

C++ Competitive Programming Library

DO NOT DISCLOSE OR DISTRIBUTE

bfs.07 - Bernardo Flores Salmeron

1	Template	3			
2	Data Structures	3			
2.1	Bit2D	3	4.2	Closest Pair Of Points	18
2.2	Merge Sort Tree (K-Esimo Maior Elemento Num Intervalo, Valores Maiores Que K Num Intervalo,	4	4.3	Condicao De Existencia De Um Triangulo	19
2.3	Mos Algorithm	4	4.4	Convex Hull	19
2.4	Sqrt Decomposition	5	4.5	Cross Product	19
2.5	Bit	5	4.6	Distance Point Segment	19
2.6	Bit (Range Update)	5	4.7	Line-Line Intersection	19
2.7	Counting Inversions (Minimum Number Of Adjacent Swaps To Sort Array) .	6	4.8	Line-Point Distance	19
2.8	Ordered Set	6	4.9	Point Inside Convex Polygon - Log(N)	20
2.9	Persistent Segment Tree	7	4.10	Point Inside Polygon	21
2.10	Segment Tree	8	4.11	Points Inside And In Boundary Polygon	22
2.11	Segment Tree 2D	9	4.12	Polygon Area (3D)	22
2.12	Segment Tree Polynomial	10	4.13	Polygon Area	22
2.13	Sparse Table	11	4.14	Segment-Segment Intersection	22
2.14	Treap	12	4.15	Upper And Lower Hull	23
3	Dp	15	4.16	Circle Circle Intersection	23
3.1	Achar Maior Palindromo	15	4.17	Circle Circle Intersection	23
3.2	Digit Dp	15	4.18	Struct Point And Line	24
3.3	Longest Common Subsequence	15	5	Graphs	25
3.4	Longest Common Substring	16	5.1	All Eulerian Path Or Tour	25
3.5	Longest Increasing Subsequence 2D (Not Sorted)	16	5.2	Articulation Points	26
3.6	Longest Increasing Subsequence 2D (Sorted)	16	5.3	Bellman Ford	27
3.7	Longest Increasing Subsequence	17	5.4	Bipartite Check	28
3.8	Subset Sum Com Bitset	17	5.5	Bridges	28
3.9	Catalan	17	5.6	Centroid Decomposition	28
3.10	Catalan	17	5.7	De Bruijn Sequence	29
3.11	Coin Change Problem	17	5.8	Diameter In Tree	29
3.12	Knapsack	18	5.9	Dijkstra + Dij Graph	29
4	Geometry	18	5.10	Dinic	30
4.1	Centro De Massa De Um Poligono	18	5.11	Dsu	34
			5.12	Floyd Warshall	34
			5.13	Functional Graph	34
			5.14	Girth (Shortest Cycle In A Graph)	36

5.15	Hld	36	6.24	Random Numbers	44
5.16	Hungarian	37	6.25	Readint	44
5.17	Kruskal	37	6.26	Time Measure	44
5.18	Lca	38	7 Math		45
5.19	Maximum Independent Set (Set Of Vertices That Arent Directly Connected)	40	7.1	Bell Numbers	45
5.20	Maximum Path Unweighted Graph	40	7.2	Binary Exponentiation	45
5.21	Minimum Edge Cover (Set Of Edges That Are Adjacent To All Vertices)	40	7.3	Chinese Remainder Theorem	45
5.22	Minimum Path Cover In Dag	40	7.4	Combinatorics	45
5.23	Minimum Path Cover In Dag	40	7.5	Diophantine Equation	46
5.24	Number Of Different Spanning Trees In A Complete Graph	40	7.6	Divisors	46
5.25	Number Of Ways To Make A Graph Connected	40	7.7	Euler Totient	46
5.26	Pruffer Decode	40	7.8	Extended Euclidean	47
5.27	Pruffer Encode	41	7.9	Factorization	47
5.28	Pruffer Properties	41	7.10	Inclusion Exclusion	47
5.29	Remove All Bridges From Graph	41	7.11	Inclusion Exclusion	47
5.30	Scc (Kosaraju)	41	7.12	Markov Chains	47
5.31	Topological Sort	42	7.13	Matrix Exponentiation	48
5.32	Tree Distance	42	7.14	Matrix Exponentiation	48
6 Language Stuff		43	7.15	Pollard Rho (Find A Divisor)	48
6.1	Climits	43	7.16	Primality Check	49
6.2	Checagem E Tranformacao De Caractere	43	7.17	Primes	49
6.3	Conta Digitos 1 Ate N	43	7.18	Sieve + Segmented Sieve	49
6.4	Escrita Em Arquivo	43	7.19	Stars And Bars	50
6.5	Gcd	43	8 Miscellaneous		50
6.6	Hipotenusa	43	8.1	2-Sat	50
6.7	Int To Binary String	43	8.2	Interval Scheduling	50
6.8	Int To String	43	8.3	Interval Scheduling	50
6.9	Leitura De Arquivo	43	8.4	Oito Rainhas	50
6.10	Max E Min Element Num Vetor	43	8.5	Sliding Window Minimum	51
6.11	Permutacao	43	8.6	Torre De Hanoi	51
6.12	Remove Repeticoes Continuas Num Vetor	43	8.7	Counting Frequency Of Digits From 1 To K	51
6.13	Rotate (Left)	43	8.8	Infix To Postfix	51
6.14	Rotate (Right)	43	8.9	Kadane	52
6.15	Scanf De Uma String	43	8.10	Kadane (Segment Tree)	52
6.16	Split Function	43	8.11	Kadane 2D	52
6.17	String To Long Long	44	8.12	Largest Area In Histogram	53
6.18	Substring	44	8.13	Point Compression	53
6.19	Width	44	8.14	Ternary Search	53
6.20	Binary String To Int	44	9 Strings		54
6.21	Check	44	9.1	Trie - Maximum Xor Sum	54
6.22	Check Overflow	44	9.2	Trie - Maximum Xor Two Elements	54
6.23	Counting Bits	44	9.3	Z-Function	54

9.4	Aho Corasick	54
9.5	Hashing	55
9.6	Kmp	56
9.7	Lcs K Strings	56
9.8	Lexicographically Smallest Rotation	57
9.9	Manacher (Longest Palindrome)	57
9.10	Suffix Array	58
9.11	Suffix Array Pessoa	60
9.12	Trie	60

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         int n, m;
6         vector<vector<int>>> tree;
7
8     private:
9         // Returns an integer which constains only the least significant bit.
10        int low(int i) {
11            return i & (-i);
12        }
13
14        void bit_update(const int x, const int y, const int delta) {
15            for(int i = x; i < n; i += low(i))
16                for(int j = y; j < m; j += low(j))
17                    this->tree[i][j] += delta;
18        }
19
20        int bit_query(const int x, const int y) {
21            int ans = 0;
22            for(int i = x; i > 0; i -= low(i))
23                for(int j = y; j > 0; j -= low(j))
24                    ans += this->tree[i][j];
25
26            return ans;
27        }
28
29    public:
30        // put the size of the array without 1 indexing.
31        /// Time Complexity: O(n * m)
32        BIT_2D(int n, int m) {
33            this->n = n + 1;
34            this->m = m + 1;
35
36            this->tree.resize(n, vector<int>(m, 0));
```

```

37 }
38
39 /// Time Complexity: O(n * m * (log(n) + log(m)))
40 BIT_2D(const vector<vector<int>> &mat) {
41     // Check if it is 1 index.
42     assert(mat[0][0] == 0);
43     this->n = mat.size();
44     this->m = mat.front().size();
45
46     this->tree.resize(n, vector<int>(m, 0));
47     for(int i = 1; i < n; i++)
48         for(int j = 1; j < m; j++)
49             update(i, j, mat[i][j]);
50 }
51
52 /// Query from (1, 1) to (x, y).
53 ///
54 /// Time Complexity: O(log(n) + log(m))
55 int prefix_query(const int x, const int y) {
56     assert(0 < x); assert(x < this->n);
57     assert(0 < y); assert(y < this->m);
58
59     return bit_query(x, y);
60 }
61
62 /// Query from (x1, y1) to (x2, y2).
63 ///
64 /// Time Complexity: O(log(n) + log(m))
65 int query(const int x1, const int y1, const int x2, const int y2) {
66     assert(0 < x1); assert(x1 <= x2); assert(x2 < this->n);
67     assert(0 < y1); assert(y1 <= y2); assert(y2 < this->m);
68
69     return bit_query(x2, y2) - bit_query(x1 - 1, y2) - bit_query(x2, y1 - 1)
70     + bit_query(x1 - 1, y1 - 1);
71 }
72
73 /// Updates point (x, y).
74 ///
75 /// Time Complexity: O(log(n) + log(m))
76 void update(const int x, const int y, const int delta) {
77     assert(0 < x); assert(x < this->n);
78     assert(0 < y); assert(y < this->m);
79
80     bit_update(x, y, delta);
81 };

```

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--;

```

```

41     query[i].ind = i;
42 }
43 sort(query, query + m, cmp);
44
45 int s,e;
46 s = e = query[0].l;
47 add(s);
48 for(int i = 0; i < m; i++) {
49     while(s > query[i].l)
50         add(--s);
51     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 /// INDEX THE ARRAY BY 1!!!
2 class BIT {
3 private:
4     vector<int> bit;
5     int n;
6
7 private:
8     int low(const int i) { return (i & (-i)); }
9
10 // point update
11 void bit_update(int i, const int delta) {
12     while (i <= this->n) {
13         this->bit[i] += delta;

```

```

14         i += this->low(i);
15     }
16 }
17
18 // point query
19 int bit_query(int i) {
20     int sum = 0;
21     while (i > 0) {
22         sum += bit[i];
23         i -= this->low(i);
24     }
25     return sum;
26 }
27
28 public:
29 BIT(const vector<int> &arr) { this->build(arr); }
30
31 BIT(const int n) {
32     // OBS: BIT IS INDEXED FROM 1
33     // THE USE OF 1-BASED ARRAY IS RECOMMENDED
34     this->n = n;
35     this->bit.resize(n + 1, 0);
36 }
37
38 // build the bit
39 void build(const vector<int> &arr) {
40     // OBS: BIT IS INDEXED FROM 1
41     // THE USE OF 1-BASED ARRAY IS RECOMMENDED
42     assert(arr.front() == 0);
43     this->n = (int)arr.size() - 1;
44     this->bit.resize(arr.size(), 0);
45
46     for (int i = 1; i <= this->n; i++)
47         this->bit_update(i, arr[i]);
48 }
49
50 // point update
51 void update(const int i, const int delta) {
52     assert(1 <= i), assert(i <= this->n);
53     this->bit_update(i, delta);
54 }
55
56 // point query
57 int query(const int i) {
58     assert(1 <= i), assert(i <= this->n);
59     return this->bit_query(i);
60 }
61
62 // range query
63 int query(const int l, const int r) {
64     assert(1 <= l), assert(l <= r), assert(r <= this->n);
65     return this->bit_query(r) - this->bit_query(l - 1);
66 }
67 };

```

2.6. Bit (Range Update)

```

1 /// INDEX THE ARRAY BY 1!!!
2 class BIT {
3 private:
4     vector<int> bit1;
5     vector<int> bit2;
6     int n;
7

```

```

8 private:
9   int low(int i) { return (i & (-i)); }
10
11   // point update
12   void update(int i, const int delta, vector<int> &bit) {
13     while (i <= this->n) {
14       bit[i] += delta;
15       i += this->low(i);
16     }
17   }
18
19   // point query
20   int query(int i, const vector<int> &bit) {
21     int sum = 0;
22     while (i > 0) {
23       sum += bit[i];
24       i -= this->low(i);
25     }
26     return sum;
27   }
28
29   // build the bit
30   void build(const vector<int> &arr) {
31     // OBS: BIT IS INDEXED FROM 1
32     // THE USE OF 1-BASED ARRAY IS MANDATORY
33     assert(arr.front() == 0);
34     this->n = (int)arr.size() - 1;
35     this->bit1.resize(arr.size(), 0);
36     this->bit2.resize(arr.size(), 0);
37
38     for (int i = 1; i <= this->n; i++)
39       this->update(i, arr[i]);
40   }
41
42 public:
43   BIT(const vector<int> &arr) { this->build(arr); }
44
45   BIT(const int n) {
46     // OBS: BIT IS INDEXED FROM 1
47     // THE USAGE OF 1-INDEXED ARRAY IS MANDATORY
48     this->n = n;
49     this->bit1.resize(n + 1, 0);
50     this->bit2.resize(n + 1, 0);
51   }
52
53   // range update
54   void update(const int l, const int r, const int delta) {
55     assert(l <= 1), assert(l <= r), assert(r <= this->n);
56     this->update(l, delta, this->bit1);
57     this->update(r + 1, -delta, this->bit1);
58     this->update(l, delta * (1 - 1), this->bit2);
59     this->update(r + 1, -delta * r, this->bit2);
60   }
61
62   // point update
63   void update(const int i, const int delta) {
64     assert(l <= i), assert(i <= this->n);
65     this->update(i, i, delta);
66   }
67
68   // range query
69   int query(const int l, const int r) {
70     assert(l <= 1), assert(l <= r), assert(r <= this->n);
71     return this->query(r) - this->query(l - 1);
72   }

```

```

73   // point prefix query
74   int query(const int i) {
75     assert(i <= this->n);
76     return (this->query(i, this->bit1) * i) - this->query(i, this->bit2);
77   }
78 };
79
80 // TESTS
81 // signed main()
82 // {
83 //   vector<int> input = {0,1,2,3,4,5,6,7};
84 //   BIT ft(input);
85
86 //   assert (1 == ft.query(1));
87 //   assert (3 == ft.query(2));
88 //   assert (6 == ft.query(3));
89 //   assert (10 == ft.query(4));
90 //   assert (15 == ft.query(5));
91 //   assert (21 == ft.query(6));
92 //   assert (28 == ft.query(7));
93 //   assert (12 == ft.query(3,5));
94 //   assert (21 == ft.query(1,6));
95 //   assert (28 == ft.query(1,7));
96 // }

```

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. 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);

```

```

17 X.insert(16);
18
19 // 1, 2, 4, 8, 16
20 // returns the k-th greatest element from 0
21 cout << *X.find_by_order(1) << endl; // 2
22 cout << *X.find_by_order(2) << endl; // 4
23 cout << *X.find_by_order(4) << endl; // 16
24 cout << (end(X) == X.find_by_order(6)) << endl; // true
25
26 // returns the number of items strictly less than a number
27 cout << X.order_of_key(-5) << endl; // 0
28 cout << X.order_of_key(1) << endl; // 0
29 cout << X.order_of_key(3) << endl; // 2
30 cout << X.order_of_key(4) << endl; // 2
31 cout << X.order_of_key(400) << endl; // 5

```

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

```

```

107  ///
108  /// Time Complexity: O(log(n))
109  void update(const int cur_version, const int prev_version, const int idx,
110            const int delta) {
111      assert(this->n > -1);
112      assert(0 <= prev_version), assert(prev_version <= cur_version);
113      this->create_version(cur_version);
114      this->version[cur_version] =
115          this->_update(this->version[cur_version],
116                     this->version[prev_version],
117                     0, this->n - 1, idx, delta);
118  }
119  /// Query from l to r.
120  ///
121  /// Time Complexity: O(log(n))
122  int query(const int version, const int l, const int r) {
123      assert(this->n > -1);
124      assert(0 <= l), assert(l <= r), assert(r < this->n);
125      return this->_query(this->version[version], 0, this->n - 1, l, r);
126  }
127  };

```

2.10. 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
69               pos) {
70         if (l == r)
71             return tree[pos] = Node(arr[l]);
72
73         int mid = (l + r) / 2;
74         return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
75                                       _build(mid + 1, r, arr, 2 * pos + 2));
76     }
77
78     int _get_first(const int l, const int r, const int i, const int j,
79                  const int v, const int pos) {
80         propagate(l, r, pos);
81
82         if (l > r || l > j || r < i)
83             return -1;
84         // Needs RMQ MAX
85         // Replace to <= for greater or (with RMQ MIN) > for smaller or
86         // equal or >= for smaller
87         if (tree[pos].val < v)
88             return -1;
89
90         if (l == r)
91             return l;
92
93         int mid = (l + r) / 2;
94         int aux = _get_first(l, mid, i, j, v, 2 * pos + 1);
95         if (aux != -1)
96             return aux;
97         return _get_first(mid + 1, r, i, j, v, 2 * pos + 2);
98     }
99
100     Node _query(const int l, const int r, const int i, const int j,
101               const int pos) {
102         propagate(l, r, pos);
103
104         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 a tree with 0.
144     ///
145     /// Time Complexity O(n)
146     Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0)); }
147
148     /// Constructor responsible for building the initial tree based on a
149     /// vector.
150     ///
151     /// Time Complexity O(n)
152     Seg_Tree(const vector<int> &arr) : n(arr.size()) { this->build(arr); }
153
154     /// Returns the first index from i to j compared to v.
155     /// Uncomment the line in the original function to get the proper element
156     /// that
157     /// may be: GREATER OR EQUAL, GREATER, SMALLER OR EQUAL, SMALLER.
158     ///
159     /// Time Complexity O(log n)
160     int get_first(const int i, const int j, const int v) {
161         assert(this->n >= 0);
162         return this->_get_first(0, this->n - 1, i, j, v, 0);
163     }
164
165     /// Update at a single index.
166     ///
167     /// Time Complexity O(log n)
168     void update(const int idx, const int delta) {

```

```

167     assert(this->n >= 0);
168     assert(0 <= idx), assert(idx < this->n);
169     this->_update(0, this->n - 1, idx, idx, delta, 0);
170 }
171
172     /// Range update from l to r.
173     ///
174     /// Time Complexity O(log n)
175     void update(const int l, const int r, const int delta) {
176         assert(this->n >= 0);
177         assert(0 <= l), assert(l <= r), assert(r < this->n);
178         this->_update(0, this->n - 1, l, r, delta, 0);
179     }
180
181     /// Query at a single index.
182     ///
183     /// Time Complexity O(log n)
184     int query(const int idx) {
185         assert(this->n >= 0);
186         assert(0 <= idx), assert(idx < this->n);
187         return this->_query(0, this->n - 1, idx, idx, 0).val;
188     }
189
190     /// Range query from l to r.
191     ///
192     /// Time Complexity O(log n)
193     int query(const int l, const int r) {
194         assert(this->n >= 0);
195         assert(0 <= l), assert(l <= r), assert(r < this->n);
196         return this->_query(0, this->n - 1, l, r, 0).val;
197     }
198 };

```

2.11. 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)
49         return tree[pos].query(y1, y2);
50
51     int mid = (l + r) / 2;
52     return merge_nodes(st_query(l, mid, x1, y1, x2, y2, 2*pos + 1),
53                       st_query(mid + 1, r, x1, y1, x2, y2, 2*pos + 2));
54 }
55
56 void st_update(const int l, const int r, const int x, const int y, const
57 int delta, const int pos) {
58     if(l > x || r < x)
59         return;
60
61     // Only supports point updates.
62     if(l == r) {
63         tree[pos].update(y, delta);
64         return;
65     }
66
67     int mid = (l + r) / 2;
68     st_update(l, mid, x, y, delta, 2*pos + 1);
69     st_update(mid + 1, r, x, y, delta, 2*pos + 2);
70     tree[pos].update(y, delta);
71 }
72
73 public:
74     Seg_Tree_2d() {
75         this->n = -1;
76         this->m = -1;
77     }
78
79     Seg_Tree_2d(const int n, const int m) {
80         this->n = n;
81         this->m = m;
82         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
83         assert(m < 10000);
84         tree.resize(4 * n, Seg_Tree(m));
85     }
86
87     Seg_Tree_2d(const int n, const int m, const vector<vector<int>> &mat) {
88         this->n = n;
89         this->m = m;
90         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
91         assert(m < 10000);
92         tree.resize(4 * n, Seg_Tree(m));
93         st_build(0, n - 1, 0, mat);

```

```

91     }
92
93     // Query from (x1, y1) to (x2, y2).
94     //
95     // Time complexity: O((log n) * (log m))
96     int query(const int x1, const int y1, const int x2, const int y2) {
97         assert(this->n > -1);
98         assert(0 <= x1); assert(x1 <= x2); assert(x2 < this->n);
99         assert(0 <= y1); assert(y1 <= y2); assert(y2 < this->n);
100         return st_query(0, this->n - 1, x1, y1, x2, y2, 0);
101     }
102
103     // Point updates on position (x, y).
104     //
105     // Time complexity: O((log n) * (log m))
106     void update(const int x, const int y, const int delta) {
107         assert(0 <= x); assert(x < this->n);
108         assert(0 <= y); assert(y < this->n);
109         st_update(0, this->n - 1, x, y, delta, 0);
110     }
111 };

```

2.12. 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
19     void apply_lazy(const int l, const int r, const int pos) {
20         tree[pos].val += (r - l + 1) * tree[pos].z0;
21         tree[pos].val += (r - l) * (r - l + 1) / 2 * tree[pos].z1;
22     }
23
24 private:
25     int n;
26
27 public:
28     vector<Node> tree;
29
30 private:
31     void st_propagate(const int l, const int r, const int pos) {
32         if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {
33             apply_lazy(l, r, pos);
34             int mid = (l + r) / 2;
35             int sz_left = mid - l + 1;
36             if (l != r) {
37                 tree[2 * pos + 1].z0 += tree[pos].z0;
38                 tree[2 * pos + 1].z1 += tree[pos].z1;
39
40                 tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
41                 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 }
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  $f(x) = z1*x + z0$ 
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  $f(x) = z1*x + z0$  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 };

```

2.13. 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             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             }
32             pow2 <<= 1;
33         }

```

```

34     }
35
36 public:
37     /// Constructor that builds the log array and the sparse table.
38     ///
39     /// Time Complexity:  $O(n \log(n))$ 
40     Sparse_Table(const vector<int> &arr) : n(arr.size()) {
41         this->build_log_array();
42         this->build_sparse_table(arr);
43     }
44
45     void print() {
46         int pow2 = 1;
47         for (int i = 0; i < table.size(); i++) {
48             int sz = (int)(table.front().size()) - pow2 + 1;
49             for (int j = 0; j < sz; j++) {
50                 cout << table[i][j] << " \n"[(j + 1) == sz];
51             }
52             pow2 <= 1;
53         }
54     }
55
56     /// Range query from l to r.
57     ///
58     /// Time Complexity:  $O(1)$ 
59     int query(const int l, const int r) {
60         assert(l <= r);
61         assert(0 <= l && r <= this->n - 1);
62
63         int lgg = lg[r - l + 1];
64         return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
65     }
66 };

```

2.14. Treap

```

1 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
2
3 // #define REVERSE
4 // #define LAZY
5 class Treap {
6 public:
7     struct Node {
8         Node *left = nullptr, *right = nullptr, *par = nullptr;
9         // Priority to be used in the treap
10        const int rank;
11        int size = 1, val;
12        // Contains the result of the range query between the node and its
13        // children.
14        int ans;
15    #ifdef LAZY
16        int lazy = 0;
17    #endif
18    #ifdef REVERSE
19        bool rev = false;
20    #endif
21
22        Node(const int val) : val(val), ans(val), rank(rng()) {}
23        Node(const int val, const int rank) : val(val), ans(val), rank(rank) {}
24    };
25 private:
26     vector<Node*> nodes;
27     int _size = 0;

```

```

28     Node *root = nullptr;
29
30 private:
31     // // Range Sum
32     // void merge_nodes(Node *node) {
33     //     node->ans = node->val;
34     //     if (node->left)
35     //         node->ans += node->left->ans;
36     //     if (node->right)
37     //         node->ans += node->right->ans;
38     // }
39
40     // #ifdef LAZY
41     // void apply_lazy(Node *node) {
42     //     node->val += node->lazy;
43     //     node->ans += node->lazy * get_size(node);
44     // }
45     // #endif
46
47     // // RMQ Min
48     // void merge_nodes(Node *node) {
49     //     node->ans = node->val;
50     //     if (node->left)
51     //         node->ans = min(node->ans, node->left->ans);
52     //     if (node->right)
53     //         node->ans = min(node->ans, node->right->ans);
54     // }
55
56     // #ifdef LAZY
57     // void apply_lazy(Node *node) {
58     //     node->val += node->lazy;
59     //     node->ans += node->lazy;
60     // }
61     // #endif
62
63     // // RMQ Max
64     // void merge_nodes(Node *node) {
65     //     node->ans = node->val;
66     //     if (node->left)
67     //         node->ans = max(node->ans, node->left->ans);
68     //     if (node->right)
69     //         node->ans = max(node->ans, node->right->ans);
70     // }
71
72     // #ifdef LAZY
73     // void apply_lazy(Node *node) {
74     //     node->val += node->lazy;
75     //     node->ans += node->lazy;
76     // }
77     // #endif
78
79     int get_size(const Node *node) { return node ? node->size : 0; }
80
81     void update_size(Node *node) {
82         if (node)
83             node->size = 1 + get_size(node->left) + get_size(node->right);
84     }
85
86     #ifdef REVERSE
87     void propagate_reverse(Node *node) {
88         if (node && node->rev) {
89             swap(node->left, node->right);
90             if (node->left)
91                 node->left->rev ^= 1;
92             if (node->right)

```

```

93     node->right->rev ^= 1;
94     node->rev = 0;
95 }
96 }
97 #endif
98
99 #ifdef LAZY
100 void propagate_lazy(Node *node) {
101     if (node && node->lazy != 0) {
102         apply_lazy(node);
103         if (node->left)
104             node->left->lazy += node->lazy;
105         if (node->right)
106             node->right->lazy += node->lazy;
107         node->lazy = 0;
108     }
109 }
110 #endif
111
112 void update_node(Node *node) {
113     if (node) {
114         update_size(node);
115     }
116     #ifdef LAZY
117     propagate_lazy(node->left);
118     propagate_lazy(node->right);
119     #endif
120     #ifdef REVERSE
121     propagate_reverse(node);
122     #endif
123     merge_nodes(node);
124 }
125
126 /// Splits the treap into to different treaps that contains nodes with
127   indexes
128   /// <= pos ans indexes > pos. The nodes l and r contains, in the end, these
129   /// two different treaps.
130 void split(Node *node, Node *l, Node *r, const int pos, Node *pl =
131   nullptr, Node *pr = nullptr) {
132     if (!node)
133         l = r = nullptr;
134     else {
135         #ifdef LAZY
136         propagate_lazy(node);
137         #endif
138         #ifdef REVERSE
139         propagate_reverse(node);
140         #endif
141         if (get_size(node->left) <= pos) {
142             node->par = pr;
143             split(node->right, node->right, r, pos - get_size(node->left) - 1,
144             pl,
145             node);
146             l = node;
147         } else {
148             node->par = pl;
149             split(node->left, l, node->left, pos, node, pr);
150             r = node;
151         }
152     }
153     update_node(node);
154 }
155
156 /// Merges to treaps (l and r) into a single one based on the rank of each

```

```

155     /// node.
156     void merge(Node *&node, Node *l, Node *r, Node *par = nullptr) {
157     #ifdef LAZY
158         propagate_lazy(l), propagate_lazy(r);
159     #endif
160     #ifdef REVERSE
161         propagate_reverse(l), propagate_reverse(r);
162     #endif
163     if (l == nullptr || r == nullptr)
164         node = (l == nullptr ? r : l);
165     else if (l->rank > r->rank) {
166         merge(l->right, l->right, r, l);
167         node = l;
168     } else {
169         merge(r->left, l, r->left, r);
170         node = r;
171     }
172     if (node)
173         node->par = par;
174     update_node(node);
175 }
176
177 Node *build(const int l, const int r, const vector<int> &arr,
178   vector<int> &rand) {
179     if (l > r)
180         return nullptr;
181
182     const int mid = (l + r) / 2;
183     Node *node = new Node(arr[mid], rand.back());
184     rand.pop_back();
185     node->right = build(mid + 1, r, arr, rand);
186     node->left = build(l, mid - 1, arr, rand);
187     update_node(node);
188
189     return node;
190 }
191
192 int _get_ith(const int idx) {
193     int ans = 0;
194     Node *cur = nodes[idx], *prev = nullptr;
195     while (cur) {
196         if (cur == nodes[idx] || prev == cur->right)
197             ans += 1 + get_size(cur->left);
198         prev = cur;
199         cur = cur->par;
200     }
201     return ans - 1;
202 }
203
204 vector<int> gen_rand(const int n) {
205     vector<int> ans(n);
206     for (int &x : ans)
207         x = rng();
208     sort(ans.begin(), ans.end());
209     return ans;
210 }
211
212 Node *_query(const int l, const int r) {
213     Node *L, *M, *R;
214     split(this->root, L, M, l - 1);
215     split(M, M, R, r - l);
216     Node *ret = new Node(*M);
217     merge(L, L, M);
218     merge(root, L, R);
219     return ret;

```

```

220     }
221
222     void _update(const int l, const int r, const int delta) {
223         Node *L, *M, *R;
224         split(this->root, L, M, l - 1);
225         split(M, M, R, r - l);
226
227         Node *node = M;
228 #ifndef LAZY
229         node->lazy = delta;
230         propagate_lazy(node);
231 #else
232         node->val += delta;
233 #endif
234
235         merge(L, L, M);
236         merge(root, L, R);
237     }
238
239     void _insert(const int pos, Node *node) {
240         this->_size += node->size;
241         Node *L, *R;
242         split(this->root, L, R, pos - 1);
243         merge(L, L, node);
244         merge(this->root, L, R);
245     }
246
247     Node *_erase(const int l, const int r) {
248         Node *L, *M, *R;
249         split(this->root, L, M, l - 1);
250         split(M, M, R, r - l);
251         merge(root, L, R);
252         this->_size -= r - l + 1;
253         return M;
254     }
255
256     void _move(const int l, const int r, const int new_pos) {
257         Node *node = _erase(l, r);
258         _insert(new_pos, node);
259     }
260
261 #ifndef REVERSE
262     void _reverse(const int l, const int r) {
263         Node *L, *M, *R;
264         split(this->root, L, M, l - 1);
265         split(M, M, R, r - l);
266
267         Node *node = M;
268         node->rev = true;
269         propagate_reverse(node);
270
271         merge(L, L, M);
272         merge(root, L, R);
273     }
274 #endif
275
276 public:
277     Treap() {}
278
279     /// Constructor that initializes the treap based on an array.
280     ///
281     /// Time Complexity: O(n)
282     Treap(const vector<int> &arr) : _size(arr.size()) {
283         vector<int> r = gen_rand(arr.size());
284         this->root = build(0, (int)arr.size() - 1, arr, r);

```

```

285     }
286
287     int size() { return _size; }
288
289     /// Moves the subarray [l, r] to the position starting at new_pos.
290     ///
291     /// Time Complexity: O(log n)
292     void move(const int l, const int r, const int new_pos) {
293         assert(0 <= new_pos), assert(new_pos <= _size - (r - l + 1));
294         _move(l, r, new_pos);
295     }
296
297     /// Moves the subarray [l, r] to the back of the array.
298     ///
299     /// Time Complexity: O(log n)
300     void move_back(const int l, const int r) {
301         assert(0 <= l), assert(l <= r), assert(r < _size);
302         _move(l, r, _size - (r - l + 1));
303     }
304
305     /// Moves the subarray [l, r] to the front of the array.
306     ///
307     /// Time Complexity: O(log n)
308     void move_front(const int l, const int r) {
309         assert(0 <= l), assert(l <= r), assert(r < _size);
310         _move(l, r, 0);
311     }
312
313 #ifndef REVERSE
314     /// Reverses the subarray [l, r].
315     ///
316     /// Time Complexity: O(log n)
317     void reverse(const int l, const int r) {
318         assert(0 <= l), assert(l <= r), assert(r < _size);
319         _reverse(l, r);
320     }
321 #endif
322
323     /// Erases the subarray [l, r].
324     ///
325     /// Time Complexity: O(log n)
326     void erase(const int l, const int r) {
327         assert(0 <= l), assert(l <= r), assert(r < _size);
328         _erase(l, r);
329     }
330
331     /// Inserts the value val at the position pos.
332     ///
333     /// Time Complexity: O(log n)
334     void insert(const int pos, const int val) {
335         assert(pos <= _size);
336         nodes.emplace_back(new Node(val));
337         _insert(pos, nodes.back());
338     }
339
340     /// Returns the index of the i-th added node.
341     ///
342     /// Time Complexity: O(log n)
343     int get_ith(const int idx) {
344         assert(0 <= idx), assert(idx < nodes.size());
345         return _get_ith(idx);
346     }
347
348     /// Sums the delta value to the position pos.
349     ///

```

```

350 /// Time Complexity: O(log n)
351 void update(const int pos, const int delta) {
352     assert(0 <= pos), assert(pos < _size);
353     _update(pos, pos, delta);
354 }
355
356 #ifndef LAZY
357 /// Sums the delta value to the subarray [l, r].
358 ///
359 /// Time Complexity: O(log n)
360 void update(const int l, const int r, const int delta) {
361     assert(0 <= l), assert(l <= r), assert(r < _size);
362     _update(l, r, delta);
363 }
364 #endif
365
366 /// Query at a single index.
367 ///
368 /// Time Complexity: O(log n)
369 int query(const int pos) {
370     assert(0 <= pos), assert(pos < _size);
371     return _query(pos, pos) ->ans;
372 }
373
374 /// Range query from l to r.
375 ///
376 /// Time Complexity: O(log n)
377 int query(const int l, const int r) {
378     assert(0 <= l), assert(l <= r), assert(r < _size);
379     return _query(l, r) ->ans;
380 }
381 };

```

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
  occurs exactly k times in x?
2 vector<int> num;
3 int a, b, d, k;
4 int DP[12][12][2];
5 /// DP[p][c][f] = Number of valid numbers <= b from this state
6 /// p = current position from left side (zero based)
7 /// c = number of times we have placed the digit d so far
8 /// f = the number we are building has already become smaller than b? [0 =
  no, 1 = yes]
9
10 int call(int pos, int cnt, int f){
11     if(cnt > k) return 0;
12
13     if(pos == num.size()){
14         if(cnt == k) return 1;
15         return 0;
16     }
17
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
51 return maior;
52 }

```

3.7. Longest Increasing Subsequence

```

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 ansj = -1;
7         while(l <= r){
8             int mid = (l+r)/2;
9             // OBS: PARA >= TROCAR SINAL EMBAIXO POR <=
10            if(arr[i] < lis[mid]){
11                r = mid - 1;
12                ansj = mid;
13            }
14            else l = mid + 1;
15        }
16        if(ansj == -1){
17            // se arr[i] e maior que todos
18            lis.push_back(arr[i]);
19        }
20        else {
21            lis[ansj] = arr[i];
22        }
23    }
24
25    return lis.size();
26 }

```

3.8. 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 }

```

3.9. Catalan

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

3.10. Catalan

```

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) = somatorio(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 // Formula acima
10
11 // Returns value of Binomial Coefficient C(n, k)
12
13 int binomialCoeff(int n, int k) {
14     int res = 1;
15
16     // Since C(n, k) = C(n, n-k)
17     if (k > n - k)
18         k = n - k;
19
20     // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
21     for (int i = 0; i < k; ++i) {
22         res *= (n - i);
23         res /= (i + 1);
24     }
25
26     return res;
27 }
28
29 // A Binomial coefficient based function to find nth catalan
30 // number in O(n) time
31 int catalan(int n) {
32     // Calculate value of 2nCn
33     int c = binomialCoeff(2*n, n);
34
35     // return 2nCn/(n+1)
36     return c/(n+1);
37 }

```

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 substituido
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. 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 }

```

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 seguem 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;

```

```

3 double diffY = y2 - y1;
4 if ((diffX == 0) && (diffY == 0)) {
5     diffX = pointX - x1;
6     diffY = pointY - y1;
7     //se os dois sao pontos
8     return hypot(pointX - x1, pointY - y1);
9 }
10 double t = ((pointX - x1) * diffX + (pointY - y1) * diffY) /
11           (diffX * diffX + diffY * diffY);
12 if (t < 0) {
13     //point is nearest to the first point i.e x1 and y1
14     // Ex:
15     // cord do pto na reta = pto inicial(x1,y1);
16     *ptox = x1, *ptoy = y1;
17     diffX = pointX - x1;
18     diffY = pointY - y1;
19 } else if (t > 1) {
20     //point is nearest to the end point i.e x2 and y2
21     // Ex:
22     // cord do pto na reta = pto final(x2,y2);
23     *ptox = x2, *ptoy = y2;
24     diffX = pointX - x2;
25     diffY = pointY - y2;
26 } else {
27     //if perpendicular line intersect the line segment.
28     // pto nao esta mais proximo de uma das bordas do segmento
29     // Ex:
30     //
31     // |
32     // | (Ângulo Reto)
33     //
34     // cord x do pto na reta = (x1 + t * diffX)
35     // cord y do pto na reta = (y1 + t * diffY)
36     *ptox = (x1 + t * diffX), *ptoy = (y1 + t * diffY);
37     diffX = pointX - (x1 + t * diffX);
38     diffY = pointY - (y1 + t * diffY);
39 }
40 //returning shortest distance
41 return sqrt(diffX * diffX + diffY * diffY);

```

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->y == it2->y && it->y <= p.y && p.y
84             <= it2->y)
85             return true;
86         return false;
87     }

```

```

86 if(itl == lower.end() || itu == upper.end()) {
87     return false;
88 }
89
90 auto ol = itl, ou = itu;
91 ol--, ou--;
92 if(cross(*ol, *itl, p) >= 0 && cross(*ou, *itu, p) <= 0)
93     return true;
94
95 auto it = lower_bound(arr.begin(), arr.end(), lo);
96 auto it2 = lower_bound(arr.begin(), arr.end(), hi);
97 it2--;
98 if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y <=
99     it2->y)
100     return true;
101
102 return false;
103 }
104
105 signed main () {
106
107     ios_base::sync_with_stdio(false);
108     cin.tie(NULL);
109
110     double n, m, k;
111
112     cin >> n >> m >> k;
113
114     vector<pto> arr(n);
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 poligono 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&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
10 /* PODE SER RETIRADO

```

```

9 int onSegmentNotBorda(Point p, Point q, Point r) {
10     if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) && q.y <= max(p.y, r.y)
11         && q.y >= min(p.y, r.y))
12         return true;
13     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y < max(p.y, r.y)
14         && q.y > min(p.y, r.y))
15         return true;
16     return false;
17 }
18 // To find orientation of ordered triplet (p, q, r).
19 // The function returns following values
20 // 0 --> p, q and r are colinear
21 // 1 --> Clockwise
22 // 2 --> Counterclockwise
23 int orientation(Point p, Point q, Point r) {
24     int val = (q.y - p.y) * (r.x - q.x) -
25             (q.x - p.x) * (r.y - q.y);
26     if (val == 0) return 0; // colinear
27     return (val > 0)? 1: 2; // clock or counterclock wise
28 }
29 // The main function that returns true if line segment 'p1p2'
30 // and 'q1q2' intersect.
31 int doIntersect(Point p1, Point p2, Point q1, Point q2) {
32     // Find the four orientations needed for general and
33     // special cases
34     int o1 = orientation(p1, p2, q1);
35     int o2 = orientation(p1, p2, q2);
36     int o3 = orientation(q1, q2, p1);
37     int o4 = orientation(q1, q2, p2);
38
39     // General case
40     if (o1 != o2 && o3 != o4) return 2;
41
42     /* PODE SER RETIRADO
43     if (o1 == o2 && o2 == o3 && o3 == o4 && o4 == 0) {
44         //INTERCEPTAM EM RETA
45         if (onSegmentNotBorda(p1, q1, p2) || onSegmentNotBorda(p1, q2, p2)) return 1;
46         if (onSegmentNotBorda(q1, p1, q2) || onSegmentNotBorda(q1, p2, q2)) return 1;
47     }
48     */
49     // Special Cases (INTERCEPTAM EM PONTO)
50     // p1, p2 and q1 are colinear and q1 lies on segment p1p2
51     if (o1 == 0 && onSegment(p1, q1, p2)) return 2;
52     // p1, p2 and q1 are colinear and q2 lies on segment p1p2
53     if (o1 == 0 && onSegment(p1, q2, p2)) return 2;
54     // q1, q2 and p1 are colinear and p1 lies on segment q1q2
55     if (o3 == 0 && onSegment(q1, p1, q2)) return 2;
56     // q1, q2 and p2 are colinear and p2 lies on segment q1q2
57     if (o4 == 0 && onSegment(q1, p2, q2)) return 2;
58     return false; // Doesn't fall in any of the above cases
59 }
60 // OBS: SE (C2/A2 == C1/A1) SÃO COLINEARES

```

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 };

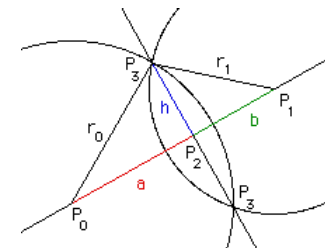
```

```

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

```



```

52 double dis_point_segment(const Point p, const Point s, const Point t) {
53     if(sgn(dot(p-s, t-s)) < 0)
54         return (p-s).norm();
55     if(sgn(dot(p-t, s-t)) < 0)
56         return (p-t).norm();
57     return abs(det(s-p, t-p) / dist(s, t));
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++) {
18                 if((int)adj[i].size() - in_degree[i] == 1) {
19                     if(src != -1)
20                         return make_pair(false, pair<int, int>());
21                     src = i;
22                 } else if((int)adj[i].size() - in_degree[i] == -1) {
23                     if(dest != -1)
24                         return make_pair(false, pair<int, int>());
25                     dest = i;
26                 } else if(abs((int)adj[i].size() - in_degree[i]) > 1)
27                     return make_pair(false, pair<int, int>());
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             return make_pair(false, pair<int, int>());
34         }
35     }
36     // Builds the path/tour for directed graphs.
37     void build(const int u, vector<int> &tour, vector<vector<edge>> &adj,
38         vector<bool> &used) {
39         while(!adj[u].empty()) {
40             const edge e = adj[u].back();
41             if(!used[e.id]) {
42                 used[e.id] = true;
43                 adj[u].pop_back();
44                 build(e.v, tour, adj, used);
45             } else
46                 adj[u].pop_back();
47         }
48         tour.push_back(u);
49     }
50 }

```

```

51 // Auxiliary function to build the eulerian tour/path.
52 vector<int> set_build(vector<vector<edge>> &adj, const int E, const int
53 first) {
54     vector<int> path;
55     vector<bool> used(E + 3);
56
57     build(first, path, adj, used);
58
59     for(int i = 0; i < adj.size(); i++)
60         // if there are some remaining edges, it's not possible to build the
61         // tour.
62         if(adj[i].size())
63             return vector<int>();
64
65     reverse(path.begin(), path.end());
66     return path;
67 }
68
69 // All vertices v should have in_degree[v] == out_degree[v]. It must not
70 // contain a specific
71 // start and end vertices.
72 // Time complexity: O(V * (log V) + E)
73 bool has_euler_tour_directed(const vector<vector<edge>> &adj, const
74     vector<int> &in_degree) {
75     const pair<bool, pair<int, int>> aux = detail::check_both_directed(adj,
76         in_degree);
77     const bool valid = aux.first;
78     const int src = aux.second.first;
79     const int dest = aux.second.second;
80     return (valid && src == -1 && dest == -1);
81 }
82
83 // A directed graph has an eulerian path/tour if has:
84 // - One vertex v such that out_degree[v] - in_degree[v] == 1
85 // - One vertex v such that in_degree[v] - out_degree[v] == 1
86 // - The remaining vertices v such that in_degree[v] == out_degree[v]
87 // or
88 // - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR
89
90 // Returns a boolean value that indicates whether there's a path or not.
91 // If there's a valid path it also returns two numbers: the source and
92 // the destination.
93 // If the source and destination can be an arbitrary vertex it will
94 // return the pair (-1, -1)
95 // for the source and destination (it means the contains an eulerian
96 // tour).
97 // Time complexity: O(V + E)
98 pair<bool, pair<int, int>> has_euler_path_directed(const
99     vector<vector<edge>> &adj, const vector<int> &in_degree) {
100     return detail::check_both_directed(adj, in_degree);
101 }
102
103 // Returns the euler path. If the graph doesn't have an euler path it
104 // returns an empty vector.
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) {

```

```

103     const pair<bool, pair<int, int>> aux = has_euler_path_directed(adj,
104         in_degree);
105     const bool valid = aux.first;
106     const int src = aux.second.first;
107     const int dest = aux.second.second;
108
109     if(!valid)
110         return vector<int>();
111
112     int first;
113     if(src != -1)
114         first = src;
115     else {
116         first = 0;
117         while(adj[first].empty())
118             first++;
119     }
120     return detail::set_build(adj, E, first);
121 }
122
123 /// Returns the euler tour. If the graph doesn't have an euler tour it
124 /// returns an empty vector.
125 /// Time Complexity: O(V + E)
126 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
127 vector<int> get_euler_tour_directed(const int E, vector<vector<edge>>
128     &adj, const vector<int> &in_degree) {
129     const bool valid = has_euler_tour_directed(adj, in_degree);
130
131     if(!valid)
132         return vector<int>();
133
134     int first = 0;
135     while(adj[first].empty())
136         first++;
137
138     return detail::set_build(adj, E, first);
139 }
140
141 // The graph has a tour that passes to every edge exactly once and gets
142 // back to the first edge on the tour.
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>
162     &degree) {
163     vector<int> odd_degree;
164     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(),
171             odd_degree.back()));
172     else
173         return make_pair(false, pair<int, int>());
174 }
175
176 vector<int> get_euler_tour_undirected(const int E, const vector<int>
177     &degree, vector<vector<edge>> &adj) {
178     if(!has_euler_tour_undirected(degree))
179         return vector<int>();
180
181     int first = 0;
182     while(adj[first].empty())
183         first++;
184
185     return detail::set_build(adj, E, first);
186 }
187
188 /// Returns the euler tour. If the graph doesn't have an euler tour it
189 /// returns an empty vector.
190 /// Time Complexity: O(V + E)
191 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
192 vector<int> get_euler_path_undirected(const int E, const vector<int>
193     &degree, vector<vector<edge>> &adj) {
194     auto aux = has_euler_path_undirected(degree);
195     const bool valid = aux.first;
196     const int x = aux.second.first;
197     const int y = aux.second.second;
198
199     if(!valid)
200         return vector<int>();
201
202     int first;
203     if(x != -1) {
204         first = x;
205         adj[x].emplace_back(y, E + 1);
206         adj[y].emplace_back(x, E + 1);
207     } else {
208         first = 0;
209         while(adj[first].empty())
210             first++;
211     }
212
213     vector<int> ans = detail::set_build(adj, E, first);
214     reverse(ans.begin(), ans.end());
215     if(x != -1)
216         ans.pop_back();
217     return ans;
218 }

```

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
7     bool operator<(const edge &a) const {
8         return weight < a.weight;
9     }
10 };
11
12 /// Works to find the shortest path with negative edges.
13 /// Also detects cycles.
14 ///
15 /// 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
44     // Check for NEGATIVE-WEIGHT CYCLES.
45     // The above step guarantees shortest distances if graph doesn't contain
46     // negative weight cycle.
47     // If we get a shorter path, then there is a cycle.
48     bool is_cycle = false;
49     int vert_in_cycle;
50     for (int i = 0; i < E; i++) {
51         int u = edges[i].src;
52         int v = edges[i].dest;
53         int weight = edges[i].weight;
54         if (dist[u] != INF && dist[u] + weight < dist[v]) {
55             is_cycle = true;
56             pai[v] = u;
57             vert_in_cycle = v;
58         }
59     }
60
61     if(is_cycle) {
62         for(int i = 0; i < n; i++)
63             vert_in_cycle = pai[vert_in_cycle];
64
65         vector<int> cycle;
66         for(int v = vert_in_cycle; (v != vert_in_cycle || cycle.size() <= 1) ; v
67             = pai[v])
68             cycle.pb(v);
69
70         reverse(cycle.begin(), cycle.end());
71
72         for(int x: cycle) {
73             cout << x + 1 << ' ';
74         }
75         cout << cycle.front() + 1 << endl;
76         return true;
77     } else
78         return false;
79 }

```

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. 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

```

5.6. 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:  $O(n * \log(n))$ 
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 };

```

5.7. 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 //

```

```

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:  $O(\text{alphabet.size()} ^ n * \log_2(\text{alphabet.size()} ^ n))$ 
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

```

5.8. 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.

5.9. 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     void _compute(const int src, const vector<vector<pair<int, int>>> &adj) {
13         _dist.resize(this->n, INF);
14         vector<bool> vis(this->n, false);
15
16         if (CREATE_GRAPH) {
17             parent.resize(this->n);
18
19             for (int i = 0; i < this->n; i++)

```

```

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

```

```

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

```

```

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 graph
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 = 2e18;
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        }
67    };
68
69    public:
70        int max_flow() {
71            _max_flow = 0;
72            while (true) {
73                vector<int> level(n, -1);
74                queue<int> q;
75                q.push(src);
76                level[src] = 0;
77                while (!q.empty()) {
78                    int u = q.front(); q.pop();
79                    for (const int idx : adj[u]) {
80                        const Edge &e = edges[idx];
81                        if (e.v == sink && e.flow < e.cap)
82                            level[sink] = level[u] + 1;
83                        else if (e.v < sink && e.flow < e.cap && level[e.v] == -1)
84                            level[e.v] = level[u] + 1;
85                        q.push(e.v);
86                    }
87                }
88                if (level[sink] == -1)
89                    return _max_flow;
90                _max_flow += dfs(src, sink, level);
91            }
92        }
93
94        void add_edge(int u, int v, int cap) {
95            add_edge(u, v, cap, 0);
96        }
97
98        void add_reverse_edge(int u, int v, int cap) {
99            add_edge(v, u, cap, 1);
100        }
101
102        void reset() {
103            edges.clear();
104            adj.clear();
105            level.clear();
106            ptr.clear();
107            _max_flow = 0;
108            COMPUTED = false;
109        }
110
111        bool is_computed() const {
112            return COMPUTED;
113        }
114
115        void set_computed() {
116            COMPUTED = true;
117        }
118
119        void print_flow_table() const {
120            _flow_table();
121        }
122
123        void print_flow() const {
124            for (int u = 1; u < sink; ++u)
125                for (const int idx : adj[u])
126                    if (!(idx & 1))
127                        cout << u << " -> " << edges[idx].v << " " << edges[idx].flow << " ";
128            cout << endl;
129        }
130
131        void print_matching() const {
132            _min_edge_cover();
133            for (const auto &p : ans)
134                cout << p.first << " -> " << p.second << " ";
135            cout << endl;
136        }
137
138        void print_covered() const {
139            _min_edge_cover();
140            for (int u = 1; u < sink; ++u)
141                cout << u << " covered: " << covered[u] << " ";
142            cout << endl;
143        }
144
145        void print_flow_table() const {
146            _flow_table();
147            for (int u = 0; u <= sink; ++u)
148                for (const int v = 0; v <= sink; ++v)
149                    cout << table[u][v] << " ";
150            cout << endl;
151        }
152
153        void print_flow_table() {
154            _flow_table();
155            for (int u = 0; u <= sink; ++u)
156                for (const int v = 0; v <= sink; ++v)
157                    cout << table[u][v] << " ";
158            cout << endl;
159        }
160
161        void print_flow_table() const {
162            _flow_table();
163            for (int u = 0; u <= sink; ++u)
164                for (const int v = 0; v <= sink; ++v)
165                    cout << table[u][v] << " ";
166            cout << endl;
167        }
168
169        void print_flow_table() {
170            _flow_table();
171            for (int u = 0; u <= sink; ++u)
172                for (const int v = 0; v <= sink; ++v)
173                    cout << table[u][v] << " ";
174            cout << endl;
175        }
176
177        void print_flow_table() const {
178            _flow_table();
179            for (int u = 0; u <= sink; ++u)
180                for (const int v = 0; v <= sink; ++v)
181                    cout << table[u][v] << " ";
182            cout << endl;
183        }
184
185        void print_flow_table() {
186            _flow_table();
187            for (int u = 0; u <= sink; ++u)
188                for (const int v = 0; v <= sink; ++v)
189                    cout << table[u][v] << " ";
190            cout << endl;
191        }
192
193        void print_flow_table() const {
194            _flow_table();
195            for (int u = 0; u <= sink; ++u)
196                for (const int v = 0; v <= sink; ++v)
197                    cout << table[u][v] << " ";
198            cout << endl;
199        }
200
201        void print_flow_table() {
202            _flow_table();
203            for (int u = 0; u <= sink; ++u)
204                for (const int v = 0; v <= sink; ++v)
205                    cout << table[u][v] << " ";
206            cout << endl;
207        }
208
209        void print_flow_table() const {
210            _flow_table();
211            for (int u = 0; u <= sink; ++u)
212                for (const int v = 0; v <= sink; ++v)
213                    cout << table[u][v] << " ";
214            cout << endl;
215        }
216
217        void print_flow_table() {
218            _flow_table();
219            for (int u = 0; u <= sink; ++u)
220                for (const int v = 0; v <= sink; ++v)
221                    cout << table[u][v] << " ";
222            cout << endl;
223        }
224
225        void print_flow_table() const {
226            _flow_table();
227            for (int u = 0; u <= sink; ++u)
228                for (const int v = 0; v <= sink; ++v)
229                    cout << table[u][v] << " ";
230            cout << endl;
231        }
232
233        void print_flow_table() {
234            _flow_table();
235            for (int u = 0; u <= sink; ++u)
236                for (const int v = 0; v <= sink; ++v)
237                    cout << table[u][v] << " ";
238            cout << endl;
239        }
240
241        void print_flow_table() const {
242            _flow_table();
243            for (int u = 0; u <= sink; ++u)
244                for (const int v = 0; v <= sink; ++v)
245                    cout << table[u][v] << " ";
246            cout << endl;
247        }
248
249        void print_flow_table() {
250            _flow_table();
251            for (int u = 0; u <= sink; ++u)
252                for (const int v = 0; v <= sink; ++v)
253                    cout << table[u][v] << " ";
254            cout << endl;
255        }
256
257        void print_flow_table() const {
258            _flow_table();
259            for (int u = 0; u <= sink; ++u)
260                for (const int v = 0; v <= sink; ++v)
261                    cout << table[u][v] << " ";
262            cout << endl;
263        }
264
265        void print_flow_table() {
266            _flow_table();
267            for (int u = 0; u <= sink; ++u)
268                for (const int v = 0; v <= sink; ++v)
269                    cout << table[u][v] << " ";
270            cout << endl;
271        }
272
273        void print_flow_table() const {
274            _flow_table();
275            for (int u = 0; u <= sink; ++u)
276                for (const int v = 0; v <= sink; ++v)
277                    cout << table[u][v] << " ";
278            cout << endl;
279        }
280
281        void print_flow_table() {
282            _flow_table();
283            for (int u = 0; u <= sink; ++u)
284                for (const int v = 0; v <= sink; ++v)
285                    cout << table[u][v] << " ";
286            cout << endl;
287        }
288
289        void print_flow_table() const {
290            _flow_table();
291            for (int u = 0; u <= sink; ++u)
292                for (const int v = 0; v <= sink; ++v)
293                    cout << table[u][v] << " ";
294            cout << endl;
295        }
296
297        void print_flow_table() {
298            _flow_table();
299            for (int u = 0; u <= sink; ++u)
300                for (const int v = 0; v <= sink; ++v)
301                    cout << table[u][v] << " ";
302            cout << endl;
303        }
304
305        void print_flow_table() const {
306            _flow_table();
307            for (int u = 0; u <= sink; ++u)
308                for (const int v = 0; v <= sink; ++v)
309                    cout << table[u][v] << " ";
310            cout << endl;
311        }
312
313        void print_flow_table() {
314            _flow_table();
315            for (int u = 0; u <= sink; ++u)
316                for (const int v = 0; v <= sink; ++v)
317                    cout << table[u][v] << " ";
318            cout << endl;
319        }
320
321        void print_flow_table() const {
322            _flow_table();
323            for (int u = 0; u <= sink; ++u)
324                for (const int v = 0; v <= sink; ++v)
325                    cout << table[u][v] << " ";
326            cout << endl;
327        }
328
329        void print_flow_table() {
330            _flow_table();
331            for (int u = 0; u <= sink; ++u)
332                for (const int v = 0; v <= sink; ++v)
333                    cout << table[u][v] << " ";
334            cout << endl;
335        }
336
337        void print_flow_table() const {
338            _flow_table();
339            for (int u = 0; u <= sink; ++u)
340                for (const int v = 0; v <= sink; ++v)
341                    cout << table[u][v] << " ";
342            cout << endl;
343        }
344
345        void print_flow_table() {
346            _flow_table();
347            for (int u = 0; u <= sink; ++u)
348                for (const int v = 0; v <= sink; ++v)
349                    cout << table[u][v] << " ";
350            cout << endl;
351        }
352
353        void print_flow_table() const {
354            _flow_table();
355            for (int u = 0; u <= sink; ++u)
356                for (const int v = 0; v <= sink; ++v)
357                    cout << table[u][v] << " ";
358            cout << endl;
359        }
360
361        void print_flow_table() {
362            _flow_table();
363            for (int u = 0; u <= sink; ++u)
364                for (const int v = 0; v <= sink; ++v)
365                    cout << table[u][v] << " ";
366            cout << endl;
367        }
368
369        void print_flow_table() const {
370            _flow_table();
371            for (int u = 0; u <= sink; ++u)
372                for (const int v = 0; v <= sink; ++v)
373                    cout << table[u][v] << " ";
374            cout << endl;
375        }
376
377        void print_flow_table() {
378            _flow_table();
379            for (int u = 0; u <= sink; ++u)
380                for (const int v = 0; v <= sink; ++v)
381                    cout << table[u][v] << " ";
382            cout << endl;
383        }
384
385        void print_flow_table() const {
386            _flow_table();
387            for (int u = 0; u <= sink; ++u)
388                for (const int v = 0; v <= sink; ++v)
389                    cout << table[u][v] << " ";
390            cout << endl;
391        }
392
393        void print_flow_table() {
394            _flow_table();
395            for (int u = 0; u <= sink; ++u)
396                for (const int v = 0; v <= sink; ++v)
397                    cout << table[u][v] << " ";
398            cout << endl;
399        }
400
401        void print_flow_table() const {
402            _flow_table();
403            for (int u = 0; u <= sink; ++u)
404                for (const int v = 0; v <= sink; ++v)
405                    cout << table[u][v] << " ";
406            cout << endl;
407        }
408
409        void print_flow_table() {
410            _flow_table();
411            for (int u = 0; u <= sink; ++u)
412                for (const int v = 0; v <= sink; ++v)
413                    cout << table[u][v] << " ";
414            cout << endl;
415        }
416
417        void print_flow_table() const {
418            _flow_table();
419            for (int u = 0; u <= sink; ++u)
420                for (const int v = 0; v <= sink; ++v)
421                    cout << table[u][v] << " ";
422            cout << endl;
423        }
424
425        void print_flow_table() {
426            _flow_table();
427            for (int u = 0; u <= sink; ++u)
428                for (const int v = 0; v <= sink; ++v)
429                    cout << table[u][v] << " ";
430            cout << endl;
431        }
432
433        void print_flow_table() const {
434            _flow_table();
435            for (int u = 0; u <= sink; ++u)
436                for (const int v = 0; v <= sink; ++v)
437                    cout << table[u][v] << " ";
438            cout << endl;
439        }
440
441        void print_flow_table() {
442            _flow_table();
443            for (int u = 0; u <= sink; ++u)
444                for (const int v = 0; v <= sink; ++v)
445                    cout << table[u][v] << " ";
446            cout << endl;
447        }
448
449        void print_flow_table() const {
450            _flow_table();
451            for (int u = 0; u <= sink; ++u)
452                for (const int v = 0; v <= sink; ++v)
453                    cout << table[u][v] << " ";
454            cout << endl;
455        }
456
457        void print_flow_table() {
458            _flow_table();
459            for (int u = 0; u <= sink; ++u)
460                for (const int v = 0; v <= sink; ++v)
461                    cout << table[u][v] << " ";
462            cout << endl;
463        }
464
465        void print_flow_table() const {
466            _flow_table();
467            for (int u = 0; u <= sink; ++u)
468                for (const int v = 0; v <= sink; ++v)
469                    cout << table[u][v] << " ";
470            cout << endl;
471        }
472
473        void print_flow_table() {
474            _flow_table();
475            for (int u = 0; u <= sink; ++u)
476                for (const int v = 0; v <= sink; ++v)
477                    cout << table[u][v] << " ";
478            cout << endl;
479        }
480
481        void print_flow_table() const {
482            _flow_table();
483            for (int u = 0; u <= sink; ++u)
484                for (const int v = 0; v <= sink; ++v)
485                    cout << table[u][v] << " ";
486            cout << endl;
487        }
488
489        void print_flow_table() {
490            _flow_table();
491            for (int u = 0; u <= sink; ++u)
492                for (const int v = 0; v <= sink; ++v)
493                    cout << table[u][v] << " ";
494            cout << endl;
495        }
496
497        void print_flow_table() const {
498            _flow_table();
499            for (int u = 0; u <= sink; ++u)
500                for (const int v = 0; v <= sink; ++v)
501                    cout << table[u][v] << " ";
502            cout << endl;
503        }
504
505        void print_flow_table() {
506            _flow_table();
507            for (int u = 0; u <= sink; ++u)
508                for (const int v = 0; v <= sink; ++v)
509                    cout << table[u][v] << " ";
510            cout << endl;
511        }
512
513        void print_flow_table() const {
514            _flow_table();
515            for (int u = 0; u <= sink; ++u)
516                for (const int v = 0; v <= sink; ++v)
517                    cout << table[u][v] << " ";
518            cout << endl;
519        }
520
521        void print_flow_table() {
522            _flow_table();
523            for (int u = 0; u <= sink; ++u)
524                for (const int v = 0; v <= sink; ++v)
525                    cout << table[u][v] << " ";
526            cout << endl;
527        }
528
529        void print_flow_table() const {
530            _flow_table();
531            for (int u = 0; u <= sink; ++u)
532                for (const int v = 0; v <= sink; ++v)
533                    cout << table[u][v] << " ";
534            cout << endl;
535        }
536
537        void print_flow_table() {
538            _flow_table();
539            for (int u = 0; u <= sink; ++u)
540                for (const int v = 0; v <= sink; ++v)
541                    cout << table[u][v] << " ";
542            cout << endl;
543        }
544
545        void print_flow_table() const {
546            _flow_table();
547            for (int u = 0; u <= sink; ++u)
548                for (const int v = 0; v <= sink; ++v)
549                    cout << table[u][v] << " ";
550            cout << endl;
551        }
552
553        void print_flow_table() {
554            _flow_table();
555            for (int u = 0; u <= sink; ++u)
556                for (const int v = 0; v <= sink; ++v)
557                    cout << table[u][v] << " ";
558            cout << endl;
559        }
560
561        void print_flow_table() const {
562            _flow_table();
563            for (int u = 0; u <= sink; ++u)
564                for (const int v = 0; v <= sink; ++v)
565                    cout << table[u][v] << " ";
566            cout << endl;
567        }
568
569        void print_flow_table() {
570            _flow_table();
571            for (int u = 0; u <= sink; ++u)
572                for (const int v = 0; v <= sink; ++v)
573                    cout << table[u][v] << " ";
574            cout << endl;
575        }
576
577        void print_flow_table() const {
578            _flow_table();
579            for (int u = 0; u <= sink; ++u)
580                for (const int v = 0; v <= sink; ++v)
581                    cout << table[u][v] << " ";
582            cout << endl;
583        }
584
585        void print_flow_table() {
586            _flow_table();
587            for (int u = 0; u <= sink; ++u)
588                for (const int v = 0; v <= sink; ++v)
589                    cout << table[u][v] << " ";
590            cout << endl;
591        }
592
593        void print_flow_table() const {
594            _flow_table();
595            for (int u = 0; u <= sink; ++u)
596                for (const int v = 0; v <= sink; ++v)
597                    cout << table[u][v] << " ";
598            cout << endl;
599        }
600
601        void print_flow_table() {
602            _flow_table();
603            for (int u = 0; u <= sink; ++u)
604                for (const int v = 0; v <= sink; ++v)
605                    cout << table[u][v] << " ";
606            cout << endl;
607        }
608
609        void print_flow_table() const {
610            _flow_table();
611            for (int u = 0; u <= sink; ++u)
612                for (const int v = 0; v <= sink; ++v)
613                    cout << table[u][v] << " ";
614            cout << endl;
615        }
616
617        void print_flow_table() {
618            _flow_table();
619            for (int u = 0; u <= sink; ++u)
620                for (const int v = 0; v <= sink; ++v)
621                    cout << table[u][v] << " ";
622            cout << endl;
623        }
624
625        void print_flow_table() const {
626            _flow_table();
627            for (int u = 0; u <= sink; ++u)
628                for (const int v = 0; v <= sink; ++v)
629                    cout << table[u][v] << " ";
630            cout << endl;
631        }
632
633        void print_flow_table() {
634            _flow_table();
635            for (int u = 0; u <= sink; ++u)
636                for (const int v = 0; v <= sink; ++v)
637                    cout << table[u][v] << " ";
638            cout << endl;
639        }
640
641        void print_flow_table() const {
642            _flow_table();
643            for (int u = 0; u <= sink; ++u)
644                for (const int v = 0; v <= sink; ++v)
645                    cout << table[u][v] << " ";
646            cout << endl;
647        }
648
649        void print_flow_table() {
650            _flow_table();
651            for (int u = 0; u <= sink; ++u)
652                for (const int v = 0; v <= sink; ++v)
653                    cout << table[u][v] << " ";
654            cout << endl;
655        }
656
657        void print_flow_table() const {
658            _flow_table();
659            for (int u = 0; u <= sink; ++u)
660                for (const int v = 0; v <= sink; ++v)
661                    cout << table[u][v] << " ";
662            cout << endl;
663        }
664
665        void print_flow_table() {
666            _flow_table();
667            for (int u = 0; u <= sink; ++u)
668                for (const int v = 0; v <= sink; ++v)
669                    cout << table[u][v] << " ";
670            cout << endl;
671        }
672
673        void print_flow_table() const {
674            _flow_table();
675            for (int u = 0; u <= sink; ++u)
676                for (const int v = 0; v <= sink; ++v)
677                    cout << table[u][v] << " ";
678            cout << endl;
679        }
680
681        void print_flow_table() {
682            _flow_table();
683            for (int u = 0; u <= sink; ++u)
684                for (const int v = 0; v <= sink; ++v)
685                    cout << table[u][v] << " ";
686            cout << endl;
687        }
688
689        void print_flow_table() const {
690            _flow_table();
691            for (int u = 0; u <= sink; ++u)
692                for (const int v = 0; v <= sink; ++v)
693                    cout << table[u][v] << " ";
694            cout << endl;
695        }
696
697        void print_flow_table() {
698            _flow_table();
699            for (int u = 0; u <= sink; ++u)
700                for (const int v = 0; v <= sink; ++v)
701                    cout << table[u][v] << " ";
702            cout << endl;
703        }
704
705        void print_flow_table() const {
706            _flow_table();
707            for (int u = 0; u <= sink; ++u)
708                for (const int v = 0; v <= sink; ++v)
709                    cout << table[u][v] << " ";
710            cout << endl;
711        }
712
713        void print_flow_table() {
714            _flow_table();
715            for (int u = 0; u <= sink; ++u)
716                for (const int v = 0; v <= sink; ++v)
717                    cout << table[u][v] << " ";
718            cout << endl;
719        }
720
721        void print_flow_table() const {
722            _flow_table();
723            for (int u = 0; u <= sink; ++u)
724                for (const int v = 0; v <= sink; ++v)
725                    cout << table[u][v] << " ";
726            cout << endl;
727        }
728
729        void print_flow_table() {
730            _flow_table();
731            for (int u = 0; u <= sink; ++u)
732                for (const int v = 0; v <= sink; ++v)
733                    cout << table[u][v] << " ";
734            cout << endl;
735        }
736
737        void print_flow_table() const {
738            _flow_table();
739            for (int u = 0; u <= sink; ++u)
740                for (const int v = 0; v <= sink; ++v)
741                    cout << table[u][v] << " ";
742            cout << endl;
743        }
744
745        void print_flow_table() {
746            _flow_table();
747            for (int u = 0; u <= sink; ++u)
748                for (const int v = 0; v <= sink; ++v)
749                    cout << table[u][v] << " ";
750            cout << endl;
751        }
752
753        void print_flow_table() const {
754            _flow_table();
755            for (int u = 0; u <= sink; ++u)
756                for (const int v = 0; v <= sink; ++v)
757                    cout << table[u][v] << " ";
758            cout << endl;
759        }
760
761        void print_flow_table() {
762            _flow_table();
763            for (int u = 0; u <= sink; ++u)
764                for (const int v = 0; v <= sink; ++v)
765                    cout << table[u][v] << " ";
766            cout << endl;
767        }
768
769        void print_flow_table() const {
770            _flow_table();
771            for (int u = 0; u <= sink; ++u)
772                for (const int v = 0; v <= sink; ++v)
773                    cout << table[u][v] << " ";
774            cout << endl;
775        }
776
777        void print_flow_table() {
778            _flow_table();
779            for (int u = 0; u <= sink; ++u)
780                for (const int v = 0; v <= sink; ++v)
781                    cout << table[u][v] << " ";
782            cout << endl;
783        }
784
785        void print_flow_table() const {
786            _flow_table();
787            for (int u = 0; u <= sink; ++u)
788                for (const int v = 0; v <= sink; ++v)
789                    cout << table[u][v] << " ";
790            cout << endl;
791        }
792
793        void print_flow_table() {
794            _flow_table();
795            for (int u = 0; u <= sink; ++u)
796                for (const int v = 0; v <= sink; ++v)
797                    cout << table[u][v] << " ";
798            cout << endl;
799        }
800
801        void print_flow_table() const {
802            _flow_table();
803            for (int u = 0; u <= sink; ++u)
804                for (const int v = 0; v <= sink; ++v)
805                    cout << table[u][v] << " ";
806            cout << endl;
807        }
808
809        void print_flow_table() {
810            _flow_table();
811            for (int u = 0; u <= sink; ++u)
812                for (const int v = 0; v <= sink; ++v)
813                    cout << table[u][v] << " ";
814            cout << endl;
815        }
816
817        void print_flow_table() const {
818            _flow_table();
819            for (int u = 0; u <= sink; ++u)
820                for (const int v = 0; v <= sink; ++v)
821                    cout << table[u][v] << " ";
822            cout << endl;
823        }
824
825        void print_flow_table() {
826            _flow_table();
827            for (int u = 0; u <= sink; ++u)
828                for (const int v = 0; v <= sink; ++v)
829                    cout << table[u][v] << " ";
830            cout << endl;
831        }
832
833        void print_flow_table() const {
834            _flow_table();
835            for (int u = 0; u <= sink; ++u)
836                for (const int v = 0; v <= sink; ++v)
837                    cout << table[u][v] << " ";
838            cout << endl;
839        }
840
841        void print_flow_table() {
842            _flow_table();
843            for (int u = 0; u <= sink; ++u)
844                for (const int v = 0; v <= sink; ++v)
845                    cout << table[u][v] << " ";
846            cout << endl;
847        }
848
849        void print_flow_table() const {
850            _flow_table();
851            for (int u = 0; u <= sink; ++u)
852                for (const int v = 0; v <= sink; ++v)
853                    cout << table[u][v] << " ";
854            cout << endl;
855        }
856
857        void print_flow_table() {
858            _flow_table();
859            for (int u = 0; u <= sink; ++u)
860                for (const int v = 0; v <= sink; ++v)
861                    cout << table[u][v] << " ";
862            cout << endl;
863        }
864
865        void print_flow_table() const {
866            _flow_table();
867            for (int u = 0; u <= sink; ++u)
868                for (const int v = 0; v <= sink; ++v)
869                    cout << table[u][v] << " ";
870            cout << endl;
871        }
872
873        void print_flow_table() {
874            _flow_table();
875            for (int u = 0; u <= sink; ++u)
876                for (const int v = 0; v <= sink; ++v)
877                    cout << table[u][v] << " ";
878            cout << endl;
879        }
880
881        void print_flow_table() const {
882            _flow_table();
883            for (int u = 0; u <= sink; ++u)
884                for (const int v = 0; v <= sink; ++v)
885                    cout << table[u][v] << " ";
886            cout << endl;
887        }
888
889        void print_flow_table() {
890            _flow_table();
891            for (int u =
```

```

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();
193         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     /// Returns the edges from the minimum edge cover of the graph.
288     /// A minimum edge cover represents a set of edges such that each vertex
289     /// present in the graph is linked to at least one edge from this set.
290     ///
291     /// Time Complexity:  $O(V + E)$ 
292     vector<pair<int, int>> min_edge_cover() {
293         this->check_computed();
294         return this->_min_edge_cover();
295     }
296
297     /// Returns the maximum independent set for the graph.
298     /// An independent set represents a set of vertices such that they're not
299     /// adjacent to each other.
300     /// It is equal to the complement of the minimum vertex cover.
301     ///
302     /// Time Complexity:  $O(V + E)$ 
303     vector<int> max_ind_set(const int max_left) {
304         this->check_computed();
305         return this->_max_ind_set(max_left);
306     }
307
308     /// Returns the minimum vertex cover of a bipartite graph.
309     /// A minimum vertex cover represents a set of vertices such that each
310     /// edge of
311     /// the graph is incident to at least one vertex of the graph.
312     /// Pass the maximum index of a vertex on the left side as an argument.
313     ///
314     /// Time Complexity:  $O(V + E)$ 
315     vector<int> min_vertex_cover(const int max_left) {
316         this->check_computed();
317         return this->_min_vertex_cover(max_left);
318     }
319
320     /// Computes all paths from src to sink.
321     /// Add all edges from the original graph. Its weights should be equal to the

```

```

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

```

5.11. Dsu

```

1 class DSU {
2 public:
3     vector<int> root;
4     vector<int> sz;
5
6     DSU(int n) {
7         this->root.resize(n + 1);
8         iota(this->root.begin(), this->root.begin() + n + 1, 0ll);
9         this->sz.resize(n + 1, 1);
10    }
11
12    int Find(int x) {
13        if (this->root[x] == x)
14            return x;
15        return this->root[x] = this->Find(this->root[x]);
16    }
17
18    bool Union(int p, int q) {
19        p = this->Find(p), q = this->Find(q);
20
21        if (p == q)
22            return false;
23
24        if (this->sz[p] > this->sz[q]) {
25            this->root[q] = p;
26            this->sz[p] += this->sz[q];
27        } else {
28            this->root[p] = q;
29            this->sz[q] += this->sz[p];
30        }
31
32        return true;
33    }
34 };

```

5.12. 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 }

```

5.13. 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>
    &in_degree) {
30     queue<int> q;
31     vector<bool> vis(n + indexed_from);
32     for(int i = indexed_from; i < n + indexed_from; i++) {
33         if(in_degree[i] == 0) {
34             q.push(i);
35             vis[i] = true;
36         }
37     }
38
39     vector<int> process_order;
40     process_order.reserve(n + indexed_from);
41     while(!q.empty()) {
42         int u = q.front();
43         q.pop();
44
45         process_order.push_back(u);
46
47         if(--in_degree[nxt[u]] == 0) {
48             q.push(nxt[u]);
49             vis[nxt[u]] = true;
50         }
51     }
52
53     int cycle_cnt = 0;
54     for(int i = indexed_from; i < n + indexed_from; i++)
55         if(!vis[i])
56             compute_cycle(i, nxt, vis);
57
58     for(int i = (int)process_order.size() - 1; i >= 0; i--) {
59         int u = process_order[i];
60
61         this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
62         this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
63         this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
64     }
65 }
66
67 void allocate(int n, int indexed_from) {
68     this->cycle.resize(n + indexed_from);
69     this->first.resize(n + indexed_from);
70 }

```

```

71     this->in_cycle.resize(n + indexed_from, false);
72     this->cycle_id.resize(n + indexed_from, -1);
73     this->id_in_cycle.resize(n + indexed_from, -1);
74     this->near_in_cycle.resize(n + indexed_from);
75     this->id_near_cycle.resize(n + indexed_from);
76     this->cycle_dist.resize(n + indexed_from);
77 }
78
79 public:
80 Functional_Graph(int n, int indexed_from, vector<int> &nxt, vector<int>
    &in_degree) {
81     this->allocate(n, indexed_from);
82     this->build(n, indexed_from, nxt, in_degree);
83 }
84
85 // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
86
87 // number of cycles
88 int cycle_cnt = 0;
89 // Vertices present in the i-th cycle.
90 vector<vector<int>> cycle;
91 // first vertex of the i-th cycle
92 vector<int> first;
93
94 // The i-th vertex is present in any cycle?
95 vector<bool> in_cycle;
96 // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
    cycle.
97 vector<int> cycle_id;
98 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
    belong to any cycle.
99 vector<int> id_in_cycle;
100 // Represents the id of the nearest vertex present in a cycle.
101 vector<int> near_in_cycle;
102 // Represents the id of the nearest cycle.
103 vector<int> id_near_cycle;
104 // Distance to the nearest cycle.
105 vector<int> cycle_dist;
106 // Represent the id of the component of the vertex.
107 // Equal to id_near_cycle
108 vector<int> &comp = id_near_cycle;
109 };
110
111 class Functional_Graph {
112     // FOR UNDIRECTED GRAPH
113 private:
114     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis,
        vector<vector<int>> &adj) {
115         int id_cycle = cycle_cnt++;
116         int cur_id = 0;
117         this->first[id_cycle] = u;
118
119         while(!vis[u]) {
120             vis[u] = true;
121
122             this->cycle[id_cycle].push_back(u);
123             nxt[u] = find_nxt(u, vis, adj);
124             if(nxt[u] == -1)
125                 nxt[u] = this->first[id_cycle];
126
127             this->in_cycle[u] = true;
128             this->cycle_id[u] = id_cycle;
129             this->id_in_cycle[u] = cur_id;
130             this->near_in_cycle[u] = u;
131             this->id_near_cycle[u] = id_cycle;

```

```

132     this->cycle_dist[u] = 0;
133
134     u = nxt[u];
135     cur_id++;
136 }
137
138
139 int find_nxt(int u, vector<bool> &vis, vector<vector<int>> &adj) {
140     for(int v: adj[u])
141         if(!vis[v])
142             return v;
143     return -1;
144 }
145
146 // Time Complexity: O(V + E)
147 void build(int n, int indexed_from, vector<int> &degree,
    vector<vector<int>> &adj) {
148     queue<int> q;
149     vector<bool> vis(n + indexed_from, false);
150     vector<int> nxt(n + indexed_from);
151     for(int i = indexed_from; i < n + indexed_from; i++) {
152         if(adj[i].size() == 1) {
153             q.push(i);
154             vis[i] = true;
155         }
156     }
157
158     vector<int> process_order;
159     process_order.reserve(n + indexed_from);
160     while(!q.empty()) {
161         int u = q.front();
162         q.pop();
163
164         process_order.push_back(u);
165
166         nxt[u] = find_nxt(u, vis, adj);
167         if(--degree[nxt[u]] == 1) {
168             q.push(nxt[u]);
169             vis[nxt[u]] = true;
170         }
171     }
172
173     int cycle_cnt = 0;
174     for(int i = indexed_from; i < n + indexed_from; i++)
175         if(!vis[i])
176             compute_cycle(i, nxt, vis, adj);
177
178     for(int i = (int)process_order.size() - 1; i >= 0; i--) {
179         int u = process_order[i];
180
181         this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
182         this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
183         this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
184     }
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.
211 vector<vector<int>>> cycle;
212 // first vertex of the i-th cycle
213 vector<int> first;
214
215 // The i-th vertex is present in any cycle?
216 vector<bool> in_cycle;
217 // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
218 // cycle.
219 vector<int> cycle_id;
220 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
221 // belong to any cycle.
222 vector<int> id_in_cycle;
223 // Represents the id of the nearest vertex present in a cycle.
224 vector<int> near_in_cycle;
225 // Represents the id of the nearest cycle.
226 vector<int> id_near_cycle;
227 // Distance to the nearest cycle.
228 vector<int> cycle_dist;
229 // Represent the id of the component of the vertex.
230 // Equal to id_near_cycle
231 vector<int> &comp = id_near_cycle;
232 };

```

5.14. 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 }

```

```

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 }

```

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

```

5.16. 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

```

```

6  ///
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++) {
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 }

```

5.17. Kruskal

```

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) {}

```

```

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

```

5.18. Lca

```

1  // #define DIST
2  // #define COST
3  /// UNCOMMENT ALSO THE LINE BELOW FOR COST!
4
5  class LCA {
6  private:
7      int n;
8      // INDEXED from 0 or 1??
9      int indexed_from;
10     /// Store all log2 from 1 to n
11     vector<int> lg;
12     /// level of the i-th node (height)
13     vector<int> level;
14     /// matrix to store the ancestors of each node in power of 2 levels
15     vector<vector<int>> anc;
16
17 #ifndef 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     // int NEUTRAL_VALUE = INF; // MIN COST
24     // int combine(const int a, const int b) {return min(a, b);}
25     vector<vector<int>> cost;
26 #endif
27
28 private:
29     void allocate() {
30         // initializes a matrix [n][lg n] with -1
31         this->build_log_array();
32         this->anc.resize(n + 1, vector<int>(lg[n] + 1, -1));
33         this->level.resize(n + 1, -1);
34
35 #ifndef 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
46     for (int i = 2; i <= this->n; i++)
47         this->lg[i] = this->lg[i / 2] + 1;
48 }
49
50 void build_anc() {
51     for (int j = 1; j < anc.front().size(); j++)
52         for (int i = 0; i < anc.size(); i++)
53             if (this->anc[i][j - 1] != -1) {
54                 this->anc[i][j] = this->anc[this->anc[i][j - 1]][j - 1];
55 #ifndef COST
56                 this->cost[i][j] =
57                     combine(this->cost[i][j - 1], this->cost[this->anc[i][j - 1]][j -
58                         1]);
59 #endif
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
65     this->build_anc();
66 }
67
68 void dfs_LCA_weighted(const int u, const int p, const int l, const int d,
69                     const vector<vector<pair<int, int>>> &adj) {
70     this->level[u] = l;
71     this->anc[u][0] = p;
72 #ifndef DIST
73     this->dist[u] = d;
74 #endif
75
76     for (const pair<int, int> &x : adj[u]) {
77         int v = x.first, w = x.second;
78         if (v == p)
79             continue;
80 #ifndef COST
81         this->cost[v][0] = w;
82 #endif
83         this->dfs_LCA_weighted(v, u, l + 1, d + w, adj);
84     }
85 }
86
87 void build_unweighted(const vector<vector<int>> &adj) {
88     this->dfs_LCA_unweighted(this->indexed_from, -1, 1, 0, adj);
89
90     this->build_anc();
91 }
92
93 void dfs_LCA_unweighted(const int u, const int p, const int l, const int d,
94                     const vector<vector<int>> &adj) {
95     this->level[u] = l;
96     this->anc[u][0] = p;
97 #ifndef DIST
98     this->dist[u] = d;
99 #endif
100
101     for (const int v : adj[u]) {
102         if (v == p)
103             continue;
104         this->dfs_LCA_unweighted(v, u, l + 1, d + 1, adj);
105     }
106 }

```

```

107 // go up k levels from x
108 int lca_go_up(int x, int k) {
109     for (int i = 0; k > 0; i++, k >>= 1)
110         if (k & 1) {
111             x = this->anc[x][i];
112             if (x == -1)
113                 return -1;
114         }
115     }
116     return x;
117 }
118
119 #ifndef COST
120 /// Query between the an ancestor of v (p) and v. It returns the
121 /// max/min edge between them.
122 int lca_query_cost_in_line(int v, int p) {
123     assert(this->level[v] >= this->level[p]);
124
125     int k = this->level[v] - this->level[p];
126     int ans = NEUTRAL_VALUE;
127
128     for (int i = 0; k > 0; i++, k >>= 1)
129         if (k & 1) {
130             ans = combine(ans, this->cost[v][i]);
131             v = this->anc[v][i];
132         }
133     }
134     return ans;
135 }
136 #endif
137
138 int get_lca(int a, int b) {
139     // a is below b
140     if (this->level[b] > this->level[a])
141         swap(a, b);
142
143     const int logg = lg[this->level[a]];
144
145     // putting a and b in the same level
146     for (int i = logg; i >= 0; i--)
147         if (this->level[a] - (1 << i) >= this->level[b])
148             a = this->anc[a][i];
149
150     if (a == b)
151         return a;
152
153     for (int i = logg; i >= 0; i--)
154         if (this->anc[a][i] != -1 && this->anc[a][i] != this->anc[b][i]) {
155             a = this->anc[a][i];
156             b = this->anc[b][i];
157         }
158
159     return anc[a][0];
160 }
161
162 public:
163     /// Builds an weighted graph.
164     ///
165     /// Time Complexity: O(n*log(n))
166     explicit LCA(const vector<vector<pair<int, int>>> &adj,
167                 const int indexed_from) {
168         this->n = adj.size();
169         this->indexed_from = indexed_from;
170         this->allocate();
171     }

```

```

172     this->build_weighted(adj);
173 }
174
175 /// Builds an unweighted graph.
176 ///
177 /// Time Complexity: O(n*log(n))
178 explicit LCA(const vector<vector<int>>> &adj, const int indexed_from) {
179     this->n = adj.size();
180     this->indexed_from = indexed_from;
181     this->allocate();
182
183     this->build_unweighted(adj);
184 }
185
186 /// Goes up k levels from v. If it passes the root, returns -1.
187 ///
188 /// Time Complexity: O(log(k))
189 int go_up(const int v, const int k) {
190     assert(indexed_from <= v);
191     assert(v < this->n + indexed_from);
192
193     return this->lca_go_up(v, k);
194 }
195
196 /// Returns the parent of v in the LCA dfs from 1.
197 ///
198 /// Time Complexity: O(1)
199 int parent(const int v) {
200     assert(indexed_from <= v);
201     assert(v < this->n + indexed_from);
202
203     return this->anc[v][0];
204 }
205
206 /// Returns the LCA of a and b.
207 ///
208 /// Time Complexity: O(log(n))
209 int query_lca(const int a, const int b) {
210     assert(indexed_from <= min(a, b));
211     assert(max(a, b) < this->n + indexed_from);
212
213     return this->get_lca(a, b);
214 }
215
216 #ifndef DIST
217 /// Returns the distance from a to b. When the graph is unweighted, it is
218 /// considered 1 as the weight of the edges.
219 ///
220 /// Time Complexity: O(log(n))
221 int query_dist(const int a, const int b) {
222     assert(indexed_from <= min(a, b));
223     assert(max(a, b) < this->n + indexed_from);
224
225     return this->dist[a] + this->dist[b] - 2 * this->dist[this->get_lca(a,
226 b)];
227 }
228 #endif
229
230 #ifndef COST
231 /// Returns the max/min weight edge from a to b.
232 ///
233 /// Time Complexity: O(log(n))
234 int query_cost(const int a, const int b) {
235     assert(indexed_from <= min(a, b));

```

```

236     assert(max(a, b) < this->n + indexed_from);
237
238     const int l = this->query_lca(a, b);
239     return combine(this->lca_query_cost_in_line(a, l),
240                 this->lca_query_cost_in_line(b, l));
241 }
242 #endif
243 };

```

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

```
1 |IS maximal| = |V| - MAXIMUM_MATCHING
```

5.20. Maximum Path Unweighted Graph

```

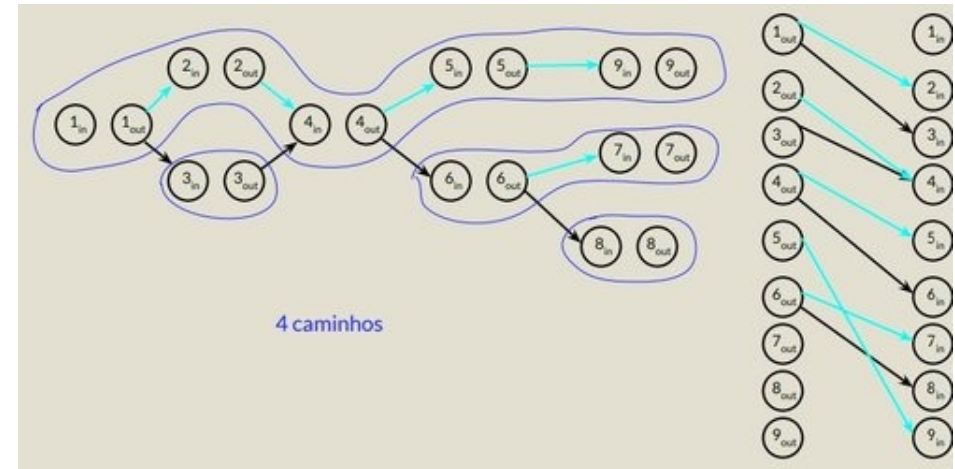
1  /// Returns the maximum path between the vertices 0 and n - 1 in a
  /// unweighted graph.
2  ///
3  /// Time Complexity: O(V + E)
4  int maximum_path(int n) {
5      vector<int> top_order = topological_sort(n);
6      vector<int> pai(n, -1);
7      if(top_order.empty())
8          return -1;
9
10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

```

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

```
1 |E minimal| = |V| - MAXIMUM_MATCHING
```

5.22. Minimum Path Cover In Dag



5.23. 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.

5.24. Number Of Different Spanning Trees In A Complete Graph

- Cayley's formula
- n^{n-2}

5.25. Number Of Ways To Make A Graph Connected

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

5.26. Pruffer Decode

- IT MUST BE INDEXED BY 0.
- /// Returns the adjacency matrix of the decoded tree.
- ///
- /// Time Complexity: O(V)
- vector**<**vector**<**int**>> pruefer_decode(**const** **vector**<**int**> &code) {
-
- int** n = code.size() + 2;
- 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 }

```

5.27. 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> prueffer_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     }

```

```

36     ptr++;
37     while (degree[ptr] != 1)
38         ptr++;
39     leaf = ptr;
40 }
41 }
42
43 return code;
44 }

```

5.28. Pruffer Properties

- 1 * 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.
- 2 * 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**.

5.29. Remove All Bridges From Graph

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

5.30. 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     stack<int> s;
39     // Fill vertices in stack according to their finishing times
40     for(int i = indexed_from; i < this->n + indexed_from; i++)
41         if(!comp[i])
42             dfs_fill_order(i, s, adj);
43
44     // Create a reversed graph
45     get_transpose(adj);
46
47     fill(comp.begin(), comp.end(), -1);
48
49     // Now process all vertices in order defined by stack
50     while(s.empty() == false) {
51         int v = s.top();
52         s.pop();
53
54         if(comp[v] == -1)
55             dfs_trans(v, this->number_of_comp++);
56     }
57 }
58
59 public:
60 // number of the component of the i-th vertex
61 // it's always indexed from 0
62 vector<int> comp;
63 // the i-th vector contains the vertices that belong to the i-th scc
64 // it's always indexed from 0
65 vector<vector<int>> scc;
66 int number_of_comp = 0;
67
68 SCC(int n, int indexed_from, vector<vector<int>>& adj) {
69     this->n = n;
70     this->indexed_from = indexed_from;
71     comp.resize(n + 1);
72     trans.resize(n + 1);
73     scc.resize(n + 1);
74
75     this->compute_SCC(adj);
76 }
77 };
78

```

5.31. 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) {
31     // There exists a cycle in the graph
32     return vector<int>();
33 }
34
35 return top_order;
36 }

```

5.32. 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 }

```

6. Language Stuff

6.1. Climits

```
1 LONG_MIN -> (-2^31+1) :: LONG_MAX -> (2^31-1)
2 ULONG_MAX -> (2^32-1) -> UNSIGNED
3 LLONG_MIN, LLONG_MAX, ULLONG_MAX
```

6.2. Checagem E Transformacao De Caractere

```
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 minusculo
8 toupper(str[i]); //converte para maiusculo
```

6.3. Conta Digitos 1 Ate N

```
1 int solve(int n) {
2
3     int maxx = 9, minn = 1, dig = 1, ret = 0;
4
5     for(int i = 1; i <= 17; i++) {
6         int q = min(maxx, n);
7         ret += max(0ll, (q - minn + 1) * dig);
8         maxx = (maxx * 10 + 9), minn *= 10, dig++;
9     }
10
11     return ret;
12 }
```

6.4. Escrita Em Arquivo

```
1 ofstream cout("output.txt");
```

6.5. Gcd

```
1 int _gcd(int a, int b){
2     if(a == 0 || b == 0) return 0;
3     else return abs(__gcd(a,b));
4 }
```

6.6. Hipotenusa

```
1 cout << hypot(3,4); // output: 5
```

6.7. Int To Binary String

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

6.8. Int To String

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

6.9. Leitura De Arquivo

```
1 ifstream cin("input.txt");
```

6.10. Max E Min Element Num Vetor

```
1 int maior = *max_element(arr.begin(), arr.end());
2 int menor = *min_element(arr.begin(), arr.end());
3 // OBS: Retorna iterador
```

6.11. Permutacao

```
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));
```

6.12. Remove Repeticoes Continuas Num Vetor

```
1 // arr = {10,20,20,20,30,20,20,10}
2 it = unique(arr.begin(), arr.end());
3 // arr = {10,20,30,20,10, iterator aponta pra aqui, ...}
4 arr.resize(distance(arr.begin(), it));
5 // arr = {10,20,30,20,10}
```

6.13. Rotate (Left)

```
1 Passado o inicio o meio e o fim ele rotaciona de forma que o meio seja o
  novo inicio.
2 vector<int> arr(n); // 1 2 3 4 5 6 7 8 9
3 rotate(arr.begin(),arr.begin()+3,arr.end()); //4 5 6 7 8 9 1 2 3
```

6.14. 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
```

6.15. Scanf De Uma String

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

6.16. Split Function

```
1 // SEPARA STRING POR UM DELIMITADOR
2 // EX: str=A-B-C split -> x = {A,B,C}
3 vector<string> split(const string &s, char delim) {
4     stringstream ss(s);
5     string item;
6     vector<string> tokens;
7     while (getline(ss, item, delim)) {
8         tokens.push_back(item);
9     }
10    return tokens;
```

```

11 }
12 int main () {
13     vector<string> x = split("cap-one-best-opinion-language", '-');
14     // x = {cap,one,best,opinion,language};
15 }

```

6.17. String To Long Long

```

1 string s = "0xFFFF"; int base = 16;
2 string::size_type sz = 0;
3 int ll = stoll(s,&sz,base); // ll = 65535, sz = 6;
4 OBS: Não precisa colocar o sz, pode colocar 0; // stoll(s,0,base);

```

6.18. Substring

```

1 string s = "abcdef";
2 s.substr(posição inicial, qtd de char(opcional));
3 string s2 = s.substr(3,2); // s2 = "de"
4 string s3 = s.substr(2); // s3 = "cdef"

```

6.19. Width

```

1 cout << width(13);
2 cout << 100 << endl; // "      100      "
3 cout.fill('x');
4 cout.width(13);
5 cout << 100 << endl; // "xxxxx100xxxxx"
6 cout << right << 100 << endl; "xxxxxxx100"

```

6.20. 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

```

6.21. Check

```

1 #!/bin/bash
2 g++ -std=c++17 gen.cpp -o gen
3 g++ -std=c++17 a.cpp -o a
4 g++ -std=c++17 brute.cpp -o brute
5
6 for((i=1;;i++)); do
7     echo $i
8     ./gen $i > in
9     diff <./a <in <./brute <in || break
10 done
11
12 cat in
13 #sed -i 's/\r$//' filename ----- remover \r do txt

```

6.22. 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_saddll_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
    int *res)

```

```

5 bool __builtin_uadd_overflow (unsigned int a, unsigned int b, unsigned int
    *res)
6 bool __builtin_uaddl_overflow (unsigned long int a, unsigned long int b,
    unsigned long int *res)
7 bool __builtin_uaddll_overflow (unsigned long long int a, unsigned long long
    int b, unsigned long long int *res)
8
9 bool __builtin_sub_overflow (type1 a, type2 b, type3 *res)
10 bool __builtin_ssub_overflow (int a, int b, int *res)
11 bool __builtin_ssubl_overflow (long int a, long int b, long int *res)
12 bool __builtin_ssubll_overflow (long long int a, long long int b, long long
    int *res)
13 bool __builtin_usub_overflow (unsigned int a, unsigned int b, unsigned int
    *res)
14 bool __builtin_usubl_overflow (unsigned long int a, unsigned long int b,
    unsigned long int *res)
15 bool __builtin_usubll_overflow (unsigned long long int a, unsigned long long
    int b, unsigned long long int *res)
16
17 bool __builtin_mul_overflow (type1 a, type2 b, type3 *res)
18 bool __builtin_smul_overflow (int a, int b, int *res)
19 bool __builtin_smull_overflow (long int a, long int b, long int *res)
20 bool __builtin_smulll_overflow (long long int a, long long int b, long long
    int *res)
21 bool __builtin_umul_overflow (unsigned int a, unsigned int b, unsigned int
    *res)
22 bool __builtin_umull_overflow (unsigned long int a, unsigned long int b,
    unsigned long int *res)
23 bool __builtin_umulll_overflow (unsigned long long int a, unsigned long long
    int b, unsigned long long int *res)

```

6.23. 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

```

6.24. Random Numbers

```

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

```

6.25. 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 }

```

6.26. 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);

```

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  int inv(int a, int m) {
2      int m0 = m, t, q;
3      int x0 = 0, x1 = 1;
4
5      if (m == 1)
6          return 0;
7
8      // Apply extended Euclid Algorithm
9      while (a > 1) {
10         // q is quotient
11         if (m == 0)
12             return INF;
13         q = a / m;
14         t = m;

```

```

15         // m is remainder now, process same as euclid's algo
16         m = a % m, a = t;
17         t = x0;
18         x0 = x1 - q * x0;
19         x1 = t;
20     }
21
22     // Make x1 positive
23     if (x1 < 0)
24         x1 += m0;
25
26     return x1;
27 }
28 // k is size of num[] and rem[]. Returns the smallest
29 // number x such that:
30 // x % num[0] = rem[0],
31 // x % num[1] = rem[1],
32 // .....
33 // x % num[k-2] = rem[k-1]
34 // Assumption: Numbers in num[] are pairwise coprimes
35 // (gcd for every pair is 1)
36 int findMinX(const vector<int> &num, const vector<int> &rem, const int k) {
37     // Compute product of all numbers
38     int prod = 1;
39     for (int i = 0; i < k; i++)
40         prod *= num[i];
41
42     int result = 0;
43
44     // Apply above formula
45     for (int i = 0; i < k; i++) {
46         int pp = prod / num[i];
47         int iv = inv(pp, num[i]);
48         if (iv == INF)
49             return INF;
50         result += rem[i] * inv(pp, num[i]) * pp;
51     }
52
53     // IF IS NOT VALID RETURN INF
54     return (result % prod == 0 ? INF : result % prod);
55 }

```

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 & 111)
22         ans = (ans * cur_pow) % MOD;
23
24     cur_pow = (cur_pow * cur_pow) % MOD;
25     p >>= 111;
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. 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 }

```

7.7. 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 }

```

7.8. Extended Euclidean

```

1 int gcd, x, y;
2
3 // Ax + By = gcd(A,B)
4
5 void extended_euclidian(const int a, const int b) {
6     if (b == 0) {
7         gcd = a;
8         x = 1;
9         y = 0;
10    } else {
11        extended_euclidian(b, a % b);
12        const int temp = x;
13        x = y;
14        y = temp - (a / b) * y;
15    }
16 }

```

7.9. 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 }

```

7.10. Inclusion Exclusion

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

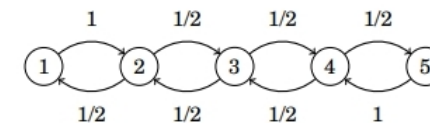
7.11. Inclusion Exclusion

```

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
8     for (int mask = 1; mask < (1ll << n); mask++) {
9         int lcm = 1;
10        for (int i = 0; i < n; i++)
11            if (mask & (1ll << i))
12                lcm = lcm * arr[i] / __gcd(lcm, arr[i]);
13        // if the number of element is odd, then add
14        if (__builtin_popcount_ll(mask) % 2 == 1)
15            c += LIMIT / lcm;
16        else // otherwise subtract
17            c -= LIMIT / lcm;
18    }
19    return LIMIT - c;
20 }

```

7.12. 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

7.13. 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

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

7.14. Matrix Exponentiation

```

1 struct Matrix {
2     static constexpr int MOD = 1e9 + 7;
3
4     // static matrix, if it's created multiple times, it's recommended
5     // to avoid TLE.
6     static constexpr int MAXN = 4, MAXM = 4;
7     array<array<int, MAXM>, MAXN> mat = {};
8     int n, m;
9     Matrix(const int n, const int m) : n(n), m(m) {}
10
11     static int mod(int n) {
12         n %= MOD;
13         if (n < 0)
14             n += MOD;
15         return n;
16     }
17
18     /// Creates a n x n identity matrix.
19     ///
20     /// Time Complexity: O(n*n)
21     Matrix identity() {
22         assert(n == m);
23         Matrix mat_identity(n, m);
24         for (int i = 0; i < n; ++i)
25             mat_identity.mat[i][i] = 1;
26         return mat_identity;
27     }
28

```

```

29 /// Multiplies matrices mat and other.
30 ///
31 /// Time Complexity: O(mat.size() ^ 3)
32 Matrix operator*(const Matrix &other) const {
33     assert(m == other.n);
34     Matrix ans(n, other.m);
35     for (int i = 0; i < n; ++i)
36         for (int j = 0; j < m; ++j)
37             for (int k = 0; k < m; ++k)
38                 ans.mat[i][j] = mod(ans.mat[i][j] + mat[i][k] * other.mat[k][j]);
39     return ans;
40 }
41
42 /// Exponentiates the matrix mat to the power of p.
43 ///
44 /// Time Complexity: O((mat.size() ^ 3) * log2(p))
45 Matrix expo(int p) {
46     assert(p >= 0);
47     Matrix ans = identity(), cur_power(n, m);
48     cur_power.mat = mat;
49     while (p) {
50         if (p & 1)
51             ans = ans * cur_power;
52
53         cur_power = cur_power * cur_power;
54         p >>= 1;
55     }
56     return ans;
57 }
58 };

```

7.15. Pollard Rho (Find A Divisor)

```

1 // Requires binary_exponentiation.cpp
2
3 /// Returns a prime divisor for n.
4 ///
5 /// Expected Time Complexity: O(nl/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 }

```

7.16. 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 }

```

7.17. 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

```

7.18. 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 int spf[MAXN + 5];
9
10 vector<int> primes;
11 bitset<MAXN + 5> isPrime;
12
13 void sieve(int n = MAXN + 2) {
14
15     for (int i = 0; i <= n; i++)
16         spf[i] = i;
17
18     isPrime.set();
19     for (int i = 2; i <= n; i++) {
20         if (!isPrime[i])
21             continue;
22
23         for (int j = i * i; j <= n; j += i) {
24             isPrime[j] = false;
25             spf[j] = min(i, spf[j]);
26         }
27         primes.emplace_back(i);
28     }
29 }
30
31 vector<int> getFactorization(int x) {
32     vector<int> ret;
33     while (x != 1) {
34         ret.emplace_back(spf[x]);
35         x = x / spf[x];
36     }
37     return ret;
38 }
39
40 /// Gets all primes from l to r
41 void segSieve(int l, int r) {
42     // primes from l to r
43     // transferred to 0..(l-r)
44     segPrimes.clear();
45     primesInSeg.set();
46     int sq = sqrt(r) + 5;
47
48     for (int p : primes) {
49         if (p > sq)
50             break;
51
52         for (int i = l - l % p; i <= r; i += p) {
53             if (i - l < 0)
54                 continue;
55
56             // if i is less than 1e6, it could be checked in the
57             // array of the sieve
58             if (i >= (int)1e6 || !isPrime[i])
59                 primesInSeg[i - l] = false;
60         }
61     }
62
63     for (int i = 0; i < r - l + 1; i++) {
64         if (primesInSeg[i])
65             segPrimes.emplace_back(i + l);
66     }
67 }

```

7.19. 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 // REQUIRES SCC code
2
3 // OBS: INDEXED FROM 0
4 class SAT {
5
6 private:
7     vector<vector<int>> adj;
8     int n;
9
10 public:
11     vector<bool> ans;
12
13     SAT(int n) {
14         this->n = n;
15         adj.resize(2 * n);
16         ans.resize(n);
17     }
18
19     // (X v Y) = (X -> ~Y) & (~X -> Y)
20     void add_or(int x, bool pos_x, int y, bool pos_y) {
21         assert(0 <= x), assert(x < n);
22         assert(0 <= y), assert(y < n);

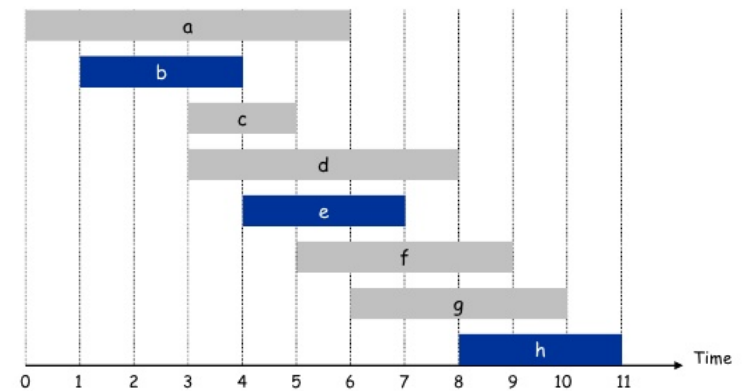
```

```

23         adj[(x << 1) ^ pos_x].pb((y << 1) ^ (pos_y ^ 1));
24         adj[(y << 1) ^ pos_y].pb((x << 1) ^ (pos_x ^ 1));
25     }
26
27     // (X xor Y) = (X v Y) & (~X v ~Y)
28     // for this function the result is always 0 1 or 1 0
29     void add_xor(int x, bool pos_x, int y, bool pos_y) {
30         assert(0 <= x), assert(x < n);
31         assert(0 <= y), assert(y < n);
32         add_or(x, y, pos_x, pos_y);
33         add_or(x, y, pos_x ^ 1, pos_y ^ 1);
34     }
35
36     bool check() {
37         SCC scc(2 * n, 0, adj);
38
39         for (int i = 0; i < n; i++) {
40             if (scc.comp[(i << 1) | 1] == scc.comp[(i << 1) | 0])
41                 return false;
42             ans[i] = (scc.comp[(i << 1) | 1] < scc.comp[(i << 1) | 0]);
43         }
44
45         return true;
46     }
47 };

```

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. Oito Rainhas

```

1 #define N 4
2 bool isSafe(int mat[N][N], int row, int col) {
3     for(int i = row - 1; i >= 0; i--)

```

```

4     if(mat[i][col])
5         return false;
6     for(int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--,j--)
7         if(mat[i][j])
8             return false;
9     for(int i = row - 1, j = col + 1; i >= 0 && j < N; i--,j++)
10        if(mat[i][j])
11            return false;
12    return true;
13 }
14 // inicialmente a matriz esta zerada
15 int queen(int mat[N][N], int row = 0) {
16     if(row >= N) {
17         for(int i = 0; i < N; i++) {
18             for(int j = 0; j < N; j++) {
19                 cout << mat[i][j] << ' ';
20             }
21             cout << endl;
22         }
23         cout << endl << endl;
24         return false;
25     }
26     for(int i = 0; i < N; i++) {
27         if(isSafe(mat, row, i)) {
28             mat[row][i] = 1;
29             if(queen(mat, row+1))
30                 return true;
31             mat[row][i] = 0;
32         }
33     }
34     return false;
35 }

```

8.5. 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 }

```

8.6. 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 }

```

8.7. 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

```

8.8. 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. 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 }

```

8.10. Kadane (Segment Tree)

```

1  struct Node {
2      int pref, suf, tot, best;
3      Node () {}
4      Node(int pref, int suf, int tot, int best) : pref(pref), suf(suf),
5          tot(tot), best(best) {}
6  };
7
8  const int MAXN = 2E5 + 10;
9  Node tree[5*MAXN];
10 int arr[MAXN];
11
12 Node query(const int l, const int r, const int i, const int j, const int
13 pos) {
14     if (l > r || l > j || r < i)
15         return Node(-INF, -INF, -INF, -INF);
16
17     if (i <= l && r <= j)
18         return Node(tree[pos].pref, tree[pos].suf, tree[pos].tot,
19 tree[pos].best);
20
21     int mid = (l + r) / 2;
22     Node left = query(l, mid, i, j, 2*pos+1), right = query(mid+1, r, i, j, 2*pos+2);
23     Node x;
24     x.pref = max({left.pref, left.tot, left.tot + right.pref});
25     x.suf = max({right.suf, right.tot, right.tot + left.suf});
26     x.tot = left.tot + right.tot;
27     x.best = max({left.best, right.best, left.suf + right.pref});
28     return x;
29 }

```

```

29 // Update arr[idx] to v
30 // ITS NOT DELTA!!!
31 void update(int l, int r, const int idx, const int v, const int pos) {
32     if (l > r || l > idx || r < idx)
33         return;
34
35     if (l == idx && r == idx) {
36         tree[pos] = Node(v, v, v, v);
37         return;
38     }
39
40     int mid = (l + r) / 2;
41     update(l, mid, idx, v, 2*pos+1); update(mid+1, r, idx, v, 2*pos+2);
42     l = 2*pos+1, r = 2*pos+2;
43     tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
44 tree[r].pref});
45     tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
46     tree[pos].tot = tree[l].tot + tree[r].tot;
47     tree[pos].best = max({tree[l].best, tree[r].best, tree[l].suf +
48 tree[r].pref});
49 }
50
51 void build(int l, int r, const int pos) {
52     if (l == r) {
53         tree[pos] = Node(arr[l], arr[l], arr[l], arr[l]);
54         return;
55     }
56
57     int mid = (l + r) / 2;
58     build(l, mid, 2*pos+1); build(mid+1, r, 2*pos+2);
59     l = 2*pos+1, r = 2*pos+2;
60     tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
61 tree[r].pref});
62     tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
63     tree[pos].tot = tree[l].tot + tree[r].tot;
64     tree[pos].best = max({tree[l].best, tree[r].best, tree[l].suf +
65 tree[r].pref});
66 }

```

8.11. 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  // Implementation of Kadane's algorithm for 1D array. The function
9  // returns the maximum sum and stores starting and ending indexes of the
10 // maximum sum subarray at addresses pointed by start and finish pointers
11 // respectively.
12 int kadane(int* arr, int* start, int* finish, int n) {
13     // initialize sum, maxSum and
14     int sum = 0, maxSum = INT_MIN, i;
15
16     // Just some initial value to check for all negative values case
17     *finish = -1;
18
19     // local variable
20     int local_start = 0;
21
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 }

```

8.12. 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 }

```

8.13. 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.erase(unique(aux.begin(), aux.end()), aux.end());
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 }

```

8.14. 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))

```

```

6 int ternary_search(const vector<int> &arr) {
7     int l = 0, r = (int)arr.size() - 1;
8     while (r - l > 2) {
9         int lc = l + (r - l) / 3;
10        int rc = r - (r - l) / 3;
11        // the function f(x) returns the element on the position x
12        if (f(lc) > f(rc))
13            // the function is going down, then the middle is on the right.
14            l = lc;
15        else
16            r = rc;
17    }
18    // the range [l, r] contains the minimum element.
19
20    int minn = f(l), idx = l;
21    for (int i = l + 1; i <= r; ++i)
22        if (f(i) < minn) {
23            idx = i;
24            minn = f(i);
25        }
26
27    return idx;
28 }

```

9. Strings

9.1. 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

```

9.2. 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.

```

9.3. 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 }

```

9.4. 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 };
```

9.5. 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
51    /// Time Complexity: O(n)
52    void build_table(const string &s) {
53        pot.resize(m.size(), vector<int>(this->n));
54        hash_table.resize(m.size(), vector<int>(this->n));
55
56        for (int i = 0; i < m.size(); i++) {
57            pot[i][0] = 1;
58            hash_table[i][0] = s[0];
59            for (int j = 1; j < this->n; 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     /// Time Complexity: O(1) -> Actually O(number_of_primes)
77     pair<int, int> query(const int l, const int r) {
78         assert(0 <= l), assert(l <= r), assert(r < this->n);
79         return hash_query(l, r);
80     }
81 }
82 };

```

9.6. 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 }

```

9.7. 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     }
57
58     Suffix_Array say(concat);
59     vector<int> sa = say.get_suffix_array();
60     Sparse_Table spt(say.get_lcp());
61
62     vector<int> freq(n);
63     int cnt1 = 0;
64
65     /// Ignore separators
66     int i = n, j = n - 1;

```



```

66 int ans = 0;
67
68 while(true) {
69     if(cnt1 == n) {
70         ans = max(ans, spt.query(i, j - 1));
71
72         int idx = ind[sa[i]];
73         freq[idx]--;
74         if(freq[idx] == 0)
75             cnt1--;
76         i++;
77     } else if(j == (int)sa.size() - 1)
78         break;
79     else {
80         j++;
81         int idx = ind[sa[j]];
82         freq[idx]++;
83         if(freq[idx] == 1)
84             cnt1++;
85     }
86 }
87
88 cout << ans << endl;
89
90 }

```

9.8. 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 }

```

9.9. Manacher (Longest Palindrome)

```

1 //
2     https://medium.com/hackernoon/manachers-algorithm-explained-longest-palindrome-substring-22eb274e998f
3
4 /// Create a string containing '#' characters between any two characters.
5 string get_modified_string(string &s){
6     string ret;
7     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
17 /// Time Complexity: O(n)
18 string get_best(const int max_len, const string &str, const vector<int>
19    &lps) {
20     for(int i = 0; i < lps.size(); i++) {
21         if(lps[i] == max_len) {
22             string ans;
23             int cnt = max_len / 2;
24             int io = i - 1;
25             while(cnt) {
26                 if(str[io] != '#') {
27                     ans += str[io];
28                     cnt--;
29                 }
30                 io--;
31             }
32             reverse(ans.begin(), ans.end());
33             if(str[i] != '#')
34                 ans += str[i];
35             cnt = max_len / 2;
36             io = i + 1;
37             while(cnt) {
38                 if(str[io] != '#') {
39                     ans += str[io];
40                     cnt--;
41                 }
42                 io++;
43             }
44             return ans;
45         }
46     }
47
48 /// Returns a pair containing the size of the longest palindrome and the
49    first occurrence of it.
50
51 /// Time Complexity: O(n)
52 pair<int, string> manacher(string &s) {
53     int n = s.size();
54     string str = get_modified_string(s);
55     int len = (2 * n) + 1;
56     //the i-th index contains the longest palindromic substring with the i-th
57     char as the center
58     vector<int> lps(len);
59     int c = 0; //stores the center of the longest palindromic substring until
60     now
61     int r = 0; //stores the right boundary of the longest palindromic
62     substring until now
63     int max_len = 0;
64     for(int i = 0; i < len; i++) {
65         //get mirror index of i
66         int mirror = (2 * c) - i;
67
68         //see if the mirror of i is expanding beyond the left boundary of
69         current longest palindrome at center c
70         //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
79 //boundary of current longest palindrome at center c
80 //if it is, the new center is i
81 if(i + lps[i] > r) {
82     c = i;
83     r = i + lps[i];
84
85     if(lps[i] > max_len) //update max_len
86         max_len = lps[i];
87 }
88
89 return make_pair(max_len, get_best(max_len, str, lps));
90 }

```

9.10. Suffix Array

```

1 // To use the compare method use the macro below.
2 #define BUILD_TABLE
3
4 namespace RadixSort {
5     /// Sorts the array arr stably in ascending order.
6     ///
7     /// Time Complexity: O(n + max_element)
8     /// Space Complexity: O(n + max_element)
9     template <typename T>
10     void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),
11             const int begin = 0) {
12         const int n = arr.size();
13         vector<T> new_order(n);
14         vector<int> count(max_element + 1, 0);
15
16         for (int i = begin; i < n; ++i)
17             ++count[get_key(arr[i])];
18
19         for (int i = 1; i <= max_element; ++i)
20             count[i] += count[i - 1];
21
22         for (int i = n - 1; i >= begin; --i) {
23             new_order[count[get_key(arr[i])] - (begin == 0)] = arr[i];
24             --count[get_key(arr[i])];
25         }
26
27         arr = move(new_order);
28     }
29
30     /// Sorts an array by their pair of ranks stably in ascending order.
31     template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
32         // sort by the second rank
33         RadixSort::sort<T>(
34             arr, rank_size, [](T &item) { return item.first.second; }, 0);

```

```

35 // sort by the first rank
36 RadixSort::sort<T>(
37     arr, rank_size, [](T &item) { return item.first.first; }, 0);
38 }
39 } // namespace RadixSort
40
41 /// It is indexed by 0.
42 /// Let the given string be "banana".
43 ///
44 /// 0 banana          5 a
45 /// 1 anana          Sort the Suffixes 3 ana
46 /// 2 nana           ----->         1 anana
47 /// 3 ana            alphabetically    0 banana
48 /// 4 na              4 na
49 /// 5 a                2 nana
50 /// So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
51 ///
52 /// LCP
53 /// 1 a
54 /// 3 ana
55 /// 0 anana
56 /// 0 banana
57 /// 2 na
58 /// 0 nana (The last position will always be zero)
59 /// So the LCP for "banana" is {1, 3, 0, 0, 2, 0}
60 class Suffix_Array {
61 private:
62     const string s;
63     const int n;
64
65     typedef pair<int, int> Rank;
66 #ifdef BUILD_TABLE
67     vector<vector<int>> rank_table;
68     const vector<int> log_array = build_log_array();
69 #endif
70
71 public:
72     Suffix_Array(const string &s) : n(s.size()), s(s) {}
73
74 private:
75     vector<int> build_log_array() {
76         vector<int> log_array(this->n + 1, 0);
77         for (int i = 2; i <= this->n; ++i)
78             log_array[i] = log_array[i / 2] + 1;
79         return log_array;
80     }
81
82     static void build_ranks(const vector<pair<Rank, int>> &ranks,
83                           vector<int> &ret) {
84         // The vector containing the ranks will be present at ret
85         ret[ranks[0].second] = 1;
86         for (int i = 1; i < ranks.size(); ++i) {
87             // if their rank are equal, then their position should be the same
88             if (ranks[i - 1].first == ranks[i].first)
89                 ret[ranks[i].second] = ret[ranks[i - 1].second];
90             else
91                 ret[ranks[i].second] = ret[ranks[i - 1].second] + 1;
92         }
93     }
94
95     /// Time Complexity: O(n*log(n))
96     vector<int> build_suffix_array() {
97         // the tuple below represents the rank and the index associated with it
98         vector<pair<Rank, int>> ranks(this->n);

```

```

100     vector<int> arr(this->n);
101
102     for (int i = 0; i < n; ++i)
103         ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);
104
105 #ifdef BUILD_TABLE
106     int rank_table_size = 0;
107     this->rank_table.resize(log_array[this->n] + 2);
108 #endif
109     RadixSort::sort_pairs(ranks, 256);
110     build_ranks(ranks, arr);
111
112     {
113         int jump = 1;
114         int max_rank = arr[ranks.back().second];
115
116         // it will be compared intervals a pair of intervals (i, jump-1), (i +
117         // jump, i + 2*jump - 1). The variable jump is always a power of 2
118 #ifdef BUILD_TABLE
119         while (jump / 2 < this->n) {
120 #else
121         while (max_rank != this->n) {
122 #endif
123             for (int i = 0; i < this->n; ++i) {
124                 ranks[i].first.first = arr[i];
125                 ranks[i].first.second = (i + jump < this->n ? arr[i + jump] : 0);
126                 ranks[i].second = i;
127             }
128
129 #ifdef BUILD_TABLE
130             // inserting only the ranks in the table
131             transform(ranks.begin(), ranks.end(),
132                     back_inserter(rank_table[rank_table_size++]),
133                     [](pair<Rank, int> &pair) { return pair.first.first; });
134 #endif
135             RadixSort::sort_pairs(ranks, n);
136             build_ranks(ranks, arr);
137
138             max_rank = arr[ranks.back().second];
139             jump *= 2;
140         }
141
142     }
143
144     vector<int> sa(this->n);
145     for (int i = 0; i < this->n; ++i)
146         sa[arr[i] - 1] = i;
147     return sa;
148 }
149
150 /// Builds the lcp (Longest Common Prefix) array for the string s.
151 /// A value lcp[i] indicates length of the longest common prefix of the
152 /// suffixes indexed by i and i + 1. Implementation of the Kasai's
153 /// Algorithm.
154 /// Time Complexity: O(n)
155 vector<int> build_lcp() {
156     vector<int> lcp(this->n, 0);
157     vector<int> inverse_suffix(this->n);
158
159     for (int i = 0; i < this->n; ++i)
160         inverse_suffix[sa[i]] = i;
161
162     for (int i = 0, k = 0; i < this->n; ++i) {
163         if (inverse_suffix[i] == this->n - 1) {
164             k = 0;

```

```

164     } else {
165         int j = sa[inverse_suffix[i] + 1];
166         while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
167             ++k;
168
169         lcp[inverse_suffix[i]] = k;
170
171         if (k > 0)
172             --k;
173     }
174 }
175
176 return lcp;
177 }
178
179 int _lcs(const int separator) {
180     int ans = 0;
181     for (int i = 0; i + 1 < this->sa.size(); ++i) {
182         const int left = this->sa[i];
183         const int right = this->sa[i + 1];
184         if ((left < separator && right > separator) ||
185             (left > separator && right < separator))
186             ans = max(ans, lcp[i]);
187     }
188     return ans;
189 }
190
191 #ifdef BUILD_TABLE
192 int _compare(const int i, const int j, const int length) {
193     const int k = this->log_array[length]; // floor log2(length)
194     const int jump = length - (1ll << k);
195
196     const pair<int, int> iRank = {
197         this->rank_table[k][i],
198         (i + jump < this->n ? this->rank_table[k][i + jump] : -1)};
199     const pair<int, int> jRank = {
200         this->rank_table[k][j],
201         (j + jump < this->n ? this->rank_table[k][j + jump] : -1)};
202     return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
203 }
204 #endif
205
206 public:
207     const vector<int> sa = build_suffix_array();
208     const vector<int> lcp = build_lcp();
209
210     /// LCS of two strings A and B. The string s must be initialized in the
211     /// constructor as the string (A + '$' + B).
212     /// The string A starts at index 1 and ends at index (separator - 1).
213     /// The string B starts at index (separator + 1) and ends at the end of the
214     /// string.
215     /// Time Complexity: O(n)
216     int lcs(const int separator) {
217         assert(!isalpha(this->s[separator]) && !isdigit(this->s[separator]));
218         return _lcs(separator);
219     }
220 }
221
222 #ifdef BUILD_TABLE
223     /// Compares two substrings beginning at indexes i and j of a fixed length.
224     /// Time Complexity: O(1)
225     int compare(const int i, const int j, const int length) {
226         assert(0 <= i && i < this->n && 0 <= j && j < this->n);
227         assert(i + length - 1 < this->n && j + length - 1 < this->n);

```

```

229     return _compare(i, j, length);
230 }
231 #endif
232 };

```

9.11. Suffix Array Pessoa

```

1 // OBS: Suffix Array build code imported from:
2 // https://github.com/gabrielpessoal/Biblioteca-Maratona/
3 // blob/master/code/String/SuffixArray.cpp
4 // Because it's faster.
5 // Swap the method below with the one in "suffix_array.cpp"
6
7 vector<int> build_suffix_array() {
8     int n = this->s.size(), c = 0;
9     vector<int> temp(n), posBucket(n), bucket(n), bpos(n), out(n);
10    for (int i = 0; i < n; i++)
11        out[i] = i;
12    sort(out.begin(), out.end(),
13         [&](int a, int b) { return this->s[a] < this->s[b]; });
14    for (int i = 0; i < n; i++) {
15        bucket[i] = c;
16        if (i + 1 == n || this->s[out[i]] != this->s[out[i + 1]])
17            c++;
18    }
19    for (int h = 1; h < n && c < n; h <= 1) {
20        for (int i = 0; i < n; i++)
21            posBucket[out[i]] = bucket[i];
22        for (int i = n - 1; i >= 0; i--)
23            bpos[bucket[i]] = i;
24        for (int i = 0; i < n; i++) {
25            if (out[i] >= n - h)
26                temp[bpos[bucket[i]]++] = out[i];
27        }
28        for (int i = 0; i < n; i++) {
29            if (out[i] >= h)
30                temp[bpos[posBucket[out[i] - h]]++] = out[i] - h;
31        }
32        c = 0;
33        for (int i = 0; i + 1 < n; i++) {
34            int a = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h) ||
35                (posBucket[temp[i + 1] + h] != posBucket[temp[i] + h]);
36            bucket[i] = c;
37            c += a;
38        }
39        bucket[n - 1] = c++;
40        temp.swap(out);
41    }
42    return out;
43 }

```

9.12. 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 << 1ll) | (inv - '0');
75                 aux = aux->next[inv];

```

```

76     } else {
77         ans = (ans << 1ll) | (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 };

```