
203             if (vis[i] && !vis[j])
204                 // if there's an edge from i to j.
205                 if (mat_adj[i][j] > 0) {
206                     weight += mat_adj[i][j];
207                     cut.emplace_back(i, j);
208                 }
209
210     return make_pair(weight, cut);
211 }
212
213 void _add_edge(const int u, const int v, const int cap) {
214     adj[u].emplace_back(edges.size());
215     edges.emplace_back(v, cap);
216     // adding reverse edge
217     adj[v].emplace_back(edges.size());
218     edges.emplace_back(u, 0);
219 }
220
221 bool bfs_flow() {
222     queue<int> q;
223     memset(level.data(), -1, sizeof(*level.data()) * level.size());
224     q.emplace(src);
225     level[src] = 0;
226     while (!q.empty()) {
227         const int u = q.front();
228         q.pop();
229         for (const int idx : adj[u]) {
230             const Edge &e = edges[idx];
231             if (e.cap == e.flow || level[e.v] != -1)
232                 continue;
233             level[e.v] = level[u] + 1;
234             q.emplace(e.v);
235         }
236     }
237     return (level[sink] != -1);
238 }
239
240 int dfs_flow(const int u, const int cur_flow) {
241     if (u == sink)
242         return cur_flow;
243
244     for (int &idx = ptr[u]; idx < adj[u].size(); ++idx) {
245         Edge &e = edges[adj[u][idx]];
246         if (level[u] + 1 != level[e.v] || e.cap == e.flow)
247             continue;
248         const int flow = dfs_flow(e.v, min(e.cap - e.flow, cur_flow));
249         if (flow == 0)
250             continue;
251         e.flow += flow;
252         edges[adj[u][idx] ^ 1].flow -= flow;
253         return flow;
254     }
255     return 0;
256 }

```

```

257
258 int compute() {
259     int ans = 0;
260     while (bfs_flow()) {
261         memset(ptr.data(), 0, sizeof(*ptr.data()) * ptr.size());
262         while (const int cur = dfs_flow(src, INF))
263             ans += cur;
264     }
265     return ans;
266 }
267
268 void check_computed() {
269     if (!COMPUTED) {
270         COMPUTED = true;
271         this->_max_flow = compute();
272     }
273 }
274
275 public:
276     /// Constructor that makes assignments and allocations.
277     ///
278     /// Time Complexity:  $O(V)$ 
279     Dinic(const int n : n(n + 2), src(0), sink(n + 1) {
280         assert(n >= 0);
281
282         adj.resize(this->n);
283         level.resize(this->n);
284         ptr.resize(this->n);
285     }
286
287     /// Prints all the added edges. Use it to test in [CSA Graph
288     /// Editor] (https://csacademy.com/app/graph\_editor/).
289     void print() {
290         for (int u = 0; u < n; ++u)
291             for (const int idx : adj[u])
292                 if (!(idx & 1))
293                     cerr << u << ' ' << edges[idx].v << ' ' << edges[idx].cap << endl;
294     }
295
296     /// Returns the edges from the minimum edge cover of the graph.
297     /// A minimum edge cover represents a set of edges such that each vertex
298     /// present in the graph is linked to at least one edge from this set.
299     ///
300     /// Time Complexity:  $O(V + E)$ 
301     vector<pair<int, int>> min_edge_cover() {
302         this->check_computed();
303         return this->_min_edge_cover();
304     }
305
306     /// Returns the maximum independent set for the graph.
307     /// An independent set represents a set of vertices such that they're not
308     /// adjacent to each other.
309     /// It is equal to the complement of the minimum vertex cover.
310     ///
311     /// Time Complexity:  $O(V + E)$ 
312     vector<int> max_ind_set(const int max_left) {
313         this->check_computed();
314         return this->_max_ind_set(max_left);
315     }
316
317     /// Returns the minimum vertex cover of a bipartite graph.
318     /// A minimum vertex cover represents a set of vertices such that each
319     /// edge of
320     /// the graph is incident to at least one vertex of the graph.
321     /// Pass the maximum index of a vertex on the left side as an argument.

```

```

321
322     /// Time Complexity:  $O(V + E)$ 
323     vector<int> min_vertex_cover(const int max_left) {
324         this->check_computed();
325         return this->_min_vertex_cover(max_left);
326     }
327
328     /// Computes all paths from src to sink.
329     /// Add all edges from the original graph. Its weights should be equal to
330     /// the
331     /// number of edges between the vertices. Pass the adjacency list with
332     /// repeated vertices if there are multiple edges.
333     ///
334     /// Time Complexity:  $O(\max\_flow * V + E)$ 
335     vector<vector<int>> compute_all_paths(const vector<vector<int>> &adj) {
336         this->check_computed();
337         return this->_compute_all_paths(adj);
338     }
339
340     /// Returns the weight and the edges present in the minimum cut of the
341     /// graph.
342     /// A minimum cut represents a set of edges with minimum weight such that
343     /// after removing these edges, it disconnects the graph. If the graph is
344     /// undirected you can safely add edges in both directions. It doesn't work
345     /// with parallel edges, it's required to merge them.
346     ///
347     /// Time Complexity:  $O(V^2 + E)$ 
348     pair<int, vector<pair<int, int>>> min_cut() {
349         this->check_computed();
350         return this->_min_cut();
351     }
352
353     /// Returns a table with the flow values for each pair of vertices.
354     ///
355     /// Time Complexity:  $O(V^2 + E)$ 
356     vector<vector<int>> flow_table() {
357         this->check_computed();
358         return this->_flow_table();
359     }
360
361     /// Adds a directed edge between u and v and its reverse edge.
362     ///
363     /// Time Complexity:  $O(1)$ ;
364     void add_to_sink(const int u, const int cap) {
365         assert(!COMPUTED);
366         assert(src <= u), assert(u < sink);
367         this->_add_edge(u, sink, cap);
368     }
369
370     /// Adds a directed edge between u and v and its reverse edge.
371     ///
372     /// Time Complexity:  $O(1)$ ;
373     void add_to_src(const int v, const int cap) {
374         assert(!COMPUTED);
375         assert(src < v), assert(v <= sink);
376         this->_add_edge(src, v, cap);
377     }
378
379     /// Adds a directed edge between u and v and its reverse edge.
380     ///
381     /// Time Complexity:  $O(1)$ ;
382     void add_edge(const int u, const int v, const int cap) {
383         assert(!COMPUTED);
384         assert(src <= u), assert(u <= sink);
385         this->_add_edge(u, v, cap);

```



```

384 }
385
386 /// Computes the maximum flow for the network.
387 ///
388 /// Time Complexity:  $O(V^2 \cdot E)$  or  $O(E \cdot \sqrt{V})$  for matching.
389 int max_flow() {
390     this->check_computed();
391     return this->_max_flow;
392 }
393 };

```

5.16. Dsu

```

1 // Remove comments to add rollback
2 class DSU {
3 public:
4     vector<int> root, sz;
5     // stack<tuple<int, int, int>> old_root, old_sz;
6
7     DSU(const int n) {
8         root.resize(n + 1);
9         iota(root.begin(), root.begin() + n + 1, 0ll);
10        sz.resize(n + 1, 1);
11    }
12
13    /// Returns the id of the set in which the element x belongs.
14    ///
15    /// Time Complexity:  $O(1)$ 
16    int Find(const int x) {
17        if (root[x] == x)
18            return x;
19        return root[x] = Find(root[x]);
20        // DONT USE PATH COMPRESSION WITH ROLLBACK!!
21        // return Find(root[x]);
22    }
23
24    /// Unites two sets in which u and v belong.
25    /// Returns false if they already belong to the same set.
26    ///
27    /// Time Complexity:  $O(1)$ 
28    bool Union(int u, int v /* , int idx */) {
29        u = Find(u), v = Find(v);
30        if (u == v)
31            return false;
32
33        if (sz[u] < sz[v])
34            swap(u, v);
35
36        // old_root.emplace(idx, v, root[v]);
37        // old_sz.emplace(idx, u, sz[u]);
38        root[v] = u;
39        sz[u] += sz[v];
40        return true;
41    }
42
43    // void rollback() {
44    //     int idx, u, val;
45    //     tie(idx, u, val) = old_root.top();
46    //     old_root.pop();
47    //     root[u] = val;
48    //     tie(idx, u, val) = old_sz.top();
49    //     old_sz.pop();
50    //     sz[u] = val;
51    // }

```

```
52 };
```

5.17. Dsu On Tree

```

1 /// Problem: What's the level of the subtree of u which contains the most
2   number
3   of nodes? In case of tie, choose the level with small number.
4
5 vector<int> sub_sz(const int root_idx, const vector<vector<int>>& adj) {
6     vector<int> sub(adj.size());
7     function<int(int, int)> dfs = [&](const int u, const int p) {
8         sub[u] = 1;
9         for (int v : adj[u])
10             if (v != p)
11                 sub[u] += dfs(v, u);
12         return sub[u];
13     };
14     dfs(root_idx, -1);
15     return sub;
16 }
17
18 vector<int> sz;
19 int dep[MAXN];
20 vector<vector<int>> adj(MAXN);
21 int maxx, ans;
22 void add(int u, int p, int l, int big_child, int val) {
23     dep[l] += val;
24     if (dep[l] > maxx || (dep[l] == maxx && l < ans)) {
25         ans = l;
26         maxx = dep[l];
27     }
28     for (int v : adj[u]) {
29         if (v == p || big_child == v)
30             continue;
31         add(v, u, l + 1, big_child, val);
32     }
33 }
34
35 vector<int> q(MAXN);
36 void dfs(int u, int p, int l, bool keep) {
37     int idx = -1, val = -1;
38     for (int v : adj[u]) {
39         if (v == p)
40             continue;
41         if (sz[v] > val) {
42             val = sz[v];
43             idx = v;
44         }
45     }
46     // idx now contains the index of the node of the biggest subtree
47     for (int v : adj[u]) {
48         if (v == p || v == idx)
49             continue;
50         // precalculate the answer for small subtrees
51         dfs(v, u, l + 1, 0);
52     }
53
54     if (idx != -1) {
55         // precalculate the answer for the biggest subtree and keep the results
56         dfs(idx, u, l + 1, 1);
57     }
58
59     // Change below to apply the brute force you need. GENERALLY YOU SHOULD ONLY
60     // MODIFY BELOW.

```

```

60 // bruteforce all subtrees other than idx
61 add(u, p, l, idx, l);
62
63 // the answer of u is the level ans. As it is relative to the input tree we
64 // need to subtract it to the current level of u
65 q[u] = ans - l;
66 if (keep == 0) {
67     // removing the calculated answer for the subtree, if it doesn't belong
68     // to
69     // the biggest subtree of it's parent (keep = 0)
70     add(u, p, l, -1, -1);
71     // clearing the answer
72     maxx = 0, ans = 0;
73 }
74
75 /// MODIFY TO WORK WITH DISCONNECTED GRAPHS!!!
76 ///
77 /// Time Complexity: O(n log n)
78 void precalculate() {
79     sz = sub_sz(1, adj);
80     dfs(1, -1, 0, 0);
81 }

```

5.18. Floyd Warshall

```

1 /// Put n = n + 1 for 1 based.
2 void floyd_warshall(const int n) {
3     // OBS: Always assign adj[i][i] = 0.
4     for (int i = 0; i < n; i++)
5         adj[i][i] = 0;
6
7     for (int k = 0; k < n; k++)
8         for (int i = 0; i < n; i++)
9             for (int j = 0; j < n; j++)
10                 adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
11 }

```

5.19. Functional Graph

```

1 // Based on:
2 // http://maratona.ic.unicamp.br/MaratonaVerao2020/lecture-b/20200122.pdf
3 class Functional_Graph {
4     // FOR DIRECTED GRAPH
5     private:
6     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis) {
7         int id_cycle = cycle_cnt++;
8         int cur_id = 0;
9         this->first[id_cycle] = u;
10
11         while(!vis[u]) {
12             vis[u] = true;
13
14             this->cycle[id_cycle].push_back(u);
15
16             this->in_cycle[u] = true;
17             this->cycle_id[u] = id_cycle;
18             this->id_in_cycle[u] = cur_id;
19             this->near_in_cycle[u] = u;
20             this->id_near_cycle[u] = id_cycle;
21             this->cycle_dist[u] = 0;
22 }

```

```

23     u = nxt[u];
24     cur_id++;
25 }
26 }
27
28 // Time Complexity: O(V)
29 void build(int n, int indexed_from, vector<int> &nxt, vector<int>
30     &in_degree) {
31     queue<int> q;
32     vector<bool> vis(n + indexed_from);
33     for(int i = indexed_from; i < n + indexed_from; i++) {
34         if(in_degree[i] == 0) {
35             q.push(i);
36             vis[i] = true;
37         }
38     }
39
40     vector<int> process_order;
41     process_order.reserve(n + indexed_from);
42     while(!q.empty()) {
43         int u = q.front();
44         q.pop();
45
46         process_order.push_back(u);
47
48         if(--in_degree[nxt[u]] == 0) {
49             q.push(nxt[u]);
50             vis[nxt[u]] = true;
51         }
52     }
53
54     int cycle_cnt = 0;
55     for(int i = indexed_from; i < n + indexed_from; i++)
56         if(!vis[i])
57             compute_cycle(i, nxt, vis);
58
59     for(int i = (int)process_order.size() - 1; i >= 0; i--) {
60         int u = process_order[i];
61
62         this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
63         this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
64         this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
65     }
66 }
67
68 void allocate(int n, int indexed_from) {
69     this->cycle.resize(n + indexed_from);
70     this->first.resize(n + indexed_from);
71
72     this->in_cycle.resize(n + indexed_from, false);
73     this->cycle_id.resize(n + indexed_from, -1);
74     this->id_in_cycle.resize(n + indexed_from, -1);
75     this->near_in_cycle.resize(n + indexed_from);
76     this->id_near_cycle.resize(n + indexed_from);
77     this->cycle_dist.resize(n + indexed_from);
78 }
79
80 public:
81 Functional_Graph(int n, int indexed_from, vector<int> &nxt, vector<int>
82     &in_degree) {
83     this->allocate(n, indexed_from);
84     this->build(n, indexed_from, nxt, in_degree);
85 }
86
87 // THE CYCLES ARE ALWAYS INDEXED BY ZERO!

```

```

86 // number of cycles
87 int cycle_cnt = 0;
88 // Vertices present in the i-th cycle.
89 vector<vector<int>> cycle;
90 // first vertex of the i-th cycle
91 vector<int> first;
92
93 // The i-th vertex is present in any cycle?
94 vector<bool> in_cycle;
95 // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
96 // cycle.
97 vector<int> cycle_id;
98 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
99 // belong to any cycle.
100 vector<int> id_in_cycle;
101 // Represents the id of the nearest vertex present in a cycle.
102 vector<int> near_in_cycle;
103 // Represents the id of the nearest cycle.
104 vector<int> id_near_cycle;
105 // Distance to the nearest cycle.
106 vector<int> cycle_dist;
107 // Represent the id of the component of the vertex.
108 // Equal to id_near_cycle
109 vector<int> &comp = id_near_cycle;
110 };
111
112 class Functional_Graph {
113     // FOR UNDIRECTED GRAPH
114     private:
115     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis,
116         vector<vector<int>> &adj) {
117         int id_cycle = cycle_cnt++;
118         int cur_id = 0;
119         this->first[id_cycle] = u;
120
121         while(!vis[u]) {
122             vis[u] = true;
123
124             this->cycle[id_cycle].push_back(u);
125             nxt[u] = find_nxt(u, vis, adj);
126             if(nxt[u] == -1)
127                 nxt[u] = this->first[id_cycle];
128
129             this->in_cycle[u] = true;
130             this->cycle_id[u] = id_cycle;
131             this->id_in_cycle[u] = cur_id;
132             this->near_in_cycle[u] = u;
133             this->id_near_cycle[u] = id_cycle;
134             this->cycle_dist[u] = 0;
135
136             u = nxt[u];
137             cur_id++;
138         }
139     }
140
141     int find_nxt(int u, vector<bool> &vis, vector<vector<int>> &adj) {
142         for(int v: adj[u])
143             if(!vis[v])
144                 return v;
145         return -1;
146     }
147
148     // Time Complexity: O(V + E)

```

```

147 void build(int n, int indexed_from, vector<int> &degree,
148     vector<vector<int>> &adj) {
149     queue<int> q;
150     vector<bool> vis(n + indexed_from, false);
151     vector<int> nxt(n + indexed_from);
152     for(int i = indexed_from; i < n + indexed_from; i++) {
153         if(adj[i].size() == 1) {
154             q.push(i);
155             vis[i] = true;
156         }
157     }
158
159     vector<int> process_order;
160     process_order.reserve(n + indexed_from);
161     while(!q.empty()) {
162         int u = q.front();
163         q.pop();
164
165         process_order.push_back(u);
166
167         nxt[u] = find_nxt(u, vis, adj);
168         if(--degree[nxt[u]] == 1) {
169             q.push(nxt[u]);
170             vis[nxt[u]] = true;
171         }
172     }
173
174     int cycle_cnt = 0;
175     for(int i = indexed_from; i < n + indexed_from; i++)
176         if(!vis[i])
177             compute_cycle(i, nxt, vis, adj);
178
179     for(int i = (int)process_order.size() - 1; i >= 0; i--) {
180         int u = process_order[i];
181
182         this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
183         this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
184         this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
185     }
186
187     void allocate(int n, int indexed_from) {
188         this->cycle.resize(n + indexed_from);
189         this->first.resize(n + indexed_from);
190
191         this->in_cycle.resize(n + indexed_from, false);
192         this->cycle_id.resize(n + indexed_from, -1);
193         this->id_in_cycle.resize(n + indexed_from, -1);
194         this->near_in_cycle.resize(n + indexed_from);
195         this->id_near_cycle.resize(n + indexed_from);
196         this->cycle_dist.resize(n + indexed_from);
197     }
198
199     public:
200     Functional_Graph(int n, int indexed_from, vector<int> degree,
201         vector<vector<int>> &adj) {
202         this->allocate(n, indexed_from);
203         this->build(n, indexed_from, degree, adj);
204     }
205
206     // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
207
208     // number of cycles
209     int cycle_cnt = 0;
210     // Vertices present in the i-th cycle.

```

```

210 vector<vector<int>> cycle;
211 // first vertex of the i-th cycle
212 vector<int> first;
213
214 // The i-th vertex is present in any cycle?
215 vector<bool> in_cycle;
216 // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
    cycle.
217 vector<int> cycle_id;
218 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
    belong to any cycle.
219 vector<int> id_in_cycle;
220 // Represents the id of the nearest vertex present in a cycle.
221 vector<int> near_in_cycle;
222 // Represents the id of the nearest cycle.
223 vector<int> id_near_cycle;
224 // Distance to the nearest cycle.
225 vector<int> cycle_dist;
226 // Represent the id of the component of the vertex.
227 // Equal to id_near_cycle
228 vector<int> &comp = id_near_cycle;
229 };

```

5.20. Girth (Shortest Cycle In A Graph)

```

1 int bfs(const int src) {
2     vector<int> dist(MAXN, INF);
3     queue<pair<int, int>> q;
4
5     q.emplace(src, -1);
6     dist[src] = 0;
7
8     int ans = INF;
9     while (!q.empty()) {
10         pair<int, int> aux = q.front();
11         const int u = aux.first, p = aux.second;
12         q.pop();
13
14         for (const int v : adj[u]) {
15             if (v == p)
16                 continue;
17             if (dist[v] < INF)
18                 ans = min(ans, dist[u] + dist[v] + 1);
19             else {
20                 dist[v] = dist[u] + 1;
21                 q.emplace(v, u);
22             }
23         }
24     }
25
26     return ans;
27 }
28
29 /// Returns the shortest cycle in the graph
30 ///
31 /// Time Complexity:  $O(V^2)$ 
32 int get_girth(const int n) {
33     int ans = INF;
34     for (int u = 1; u <= n; u++)
35         ans = min(ans, bfs(u));
36     return ans;
37 }

```

5.21. Hld

```

1 class HLD {
2 private:
3     int n;
4     // number of nodes below the i-th node
5     vector<int> sz;
6
7 private:
8     void allocate() {
9         // this->id_in_tree.resize(this->n + 1, -1);
10        this->chain_head.resize(this->n + 1, -1);
11        this->chain_id.resize(this->n + 1, -1);
12        this->sz.resize(this->n + 1);
13        this->parent.resize(this->n + 1, -1);
14        // this->id_in_chain.resize(this->n + 1, -1);
15        // this->chain_size.resize(this->n + 1);
16    }
17
18    int get_sz(const int u, const int p, const vector<vector<int>> &adj) {
19        this->sz[u] = 1;
20        for (const int v : adj[u]) {
21            if (v == p)
22                continue;
23            this->sz[u] += this->get_sz(v, u, adj);
24        }
25        return this->sz[u];
26    }
27
28    void dfs(const int u, const int id, const int p,
29            const vector<vector<int>> &adj, int &nidx) {
30        // this->id_in_tree[u] = nidx++;
31        this->chain_id[u] = id;
32        // this->id_in_chain[u] = chain_size[id]++;
33        this->parent[u] = p;
34
35        if (this->chain_head[id] == -1)
36            this->chain_head[id] = u;
37
38        int maxx = -1, idx = -1;
39        for (const int v : adj[u]) {
40            if (v == p)
41                continue;
42            if (sz[v] > maxx) {
43                maxx = sz[v];
44                idx = v;
45            }
46        }
47
48        if (idx != -1)
49            this->dfs(idx, id, u, adj, nidx);
50
51        for (const int v : adj[u]) {
52            if (v == idx || v == p)
53                continue;
54            this->dfs(v, this->number_of_chains++, u, adj, nidx);
55        }
56    }
57
58    void build(const int root_idx, const vector<vector<int>> &adj) {
59        this->get_sz(root_idx, -1, adj);
60        int nidx = 0;
61        this->dfs(root_idx, 0, -1, adj, nidx);
62    }
63 }

```

```

64 // int _compute(const int u, const int limit, Seg_Tree &st) {
65 //     int ans = 0, v;
66 //     for (v = u; chain_id[v] != chain_id[limit];
67 //         v = parent[chain_head[chain_id[v]]]) {
68 //         // change below
69 //         ans = max(ans, st.query(id_in_tree[chain_head[chain_id[v]]],
70 //                                 id_in_tree[v]));
71 //     }
72 //     ans = max(ans, st.query(id_in_tree[limit], id_in_tree[v]));
73 //     return ans;
74 // }
75
76 public:
77     /// Builds the chains.
78     ///
79     /// Time Complexity: O(n)
80     HLD(const int root_idx, const vector<vector<int>> &adj) : n(adj.size()) {
81         allocate();
82         build(root_idx, adj);
83     }
84
85     /// Computes the paths until a limit using segment tree.
86     /// Uncomment id_in_tree!!!
87     ///
88     /// Time Complexity: O(log^2(n))
89     int compute(const int u, const int limit, Seg_Tree &st) {
90         // return _compute(u, limit, st);
91         // }
92
93         // TAKE CARE, YOU MAY GET MLE!!!
94         // the chains are indexed from 0
95         int number_of_chains = 1;
96         // topmost node of the chain
97         vector<int> chain_head;
98         // id of the node based on the order of the dfs (indexed by 0)
99         // vector<int> id_in_tree;
100         // id of the i-th node in his chain
101         // vector<int> id_in_chain;
102         // id of the chain that the i-th node belongs
103         vector<int> chain_id;
104         // size of the i-th chain
105         // vector<int> chain_size;
106         // parent of the i-th node, -1 for root
107         vector<int> parent;
108     };

```

```

16 vector<int> minv(m + 1, INF);
17 vector<int> way(m + 1, 0);
18 vector<bool> used(m + 1, 0);
19 p[0] = i;
20 int k0 = 0;
21 do {
22     used[k0] = 1;
23     int i0 = p[k0], delta = INF, k1;
24     for (int j = 1; j <= m; j++) {
25         if (!used[j]) {
26             const int cur = matrix[i0 - 1][j - 1] - u[i0] - v[j];
27             if (cur < minv[j]) {
28                 minv[j] = cur;
29                 way[j] = k0;
30             }
31             if (minv[j] < delta) {
32                 delta = minv[j];
33                 k1 = j;
34             }
35         }
36     }
37     for (int j = 0; j <= m; j++) {
38         if (used[j]) {
39             u[p[j]] += delta;
40             v[j] -= delta;
41         } else {
42             minv[j] -= delta;
43         }
44     }
45     k0 = k1;
46 } while (p[k0]);
47 do {
48     const int k1 = way[k0];
49     p[k0] = p[k1];
50     k0 = k1;
51 } while (k0);
52 }
53 vector<int> ans(n, -1);
54 for (int j = 1; j <= m; j++) {
55     if (!p[j])
56         continue;
57     ans[p[j] - 1] = j - 1;
58 }
59 return {ans, -v[0]};
60 }

```

5.22. Hungarian

```

1 /// Returns a vector p of size n, where p[i] is the match for i
2 /// and the minimum cost.
3 ///
4 /// Code copied from:
5 ///
6 /// github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/Graph/Hungarian.cpp
7 /// Time Complexity: O(n^2 * m)
8 pair<vector<int>, int> solve(const vector<vector<int>> &matrix) {
9     const int n = matrix.size();
10    if (n == 0)
11        return {vector<int>(), 0};
12    const int m = matrix[0].size();
13    assert(n <= m);
14    vector<int> u(n + 1, 0), v(m + 1, 0), p(m + 1, 0), way, minv;
15    for (int i = 1; i <= n; i++) {

```

5.23. Kuhn

```

1 /// Created by viniciustht
2 struct Kuhn {
3     vector<vector<int>> adj;
4     vector<int> matchA, matchB, marcB;
5     int n, m;
6     bool matched = false;
7     Kuhn(int n, int m) : n(n), m(m) {
8         adj.resize(n, vector<int>());
9         matchA.resize(n);
10        matchB = marcB = vector<int>(m);
11    }
12    void add_edge(int u, int v) {
13        adj[u].emplace_back(v);
14        matched = false;
15    }
16    bool dfs(int u) {

```

```

17     for (int &v : adj[u]) {
18         if (marcB[v]) // || w > mid // use with binary search
19             continue;
20         marcB[v] = 1;
21         if (matchB[v] == -1 or dfs(matchB[v])) {
22             matchB[v] = u;
23             matchA[u] = v;
24             return true;
25         }
26     }
27     return false;
28 }
29
30 int matching() {
31     memset(marcA.data(), -1, sizeof(int) * n);
32     memset(matchB.data(), -1, sizeof(int) * m);
33     // shuffle(adj.begin(), adj.end(), rng); // se o grafo pode ser esparso
34     // for (auto v : adj)
35     //     shuffle(v.begin(), v.end(), rng);
36     int res = 0;
37     bool aux = true;
38     while (aux) {
39         memset(marcB.data(), 0, sizeof(int) * m);
40         aux = false;
41         for (int i = 0; i < n; i++) {
42             if (matchA[i] != -1)
43                 continue;
44             if (dfs(i)) {
45                 res++;
46                 aux = true;
47             }
48         }
49     }
50     matched = true;
51     return res;
52 }
53 void print_matching() {
54     if (!matched)
55         matching();
56     for (int i = 0; i < n; i++)
57         if (matchA[i] != -1)
58             cerr << i + 1 << " " << matchA[i] + 1 << endl;
59 }
60 };

```

5.24. Lca

```

1 // #define DIST
2 // #define COST
3 /// UNCOMMENT ALSO THE LINE BELOW FOR COST!
4
5 // clang-format off
6 class LCA {
7 private:
8     int n;
9     // INDEXED from 0 or 1??
10    int indexed_from;
11    /// Store all log2 from 1 to n
12    vector<int> lg;
13    // level of the i-th node (height)
14    vector<int> level;
15    // matrix to store the ancestors of each node in power of 2 levels
16    vector<vector<int>> anc;
17    #ifdef DIST

```

```

18    vector<int> dist;
19    #endif
20    #ifdef COST
21    // int NEUTRAL_VALUE = -INF; // MAX COST
22    // int combine(const int a, const int b) {return max(a, b);}
23
24    // int NEUTRAL_VALUE = INF; // MIN COST
25    // int combine(const int a, const int b) {return min(a, b);}
26    vector<vector<int>> cost;
27    #endif
28
29 private:
30     void allocate() {
31         // initializes a matrix [n][lg n] with -1
32         this->build_log_array();
33         this->anc.resize(n + 1, vector<int>(lg[n] + 1, -1));
34         this->level.resize(n + 1, -1);
35         #ifdef DIST
36         this->dist.resize(n + 1, 0);
37         #endif
38         #ifdef COST
39         this->cost.resize(n + 1, vector<int>(lg[n] + 1, NEUTRAL_VALUE));
40         #endif
41     }
42
43     void build_log_array() {
44         this->lg.resize(this->n + 1);
45         for (int i = 2; i <= this->n; i++)
46             this->lg[i] = this->lg[i / 2] + 1;
47     }
48
49     void build_anc() {
50         for (int j = 1; j < anc.front().size(); j++)
51             for (int i = 0; i < anc.size(); i++)
52                 if (this->anc[i][j - 1] != -1) {
53                     this->anc[i][j] = this->anc[this->anc[i][j - 1]][j - 1];
54                     #ifdef COST
55                     this->cost[i][j] =
56                         combine(this->cost[i][j - 1], this->cost[anc[i][j - 1]][j -
57                             1]);
58                     #endif
59                 }
60     }
61
62     void build_weighted(const vector<vector<pair<int, int>>> &adj) {
63         this->dfs_LCA_weighted(this->indexed_from, -1, 1, 0, adj);
64         this->build_anc();
65     }
66
67     void dfs_LCA_weighted(const int u, const int p, const int l, const int d,
68                          const vector<vector<pair<int, int>>> &adj) {
69         this->level[u] = l;
70         this->anc[u][0] = p;
71         #ifdef DIST
72         this->dist[u] = d;
73         #endif
74
75         for (const pair<int, int> &x : adj[u]) {
76             int v = x.first, w = x.second;
77             if (v == p)
78                 continue;
79             #ifdef COST
80             this->cost[v][0] = w;
81             #endif
82             this->dfs_LCA_weighted(v, u, l + 1, d + w, adj);

```

```

82     }
83 }
84
85 void build_unweighted(const vector<vector<int>> &adj) {
86     this->dfs_LCA_unweighted(this->indexed_from, -1, 1, 0, adj);
87     this->build_anc();
88 }
89
90 void dfs_LCA_unweighted(const int u, const int p, const int l, const int d,
91                       const vector<vector<int>> &adj) {
92     this->level[u] = l;
93     this->anc[u][0] = p;
94     #ifdef DIST
95     this->dist[u] = d;
96     #endif
97
98     for (const int v : adj[u]) {
99         if (v == p)
100             continue;
101         this->dfs_LCA_unweighted(v, u, l + 1, d + 1, adj);
102     }
103 }
104
105 // go up k levels from x
106 int lca_go_up(int x, int k) {
107     for (int i = 0; k > 0; i++, k >>= 1)
108         if (k & 1) {
109             x = this->anc[x][i];
110             if (x == -1)
111                 return -1;
112         }
113     return x;
114 }
115
116 #ifdef COST
117 /// Query between the an ancestor of v (p) and v. It returns the
118 /// max/min edge between them.
119 int lca_query_cost_in_line(int v, int p) {
120     assert(this->level[v] >= this->level[p]);
121
122     int k = this->level[v] - this->level[p];
123     int ans = NEUTRAL_VALUE;
124
125     for (int i = 0; k > 0; i++, k >>= 1)
126         if (k & 1) {
127             ans = combine(ans, this->cost[v][i]);
128             v = this->anc[v][i];
129         }
130
131     return ans;
132 }
133 #endif
134
135 int get_lca(int a, int b) {
136     // a is below b
137     if (this->level[b] > this->level[a])
138         swap(a, b);
139
140     const int logg = lg[this->level[a]];
141     // putting a and b in the same level
142     for (int i = logg; i >= 0; i--)
143         if (this->level[a] - (1 << i) >= this->level[b])
144             a = this->anc[a][i];
145
146     if (a == b)

```

```

147     return a;
148
149     for (int i = logg; i >= 0; i--)
150         if (this->anc[a][i] != -1 && this->anc[a][i] != this->anc[b][i]) {
151             a = this->anc[a][i];
152             b = this->anc[b][i];
153         }
154
155     return anc[a][0];
156 }
157
158 public:
159     /// Builds an weighted graph.
160     ///
161     /// Time Complexity: O(n*log(n))
162     explicit LCA(const vector<vector<pair<int, int>>> &adj,
163                const int indexed_from)
164         : n(adj.size()), indexed_from(indexed_from) {
165         this->allocate();
166         this->build_weighted(adj);
167     }
168
169     /// Builds an unweighted graph.
170     ///
171     /// Time Complexity: O(n*log(n))
172     explicit LCA(const vector<vector<int>> &adj, const int indexed_from)
173         : n(adj.size()), indexed_from(indexed_from) {
174         this->allocate();
175         this->build_unweighted(adj);
176     }
177
178     /// Goes up k levels from v. If it passes the root, returns -1.
179     ///
180     /// Time Complexity: O(log(k))
181     int go_up(const int v, const int k) {
182         assert(indexed_from <= v, assert(v < this->n + indexed_from);
183         return this->lca_go_up(v, k);
184     }
185
186     /// Returns the parent of v in the LCA dfs from l.
187     ///
188     /// Time Complexity: O(1)
189     int parent(int v) {
190         assert(indexed_from <= v, assert(v < this->n + indexed_from);
191         return this->anc[v][0];
192     }
193
194     /// Returns the LCA of a and b.
195     ///
196     /// Time Complexity: O(log(n))
197     int query_lca(const int a, const int b) {
198         assert(indexed_from <= min(a, b)),
199         assert(max(a, b) < this->n + indexed_from);
200         return this->get_lca(a, b);
201     }
202
203     #ifdef DIST
204     /// Returns the distance from a to b. When the graph is unweighted, it is
205     /// considered 1 as the weight of the edges.
206     ///
207     /// Time Complexity: O(log(n))
208     int query_dist(const int a, const int b) {
209         assert(indexed_from <= min(a, b)),
210         assert(max(a, b) < this->n + indexed_from);

```

```

211     return this->dist[a] + this->dist[b] - 2 * this->dist[this->get_lca(a,
212         b)];
213 }
214 #endif
215 #ifdef COST
216 /// Returns the max/min weight edge from a to b.
217 ///
218 /// Time Complexity: O(log(n))
219 int query_cost(const int a, const int b) {
220     assert(indexed_from <= min(a, b)),
221         assert(max(a, b) < this->n + indexed_from);
222     const int l = this->query_lca(a, b);
223     return combine(this->lca_query_cost_in_line(a, l),
224         this->lca_query_cost_in_line(b, l));
225 }
226 #endif
227 };
228 // clang-format on

```

5.25. Longest Path In Dag

```

1 /// Requires topological_sort.cpp
2
3 /// Returns a vector with the maximal distance from src (must be 0 or 1) to
4 /// every node or a maximal path from src to (n - 1).
5 ///
6 /// Time Complexity: O(n)
7 vector<int> longest_path_in_dag(const int src, const vector<vector<int>>
8     &adj) {
9     const int n = adj.size();
10    vector<int> dp(n, -1), prev(n, -1);
11    dp[src] = 0;
12    for (int u : topological_sort(src, adj))
13        for (int v : adj[u])
14            if (dp[u] != -1 && dp[u] + 1 > dp[v]) {
15                dp[v] = dp[u] + 1;
16                prev[v] = u;
17            }
18
19    // Returns the longest path to each node
20    // return dp;
21
22    vector<int> path;
23    // Assuming that the last node is the node (n - 1)
24    int cur = n - 1;
25    while (cur != -1) {
26        path.emplace_back(cur);
27        cur = prev[cur];
28    }
29    reverse(path.begin(), path.end());
30    // Returns the maximal path from src to (n - 1)
31    return path;

```

5.26. Maximum Independent Set (Set Of Vertices That Arent Directly Connected)

```

1 |IS maximal| = |V| - MAXIMUM_MATCHING

```

5.27. Maximum Path Unweighted Graph

```

1 /// Returns the maximum path between the vertices 0 and n - 1 in a
2 /// unweighted graph.
3 ///
4 /// Time Complexity: O(V + E)
5 int maximum_path(int n) {
6     vector<int> top_order = topological_sort(n);
7     vector<int> pai(n, -1);
8     if(top_order.empty())
9         return -1;
10
11    vector<int> dp(n);
12    dp[0] = 1;
13    for(int u: top_order)
14        for(int v: adj[u])
15            if(dp[u] && dp[u] + 1 > dp[v]) {
16                dp[v] = dp[u] + 1;
17                pai[v] = u;
18            }
19
20    if(dp[n - 1] == 0)
21        return -1;
22
23    vector<int> path;
24    int cur = n - 1;
25    while(cur != -1) {
26        path.pb(cur);
27        cur = pai[cur];
28    }
29    reverse(path.begin(), path.end());
30
31    // cout << path.size() << endl;
32    // for(int x: path) {
33    //     cout << x + 1 << ' ';
34    // }
35    // cout << endl;
36
37    return dp[n - 1];

```

5.28. Min Cost Flow Gpresso

```

1 /// MINIMIZES COST * FLOW!!!
2 /// Code copied from:
3 ///
4 https://github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/Graph/MinC
5 template <class T = int> class MCMF {
6 public:
7     struct Edge {
8         Edge(int a, T b, T c) : to(a), cap(b), cost(c) {}
9         int to;
10        T cap, cost;
11    };
12
13    MCMF(int size) {
14        n = size;
15        edges.resize(n);
16        pot.assign(n, 0);
17        dist.resize(n);
18        visit.assign(n, false);
19    }
20
21    std::pair<T, T> mcmf(int src, int sink) {
22        std::pair<T, T> ans(0, 0);

```



```

22     if (!SPFA(src, sink))
23         return ans;
24     fixPot();
25     // can use dijkstra to speed up depending on the graph
26     while (SPFA(src, sink)) {
27         auto flow = augment(src, sink);
28         ans.first += flow.first;
29         ans.second += flow.first * flow.second;
30         fixPot();
31     }
32     return ans;
33 }
34
35 void addEdge(int from, int to, T cap, T cost) {
36     edges[from].push_back(list.size());
37     list.push_back(Edge(to, cap, cost));
38     edges[to].push_back(list.size());
39     list.push_back(Edge(from, 0, -cost));
40 }
41
42 private:
43 int n;
44 std::vector<std::vector<int>> edges;
45 std::vector<Edge> list;
46 std::vector<int> from;
47 std::vector<T> dist, pot;
48 std::vector<bool> visit;
49
50 /*bool dij(int src, int sink) {
51     T INF = std::numeric_limits<T>::max();
52     dist.assign(n, INF);
53     from.assign(n, -1);
54     visit.assign(n, false);
55     dist[src] = 0;
56     for(int i = 0; i < n; i++) {
57         int best = -1;
58         for(int j = 0; j < n; j++) {
59             if(visit[j]) continue;
60             if(best == -1 || dist[best] > dist[j]) best = j;
61         }
62         if(dist[best] >= INF) break;
63         visit[best] = true;
64         for(auto e : edges[best]) {
65             auto ed = list[e];
66             if(ed.cap == 0) continue;
67             T toDist = dist[best] + ed.cost + pot[best] - pot[ed.to];
68             assert(toDist >= dist[best]);
69             if(toDist < dist[ed.to]) {
70                 dist[ed.to] = toDist;
71                 from[ed.to] = e;
72             }
73         }
74     }
75     return dist[sink] < INF;
76 }*/
77
78 std::pair<T, T> augment(int src, int sink) {
79     std::pair<T, T> flow = {list[from[sink]].cap, 0};
80     for (int v = sink; v != src; v = list[from[v] ^ 1].to) {
81         flow.first = std::min(flow.first, list[from[v]].cap);
82         flow.second += list[from[v]].cost;
83     }
84     for (int v = sink; v != src; v = list[from[v] ^ 1].to) {
85         list[from[v]].cap -= flow.first;
86         list[from[v] ^ 1].cap += flow.first;

```

```

87     }
88     return flow;
89 }
90
91 std::queue<int> q;
92 bool SPFA(int src, int sink) {
93     T INF = std::numeric_limits<T>::max();
94     dist.assign(n, INF);
95     from.assign(n, -1);
96     q.push(src);
97     dist[src] = 0;
98     while (!q.empty()) {
99         int on = q.front();
100        q.pop();
101        visit[on] = false;
102        for (auto e : edges[on]) {
103            auto ed = list[e];
104            if (ed.cap == 0)
105                continue;
106            T toDist = dist[on] + ed.cost + pot[on] - pot[ed.to];
107            if (toDist < dist[ed.to]) {
108                dist[ed.to] = toDist;
109                from[ed.to] = e;
110                if (!visit[ed.to]) {
111                    visit[ed.to] = true;
112                    q.push(ed.to);
113                }
114            }
115        }
116    }
117    return dist[sink] < INF;
118 }
119
120 void fixPot() {
121     T INF = std::numeric_limits<T>::max();
122     for (int i = 0; i < n; i++)
123         if (dist[i] < INF)
124             pot[i] += dist[i];
125 }
126 };

```

5.29. Min Cost Flow Katcl

```

1  /// MINIMIZES COST * FLOW!!!!
2  /// DOESN'T SUPPORT PARALLEL EDGES!!!!!!
3
4  /// Code copied from:
5  ///
6  /// github.com/kth-competitive-programming/kactl/blob/master/content/graph/MinCostMaxFlow.cpp
7  #include <bits/stdc++.h> /// include-line, keep-include
8
9  // #define all(x) begin(x), end(x)
10 // typedef pair<int, int> ii;
11 // typedef vector<int> vi;
12 typedef long long ll;
13 typedef vector<ll> VL;
14 #define sz(x) (int)(x).size()
15 #define rep(i, a, b) for (int i = a; i < (b); ++i)
16
17 const ll INF1 = numeric_limits<ll>::max() / 4;
18
19 // clang-format off
20 struct MCMF {
21     int N;

```

```

21 vector<vi> ed, red;
22 vector<VL> cap, flow, cost;
23 vi seen;
24 VL dist, pi;
25 vector<ii> par;
26
27 MCMF(int N) :
28     N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(cap),
29     seen(N), dist(N), pi(N), par(N) {}
30
31 void addEdge(int from, int to, ll cap, ll cost) {
32     this->cap[from][to] = cap;
33     this->cost[from][to] = cost;
34     ed[from].push_back(to);
35     red[to].push_back(from);
36 }
37
38 void path(int s) {
39     fill(all(seen), 0);
40     fill(all(dist), INF1);
41     dist[s] = 0; ll di;
42
43     __gnu_pbds::priority_queue<pair<ll, int>> q;
44     vector<decltype(q)::point_iterator> its(N);
45     q.push({0, s});
46
47     auto relax = [&](int i, ll cap, ll cost, int dir) {
48         ll val = di - pi[i] + cost;
49         if (cap && val < dist[i]) {
50             dist[i] = val;
51             par[i] = {s, dir};
52             if (its[i] == q.end()) its[i] = q.push({-dist[i], i});
53             else q.modify(its[i], {-dist[i], i});
54         }
55     };
56
57     while (!q.empty()) {
58         s = q.top().second; q.pop();
59         seen[s] = 1; di = dist[s] + pi[s];
60         for (int i : ed[s]) if (!seen[i])
61             relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
62         for (int i : red[s]) if (!seen[i])
63             relax(i, flow[i][s], -cost[i][s], 0);
64     }
65     rep(i, 0, N) pi[i] = min(pi[i] + dist[i], INF1);
66 }
67
68 pair<ll, ll> maxflow(int s, int t) {
69     ll totflow = 0, totcost = 0;
70     while (path(s), seen[t]) {
71         ll fl = INF1;
72         for (int p, r, x = t; tie(p, r) = par[x], x != s; x = p)
73             fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
74         totflow += fl;
75         for (int p, r, x = t; tie(p, r) = par[x], x != s; x = p)
76             if (r) flow[p][x] += fl;
77             else flow[x][p] -= fl;
78     }
79     rep(i, 0, N) rep(j, 0, N) totcost += cost[i][j] * flow[i][j];
80     return {totflow, totcost};
81 }
82
83 // If some costs can be negative, call this before maxflow:
84 void setpi(int s) { // (otherwise, leave this out)
85     fill(all(pi), INF1); pi[s] = 0;

```

```

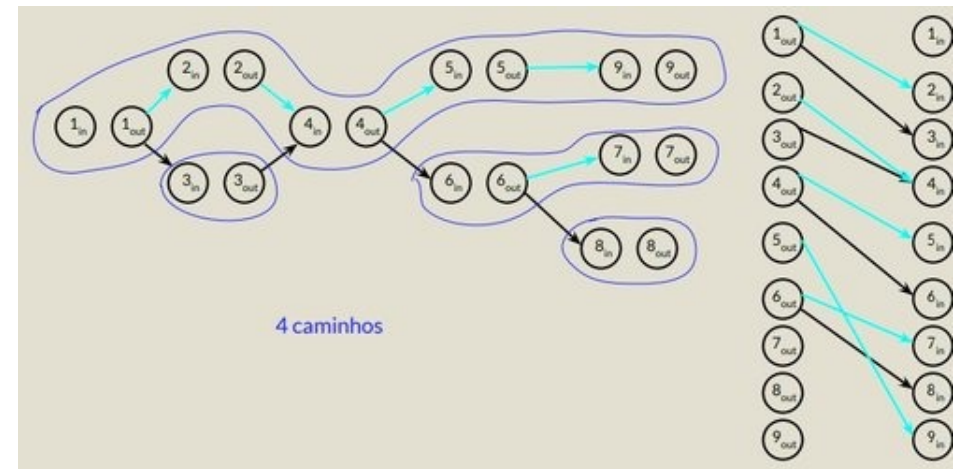
86 int it = N, ch = 1; ll v;
87 while (ch-- && it--)
88     rep(i, 0, N) if (pi[i] != INF1)
89         for (int to : ed[i]) if (cap[i][to])
90             if ((v = pi[i] + cost[i][to]) < pi[to])
91                 pi[to] = v, ch = 1;
92     assert(it >= 0); // negative cost cycle
93 }
94 };
95 // clang-format on

```

5.30. Minimum Edge Cover (Set Of Edges That Are Adjacent To All Vertices)

1 $|E_{\text{minimal}}| = |V| - \text{MAXIMUM_MATCHING}$

5.31. Minimum Path Cover In Dag



5.32. Minimum Path Cover In Dag

- Given the paths we can split the vertices into two different vertices: IN and OUT. Then, we can build a bipartite graph in which the OUT vertices are present on the left side of the graph and the IN vertices on the right side. After that, we create an edge between a vertex on the left side to the right side if there's a connection between them in the original graph.
- The answer at the end will be equal to $|V| - \text{MAXIMUM_MATCHING}$, because the OUT vertices in which don't have a match represent the end of a path.

5.33. Mst

```

1 /// Requires DSU.cpp
2 struct edge {
3     int u, v, w;
4     edge() {}
5     edge(int u, int v, int w) : u(u), v(v), w(w) {}

```

```

6   bool operator<(const edge &a) const { return w < a.w; }
7   };
8
9   /// Returns weight of the minimum spanning tree of the graph.
10  ///
11  /// Time Complexity: O(V log V)
12  int kruskal(int n, vector<edge> &edges) {
13      DSU dsu(n);
14      sort(edges.begin(), edges.end());
15
16      int weight = 0;
17      for (int i = 0; i < edges.size(); i++) {
18          if (dsu.Union(edges[i].u, edges[i].v)) {
19              weight += edges[i].w;
20          }
21      }
22
23      return weight;
24  }
25  }

```

5.34. Number Of Different Spanning Trees In A Complete Graph

```

1  Cayley's formula
2
3  n ^ (n - 2)

```

5.35. Number Of Ways To Make A Graph Connected

```

1  s_{1} * s_{2} * s_{3} * (...) * s_{k} * (n ^ (k - 2))
2  n = number of vertices
3  s_{i} = size of the i-th connected component
4  k = number of connected components

```

5.36. Pruffer Decode

```

1  // IT MUST BE INDEXED BY 0.
2  /// Returns the adjacency matrix of the decoded tree.
3  ///
4  /// Time Complexity: O(V)
5  vector<vector<int>> pruefer_decode(const vector<int> &code) {
6
7      int n = code.size() + 2;
8      vector<vector<int>> adj = vector<vector<int>>(n, vector<int>());
9      vector<int> degree(n, 1);
10     for (int x : code)
11         degree[x]++;
12
13     int ptr = 0;
14     while (degree[ptr] > 1)
15         ++ptr;
16
17     int nxt = ptr;
18     for (int u : code) {
19         adj[u].push_back(nxt);
20         adj[nxt].push_back(u);
21
22         if (--degree[u] == 1 && u < ptr)
23             nxt = u;
24         else {
25             while (degree[++ptr] > 1)
26                 ;

```

```

27         nxt = ptr;
28     }
29 }
30 adj[n - 1].push_back(nxt);
31 adj[nxt].push_back(n - 1);
32
33 return adj;
34 }

```

5.37. Pruffer Encode

```

1  void dfs(int v, const vector<vector<int>> &adj, vector<int> &parent) {
2      for (int u : adj[v]) {
3          if (u != parent[v]) {
4              parent[u] = v;
5              dfs(u, adj, parent);
6          }
7      }
8  }
9
10 // IT MUST BE INDEXED BY 0.
11 /// Returns prueffer code of the tree.
12 ///
13 /// Time Complexity: O(V)
14 vector<int> pruefer_code(const vector<vector<int>> &adj) {
15     int n = adj.size();
16     vector<int> parent(n);
17     parent[n - 1] = -1;
18     dfs(n - 1, adj, parent);
19
20     int ptr = -1;
21     vector<int> degree(n);
22     for (int i = 0; i < n; i++) {
23         degree[i] = adj[i].size();
24         if (degree[i] == 1 && ptr == -1)
25             ptr = i;
26     }
27
28     vector<int> code(n - 2);
29     int leaf = ptr;
30     for (int i = 0; i < n - 2; i++) {
31         int next = parent[leaf];
32         code[i] = next;
33         if (--degree[next] == 1 && next < ptr)
34             leaf = next;
35         else {
36             ptr++;
37             while (degree[ptr] != 1)
38                 ptr++;
39             leaf = ptr;
40         }
41     }
42
43     return code;
44 }

```

5.38. Pruffer Properties

- * After constructing the Prüfer code two vertices will remain. One of them is the highest vertex $n-1$, but nothing **else** can be said about the other one.
- * Each vertex appears in the Prüfer code exactly a fixed number of times - its degree minus one. This can be easily checked, since the degree will

get smaller every time we record its label in the code, **and** we remove it once the degree is 1. For the two remaining vertices **this** fact is also **true**.

5.39. Remove All Bridges From Graph

1. Start a DFS **and** store the leafs in an array.
2. Connect the first leaf vertex in the array with the one in the middle, the second one **and** the middle + 1, **and** so on.

5.40. Scc (Kosaraju)

```

1 class SCC {
2     private:
3         // number of vertices
4         int n;
5         // indicates whether it is indexed from 0 or 1
6         int indexed_from;
7         // reversed graph
8         vector<vector<int>> trans;
9
10    private:
11    void dfs_trans(int u, int id) {
12        comp[u] = id;
13        scc[id].push_back(u);
14
15        for (int v: trans[u])
16            if (comp[v] == -1)
17                dfs_trans(v, id);
18    }
19
20    void get_transpose(vector<vector<int>>& adj) {
21        for (int u = indexed_from; u < this->n + indexed_from; u++)
22            for (int v: adj[u])
23                trans[v].push_back(u);
24    }
25
26    void dfs_fill_order(int u, stack<int> &s, vector<vector<int>>& adj) {
27        comp[u] = true;
28
29        for (int v: adj[u])
30            if (!comp[v])
31                dfs_fill_order(v, s, adj);
32
33        s.push(u);
34    }
35
36    // The main function that finds all SCCs
37    void compute_SCC(vector<vector<int>>& adj) {
38
39        stack<int> s;
40        // Fill vertices in stack according to their finishing times
41        for (int i = indexed_from; i < this->n + indexed_from; i++)
42            if (!comp[i])
43                dfs_fill_order(i, s, adj);
44
45        // Create a reversed graph
46        get_transpose(adj);
47
48        fill(comp.begin(), comp.end(), -1);
49
50        // Now process all vertices in order defined by stack
51        while(s.empty() == false) {

```

```

52            int v = s.top();
53            s.pop();
54
55            if (comp[v] == -1)
56                dfs_trans(v, this->number_of_comp++);
57        }
58    }
59
60    public:
61    // number of the component of the i-th vertex
62    // it's always indexed from 0
63    vector<int> comp;
64    // the i-th vector contains the vertices that belong to the i-th scc
65    // it's always indexed from 0
66    vector<vector<int>> scc;
67    int number_of_comp = 0;
68
69    SCC(int n, int indexed_from, vector<vector<int>>& adj) {
70        this->n = n;
71        this->indexed_from = indexed_from;
72        comp.resize(n + 1);
73        trans.resize(n + 1);
74        scc.resize(n + 1);
75
76        this->compute_SCC(adj);
77    }
78 };

```

5.41. Small To Large (Merge Sets)

```

1 // Problem: How many distinct colors in the subtree of u?
2
3 vector<int> sub_sz(const int root_idx, const vector<vector<int>> &adj) {
4     vector<int> sub(adj.size());
5     function<int(int, int)> dfs = [&](const int u, const int p) {
6         sub[u] = 1;
7         for (int v : adj[u])
8             if (v != p)
9                 sub[u] += dfs(v, u);
10        return sub[u];
11    };
12    dfs(root_idx, -1);
13    return sub;
14 }
15
16 vi color(MAXN), b(MAXN);
17 vector<int> sz;
18 int ans[MAXN];
19 vector<vector<int>> adj(MAXN);
20
21 set<int> dfs(int u, int p, int l) {
22     int idx = -1, val = -1;
23     for (int v : adj[u]) {
24         if (v == p)
25             continue;
26         if (sz[v] > val) {
27             val = sz[v];
28             idx = v;
29         }
30     }
31
32     set<int> s;
33     if (idx != -1)
34         // precalculate the answer for the biggest subtree and keep the results

```

```

35     s = dfs(idx, u, l + 1);
36
37 // idx now contains the index of the node of the biggest subtree
38 for (int v : adj[u]) {
39     if (v == p || v == idx)
40         continue;
41     // precalculate the answer for small subtrees
42     for (int x : dfs(v, u, l + 1))
43         s.emplace(x);
44 }
45
46 s.emplace(color[u]);
47 ans[u] = s.size();
48 return s;
49 }
50
51 /// MODIFY TO WORK WITH DISCONNECTED GRAPHS!!!
52 ///
53 /// Time Complexity: O(n log n)
54 void precalculate() {
55     sz = sub_sz(1, adj);
56     dfs(1, -1, 0);
57 }

```

5.42. Topological Sort

```

1 /// Time Complexity: O(V + E)
2 vector<int> topological_sort(const int indexed_from,
3                             const vector<vector<int>> &adj) {
4     const int n = adj.size();
5     vector<int> in_degree(n, 0);
6
7     for (int u = indexed_from; u < n; ++u)
8         for (const int v : adj[u])
9             in_degree[v]++;
10
11     queue<int> q;
12     for (int i = indexed_from; i < n; ++i)
13         if (in_degree[i] == 0)
14             q.emplace(i);
15
16     int cnt = 0;
17     vector<int> top_order;
18     while (!q.empty()) {
19         const int u = q.front();
20         q.pop();
21
22         top_order.emplace_back(u);
23         ++cnt;
24
25         for (const int v : adj[u])
26             if (--in_degree[v] == 0)
27                 q.emplace(v);
28     }
29
30     if (cnt != n - indexed_from) {
31         // There exists a cycle in the graph
32         return vector<int>();
33     }
34
35     return top_order;
36 }

```

5.43. Tree Diameter

```

1 namespace tree {
2     /// Returns a pair which contains the most distant vertex from src and the
3     /// value of this distance.
4     pair<int, int> bfs(const int src, const vector<vector<int>> &adj) {
5         queue<tuple<int, int, int>> q;
6         q.emplace(0, src, -1);
7         int furthest = src, dist = 0;
8         while (!q.empty()) {
9             int d, u, p;
10            tie(d, u, p) = q.front();
11            q.pop();
12            if (d > dist) {
13                furthest = u;
14                dist = d;
15            }
16            for (const int v : adj[u]) {
17                if (v == p)
18                    continue;
19                q.emplace(d + 1, v, u);
20            }
21        }
22        return make_pair(furthest, dist);
23    }
24
25    /// Returns the length of the diameter and two vertices that belong to it.
26    ///
27    /// Time Complexity: O(n)
28    tuple<int, int, int> diameter(const int root_idx,
29                                const vector<vector<int>> &adj) {
30        int ini = bfs(root_idx, adj).first, end, dist;
31        tie(end, dist) = bfs(ini, adj);
32        return {dist, ini, end};
33    }
34 }; // namespace tree

```

5.44. Tree Distance

```

1 vector<pair<int, int>> sub(MAXN, pair<int, int>(0, 0));
2
3 void subu(int u, int p) {
4     for (const pair<int, int> x : adj[u]) {
5         int v = x.first, w = x.second;
6         if (v == p)
7             continue;
8         subu(v, u);
9         if (sub[v].first + w > sub[u].first) {
10             swap(sub[u].first, sub[u].second);
11             sub[u].first = sub[v].first + w;
12         } else if (sub[v].first + w > sub[u].second) {
13             sub[u].second = sub[v].first + w;
14         }
15     }
16 }
17
18 /// Contains the maximum distance to the node i
19 vector<int> ans(MAXN);
20
21 void dfs(int u, int d, int p) {
22     ans[u] = max(d, sub[u].first);
23     for (const pair<int, int> x : adj[u]) {
24         int v = x.first, w = x.second;
25         if (v == p)

```

```

26     continue;
27     if (sub[v].first + w == ans[u]) {
28         dfs(v, max(d, sub[u].second) + w, u);
29     } else {
30         dfs(v, ans[u] + w, u);
31     }
32 }
33 }
34
35 // Returns the maximum tree distance
36 int solve() {
37     subu(0, -1);
38     dfs(0, 0, -1);
39     return *max_element(ans.begin(), ans.end());
40 }

```

5.45. Tree Isomorphism

```

1  /// THE VALUES OF THE VERTICES MUST BELONG FROM 1 TO N.
2  namespace tree {
3  mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
4
5  vector<uint64_t> base;
6  uint64_t build(const int u, const int p, const vector<vector<int>> &adj,
7               const int level = 0) {
8      if (level == base.size())
9          base.emplace_back(rng());
10     uint64_t hsh = 1;
11     vector<uint64_t> child;
12     for (const int v : adj[u])
13         if (v != p)
14             child.emplace_back(build(v, u, adj, level + 1));
15     sort(child.begin(), child.end());
16     for (const uint64_t x : child)
17         hsh = hsh * base[level] + x;
18     return hsh;
19 }
20
21 /// Returns whether two rooted trees are isomorphic or not.
22 ///
23 /// Time Complexity: O(n)
24 bool same(const int root_1, const vector<vector<int>> &adj1, const int
25          root_2,
26          const vector<vector<int>> &adj2) {
27     if (adj1.size() != adj2.size())
28         return false;
29     return build(root_1, -1, adj1) == build(root_2, -1, adj2);
30 }
31
32 /// Returns whether two non-rooted trees are isomorphic or not.
33 /// REQUIRES centroid.cpp!!!
34 ///
35 /// Time Complexity: O(n)
36 bool same(const int n, const int indexed_from, const vector<vector<int>>
37          &adj1,
38          const vector<vector<int>> &adj2) {
39     vector<int> c1 = centroid(n, indexed_from, adj1),
40               c2 = centroid(n, indexed_from, adj2);
41     for (const int v : c2)
42         if (same(c1.front(), adj1, v, adj2))
43             return true;
44     return false;
45 }
46 } // namespace tree

```

6. Language Stuff

6.1. Binary String To Int

```

1 int y = bitset<number_of_bits>(string_var).to_ulong();
2 Ex : x = 1010, number_of_bits = 32;
3 y = bitset<32>(x).to_ulong(); // y = 10

```

6.2. Check Char Type

```

1 #include <cctype>
2 isdigit(str[i]); //checa se str[i] é número
3 isalpha(str[i]); //checa se é uma letra
4 islower(str[i]); //checa minúsculo
5 isupper(str[i]); //checa maiúsculo
6 isalnum(str[i]); //checa letra ou número
7 tolower(str[i]); //converte para minúsculo
8 toupper(str[i]); //converte para maiúsculo

```

6.3. Check Overflow

```

1 bool __builtin_add_overflow (type1 a, type2 b, type3 *res)
2 bool __builtin_sadd_overflow (int a, int b, int *res)
3 bool __builtin_saddl_overflow (long int a, long int b, long int *res)
4 bool __builtin_saddll_overflow (long long int a, long long int b, long long
5 int *res)
6 bool __builtin_uadd_overflow (unsigned int a, unsigned int b, unsigned int
7 *res)
8 bool __builtin_uaddl_overflow (unsigned long int a, unsigned long int b,
9 unsigned long int *res)
10 bool __builtin_uaddll_overflow (unsigned long long int a, unsigned long long
11 int b, unsigned long long int *res)
12
13 bool __builtin_sub_overflow (type1 a, type2 b, type3 *res)
14 bool __builtin_ssub_overflow (int a, int b, int *res)
15 bool __builtin_ssubl_overflow (long int a, long int b, long int *res)
16 bool __builtin_ssubll_overflow (long long int a, long long int b, long long
17 int *res)
18 bool __builtin_usub_overflow (unsigned int a, unsigned int b, unsigned int
19 *res)
20 bool __builtin_usubl_overflow (unsigned long int a, unsigned long int b,
21 unsigned long int *res)
22 bool __builtin_usubll_overflow (unsigned long long int a, unsigned long long
23 int b, unsigned long long int *res)
24
25 bool __builtin_mul_overflow (type1 a, type2 b, type3 *res)
26 bool __builtin_smul_overflow (int a, int b, int *res)
27 bool __builtin_smull_overflow (long int a, long int b, long int *res)
28 bool __builtin_smulll_overflow (long long int a, long long int b, long long
29 int *res)
30 bool __builtin_umul_overflow (unsigned int a, unsigned int b, unsigned int
31 *res)
32 bool __builtin_umull_overflow (unsigned long int a, unsigned long int b,
33 unsigned long int *res)
34 bool __builtin_umulll_overflow (unsigned long long int a, unsigned long long
35 int b, unsigned long long int *res)

```

6.4. Counting Bits

```

1 #pragma GCC target ("sse4.2")
2 // Use the pragma above to optimize the time complexity to O(1)
3 __builtin_popcount(int) -> Number of active bits

```

```

4 __builtin_popcountll(ll) -> Number of active bits
5 __builtin_ctz(int) -> Number of trailing zeros in binary representation
6 __builtin_clz(int) -> Number of leading zeros in binary representation
7 __builtin_parity(int) -> Parity of the number of bits

```

6.5. Gen Random Numbers (Rng)

```

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

```

6.6. Int To Binary String

```

1 string s = bitset<number_of_bits>(intVar).to_string();
2 Ex : x = 10, number_of_bits = 32;
3 s = bitset<32>(x).to_string(); // s = 00...0001010

```

6.7. Int To String

```

1 int a;
2 string b = to_string(a);

```

6.8. Permutation

```

1 int v[] = {1,2,3};
2 sort(v, v+3);
3 do {
4     cout << v[0] << ' ' << v[1] << ' ' << v[2];
5     while(next_permutation(v, v+3));

```

6.9. Print Int128 T

```

1 void print(__int128_t x) {
2     if (x == 0)
3         return void(cout << 0 << endl);
4     bool neg = false;
5     if (x < 0) {
6         neg = true;
7         x *= -1;
8     }
9     string ans;
10    while (x) {
11        ans += char(x % 10 + '0');
12        x /= 10;
13    }
14
15    if (neg)
16        ans += "-";
17    reverse(all(ans));
18    cout << ans << endl;
19 }

```

6.10. Read And Write From File

```

1 freopen("filename.in", "r", stdin);
2 freopen("filename.out", "w", stdout);

```

6.11. Readint

```

1 int readInt() {
2     int a = 0;
3     char c;
4     while (!(c >= '0' && c <= '9'))
5         c = getchar();
6     while (c >= '0' && c <= '9')
7         a = 10 * a + (c - '0'), c = getchar();
8     return a;
9 }

```

6.12. Rotate Left

```

1 vector<int> arr(n); // 1 2 3 4 5 6 7 8 9
2 rotate(arr.begin(), arr.begin() + 3, arr.end()); // 4 5 6 7 8 9 1 2 3

```

6.13. Rotate Right

```

1 vector<int> arr(n); // 1 2 3 4 5 6 7 8 9
2 rotate(arr.begin(), arr.rbegin() + 3, arr.rend()); // 7 8 9 1 2 3 4 5 6

```

6.14. Scanf From String

```

1 char sentence[] = "Rudolph is 12 years old";
2 char str[20];
3 int i;
4 sscanf(sentence, "%s %s %d", str, &i);
5 printf("%s -> %d\n", str, i);
6 // Output: Rudolph -> 12

```

6.15. Split Function

```

1 /// Splits a string into a vector. A separator can be specified
2 /// EX: str=A-B-C -> split -> x = {A,B,C}
3 ///
4 /// Time Complexity: O(s.size())
5 vector<string> split(const string &s, char separator = ' ') {
6     stringstream ss(s);
7     string item;
8     vector<string> tokens;
9     while (getline(ss, item, separator))
10         tokens.emplace_back(item);
11     return tokens;
12 }
13 int main() {
14     vector<string> x = split("cap-one-best-opinion-language", '-');
15     // x = {cap,one,best,opinion,language};
16 }

```

6.16. String To Long Long

```

1 string s = "0xFFFF";
2 int base = 16;
3 string::size_type sz = 0;
4 int ll = stoll(s, &sz, base);
5 // ll = 65535, sz = 6;
6 // if base is equal to 10 you may leave it empty.
7 // OBS: You can place anything (like 0) instead of sz stoll(s,0,base);

```

6.17. Substring

```

1 string s = "abcdef";
2 // s.substr(first position, size);
3 string s2 = s.substr(3, 2); // s2 = "de"
4 // if the size is empty it takes the substring from first pos to the end
5 string s3 = s.substr(2); // s3 = "cdef"

```

6.18. Time Measure

```

1 clock_t start = clock();
2
3 /* Execute the program */
4
5 clock_t end = clock();
6
7 double time_taken = double(end - start) / double(CLOCKS_PER_SEC);

```

6.19. Unique Vector

```

1 sort(arr.begin(), arr.end());
2 arr.resize(unique(arr.begin(), arr.end()) - arr.begin());

```

6.20. Width

```

1 cout << width(13);
2 cout << 100 << endl; // "      100      "
3 cout.fill('x');
4 cout.width(13);
5 cout << 100 << endl; // "xxxxxx100xxxxxx"
6 cout << right << 100 << endl; "xxxxxxx100"

```

7. Math

7.1. Bell Numbers

```

1 // Number of ways to partition a set.
2 // For example, the set {a, b, c}.
3 // It can be partitioned in five ways: {(a) (b) (c)}, {(a, b), (c)},
4 // {(a, c) (b)}, {(b, c), a}, {(a, b, c)}.
5 //
6 // Time Complexity: O(n * n)
7 int bellNumber(int n) {
8     int bell[n + 1][n + 1];
9     bell[0][0] = 1;
10    for (int i = 1; i <= n; i++) {
11        bell[i][0] = bell[i - 1][i - 1];
12
13        for (int j = 1; j <= i; j++)
14            bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
15    }
16    return bell[n][0];
17 }

```

7.2. Binary Exponentiation

```

1 int bin_pow(const int n, int p) {
2     assert(p >= 0);
3     int ans = 1;
4     int cur_pow = n;

```

```

5
6     while (p) {
7         if (p & 1)
8             ans = (ans * cur_pow) % MOD;
9
10        cur_pow = (cur_pow * cur_pow) % MOD;
11        p >>= 1;
12    }
13
14    return ans;
15 }

```

7.3. Chinese Remainder Theorem

```

1 inline int mod(int x, const int MOD) {
2     x %= MOD;
3     if (x < 0)
4         x += MOD;
5     return x;
6 }
7
8 tuple<int, int, int> extended_gcd(int a, int b) {
9     int x = 0, y = 1, x1 = 1, y1 = 0;
10    while (a != 0) {
11        const int q = b / a;
12        tie(x, x1) = make_pair(x1, x - q * x1);
13        tie(y, y1) = make_pair(y1, y - q * y1);
14        tie(a, b) = make_pair(b % a, a);
15    }
16    return make_tuple(b, x, y);
17 }
18
19 // USE __int128_t if LCM can get close to LLONG_MAX!!!
20 // Returns the smallest number x such that:
21 // x % num[0] = rem[0],
22 // x % num[1] = rem[1],
23 // .....
24 // x % num[n - 1] = rem[n - 1]
25 // It also works when gcd(rem[i], rem[j]) != 1
26 //
27 // Time Complexity: O(n*log(n))
28 int crt(vector<int> &rem, const vector<int> &md) {
29     const int n = rem.size();
30     for (int i = 0; i < n; i++)
31         rem[i] = mod(rem[i], md[i]);
32     int ans = rem.front(), LCM = md.front();
33     for (int i = 1; i < n; i++) {
34         int x, g;
35         tie(g, x, ignore) = extended_gcd(LCM, md[i]);
36         if ((rem[i] - ans) % g != 0)
37             return -1;
38         // the multiplication below may overflow if LCM can get close to
39         // LLONG_MAX
40         // use __int128_t in this case
41         ans =
42             mod(ans + x * (rem[i] - ans) / g % (md[i] / g) * LCM, LCM / g *
43             md[i]);
44         // lcm of LCM, md[i]
45         LCM = LCM / g * md[i];
46     }
47     return ans;
48 }

```


7.4. Combinatorics

```

1 class Combinatorics {
2 private:
3     static constexpr int MOD = 1e9 + 7;
4     const int max_val;
5     vector<int> _inv, _fat;
6
7 private:
8     int mod(int x) {
9         x %= MOD;
10        if (x < 0)
11            x += MOD;
12        return x;
13    }
14
15    static int bin_pow(const int n, int p) {
16        assert(p >= 0);
17        int ans = 1;
18        int cur_pow = n;
19
20        while (p) {
21            if (p & 1ll)
22                ans = (ans * cur_pow) % MOD;
23
24            cur_pow = (cur_pow * cur_pow) % MOD;
25            p >>= 1ll;
26        }
27
28        return ans;
29    }
30
31    vector<int> build_inverse(const int max_val) {
32        vector<int> inv(max_val + 1);
33        inv[1] = 1;
34        for (int i = 2; i <= max_val; ++i)
35            inv[i] = mod(-MOD / i * inv[MOD % i]);
36        return inv;
37    }
38
39    vector<int> build_fat(const int max_val) {
40        vector<int> fat(max_val + 1);
41        fat[0] = 1;
42        for (int i = 1; i <= max_val; ++i)
43            fat[i] = mod(i * fat[i - 1]);
44        return fat;
45    }
46
47 public:
48     /// Builds both factorial and modular inverse array.
49     ///
50     /// Time Complexity: O(max_val)
51     Combinatorics(const int max_val) : max_val(max_val) {
52         assert(0 <= max_val), assert(max_val <= MOD);
53         this->_inv = this->build_inverse(max_val);
54         this->_fat = this->build_fat(max_val);
55     }
56
57     /// Returns the modular inverse of n % MOD.
58     ///
59     /// Time Complexity: O(log(MOD))
60     static int inv_log(const int n) { return bin_pow(n, MOD - 2); }
61
62     /// Returns the modular inverse of n % MOD.
63     ///

```

```

64     /// Time Complexity: O((n <= max_val ? 1 : log(MOD))
65     int inv(const int n) {
66         assert(0 <= n);
67         if (n <= max_val)
68             return this->_inv[n];
69         else
70             return inv_log(n);
71     }
72
73     /// Returns the factorial of n % MOD.
74     int fat(const int n) {
75         assert(0 <= n), assert(n <= max_val);
76         return this->_fat[n];
77     }
78
79     /// Returns C(n, k) % MOD.
80     ///
81     /// Time Complexity: O(1)
82     int choose(const int n, const int k) {
83         assert(0 <= k), assert(k <= n), assert(n <= this->max_val);
84         return mod(fat(n) * mod(inv(fat(k)) * inv(fat(n - k))));
85     }
86 };

```

7.5. Diophantine Equation

```

1 int gcd(int a, int b, int &x, int &y) {
2     if (a == 0) {
3         x = 0;
4         y = 1;
5         return b;
6     }
7     int x1, y1;
8     int d = gcd(b % a, a, x1, y1);
9     x = y1 - (b / a) * x1;
10    y = x1;
11    return d;
12 }
13
14 bool diophantine(int a, int b, int c, int &x0, int &y0, int &g) {
15     g = gcd(abs(a), abs(b), x0, y0);
16     if (c % g)
17         return false;
18
19     x0 *= c / g;
20     y0 *= c / g;
21     if (a < 0)
22         x0 = -x0;
23     if (b < 0)
24         y0 = -y0;
25     return true;
26 }

```

7.6. Divide Fraction

```

1 /// Prints precision floating point places of a / b.
2 string divide(int a, int b, const int precision) {
3     assert(a < b);
4     string ans;
5     for (int i = 0; i < precision; ++i) {
6         a *= 10;
7         ans += a / b + '0';
8         a %= b;

```

```

9      }
10     return ans;
11 }

```

7.7. Divisors

```

1  /// OBS: Each number has at most  $\sqrt[3]{N}$  divisors
2  /// THE NUMBERS ARE NOT SORTED!!!
3  ///
4  /// Time Complexity:  $O(\sqrt{n})$ 
5  vector<int> divisors(int n) {
6      vector<int> ans;
7      for (int i = 1; i * i <= n; i++) {
8          if (n % i == 0) {
9              if (n / i == i)
10                 ans.emplace_back(i);
11             else
12                 ans.emplace_back(i), ans.emplace_back(n / i);
13         }
14     }
15     // sort(ans.begin(), ans.end());
16     return ans;
17 }

```

7.8. Euler Totient

```

1  /// Returns the amount of numbers less than or equal to n which are co-primes
2  /// to it.
3  int phi(int n) {
4      int result = n;
5      for (int i = 2; i * i <= n; i++) {
6          if (n % i == 0) {
7              while (n % i == 0)
8                  n /= i;
9              result -= result / i;
10         }
11     }
12
13     if (n > 1)
14         result -= result / n;
15     return result;
16 }

```

7.9. Extended Euclidean

```

1 // Created by tysm.
2
3 /// Returns a tuple containing the gcd(a, b) and the roots for
4 ///  $a \cdot x + b \cdot y = \text{gcd}(a, b)$ .
5 ///
6 /// Time Complexity:  $O(\log(\min(a, b)))$ .
7 tuple<int, int, int> extended_gcd(int a, int b) {
8     int x = 0, y = 1, x1 = 1, y1 = 0;
9     while (a != 0) {
10         const int q = b / a;
11         tie(x, x1) = make_pair(x1, x - q * x1);
12         tie(y, y1) = make_pair(y1, y - q * y1);
13         tie(a, b) = make_pair(b % a, a);
14     }
15     return make_tuple(b, x, y);
16 }

```

7.10. Factorization

```

1  /// Factorizes a number.
2  ///
3  /// Time Complexity:  $O(\sqrt{n})$ 
4  map<int, int> factorize(int n) {
5      map<int, int> fat;
6      while (n % 2 == 0) {
7          ++fat[2];
8          n /= 2;
9      }
10
11     for (int i = 3; i * i <= n; i += 2) {
12         while (n % i == 0) {
13             ++fat[i];
14             n /= i;
15         }
16         /* OBS1
17            IF (N < 1E7)
18                you can optimize by factoring with SPF
19            */
20     }
21     if (n > 2)
22         ++fat[n];
23     return fat;
24 }

```

7.11. Fft

```

1  // Code copied from:
2  //
3      https://github.com/kth-competitive-programming/kactl/blob/08eb36f4bd9b8ce358e2f3f
4  #define double long double
5  typedef complex<double> C;
6  typedef vector<double> vd;
7  void fft(vector<C> &a) {
8      int n = a.size(), L = 31 - __builtin_clz(n);
9      static vector<complex<double>> R(2, 1);
10     // uncomment if you'll use only 'double'.
11     // static vector<complex<long double>> R(2, 1);
12     static vector<C> rt(2, 1); // (^ 10% faster if double)
13     for (static int k = 2; k < n; k *= 2) {
14         R.resize(n);
15         rt.resize(n);
16         auto x = polar(1.0L, acos(-1.0L) / k);
17         for (int i = k; i < 2 * k; ++i)
18             rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
19     }
20     vi rev(n);
21     for (int i = 0; i < n; ++i)
22         rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
23     for (int i = 0; i < n; ++i)
24         if (i < rev[i])
25             swap(a[i], a[rev[i]]);
26     for (int k = 1; k < n; k *= 2)
27         for (int i = 0; i < n; i += 2 * k)
28             for (int j = 0; j < k; ++j) {
29                 auto x = (double *)&rt[j + k],
30                     y = (double *)&a[i + j + k]; /// exclude-line
31                 C z(x[0] * y[0] - x[1] * y[1],
32                    x[0] * y[1] + x[1] * y[0]); /// exclude-line
33                 a[i + j + k] = a[i + j] - z;
34                 a[i + j] += z;

```

```

35     }
36 }
37
38 /// Polynomial convolution of 'a' and 'b'.
39 ///
40 /// Time Complexity: O(n log n)
41 vector<long long> convolve(const vd &a, const vd &b) {
42     if (a.empty() || b.empty())
43         return {};
44     vd res(a.size() + b.size() - 1);
45     int L = 32 - __builtin_clz(res.size()), n = 1 << L;
46     vector<C> in(n), out(n);
47     copy(all(a), begin(in));
48     for (int i = 0; i < b.size(); ++i)
49         in[i].imag(b[i]);
50     fft(in);
51     for (C &x : in)
52         x *= x;
53     for (int i = 0; i < n; ++i)
54         out[i] = in[-i & (n - 1)] - conj(in[i]);
55     fft(out);
56     for (int i = 0; i < res.size(); ++i)
57         res[i] = imag(out[i]) / (4 * n);
58     vector<long long> arr(res.size());
59     for (int i = 0; i < res.size(); ++i)
60         arr[i] = round(res[i]);
61     return arr;
62 }

```

7.12. Inclusion Exclusion

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{k=1}^n (-1)^{k+1} \left(\sum_{1 \leq i_1 < \dots < i_k \leq n} |A_{i_1} \cap \dots \cap A_{i_k}| \right)$$

7.13. Inclusion Exclusion

```

1 // |A ∪ B ∪ C| = |A| + |B| + |C| - |A ∩ B| - |A ∩ C| - |B ∩ C| + |A ∩ B ∩ C|
2 // EXAMPLE: How many numbers from 1 to 10^9 are multiple of 42, 54, 137 or
3 // 201?
4 int f(const vector<int> &arr, const int LIMIT) {
5     int n = arr.size();
6     int c = 0;
7     for (int mask = 1; mask < (1ll << n); mask++) {
8         int lcm = 1;
9         for (int i = 0; i < n; i++)
10             if (mask & (1ll << i))
11                 lcm = lcm * arr[i] / __gcd(lcm, arr[i]);
12         // if the number of element is odd, then add
13         if (__builtin_popcount_ll(mask) % 2 == 1)
14             c += LIMIT / lcm;
15         else // otherwise subtract
16             c -= LIMIT / lcm;
17     }
18     return LIMIT - c;
19 }

```

```

20 }

```

7.14. Karatsuba

```

1 /// Code copied from:
2 ///
3     https://github.com/iam0rchld/algospot/blob/98476cf0513967cd2481d8dc8dc02015984209
4 const int MINIMUM_KARATSUBA_A_SIZE = 50;
5
6 vector<int> multiply(const vector<int> &a, const vector<int> &b) {
7     vector<int> multiplication(a.size() + b.size() + 1, 0);
8     for (int i = 0; i < a.size(); i++)
9         for (int j = 0; j < b.size(); j++)
10             multiplication[i + j] += a[i] * b[j];
11     return multiplication;
12 }
13
14 void add(vector<int> &to, vector<int> &howMuch, int howMuchExponent) {
15     const int howMuchSize = howMuch.size();
16
17     if (to.size() < howMuch.size() + howMuchExponent)
18         to.resize(howMuch.size() + howMuchExponent);
19
20     for (int i = 0; i < howMuchSize; i++)
21         to[howMuchExponent + i] += howMuch[i];
22 }
23
24 void subtract(vector<int> &from, vector<int> &howMuch) {
25     for (int i = 0; i < howMuch.size(); i++)
26         from[i] -= howMuch[i];
27 }
28
29 /// Multiplies two polynomials a and b using Karatsuba algorithm.
30 ///
31 /// Time complexity: O(n^(1.59))
32 vector<int> multiplyKaratsuba(const vector<int> &a, const vector<int> &b) {
33     const int aSize = a.size(), bSize = b.size();
34
35     if (aSize < bSize)
36         return multiplyKaratsuba(b, a);
37
38     if (aSize == 0 || bSize == 0)
39         return vector<int>();
40
41     if (aSize < MINIMUM_KARATSUBA_A_SIZE)
42         return multiply(a, b);
43
44     const int aNumberHalfSize = aSize / 2;
45     vector<int> aDivision0(a.begin(), a.begin() + aNumberHalfSize);
46     vector<int> aDivision1(a.begin() + aNumberHalfSize, a.end());
47     vector<int> bDivision0(b.begin(),
48                             b.begin() + min<int>(bSize, aNumberHalfSize));
49     vector<int> bDivision1(b.begin() + min<int>(bSize, aNumberHalfSize),
50                             b.end());
51     vector<int> karatsubaFactor0 = multiplyKaratsuba(aDivision0, bDivision0);
52     vector<int> karatsubaFactor2 = multiplyKaratsuba(aDivision1, bDivision1);
53
54     add(aDivision0, aDivision1, 0);
55     add(bDivision0, bDivision1, 0);
56
57     vector<int> karatsubaFactor1 = multiplyKaratsuba(aDivision0, bDivision0);
58     subtract(karatsubaFactor1, karatsubaFactor0);
59     subtract(karatsubaFactor1, karatsubaFactor2);

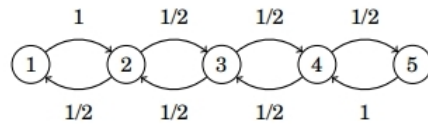
```

```

59 vector<int> multiplication;
60 add(multiplication, karatsubaFactor0, 0);
61 add(multiplication, karatsubaFactor1, aNumberHalfSize);
62 add(multiplication, karatsubaFactor2, aNumberHalfSize + aNumberHalfSize);
63
64 return multiplication;
65 }

```

7.15. Markov Chains



$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Probably after moving 1 step from 1

7.16. Matrix Exponentiation

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k)$$

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}$$

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \dots & c_1 \end{bmatrix}$$

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}$$

Fibonacci

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

7.17. Matrix Exponentiation

```

1 // USE #define int long long!!!!
2 // Remember to MOD the numbers before putting them into the matrix !!!
3 struct Matrix {
4     static constexpr int MOD = 1e9 + 7;
5
6     // static matrix, if it's created multiple times, it's recommended
7     // to avoid TLE.
8     static constexpr int MAXN = 4, MAXM = 4;
9     array<array<int, MAXM>, MAXN> mat = {};
10    int n, m;
11    Matrix(const int n, const int m) : n(n), m(m) {}
12
13    static int mod(int n) {
14        n %= MOD;
15        if (n < 0)
16            n += MOD;
17        return n;
18    }
19
20    /// Creates a n x n identity matrix.
21    ///
22    /// Time Complexity: O(n*n)
23    Matrix identity() {
24        assert(n == m);
25        Matrix mat_identity(n, m);
26        for (int i = 0; i < n; ++i)
27            mat_identity.mat[i][i] = 1;
28        return mat_identity;
29    }
30
31    /// Multiplies matrices mat and other.
32    ///
33    /// Time Complexity: O(mat.size() ^ 3)
34    Matrix operator*(const Matrix &other) const {
35        assert(m == other.n);
36        Matrix ans(n, other.m);
37        for (int i = 0; i < n; ++i)
38            for (int j = 0; j < m; ++j)
39                for (int k = 0; k < m; ++k)
40                    ans.mat[i][j] = mod(ans.mat[i][j] + mat[i][k] * other.mat[k][j]);
41        return ans;
42    }
43
44    /// Exponents the matrix mat to the power of p.
45    ///
46    /// Time Complexity: O((mat.size() ^ 3) * log2(p))
47    Matrix pow(int p) {
48        assert(p >= 0);
49        Matrix ans = identity(), cur_power(n, m);
50        cur_power.mat = mat;
51        while (p) {
52            if (p & 1)
53                ans = ans * cur_power;
54            cur_power = cur_power * cur_power;
55            p >>= 1;
56        }
57        return ans;
58    }
59 }
60

```

7.18. Pollard Rho (Factorize)

```

1  /// Copied from:
2  /// https://codeforces.com/contest/1305/submission/73826085
3  #include <bits/stdc++.h>
4  using namespace std;
5  #define rep(i, from, to) for (int i = from; i < (to); ++i)
6  #define trav(a, x) for (auto &a : x)
7  #define all(x) x.begin(), x.end()
8  #define sz(x) (int)(x).size()
9  typedef long long ll;
10 typedef pair<int, int> pii;
11 typedef vector<int> vi;
12
13 typedef long long ll;
14 typedef unsigned long long ull;
15 typedef long double ld;
16
17 ull gcd(ull u, ull v) {
18     if (u == 0 || v == 0)
19         return v ^ u;
20     int shift = __builtin_ctzll(u | v);
21     u >>= __builtin_ctzll(u);
22     v >>= __builtin_ctzll(v);
23     while (u > v) {
24         ull t = v;
25         v = u;
26         u = t;
27     }
28     v -= u;
29     while (v);
30     return u << shift;
31 }
32
33 ull mod_mul(ull a, ull b, ull M) {
34     ll ret = a * b - M * ull(1 / (double)M * a * b);
35     return ret + M * (ret < 0) - M * (ret >= (ll)M);
36 }
37
38 ull mod_pow(ull b, ull e, ull mod) {
39     ull ans = 1;
40     for (; e; b = mod_mul(b, b, mod), e /= 2)
41         if (e & 1)
42             ans = mod_mul(ans, b, mod);
43     return ans;
44 }
45
46 bool isPrime(ull n) {
47     if (n < 2 || n % 6 % 4 != 1)
48         return (n | 1) == 3;
49     ull A[] = {2, 13, 23, 1662803}, s = __builtin_ctzll(n - 1), d = n >> s;
50     for (auto a : A) { // ^ count trailing zeroes
51         ull p = mod_pow(a % n, d, n), i = s;
52         while (p != 1 && p != n - 1 && a % n && i--)
53             p = mod_mul(p, p, n);
54         if (p != n - 1 && i != s)
55             return 0;
56     }
57     return 1;
58 }
59
60 typedef ull u64;
61 typedef unsigned int u32;
62
63 typedef __uint128_t ul28;
64 // typedef __int128_t il28;
65
66 typedef long long i64;
67 typedef unsigned long long u64;
68
69 u64 hi(ul28 x) { return (x >> 64); }
70 u64 lo(ul28 x) { return (x << 64) >> 64; }
71 struct Mont {
72     Mont(u64 n) : mod(n) {
73         inv = n;
74         rep(i, 0, 6) inv *= 2 - n * inv;
75         r2 = -n % n;
76         rep(i, 0, 4) if ((r2 <= 1) >= mod) r2 -= mod;
77         rep(i, 0, 5) r2 = mul(r2, r2);
78     }
79     u64 reduce(ul28 x) const {
80         u64 y = hi(x) - hi(ul28(lo(x) * inv) * mod);
81         return i64(y) < 0 ? y + mod : y;
82     }
83     u64 reduce(u64 x) const { return reduce(x); }
84     u64 init(u64 n) const { return reduce(ul28(n) * r2); }
85     u64 mul(u64 a, u64 b) const { return reduce(ul28(a) * b); }
86     u64 mod, inv, r2;
87 };
88 ull pollard(ull n) {
89     if (n == 9)
90         return 3;
91     if (n == 25)
92         return 5;
93     if (n == 49)
94         return 7;
95     if (n == 323)
96         return 17;
97     Mont mont(n);
98     auto f = [n, &mont](ull x) { return mont.mul(x, x) + 1; };
99     ull x = 0, y = 0, t = 0, prd = 2, i = 1, q;
100    while (t++ % 32 || gcd(prd, n) == 1) {
101        if (x == y)
102            x = ++i, y = f(x);
103        if ((q = mont.mul(prd, max(x, y) - min(x, y)))
104            prd = q;
105        x = f(x), y = f(f(y));
106    }
107    return gcd(prd, n);
108 }
109
110 unordered_set<ll> primes;
111 unordered_set<ll> seen;
112 set<ll> prm;
113 void factor(ull n) {
114     if (n <= 1 || seen.count(n))
115         return;
116     seen.insert(n);
117     if (isPrime(n)) {
118         primes.insert(n);
119         prm.insert(n);
120     } else {
121         ull x = pollard(n);
122         factor(x), factor(n / x);
123     }
124 }
125 signed main() {
126     ull x;
127     // Factorizes 3e4 numbers in less than 1 sec in my PC.
128     for (int i = 0; i < 30000; i++) {
129         prm.clear();

```

```

130     seen.clear();
131     cin >> x;
132     factor(x);
133     // for (ll y : prm) {
134     //     cout << y << " ";
135     //     while (x % y == 0)
136     //         x /= y;
137     // }
138     // cout << endl;
139     // assert(x == 1);
140 }
141 cout << endl;
142 }

```

7.19. Pollard Rho (Find A Divisor)

```

1 // Requires binary_exponentiation.cpp
2
3 /// Returns a prime divisor for n.
4 ///
5 /// Expected Time Complexity: O(n1/4)
6 int pollard_rho(const int n) {
7     srand(time(NULL));
8
9     /* no prime divisor for 1 */
10    if (n == 1)
11        return n;
12
13    if (n % 2 == 0)
14        return 2;
15
16    /* we will pick from the range [2, N) */
17    int x = (rand() % (n - 2)) + 2;
18    int y = x;
19
20    /* the constant in f(x).
21     * Algorithm can be re-run with a different c
22     * if it throws failure for a composite. */
23    int c = (rand() % (n - 1)) + 1;
24
25    /* Initialize candidate divisor (or result) */
26    int d = 1;
27
28    /* until the prime factor isn't obtained.
29     * If n is prime, return n */
30    while (d == 1) {
31        /* Tortoise Move: x(i+1) = f(x(i)) */
32        x = (modular_pow(x, 2, n) + c + n) % n;
33
34        /* Hare Move: y(i+1) = f(f(y(i))) */
35        y = (modular_pow(y, 2, n) + c + n) % n;
36        y = (modular_pow(y, 2, n) + c + n) % n;
37
38        d = __gcd(abs(x - y), n);
39
40        /* retry if the algorithm fails to find prime factor
41         * with chosen x and c */
42        if (d == n)
43            return pollard_rho(n);
44    }
45
46    return d;
47 }

```

7.20. Polynomial Convolution

```

1 /// Returns the resulting polynomial after convolution of polynomials a and
  b.
2 ///
3 /// Time Complexity: O(a.size() * b.size())
4 vector<int> convolution(const vector<int> &a, const vector<int> &b) {
5     const int n = a.size(), m = b.size();
6     vector<int> ans(n + m - 1);
7     for (int i = 0; i < n; ++i)
8         for (int j = 0; j < m; ++j)
9             ans[i + j] += a[i] * b[j];
10    return ans;
11 }

```

7.21. Primality Check

```

1 bool is_prime(int n) {
2     if (n <= 1)
3         return false;
4     if (n <= 3)
5         return true;
6     // This is checked so that we can skip
7     // middle five numbers in below loop
8     if (n % 2 == 0 || n % 3 == 0)
9         return false;
10    for (int i = 5; i * i <= n; i += 6)
11        if (n % i == 0 || n % (i + 2) == 0)
12            return false;
13    return true;
14 }

```

7.22. Primes

```

1 0 -> 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
   71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
   149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
   227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293,
   307, 311, 313, 317, 331, 337, 347, 349, 353
2 1e5 -> 100003, 100019, 100043, 100049, 100057, 100069, 100103, 100109,
   100129, 100151
3 2e5 -> 200003, 200009, 200017, 200023, 200029, 200033, 200041, 200063,
   200087, 200117
4 1e6 -> 1000003, 1000033, 1000037, 1000039, 1000081, 1000099, 1000117,
   1000121, 1000133, 1000151
5 2e6 -> 2000003, 2000029, 2000039, 2000081, 2000083, 2000093, 2000107,
   2000113, 2000143, 2000147
6 1e9 -> 1000000007, 1000000009, 1000000021, 1000000033, 1000000087,
   1000000093, 1000000097, 1000000103, 1000000123, 1000000181, 1000000207,
   1000000223, 1000000241
7 2e9 -> 2000000011, 2000000033, 2000000063, 2000000087, 2000000089,
   2000000099, 2000000137, 2000000141, 2000000143, 2000000153

```

7.23. Sieve + Segmented Sieve

```

1 const int MAXN = 1e6;
2
3 /// Contains all the primes in the segments
4 vector<int> segPrimes;
5 bitset<MAXN + 5> primesInSeg;
6
7 /// smallest prime factor

```

```

8 vector<int> spf(MAXN + 5);
9
10 vector<int> primes;
11 bitset<MAXN + 5> isPrime;
12
13 void sieve(int n = MAXN + 2) {
14     iota(spf.begin(), spf.end(), 0ll);
15     isPrime.set();
16     for (int64_t i = 2; i <= n; i++) {
17         if (isPrime[i]) {
18             for (int64_t j = i * i; j <= n; j += i) {
19                 isPrime[j] = false;
20                 spf[j] = min(i, int64_t(spf[j]));
21             }
22             primes.emplace_back(i);
23         }
24     }
25 }
26
27 vector<int> getFactorization(int x) {
28     vector<int> ret;
29     while (x != 1) {
30         ret.emplace_back(spf[x]);
31         x = x / spf[x];
32     }
33     return ret;
34 }
35
36 /// Gets all primes from l to r
37 void segSieve(int l, int r) {
38     // primes from l to r
39     // transferred to 0..(l-r)
40     segPrimes.clear();
41     primesInSeg.set();
42     int sq = sqrt(r) + 5;
43
44     for (int p : primes) {
45         if (p > sq)
46             break;
47
48         for (int i = l - l % p; i <= r; i += p) {
49             if (i - l < 0)
50                 continue;
51
52             // if i is less than 1e6, it could be checked in the
53             // array of the sieve
54             if (i >= (int)1e6 || !isPrime[i])
55                 primesInSeg[i - l] = false;
56         }
57     }
58
59     for (int i = 0; i < r - l + 1; i++) {
60         if (primesInSeg[i])
61             segPrimes.emplace_back(i + l);
62     }
63 }

```

7.24. Stars And Bars

I. positive integers x_i

For any pair of positive integers n and k , the number of distinct k -tuples of **positive integers** whose sum is n is given by the binomial coefficient

$$\binom{n-1}{k-1}.$$

In your case, $k = 4$, $n = 22$. So the number of distinct solutions (x_1, x_2, x_3, x_4) where the $x_i \in \mathbb{Z}$, $x_i > 0$ is given by

$$\binom{22-1}{4-1} = \binom{21}{3} = \frac{21!}{3!18!} = 1330$$

II. non-negative integers x_i

For any pair of natural numbers n and k , the number of distinct k -tuples of **non-negative integers** (which includes the possibility that one or more of the x_i are zero) whose sum is n is given by the binomial coefficient

$$\binom{n+k-1}{n} = \binom{n+k-1}{k-1}.$$

In your problem, $k = 4$, $n = 22$. Here, the distinct solutions (x_1, x_2, x_3, x_4) will include those from I ., but also allows 4-tuples in which one or more of the x_i are zero: $x_i \in \mathbb{Z}$, $x_i \geq 0$.

$$\binom{22+4-1}{22} = \binom{25}{22} = \frac{25!}{22!3!} = 2300$$

8. Miscellaneous

8.1. 2-Sat

```

1 // OBS: INDEXED FROM 0
2 // USE POS_X = 1 FOR POSITIVE CLAUSES AND 0 FOR NEGATIVE. OTHERWISE THE FINAL
3 // ANSWER ARRAY WILL BE FLIPPED.
4 class SAT {
5 private:
6     vector<vector<int>>> adj;
7     int n;
8
9 public:
10     SAT(const int n) : n(n) {
11         adj.resize(2 * n);
12         ans.resize(n);
13     }
14
15     // (X v Y) = (~X -> Y) & (~Y -> X)
16     void add_or(const int x, const bool pos_x, const int y, const bool pos_y) {
17         assert(0 <= x), assert(x < n), assert(0 <= y), assert(y < n);
18         adj[(x << 1) ^ (pos_x ^ 1)].emplace_back((y << 1) ^ pos_y);
19         adj[(y << 1) ^ (pos_y ^ 1)].emplace_back((x << 1) ^ pos_x);
20     }
21
22     // (X xor Y) = (X v Y) & (~X v ~Y)

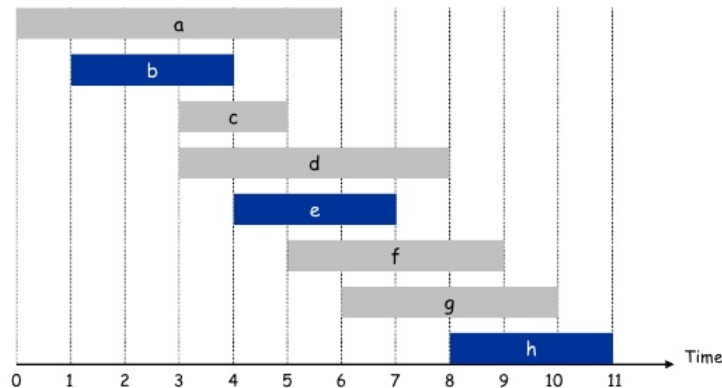
```

```

23 // for this operation the result is always 0 1 or 1 0
24 void add_xor(const int x, const bool pos_x, const int y, const bool pos_y)
    {
25     assert(0 <= x), assert(x < n), assert(0 <= y), assert(y < n);
26     add_or(x, pos_x, y, pos_y);
27     add_or(x, pos_x ^ 1, y, pos_y ^ 1);
28 }
29
30 vector<bool> ans;
31 /// Checks whether the system is feasible or not. If it's feasible, it
    stores
32 /// a satisfiable answer in the array 'ans'.
33 ///
34 /// Time Complexity: O(n)
35 bool check() {
36     SCC scc(2 * n, 0, adj);
37     for (int i = 0; i < n; i++) {
38         if (scc.comp[(i << 1) | 1] == scc.comp[(i << 1) | 0])
39             return false;
40         ans[i] = (scc.comp[(i << 1) | 1] > scc.comp[(i << 1) | 0]);
41     }
42     return true;
43 }
44 };

```

8.2. Interval Scheduling



8.3. Interval Scheduling

```

1 1 -> Ordena pelo final do evento, depois pelo inicio.
2 2 -> Vai iterando pelos eventos, se eles não tiverem horário em comum então
    adiciona o evento à lista.

```

8.4. Sliding Window Minimum

```

1 // mínimo num vetor arr de arr[0] ... arr[k-1], arr[1] ... arr[k], arr[2]
  ... arr[k+1]
2
3 void swma(vector<int> arr, int k) {

```

```

4 deque<ii> window;
5 for(int i = 0; i < arr.size(); i++) {
6     while(!window.empty() && window.back().ff > arr[i])
7         window.pop_back();
8     window.pb(ii(arr[i], i));
9     while(window.front().ss <= i - k)
10         window.pop_front();
11
12     if(i >= k)
13         cout << ' ';
14     if(i - k + 1 >= 0)
15         cout << window.front().ff;
16 }
17 }

```

8.5. Torre De Hanoi

```

1 #include <stdio.h>
2
3 // C recursive function to solve tower of hanoi puzzle
4 void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
5     if (n == 1) {
6         printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
7         return;
8     }
9     towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
10    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
11    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
12 }
13
14 int main() {
15     int n = 4; // Number of disks
16     towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
17     return 0;
18 }

```

8.6. Counting Frequency Of Digits From 1 To K

```

1 def check(k):
2     ans = [0] * 10
3     for d in range(1, 10):
4         pot = 10
5         last = 1
6         for i in range(20):
7             v = (k // pot * last) + min(max(0, ((k % pot) - (last * d)) + 1), last)
8             ans[d] += v
9             pot *= 10
10            last *= 10
11
12     return ans

```

8.7. Counting Number Of Digits Up To N

```

1 int solve(int n) {
2     int maxx = 9, minn = 1, dig = 1, ret = 0;
3     for (int i = 1; i <= 17; i++) {
4         int q = min(maxx, n);
5         ret += max(0ll, (q - minn + 1) * dig);
6         maxx = (maxx * 10 + 9), minn *= 10, dig++;
7     }
8     return ret;
9 }

```


8.8. Infix To Postfix

```

1  /// Infix Expression | Prefix Expression | Postfix Expression
2  ///      A + B      |      + A B      |      A B +
3  ///      A + B * C   |      + A * B C   |      A B C * +
4  /// Time Complexity: O(n)
5  int infix_to_postfix(const string &infix) {
6      map<char, int> prec;
7      stack<char> op;
8      string postfix;
9
10     prec['+'] = prec['-'] = 1;
11     prec['*'] = prec['/'] = 2;
12     prec['^'] = 3;
13     for (int i = 0; i < infix.size(); ++i) {
14         char c = infix[i];
15         if (is_digit(c)) {
16             while (i < infix.size() && isdigit(infix[i])) {
17                 postfix += infix[i];
18                 ++i;
19             }
20             --i;
21         } else if (isalpha(c))
22             postfix += c;
23         else if (c == '(')
24             op.push('(');
25         else if (c == ')') {
26             while (!op.empty() && op.top() != '(') {
27                 postfix += op.top();
28                 op.pop();
29             }
30             op.pop();
31         } else {
32             while (!op.empty() && prec[op.top()] >= prec[c]) {
33                 postfix += op.top();
34                 op.pop();
35             }
36             op.push(c);
37         }
38     }
39     while (!op.empty()) {
40         postfix += op.top();
41         op.pop();
42     }
43     return postfix;
44 }

```

8.9. Iterate Over Subsets Of Mask

```

1  for (int j = mask; j > 0; j = (j - 1) & mask) {
2  }

```

8.10. Kadane

```

1  /// Returns the maximum contiguous sum in the array.
2  ///
3  /// Time Complexity: O(n)
4  int kadane(vector<int> &arr) {
5      if (arr.empty())
6          return 0;
7      int sum, tot;
8      sum = tot = arr[0];
9

```

```

10     for (int i = 1; i < arr.size(); i++) {
11         sum = max(arr[i], arr[i] + sum);
12         if (sum > tot)
13             tot = sum;
14     }
15     return tot;
16 }

```

8.11. Kadane (Segment Tree)

```

1  struct Seg_Tree {
2      struct Node {
3          int pref, suf, tot, best;
4          Node() {}
5          Node(int pref, int suf, int tot, int best)
6              : pref(pref), suf(suf), tot(tot), best(best) {}
7      };
8
9      int n;
10     vector<Node> tree;
11     vi arr;
12
13     Seg_Tree(vi &arr) : n(arr.size()), arr(arr) {
14         tree.resize(4 * n);
15         build(0, n - 1, 0);
16     }
17
18     Node query(const int l, const int r, const int i, const int j,
19               const int pos) {
20         if (l > r || l > j || r < i)
21             return Node(-INF, -INF, -INF, -INF);
22
23         if (i <= l && r <= j)
24             return Node(tree[pos].pref, tree[pos].suf, tree[pos].tot,
25                           tree[pos].best);
26
27         int mid = (l + r) / 2;
28         Node left = query(l, mid, i, j, 2 * pos + 1);
29         Node right = query(mid + 1, r, i, j, 2 * pos + 2);
30         Node x;
31         x.pref = max({left.pref, left.tot, left.tot + right.pref});
32         x.suf = max({right.suf, right.tot, right.tot + left.suf});
33         x.tot = left.tot + right.tot;
34         x.best = max({left.best, right.best, left.suf + right.pref});
35         return x;
36     }
37
38     // Update arr[idx] to v
39     // ITS NOT DELTA!!!
40     void update(int l, int r, const int idx, const int v, const int pos) {
41         if (l > r || l > idx || r < idx)
42             return;
43
44         if (l == idx && r == idx) {
45             tree[pos] = Node(v, v, v, v);
46             return;
47         }
48
49         int mid = (l + r) / 2;
50         update(l, mid, idx, v, 2 * pos + 1);
51         update(mid + 1, r, idx, v, 2 * pos + 2);
52         l = 2 * pos + 1, r = 2 * pos + 2;
53         tree[pos].pref =
54             max({tree[l].pref, tree[l].tot, tree[l].tot + tree[r].pref});

```

```

54     tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot +
55     tree[l].suf});
56     tree[pos].tot = tree[l].tot + tree[r].tot;
57     tree[pos].best =
58     max({tree[l].best, tree[r].best, tree[l].suf + tree[r].pref});
59 }
60 void build(int l, int r, const int pos) {
61     if (l == r) {
62         tree[pos] = Node(arr[l], arr[l], arr[l], arr[l]);
63         return;
64     }
65
66     int mid = (l + r) / 2;
67     build(l, mid, 2 * pos + 1);
68     build(mid + 1, r, 2 * pos + 2);
69     l = 2 * pos + 1, r = 2 * pos + 2;
70     tree[pos].pref =
71     max({tree[l].pref, tree[l].tot, tree[l].tot + tree[r].pref});
72     tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot +
73     tree[l].suf});
74     tree[pos].tot = tree[l].tot + tree[r].tot;
75     tree[pos].best =
76     max({tree[l].best, tree[r].best, tree[l].suf + tree[r].pref});
77 };

```

8.12. Kadane 2D

```

1 // Program to find maximum sum subarray in a given 2D array
2 #include <stdio.h>
3 #include <string.h>
4 #include <limits.h>
5 int mat[1001][1001]
6 int ROW = 1000, COL = 1000;
7
8
9 // Implementation of Kadane's algorithm for 1D array. The function
10 // returns the maximum sum and stores starting and ending indexes of the
11 // maximum sum subarray at addresses pointed by start and finish pointers
12 // respectively.
13 int kadane(int* arr, int* start, int* finish, int n) {
14     // initialize sum, maxSum and
15     int sum = 0, maxSum = INT_MIN, i;
16
17     // Just some initial value to check for all negative values case
18     *finish = -1;
19
20     // local variable
21     int local_start = 0;
22
23     for (i = 0; i < n; ++i) {
24         sum += arr[i];
25         if (sum < 0) {
26             sum = 0;
27             local_start = i+1;
28         }
29         else if (sum > maxSum) {
30             maxSum = sum;
31             *start = local_start;
32             *finish = i;
33         }
34     }
35 }

```

```

36 // There is at-least one non-negative number
37 if (*finish != -1)
38     return maxSum;
39
40 // Special Case: When all numbers in arr[] are negative
41 maxSum = arr[0];
42 *start = *finish = 0;
43
44 // Find the maximum element in array
45 for (i = 1; i < n; i++) {
46     if (arr[i] > maxSum) {
47         maxSum = arr[i];
48         *start = *finish = i;
49     }
50 }
51 return maxSum;
52 }
53
54 // The main function that finds maximum sum rectangle in mat[][]
55 int findMaxSum() {
56     // Variables to store the final output
57     int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;
58
59     int left, right, i;
60     int temp[ROW], sum, start, finish;
61
62     // Set the left column
63     for (left = 0; left < COL; ++left) {
64         // Initialize all elements of temp as 0
65         for (int i = 0; i < ROW; i++)
66             temp[i] = 0;
67
68         // Set the right column for the left column set by outer loop
69         for (right = left; right < COL; ++right) {
70             // Calculate sum between current left and right for every row 'i'
71             for (i = 0; i < ROW; ++i)
72                 temp[i] += mat[i][right];
73
74             // Find the maximum sum subarray in temp[]. The kadane()
75             // function also sets values of start and finish. So 'sum' is
76             // sum of rectangle between (start, left) and (finish, right)
77             // which is the maximum sum with boundary columns strictly as
78             // left and right.
79             sum = kadane(temp, &start, &finish, ROW);
80
81             // Compare sum with maximum sum so far. If sum is more, then
82             // update maxSum and other output values
83             if (sum > maxSum) {
84                 maxSum = sum;
85                 finalLeft = left;
86                 finalRight = right;
87                 finalTop = start;
88                 finalBottom = finish;
89             }
90         }
91     }
92
93     return maxSum;
94     // Print final values
95     printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
96     printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
97     printf("Max sum is: %d\n", maxSum);
98 }

```

8.13. Largest Area In Histogram

```

1  /// Time Complexity: O(n)
2  int largest_area_in_histogram(vector<int> &arr) {
3      arr.emplace_back(0);
4
5      stack<int> s;
6      int ans = 0;
7      for (int i = 0; i < arr.size(); ++i) {
8          while (!s.empty() && arr[s.top()] >= arr[i]) {
9              int height = arr[s.top()];
10             s.pop();
11             int l = (s.empty() ? 0 : s.top() + 1);
12             // creates a rectangle from l to i - 1
13             ans = max(ans, height * (i - l));
14         }
15         s.emplace(i);
16     }
17     return ans;
18 }

```

8.14. Modular Integer

```

1  // Created by tysm.
2
3  /// Returns a tuple containing the gcd(a, b) and the roots for
4  /// a*x + b*y = gcd(a, b).
5  ///
6  /// Time Complexity: O(log(min(a, b))).
7  tuple<uint, int, int> extended_gcd(uint a, uint b) {
8      int x = 0, y = 1, x1 = 1, y1 = 0;
9      while (a != 0) {
10         uint q = b / a;
11         tie(x, x1) = make_pair(x1, x - q * x1);
12         tie(y, y1) = make_pair(y1, y - q * y1);
13         tie(a, b) = make_pair(b % a, a);
14     }
15     return make_tuple(b, x, y);
16 }
17
18 /// Provides modular operations such as +, -, *, /, multiplicative inverse
19 /// and
20 /// binary exponentiation.
21 ///
22 /// Time Complexity: O(1).
23 template <uint M> struct modular {
24     static_assert(0 < M && M <= INT_MAX, "M must be a positive 32 bits
25     integer.");
26
27     uint value;
28
29     modular() : value(0) {}
30
31     template <typename T> modular(const T value) {
32         if (value >= 0)
33             this->value = ((uint)value < M ? value : (uint)value % M);
34         else {
35             uint abs_value = -(uint)value % M;
36             this->value = (abs_value == 0 ? 0 : M - abs_value);
37         }
38     }
39
40     template <typename T> explicit operator T() const { return value; }

```

```

40 modular operator-() const { return modular(value == 0 ? 0 : M - value); }
41
42 modular &operator+=(const modular &rhs) {
43     if (rhs.value >= M - value)
44         value = rhs.value - (M - value);
45     else
46         value += rhs.value;
47     return *this;
48 }
49
50 modular &operator-=(const modular &rhs) {
51     if (rhs.value > value)
52         value = M - (rhs.value - value);
53     else
54         value -= rhs.value;
55     return *this;
56 }
57
58 modular &operator*=(const modular &rhs) {
59     value = (uint64_t)value * rhs.value % M;
60     return *this;
61 }
62
63 modular &operator/=(const modular &rhs) { return *this *= inverse(rhs); }
64
65 /// Computes pow(b, e) % M.
66 ///
67 /// Time Complexity: O(log(e)).
68 friend modular exp(modular b, uint e) {
69     modular res = 1;
70     for (; e > 0; e >>= 1) {
71         if (e & 1)
72             res *= b;
73         b *= b;
74     }
75     return res;
76 }
77
78 /// Computes the modular multiplicative inverse of a with mod M.
79 ///
80 /// Time Complexity: O(log(a)).
81 friend modular inverse(const modular &a) {
82     assert(a.value > 0);
83     auto aux = extended_gcd(a.value, M);
84     assert(get<0>(aux) == 1); // a and M must be coprimes.
85     return modular(get<1>(aux));
86 }
87
88 friend modular operator+(modular lhs, const modular &rhs) {
89     return lhs += rhs;
90 }
91
92 friend modular operator-(modular lhs, const modular &rhs) {
93     return lhs -= rhs;
94 }
95
96 friend modular operator*(modular lhs, const modular &rhs) {
97     return lhs *= rhs;
98 }
99
100 friend modular operator/(modular lhs, const modular &rhs) {
101     return lhs /= rhs;
102 }
103
104 friend bool operator==(const modular &lhs, const modular &rhs) {

```

```

105     return lhs.value == rhs.value;
106 }
107
108 friend bool operator!=(const modular &lhs, const modular &rhs) {
109     return !(lhs == rhs);
110 }
111
112 friend string to_string(const modular &a) { return to_string(a.value); }
113
114 friend ostream &operator<<(ostream &lhs, const modular &rhs) {
115     return lhs << to_string(rhs);
116 }
117 };
118
119 using mint = modular<MOD>;

```

8.15. Point Compression

```

1 // map<int, int> rev;
2
3 /// Compress points in the array arr to the range [0..n-1].
4 ///
5 /// Time Complexity: O(n log n)
6 vector<int> compress(vector<int> &arr) {
7     vector<int> aux = arr;
8     sort(aux.begin(), aux.end());
9     aux.resize(unique(aux.begin(), aux.end()) - aux.begin());
10
11     for (size_t i = 0; i < arr.size(); i++) {
12         int id = lower_bound(aux.begin(), aux.end(), arr[i]) - aux.begin();
13         // rev[id] = arr[i];
14         arr[i] = id;
15     }
16     return arr;
17 }

```

8.16. Ternary Search

```

1 /// Returns the index in the array which contains the minimum element. In
2 case
3 /// of draw, it returns the first occurrence. The array should, first,
4 decrease,
5 /// then increase.
6 ///
7 /// Time Complexity: O(log3(n))
8 int ternary_search(const vector<int> &arr) {
9     int l = 0, r = (int)arr.size() - 1;
10     while (r - l > 2) {
11         int lc = l + (r - l) / 3;
12         int rc = r - (r - l) / 3;
13         // the function f(x) returns the element on the position x
14         if (f(lc) > f(rc))
15             // the function is going down, then the middle is on the right.
16             l = rc;
17         else
18             r = lc;
19     }
20     // the range [l, r] contains the minimum element.
21
22     int minn = f(l), idx = l;
23     for (int i = l + 1; i <= r; ++i)
24         if (f(i) < minn) {
25             idx = i;
26         }
27 }

```

```

24     minn = f(i);
25 }
26
27 return idx;
28 }

```

9. Stress Testing

9.1. Check

```

1 #!/bin/bash
2
3 # Tests infinite inputs generated by gen.
4 # It compares the output of a.cpp and brute.cpp and
5 # stops if there's any difference.
6
7 g++ -std=c++17 gen.cpp -o gen
8 g++ -std=c++17 a.cpp -o a
9 g++ -std=c++17 brute.cpp -o brute
10
11 for((i=1;;i++)); do
12     echo $i
13     ./gen $i > in
14     time ./a < in > o1
15     ./brute < in > o2
16     diff <./a < in> <./brute < in> || break
17 done
18
19 cat in
20 echo 'mine'
21 cat o1
22 echo 'not mine'
23 cat o2
24 #sed -i 's/\r$//' filename ----- remover \r do txt

```

9.2. Gen

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define eb emplace_back
6 #define ii pair<int, int>
7 #define OK (cerr << "OK" << endl)
8 #define debug(x) cerr << #x " = " << (x) << endl
9 #define ff first
10 #define ss second
11 #define int long long
12 #define tt tuple<int, int, int>
13 #define all(x) x.begin(), x.end()
14 #define vi vector<int>
15 #define vii vector<pair<int, int>>
16 #define vvi vector<vector<int>>
17 #define vvii vector<vector<pair<int, int>>>
18 #define Matrix(n, m, v) vector<vector<int>>(n, vector<int>(m, v))
19 #define endl '\n'
20
21 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
22
23 // Generates a string of (n) characters from 'a' to 'a' + (c)
24 string str(const int n, const int c);
25 // Generates (size) strings of (n) characters from 'a' to 'a' + (c)
26 string spaced_str(const int n, const int size, const int c);

```

```

27 // Generates a string of (n) 01 characters.
28 string str01(const int n);
29 // Generates a number in the range [l, r].
30 int num(const int l, const int r);
31 // Generates a vector of (n) numbers in the range [l, r].
32 vector<int> vec(const int n, const int l, const int r);
33 // Generates a matrix of (n x m) numbers in the range [l, r].
34 vector<vector<int>> matrix(const int n, const int m, const int l, const int
    r);
35 // Generates a tree with n vertices
36 vector<pair<int, int>> tree(const int n);
37 // Generates a forest with n vertices.
38 vector<pair<int, int>> forest(const int n);
39 // Generates a connected graph with n vertices.
40 vector<pair<int, int>> connected_graph(const int n);
41 // Generates a graph with n vertices.
42 vector<pair<int, int>> graph(const int n);
43
44 signed main() {
45     int t = num(1, 1);
46     // cout << t << endl;
47     while (t--) {
48         int n = num(1, 2e5);
49         int m = num(1, 2e5);
50         cout << n << endl;
51     }
52 }
53
54 vector<pair<int, int>> tree(const int n) {
55     const int root = num(1, n);
56     vector<int> v1, v2;
57     v1.emplace_back(root);
58     for (int i = 1; i <= n; ++i)
59         if (i != root)
60             v2.emplace_back(i);
61     random_shuffle(all(v2));
62     vector<pair<int, int>> edges;
63     while (!v2.empty()) {
64         const int idx = num(0, (int)v1.size() - 1);
65         edges.emplace_back(v1[idx], v2.back());
66         v1.emplace_back(v2.back());
67         v2.pop_back();
68     }
69     return edges;
70 }
71
72 vector<pair<int, int>> forest(const int n) {
73     int val = n;
74     vector<pair<int, int>> edges;
75     int oft = 0;
76     while (val > 0) {
77         const int cur = num(1, val);
78         auto e = tree(cur);
79         for (auto [u, v] : e)
80             edges.emplace_back(u + oft, v + oft);
81         val -= cur;
82         oft += cur;
83     }
84     return edges;
85 }
86
87 vector<pair<int, int>> connected_graph(const int n) {
88     auto e = tree(n);
89     set<pair<int, int>> s(e.begin(), e.end());
90     const int ERROR = n;

```

```

91     int q = num(0, max(0ll, (n - 1) * (n - 2)) / 2 + ERROR);
92     while (q--) {
93         int u = num(1, n), v = num(1, n);
94         if (u == v || s.count(make_pair(u, v)) || s.count(make_pair(v, u)))
95             continue;
96         e.emplace_back(u, v);
97         s.emplace(u, v);
98     }
99     return e;
100 }
101
102 vector<pair<int, int>> graph(const int n) {
103     int q = num(0, n * (n - 1) / 2);
104     set<pair<int, int>> s;
105     while (q--) {
106         int u = num(1, n), v = num(1, n);
107         if (u == v)
108             continue;
109         if (u > v)
110             swap(u, v);
111         s.emplace(u, v);
112     }
113     vector<pair<int, int>> edges;
114     for (auto [u, v] : s) {
115         if (rng() % 2)
116             swap(u, v);
117         edges.emplace(u, v);
118     }
119     return edges;
120 }
121
122 int num(const int l, const int r) {
123     int sz = r - l + 1;
124     int n = rng() % sz;
125     return n + l;
126 }
127
128 vector<int> vec(const int n, const int l, const int r) {
129     vector<int> arr(n);
130     for (int &x : arr)
131         x = num(l, r);
132     return arr;
133 }
134
135 vector<vector<int>> matrix(const int n, const int m, const int l, const int
    r) {
136     vector<vector<int>> mt;
137     for (int i = 0; i < n; ++i)
138         mt.emplace_back(vec(m, l, r));
139     return mt;
140 }
141
142 string str(const int n, const int c = 26) {
143     string ans;
144     for (int i = 0; i < n; ++i)
145         ans += char(rng() % c + 'a');
146     return ans;
147 }
148
149 string str01(const int n) {
150     string ans;
151     for (int i = 0; i < n; ++i) {
152         ans += char(rng() % 2 + '0');
153     }
154     return ans;

```

```

155 }
156
157 string spaced_str(const int n, const int size, const int c = 26) {
158     string ans;
159     for (int i = 0; i < size; ++i) {
160         if (i)
161             ans += ' ';
162         ans += str(n, c);
163     }
164     return ans;
165 }

```

9.3. Run

```

1 #!/bin/bash
2
3 # Runs a.cpp infinitely against a gen.cpp input.
4 # Stops if there's an error like assertion error.
5
6 g++ -std=c++17 gen.cpp -o gen
7 g++ -std=c++17 a.cpp -o a
8
9 for((i=1;;i++)); do
10     echo $i
11     ./gen $i > in
12     time ./a < in > ol
13     if [[ $? -ne 0 ]]; then
14         break
15     fi
16 done
17
18 cat in

```

10. Strings

10.1. Trie - Maximum Xor Sum

```

1 // XOR(L,R) = XOR(L,L-1) ^ XOR(L,R)
2 ans= pre = 0
3 Trie.insert(0)
4 for i=1 to N:
5     pre = pre XOR a[i]
6     Trie.insert(pre)
7     ans=max(ans, Trie.query(pre))
8 print ans
9
10 // a funcao query é a mesma da maximum xor between two elements

```

10.2. Trie - Maximum Xor Two Elements

```

1 1. Dada uma trie de números binários e um numero X, tente achar o número
   máximo que resultante da operação XOR
2
3 Ex: Para o número 10(=(1010)2), o número que resulta no xor máximo é (0101)2
   , tente acha-lo na trie.

```

10.3. Z-Function

```

1 // What is Z Array?
2 // For a string str[0..n-1], Z array is of same length as string.
3 // An element Z[i] of Z array stores length of the longest substring

```

```

4 // starting from str[i] which is also a prefix of str[0..n-1]. The
5 // first entry of Z array is meaning less as complete string is always
6 // prefix of itself.
7 // Example:
8 // Index
9 // 0 1 2 3 4 5 6 7 8 9 10 11
10 // Text
11 // a a b c a a b x a a a z
12 // Z values
13 // X 1 0 0 3 1 0 0 2 2 1 0
14 // More Examples:
15 // str = "aaaaaa"
16 // Z[] = {x, 5, 4, 3, 2, 1}
17
18 // str = "aabaacd"
19 // Z[] = {x, 1, 0, 2, 1, 0, 0}
20
21 // str = "abababab"
22 // Z[] = {x, 0, 6, 0, 4, 0, 2, 0}
23
24 vector<int> z_function(const string &s) {
25     vector<int> z(s.size());
26     int l = -1, r = -1;
27     for (int i = 1; i < s.size(); ++i) {
28         z[i] = i >= r ? 0 : min(r - i, z[i - l]);
29         while (i + z[i] < s.size() && s[i + z[i]] == s[z[i]])
30             z[i]++;
31         if (i + z[i] > r)
32             l = i, r = i + z[i];
33     }
34     return z;
35 }

```

10.4. Aho Corasick

```

1 /// REQUIRES trie.cpp
2
3 class Aho {
4 private:
5     // node of the output list
6     struct Out_Node {
7         vector<int> str_idx;
8         Out_Node *next = nullptr;
9     };
10
11     vector<Trie::Node *> fail;
12     Trie trie;
13     // list of nodes of output
14     vector<Out_Node *> out_node;
15     const vector<string> arr;
16
17     /// Time Complexity: O(number of characters in arr)
18     void build_trie() {
19         const int n = arr.size();
20         int node_cnt = 1;
21
22         for (int i = 0; i < n; ++i)
23             node_cnt += arr[i].size();
24
25         out_node.reserve(node_cnt);
26         for (int i = 0; i < node_cnt; ++i)
27             out_node.push_back(new Out_Node());
28
29         fail.resize(node_cnt);

```

```

30     for (int i = 0; i < n; ++i) {
31         const int id = trie.insert(arr[i]);
32         out_node[id]->str_idx.push_back(i);
33     }
34
35     this->build_failures();
36 }
37
38 /// Returns the fail node of cur.
39 Trie::Node *find_fail_node(Trie::Node *cur, char c) {
40     while (cur != this->trie.root() && !cur->next.count(c))
41         cur = fail[cur->id];
42     // if cur is pointing to the root node and c is not a child
43     if (!cur->next.count(c))
44         return trie.root();
45     return cur->next[c];
46 }
47
48 /// Time Complexity: O(number of characters in arr)
49 void build_failures() {
50     queue<const Trie::Node *> q;
51
52     fail[trie.root()->id] = trie.root();
53     for (const pair<char, Trie::Node *> v : trie.root()->next) {
54         q.emplace(v.second);
55         fail[v.second->id] = trie.root();
56         out_node[v.second->id]->next = out_node[trie.root()->id];
57     }
58
59     while (!q.empty()) {
60         const Trie::Node *u = q.front();
61         q.pop();
62
63         for (const pair<char, Trie::Node *> x : u->next) {
64             const char c = x.first;
65             const Trie::Node *v = x.second;
66             Trie::Node *fail_node = find_fail_node(fail[u->id], c);
67             fail[v->id] = fail_node;
68
69             if (!out_node[fail_node->id]->str_idx.empty())
70                 out_node[v->id]->next = out_node[fail_node->id];
71             else
72                 out_node[v->id]->next = out_node[fail_node->id]->next;
73
74             q.emplace(v);
75         }
76     }
77 }
78
79 vector<vector<pair<int, int>>> aho_find_occurrences(const string &text) {
80     vector<vector<pair<int, int>>> ans(arr.size());
81     Trie::Node *cur = trie.root();
82
83     for (int i = 0; i < text.size(); ++i) {
84         cur = find_fail_node(cur, text[i]);
85         for (Out_Node *node = out_node[cur->id]; node != nullptr;
86              node = node->next)
87             for (const int idx : node->str_idx)
88                 ans[idx].emplace_back(i - (int)arr[idx].size() + 1, i);
89     }
90     return ans;
91 }
92
93 public:
94     /// Constructor that builds the trie and the failures.

```

```

95     ///
96     /// Time Complexity: O(number of characters in arr)
97     Aho(const vector<string> &arr) : arr(arr) { this->build_trie(); }
98
99     /// Searches in text for all occurrences of all strings in array arr.
100    ///
101    /// Time Complexity: O(text.size() + number of characters in arr)
102    vector<vector<pair<int, int>>> find_occurrences(const string &text) {
103        return this->aho_find_occurrences(text);
104    }
105 };

```

10.5. Hashing

```

1  // Global vector used in the class.
2  vector<int> hash_base;
3
4  class Hash {
5      /// Prime numbers to be used in mod operations
6      const vector<int> m = {1000000007, 1000000009};
7
8      vector<vector<int>>> hash_table;
9      vector<vector<int>>> pot;
10     // size of the string
11     const int n;
12
13 private:
14     static int mod(int n, int m) {
15         n %= m;
16         if (n < 0)
17             n += m;
18         return n;
19     }
20
21     /// Time Complexity: O(1)
22     pair<int, int> hash_query(const int l, const int r) {
23         vector<int> ans(m.size());
24
25         if (l == 0) {
26             for (int i = 0; i < m.size(); i++)
27                 ans[i] = hash_table[i][r];
28         } else {
29             for (int i = 0; i < m.size(); i++)
30                 ans[i] =
31                     mod((hash_table[i][r] - hash_table[i][l - 1] * pot[i][r - 1 +
32                             1])),
33                         m[i]);
34         }
35
36         return {ans.front(), ans.back()};
37     }
38
39     /// Time Complexity: O(m.size())
40     void build_base() {
41         if (!hash_base.empty())
42             return;
43         random_device rd;
44         mt19937 gen(rd());
45         uniform_int_distribution<int> distribution(CHAR_MAX, INT_MAX);
46         hash_base.resize(m.size());
47         for (int i = 0; i < hash_base.size(); ++i)
48             hash_base[i] = distribution(gen);
49     }

```

```

50 /// Time Complexity: O(n)
51 void build_table(const string &s) {
52     pot.resize(m.size(), vector<int>(this->n));
53     hash_table.resize(m.size(), vector<int>(this->n));
54
55     for (int i = 0; i < m.size(); i++) {
56         pot[i][0] = 1;
57         hash_table[i][0] = s[0];
58         for (int j = 1; j < this->n; j++) {
59             hash_table[i][j] =
60                 mod(s[j] + hash_table[i][j - 1] * hash_base[i], m[i]);
61             pot[i][j] = mod(pot[i][j - 1] * hash_base[i], m[i]);
62         }
63     }
64 }
65
66 public:
67 /// Constructor that builds the hash and pot tables and the hash_base
68     vector.
69 /// Time Complexity: O(n)
70 Hash(const string &s) : n(s.size()) {
71     build_base();
72     build_table(s);
73 }
74
75 /// Returns the hash from l to r.
76 ///
77 /// Time Complexity: O(1) -> Actually O(number_of_primes)
78 pair<int, int> query(const int l, const int r) {
79     assert(0 <= l), assert(l <= r), assert(r < this->n);
80     return hash_query(l, r);
81 }
82 };

```

10.6. Kmp

```

1 /// Builds the pi array for the KMP algorithm.
2 ///
3 /// Time Complexity: O(n)
4 vector<int> pi(const string &pat) {
5     vector<int> ans(pat.size() + 1, -1);
6     int i = 0, j = -1;
7     while (i < pat.size()) {
8         while (j >= 0 && pat[i] != pat[j])
9             j = ans[j];
10        ++i, ++j;
11        ans[i] = j;
12    }
13    return ans;
14 }
15
16 /// Returns the occurrences of a pattern in a text.
17 ///
18 /// Time Complexity: O(n + m)
19 vector<int> kmp(const string &txt, const string &pat) {
20     vector<int> p = pi(pat);
21     vector<int> ans;
22
23     for (int i = 0, j = 0; i < txt.size(); ++i) {
24         while (j >= 0 && pat[j] != txt[i])
25             j = p[j];
26         if (++j == pat.size()) {
27             ans.emplace_back(i);

```

```

28         j = p[j];
29     }
30 }
31 return ans;
32 }

```

10.7. Lcs K Strings

```

1 // Make the change below in SuffixArray code.
2 int MaximumNumberOfStrings;
3
4 void build_suffix_array() {
5     vector<pair<Rank, int>> ranks(this->n + 1);
6     vector<int> arr;
7
8     for (int i = 1, separators = 0; i <= n; i++)
9         if (this->s[i] > 0) {
10             ranks[i] = pair<Rank, int>(Rank((int)this->s[i] +
11                 MaximumNumberOfStrings, 0), i);
12             this->s[i] += MaximumNumberOfStrings;
13         } else {
14             ranks[i] = pair<Rank, int>(Rank(separators, 0), i);
15             this->s[i] = separators;
16             separators++;
17         }
18
19     RadixSort::sort_pairs(ranks, 256 + MaximumNumberOfStrings);
20     ...
21 }
22
23 /// Program to find the LCS between k different strings.
24 ///
25 /// Time Complexity: O(n*log(n))
26 /// Space Complexity: O(n*log(n))
27 int main() {
28     int n;
29
30     cin >> n;
31
32     MaximumNumberOfStrings = n;
33
34     vector<string> arr(n);
35
36     int sum = 0;
37     for (string &x: arr) {
38         cin >> x;
39         sum += x.size() + 1;
40     }
41
42     string concat;
43     vector<int> ind(sum + 1);
44     int cnt = 0;
45     for (string &x: arr) {
46         if (concat.size())
47             concat += (char)cnt;
48         concat += x;
49     }
50
51     cnt = 0;
52     for (int i = 0; i < concat.size(); i++) {
53         ind[i + 1] = cnt;
54         if (concat[i] < MaximumNumberOfStrings)
55             cnt++;

```



```

56 Suffix_Array say(concat);
57 vector<int> sa = say.get_suffix_array();
58 Sparse_Table spt(say.get_lcp());
59
60 vector<int> freq(n);
61 int cnt1 = 0;
62
63 /// Ignore separators
64 int i = n, j = n - 1;
65 int ans = 0;
66
67 while(true) {
68     if(cnt1 == n) {
69         ans = max(ans, spt.query(i, j - 1));
70
71         int idx = ind[sa[i]];
72         freq[idx]--;
73         if(freq[idx] == 0)
74             cnt1--;
75         i++;
76     } else if(j == (int)sa.size() - 1)
77         break;
78     else {
79         j++;
80         int idx = ind[sa[j]];
81         freq[idx]++;
82         if(freq[idx] == 1)
83             cnt1++;
84     }
85 }
86
87 cout << ans << endl;
88 }
89
90
91

```

10.8. Lexicographically Smallest Rotation

```

1 int booth(string &s) {
2     s += s;
3     int n = s.size();
4
5     vector<int> f(n, -1);
6     int k = 0;
7     for(int j = 1; j < n; j++) {
8         int sj = s[j];
9         int i = f[j - k - 1];
10        while(i != -1 && sj != s[k + i + 1]) {
11            if(sj < s[k + i + 1])
12                k = j - i - 1;
13            i = f[i];
14        }
15        if(sj != s[k + i + 1]) {
16            if(sj < s[k])
17                k = j;
18            f[j - k] = -1;
19        }
20        else
21            f[j - k] = i + 1;
22    }
23    return k;
24 }

```

10.9. Manacher (Longest Palindrome)

```

1 // https://medium.com/hackernoon/manachers-algorithm-explained-longest-palindromic-s
2
3 /// Create a string containing '#' characters between any two characters.
4 string get_modified_string(string &s) {
5     string ret;
6     for(int i = 0; i < s.size(); i++) {
7         ret.push_back('#');
8         ret.push_back(s[i]);
9     }
10    ret.push_back('#');
11    return ret;
12 }
13
14 /// Returns the first occurrence of the longest palindrome based on the lps
15    array.
16 /// Time Complexity: O(n)
17 string get_best(const int max_len, const string &str, const vector<int>
18    &lps) {
19     for(int i = 0; i < lps.size(); i++) {
20         if(lps[i] == max_len) {
21             string ans;
22             int cnt = max_len / 2;
23             int io = i - 1;
24             while(cnt) {
25                 if(str[io] != '#') {
26                     ans += str[io];
27                     cnt--;
28                 }
29                 io--;
30             }
31             reverse(ans.begin(), ans.end());
32             if(str[i] != '#')
33                 ans += str[i];
34             cnt = max_len / 2;
35             io = i + 1;
36             while(cnt) {
37                 if(str[io] != '#') {
38                     ans += str[io];
39                     cnt--;
40                 }
41                 io++;
42             }
43             return ans;
44         }
45     }
46 }
47
48 /// Returns a pair containing the size of the longest palindrome and the
49    first occurrence of it.
50 /// Time Complexity: O(n)
51 pair<int, string> manacher(string &s) {
52     int n = s.size();
53     string str = get_modified_string(s);
54     int len = (2 * n) + 1;
55     //the i-th index contains the longest palindromic substring with the i-th
56     char as the center
57     vector<int> lps(len);
58     int c = 0; //stores the center of the longest palindromic substring until
59     now

```

```

57 int r = 0; //stores the right boundary of the longest palindromic
    substring until now
58 int max_len = 0;
59 for(int i = 0; i < len; i++) {
60     //get mirror index of i
61     int mirror = (2 * c) - i;
62
63     //see if the mirror of i is expanding beyond the left boundary of
    current longest palindrome at center c
64     //if it is, then take r - i as lps[i]
65     //else take lps[mirror] as lps[i]
66     if(i < r)
67         lps[i] = min(r - i, lps[mirror]);
68
69     //expand at i
70     int a = i + (1 + lps[i]);
71     int b = i - (1 + lps[i]);
72     while(a < len && b >= 0 && str[a] == str[b]) {
73         lps[i]++;
74         a++;
75         b--;
76     }
77
78     //check if the expanded palindrome at i is expanding beyond the right
    boundary of current longest palindrome at center c
79     //if it is, the new center is i
80     if(i + lps[i] > r) {
81         c = i;
82         r = i + lps[i];
83
84         if(lps[i] > max_len) //update max_len
85             max_len = lps[i];
86     }
87 }
88
89 return make_pair(max_len, get_best(max_len, str, lps));
90 }

```

10.10. Suffix Array

```

1 // #define LCP
2 // clang-format off
3 class Suffix_Array {
4 private:
5     const string s;
6     const int n;
7
8 private:
9     /// OBS: Suffix Array build code imported from:
10    ///
11    /// https://github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/Strings/SuffixArray.cpp
12    /// Time Complexity: O(n*(log n))
13    vector<int> build_suffix_array() {
14        int c = 0;
15        vector<int> temp(n), posBucket(n), bucket(n), bpos(n), out(n);
16        for (int i = 0; i < n; i++)
17            out[i] = i;
18        sort(out.begin(), out.end(),
19             [&](int a, int b) { return this->s[a] < this->s[b]; });
20        for (int i = 0; i < n; i++) {
21            bucket[i] = c;
22            if (i + 1 == n || this->s[out[i]] != this->s[out[i + 1]])
23                c++;

```

```

24    }
25    for (int h = 1; h < n && c < n; h <= 1) {
26        for (int i = 0; i < n; i++)
27            posBucket[out[i]] = bucket[i];
28        for (int i = n - 1; i >= 0; i--)
29            bpos[bucket[i]] = i;
30        for (int i = 0; i < n; i++)
31            if (out[i] >= n - h)
32                temp[bpos[bucket[i]]++] = out[i];
33        for (int i = 0; i < n; i++)
34            if (out[i] >= h)
35                temp[bpos[posBucket[out[i] - h]]++] = out[i] - h;
36        c = 0;
37        for (int i = 0; i + 1 < n; i++) {
38            const int tmp = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h) ||
39                (posBucket[temp[i + 1] + h] != posBucket[temp[i] +
40                    h]);
41            bucket[i] = c;
42            c += tmp;
43        }
44        bucket[n - 1] = c++;
45        temp.swap(out);
46    }
47    return out;
48 }
49
50 vector<int> build_inverse_suffix() {
51     vector<int> inverse_suffix(this->n);
52     for (int i = 0; i < this->n; ++i)
53         inverse_suffix[sa[i]] = i;
54     return inverse_suffix;
55 }
56
57 #ifdef LCP
58 /// Builds the lcp (Longest Common Prefix) array for the string s.
59 /// A value lcp[i] indicates length of the longest common prefix of the
60 /// suffixes indexed by i and i + 1. Implementation of the Kasai's
61 /// Algorithm.
62 ///
63 /// Time Complexity: O(n)
64 vector<int> build_lcp() {
65     vector<int> lcp(this->n, 0);
66     for (int i = 0, k = 0; i < this->n; ++i) {
67         if (inverse_suffix[i] == this->n - 1)
68             k = 0;
69         else {
70             const int j = sa[inverse_suffix[i] + 1];
71             while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
72                 ++k;
73             lcp[inverse_suffix[i]] = k;
74             k -= k > 0;
75         }
76     }
77     return lcp;
78 }
79
80 int _lcs(const int separator) {
81     int ans = 0;
82     for (int i = 0; i + 1 < this->sa.size(); ++i) {
83         const int left = this->sa[i], right = this->sa[i + 1];
84         if ((left < separator && right > separator) ||
85             (left > separator && right < separator))
86             ans = max(ans, lcp[i]);
87     }
88     return ans;
89 }

```

```

87     }
88     #endif
89
90     /// Returns the minimum index, in the range [l, r], in which after advance
91     i positions the character c is present.
92     ///
93     /// Time Complexity: O(log n)
94     int lower(const char c, const int i, int l, int r) {
95         int ans = -1;
96         while (l <= r) {
97             int mid = (l + r) / 2;
98             if (sa[mid] + i < s.size() && s[sa[mid] + i] >= c) {
99                 ans = mid;
100                 r = mid - 1;
101             } else
102                 l = mid + 1;
103         }
104         return ans;
105     };
106
107     /// Returns the maximum index in the range [l, r], such that after advance
108     i positions the character c is present.
109     ///
110     /// Time Complexity: O(log n)
111     int upper(const char c, const int i, int l, int r) {
112         int ans = -1;
113         while (l <= r) {
114             int mid = (l + r) / 2;
115             if (sa[mid] + i >= s.size() || s[sa[mid] + i] <= c) {
116                 ans = mid;
117                 l = mid + 1;
118             } else
119                 r = mid - 1;
120         }
121         return ans;
122     };
123
124 public:
125     Suffix_Array(const string &s) : n(s.size()), s(s) {}
126
127     const vector<int> sa = build_suffix_array();
128     /// Position of the i-th character in suffix array.
129     const vector<int> inverse_suffix = build_inverse_suffix();
130     #ifdef LCP
131     const vector<int> lcp = build_lcp();
132     #endif
133
134     /// LCS of two strings A and B. The string s must be initialized in the
135     /// constructor as the string (A + '$' + B).
136     /// The string A starts at index 1 and ends at index (separator - 1).
137     /// The string B starts at index (separator + 1) and ends at the end of the
138     /// string.
139     ///
140     /// Time Complexity: O(n)
141     int lcs(const int separator) {
142         assert(!isalpha(this->s[separator]) && !isdigit(this->s[separator]));
143         return _lcs(separator);
144     }
145     #endif
146
147     void print() {
148         for(int i = 0; i < n; ++i)
149             cerr << s.substr(sa[i]) << endl;

```

```

150     }
151
152     /// Returns the range, inside the range [l, r], in which after advance i
153     /// positions the character c is present.
154     ///
155     /// Time Complexity: O(log n)
156     pair<int, int> range(const char c, const int i, int l, int r) {
157         l = lower(c, i, l, r), r = upper(c, i, l, r);
158         return min(l, r) == -1 ? pair<int, int>(-1, -1) : pair<int, int>(l, r);
159     }
160 };
161 // clang-format on

```

10.11. Suffix Array Mine

```

1 namespace RadixSort {
2     /// Sorts the array arr stably in ascending order.
3     ///
4     /// Time Complexity: O(n + max_element)
5     /// Space Complexity: O(n + max_element)
6     template <typename T>
7     void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),
8               const int begin = 0) {
9         const int n = arr.size();
10        vector<T> new_order(n);
11        vector<int> count(max_element + 1, 0);
12
13        for (int i = begin; i < n; ++i)
14            ++count[get_key(arr[i])];
15
16        for (int i = 1; i <= max_element; ++i)
17            count[i] += count[i - 1];
18
19        for (int i = n - 1; i >= begin; --i) {
20            new_order[count[get_key(arr[i])] - (begin == 0)] = arr[i];
21            --count[get_key(arr[i])];
22        }
23
24        arr = move(new_order);
25    }
26
27    /// Sorts an array by their pair of ranks stably in ascending order.
28    template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
29        // sort by the second rank
30        RadixSort::sort<T>(
31            arr, rank_size, [](T &item) { return item.first.second; }, 0);
32
33        // sort by the first rank
34        RadixSort::sort<T>(
35            arr, rank_size, [](T &item) { return item.first.first; }, 0);
36    }
37 } // namespace RadixSort
38
39 class Suffix_Array {
40     typedef pair<int, int> Rank;
41
42     vector<vector<int>> rank_table;
43     const vector<int> log_array = build_log_array();
44
45     vector<int> build_log_array() {
46         vector<int> log_array(this->n + 1, 0);
47         for (int i = 2; i <= this->n; ++i)
48             log_array[i] = log_array[i / 2] + 1;
49         return log_array;

```

```

50 }
51
52 /// Time Complexity: O(n*log(n))
53 vector<int> build_suffix_array() {
54     // the tuple below represents the rank and the index associated with it
55     vector<pair<Rank, int>> ranks(this->n);
56     vector<int> arr(this->n);
57
58     for (int i = 0; i < n; ++i)
59         ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);
60
61     #ifdef BUILD_TABLE
62     int rank_table_size = 0;
63     this->rank_table.resize(log_array[this->n] + 2);
64     #endif
65     RadixSort::sort_pairs(ranks, 256);
66     build_ranks(ranks, arr);
67
68     {
69         int jump = 1;
70         int max_rank = arr[ranks.back().second];
71
72         // it will be compared intervals a pair of intervals (i, jump-1), (i +
73         // jump, i + 2*jump - 1). The variable jump is always a power of 2
74         #ifdef BUILD_TABLE
75         while (jump / 2 < this->n) {
76             #else
77             while (max_rank != this->n) {
78                 #endif
79                 for (int i = 0; i < this->n; ++i) {
80                     ranks[i].first.first = arr[i];
81                     ranks[i].first.second = (i + jump < this->n ? arr[i + jump] : 0);
82                     ranks[i].second = i;
83                 }
84
85                 #ifdef BUILD_TABLE
86                 // inserting only the ranks in the table
87                 transform(ranks.begin(), ranks.end(),
88                     back_inserter(rank_table[rank_table_size++]),
89                     [](pair<Rank, int> &pair) { return pair.first.first; });
90                 #endif
91                 RadixSort::sort_pairs(ranks, n);
92                 build_ranks(ranks, arr);
93
94                 max_rank = arr[ranks.back().second];
95                 jump *= 2;
96             }
97         }
98
99         vector<int> sa(this->n);
100         for (int i = 0; i < this->n; ++i)
101             sa[arr[i] - 1] = i;
102         return sa;
103     }
104
105     int _compare(const int i, const int j, const int length) {
106         const int k = this->log_array[length]; // floor log2(length)
107         const int jump = length - (1ll << k);
108
109         const pair<int, int> iRank = {
110             this->rank_table[k][i],
111             (i + jump < this->n ? this->rank_table[k][i + jump] : -1)};
112         const pair<int, int> jRank = {
113             this->rank_table[k][j],
114             (j + jump < this->n ? this->rank_table[k][j + jump] : -1)};

```

```

115         return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
116     }
117
118     /// Compares two substrings beginning at indexes i and j of a fixed length.
119     ///
120     /// Time Complexity: O(1)
121     int compare(const int i, const int j, const int length) {
122         assert(0 <= i && i < this->n && 0 <= j && j < this->n);
123         assert(i + length - 1 < this->n && j + length - 1 < this->n);
124         return _compare(i, j, length);
125     }
126 };

```

10.12. Suffix Automaton

```

1 class Suffix_Automaton {
2 private:
3     struct state {
4         map<char, int> next;
5         // Length of the current substring which is the longest in the ith class.
6         const int len;
7         /// Contains a link to the previous substring from the ith class.
8         Contains
9         /// unique substrings from next(prev) this state.
10        int prev;
11        /// Contains the index of the last position of the first substring.
12        const int first_pos;
13        /// Whether the ith node is terminal or not.
14        bool is_terminal = false;
15
16        state(const map<char, int> next, const int len, const int prev,
17            const int first_pos)
18            : next(next), len(len), prev(prev), first_pos(first_pos) {}
19    };
20    vector<state> st;
21    int last = 0;
22
23    void build(const string &s) {
24        st.emplace_back(map<char, int>(), 0, -1, -1);
25
26        for (int i = 0; i < s.size(); ++i) {
27            st.emplace_back(map<char, int>(), i + 1, 0, i);
28            const int cur = (int)st.size() - 1;
29
30            int prev = last;
31            while (prev >= 0 && !st[prev].next.count(s[i])) {
32                st[prev].next[s[i]] = cur;
33                prev = st[prev].prev;
34            }
35
36            if (prev != -1) {
37                const int q = st[prev].next[s[i]];
38                if (st[prev].len + 1 == st[q].len) {
39                    st[cur].prev = q;
40                } else {
41                    st.emplace_back(st[q].next, st[prev].len + 1, st[q].prev,
42                        st[q].first_pos);
43                    const int qq = (int)st.size() - 1;
44                    st[q].prev = st[cur].prev = qq;
45                    while (prev >= 0) {
46                        auto it = st[prev].next.find(s[i]);
47                        if (it == st[prev].next.end() || it->second != q)
48                            break;
49                        it->second = qq;

```

```

49         prev = st[prev].prev;
50     }
51 }
52 }
53 last = cur;
54 }
55 }
56
57 void find_terminals() {
58     int p = last;
59     while (p > 0) {
60         st[p].is_terminal = true;
61         p = st[p].prev;
62     }
63 }
64
65 vector<int> dp_ocur;
66 int _ocur(const int idx) {
67     int &ret = dp_ocur[idx];
68     if (~ret)
69         return ret;
70     ret = st[idx].is_terminal;
71     for (const pair<char, int> &p : st[idx].next)
72         ret += _ocur(p.second);
73     return ret;
74 }
75
76 public:
77 Suffix_Automaton(const string &s) {
78     st.reserve(2 * s.size());
79     build(s);
80     find_terminals();
81 }
82
83 int size() { return st.size(); }
84
85 int prev(const int idx) { return st[idx].prev; }
86
87 int len(const int idx) { return st[idx].len; }
88
89 int first_pos(const int idx) { return st[idx].first_pos; }
90
91 /// Returns the next state from state cur with character c.
92 /// Returns -1 if this state doesn't exists.
93 int next(const int cur, const char c) {
94     auto it = st[cur].next.find(c);
95     if (it == st[cur].next.end())
96         return -1;
97     return it->second;
98 }
99
100 void print() {
101     cerr << "Terminals" << endl;
102     for (int i = 0; i < st.size(); ++i)
103         if (st[i].is_terminal)
104             cerr << i << ' ';
105     cerr << endl;
106     cerr << "Edges" << endl;
107     for (int i = 0; i < st.size(); ++i)
108         for (auto [a, b] : st[i].next)
109             cerr << i << ' ' << b << ' ' << a << endl;
110 }
111
112 /// Returns the number of occurrences of the pattern ending at state idx.
113 ///

```

```

114 /// Time Complexity: O(n), amortized for q queries.
115 int ocur(const int idx) {
116     if (dp_ocur.empty())
117         dp_ocur.resize(st.size(), -1);
118     return _ocur(idx);
119 }
120
121 /// Returns the state in which the pattern s ends.
122 ///
123 /// Time complexity: O(s.size())
124 int find(const string &s) {
125     int cur = 0;
126     for (char c : s) {
127         auto it = st[cur].next.find(c);
128         if (it == st[cur].next.end())
129             return -1;
130         cur = it->second;
131     }
132     return cur;
133 }
134 };
135
136 /// To output all occurrences build the inverse_prev adjacency list
137 ///
138 /// for (int i = 1; i < st.size(); ++i)
139 ///     inverse_prev[st[i].prev].emplace_back(i);
140 ///
141 /// Then take all occurrences from state cur (where the substring ends)
142 ///
143 void output_all_occurrences(int cur, int pat_length) {
144     occ.emplace_back(st[cur].first_pos - pat_length + 1);
145     for (const int u : inverse_prev[cur])
146         output_all_occurrences(u, pat_length);
147 }
148 ///
149 /// Take care and remove all duplicates after that
150 ///
151 sort(occ.begin(), occ.end())
152 occ.resize(unique(occ.begin(), occ.end()) - occ.begin())

```

10.13. Trie

```

1 class Trie {
2 private:
3     static const int INT_LEN = 31;
4     // static const int INT_LEN = 63;
5
6 public:
7     struct Node {
8         map<char, Node *> next;
9         int id;
10        // cnt counts the number of words which pass in that node
11        int cnt = 0;
12        // word counts the number of words ending at that node
13        int word_cnt = 0;
14
15        Node(const int x) : id(x) {}
16    };
17
18 private:
19     int trie_size = 0;
20     // contains the next id to be used in a node
21     int node_cnt = 0;
22     Node *trie_root = this->make_node();

```

```

23
24 private:
25     Node *make_node() { return new Node(node_cnt++); }
26
27     int trie_insert(const string &s) {
28         Node *aux = this->root();
29         for (const char c : s) {
30             if (!aux->next.count(c))
31                 aux->next[c] = this->make_node();
32             aux = aux->next[c];
33             ++aux->cnt;
34         }
35         ++aux->word_cnt;
36         ++this->trie_size;
37         return aux->id;
38     }
39
40     void trie_erase(const string &s) {
41         Node *aux = this->root();
42         for (const char c : s) {
43             Node *last = aux;
44             aux = aux->next[c];
45             --aux->cnt;
46             if (aux->cnt == 0) {
47                 last->next.erase(c);
48                 aux = nullptr;
49                 break;
50             }
51         }
52         if (aux != nullptr)
53             --aux->word_cnt;
54         --this->trie_size;
55     }
56
57     int trie_count(const string &s) {
58         Node *aux = this->root();
59         for (const char c : s) {
60             if (aux->next.count(c))
61                 aux = aux->next[c];
62             else
63                 return 0;
64         }
65         return aux->word_cnt;
66     }
67
68     int trie_query_xor_max(const string &s) {
69         Node *aux = this->root();
70         int ans = 0;
71         for (const char c : s) {
72             const char inv = (c == '0' ? '1' : '0');
73             if (aux->next.count(inv)) {
74                 ans = (ans << 111) | (inv - '0');
75                 aux = aux->next[inv];
76             } else {
77                 ans = (ans << 111) | (c - '0');
78                 aux = aux->next[c];
79             }
80         }
81         return ans;
82     }
83
84 public:
85     Trie() {}
86
87     Node *root() { return this->trie_root; }

```

```

88
89     int size() { return this->trie_size; }
90
91     /// Returns the number of nodes present in the trie.
92     int node_count() { return this->node_cnt; }
93
94     /// Inserts s in the trie.
95     ///
96     /// Returns the id of the last character of the string in the trie.
97     ///
98     /// Time Complexity: O(s.size())
99     int insert(const string &s) { return this->trie_insert(s); }
100
101     /// Inserts the binary representation of x in the trie.
102     ///
103     /// Time Complexity: O(log x)
104     int insert(const int x) {
105         assert(x >= 0);
106         // converting x to binary representation
107         return this->trie_insert(bitset<INT_LEN>(x).to_string());
108     }
109
110     /// Removes the string s from the trie.
111     ///
112     /// Time Complexity: O(s.size())
113     void erase(const string &s) { this->trie_erase(s); }
114
115     /// Removes the binary representation of x from the trie.
116     ///
117     /// Time Complexity: O(log x)
118     void erase(const int x) {
119         assert(x >= 0);
120         // converting x to binary representation
121         this->trie_erase(bitset<INT_LEN>(x).to_string());
122     }
123
124     /// Returns the number of maximum xor sum with x present in the trie.
125     ///
126     /// Time Complexity: O(log x)
127     int query_xor_max(const int x) {
128         assert(x >= 0);
129         // converting x to binary representation
130         return this->trie_query_xor_max(bitset<INT_LEN>(x).to_string());
131     }
132
133     /// Returns the number of strings equal to s present in the trie.
134     ///
135     /// Time Complexity: O(s.size())
136     int count(const string &s) { return this->trie_count(s); }
137 };

```