

C++ Competitive Programming Library

DO NOT DISCLOSE OR DISTRIBUTE

bfs.07 - Bernardo Flores Salmeron

1	Template	3	4.13	Subset Sum (Bitset)	22
2	.Vscode	3	5	Graphs	23
3	Data Structures	3	5.1	All Eulerian Path Or Tour	23
3.1	Bit	3	5.2	Articulation Points	24
3.2	Bit (Range Update)	4	5.3	Bellman Ford	25
3.3	Bit 2D	5	5.4	Bipartite Check	25
3.4	Merge Sort Tree	6	5.5	Block Cut Tree	26
3.5	Min Queue	6	5.6	Bridges	27
3.6	Mos Algorithm	6	5.7	Centroid	27
3.7	Ordered Set	7	5.8	Centroid Decomposition	27
3.8	Persistent Segment Tree	7	5.9	Compress ScCs In Dag	28
3.9	Segment Tree	8	5.10	Count (3-4) Cycles	28
3.10	Segment Tree 2D	9	5.11	Cycle Detection	29
3.11	Segment Tree Beats	10	5.12	De Bruijn Sequence	29
3.12	Segment Tree Polynomial	13	5.13	Dijkstra + Dij Graph	29
3.13	Sparse Table	14	5.14	Dinic	31
3.14	Sqrt Decomposition	14	5.15	Dsu	34
3.15	Treap	15	5.16	Dsu On Tree	34
3.16	Treap Maximum Contiguous Segment	18	5.17	Floyd Warshall	35
4	Dp	18	5.18	Functional Graph	35
4.1	Binary Lifting	18	5.19	Girth (Shortest Cycle In A Graph)	37
4.2	Catalan	19	5.20	Hld	37
4.3	Catalan 1 1 2 5 14 42 132	19	5.21	Hungarian	38
4.4	Cht Optimization	19	5.22	Lca	39
4.5	Digit Dp	20	5.23	Longest Path In Dag	40
4.6	Divide And Conquer Optimization	20	5.24	Maximum Independent Set (Set Of Vertices That Arent Directly Connected)	41
4.7	Edit Distance	21	5.25	Maximum Path Unweighted Graph	41
4.8	Knuth Optimization	21	5.26	Min Cost Flow Gpresso	41
4.9	Lis	21	5.27	Min Cost Flow Katcl	42
4.10	Longest Common Subsequence	21	5.28	Minimum Edge Cover (Set Of Edges That Are Adjacent To All Vertices)	43
4.11	Longest Increasing Subsequence 2D (Not Sorted)	22	5.29	Minimum Path Cover In Dag	43
4.12	Longest Increasing Subsequence 2D (Sorted)	22	5.30	Minimum Path Cover In Dag	43

5.31	Mst	43	7.8	Euler Totient	50
5.32	Number Of Different Spanning Trees In A Complete Graph	44	7.9	Extended Euclidean	50
5.33	Number Of Ways To Make A Graph Connected	44	7.10	Factorization	51
5.34	Pruffer Decode	44	7.11	Fft	51
5.35	Pruffer Encode	44	7.12	Inclusion Exclusion	51
5.36	Pruffer Properties	44	7.13	Inclusion Exclusion	51
5.37	Remove All Bridges From Graph	44	7.14	Karatsuba	52
5.38	Scc (Kosaraju)	44	7.15	Markov Chains	52
5.39	Small To Large (Merge Maps)	45	7.16	Matrix Exponentiation	53
5.40	Topological Sort	45	7.17	Matrix Exponentiation	53
5.41	Tree Diameter	46	7.18	Pollard Rho (Factorize)	53
5.42	Tree Distance	46	7.19	Pollard Rho (Find A Divisor)	54
5.43	Tree Isomorphism	46	7.20	Polynomial Convolution	55
6	Language Stuff	47	7.21	Primality Check	55
6.1	Binary String To Int	47	7.22	Primes	55
6.2	Check Char Type	47	7.23	Sieve + Segmented Sieve	55
6.3	Check Overflow	47	7.24	Stars And Bars	56
6.4	Counting Bits	47	8	Miscellaneous	56
6.5	Gen Random Numbers (Rng)	47	8.1	Counting Frequency Of Digits From 1 To K	56
6.6	Int To Binary String	47	8.2	Counting Number Of Digits Up To N	56
6.7	Int To String	47	8.3	Infix To Postfix	57
6.8	Permutation	47	8.4	Interval Scheduling	57
6.9	Print Int128 T	47	8.5	Interval Scheduling	57
6.10	Read And Write From File	48	8.6	Iterate Over Subsets Of Mask	57
6.11	Readint	48	8.7	Kadane	57
6.12	Rotate Left	48	8.8	Kadane (Segment Tree)	57
6.13	Rotate Right	48	8.9	Kadane 2D	58
6.14	Scanf From String	48	8.10	Largest Area In Histogram	59
6.15	Split Function	48	8.11	Point Compression	59
6.16	String To Long Long	48	8.12	Ternary Search	59
6.17	Substring	48	8.13	Tower Of Hanoi	59
6.18	Time Measure	48	8.14	Two Sat	60
6.19	Unique Vector	48	9	Stress Testing	60
6.20	Width	48	9.1	Check	60
7	Math	48	9.2	Gen	60
7.1	Bell Numbers	48	9.3	Run	62
7.2	Binary Exponentiation	49	10	Strings	62
7.3	Chinese Remainder Theorem	49	10.1	Aho Corasick	62
7.4	Combinatorics	49	10.2	Hashing	63
7.5	Diophantine Equation	50	10.3	Kmp	63
7.6	Divide Fraction	50	10.4	Lcs K Strings	64
7.7	Divisors	50	10.5	Lexicographically Smallest Rotation	64

10.6 Manacher (Longest Palindrome)	64
10.7 Suffix Array	65
10.8 Suffix Array Mine	66
10.9 Suffix Automaton	67
10.10 Trie	69
10.11 Z Function	70

1. Template

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define eb emplace_back
6 #define ii pair<int, int>
7 #define OK (cerr << "OK" << endl)
8 #define debug(x) cerr << #x " = " << (x) << endl
9 #define ff first
10 #define ss second
11 #define int long long
12 #define tt tuple<int, int, int>
13 #define all(x) x.begin(), x.end()
14 #define vi vector<int>
15 #define vii vector<pair<int, int>>
16 #define vvi vector<vector<int>>
17 #define vvii vector<vector<pair<int, int>>>
18 #define Mat(n, m, v) vector<vector<int>>(n, vector<int>(m, v))
19 #define endl '\n'
20
21 constexpr int INF = (sizeof(int) == 4 ? 1e9 : 2e18) + 1e5;
22 constexpr int MOD = 1e9 + 7;
23 constexpr int MAXN = 2e5 + 3;
24
25 #define MULTIPLE_TEST_CASES
26 void solve(const int test) {
27     int n;
28     cin >> n;
29 }
30
31 signed main() {
32     // const string FILE_NAME = "";
33     // freopen((FILE_NAME + string(".in")).c_str(), "r", stdin);
34     // freopen((FILE_NAME + string(".out")).c_str(), "w", stdout);
35     ios_base::sync_with_stdio(false);
36     cin.tie(nullptr), cout.tie(nullptr);
37
38     int t = 1;
39 #ifdef MULTIPLE_TEST_CASES
40     cin >> t;
41 #endif
42     for (int i = 1; i <= t; ++i)
43         solve(i);
44 }

```

2. .Vscode

3. Data Structures

3.1. Bit

```

1 // #define RANGE_SUM
2 // #define RANGE_UPDATE
3 // Uncomment ONLY ONE above!
4 // clang-format off
5 class BIT {
6 private:
7     vector<int> bit;
8     const int n, offset;
9
10 private:

```

```

11 int low(const int i) { return i & (-i); }
12
13 // Point update
14 void update(int i, const int delta) {
15     while (i <= n) {
16         bit[i] += delta;
17         i += low(i);
18     }
19 }
20
21 // Prefix query
22 int _query(int i) {
23     int sum = 0;
24     while (i > 0) {
25         sum += bit[i];
26         i -= low(i);
27     }
28     return sum;
29 }
30
31 void build(const vector<int> &arr) {
32     bit.resize(arr.size() + offset, 0);
33     for (int i = 1; i <= n; i++)
34         #ifdef RANGE_UPDATE
35             update(i - offset, i - offset, arr[i - offset]);
36         #endif
37         #ifdef RANGE_SUM
38             update(i - offset, arr[i - offset]);
39         #endif
40 }
41
42 public:
43     /// Constructor responsible for initializing the tree with 0's.
44     ///
45     /// Time Complexity: O(n log n)
46     BIT(const vector<int> &arr, const int indexed_from)
47         : n(arr.size() - indexed_from), offset(indexed_from ^ 1) {
48         assert(indexed_from == 0 || indexed_from == 1);
49         build(arr);
50     }
51
52     /// Constructor responsible for building the tree based on a vector.
53     ///
54     /// Time Complexity O(n)
55     BIT(const int n, const int indexed_from) : n(n), offset(indexed_from ^ 1) {
56         bit.resize(n + 1, 0);
57     }
58
59     #ifdef RANGE_UPDATE
60     void update(int l, int r, const int val) {
61         l += offset, r += offset;
62         assert(l <= 1), assert(l <= r), assert(r <= n);
63         _update(l, val);
64         _update(r + 1, -val);
65     }
66     #endif
67
68     #ifdef RANGE_SUM
69     /// Update at a single index.
70     ///
71     /// Time Complexity O(log n)
72     void update(int idx, const int delta) {
73         idx += offset;
74         assert(l <= idx), assert(idx <= n);
75         _update(idx, delta);

```

```

76     }
77     #endif
78
79     #ifdef RANGE_UPDATE
80     /// Query at a single index.
81     ///
82     /// Time Complexity O(log n)
83     int query(int idx) {
84         idx += offset;
85         assert(1 <= idx), assert(idx <= n);
86         return _query(idx);
87     }
88     #endif
89
90     #ifdef RANGE_SUM
91     /// Range query from l to r.
92     ///
93     /// Time Complexity O(log n)
94     int query(int l, int r) {
95         l += offset, r += offset;
96         assert(1 <= l), assert(l <= r), assert(r <= n);
97         return _query(r) - _query(l - 1);
98     }
99     #endif
100 };
101 // clang-format on

```

3.2. Bit (Range Update)

```

1  /// INDEX THE ARRAY BY 1!!!
2  class BIT {
3  private:
4      vector<int> bit1, bit2;
5      int n;
6
7  private:
8      int low(int i) { return i & (-i); }
9
10     // Point update
11     void update(int i, const int delta, vector<int> &bit) {
12         while (i <= n) {
13             bit[i] += delta;
14             i += low(i);
15         }
16     }
17
18     // Prefix query
19     int query(int i, const vector<int> &bit) {
20         int sum = 0;
21         while (i > 0) {
22             sum += bit[i];
23             i -= low(i);
24         }
25         return sum;
26     }
27
28     // Builds the bit
29     void build(const vector<int> &arr) {
30         // OBS: BIT IS INDEXED FROM 1
31         // THE USAGE OF 1-BASED ARRAY IS MANDATORY
32         assert(arr.front() == 0);
33         this->n = (int)arr.size() - 1;
34         bit1.resize(arr.size(), 0);
35         bit2.resize(arr.size(), 0);

```

```

36     for (int i = 1; i <= n; i++)
37         update(i, arr[i]);
38     }
39 }
40
41 public:
42     /// Constructor responsible for initializing the tree with 0's.
43     ///
44     /// Time Complexity: O(n log n)
45     BIT(const vector<int> &arr) { build(arr); }
46
47     /// Constructor responsible for building the tree based on a vector.
48     ///
49     /// Time Complexity O(n)
50     BIT(const int n) {
51         // OBS: BIT IS INDEXED FROM 1
52         // THE USAGE OF 1-INDEXED ARRAY IS MANDATORY
53         this->n = n;
54         bit1.resize(n + 1, 0);
55         bit2.resize(n + 1, 0);
56     }
57
58     /// Range update from l to r.
59     ///
60     /// Time Complexity O(log n)
61     void update(const int l, const int r, const int delta) {
62         assert(l <= l), assert(l <= r), assert(r <= n);
63         update(l, delta, bit1);
64         update(r + 1, -delta, bit1);
65         update(l, delta * (l - 1), bit2);
66         update(r + 1, -delta * r, bit2);
67     }
68
69     /// Update at a single index.
70     ///
71     /// Time Complexity O(log n)
72     void update(const int i, const int delta) {
73         assert(l <= i), assert(i <= n);
74         update(i, i, delta);
75     }
76
77     /// Range query from l to r.
78     ///
79     /// Time Complexity O(log n)
80     int query(const int l, const int r) {
81         assert(l <= l), assert(l <= r), assert(r <= n);
82         return query(r) - query(l - 1);
83     }
84
85     /// Prefix query from l to i.
86     ///
87     /// Time Complexity O(log n)
88     int query(const int i) {
89         assert(i <= n);
90         return (query(i, bit1) * i) - query(i, bit2);
91     }
92 };

```

3.3. Bit 2D

```

1 // INDEX BY ONE ALWAYS!!!
2 class BIT_2D {
3 private:
4     // row, column

```

```

5     const int n, m;
6     vector<vector<int>> tree;
7
8 private:
9     // Returns an integer which contains only the least significant bit.
10    int low(const int i) { return i & (-i); }
11
12    void _update(const int x, const int y, const int delta) {
13        for (int i = x; i < n; i += low(i))
14            for (int j = y; j < m; j += low(j))
15                tree[i][j] += delta;
16    }
17
18    int _query(const int x, const int y) {
19        int ans = 0;
20        for (int i = x; i > 0; i -= low(i))
21            for (int j = y; j > 0; j -= low(j))
22                ans += tree[i][j];
23        return ans;
24    }
25
26 public:
27     // put the size of the array without 1 indexing.
28     /// Time Complexity: O(n * m)
29     BIT_2D(const int n, const int m) : n(n + 1), m(m + 1) {
30         tree.resize(this->n, vector<int>(this->m, 0));
31     }
32
33     /// Time Complexity: O(n * m * (log(n) + log(m)))
34     BIT_2D(const vector<vector<int>> &mat)
35         : n(mat.size()), m(mat.front().size()) {
36         // Check if it is 1 indexed.
37         assert(mat[0][0] == 0);
38         tree.resize(n, vector<int>(m, 0));
39         for (int i = 1; i < n; i++)
40             for (int j = 1; j < m; j++)
41                 _update(i, j, mat[i][j]);
42     }
43
44     /// Query from (1, 1) to (x, y).
45     ///
46     /// Time Complexity: O(log(n) + log(m))
47     int prefix_query(const int x, const int y) {
48         assert(0 < x), assert(x < n);
49         assert(0 < y), assert(y < m);
50         return _query(x, y);
51     }
52
53     /// Query from (x1, y1) to (x2, y2).
54     ///
55     /// Time Complexity: O(log(n) + log(m))
56     int query(const int x1, const int y1, const int x2, const int y2) {
57         assert(0 < x1), assert(x1 <= x2), assert(x2 < n);
58         assert(0 < y1), assert(y1 <= y2), assert(y2 < m);
59         return _query(x2, y2) - _query(x1 - 1, y2) - _query(x2, y1 - 1) +
60             _query(x1 - 1, y1 - 1);
61     }
62
63     /// Updates point (x, y).
64     ///
65     /// Time Complexity: O(log(n) + log(m))
66     void update(const int x, const int y, const int delta) {
67         assert(0 < x), assert(x < n);
68         assert(0 < y), assert(y < m);
69         _update(x, y, delta);

```

```
70 }
71 };
```

3.4. Merge Sort Tree

```
1 // Returns the amount of numbers greater than k from i to 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) +
19         query(mid + 1, r, i, j, k, 2 * pos + 2);
20 }
21
22 void build(int l, int r, int pos) {
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 }
```

3.5. Min Queue

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

```
21     d.emplace_back(val, idx);
22 }
23
24 int min_element() { return d.front().first; }
25 };
```

3.6. Mos Algorithm

```
1 struct Tree {
2     int l, r, ind;
3 };
4 Tree query[311111];
5 int arr[311111];
6 int freq[1111111];
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 void del(int pos) {
23     freq[arr[pos]]--;
24     if (freq[arr[pos]] == 0)
25         cont--;
26 }
27 int main() {
28     int n;
29     cin >> n;
30     block = sqrt(n);
31
32     for (int i = 0; i < n; i++) {
33         cin >> arr[i];
34         freq[arr[i]] = 0;
35     }
36
37     int m;
38     cin >> m;
39
40     for (int i = 0; i < m; i++) {
41         cin >> query[i].l >> query[i].r;
42         query[i].l--, query[i].r--;
43         query[i].ind = i;
44     }
45     sort(query, query + m, cmp);
46
47     int s, e;
48     s = e = query[0].l;
49     add(s);
50     for (int i = 0; i < m; i++) {
51         while (s > query[i].l)
52             add(--s);
53         while (s < query[i].l)
54             del(++s);
55         while (e < query[i].r)
56             add(++e);
```

```

57     while (e > query[i].r)
58         del(e--);
59     ans[query[i].ind] = cont;
60 }
61 for (int i = 0; i < m; i++)
62     cout << ans[i] << endl;
63 }

```

3.7. Ordered Set

```

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

```

3.8. Persistent Segment Tree

```

1  class Persistent_Seg_Tree {
2  public:
3      struct Node {
4          int val;
5          Node *left, *right;
6          Node(const int v) : val(v), left(nullptr), right(nullptr) {}
7      };
8
9  private:
10     const Node NEUTRAL_NODE = Node(0);
11     int merge_nodes(const int x, const int y) { return x + y; }
12
13 private:
14     const int n;
15     vector<Node *> version = {nullptr};
16
17 public:
18     /// Builds version[0] with the values in the array.
19     ///

```

```

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, idx, delta);
53     cur_tree->right =
54         _update(cur_tree->right, prev_tree ? prev_tree->right : nullptr,
55               mid + 1, r, idx, delta);
56     cur_tree->val =
57         merge_nodes(cur_tree->left ? cur_tree->left->val : NEUTRAL_NODE.val,
58               cur_tree->right ? cur_tree->right->val :
59               NEUTRAL_NODE.val);
60     return cur_tree;
61 }
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) {}

```

```

81
82 /// Constructor that initializes the segment tree empty. It's allowed to
    query
83 /// from 0 to MAXN - 1.
84 ///
85 /// Time Complexity: O(1)
86 Persistent_Seg_Tree(const int MAXN) : n(MAXN) {}
87
88 /// Constructor that allows to pass initial values to the leafs. It's
    allowed
89 /// to query from 0 to n - 1.
90 ///
91 /// Time Complexity: O(n)
92 Persistent_Seg_Tree(const vector<int> &arr) : n(arr.size()) {
93     this->version[0] = this->build(this->version[0], 0, this->n - 1, arr);
94 }
95
96 /// Links the root of a version to a previous version.
97 ///
98 /// Time Complexity: O(1)
99 void link(const int version, const int prev_version) {
100     assert(this->n > -1);
101     assert(0 <= prev_version, assert(prev_version <= version);
102     this->create_version(version);
103     this->version[version] = this->version[prev_version];
104 }
105
106 /// 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,
    const int delta) {
110     assert(this->n > -1);
111     assert(0 <= prev_version, assert(prev_version <= cur_version);
112     this->create_version(cur_version);
113     this->version[cur_version] =
114         this->update(this->version[cur_version],
115             this->version[prev_version],
116             0, this->n - 1, idx, delta);
117 }
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 };

```

3.9. Segment Tree

```

1 class Seg_Tree {
2 public:
3     struct Node {
4         int val, lazy;
5
6         Node() {}
7         Node(const int val) : val(val), lazy(0) {}
8     };
9
10 private:
11     // // Range Sum

```

```

12 // Node NEUTRAL_NODE = Node(0);
13 // Node merge_nodes(const Node &x, const Node &y) {
14 //     return Node(x.val + y.val);
15 // };
16 //
17 // void apply_lazy(const int l, const int r, const int pos) {
18 //     // for set change this to =
19 //     tree[pos].val += (r - l + 1) * tree[pos].lazy;
20 // }
21
22 // // RMQ Max
23 // Node NEUTRAL_NODE = Node(-INF);
24 // Node merge_nodes(const Node &x, const Node &y) {
25 //     return Node(max(x.val, y.val));
26 // }
27 // void apply_lazy(const int l, const int r, const int pos) {
28 //     tree[pos].val += tree[pos].lazy;
29 // }
30
31 // // RMQ Min
32 // Node NEUTRAL_NODE = Node(INF);
33 // Node merge_nodes(const Node &x, const Node &y) {
34 //     return Node(min(x.val, y.val));
35 // }
36 // void apply_lazy(const int l, const int r, const int pos) {
37 //     tree[pos].val += tree[pos].lazy;
38 // }
39
40 // // XOR
41 // // Only works with point updates
42 // Node NEUTRAL_NODE = Node(0);
43 // Node merge_nodes(const Node &x, const Node &y) {
44 //     return Node(x.val ^ y.val);
45 // };
46 //
47 // void apply_lazy(const int l, const int r, const int pos) {}
48
49 private:
50     int n;
51
52 public:
53     vector<Node> tree;
54
55 private:
56     void propagate(const int l, const int r, const int pos) {
57         if (tree[pos].lazy != 0) {
58             apply_lazy(l, r, pos);
59             if (l != r) {
60                 // for set change this to =
61                 tree[2 * pos + 1].lazy += tree[pos].lazy;
62                 tree[2 * pos + 2].lazy += tree[pos].lazy;
63             }
64             tree[pos].lazy = 0;
65         }
66     }
67
68 Node _build(const int l, const int r, const vector<int> &arr, const int
    pos) {
69     if (l == r)
70         return tree[pos] = Node(arr[l]);
71
72     int mid = (l + r) / 2;
73     return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
74         _build(mid + 1, r, arr, 2 * pos + 2));
75 }

```



```

76
77 int _get_first(const int l, const int r, const int i, const int j,
78               const int v, const int pos) {
79     propagate(l, r, pos);
80
81     if (l > r || l > j || r < i)
82         return -1;
83     // Needs RMQ MAX
84     // Replace to <= for greater or (with RMQ MIN) > for smaller or
85     // equal or >= for smaller
86     if (tree[pos].val < v)
87         return -1;
88
89     if (l == r)
90         return l;
91
92     int mid = (l + r) / 2;
93     int aux = _get_first(l, mid, i, j, v, 2 * pos + 1);
94     if (aux != -1)
95         return aux;
96     return _get_first(mid + 1, r, i, j, v, 2 * pos + 2);
97 }
98
99 Node _query(const int l, const int r, const int i, const int j,
100            const int pos) {
101     propagate(l, r, pos);
102
103     if (l > r || l > j || r < i)
104         return NEUTRAL_NODE;
105
106     if (i <= l && r <= j)
107         return tree[pos];
108
109     int mid = (l + r) / 2;
110     return merge_nodes(_query(l, mid, i, j, 2 * pos + 1),
111                       _query(mid + 1, r, i, j, 2 * pos + 2));
112 }
113
114 // It adds a number delta to the range from i to j
115 Node _update(const int l, const int r, const int i, const int j,
116             const int delta, const int pos) {
117     propagate(l, r, pos);
118
119     if (l > r || l > j || r < i)
120         return tree[pos];
121
122     if (i <= l && r <= j) {
123         tree[pos].lazy = delta;
124         propagate(l, r, pos);
125         return tree[pos];
126     }
127
128     int mid = (l + r) / 2;
129     return tree[pos] =
130         merge_nodes(_update(l, mid, i, j, delta, 2 * pos + 1),
131                   _update(mid + 1, r, i, j, delta, 2 * pos + 2));
132 }
133
134 void build(const vector<int> &arr) {
135     this->tree.resize(4 * this->n);
136     this->_build(0, this->n - 1, arr, 0);
137 }
138
139 public:
140     /// N equals to -1 means the Segment Tree hasn't been created yet.

```

```

141 Seg_Tree() : n(-1) {}
142
143     /// Constructor responsible for initializing the tree with val.
144     ///
145     /// Time Complexity O(n)
146     Seg_Tree(const int n, const int val = 0) : n(n) {
147         this->tree.resize(4 * this->n, Node(val));
148     }
149
150     /// Constructor responsible for building the tree based on a vector.
151     ///
152     /// Time Complexity O(n)
153     Seg_Tree(const vector<int> &arr) : n(arr.size()) { this->build(arr); }
154
155     /// Returns the first index from i to j compared to v.
156     /// Uncomment the line in the original function to get the proper element
157     /// that
158     /// may be: GREATER OR EQUAL, GREATER, SMALLER OR EQUAL, SMALLER.
159     ///
160     /// Time Complexity O(log n)
161     int get_first(const int i, const int j, const int v) {
162         assert(this->n >= 0);
163         return this->_get_first(0, this->n - 1, i, j, v, 0);
164     }
165
166     /// Update at a single index.
167     ///
168     /// Time Complexity O(log n)
169     void update(const int idx, const int delta) {
170         assert(this->n >= 0);
171         assert(0 <= idx), assert(idx < this->n);
172         this->_update(0, this->n - 1, idx, idx, delta, 0);
173     }
174
175     /// Range update from l to r.
176     ///
177     /// Time Complexity O(log n)
178     void update(const int l, const int r, const int delta) {
179         assert(this->n >= 0);
180         assert(0 <= l), assert(l <= r), assert(r < this->n);
181         this->_update(0, this->n - 1, l, r, delta, 0);
182     }
183
184     /// Query at a single index.
185     ///
186     /// Time Complexity O(log n)
187     int query(const int idx) {
188         assert(this->n >= 0);
189         assert(0 <= idx), assert(idx < this->n);
190         return this->_query(0, this->n - 1, idx, idx, 0).val;
191     }
192
193     /// Range query from l to r.
194     ///
195     /// Time Complexity O(log n)
196     int query(const int l, const int r) {
197         assert(this->n >= 0);
198         assert(0 <= l), assert(l <= r), assert(r < this->n);
199         return this->_query(0, this->n - 1, l, r, 0).val;
200     };

```

3.10. 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,
30                  const 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,
44                 const 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,
57                   const 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() { this->n = -1, this->m = -1; }
75
76     Seg_Tree_2d(const int n, const int m) {
77         this->n = n, this->m = m;
78         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
79         assert(m < 10000);
80         tree.resize(4 * n, Seg_Tree(m));
81     }
82
83     Seg_Tree_2d(const int n, const int m, const vector<vector<int>> &mat) {
84         this->n = n, this->m = m;
85         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
86         assert(m < 10000);
87         tree.resize(4 * n, Seg_Tree(m));
88         st_build(0, n - 1, 0, mat);
89     }
90
91     /// Query from (x1, y1) to (x2, y2).
92     ///
93     /// Time complexity: O((log n) * (log m))
94     int query(const int x1, const int y1, const int x2, const int y2) {
95         assert(this->n > -1);
96         assert(0 <= x1, assert(x1 <= x2), assert(x2 < this->n);
97         assert(0 <= y1, assert(y1 <= y2), assert(y2 < this->m);
98         return st_query(0, this->n - 1, x1, y1, x2, y2, 0);
99     }
100
101     /// Point updates on position (x, y).
102     ///
103     /// Time complexity: O((log n) * (log m))
104     void update(const int x, const int y, const int delta) {
105         assert(0 <= x), assert(x < this->n);
106         assert(0 <= y), assert(y < this->m);
107         st_update(0, this->n - 1, x, y, delta, 0);
108     }
109 };

```

3.11. Segment Tree Beats

```

1 #define MIN_UPDATE // supports for i in [l, r] do a[i] = min(a[i], x)
2 #define MAX_UPDATE // supports for i in [l, r] do a[i] = max(a[i], x)
3 #define ADD_UPDATE // supports for i in [l, r] a[i] += x
4
5 // clang-format off
6 class Seg_Tree_Beats {
7     const static int INF = (sizeof(int) == 4 ? 1e9 : 2e18) + 1e5;
8
9 public:
10     struct Node {
11         int sum;
12         #ifdef ADD_UPDATE
13         int lazy = 0;
14         #endif
15         #ifdef MIN_UPDATE
16         // Stores the maximum value, its frequency, and 2nd max value.
17         int maxx, cnt_maxx, smaxx;

```

```

18     #endif
19     #ifdef MAX_UPDATE
20     // Stores the minimum value, its frequency, and 2nd min value.
21     int minn, cnt_minn, sminn;
22     #endif
23     Node() {}
24     Node(const int val) : sum(val) {
25         #ifdef MIN_UPDATE
26         maxx = val, cnt_maxx = 1, smaxx = -INF;
27         #endif
28         #ifdef MAX_UPDATE
29         minn = val, cnt_minn = 1, sminn = INF;
30         #endif
31     }
32 };
33
34 private:
35     // Range Sum
36     Node merge_nodes(const Node &x, const Node &y) {
37         Node node;
38         node.sum = x.sum + y.sum;
39
40         #ifdef MIN_UPDATE
41         node.maxx = max(x.maxx, y.maxx);
42         node.smaxx = max(x.smaxx, y.smaxx);
43         node.cnt_maxx = 0;
44         if (node.maxx == x.maxx)
45             node.cnt_maxx += x.cnt_maxx;
46         else
47             node.smaxx = max(node.smaxx, x.maxx);
48         if (node.maxx == y.maxx)
49             node.cnt_maxx += y.cnt_maxx;
50         else
51             node.smaxx = max(node.smaxx, y.maxx);
52         #endif
53
54         #ifdef MAX_UPDATE
55         node.minn = min(x.minn, y.minn);
56         node.sminn = min(x.sminn, y.sminn);
57         node.cnt_minn = 0;
58         if (node.minn == x.minn)
59             node.cnt_minn += x.cnt_minn;
60         else
61             node.sminn = min(node.sminn, x.minn);
62         if (node.minn == y.minn)
63             node.cnt_minn += y.cnt_minn;
64         else
65             node.sminn = min(node.sminn, y.minn);
66         #endif
67         return node;
68     }
69
70 private:
71     int n;
72
73 public:
74     vector<Node> tree;
75
76 private:
77     #ifdef MIN_UPDATE
78     // in queries a[i] = min(a[i], x)
79     void apply_update_min(const int pos, const int x) {
80         Node &node = tree[pos];
81         node.sum -= (node.maxx - x) * node.cnt_maxx;
82         #ifdef MAX_UPDATE

```

```

83         if (node.maxx == node.minn)
84             node.minn = x;
85         else if (node.maxx == node.sminn)
86             node.sminn = x;
87         #endif
88         node.maxx = x;
89     }
90 #endif
91
92 #ifdef MAX_UPDATE
93 void apply_update_max(const int pos, const int x) {
94     Node &node = tree[pos];
95     node.sum += (x - node.minn) * node.cnt_minn;
96     #ifdef MIN_UPDATE
97     if (node.minn == node.maxx)
98         node.maxx = x;
99     else if (node.minn == node.smaxx)
100         node.smaxx = x;
101     #endif
102     node.minn = x;
103 }
104 #endif
105
106 #ifdef ADD_UPDATE
107 void apply_update_sum(const int l, const int r, const int pos, const int
108 v) {
109     tree[pos].sum += (r - l + 1) * v;
110     #ifdef ADD_UPDATE
111     tree[pos].lazy += v;
112     #endif
113     #ifdef MIN_UPDATE
114     tree[pos].maxx += v;
115     tree[pos].smaxx += v;
116     #endif
117     #ifdef MAX_UPDATE
118     tree[pos].minn += v;
119     tree[pos].sminn += v;
120     #endif
121 }
122 #endif
123
124 void propagate(const int l, const int r, const int pos) {
125     if (l == r)
126         return;
127     Node &node = tree[pos];
128     const int c1 = 2 * pos + 1, c2 = 2 * pos + 2;
129
130     #ifdef ADD_UPDATE
131     if (node.lazy != 0) {
132         const int mid = (l + r) / 2;
133         apply_update_sum(l, mid, c1, node.lazy);
134         apply_update_sum(mid + 1, r, c2, node.lazy);
135         node.lazy = 0;
136     }
137     #endif
138
139     #ifdef MIN_UPDATE
140     // min update
141     if (tree[c1].maxx > node.maxx)
142         apply_update_min(c1, node.maxx);
143     if (tree[c2].maxx > node.maxx)
144         apply_update_min(c2, node.maxx);
145     #endif
146
147     #ifdef MAX_UPDATE

```

```

147 // max_update
148 if (tree[c1].minn < node.minn)
149     apply_update_max(c1, node.minn);
150 if (tree[c2].minn < node.minn)
151     apply_update_max(c2, node.minn);
152 #endif
153 }
154
155 Node _build(const int l, const int r, const vector<int> &arr, const int
pos) {
156     if (l == r)
157         return tree[pos] = Node(arr[l]);
158
159     const int mid = (l + r) / 2;
160     return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
161                                   _build(mid + 1, r, arr, 2 * pos + 2));
162 }
163
164 Node _query(const int l, const int r, const int i, const int j, const int
pos,
165            const Node &NEUTRAL_NODE) {
166     propagate(l, r, pos);
167
168     if (l > r || l > j || r < i)
169         return NEUTRAL_NODE;
170
171     if (i <= l && r <= j)
172         return tree[pos];
173
174     const int mid = (l + r) / 2;
175     return merge_nodes(_query(l, mid, i, j, 2 * pos + 1, NEUTRAL_NODE),
176                       _query(mid + 1, r, i, j, 2 * pos + 2, NEUTRAL_NODE));
177 }
178
179 #ifdef ADD_UPDATE
180 Node _update_sum(const int l, const int r, const int i, const int j,
181                 const int v, const int pos) {
182     propagate(l, r, pos);
183
184     if (l > r || l > j || r < i)
185         return tree[pos];
186
187     if (i <= l && r <= j) {
188         apply_update_sum(l, r, pos, v);
189         return tree[pos];
190     }
191
192     int mid = (l + r) / 2;
193     return tree[pos] =
194         merge_nodes(_update_sum(l, mid, i, j, v, 2 * pos + 1),
195                   _update_sum(mid + 1, r, i, j, v, 2 * pos + 2));
196 }
197 #endif
198
199 #ifdef MIN_UPDATE
200 Node _update_min(const int l, const int r, const int i, const int j,
201                 const int x, const int pos) {
202     propagate(l, r, pos);
203
204     if (l > r || l > j || r < i || tree[pos].maxx <= x)
205         return tree[pos];
206
207     if (i <= l && r <= j && tree[pos].smaxx < x) {
208         apply_update_min(pos, x);
209         return tree[pos];

```

```

210     }
211
212     const int mid = (l + r) / 2;
213     return tree[pos] =
214         merge_nodes(_update_min(l, mid, i, j, x, 2 * pos + 1),
215                   _update_min(mid + 1, r, i, j, x, 2 * pos + 2));
216 }
217 #endif
218
219 #ifndef MAX_UPDATE
220 Node _update_max(const int l, const int r, const int i, const int j,
221                 const int x, const int pos) {
222     propagate(l, r, pos);
223
224     if (l > r || l > j || r < i || tree[pos].minn >= x)
225         return tree[pos];
226
227     if (i <= l && r <= j && tree[pos].sminn > x) {
228         apply_update_max(pos, x);
229         return tree[pos];
230     }
231
232     const int mid = (l + r) / 2;
233     return tree[pos] =
234         merge_nodes(_update_max(l, mid, i, j, x, 2 * pos + 1),
235                   _update_max(mid + 1, r, i, j, x, 2 * pos + 2));
236 }
237 #endif
238
239 void build(const vector<int> &arr) {
240     this->tree.resize(4 * this->n);
241     this->_build(0, this->n - 1, arr, 0);
242 }
243
244 public:
245     /// N equals to -1 means the Segment Tree hasn't been created yet.
246     Seg_Tree Beats() : n(-1) {}
247
248     /// Constructor responsible for initializing the tree with 0's.
249     ///
250     /// Time Complexity O(n)
251     Seg_Tree Beats(const int n) : n(n) {
252         this->tree.resize(4 * this->n, Node(0));
253     }
254
255     /// Constructor responsible for building the tree based on a vector.
256     ///
257     /// Time Complexity O(n)
258     Seg_Tree Beats(const vector<int> &arr) : n(arr.size()) { this->build(arr);
259     }
260
261     #ifdef ADD_UPDATE
262     /// Range update from l to r.
263     /// Type: for i in range [l, r] do a[i] += x
264     void update_sum(const int l, const int r, const int x) {
265         assert(this->n >= 0);
266         assert(0 <= l), assert(l <= r), assert(r < this->n);
267         this->_update_sum(0, this->n - 1, l, r, x, 0);
268     }
269 #endif
270
271     #ifdef MIN_UPDATE
272     /// Range update from l to r.
273     /// Type: for i in range [l, r] do a[i] = min(a[i], x)
274     void update_min(const int l, const int r, const int x) {

```

```

274     assert(this->n >= 0);
275     assert(0 <= l), assert(l <= r), assert(r < this->n);
276     this->_update_min(0, this->n - 1, l, r, x, 0);
277 }
278 #endif
279
280 #ifdef MAX_UPDATE
281 /// Range update from l to r.
282 /// Type: for i in range [l, r] do a[i] = max(a[i], x)
283 void update_max(const int l, const int r, const int x) {
284     assert(this->n >= 0);
285     assert(0 <= l), assert(l <= r), assert(r < this->n);
286     this->_update_max(0, this->n - 1, l, r, x, 0);
287 }
288 #endif
289
290 /// Range Sum query from l to r.
291 ///
292 /// Time Complexity O(log n)
293 int query_sum(const int l, const int r) {
294     assert(this->n >= 0);
295     assert(0 <= l), assert(l <= r), assert(r < this->n);
296     return this->_query(0, this->n - 1, l, r, 0, Node(0)).sum;
297 }
298
299 #ifdef MAX_UPDATE
300 /// Range Min query from l to r.
301 ///
302 /// Time Complexity O(log n)
303 int query_min(const int l, const int r) {
304     assert(this->n >= 0);
305     assert(0 <= l), assert(l <= r), assert(r < this->n);
306     return this->_query(0, this->n - 1, l, r, 0, Node(INF)).minn;
307 }
308 #endif
309
310 #ifdef MIN_UPDATE
311 /// Range Max query from l to r.
312 ///
313 /// Time Complexity O(log n)
314 int query_max(const int l, const int r) {
315     assert(this->n >= 0);
316     assert(0 <= l), assert(l <= r), assert(r < this->n);
317     return this->_query(0, this->n - 1, l, r, 0, Node(-INF)).maxx;
318 }
319 #endif
320 };
321 // clang-format on
322 // OBS: Q updates of the type a[i] = (min/max)(a[i], x) have the amortized
323 // complexity of O(q * (log(n) ^ 2)).

```

3.12. 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     void apply_lazy(const int l, const int r, const int pos) {
19         tree[pos].val += (r - l + 1) * tree[pos].z0;
20         tree[pos].val += (r - l) * (r - l + 1) / 2 * tree[pos].z1;
21     }
22
23 private:
24     int n;
25
26 public:
27     vector<Node> tree;
28
29 private:
30     void st_propagate(const int l, const int r, const int pos) {
31         if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {
32             apply_lazy(l, r, pos);
33             int mid = (l + r) / 2;
34             int sz_left = mid - l + 1;
35             if (l != r) {
36                 tree[2 * pos + 1].z0 += tree[pos].z0;
37                 tree[2 * pos + 1].z1 += tree[pos].z1;
38
39                 tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
40                 tree[2 * pos + 2].z1 += tree[pos].z1;
41             }
42             tree[pos].z0 = 0;
43             tree[pos].z1 = 0;
44         }
45     }
46
47     Node st_build(const int l, const int r, const vector<int> &arr,
48                  const int pos) {
49         if (l == r)
50             return tree[pos] = Node(arr[l], 0, 0);
51
52         int mid = (l + r) / 2;
53         return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
54                                       st_build(mid + 1, r, arr, 2 * pos + 2));
55     }
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 };

```

3.13. 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*log(n))
22     void build_sparse_table(const vector<int> &arr) {
23         table.resize(lg[this->n] + 1, vector<int>(this->n));
24
25         table[0] = arr;
26         int pow2 = 1;
27         for (int i = 1; i < table.size(); i++) {
28             const int lastsz = this->n - pow2 + 1;
29             for (int j = 0; j + pow2 < lastsz; j++)
30                 table[i][j] = merge(table[i - 1][j], table[i - 1][j + pow2]);
31             pow2 <= 1;
32         }
33     }
34
35 public:
36     /// Constructor that builds the log array and the sparse table.
37     ///
38     /// Time Complexity: O(n*log(n))
39     Sparse_Table(const vector<int> &arr) : n(arr.size()) {
40         this->build_log_array();
41         this->build_sparse_table(arr);
42     }
43
44     void print() {
45         int pow2 = 1;
46         for (int i = 0; i < table.size(); i++) {
47             const int sz = (int)(table.front().size()) - pow2 + 1;
48             for (int j = 0; j < sz; j++)
49                 cout << table[i][j] << " \n"[(j + 1) == sz];
50             pow2 <= 1;
51         }
52     }
53
54     /// Range query from l to r.
55     ///
56     /// Time Complexity: O(1)
57     int query(const int l, const int r) {
58         assert(0 <= l), assert(l <= r), assert(r < this->n);
59         int lgg = lg[r - l + 1];
60         return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
61     }
62 };

```

3.14. Sqrt Decomposition

```

1 // Problem: Sum from l to r
2 // See also MO'S ALGORITHM
3 // -----

```

```

4 int getId(int indx, int blockSZ) { return indx / blockSZ; }
5
6 void init(int sz) {
7     for (int i = 0; i <= sz; i++)
8         BLOCK[i] = inf;
9 }
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 }

```

3.15. Treap

```

1 // PLEASE DO NOT COPY!
2
3 // clang-format off
4 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
5 // #define REVERSE
6 // #define LAZY
7 class Treap {
8 public:
9     struct Node {
10         Node *left = nullptr, *right = nullptr, *par = nullptr;
11         // Priority to be used in the treap
12         const int rank;
13         int size = 1, val;
14         // Contains the result of the range query between the node and its
15         // children.
16         int ans;
17         #ifdef LAZY
18         int lazy = 0;
19         #endif
20         #ifdef REVERSE
21         bool rev = false;
22         #endif
23
24         Node(const int val) : val(val), ans(val), rank(rng()) {}
25         Node(const int val, const int rank) : val(val), ans(val), rank(rank) {}
26     };
27 private:
28     vector<Node*> nodes;
29     int _size = 0;
30     Node *root = nullptr;
31
32 private:
33     // // Range Sum
34     // void merge_nodes(Node *node) {
35     //     node->ans = node->val;
36     //     if (node->left)
37     //         node->ans += node->left->ans;
38     //     if (node->right)
39     //         node->ans += node->right->ans;
40     // }
41

```

```

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

```

```

107     }
108 }
109
110 #ifdef REVERSE
111 void propagate_reverse(Node *node) {
112     if (node && node->rev) {
113         apply_reverse(node);
114         if (node->left)
115             node->left->rev ^= 1;
116         if (node->right)
117             node->right->rev ^= 1;
118         node->rev = 0;
119     }
120 }
121 #endif
122
123 #ifdef LAZY
124 void propagate_lazy(Node *node) {
125     if (node && node->lazy != 0) {
126         apply_lazy(node);
127         if (node->left)
128             node->left->lazy += node->lazy;
129         if (node->right)
130             node->right->lazy += node->lazy;
131         node->lazy = 0;
132     }
133 }
134 #endif
135
136 void update_node(Node *node) {
137     if (node) {
138         update_size(node);
139         #ifdef LAZY
140             propagate_lazy(node->left);
141             propagate_lazy(node->right);
142         #endif
143         #ifdef REVERSE
144             propagate_reverse(node->left);
145             propagate_reverse(node->right);
146         #endif
147         merge_nodes(node);
148     }
149 }
150
151 /// Splits the treap into to different treaps that contains nodes with
152     indexes
153 /// <= pos and indexes > pos. The nodes l and r contains, in the end, these
154 /// two different treaps.
155 void split(Node *node, Node *l, Node *r, const int pos, Node *pl =
156     nullptr, Node *pr = nullptr) {
157     if (!node)
158         l = r = nullptr;
159     else {
160         #ifdef LAZY
161             propagate_lazy(node);
162         #endif
163         #ifdef REVERSE
164             propagate_reverse(node);
165         #endif
166         if (get_size(node->left) <= pos) {
167             node->par = pr;
168             split(node->right, node->right, r, pos - get_size(node->left) - 1,
169                 pl,

```

```

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

```



```

234     return ans;
235 }
236
237 Node *_query(const int l, const int r) {
238     Node *L, *M, *R;
239     split(this->root, L, M, l - 1);
240     split(M, M, R, r - 1);
241     Node *ret = new Node(*M);
242     merge(L, L, M);
243     merge(root, L, R);
244     return ret;
245 }
246
247 void _update(const int l, const int r, const int delta) {
248     Node *L, *M, *R;
249     split(this->root, L, M, l - 1);
250     split(M, M, R, r - 1);
251
252     Node *node = M;
253     #ifdef LAZY
254     node->lazy = delta;
255     propagate_lazy(node);
256     #else
257     node->val += delta;
258     #endif
259
260     merge(L, L, M);
261     merge(root, L, R);
262 }
263
264 void _insert(const int pos, Node *node) {
265     this->_size += node->size;
266     Node *L, *R;
267     split(this->root, L, R, pos - 1);
268     merge(L, L, node);
269     merge(this->root, L, R);
270 }
271
272 Node *_erase(const int l, const int r) {
273     Node *L, *M, *R;
274     split(this->root, L, M, l - 1);
275     split(M, M, R, r - 1);
276     merge(root, L, R);
277     this->_size -= r - l + 1;
278     return M;
279 }
280
281 void _move(const int l, const int r, const int new_pos) {
282     Node *node = _erase(l, r);
283     _insert(new_pos, node);
284 }
285
286 #ifdef REVERSE
287 void _reverse(const int l, const int r) {
288     Node *L, *M, *R;
289     split(this->root, L, M, l - 1);
290     split(M, M, R, r - 1);
291
292     Node *node = M;
293     node->rev ^= true;
294
295     merge(L, L, M);
296     merge(root, L, R);
297 }
298 #endif

```

```

299 public:
300     Treap() {}
301
302     /// Constructor that initializes the treap based on an array.
303     ///
304     /// Time Complexity: O(n)
305     Treap(const vector<int> &arr) : _size(arr.size()) {
306         vector<int> r = gen_rand(arr.size());
307         this->root = build(0, (int)arr.size() - 1, arr, r);
308     }
309
310     int size() { return _size; }
311
312     /// Moves the subarray [l, r] to the position starting at new_pos.
313     /// new_pos represents the position BEFORE the subarray is deleted!!!
314     ///
315     /// Time Complexity: O(log n)
316     void move(const int l, const int r, int new_pos) {
317         assert(0 <= new_pos, assert(new_pos <= _size);
318         if(new_pos > l)
319             // after erase the index will be different if new_pos > l
320             new_pos -= r - l + 1;
321         _move(l, r, new_pos);
322     }
323
324     /// Moves the subarray [l, r] to the back of the array.
325     ///
326     /// Time Complexity: O(log n)
327     void move_back(const int l, const int r) {
328         assert(0 <= l), assert(l <= r), assert(r < _size);
329         move(l, r, _size);
330     }
331
332     /// Moves the subarray [l, r] to the front of the array.
333     ///
334     /// Time Complexity: O(log n)
335     void move_front(const int l, const int r) {
336         assert(0 <= l), assert(l <= r), assert(r < _size);
337         move(l, r, 0);
338     }
339
340     #ifdef REVERSE
341     /// Reverses the subarray [l, r].
342     ///
343     /// Time Complexity: O(log n)
344     void reverse(const int l, const int r) {
345         assert(0 <= l), assert(l <= r), assert(r < _size);
346         _reverse(l, r);
347     }
348     #endif
349
350     /// Erases the subarray [l, r].
351     ///
352     /// Time Complexity: O(log n)
353     void erase(const int l, const int r) {
354         assert(0 <= l), assert(l <= r), assert(r < _size);
355         _erase(l, r);
356     }
357
358     /// Inserts the value val at the position pos.
359     ///
360     /// Time Complexity: O(log n)
361     void insert(const int pos, const int val) {
362         assert(pos <= _size);
363     }

```

```

364     nodes.emplace_back(new Node(val));
365     _insert(pos, nodes.back());
366 }
367
368 /// Returns the index of the i-th added node.
369 ///
370 /// Time Complexity: O(log n)
371 int get_ith(const int idx) {
372     assert(0 <= idx), assert(idx < nodes.size());
373     return _get_ith(idx);
374 }
375
376 /// Sums the delta value to the position pos.
377 ///
378 /// Time Complexity: O(log n)
379 void update(const int pos, const int delta) {
380     assert(0 <= pos), assert(pos < _size);
381     _update(pos, pos, delta);
382 }
383
384 #ifdef LAZY
385 /// Sums the delta value to the subarray [l, r].
386 ///
387 /// Time Complexity: O(log n)
388 void update(const int l, const int r, const int delta) {
389     assert(0 <= l), assert(l <= r), assert(r < _size);
390     _update(l, r, delta);
391 }
392 #endif
393
394 /// Query at a single index.
395 ///
396 /// Time Complexity: O(log n)
397 int query(const int pos) {
398     assert(0 <= pos), assert(pos < _size);
399     return _query(pos, pos)->ans;
400 }
401
402 /// Range query from l to r.
403 ///
404 /// Time Complexity: O(log n)
405 int query(const int l, const int r) {
406     assert(0 <= l), assert(l <= r), assert(r < _size);
407     return _query(l, r)->ans;
408 }
409 };
410 // clang-format on

```

3.16. Treap Maximum Contiguous Segment

```

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

```

```

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

```

4. Dp

4.1. Binary Lifting

```

1 // clang-format off
2 // #define COST
3 class Binary_Lifting {
4 private:
5     const int NEUTRAL_VALUE = 0;
6     // Up to ~1e9
7     const int MAXIMUM_POW = 30;
8     vector<int> lg;
9     const int n;
10    vector<vector<int>>> nxt, cost;
11
12    int combine(const int a, const int b) { return a + b; }
13
14    void build_log_array() {
15        lg.resize(n + 1);
16        for (int i = 2; i <= n; i++)
17            lg[i] = lg[i / 2] + 1;
18    }
19
20    void allocate() {

```

```

21 // initializes a matrix [n][lg n] with -1
22 build_log_array();
23 nxt.resize(MAXIMUM_POW, vector<int>(n + 1, -1));
24 #ifdef COST
25 cost.resize(MAXIMUM_POW, vector<int>(n + 1, NEUTRAL_VALUE));
26 #endif
27 }
28
29 void build_nxt() {
30     for (int j = 1; j < nxt.size(); j++)
31         for (int i = 0; i < nxt.front().size(); i++)
32             if (nxt[j - 1][i] != -1) {
33                 nxt[j][i] = nxt[j - 1][nxt[j - 1][i]];
34                 #ifdef COST
35                     cost[j][i] = combine(cost[j - 1][i], cost[j - 1][nxt[j - 1][i]]);
36                 #endif
37             }
38 }
39
40 public:
41 Binary_Lifting(const vector<int> &nxt
42               , #ifdef COST
43               , const vector<int> &cost
44               , #endif
45               ) : n(nxt.size()) {
46     allocate();
47     this->nxt[0] = nxt;
48     #ifdef COST
49     this->cost[0] = cost;
50     assert(nxt.size() == cost.size());
51     #endif
52     build_nxt();
53 }
54
55 /// Advance k steps from x.
56 ///
57 /// Time Complexity: O(log(k))
58 int go_nxt(int x, int k) {
59     for (int i = 0; k > 0; i++, k >>= 1)
60         if (k & 1) {
61             x = nxt[i][x];
62             if (x == -1)
63                 return -1;
64         }
65     return x;
66 }
67
68 #ifdef COST
69 /// Compute the cost after k steps from x.
70 ///
71 /// Time Complexity: O(log(k))
72 int compute_cost(int x, int k) {
73     assert(k < (1 << MAXIMUM_POW));
74     int ans = 0;
75     for (int i = 0; k > 0; i++, k >>= 1)
76         if (k & 1) {
77             ans += cost[i][x];
78             x = nxt[i][x];
79             if (x == -1)
80                 return -1;
81         }
82     return ans;
83 }
84 #endif
85 };

```

```
86 // clang-format on
```

4.2. Catalan

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

4.3. Catalan 1 1 2 5 14 42 132

```

1 // The first few Catalan numbers for n = 0, 1, 2, 3, ...
2 // are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
3 // Recursive formula:
4 // cat(0) = 0
5 // cat(n+1) = sum(i from 0 to n) (cat(i)*cat(n-i))
6 //
7 // Using Binomial Coefficient
8 // We can also use the below formula to find nth catalan number in O(n) time.
9
10 // Returns value of Binomial Coefficient C(n, k)
11
12 // REQUIRES combinatorics.cpp
13 int catalan(int n) {
14     return comb.fat(2 * n) * comb.inv(comb.fat(n + 1)) % MOD *
15           comb.inv(comb.fat(n)) % MOD;
16 }

```

4.4. Cht Optimization

```

1 /// Copied from:
2 ///
3     https://github.com/kth-competitive-programming/kactl/content/data-structures/Line
4
5 // clang-format off
6 /// Uncomment the line below to get the minimum answer, otherwise it will
7     try to
8     get the maximum answer.
9 // #define MINIMUM
10 struct Line {
11     // f(x) = aX + b
12     mutable int a, b, p;
13     bool operator<(const Line &o) const { return a < o.a; }
14     bool operator<(int x) const { return p < x; }
15     // Use the methods below to get the real value of the attributes!!!
16     int get_a() {
17         #ifdef MINIMUM
18             return -a;
19         #else
20             return a;
21         #endif
22     }
23     int get_b() {
24         #ifdef MINIMUM
25             return -b;
26         #else
27             return b;
28         #endif
29     }

```

```

28 };
29
30 struct LineContainer : multiset<Line, less<>> {
31     // (for doubles, use inf = 1/.0, div(a,b) = a/b)
32     static const int inf = LLONG_MAX;
33     int div(int a, int b) { // floored division
34         return a / b - ((a ^ b) < 0 && a % b);
35     }
36     bool isect(iterator x, iterator y) {
37         if (y == end())
38             return x->p = inf, 0;
39         if (x->a == y->a)
40             x->p = x->b > y->b ? inf : -inf;
41         else
42             x->p = div(y->b - x->b, x->a - y->a);
43         return x->p >= y->p;
44     }
45     /// Inserts the line a * x + b.
46     ///
47     /// Time Complexity: O(log n)
48     void add(int a, int b) {
49         #ifdef MINIMUM
50             a = -a, b = -b;
51         #endif
52         auto z = insert({a, b, 0}), y = z++, x = y;
53         while (isect(y, z))
54             z = erase(z);
55         if (x != begin() && isect(--x, y))
56             isect(x, y = erase(y));
57         while ((y = x) != begin() && (--x)->p >= y->p)
58             isect(x, erase(y));
59     }
60     /// Query the best line such that a * x + b is maximum/minimum.
61     ///
62     /// Time Complexity: O(log n)
63     int query(int x) {
64         assert(!empty());
65         auto l = *lower_bound(x);
66         #ifdef MINIMUM
67             return -(l.a * x + l.b);
68         #else
69             return l.a * x + l.b;
70         #endif
71     }
72 };
73 // clang-format on

```

4.5. Digit Dp

```

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

```

```

14     return 0;
15
16     if (pos == num.size()) {
17         if (cnt == k)
18             return 1;
19         return 0;
20     }
21
22     if (DP[pos][cnt][f] != -1)
23         return DP[pos][cnt][f];
24     int res = 0;
25     int lim = (f ? 9 : num[pos]);
26
27     /// Try to place all the valid digits such that the number doesn't exceed b
28     for (int dgt = 0; dgt <= LMT; dgt++) {
29         int nf = f;
30         int ncnt = cnt;
31         if (f == 0 && dgt < LMT)
32             nf = 1; /// The number is getting smaller at this position
33         if (dgt == d)
34             ncnt++;
35         if (ncnt <= k)
36             res += call(pos + 1, ncnt, nf);
37     }
38
39     return DP[pos][cnt][f] = res;
40 }
41
42 int solve(int b) {
43     num.clear();
44     while (b > 0) {
45         num.push_back(b % 10);
46         b /= 10;
47     }
48     reverse(num.begin(), num.end());
49     /// Stored all the digits of b in num for simplicity
50
51     memset(DP, -1, sizeof(DP));
52     int res = call(0, 0, 0);
53     return res;
54 }
55
56 int main() {
57
58     cin >> a >> b >> d >> k;
59     int res = solve(b) - solve(a - 1);
60     cout << res << endl;
61
62     return 0;
63 }

```

4.6. Divide And Conquer Optimization

```

1  /// Problem: Split the array into k buckets such that the cost of each
2  /// bucket is
3  /// the square sum of each subarray.
4  ///
5  /// resize below
6  vector<int> last(MAXN, INF), dp(MAXN, INF);
7  /// Cost for the subarray (l, r).
8  int C(int l, int r) {
9     int val = 0;
10    val += sq(sum(l, r));
11    return val;

```

```

11 }
12
13 /// dp[i] represents the cost of splitting the array into k buckets (after k
14 /// iterations), with the index i as the last index.
15 ///
16 /// Time Complexity: O(n*k*(log n))
17 void f(int l, int r, int optl, int optr) {
18     if (l > r)
19         return;
20
21     int mid = (l + r) / 2;
22     auto best = make_pair(INF, INF); // change to (-INF, -INF) to maximize
23     // change mid - 1 to mid if buckets can intercept.
24     for (int i = optl; i <= min(mid, optr); ++i)
25         best = min(best, {(i == 1 ? 0 : last[i - 1]) + C(i, mid), i});
26
27     dp[mid] = best.first;
28     const int opt = best.second;
29
30     f(l, mid - 1, optl, opt);
31     f(mid + 1, r, opt, optr);
32 }
33
34 // 1-indexed, change to 0-index if necessary.
35 int compute(const int k) {
36     for (int i = 0; i < k; ++i) {
37         f(1, n, 1, n);
38         last.swap(dp);
39     }
40
41     return last[n];
42 }

```

4.7. Edit Distance

```

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

```

4.8. Knuth Optimization

```

1 /// Problem: Given an array of n numbers split the array until get n
2 /// of one element. The cost of each split is the sum of values in the
3 /// subarray.

```

```

3 ///
4 /// Time Complexity: O(n^2)
5 void knuth() {
6     // length of the cut
7     for (int i = 0; i < n; ++i)
8         // cutting from j to j + i
9         for (int j = 0; j + i < n; ++j) {
10             if (i == 0) {
11                 dp[j][i + j] = 0;
12                 idx[j][i + j] = j;
13             } else {
14                 dp[j][j + i] = INF;
15                 // searching for the optimal place to cut
16                 for (int k = idx[j][j + i - 1]; k <= min(j + i - 1, idx[j + 1][i +
17                     ++k) {
18                     int val = dp[j][k] + dp[k + 1][j + i] + sum(j, j + i);
19                     if (val < dp[j][j + i]) {
20                         dp[j][j + i] = val;
21                         idx[j][j + i] = k;
22                     }
23                 }
24             }
25         }
26 }

```

4.9. Lis

```

1 int lis(vector<int> &arr) {
2     int n = arr.size();
3     vector<int> lis;
4     for (int i = 0; i < n; i++) {
5         int l = 0, r = (int)lis.size() - 1;
6         int ans = -1;
7         while (l <= r) {
8             int mid = (l + r) / 2;
9             // OBS: - To >= LIS change to the operation below to >
10             // - Put <= or >= for strictly!!
11             if (arr[i] < lis[mid]) {
12                 r = mid - 1;
13                 ans = mid;
14             } else
15                 l = mid + 1;
16         }
17         if (ans == -1)
18             lis.emplace_back(arr[i]);
19         else
20             lis[ans] = arr[i];
21     }
22     return lis.size();
23 }
24

```

4.10. Longest Common Subsequence

```

1 string lcs(string &s, string &t) {
2     const int n = s.size(), m = t.size();
3     s.insert(s.begin(), '#'), t.insert(t.begin(), '$');
4     vector<vector<int>> mat(n + 1, vector<int>(m + 1, 0));
5     for (int i = 1; i <= n; i++) {
6         for (int j = 1; j <= m; j++) {
7             if (s[i] == t[j])
8                 mat[i][j] = mat[i - 1][j - 1] + 1;

```

```

9         else
10             mat[i][j] = max(mat[i - 1][j], mat[i][j - 1]);
11     }
12 }
13
14 string ans;
15 int i = n, j = m;
16 while (i > 0 && j > 0) {
17     if (s[i] == t[j])
18         ans += s[i], i--, j--;
19     else if (mat[i][j - 1] > mat[i - 1][j])
20         j--;
21     else
22         i--;
23 }
24
25 reverse(ans.begin(), ans.end());
26 return ans;
27 }

```

4.11. Longest Increasing Subsequence 2D (Not Sorted)

```

1 set<ii> s[(int)2e6];
2 bool check(ii par, int ind) {
3     auto it = s[ind].lower_bound(ii(par.ff, -INF));
4     if (it == s[ind].begin())
5         return false;
6     it--;
7     if (it->ss < par.ss)
8         return true;
9     return false;
10 }
11
12 int lis2d(vector<ii> &arr) {
13     int n = arr.size();
14     s[1].insert(arr[0]);
15
16     int maior = 1;
17     for (int i = 1; i < n; i++) {
18         ii x = arr[i];
19         int l = 1, r = maior;
20         int ansbb = 0;
21         while (l <= r) {
22             int mid = (l + r) / 2;
23             if (check(x, mid)) {
24                 l = mid + 1;
25                 ansbb = mid;
26             } else
27                 r = mid - 1;
28         }
29
30         // inserting in list
31         auto it = s[ansbb + 1].lower_bound(ii(x.ff, -INF));
32         while (it != s[ansbb + 1].end() && it->ss >= x.ss)
33             it = s[ansbb + 1].erase(it);
34
35         it = s[ansbb + 1].lower_bound(ii(x.ff, -INF));
36         if (s[ansbb + 1].size() > 0 && it != s[ansbb + 1].end() && it->ff ==
37             x.ff &&
38             it->ss <= x.ss)
39             continue;
39         s[ansbb + 1].insert(arr[i]);
40
41         maior = max(maior, ansbb + 1);
42     }
43 }

```

```

42     }
43
44     return maior;
45 }

```

4.12. Longest Increasing Subsequence 2D (Sorted)

```

1 set<pair<int, int>> s[(int)2e6];
2 bool check(pair<int, int> par, int ind) {
3     auto it = s[ind].lower_bound(pair<int, int>(par.ff, -INF));
4     if (it == s[ind].begin())
5         return false;
6     it--;
7     if (it->ss < par.ss)
8         return true;
9     return false;
10 }
11
12 int lis2d(vector<pair<int, int>> &arr) {
13     const int n = arr.size();
14     s[1].insert(arr[0]);
15
16     int maior = 1;
17     for (int i = 1; i < n; i++) {
18         pair<int, int> x = arr[i];
19         int l = 1, r = maior;
20         int ansbb = 0;
21         while (l <= r) {
22             int mid = (l + r) / 2;
23             if (check(x, mid)) {
24                 l = mid + 1;
25                 ansbb = mid;
26             } else {
27                 r = mid - 1;
28             }
29         }
30
31         // inserting in list
32         auto it = s[ansbb + 1].lower_bound(pair<int, int>(x.ff, -INF));
33         while (it != s[ansbb + 1].end() && it->ss >= x.ss)
34             it = s[ansbb + 1].erase(it);
35
36         it = s[ansbb + 1].lower_bound(pair<int, int>(x.ff, -INF));
37         if (s[ansbb + 1].size() > 0 && it != s[ansbb + 1].end() && it->ff ==
38             x.ff &&
39             it->ss <= x.ss)
40             continue;
40         s[ansbb + 1].insert(arr[i]);
41
42         maior = max(maior, ansbb + 1);
43     }
44
45     return maior;
46 }

```

4.13. Subset Sum (Bitset)

```

1 void subsetSum(const vector<int> &arr) {
2     bitset<312345> bit;
3     bit.set(0);
4     for (int i = 0; i < arr.size(); i++)
5         bit |= bit << arr[i];
6 }

```

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 vector<vector<edge>>
13         &adj,
14                                     const vector<int> &in_degree)
15 {
16     // source and destination
17     int src = -1, dest = -1;
18     // adj[i].size() represents the out degree of an vertex
19     for (int i = 0; i < adj.size(); i++) {
20         if ((int)adj[i].size() - in_degree[i] == 1) {
21             if (src != -1)
22                 return make_pair(false, pair<int, int>());
23             src = i;
24         } else if ((int)adj[i].size() - in_degree[i] == -1) {
25             if (dest != -1)
26                 return make_pair(false, pair<int, int>());
27             dest = i;
28         } else if (abs((int)adj[i].size() - in_degree[i]) > 1)
29             return make_pair(false, pair<int, int>());
30     }
31     if (src == -1 && dest == -1)
32         return make_pair(true, pair<int, int>(src, dest));
33     else if (src != -1 && dest != -1)
34         return make_pair(true, pair<int, int>(src, dest));
35     return make_pair(false, pair<int, int>());
36 }
37
38 /// Builds the path/tour for directed graphs.
39 void build(const int u, vector<int> &tour, vector<vector<edge>> &adj,
40         vector<bool> &used) {
41     while (!adj[u].empty()) {
42         const edge e = adj[u].back();
43         if (!used[e.id]) {
44             used[e.id] = true;
45             adj[u].pop_back();
46             build(e.v, tour, adj, used);
47         } else
48             adj[u].pop_back();
49     }
50     tour.push_back(u);
51 }
52
53 /// Auxiliary function to build the eulerian tour/path.
54 vector<int> set_build(vector<vector<edge>> &adj, const int E, const int
55     first) {
56     vector<int> path;
57     vector<bool> used(E + 3);
58

```

```

59     build(first, path, adj, used);
60
61     for (int i = 0; i < adj.size(); i++)
62         // if there are some remaining edges, it's not possible to build the
63         // tour.
64         if (adj[i].size())
65             return vector<int>();
66
67     reverse(path.begin(), path.end());
68     return path;
69 } // namespace detail
70
71 /// All vertices v should have in_degree[v] == out_degree[v]. It must not
72 /// contain a specific start and end vertices.
73 ///
74 /// Time complexity: O(V * (log V) + E)
75 bool has_euler_tour_directed(const vector<vector<edge>> &adj,
76     const vector<int> &in_degree) {
77     const pair<bool, pair<int, int>> aux =
78         detail::check_both_directed(adj, in_degree);
79     const bool valid = aux.first;
80     const int src = aux.second.first;
81     const int dest = aux.second.second;
82     return (valid && src == -1 && dest == -1);
83 }
84
85 /// A directed graph has an eulerian path/tour if has:
86 /// - One vertex v such that out_degree[v] - in_degree[v] == 1
87 /// - One vertex v such that in_degree[v] - out_degree[v] == 1
88 /// - The remaining vertices v such that in_degree[v] == out_degree[v]
89 /// or
90 /// - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR
91 ///
92 /// Returns a boolean value that indicates whether there's a path or not.
93 /// If there's a valid path it also returns two numbers: the source and the
94 /// destination. If the source and destination can be an arbitrary vertex it
95 /// will return the pair (-1, -1) for the source and destination (it means
96 /// the
97 /// contains an eulerian tour).
98 ///
99 /// Time complexity: O(V + E)
100 pair<bool, pair<int, int>>
101 has_euler_path_directed(const vector<vector<edge>> &adj,
102     const vector<int> &in_degree) {
103     return detail::check_both_directed(adj, in_degree);
104 }
105
106 /// Returns the euler path. If the graph doesn't have an euler path it
107 /// returns
108 /// an empty vector.
109 ///
110 /// Time Complexity: O(V + E) for directed, O(V * log(V) + E) for undirected.
111 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
112 vector<int> get_euler_path_directed(const int E, vector<vector<edge>> &adj,
113     const vector<int> &in_degree) {
114     const pair<bool, pair<int, int>> aux =
115         has_euler_path_directed(adj, in_degree);
116     const bool valid = aux.first;
117     const int src = aux.second.first;
118     const int dest = aux.second.second;
119
120     if (!valid)
121         return vector<int>();

```

```

121     int first;
122     if (src != -1)
123         first = src;
124     else {
125         first = 0;
126         while (adj[first].empty())
127             first++;
128     }
129
130     return detail::set_build(adj, E, first);
131 }
132
133 /// Returns the euler tour. If the graph doesn't have an euler tour it
134 /// returns
135 /// an empty vector.
136 ///
137 /// Time Complexity: O(V + E)
138 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
139 vector<int> get_euler_tour_directed(const int E, vector<vector<edge>> &adj,
140                                 const vector<int> &in_degree) {
141     const bool valid = has_euler_tour_directed(adj, in_degree);
142
143     if (!valid)
144         return vector<int>();
145
146     int first = 0;
147     while (adj[first].empty())
148         first++;
149
150     return detail::set_build(adj, E, first);
151 }
152
153 // The graph has a tour that passes to every edge exactly once and gets
154 // back to the first edge on the tour.
155 //
156 // A graph with an euler path has zero odd degree vertex.
157 //
158 // Time Complexity: O(V)
159 bool has_euler_tour_undirected(const vector<int> &degree) {
160     for (int i = 0; i < degree.size(); i++)
161         if (degree[i] & 1)
162             return false;
163     return true;
164 }
165
166 // The graph has a path that passes to every edge exactly once.
167 // It doesn't necessarily gets back to the beginning.
168 //
169 // A graph with an euler path has two or zero (tour) odd degree vertices.
170 //
171 // Returns a pair with the startpoint/endpoint of the path.
172 //
173 // Time Complexity: O(V)
174 pair<bool, pair<int, int>> has_euler_path_undirected(const vector<int> &degree) {
175     vector<int> odd_degree;
176     for (int i = 0; i < degree.size(); i++)
177         if (degree[i] & 1)
178             odd_degree.pb(i);
179
180     if (odd_degree.size() == 0)
181         return make_pair(true, make_pair(-1, -1));
182     else if (odd_degree.size() == 2)
183         return make_pair(true, make_pair(odd_degree.front(), odd_degree.back()));
184     else

```

```

185         return make_pair(false, pair<int, int>());
186     }
187
188     vector<int> get_euler_tour_undirected(const int E, const vector<int> &degree,
189                                         vector<vector<edge>> &adj) {
190         if (!has_euler_tour_undirected(degree))
191             return vector<int>();
192
193         int first = 0;
194         while (adj[first].empty())
195             first++;
196
197         return detail::set_build(adj, E, first);
198     }
199
200 /// Returns the euler tour. If the graph doesn't have an euler tour it
201 /// returns
202 /// an empty vector.
203 ///
204 /// Time Complexity: O(V + E)
205 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
206 vector<int> get_euler_path_undirected(const int E, const vector<int> &degree,
207                                     vector<vector<edge>> &adj) {
208     auto aux = has_euler_path_undirected(degree);
209     const bool valid = aux.first;
210     const int x = aux.second.first;
211     const int y = aux.second.second;
212
213     if (!valid)
214         return vector<int>();
215
216     int first;
217     if (x != -1) {
218         first = x;
219         adj[x].emplace_back(y, E + 1);
220         adj[y].emplace_back(x, E + 1);
221     } else {
222         first = 0;
223         while (adj[first].empty())
224             first++;
225     }
226
227     vector<int> ans = detail::set_build(adj, E, first);
228     reverse(ans.begin(), ans.end());
229     if (x != -1)
230         ans.pop_back();
231     return ans;
232 }; // namespace graph

```

5.2. Articulation Points

```

1 namespace graph {
2     unordered_set<int> ap;
3     vector<int> low, disc;
4     int cur_time = 1;
5
6     void dfs_ap(const int u, const int p, const vector<vector<int>> &adj) {
7         low[u] = disc[u] = cur_time++;
8         int children = 0;
9
10        for (const int v : adj[u]) {
11            // DO NOT ADD PARALLEL EDGES
12            if (disc[v] == 0) {

```



```

13     ++children;
14     dfs_ap(v, u, adj);
15
16     low[u] = min(low[v], low[u]);
17     if (p == -1 && children > 1)
18         ap.emplace(u);
19     if (p != -1 && low[v] >= disc[u])
20         ap.emplace(u);
21 } else if (v != p)
22     low[u] = min(low[u], disc[v]);
23 }
24 }
25
26 void init_ap(const int n) {
27     cur_time = 1;
28     ap = unordered_set<int>();
29     low = vector<int>(n, 0);
30     disc = vector<int>(n, 0);
31 }
32
33 /// THE GRAPH MUST BE UNDIRECTED!
34 ///
35 /// Returns the vertices in which their removal disconnects the graph.
36 ///
37 /// Time Complexity: O(V + E)
38 vector<int> articulation_points(const int indexed_from,
39                               const vector<vector<int>> &adj) {
40     init_ap(adj.size());
41     vector<int> ans;
42     for (int u = indexed_from; u < adj.size(); ++u) {
43         if (disc[u] == 0)
44             dfs_ap(u, -1, adj);
45         if (ap.count(u))
46             ans.emplace_back(u);
47     }
48     return ans;
49 }
50 }; // namespace graph

```

5.3. Bellman Ford

```

1 struct edge {
2     int src, dest, weight;
3     edge() {}
4     edge(int src, int dest, int weight) : src(src), dest(dest), weight(weight) {}
5
6     bool operator<(const edge &a) const { return weight < a.weight; }
7 };
8
9 /// Works to find the shortest path with negative edges.
10 /// Also detects cycles.
11 ///
12 /// Time Complexity: O(n * e)
13 /// Space Complexity: O(n)
14 bool bellman_ford(vector<edge> &edges, int src, int n) {
15     // n = qtd of vertices, E = qtd de arestas
16
17     // To calculate the shortest path uncomment the line below
18     // vector<int> dist(n, INF);
19
20     // To check cycles uncomment the line below
21     // vector<int> dist(n, 0);
22

```

```

23     vector<int> pai(n, -1);
24     int E = edges.size();
25
26     dist[src] = 0;
27     // Relax all edges n - 1 times.
28     // A simple shortest path from src to any other vertex can have at-most n
29     // - 1 edges.
30     for (int i = 1; i <= n - 1; i++) {
31         for (int j = 0; j < E; j++) {
32             int u = edges[j].src;
33             int v = edges[j].dest;
34             int weight = edges[j].weight;
35             if (dist[u] != INF && dist[u] + weight < dist[v]) {
36                 dist[v] = dist[u] + weight;
37                 pai[v] = u;
38             }
39         }
40     }
41
42     // Check for NEGATIVE-WEIGHT CYCLES.
43     // The above step guarantees shortest distances if graph doesn't contain
44     // negative weight cycle. If we get a shorter path, then there is a cycle.
45     bool is_cycle = false;
46     int vert_in_cycle;
47     for (int i = 0; i < E; i++) {
48         int u = edges[i].src;
49         int v = edges[i].dest;
50         int weight = edges[i].weight;
51         if (dist[u] != INF && dist[u] + weight < dist[v]) {
52             is_cycle = true;
53             pai[v] = u;
54             vert_in_cycle = v;
55         }
56     }
57
58     if (is_cycle) {
59         for (int i = 0; i < n; i++)
60             vert_in_cycle = pai[vert_in_cycle];
61
62         vector<int> cycle;
63         for (int v = vert_in_cycle; (v != vert_in_cycle || cycle.size() <= 1);
64              v = pai[v])
65             cycle.pb(v);
66
67         reverse(cycle.begin(), cycle.end());
68
69         for (int x : cycle) {
70             cout << x + 1 << ' ';
71         }
72         cout << cycle.front() + 1 << endl;
73         return true;
74     } else
75         return false;
76 }

```

5.4. Bipartite Check

```

1 /// Time Complexity: O(V + E)
2 bool is_bipartite(const int src, const vector<vector<int>> &adj) {
3     vector<int> color(adj.size(), -1);
4     queue<int> q;
5
6     color[src] = 1;

```

```

7   q.emplace(src);
8   while (!q.empty()) {
9       const int u = q.front();
10      q.pop();

11      for (const int v : adj[u]) {
12          if (color[v] == color[u])
13              return false;
14          else if (color[v] == -1) {
15              color[v] = !color[u];
16              q.emplace(v);
17          }
18      }
19  }
20  }
21  return true;
22  }

```

5.5. Block Cut Tree

```

1  // based on kokosha's implementation.
2  /// INDEXED FROM ZERO!!!!
3  class BCT {
4      vector<vector<pair<int, int>>> adj;
5      vector<pair<int, int>> edges;
6      /// Stores the edges in the i-th component.
7      vector<vector<int>> comps;
8      /// Stores the vertices in the i-th component.
9      vector<vector<int>> vert_in_comp;
10     int cur_time = 0;
11     vector<int> disc, conv;
12     vector<vector<int>> adj_bct;
13     const int n;

14     /// Finds the biconnected components.
15     int dfs(const int x, const int p, stack<int> &st) {
16         int low = disc[x] = ++cur_time;
17         for (const pair<int, int> &e : adj[x]) {
18             const int v = e.first, idx = e.second;
19             if (idx != p) {
20                 if (!disc[v]) { // if haven't passed
21                     st.emplace(idx); // disc[x] < low -> bridge
22                     const int low_at = dfs(v, idx, st);
23                     low = min(low, low_at);
24                     if (disc[x] <= low_at) {
25                         comps.emplace_back();
26                         vector<int> &tmp = comps.back();
27                         for (int y = -1; y != idx; st.pop())
28                             tmp.emplace_back(y = st.top());
29                     }
30                 } else if (disc[v] < disc[x]) // back_edge
31                     low = min(low, disc[v]), st.emplace(idx);
32             }
33         }
34         return low;
35     }
36 }

37 /// Splits the graph into biconnected components.
38 void split() {
39     adj_bct.resize(n + edges.size() + 1);
40     stack<int> st;
41     for (int i = 0; i < n; ++i)
42         if (!disc[i])
43             dfs(i, -1, st);
44 }

```

```

46 vector<bool> in(n);
47 for (const vector<int> &comp : comps) {
48     vert_in_comp.emplace_back();
49     for (const int e : comp) {
50         const int u = edges[e].first, v = edges[e].second;
51         if (!in[u])
52             in[u] = 1, vert_in_comp.back().emplace_back(u);
53         if (!in[v])
54             in[v] = 1, vert_in_comp.back().emplace_back(v);
55     }
56     for (const int e : comp)
57         in[edges[e].first] = in[edges[e].second] = 0;
58 }
59 }
60
61 /// Algorithm: It compresses the biconnected components into one vertex.
62 Then
63 /// it creates a bipartite graph with the original vertices on the left and
64 /// the bcc's on the right. After that, it connects with an edge the i-th
65 /// vertex on the left to the j-th on the right if the vertex i is present
66 in
67 the j-th bcc. Note that articulation points will be present in more
68 than
69 one component.
70 void build() {
71     // next new node to be used in bct
72     int nxt = n;
73     for (const vector<int> &vic : vert_in_comp) {
74         for (const int u : vic) {
75             adj_bct[u].emplace_back(nxt);
76             adj_bct[nxt].emplace_back(u);
77             conv[u] = nxt;
78         }
79         nxt++;
80     }
81
82     // if it's not an articulation point we can remove it from the bct.
83     for (int i = 0; i < n; ++i)
84         if (adj_bct[i].size() == 1)
85             adj_bct[i].clear();
86 }
87
88 void init() {
89     disc.resize(n);
90     conv.resize(n);
91     adj.resize(n);
92 }
93
94 public:
95     /// Pass the number of vertices to the constructor.
96     BCT(const int n) : n(n) { init(); }
97
98     /// Adds an bidirectional edge.
99     void add_edge(const int u, const int v) {
100         assert(0 <= min(u, v)), assert(max(u, v) < n), assert(u != v);
101         adj[u].emplace_back(v, edges.size());
102         adj[v].emplace_back(u, edges.size());
103         edges.emplace_back(u, v);
104     }
105
106     /// Returns the bct tree. It builds the tree if it's not computed.
107     ///
108     /// Time Complexity: O(n + m)
109     vector<vector<int>> tree() {
110         if (adj_bct.empty()) // if it's not calculated.

```

```

108     split(), build();
109     return adj_bct;
110 }
111
112 /// Returns whether the vertex u is an articulation point or not.
113 bool is_art_point(const int u) {
114     assert(0 <= u), assert(u < n);
115     assert(!adj_bct.empty()); // the tree method should've called before.
116     return !adj_bct[u].empty();
117 }
118
119 /// Returns the corresponding vertex of the u-th vertex in the bct.
120 int convert(const int u) {
121     assert(0 <= u), assert(u < n);
122     assert(!adj_bct.empty()); // the tree method should've called before.
123     return adj_bct[u].empty() ? conv[u] : u;
124 }
125 };

```

5.6. Bridges

```

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

```

```

44        for (int u = indexed_from; u < adj.size(); ++u)
45            if (disc[u] == 0)
46                dfs_bg(u, -1, adj);
47
48        return bg;
49    }
50 } // namespace graph

```

5.7. Centroid

```

1 /// Returns the centroids of the tree which can contains at most 2.
2 ///
3 /// Time complexity: O(n)
4 vector<int> centroid(const int n, const int indexed_from,
5                     const vector<vector<int>> &adj) {
6     vector<int> centers, sz(n + indexed_from);
7     function<void(int, int)> dfs = [&](const int u, const int p) {
8         sz[u] = 1;
9         bool is_centroid = true;
10        for (const int v : adj[u]) {
11            if (v == p)
12                continue;
13            dfs(v, u);
14            sz[u] += sz[v];
15            if (sz[v] > n / 2)
16                is_centroid = false;
17        }
18        if (n - sz[u] > n / 2)
19            is_centroid = false;
20        if (is_centroid)
21            centers.emplace_back(u);
22    };
23    dfs(indexed_from, -1);
24    return centers;
25 }

```

5.8. Centroid Decomposition

```

1 class Centroid {
2 private:
3     int it = 1, _vertex;
4     vector<int> vis, used, sub, _parent;
5     vector<vector<int>> _tree;
6
7     int dfs(const int u, int &cnt, const vector<vector<int>> &adj) {
8         vis[u] = it;
9         ++cnt;
10        sub[u] = 1;
11        for (const int v : adj[u])
12            if (vis[v] != it && !used[v])
13                sub[u] += dfs(v, cnt, adj);
14        return sub[u];
15    }
16
17    int find_centroid(const int u, const int cnt,
18                    const vector<vector<int>> &adj) {
19        vis[u] = it;
20
21        bool valid = true;
22        int max_sub = -1;
23        for (const int v : adj[u]) {
24            if (vis[v] == it || used[v])
25                continue;

```

```

26     if (sub[v] > cnt / 2)
27         valid = false;
28     if (max_sub == -1 || sub[v] > sub[max_sub])
29         max_sub = v;
30 }
31
32 if (valid && cnt - sub[u] <= cnt / 2)
33     return u;
34 return find_centroid(max_sub, cnt, adj);
35 }
36
37 int find_centroid(const int u, const vector<vector<int>> &adj) {
38     // counts the number of vertices
39     int cnt = 0;
40
41     // set up sizes and nodes in current subtree
42     dfs(u, cnt, adj);
43     ++it;
44
45     const int ctd = find_centroid(u, cnt, adj);
46     ++it;
47     used[ctd] = true;
48     return ctd;
49 }
50
51 int build_tree(const int u, const vector<vector<int>> &adj) {
52     const int ctd = find_centroid(u, adj);
53
54     for (const int v : adj[ctd]) {
55         if (used[v])
56             continue;
57         const int ctd_v = build_tree(v, adj);
58         _tree[ctd].emplace_back(ctd_v);
59         _tree[ctd_v].emplace_back(ctd);
60         _parent[ctd_v] = ctd;
61     }
62
63     return ctd;
64 }
65
66 void allocate(const int n) {
67     vis.resize(n);
68     _parent.resize(n, -1);
69     sub.resize(n);
70     used.resize(n);
71     _tree.resize(n);
72 }
73
74 public:
75     /// Constructor that creates the centroid tree.
76     ///
77     /// Time Complexity:  $O(n * \log(n))$ 
78     Centroid(const int root_idx, const vector<vector<int>> &adj) {
79         allocate(adj.size());
80         _vertex = build_tree(root_idx, adj);
81     }
82
83     /// Returns the centroid of the whole tree.
84     int vertex() { return _vertex; }
85
86     int parent(const int u) { return _parent[u]; }
87
88     vector<vector<int>> tree() { return _tree; }
89 };

```

5.9. Compress ScCs In Dag

```

1 DSU dsu(MAXN);
2 /// Compress SCC's in a directed graph.
3 ///
4 /// Time Complexity:  $O(V)$ 
5 vector<vector<int>> compress(const int indexed_from,
6                             const vector<vector<int>> &adj) {
7     const int n = adj.size();
8     SCC scc(n, indexed_from, adj);
9     vector<unordered_set<int>> g(n);
10
11     for (int i = 0; i < scc.number_of_comp; ++i)
12         for (int v : scc.scc[i])
13             dsu.Union(v, scc.scc[i].front());
14
15     for (int u = indexed_from; u < n; ++u)
16         for (int v : adj[u])
17             if (dsu.Find(u) != dsu.Find(v))
18                 g[dsu.Find(u)].emplace(dsu.Find(v));
19
20     vector<vector<int>> ret(n);
21     for (int u = indexed_from; u < n; ++u)
22         ret[u] = vector<int>(g[u].begin(), g[u].end());
23     return ret;
24 }

```

5.10. Count (3-4) Cycles

```

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

```

```

35     cnt[to2] = 0;
36 }
37 }
38 for (int to1 : g[u])
39     for (int to2 : adj[to1]) {
40         if (comp(u, to2)) {
41             ans += cnt[to2];
42             ++cnt[to2];
43         }
44         if (rep[to2] != -1)
45             cycles.push_back({u, to1, to2, rep[to2]});
46         rep[to2] = to1;
47     }
48 }
49
50 // Finding Triangles:
51 for (int to : adj[u])
52     vis[to] = 1;
53 for (int to1 : g[u])
54     for (int to2 : g[to1])
55         if (vis[to2])
56             cycles.push_back({u, to1, to2});
57 for (int to : adj[u])
58     vis[to] = 0;
59 }
60
61 return ans;
62 }

```

5.11. Cycle Detection

```

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

```

5.12. De Bruijn Sequence

```

1  // We can solve this problem by constructing a directed graph with
2  //  $k^{n-1}$  nodes with each node having  $k$  outgoing edges_order. Each node
3  // corresponds to a string of size  $n-1$ . Every edge corresponds to one of the
4  // characters in  $A$  and adds that character to the starting string. For
5  // example,
6  // if  $n=3$  and  $k=2$ , then we construct the following graph:
7  //
8  //      - 1 -> (01) - 1 ->
9  //      /      ^ |      \
10 // 0 -> (00)  1  0      (11) <- 1
11 //      \      | v      /
12 //      <- 0 - (10) <- 0 -
13
14 // The node '01' is connected to node '11' through edge '1', as adding '1' to
15 // '01' (and removing the first character) gives us '11'.
16 //
17 // We can observe that every node in this graph has equal in-degree and
18 // out-degree, which means that a Eulerian circuit exists in this graph.
19
20 namespace graph {
21     namespace detail {
22         // Finding an valid eulerian path
23         void dfs(const string &node, const string &alphabet, set<string> &vis,
24                 string &edges_order) {
25             for (char c : alphabet) {
26                 string nxt = node + c;
27                 if (vis.count(nxt))
28                     continue;
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     //
41     // Time Complexity:  $O(\text{alphabet.size()}^n \cdot \log_2(\text{alphabet.size()}^n))$ 
42     string de_bruijn(const int n, const string &alphabet) {
43         set<string> vis;
44         string edges_order;
45
46         string starting_node = string(n - 1, alphabet.front());
47         detail::dfs(starting_node, alphabet, vis, edges_order);
48
49         return edges_order + starting_node;
50     }; // namespace graph

```

5.13. Dijkstra + Dij Graph

```

1  // clang-format off
2  /// Works also with 1-indexed graphs.
3  // #define QUEUE
4  class Dijkstra {
5  private:
6      static constexpr int INF = 2e18;

```

```

7   bool CREATE_GRAPH = false;
8   int src;
9   int n;
10  vector<int> _dist;
11  vector<vector<int>> parent;
12
13 private:
14     /// Time Complexity: O(E log V)
15     void _compute(const int src, const vector<vector<pair<int, int>>> &adj) {
16         _dist.resize(this->n, INF);
17         vector<bool> vis(this->n, false);
18
19         if (CREATE_GRAPH) {
20             parent.resize(this->n);
21
22             for (int i = 0; i < this->n; i++)
23                 parent[i].emplace_back(i);
24         }
25
26         #ifdef QUEUE
27         queue<pair<int, int>> pq;
28         #else
29         priority_queue<pair<int, int>, vector<pair<int, int>>,
30             greater<pair<int, int>>>
31             pq;
32         #endif
33         pq.emplace(0, src);
34         _dist[src] = 0;
35
36         while (!pq.empty()) {
37             #ifdef QUEUE
38             int u = pq.front().second;
39             #else
40             int u = pq.top().second;
41             #endif
42             pq.pop();
43             if (vis[u])
44                 continue;
45             vis[u] = true;
46
47             for (const pair<int, int> &x : adj[u]) {
48                 int v = x.first, w = x.second;
49
50                 if (_dist[u] + w < _dist[v]) {
51                     _dist[v] = _dist[u] + w;
52                     pq.emplace(_dist[v], v);
53                     if (CREATE_GRAPH) {
54                         parent[v].clear();
55                         parent[v].emplace_back(u);
56                     }
57                 } else if (CREATE_GRAPH && _dist[u] + w == _dist[v]) {
58                     parent[v].emplace_back(u);
59                 }
60             }
61         }
62     }
63
64     vector<vector<int>> gen_dij_graph(const int dest) {
65         vector<vector<int>> dijkstra_graph(this->n);
66         vector<bool> vis(this->n, false);
67         queue<int> q;
68
69         q.emplace(dest);
70         while (!q.empty()) {
71             int v = q.front();

```

```

72         q.pop();
73
74         for (const int u : parent[v]) {
75             if (u == v)
76                 continue;
77             dijkstra_graph[u].emplace_back(v);
78             if (!vis[u]) {
79                 q.emplace(u);
80                 vis[u] = true;
81             }
82         }
83     }
84     return dijkstra_graph;
85 }
86
87 vector<int> gen_min_path(const int dest) {
88     vector<int> path, prev(this->n, -1), d(this->n, INF);
89     queue<int> q;
90
91     q.emplace(dest);
92     d[dest] = 0;
93
94     while (!q.empty()) {
95         int v = q.front();
96         q.pop();
97
98         for (const int u : parent[v]) {
99             if (u == v)
100                 continue;
101             if (d[v] + 1 < d[u]) {
102                 d[u] = d[v] + 1;
103                 prev[u] = v;
104                 q.emplace(u);
105             }
106         }
107     }
108
109     int cur = this->src;
110     while (cur != -1) {
111         path.emplace_back(cur);
112         cur = prev[cur];
113     }
114
115     return path;
116 }
117
118 public:
119     /// Allows creation of dijkstra graph and getting the minimum path.
120     Dijkstra(const int src, const bool create_graph,
121         const vector<vector<pair<int, int>>> &adj)
122         : n(adj.size()), src(src), CREATE_GRAPH(create_graph) {
123         this->_compute(src, adj);
124     }
125
126     /// Constructor that computes only the Dijkstra minimum path from src.
127     ///
128     /// Time Complexity: O(E log V)
129     Dijkstra(const int src, const vector<vector<pair<int, int>>> &adj)
130         : n(adj.size()), src(src) {
131         this->_compute(src, adj);
132     }
133
134     /// Returns the Dijkstra graph of the graph.
135     ///
136     /// Time Complexity: O(V)

```

```

137 vector<vector<int>> dijkstra(const int dest) {
138     assert(CREATE_GRAPH);
139     return gen_dijkstra(dest);
140 }
141
142 /// Returns the vertices present in a path from src to dest with
143 /// minimum cost and a minimum length.
144 ///
145 /// Time Complexity: O(V)
146 vector<int> min_path(const int dest) {
147     assert(CREATE_GRAPH);
148     return gen_min_path(dest);
149 }
150
151 /// Returns the distance from src to dest.
152 int dist(const int dest) {
153     assert(0 <= dest), assert(dest < n);
154     return _dist[dest];
155 }
156 };
157 // clang-format on

```

5.14. Dinic

```

1 class Dinic {
2     struct Edge {
3         const int v;
4         // capacity (maximum flow) of the edge
5         // if it is a reverse edge then its capacity should be equal to 0
6         const int cap;
7         // current flow of the edge
8         int flow = 0;
9         Edge(const int v, const int cap) : v(v), cap(cap) {}
10    };
11
12 private:
13     static constexpr int INF = (sizeof(int) == 4 ? 1e9 : 2e18) + 1e5;
14     bool COMPUTED = false;
15     int _max_flow;
16     vector<Edge> edges;
17     // holds the indexes of each edge present in each vertex.
18     vector<vector<int>> adj;
19     const int n;
20     // src will be always 0 and sink n+1.
21     const int src, sink;
22     vector<int> level, ptr;
23
24 private:
25     vector<vector<int>> _flow_table() {
26         vector<vector<int>> table(n, vector<int>(n, 0));
27         for (int u = 0; u <= sink; ++u)
28             for (const int idx : adj[u])
29                 // checks if it's not a reverse edge
30                 if (!(idx & 1))
31                     table[u][edges[idx].v] += edges[idx].flow;
32         return table;
33     }
34
35     /// Algorithm: Greedily all vertices from the matching will be added and,
36     /// after that, edges in which one of the vertices is not covered will
37     /// also be
38     /// added to the answer.
39     vector<pair<int, int>> _min_edge_cover() {
40         vector<bool> covered(n, false);

```

```

40     vector<pair<int, int>> ans;
41     for (int u = 1; u < sink; ++u) {
42         for (const int idx : adj[u]) {
43             const Edge &e = edges[idx];
44             // ignore if it is a reverse edge or an edge linked to the sink
45             if (idx & 1 || e.v == sink)
46                 continue;
47             if (e.flow == e.cap) {
48                 ans.emplace_back(u, e.v);
49                 covered[u] = covered[e.v] = true;
50                 break;
51             }
52         }
53     }
54
55     for (int u = 1; u < sink; ++u) {
56         for (const int idx : adj[u]) {
57             const Edge &e = edges[idx];
58             if (idx & 1 || e.v == sink)
59                 continue;
60             if (e.flow < e.cap && (!covered[u] || !covered[e.v])) {
61                 ans.emplace_back(u, e.v);
62                 covered[u] = covered[e.v] = true;
63             }
64         }
65     }
66     return ans;
67 }
68
69 /// Algorithm: Takes the complement of the vertex cover.
70 vector<int> _max_ind_set(const int max_left) {
71     const vector<int> mvc = _min_vertex_cover(max_left);
72     vector<bool> contains(n);
73     for (const int v : mvc)
74         contains[v] = true;
75     vector<int> ans;
76     for (int i = 1; i < sink; ++i)
77         if (!contains[i])
78             ans.emplace_back(i);
79     return ans;
80 }
81
82 void dfs_vc(const int u, vector<bool> &vis, const bool left,
83             const vector<vector<int>> &paths) {
84     vis[u] = true;
85     for (const int idx : adj[u]) {
86         const Edge &e = edges[idx];
87         if (vis[e.v])
88             continue;
89         // saturated edges goes from right to left
90         if (left && paths[u][e.v] == 0)
91             dfs_vc(e.v, vis, left ^ 1, paths);
92         // non-saturated edges goes from left to right
93         else if (!left && paths[e.v][u] == 1)
94             dfs_vc(e.v, vis, left ^ 1, paths);
95     }
96 }
97
98 /// Algorithm: The edges that belong to the Matching M will go from right
99 to
100 /// left, all other edges will go from left to right. A DFS will be run
101 /// starting at all left vertices that are not incident to edges in M. Some
102 /// vertices of the graph will become visited during this DFS and some
103 /// not-visited. To get minimum vertex cover all visited right
104 /// vertices of M will be taken, and all not-visited left vertices of M.

```

```

104 /// Source: codeforces.com/blog/entry/17534?comment=223759
105 vector<int> _min_vertex_cover(const int max_left) {
106     vector<bool> vis(n, false), saturated(n, false);
107     const auto paths = flow_table();
108
109     for (int i = 1; i <= max_left; ++i) {
110         for (int j = max_left + 1; j < sink; ++j)
111             if (paths[i][j] > 0) {
112                 saturated[j] = true;
113                 break;
114             }
115         if (!saturated[i] && !vis[i])
116             dfs_vc(i, vis, 1, paths);
117     }
118
119     vector<int> ans;
120     for (int i = 1; i <= max_left; ++i)
121         if (saturated[i] && !vis[i])
122             ans.emplace_back(i);
123
124     for (int i = max_left + 1; i < sink; ++i)
125         if (saturated[i] && vis[i])
126             ans.emplace_back(i);
127
128     return ans;
129 }
130
131 void dfs_build_path(const int u, vector<int> &path,
132                   vector<vector<int>> &table, vector<vector<int>> &ans,
133                   const vector<vector<int>> &adj) {
134     path.emplace_back(u);
135
136     if (u == sink) {
137         ans.emplace_back(path);
138         return;
139     }
140
141     for (const int v : adj[u]) {
142         if (table[u][v]) {
143             --table[u][v];
144             dfs_build_path(v, path, table, ans, adj);
145             return;
146         }
147     }
148 }
149
150 /// Algorithm: Run DFS's from the source and gets the paths when possible.
151 vector<vector<int>> _compute_all_paths(const vector<vector<int>> &adj) {
152     vector<vector<int>> table = flow_table();
153     vector<vector<int>> ans;
154     ans.reserve(_max_flow);
155
156     for (int i = 0; i < _max_flow; i++) {
157         vector<int> path;
158         path.reserve(n);
159         dfs_build_path(src, path, table, ans, adj);
160     }
161
162     return ans;
163 }
164
165 /// Algorithm: Find the set of vertices that are reachable from the source
166 in
167 /// the residual graph. All edges which are from a reachable vertex to
168 /// non-reachable vertex are minimum cut edges.

```

```

168 /// Source: geeksforgeeks.org/minimum-cut-in-a-directed-graph
169 pair<int, vector<pair<int, int>>> _min_cut() {
170     // checks if there's an edge from i to j.
171     vector<vector<int>> mat_adj(n, vector<int>(n, 0));
172     // checks if the residual capacity is greater than 0
173     vector<vector<bool>> residual(n, vector<bool>(n, 0));
174     for (int u = 0; u <= sink; ++u)
175         for (const int idx : adj[u])
176             // checks if it's not a reverse edge
177             if (!(idx & 1)) {
178                 mat_adj[u][edges[idx].v] = edges[idx].cap;
179                 // checks if its residual capacity is greater than zero.
180                 if (edges[idx].flow < edges[idx].cap)
181                     residual[u][edges[idx].v] = true;
182             }
183
184     vector<bool> vis(n);
185     queue<int> q;
186
187     q.emplace(src);
188     vis[src] = true;
189     while (!q.empty()) {
190         int u = q.front();
191         q.pop();
192         for (int v = 0; v < n; ++v)
193             if (residual[u][v] && !vis[v]) {
194                 q.emplace(v);
195                 vis[v] = true;
196             }
197     }
198
199     int weight = 0;
200     vector<pair<int, int>> cut;
201     for (int i = 0; i < n; ++i)
202         for (int j = 0; j < n; ++j)
203             if (vis[i] && !vis[j])
204                 // if there's an edge from i to j.
205                 if (mat_adj[i][j] > 0) {
206                     weight += mat_adj[i][j];
207                     cut.emplace_back(i, j);
208                 }
209
210     return make_pair(weight, cut);
211 }
212
213 void _add_edge(const int u, const int v, const int cap) {
214     adj[u].emplace_back(edges.size());
215     edges.emplace_back(v, cap);
216     // adding reverse edge
217     adj[v].emplace_back(edges.size());
218     edges.emplace_back(u, 0);
219 }
220
221 bool bfs_flow() {
222     queue<int> q;
223     memset(level.data(), -1, sizeof(*level.data()) * level.size());
224     q.emplace(src);
225     level[src] = 0;
226     while (!q.empty()) {
227         const int u = q.front();
228         q.pop();
229         for (const int idx : adj[u]) {
230             const Edge &e = edges[idx];
231             if (e.cap == e.flow || level[e.v] != -1)
232                 continue;

```



```

233     level[e.v] = level[u] + 1;
234     q.emplace(e.v);
235 }
236 }
237 return (level[sink] != -1);
238 }
239
240 int dfs_flow(const int u, const int cur_flow) {
241     if (u == sink)
242         return cur_flow;
243
244     for (int &idx = ptr[u]; idx < adj[u].size(); ++idx) {
245         Edge &e = edges[adj[u][idx]];
246         if (level[u] + 1 != level[e.v] || e.cap == e.flow)
247             continue;
248         const int flow = dfs_flow(e.v, min(e.cap - e.flow, cur_flow));
249         if (flow == 0)
250             continue;
251         e.flow += flow;
252         edges[adj[u][idx] ^ 1].flow -= flow;
253         return flow;
254     }
255     return 0;
256 }
257
258 int compute() {
259     int ans = 0;
260     while (bfs_flow()) {
261         memset(ptr.data(), 0, sizeof(*ptr.data()) * ptr.size());
262         while (const int cur = dfs_flow(src, INF))
263             ans += cur;
264     }
265     return ans;
266 }
267
268 void check_computed() {
269     if (!COMPUTED) {
270         COMPUTED = true;
271         this->_max_flow = compute();
272     }
273 }
274
275 public:
276     /// Constructor that makes assignments and allocations.
277     ///
278     /// Time Complexity: O(V)
279     Dinic(const int n) : n(n + 2), src(0), sink(n + 1) {
280         assert(n >= 0);
281
282         adj.resize(this->n);
283         level.resize(this->n);
284         ptr.resize(this->n);
285     }
286
287     /// Prints all the added edges. Use it to test in [CSA Graph
288     /// Editor] (https://csacademy.com/app/graph\_editor/).
289     void print() {
290         for (int u = 0; u < n; ++u)
291             for (const int idx : adj[u])
292                 if (!idx & 1)
293                     cerr << u << ' ' << edges[idx].v << ' ' << edges[idx].cap << endl;
294     }
295
296     /// Returns the edges from the minimum edge cover of the graph.
297     /// A minimum edge cover represents a set of edges such that each vertex

```

```

298     /// present in the graph is linked to at least one edge from this set.
299     ///
300     /// Time Complexity: O(V + E)
301     vector<pair<int, int>> min_edge_cover() {
302         this->check_computed();
303         return this->_min_edge_cover();
304     }
305
306     /// Returns the maximum independent set for the graph.
307     /// An independent set represents a set of vertices such that they're not
308     /// adjacent to each other.
309     /// It is equal to the complement of the minimum vertex cover.
310     ///
311     /// Time Complexity: O(V + E)
312     vector<int> max_ind_set(const int max_left) {
313         this->check_computed();
314         return this->_max_ind_set(max_left);
315     }
316
317     /// Returns the minimum vertex cover of a bipartite graph.
318     /// A minimum vertex cover represents a set of vertices such that each
319     /// edge of
320     /// the graph is incident to at least one vertex of the graph.
321     /// Pass the maximum index of a vertex on the left side as an argument.
322     ///
323     /// Time Complexity: O(V + E)
324     vector<int> min_vertex_cover(const int max_left) {
325         this->check_computed();
326         return this->_min_vertex_cover(max_left);
327     }
328
329     /// Computes all paths from src to sink.
330     /// Add all edges from the original graph. Its weights should be equal to
331     /// the
332     /// number of edges between the vertices. Pass the adjacency list with
333     /// repeated vertices if there are multiple edges.
334     ///
335     /// Time Complexity: O(max_flow * V + E)
336     vector<vector<int>> compute_all_paths(const vector<vector<int>> &adj) {
337         this->check_computed();
338         return this->_compute_all_paths(adj);
339     }
340
341     /// Returns the weight and the edges present in the minimum cut of the
342     /// graph.
343     /// A minimum cut represents a set of edges with minimum weight such that
344     /// after removing these edges, it disconnects the graph. If the graph is
345     /// undirected you can safely add edges in both directions. It doesn't work
346     /// with parallel edges, it's required to merge them.
347     ///
348     /// Time Complexity: O(V^2 + E)
349     pair<int, vector<pair<int, int>>> min_cut() {
350         this->check_computed();
351         return this->_min_cut();
352     }
353
354     /// Returns a table with the flow values for each pair of vertices.
355     ///
356     /// Time Complexity: O(V^2 + E)
357     vector<vector<int>> flow_table() {
358         this->check_computed();
359         return this->_flow_table();
360     }
361
362     /// Adds a directed edge between u and v and its reverse edge.

```

```

360 ///
361 /// Time Complexity: O(1);
362 void add_to_sink(const int u, const int cap) {
363     assert(!COMPUTED);
364     assert(src <= u), assert(u < sink);
365     this->_add_edge(u, sink, cap);
366 }
367
368 /// Adds a directed edge between u and v and its reverse edge.
369 ///
370 /// Time Complexity: O(1);
371 void add_to_src(const int v, const int cap) {
372     assert(!COMPUTED);
373     assert(src < v), assert(v <= sink);
374     this->_add_edge(src, v, cap);
375 }
376
377 /// Adds a directed edge between u and v and its reverse edge.
378 ///
379 /// Time Complexity: O(1);
380 void add_edge(const int u, const int v, const int cap) {
381     assert(!COMPUTED);
382     assert(src <= u), assert(u <= sink);
383     this->_add_edge(u, v, cap);
384 }
385
386 /// Computes the maximum flow for the network.
387 ///
388 /// Time Complexity: O(V^2*E) or O(E*sqrt(V)) for matching.
389 int max_flow() {
390     this->check_computed();
391     return this->_max_flow;
392 }
393 };

```

5.15. Dsu

```

1 // Remove comments to add rollback
2 class DSU {
3 public:
4     vector<int> root, sz;
5     // stack<tuple<int, int, int>> old_root, old_sz;
6
7     DSU(const int n) {
8         root.resize(n + 1);
9         iota(root.begin(), root.begin() + n + 1, 0ll);
10        sz.resize(n + 1, 1);
11    }
12
13    /// Returns the id of the set in which the element x belongs.
14    ///
15    /// Time Complexity: O(1)
16    int Find(const int x) {
17        if (root[x] == x)
18            return x;
19        return root[x] = Find(root[x]);
20        // DONT USE PATH COMPRESSION WITH ROLLBACK!!
21        // return Find(root[x]);
22    }
23
24    /// Unites two sets in which u and v belong.
25    /// Returns false if they already belong to the same set.
26    ///
27    /// Time Complexity: O(1)

```

```

28 bool Union(int u, int v /* , int idx */) {
29     u = Find(u), v = Find(v);
30     if (u == v)
31         return false;
32
33     if (sz[u] < sz[v])
34         swap(u, v);
35
36     // old_root.emplace(idx, v, root[v]);
37     // old_sz.emplace(idx, u, sz[u]);
38     root[v] = u;
39     sz[u] += sz[v];
40     return true;
41 }
42
43 void rollback() {
44     // int idx, u, val;
45     // tie(idx, u, val) = old_root.top();
46     // old_root.pop();
47     // root[u] = val;
48     // tie(idx, u, val) = old_sz.top();
49     // old_sz.pop();
50     // sz[u] = val;
51     // }
52 };

```

5.16. Dsu On Tree

```

1 /// Problem: What's the level of the subtree of u which contains the most
2   number
3 /// of nodes? In case of tie, choose the level with small number.
4 vector<int> sub_sz(const int root_idx, const vector<vector<int>> &adj) {
5     vector<int> sub(adj.size());
6     function<int(int, int)> dfs = [&](const int u, const int p) {
7         sub[u] = 1;
8         for (int v : adj[u])
9             if (v != p)
10                sub[u] += dfs(v, u);
11         return sub[u];
12     };
13     dfs(root_idx, -1);
14     return sub;
15 }
16
17 vector<int> sz;
18 int dep[MAXN];
19 vector<vector<int>> adj(MAXN);
20 int maxx, ans;
21 void add(int u, int p, int l, int big_child, int val) {
22     dep[l] += val;
23     if (dep[l] > maxx || (dep[l] == maxx && l < ans)) {
24         ans = l;
25         maxx = dep[l];
26     }
27     for (int v : adj[u]) {
28         if (v == p || big_child == v)
29             continue;
30         add(v, u, l + 1, big_child, val);
31     }
32 }
33
34 vector<int> q(MAXN);
35 void dfs(int u, int p, int l, bool keep) {

```

```

36 int idx = -1, val = -1;
37 for (int v : adj[u]) {
38     if (v == p)
39         continue;
40     if (sz[v] > val) {
41         val = sz[v];
42         idx = v;
43     }
44 }
45 // idx now contains the index of the node of the biggest subtree
46 for (int v : adj[u]) {
47     if (v == p || v == idx)
48         continue;
49     // precalculate the answer for small subtrees
50     dfs(v, u, l + 1, 0);
51 }
52
53 if (idx != -1) {
54     // precalculate the answer for the biggest subtree and keep the results
55     dfs(idx, u, l + 1, 1);
56 }
57
58 // Change below to apply the brute force you need. GENERALLY YOU SHOULD ONLY
59 // MODIFY BELOW.
60 // brute force all subtrees other than idx
61 add(u, p, l, idx, 1);
62
63 // the answer of u is the level ans. As it is relative to the input tree we
64 // need to subtract it to the current level of u
65 q[u] = ans - 1;
66 if (keep == 0) {
67     // removing the calculated answer for the subtree, if it doesn't belong
68     // to
69     // the biggest subtree of it's parent (keep = 0)
70     add(u, p, l, -1, -1);
71     // clearing the answer
72     maxx = 0, ans = 0;
73 }
74
75 /// MODIFY TO WORK WITH DISCONNECTED GRAPHS!!!
76 ///
77 /// Time Complexity: O(n log n)
78 void precalculate() {
79     sz = sub_sz(1, adj);
80     dfs(1, -1, 0, 0);
81 }

```

5.17. 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 }

```

5.18. Functional Graph

```

1 // Based on:
2 // http://maratona.ic.unicamp.br/MaratonaVerao2020/lecture-b/20200122.pdf
3
4 class Functional_Graph {
5     // FOR DIRECTED GRAPH
6 private:
7     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis) {
8         int id_cycle = cycle_cnt++;
9         int cur_id = 0;
10        this->first[id_cycle] = u;
11
12        while (!vis[u]) {
13            vis[u] = true;
14
15            this->cycle[id_cycle].push_back(u);
16
17            this->in_cycle[u] = true;
18            this->cycle_id[u] = id_cycle;
19            this->id_in_cycle[u] = cur_id;
20            this->near_in_cycle[u] = u;
21            this->id_near_cycle[u] = id_cycle;
22            this->cycle_dist[u] = 0;
23
24            u = nxt[u];
25            cur_id++;
26        }
27    }
28
29    // Time Complexity: O(V)
30    void build(int n, int indexed_from, vector<int> &nxt,
31              vector<int> &in_degree) {
32        queue<int> q;
33        vector<bool> vis(n + indexed_from);
34        for (int i = indexed_from; i < n + indexed_from; i++) {
35            if (in_degree[i] == 0) {
36                q.push(i);
37                vis[i] = true;
38            }
39        }
40
41        vector<int> process_order;
42        process_order.reserve(n + indexed_from);
43        while (!q.empty()) {
44            int u = q.front();
45            q.pop();
46
47            process_order.push_back(u);
48
49            if (--in_degree[nxt[u]] == 0) {
50                q.push(nxt[u]);
51                vis[nxt[u]] = true;
52            }
53        }
54
55        int cycle_cnt = 0;
56        for (int i = indexed_from; i < n + indexed_from; i++)
57            if (!vis[i])
58                compute_cycle(i, nxt, vis);
59
60        for (int i = (int)process_order.size() - 1; i >= 0; i--) {
61            int u = process_order[i];
62
63            this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
64            this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];

```

```

65     this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
66 }
67 }
68
69 void allocate(int n, int indexed_from) {
70     this->cycle.resize(n + indexed_from);
71     this->first.resize(n + indexed_from);
72
73     this->in_cycle.resize(n + indexed_from, false);
74     this->cycle_id.resize(n + indexed_from, -1);
75     this->id_in_cycle.resize(n + indexed_from, -1);
76     this->near_in_cycle.resize(n + indexed_from);
77     this->id_near_cycle.resize(n + indexed_from);
78     this->cycle_dist.resize(n + indexed_from);
79 }
80
81 public:
82     Functional_Graph(int n, int indexed_from, vector<int> &nxt,
83                     vector<int> &in_degree) {
84         this->allocate(n, indexed_from);
85         this->build(n, indexed_from, nxt, in_degree);
86     }
87
88     // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
89
90     // number of cycles
91     int cycle_cnt = 0;
92     // Vertices present in the i-th cycle.
93     vector<vector<int>> cycle;
94     // first vertex of the i-th cycle
95     vector<int> first;
96
97     // The i-th vertex is present in any cycle?
98     vector<bool> in_cycle;
99     // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
100    // cycle.
101    vector<int> cycle_id;
102    // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
    belong
103    // to any cycle.
104    vector<int> id_in_cycle;
105    // Represents the id of the nearest vertex present in a cycle.
106    vector<int> near_in_cycle;
107    // Represents the id of the nearest cycle.
108    vector<int> id_near_cycle;
109    // Distance to the nearest cycle.
110    vector<int> cycle_dist;
111    // Represent the id of the component of the vertex.
112    // Equal to id_near_cycle
113    vector<int> &comp = id_near_cycle;
114 };
115
116 class Functional_Graph {
117     // FOR UNDIRECTED GRAPH
118 private:
119     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis,
120                      vector<vector<int>> &adj) {
121         int id_cycle = cycle_cnt++;
122         int cur_id = 0;
123         this->first[id_cycle] = u;
124
125         while (!vis[u]) {
126             vis[u] = true;
127
128             this->cycle[id_cycle].push_back(u);

```

```

129     nxt[u] = find_nxt(u, vis, adj);
130     if (nxt[u] == -1)
131         nxt[u] = this->first[id_cycle];
132
133     this->in_cycle[u] = true;
134     this->cycle_id[u] = id_cycle;
135     this->id_in_cycle[u] = cur_id;
136     this->near_in_cycle[u] = u;
137     this->id_near_cycle[u] = id_cycle;
138     this->cycle_dist[u] = 0;
139
140     u = nxt[u];
141     cur_id++;
142 }
143 }
144
145 int find_nxt(int u, vector<bool> &vis, vector<vector<int>> &adj) {
146     for (int v : adj[u])
147         if (!vis[v])
148             return v;
149     return -1;
150 }
151
152 // Time Complexity: O(V + E)
153 void build(int n, int indexed_from, vector<int> &degree,
154           vector<vector<int>> &adj) {
155     queue<int> q;
156     vector<bool> vis(n + indexed_from, false);
157     vector<int> nxt(n + indexed_from);
158     for (int i = indexed_from; i < n + indexed_from; i++) {
159         if (adj[i].size() == 1) {
160             q.push(i);
161             vis[i] = true;
162         }
163     }
164
165     vector<int> process_order;
166     process_order.reserve(n + indexed_from);
167     while (!q.empty()) {
168         int u = q.front();
169         q.pop();
170
171         process_order.push_back(u);
172
173         nxt[u] = find_nxt(u, vis, adj);
174         if (--degree[nxt[u]] == 1) {
175             q.push(nxt[u]);
176             vis[nxt[u]] = true;
177         }
178     }
179
180     int cycle_cnt = 0;
181     for (int i = indexed_from; i < n + indexed_from; i++)
182         if (!vis[i])
183             compute_cycle(i, nxt, vis, adj);
184
185     for (int i = (int)process_order.size() - 1; i >= 0; i--) {
186         int u = process_order[i];
187
188         this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
189         this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
190         this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
191     }
192 }
193

```

```

194 void allocate(int n, int indexed_from) {
195     this->cycle.resize(n + indexed_from);
196     this->first.resize(n + indexed_from);
197
198     this->in_cycle.resize(n + indexed_from, false);
199     this->cycle_id.resize(n + indexed_from, -1);
200     this->id_in_cycle.resize(n + indexed_from, -1);
201     this->near_in_cycle.resize(n + indexed_from);
202     this->id_near_cycle.resize(n + indexed_from);
203     this->cycle_dist.resize(n + indexed_from);
204 }
205
206 public:
207     Functional_Graph(int n, int indexed_from, vector<int> degree,
208                     vector<vector<int>> &adj) {
209         this->allocate(n, indexed_from);
210         this->build(n, indexed_from, degree, adj);
211     }
212
213     // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
214
215     // number of cycles
216     int cycle_cnt = 0;
217     // Vertices present in the i-th cycle.
218     vector<vector<int>> cycle;
219     // first vertex of the i-th cycle
220     vector<int> first;
221
222     // The i-th vertex is present in any cycle?
223     vector<bool> in_cycle;
224     // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
225     // cycle.
226     vector<int> cycle_id;
227     // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
228     // belong
229     // to any cycle.
230     vector<int> id_in_cycle;
231     // Represents the id of the nearest vertex present in a cycle.
232     vector<int> near_in_cycle;
233     // Represents the id of the nearest cycle.
234     vector<int> id_near_cycle;
235     // Distance to the nearest cycle.
236     vector<int> cycle_dist;
237     // Represent the id of the component of the vertex.
238     // Equal to id_near_cycle
239     vector<int> &comp = id_near_cycle;
240 };

```

5.19. Girth (Shortest Cycle In A Graph)

```

1 int bfs(const int src) {
2     vector<int> dist(MAXN, INF);
3     queue<pair<int, int>> q;
4
5     q.emplace(src, -1);
6     dist[src] = 0;
7
8     int ans = INF;
9     while (!q.empty()) {
10         pair<int, int> aux = q.front();
11         const int u = aux.first, p = aux.second;
12         q.pop();
13
14         for (const int v : adj[u]) {

```

```

15             if (v == p)
16                 continue;
17             if (dist[v] < INF)
18                 ans = min(ans, dist[u] + dist[v] + 1);
19             else {
20                 dist[v] = dist[u] + 1;
21                 q.emplace(v, u);
22             }
23         }
24     }
25
26     return ans;
27 }
28
29 /// Returns the shortest cycle in the graph
30 ///
31 /// Time Complexity: O(V^2)
32 int get_girth(const int n) {
33     int ans = INF;
34     for (int u = 1; u <= n; u++)
35         ans = min(ans, bfs(u));
36     return ans;
37 }

```

5.20. Hld

```

1 class HLD {
2 private:
3     int n;
4     // number of nodes below the i-th node
5     vector<int> sz;
6
7 private:
8     void allocate() {
9         // this->id_in_tree.resize(this->n + 1, -1);
10        this->chain_head.resize(this->n + 1, -1);
11        this->chain_id.resize(this->n + 1, -1);
12        this->sz.resize(this->n + 1);
13        this->parent.resize(this->n + 1, -1);
14        // this->id_in_chain.resize(this->n + 1, -1);
15        // this->chain_size.resize(this->n + 1);
16    }
17
18    int get_sz(const int u, const int p, const vector<vector<int>> &adj) {
19        this->sz[u] = 1;
20        for (const int v : adj[u]) {
21            if (v == p)
22                continue;
23            this->sz[u] += this->get_sz(v, u, adj);
24        }
25        return this->sz[u];
26    }
27
28    void dfs(const int u, const int id, const int p,
29            const vector<vector<int>> &adj, int &nidx) {
30        // this->id_in_tree[u] = nidx++;
31        this->chain_id[u] = id;
32        // this->id_in_chain[u] = chain_size[id]++;
33        this->parent[u] = p;
34
35        if (this->chain_head[id] == -1)
36            this->chain_head[id] = u;
37
38        int maxx = -1, idx = -1;

```

```

39     for (const int v : adj[u]) {
40         if (v == p)
41             continue;
42         if (sz[v] > maxx) {
43             maxx = sz[v];
44             idx = v;
45         }
46     }
47
48     if (idx != -1)
49         this->dfs(idx, id, u, adj, nidx);
50
51     for (const int v : adj[u]) {
52         if (v == idx || v == p)
53             continue;
54         this->dfs(v, this->number_of_chains++, u, adj, nidx);
55     }
56 }
57
58 void build(const int root_idx, const vector<vector<int>> &adj) {
59     this->get_sz(root_idx, -1, adj);
60     int nidx = 0;
61     this->dfs(root_idx, 0, -1, adj, nidx);
62 }
63
64 // int _compute(const int u, const int limit, Seg_Tree &st) {
65 //     int ans = 0, v;
66 //     for (v = u; chain_id[v] != chain_id[limit];
67 //         v = parent[chain_head[chain_id[v]]]) {
68 //         // change below
69 //         ans = max(ans, st.query(id_in_tree[chain_head[chain_id[v]]],
70 //                                 id_in_tree[v]));
71 //     }
72 //     ans = max(ans, st.query(id_in_tree[limit], id_in_tree[v]));
73 //     return ans;
74 // }
75
76 public:
77     /// Builds the chains.
78     ///
79     /// Time Complexity: O(n)
80     HLD(const int root_idx, const vector<vector<int>> &adj) : n(adj.size()) {
81         allocate();
82         build(root_idx, adj);
83     }
84
85     /// Computes the paths until a limit using segment tree.
86     /// Uncomment id_in_tree!!!
87     ///
88     /// Time Complexity: O(log^2(n))
89     // int compute(const int u, const int limit, Seg_Tree &st) {
90     //     return _compute(u, limit, st);
91     // }
92
93     // TAKE CARE, YOU MAY GET MLE!!!
94     // the chains are indexed from 0
95     int number_of_chains = 1;
96     // topmost node of the chain
97     vector<int> chain_head;
98     // id of the node based on the order of the dfs (indexed by 0)
99     // vector<int> id_in_tree;
100    // id of the i-th node in his chain
101    // vector<int> id_in_chain;
102    // id of the chain that the i-th node belongs
103    vector<int> chain_id;

```

```

104    // size of the i-th chain
105    // vector<int> chain_size;
106    // parent of the i-th node, -1 for root
107    vector<int> parent;
108 };

```

5.21. Hungarian

```

1  /// Returns a vector p of size n, where p[i] is the match for i
2  /// and the minimum cost.
3  ///
4  /// Code copied from:
5  ///
6  ///     github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/Graph/Hungarian.cpp
7  /// Time Complexity: O(n^2 * m)
8  pair<vector<int>, int> solve(const vector<vector<int>> &matrix) {
9      const int n = matrix.size();
10     if (n == 0)
11         return {vector<int>(), 0};
12     const int m = matrix[0].size();
13     assert(n <= m);
14     vector<int> u(n + 1, 0), v(m + 1, 0), p(m + 1, 0), way, minv;
15     for (int i = 1; i <= n; i++) {
16         vector<int> minv(m + 1, INF);
17         vector<int> way(m + 1, 0);
18         vector<bool> used(m + 1, 0);
19         p[0] = i;
20         int k0 = 0;
21         do {
22             used[k0] = 1;
23             int i0 = p[k0], delta = INF, k1;
24             for (int j = 1; j <= m; j++) {
25                 if (!used[j]) {
26                     const int cur = matrix[i0 - 1][j - 1] - u[i0] - v[j];
27                     if (cur < minv[j]) {
28                         minv[j] = cur;
29                         way[j] = k0;
30                     }
31                     if (minv[j] < delta) {
32                         delta = minv[j];
33                         k1 = j;
34                     }
35                 }
36             }
37             for (int j = 0; j <= m; j++) {
38                 if (used[j]) {
39                     u[p[j]] += delta;
40                     v[j] -= delta;
41                 } else {
42                     minv[j] -= delta;
43                 }
44             }
45             k0 = k1;
46         } while (p[k0]);
47         do {
48             const int k1 = way[k0];
49             p[k0] = p[k1];
50             k0 = k1;
51         } while (k0);
52     }
53     vector<int> ans(n, -1);
54     for (int j = 1; j <= m; j++) {
55         if (!p[j])

```

```

56     continue;
57     ans[p[j] - 1] = j - 1;
58 }
59 return {ans, -v[0]};
60 }

```

5.22. Lca

```

1 // #define DIST
2 // #define COST
3 /// UNCOMMENT ALSO THE LINE BELOW FOR COST!
4
5 // clang-format off
6 class LCA {
7 private:
8     int n;
9     // INDEXED from 0 or 1??
10    int indexed_from;
11    /// Store all log2 from 1 to n
12    vector<int> lg;
13    // level of the i-th node (height)
14    vector<int> level;
15    // matrix to store the ancestors of each node in power of 2 levels
16    vector<vector<int>> anc;
17    #ifdef DIST
18    vector<int> dist;
19    #endif
20    #ifdef COST
21    // int NEUTRAL_VALUE = -INF; // MAX COST
22    // int combine(const int a, const int b) {return max(a, b);}
23
24    // int NEUTRAL_VALUE = INF; // MIN COST
25    // int combine(const int a, const int b) {return min(a, b);}
26    vector<vector<int>> cost;
27    #endif
28
29 private:
30    void allocate() {
31        // initializes a matrix [n][lg n] with -1
32        this->build_log_array();
33        this->anc.resize(n + 1, vector<int>(lg[n] + 1, -1));
34        this->level.resize(n + 1, -1);
35        #ifdef DIST
36        this->dist.resize(n + 1, 0);
37        #endif
38        #ifdef COST
39        this->cost.resize(n + 1, vector<int>(lg[n] + 1, NEUTRAL_VALUE));
40        #endif
41    }
42
43    void build_log_array() {
44        this->lg.resize(this->n + 1);
45        for (int i = 2; i <= this->n; i++)
46            this->lg[i] = this->lg[i / 2] + 1;
47    }
48
49    void build_anc() {
50        for (int j = 1; j < anc.front().size(); j++)
51            for (int i = 0; i < anc.size(); i++)
52                if (this->anc[i][j - 1] != -1) {
53                    this->anc[i][j] = this->anc[this->anc[i][j - 1]][j - 1];
54                    #ifdef COST
55                    this->cost[i][j] =

```

```

56        combine(this->cost[i][j - 1], this->cost[anc[i][j - 1]][j -
57        1]);
58        #endif
59    }
60
61    void build_weighted(const vector<vector<pair<int, int>>> &adj) {
62        this->dfs_LCA_weighted(this->indexed_from, -1, 1, 0, adj);
63        this->build_anc();
64    }
65
66    void dfs_LCA_weighted(const int u, const int p, const int l, const int d,
67        const vector<vector<pair<int, int>>> &adj) {
68        this->level[u] = l;
69        this->anc[u][0] = p;
70        #ifdef DIST
71        this->dist[u] = d;
72        #endif
73
74        for (const pair<int, int> &x : adj[u]) {
75            int v = x.first, w = x.second;
76            if (v == p)
77                continue;
78            #ifdef COST
79            this->cost[v][0] = w;
80            #endif
81            this->dfs_LCA_weighted(v, u, l + 1, d + w, adj);
82        }
83    }
84
85    void build_unweighted(const vector<vector<int>> &adj) {
86        this->dfs_LCA_unweighted(this->indexed_from, -1, 1, 0, adj);
87        this->build_anc();
88    }
89
90    void dfs_LCA_unweighted(const int u, const int p, const int l, const int d,
91        const vector<vector<int>> &adj) {
92        this->level[u] = l;
93        this->anc[u][0] = p;
94        #ifdef DIST
95        this->dist[u] = d;
96        #endif
97
98        for (const int v : adj[u]) {
99            if (v == p)
100                continue;
101            this->dfs_LCA_unweighted(v, u, l + 1, d + 1, adj);
102        }
103    }
104
105    // go up k levels from x
106    int lca_go_up(int x, int k) {
107        for (int i = 0; k > 0; i++, k >>= 1)
108            if (k & 1) {
109                x = this->anc[x][i];
110                if (x == -1)
111                    return -1;
112            }
113        return x;
114    }
115
116    #ifdef COST
117    /// Query between the an ancestor of v (p) and v. It returns the
118    /// max/min edge between them.
119    int lca_query_cost_in_line(int v, int p) {

```

```

120     assert(this->level[v] >= this->level[p]);
121
122     int k = this->level[v] - this->level[p];
123     int ans = NEUTRAL_VALUE;
124
125     for (int i = 0; k > 0; i++, k >>= 1)
126         if (k & 1) {
127             ans = combine(ans, this->cost[v][i]);
128             v = this->anc[v][i];
129         }
130
131     return ans;
132 }
133 #endif
134
135 int get_lca(int a, int b) {
136     // a is below b
137     if (this->level[b] > this->level[a])
138         swap(a, b);
139
140     const int logg = lg[this->level[a]];
141     // putting a and b in the same level
142     for (int i = logg; i >= 0; i--)
143         if (this->level[a] - (1 << i) >= this->level[b])
144             a = this->anc[a][i];
145
146     if (a == b)
147         return a;
148
149     for (int i = logg; i >= 0; i--)
150         if (this->anc[a][i] != -1 && this->anc[a][i] != this->anc[b][i]) {
151             a = this->anc[a][i];
152             b = this->anc[b][i];
153         }
154
155     return anc[a][0];
156 }
157
158 public:
159     /// Builds an weighted graph.
160     ///
161     /// Time Complexity: O(n*log(n))
162     explicit LCA(const vector<vector<pair<int, int>>> &adj,
163                 const int indexed_from)
164         : n(adj.size()), indexed_from(indexed_from) {
165         this->allocate();
166         this->build_weighted(adj);
167     }
168
169     /// Builds an unweighted graph.
170     ///
171     /// Time Complexity: O(n*log(n))
172     explicit LCA(const vector<vector<int>>> &adj, const int indexed_from)
173         : n(adj.size()), indexed_from(indexed_from) {
174         this->allocate();
175         this->build_unweighted(adj);
176     }
177
178     /// Goes up k levels from v. If it passes the root, returns -1.
179     ///
180     /// Time Complexity: O(log(k))
181     int go_up(const int v, const int k) {
182         assert(indexed_from <= v), assert(v < this->n + indexed_from);
183         return this->lca_go_up(v, k);
184     }

```

```

185
186     /// Returns the parent of v in the LCA dfs from 1.
187     ///
188     /// Time Complexity: O(1)
189     int parent(int v) {
190         assert(indexed_from <= v), assert(v < this->n + indexed_from);
191         return this->anc[v][0];
192     }
193
194     /// Returns the LCA of a and b.
195     ///
196     /// Time Complexity: O(log(n))
197     int query_lca(const int a, const int b) {
198         assert(indexed_from <= min(a, b)),
199             assert(max(a, b) < this->n + indexed_from);
200         return this->get_lca(a, b);
201     }
202
203     #ifndef DIST
204     /// Returns the distance from a to b. When the graph is unweighted, it is
205     /// considered 1 as the weight of the edges.
206     ///
207     /// Time Complexity: O(log(n))
208     int query_dist(const int a, const int b) {
209         assert(indexed_from <= min(a, b)),
210             assert(max(a, b) < this->n + indexed_from);
211         return this->dist[a] + this->dist[b] - 2 * this->dist[this->get_lca(a,
212             b)];
213     }
214     #endif
215
216     #ifndef COST
217     /// Returns the max/min weight edge from a to b.
218     ///
219     /// Time Complexity: O(log(n))
220     int query_cost(const int a, const int b) {
221         assert(indexed_from <= min(a, b)),
222             assert(max(a, b) < this->n + indexed_from);
223         const int l = this->query_lca(a, b);
224         return combine(this->lca_query_cost_in_line(a, l),
225             this->lca_query_cost_in_line(b, l));
226     }
227     #endif
228 };
229 // clang-format on

```

5.23. Longest Path In Dag

```

1  /// Requires topological_sort.cpp
2
3  /// Returns a vector with the maximal distance from src (must be 0 or 1) to
4  /// every node or a maximal path from src to (n - 1).
5  ///
6  /// Time Complexity: O(n)
7  vector<int> longest_path_in_dag(const int src, const vector<vector<int>>
8      &adj) {
9      const int n = adj.size();
10     vector<int> dp(n, -1), prev(n, -1);
11     dp[src] = 0;
12     for (int u : topological_sort(src, adj))
13         for (int v : adj[u])
14             if (dp[u] != -1 && dp[u] + 1 > dp[v]) {
15                 dp[v] = dp[u] + 1;
16                 prev[v] = u;
17             }
18 }

```



```

16     }
17
18     // Returns the longest path to each node
19     // return dp;
20
21     vector<int> path;
22     // Assuming that the last node is the node (n - 1)
23     int cur = n - 1;
24     while (cur != -1) {
25         path.emplace_back(cur);
26         cur = prev[cur];
27     }
28     reverse(path.begin(), path.end());
29     // Returns the maximal path from src to (n - 1)
30     return path;
31 }

```

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

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

5.25. Maximum Path Unweighted Graph

```

1 // Returns the maximum path between the vertices 0 and n - 1 in a unweighted
2 // graph.
3 //
4 // Time Complexity: O(V + E)
5 int maximum_path(int n) {
6     vector<int> top_order = topological_sort(n);
7     vector<int> pai(n, -1);
8     if (top_order.empty())
9         return -1;
10
11     vector<int> dp(n);
12     dp[0] = 1;
13     for (int u : top_order)
14         for (int v : adj[u])
15             if (dp[u] && dp[u] + 1 > dp[v]) {
16                 dp[v] = dp[u] + 1;
17                 pai[v] = u;
18             }
19
20     if (dp[n - 1] == 0)
21         return -1;
22
23     vector<int> path;
24     int cur = n - 1;
25     while (cur != -1) {
26         path.pb(cur);
27         cur = pai[cur];
28     }
29     reverse(path.begin(), path.end());
30
31     // cout << path.size() << endl;
32     // for(int x: path) {
33     //     cout << x + 1 << ' ';
34     // }
35     // cout << endl;
36
37     return dp[n - 1];
38 }

```

5.26. Min Cost Flow Gpresso

```

1 /// MINIMIZES COST * FLOW!!!!
2 /// Code copied from:
3 ///
4 https://github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/Graph/MinC
5 template <class T = int> class MCMF {
6 public:
7     struct Edge {
8         Edge(int a, T b, T c) : to(a), cap(b), cost(c) {}
9         int to;
10         T cap, cost;
11     };
12
13     MCMF(int size) {
14         n = size;
15         edges.resize(n);
16         pot.assign(n, 0);
17         dist.resize(n);
18         visit.assign(n, false);
19     }
20
21     std::pair<T, T> mcmf(int src, int sink) {
22         std::pair<T, T> ans(0, 0);
23         if (!SPFA(src, sink))
24             return ans;
25         fixPot();
26         // can use dijkstra to speed up depending on the graph
27         while (SPFA(src, sink)) {
28             auto flow = augment(src, sink);
29             ans.first += flow.first;
30             ans.second += flow.first * flow.second;
31             fixPot();
32         }
33         return ans;
34     }
35
36     void addEdge(int from, int to, T cap, T cost) {
37         edges[from].push_back(list.size());
38         list.push_back(Edge(to, cap, cost));
39         edges[to].push_back(list.size());
40         list.push_back(Edge(from, 0, -cost));
41     }
42
43 private:
44     int n;
45     std::vector<std::vector<int>> edges;
46     std::vector<Edge> list;
47     std::vector<int> from;
48     std::vector<T> dist, pot;
49     std::vector<bool> visit;
50
51     /*bool dij(int src, int sink) {
52         T INF = std::numeric_limits<T>::max();
53         dist.assign(n, INF);
54         from.assign(n, -1);
55         visit.assign(n, false);
56         dist[src] = 0;
57         for(int i = 0; i < n; i++) {
58             int best = -1;
59             for(int j = 0; j < n; j++) {
60                 if(visit[j]) continue;
61                 if(best == -1 || dist[best] > dist[j]) best = j;
62             }
63             if(dist[best] >= INF) break;
64         }
65     }
66 */

```

```

63     visit[best] = true;
64     for(auto e : edges[best]) {
65         auto ed = list[e];
66         if(ed.cap == 0) continue;
67         T toDist = dist[best] + ed.cost + pot[best] - pot[ed.to];
68         assert(toDist >= dist[best]);
69         if(toDist < dist[ed.to]) {
70             dist[ed.to] = toDist;
71             from[ed.to] = e;
72         }
73     }
74 }
75 return dist[sink] < INF;
76 }*/
77
78 std::pair<T, T> augment(int src, int sink) {
79     std::pair<T, T> flow = {list[from[sink]].cap, 0};
80     for (int v = sink; v != src; v = list[from[v] ^ 1].to) {
81         flow.first = std::min(flow.first, list[from[v]].cap);
82         flow.second += list[from[v]].cost;
83     }
84     for (int v = sink; v != src; v = list[from[v] ^ 1].to) {
85         list[from[v]].cap -= flow.first;
86         list[from[v] ^ 1].cap += flow.first;
87     }
88     return flow;
89 }
90
91 std::queue<int> q;
92 bool SPFA(int src, int sink) {
93     T INF = std::numeric_limits<T>::max();
94     dist.assign(n, INF);
95     from.assign(n, -1);
96     q.push(src);
97     dist[src] = 0;
98     while (!q.empty()) {
99         int on = q.front();
100        q.pop();
101        visit[on] = false;
102        for (auto e : edges[on]) {
103            auto ed = list[e];
104            if (ed.cap == 0)
105                continue;
106            T toDist = dist[on] + ed.cost + pot[on] - pot[ed.to];
107            if (toDist < dist[ed.to]) {
108                dist[ed.to] = toDist;
109                from[ed.to] = e;
110                if (!visit[ed.to]) {
111                    visit[ed.to] = true;
112                    q.push(ed.to);
113                }
114            }
115        }
116    }
117    return dist[sink] < INF;
118 }
119
120 void fixPot() {
121     T INF = std::numeric_limits<T>::max();
122     for (int i = 0; i < n; i++)
123         if (dist[i] < INF)
124             pot[i] += dist[i];
125 }
126 };

```

5.27. Min Cost Flow Kactl

```

1  /// MINIMIZES COST * FLOW!!!!
2  /// DOESN'T SUPPORT PARALLEL EDGES!!!!!!
3
4  /// Code copied from:
5  ///
6  ///     github.com/kth-competitive-programming/kactl/blob/master/content/graph/MinCostMaxFlow.cpp
7  #include <bits/extc++.h> /// include-line, keep-include
8
9  // #define all(x) begin(x), end(x)
10 // typedef pair<int, int> ii;
11 // typedef vector<int> vi;
12 typedef long long ll;
13 typedef vector<ll> VL;
14 #define sz(x) (int)(x).size()
15 #define rep(i, a, b) for (int i = a; i < (b); ++i)
16
17 const ll INF1 = numeric_limits<ll>::max() / 4;
18
19 // clang-format off
20 struct MCMF {
21     int N;
22     vector<vi> ed, red;
23     vector<VL> cap, flow, cost;
24     vi seen;
25     VL dist, pi;
26     vector<ii> par;
27
28     MCMF(int N) :
29         N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(cap),
30         seen(N), dist(N), pi(N), par(N) {}
31
32     void addEdge(int from, int to, ll cap, ll cost) {
33         this->cap[from][to] = cap;
34         this->cost[from][to] = cost;
35         ed[from].push_back(to);
36         red[to].push_back(from);
37     }
38
39     void path(int s) {
40         fill(all(seen), 0);
41         fill(all(dist), INF1);
42         dist[s] = 0; ll di;
43
44         __gnu_pbds::priority_queue<pair<ll, int>> q;
45         vector<decltype(q)::point_iterator> its(N);
46         q.push({0, s});
47
48         auto relax = [&](int i, ll cap, ll cost, int dir) {
49             ll val = di - pi[i] + cost;
50             if (cap && val < dist[i]) {
51                 dist[i] = val;
52                 par[i] = {s, dir};
53                 if (its[i] == q.end()) its[i] = q.push({-dist[i], i});
54                 else q.modify(its[i], {-dist[i], i});
55             }
56         };
57
58         while (!q.empty()) {
59             s = q.top().second; q.pop();
60             seen[s] = 1; di = dist[s] + pi[s];
61             for (int i : ed[s]) if (!seen[i])
62                 relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
63             for (int i : red[s]) if (!seen[i])

```

```

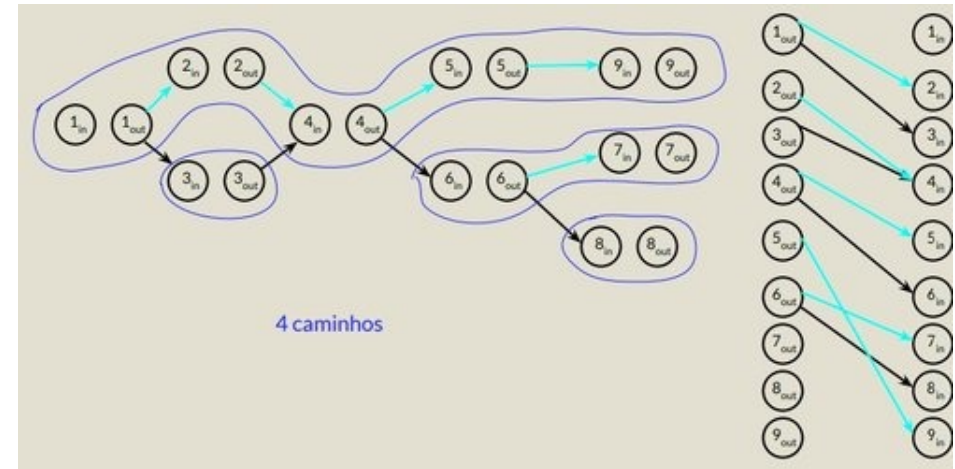
63     relax(i, flow[i][s], -cost[i][s], 0);
64 }
65 rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF1);
66 }
67
68 pair<ll, ll> maxflow(int s, int t) {
69     ll totflow = 0, totcost = 0;
70     while (path(s), seen[t]) {
71         ll fl = INF1;
72         for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
73             fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
74         totflow += fl;
75         for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
76             if (r) flow[p][x] += fl;
77             else flow[x][p] -= fl;
78     }
79     rep(i,0,N) rep(j,0,N) totcost += cost[i][j] * flow[i][j];
80     return {totflow, totcost};
81 }
82
83 // If some costs can be negative, call this before maxflow:
84 void setpi(int s) { // (otherwise, leave this out)
85     fill(all(pi), INF1); pi[s] = 0;
86     int it = N, ch = 1; ll v;
87     while (ch-- && it--)
88         rep(i,0,N) if (pi[i] != INF1)
89             for (int to : ed[i]) if (cap[i][to])
90                 if ((v = pi[i] + cost[i][to]) < pi[to])
91                     pi[to] = v, ch = 1;
92     assert(it >= 0); // negative cost cycle
93 }
94 };
95 // clang-format on

```

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

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

5.29. Minimum Path Cover In Dag



5.30. Minimum Path Cover In Dag

- 1 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.
- 2 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.

5.31. Mst

```

1  /// Requires DSU.cpp
2  struct edge {
3      int u, v, w;
4      edge() {}
5      edge(int u, int v, int w) : u(u), v(v), w(w) {}
6
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 }

```

5.32. Number Of Different Spanning Trees In A Complete Graph

```

1 Cayley's formula
2
3  $n^{n-2}$ 

```

5.33. Number Of Ways To Make A Graph Connected

```

1  $s_1 * s_2 * s_3 * \dots * s_k * (n^{k-2})$ 
2 n = number of vertices
3  $s_i$  = size of the i-th connected component
4 k = number of connected components

```

5.34. Pruffer Decode

```

1 // IT MUST BE INDEXED BY 0.
2 /// Returns the adjacency matrix of the decoded tree.
3 ///
4 /// Time Complexity: O(V)
5 vector<vector<int>> pruefer_decode(const vector<int> &code) {
6
7     int n = code.size() + 2;
8     vector<vector<int>> adj = vector<vector<int>>(n, vector<int>());
9     vector<int> degree(n, 1);
10    for (int x : code)
11        degree[x]++;
12
13    int ptr = 0;
14    while (degree[ptr] > 1)
15        ++ptr;
16
17    int nxt = ptr;
18    for (int u : code) {
19        adj[u].push_back(nxt);
20        adj[nxt].push_back(u);
21
22        if (--degree[u] == 1 && u < ptr)
23            nxt = u;
24        else {
25            while (degree[++ptr] > 1)
26                ;
27            nxt = ptr;
28        }
29    }
30    adj[n - 1].push_back(nxt);
31    adj[nxt].push_back(n - 1);
32
33    return adj;
34 }

```

5.35. 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 }

```

```

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

```

5.36. 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**.

5.37. 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.

5.38. 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     for (int v : adj[u])
29         if (!comp[v])
30             dfs_fill_order(v, s, adj);
31     s.push(u);
32 }
33
34 // The main function that finds all SCCs
35 void compute_SCC(vector<vector<int>> &adj) {
36     stack<int> s;
37     // Fill vertices in stack according to their finishing times
38     for (int i = indexed_from; i < this->n + indexed_from; i++)
39         if (!comp[i])
40             dfs_fill_order(i, s, adj);
41
42     // Create a reversed graph
43     get_transpose(adj);
44
45     fill(comp.begin(), comp.end(), -1);
46
47     // Now process all vertices in order defined by stack
48     while (s.empty() == false) {
49         int v = s.top();
50         s.pop();
51
52         if (comp[v] == -1)
53             dfs_trans(v, this->number_of_comp++);
54     }
55 }
56
57 public:
58 // number of the component of the i-th vertex
59 // it's always indexed from 0
60 vector<int> comp;
61 // the i-th vector contains the vertices that belong to the i-th scc
62 // it's always indexed from 0
63 vector<vector<int>> scc;
64 int number_of_comp = 0;
65
66 SCC(int n, int indexed_from, vector<vector<int>> &adj) {
67     this->n = n;
68     this->indexed_from = indexed_from;
69     comp.resize(n + 1);

```

```

70     trans.resize(n + 1);
71     scc.resize(n + 1);
72
73     this->compute_SCC(adj);
74 }
75 };

```

5.39. Small To Large (Merge Maps)

```

1 // Problem: How many nodes in each depth in the subtree of u?
2
3 void combine(map<int, int> &a, map<int, int> &b) {
4     if (a.size() < b.size())
5         swap(a, b);
6     for (auto [k, val] : b)
7         a[k] += val;
8 }
9
10 vector<vector<int>> adj(MAXN);
11
12 map<int, int> dfs(int u, int p, int l) {
13     map<int, int> mp;
14     for (int v : adj[u]) {
15         if (v != p) {
16             auto tmp = dfs(v, u, l + 1);
17             combine(mp, tmp);
18         }
19     }
20
21     mp[l]++;
22     // the map mp contains the answer here
23     return mp;
24 }

```

5.40. Topological Sort

```

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

```

```

28 }
29
30 if (cnt != n - indexed_from) {
31     // There exists a cycle in the graph
32     return vector<int>();
33 }
34
35 return top_order;
36 }

```

5.41. Tree Diameter

```

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

```

5.42. 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 }

```

```

15 }
16 }
17
18 /// Contains the maximum distance to the node i
19 vector<int> ans(MAXN);
20
21 void dfs(int u, int d, int p) {
22     ans[u] = max(d, sub[u].first);
23     for (const pair<int, int> x : adj[u]) {
24         int v = x.first, w = x.second;
25         if (v == p)
26             continue;
27         if (sub[v].first + w == ans[u]) {
28             dfs(v, max(d, sub[u].second) + w, u);
29         } else {
30             dfs(v, ans[u] + w, u);
31         }
32     }
33 }
34
35 /// Returns the maximum tree distance
36 int solve() {
37     subu(0, -1);
38     dfs(0, 0, -1);
39     return *max_element(ans.begin(), ans.end());
40 }

```

5.43. Tree Isomorphism

```

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

```

```

35 bool same(const int n, const int indexed_from, const vector<vector<int>>
    &adj1,
36           const vector<vector<int>> &adj2) {
37     vector<int> c1 = centroid(n, indexed_from, adj1),
38     c2 = centroid(n, indexed_from, adj2);
39     for (const int v : c2)
40       if (same(c1.front(), adj1, v, adj2))
41         return true;
42     return false;
43 }
44 } // namespace tree

```

6. Language Stuff

6.1. Binary String To Int

```

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

```

6.2. Check Char Type

```

1 #include <cctype>
2 isdigit(str[i]); // check if it's a number
3 isalpha(str[i]); // check if it's a letter
4 islower(str[i]); // check if it's lowercase
5 isupper(str[i]); // check if it's uppercase
6 isalnum(str[i]); // check if it's a number or letter
7 tolower(str[i]); // converts to lowercase
8 toupper(str[i]); // converts to uppercase

```

6.3. Check Overflow

```

1 bool __builtin_add_overflow (type1 a, type2 b, type3 *res)
2 bool __builtin_sadd_overflow (int a, int b, int *res)
3 bool __builtin_saddl_overflow (long int a, long int b, long int *res)
4 bool __builtin_saddll_overflow (long long int a, long long int b, long long
    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)

```

6.4. Counting Bits

```

1 #pragma GCC target ("sse4.2")
2 // Use the pragma above to optimize the time complexity to O(1)
3 __builtin_popcount(int) -> Number of active bits
4 __builtin_popcountll(int) -> Number of active bits
5 __builtin_ctz(int) -> Number of trailing zeros in binary representation
6 __builtin_clz(int) -> Number of leading zeros in binary representation
7 __builtin_parity(int) -> Parity of the number of bits

```

6.5. Gen Random Numbers (Rng)

```

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

```

6.6. Int To Binary String

```

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

```

6.7. Int To String

```

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

```

6.8. Permutation

```

1 int v[] = {1, 2, 3};
2 // The array must be sorted.
3 sort(v, v + 3);
4 do {
5     cout << v[0] << ' ' << v[1] << ' ' << v[2];
6 } while (next_permutation(v, v + 3));

```

6.9. Print Int128 T

```

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

```

```

17 reverse(all(ans));
18 cout << ans << endl;
19 }

```

6.10. Read And Write From File

```

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

```

6.11. Readint

```

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

```

6.12. Rotate Left

```

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

```

6.13. Rotate Right

```

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

```

6.14. Scanf From String

```

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

```

6.15. Split Function

```

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

```

6.16. String To Long Long

```

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

```

6.17. Substring

```

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

```

6.18. Time Measure

```

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

```

6.19. Unique Vector

```

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

```

6.20. Width

```

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

```

7. Math

7.1. Bell Numbers

```

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

```



```

15 }
16 return bell[n][0];
17 }

```

7.2. Binary Exponentiation

```

1 int bin_pow(const int n, int p) {
2     assert(p >= 0);
3     int ans = 1;
4     int cur_pow = n;
5
6     while (p) {
7         if (p & 1)
8             ans = (ans * cur_pow) % MOD;
9
10        cur_pow = (cur_pow * cur_pow) % MOD;
11        p >>= 1;
12    }
13
14    return ans;
15 }

```

7.3. Chinese Remainder Theorem

```

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

```

```

39 // use __int128_t in this case
40 ans =
41     mod(ans + x * (rem[i] - ans) / g % (md[i] / g) * LCM, LCM / g *
42         md[i]);
43     // lcm of LCM, md[i]
44     LCM = LCM / g * md[i];
45 }
46 return ans;
47 }

```

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 & 1)
22                ans = (ans * cur_pow) % MOD;
23
24            cur_pow = (cur_pow * cur_pow) % MOD;
25            p >>= 1;
26        }
27
28        return ans;
29    }
30
31    vector<int> build_inverse(const int max_val) {
32        vector<int> inv(max_val + 1);
33        inv[1] = 1;
34        for (int i = 2; i <= max_val; ++i)
35            inv[i] = mod(-MOD / i * inv[MOD % i]);
36        return inv;
37    }
38
39    vector<int> build_fat(const int max_val) {
40        vector<int> fat(max_val + 1);
41        fat[0] = 1;
42        for (int i = 1; i <= max_val; ++i)
43            fat[i] = mod(i * fat[i - 1]);
44        return fat;
45    }
46
47 public:
48     /// Builds both factorial and modular inverse array.
49     ///
50     /// Time Complexity: O(max_val)
51     Combinatorics(const int max_val) : max_val(max_val) {
52         assert(0 <= max_val), assert(max_val <= MOD);

```

```

53     this->_inv = this->build_inverse(max_val);
54     this->_fat = this->build_fat(max_val);
55 }
56
57 /// Returns the modular inverse of n % MOD.
58 ///
59 /// Time Complexity: O(log(MOD))
60 static int inv_log(const int n) { return bin_pow(n, MOD - 2); }
61
62 /// Returns the modular inverse of n % MOD.
63 ///
64 /// Time Complexity: O((n <= max_val ? 1 : log(MOD)))
65 int inv(const int n) {
66     assert(0 <= n);
67     if (n <= max_val)
68         return this->_inv[n];
69     else
70         return inv_log(n);
71 }
72
73 /// Returns the factorial of n % MOD.
74 int fat(const int n) {
75     assert(0 <= n), assert(n <= max_val);
76     return this->_fat[n];
77 }
78
79 /// Returns C(n, k) % MOD.
80 ///
81 /// Time Complexity: O(1)
82 int choose(const int n, const int k) {
83     assert(0 <= k), assert(k <= n), assert(n <= this->max_val);
84     return mod(fat(n) * mod(inv(fat(k)) * inv(fat(n - k))));
85 }
86 };

```

7.5. Diophantine Equation

```

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

```

7.6. Divide Fraction

```

1  /// Prints precision floating point places of a / b.
2  string divide(int a, int b, const int precision) {
3      assert(a < b);
4      string ans;
5      for (int i = 0; i < precision; ++i) {
6          a *= 10;
7          ans += a / b + '0';
8          a %= b;
9      }
10     return ans;
11 }

```

7.7. Divisors

```

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

```

7.8. Euler Totient

```

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

```

7.9. Extended Euclidean

```

1  // Created by tysm.
2
3  /// Returns a tuple containing the gcd(a, b) and the roots for
4  ///  $a \cdot x + b \cdot y = \text{gcd}(a, b)$ .
5  ///
6  /// Time Complexity: O(log(min(a, b))).
7  tuple<int, int, int> extended_gcd(int a, int b) {

```

```

8   int x = 0, y = 1, x1 = 1, y1 = 0;
9   while (a != 0) {
10      const int q = b / a;
11      tie(x, x1) = make_pair(x1, x - q * x1);
12      tie(y, y1) = make_pair(y1, y - q * y1);
13      tie(a, b) = make_pair(b % a, a);
14   }
15   return make_tuple(b, x, y);
16 }

```

7.10. Factorization

```

1  /// Factorizes a number.
2  ///
3  /// Time Complexity: O(sqrt(n))
4  map<int, int> factorize(int n) {
5      map<int, int> fat;
6      while (n % 2 == 0) {
7          ++fat[2];
8          n /= 2;
9      }
10
11     for (int i = 3; i * i <= n; i += 2) {
12         while (n % i == 0) {
13             ++fat[i];
14             n /= i;
15         }
16         /* OBS1
17            IF(N < 1E7)
18                you can optimize by factoring with SPF
19
20         */
21     }
22     if (n > 2)
23         ++fat[n];
24     return fat;
25 }

```

7.11. Fft

```

1  /// Code copied from:
2  ///
3  /// https://github.com/kth-competitive-programming/kactl/blob/08eb36f4bd9b8ce358e2f3fa8ffc329f62ebe811/content/numerical/FastFourierTransform.h
4
5  #define double long double
6  typedef complex<double> C;
7  typedef vector<double> vd;
8  void fft(vector<C> &a) {
9      int n = a.size(), L = 31 - __builtin_clz(n);
10     static vector<complex<double>> R(2, 1);
11     // uncomment if you'll use only 'double'.
12     // static vector<complex<long double>> R(2, 1);
13     static vector<C> rt(2, 1); // (^ 10% faster if double)
14     for (static int k = 2; k < n; k *= 2) {
15         R.resize(n);
16         rt.resize(n);
17         auto x = polar(1.0L, acos(-1.0L) / k);
18         for (int i = k; i < 2 * k; ++i)
19             rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
20     }
21     vi rev(n);
22     for (int i = 0; i < n; ++i)
23         rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
24     for (int i = 0; i < n; ++i)

```

```

24     if (i < rev[i])
25         swap(a[i], a[rev[i]]);
26     for (int k = 1; k < n; k *= 2)
27         for (int i = 0; i < n; i += 2 * k)
28             for (int j = 0; j < k; ++j) {
29                 auto x = (double *) &rt[j + k],
30                     y = (double *) &a[i + j + k]; /// exclude-line
31                 C z(x[0] * y[0] - x[1] * y[1],
32                   x[0] * y[1] + x[1] * y[0]); /// exclude-line
33                 a[i + j + k] = a[i + j] - z;
34                 a[i + j] += z;
35             }
36     }
37
38     /// Polynomial convolution of 'a' and 'b'.
39     ///
40     /// Time Complexity: O(n log n)
41     vector<long long> convolve(const vd &a, const vd &b) {
42         if (a.empty() || b.empty())
43             return {};
44         vd res(a.size() + b.size() - 1);
45         int L = 32 - __builtin_clz(res.size()), n = 1 << L;
46         vector<C> in(n), out(n);
47         copy(all(a), begin(in));
48         for (int i = 0; i < b.size(); ++i)
49             in[i].imag(b[i]);
50         fft(in);
51         for (C &x : in)
52             x *= x;
53         for (int i = 0; i < n; ++i)
54             out[i] = in[-i & (n - 1)] - conj(in[i]);
55         fft(out);
56         for (int i = 0; i < res.size(); ++i)
57             res[i] = imag(out[i]) / (4 * n);
58         vector<long long> arr(res.size());
59         for (int i = 0; i < res.size(); ++i)
60             arr[i] = round(res[i]);
61         return arr;
62     }

```

7.12. Inclusion Exclusion

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

7.13. Inclusion Exclusion

```

1  // |A ∪ B ∪ C| = |A| + |B| + |C| - |A ∩ B| - |A ∩ C| - |B ∩ C| + |A ∩ B ∩ C|
2  // EXAMPLE: How many numbers from 1 to 10^9 are multiple of 42, 54, 137 or
3  // 201?
4  int f(const vector<int> &arr, const int LIMIT) {
5      int n = arr.size();
6      int c = 0;
7
8      for (int mask = 1; mask < (1ll << n); mask++) {
9          int lcm = 1;

```

```

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

```

7.14. Karatsuba

```

1   /// Code copied from:
2   ///
3   /// https://github.com/iam0rch1d/algospot/blob/98476cf0513967cd2481d8dc8dc0201598420979/fanmeeting.cpp
4   const int MINIMUM_KARATSUBA_A_SIZE = 50;
5
6   vector<int> multiply(const vector<int> &a, const vector<int> &b) {
7       vector<int> multiplication(a.size() + b.size() + 1, 0);
8       for (int i = 0; i < a.size(); i++)
9           for (int j = 0; j < b.size(); j++)
10               multiplication[i + j] += a[i] * b[j];
11       return multiplication;
12   }
13
14   void add(vector<int> &to, vector<int> &howMuch, int howMuchExponent) {
15       const int howMuchSize = howMuch.size();
16
17       if (to.size() < howMuch.size() + howMuchExponent)
18           to.resize(howMuch.size() + howMuchExponent);
19
20       for (int i = 0; i < howMuchSize; i++)
21           to[howMuchExponent + i] += howMuch[i];
22   }
23
24   void subtract(vector<int> &from, vector<int> &howMuch) {
25       for (int i = 0; i < howMuch.size(); i++)
26           from[i] -= howMuch[i];
27   }
28
29   /// Multiplies two polynomials a and b using Karatsuba algorithm.
30   ///
31   /// Time complexity: O(n^(1.59))
32   vector<int> multiplyKaratsuba(const vector<int> &a, const vector<int> &b) {
33       const int aSize = a.size(), bSize = b.size();
34
35       if (aSize < bSize)
36           return multiplyKaratsuba(b, a);
37
38       if (aSize == 0 || bSize == 0)
39           return vector<int>();
40
41       if (aSize < MINIMUM_KARATSUBA_A_SIZE)
42           return multiply(a, b);
43
44       const int aNumberHalfSize = aSize / 2;
45       vector<int> aDivision0(a.begin(), a.begin() + aNumberHalfSize);
46       vector<int> aDivision1(a.begin() + aNumberHalfSize, a.end());
47       vector<int> bDivision0(b.begin(),
48                           b.begin() + min<int>(bSize, aNumberHalfSize));

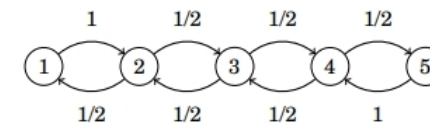
```

```

49       vector<int> bDivision1(b.begin() + min<int>(bSize, aNumberHalfSize),
50                           b.end());
51       vector<int> karatsubaFactor0 = multiplyKaratsuba(aDivision0, bDivision0);
52       vector<int> karatsubaFactor2 = multiplyKaratsuba(aDivision1, bDivision1);
53
54       add(aDivision0, aDivision1, 0);
55       add(bDivision0, bDivision1, 0);
56
57       vector<int> karatsubaFactor1 = multiplyKaratsuba(aDivision0, bDivision0);
58       subtract(karatsubaFactor1, karatsubaFactor0);
59       subtract(karatsubaFactor1, karatsubaFactor2);
60
61       vector<int> multiplication;
62       add(multiplication, karatsubaFactor0, 0);
63       add(multiplication, karatsubaFactor1, aNumberHalfSize);
64       add(multiplication, karatsubaFactor2, aNumberHalfSize + aNumberHalfSize);
65
66       return multiplication;
67   }

```

7.15. Markov Chains



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

Probably after moving 1 step from 1

7.16. Matrix Exponentiation

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

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

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

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

Fibonacci

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

7.17. Matrix Exponentiation

```

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

```

```

29    }
30
31    /// Multiplies matrices mat and other.
32    ///
33    /// Time Complexity: O(mat.size() ^ 3)
34    Matrix operator*(const Matrix &other) const {
35        assert(m == other.n);
36        Matrix ans(n, other.m);
37        for (int i = 0; i < n; ++i)
38            for (int j = 0; j < m; ++j)
39                for (int k = 0; k < m; ++k)
40                    ans.mat[i][j] = mod(ans.mat[i][j] + mat[i][k] * other.mat[k][j]);
41        return ans;
42    }
43
44    /// Exponents the matrix mat to the power of p.
45    ///
46    /// Time Complexity: O((mat.size() ^ 3) * log2(p))
47    Matrix pow(int p) {
48        assert(p >= 0);
49        Matrix ans = identity(), cur_power(n, m);
50        cur_power.mat = mat;
51        while (p) {
52            if (p & 1)
53                ans = ans * cur_power;
54
55            cur_power = cur_power * cur_power;
56            p >>= 1;
57        }
58        return ans;
59    }
60 };

```

7.18. Pollard Rho (Factorize)

```

1 // Copied from:
2 // https://codeforces.com/contest/1305/submission/73826085
3 #include <bits/stdc++.h>
4 using namespace std;
5 #define rep(i, from, to) for (int i = from; i < (to); ++i)
6 #define trav(a, x) for (auto &a : x)
7 #define all(x) x.begin(), x.end()
8 #define sz(x) (int)(x).size()
9 typedef long long ll;
10 typedef pair<int, int> pii;
11 typedef vector<int> vi;
12
13 typedef long long ll;
14 typedef unsigned long long ull;
15 typedef long double ld;
16
17 ull gcd(ull u, ull v) {
18     if (u == 0 || v == 0)
19         return v ^ u;
20     int shift = __builtin_ctzll(u | v);
21     u >>= __builtin_ctzll(u);
22     do {
23         v >>= __builtin_ctzll(v);
24         if (u > v) {
25             ull t = v;
26             v = u;
27             u = t;
28         }
29         v -= u;

```

```

30     } while (v);
31     return u << shift;
32 }
33
34 ull mod_mul(ull a, ull b, ull M) {
35     ll ret = a * b - M * ull(1 / (double)M * a * b);
36     return ret + M * (ret < 0) - M * (ret >= (ll)M);
37 }
38
39 ull mod_pow(ull b, ull e, ull mod) {
40     ull ans = 1;
41     for (; e; b = mod_mul(b, b, mod), e /= 2)
42         if (e & 1)
43             ans = mod_mul(ans, b, mod);
44     return ans;
45 }
46
47 bool isPrime(ull n) {
48     if (n < 2 || n % 6 % 4 != 1)
49         return (n | 1) == 3;
50     ull A[] = {2, 13, 23, 1662803}, s = __builtin_ctzll(n - 1), d = n >> s;
51     for (auto a : A) { // ^ count trailing zeroes
52         ull p = mod_pow(a % n, d, n), i = s;
53         while (p != 1 && p != n - 1 && a % n && i--)
54             p = mod_mul(p, p, n);
55         if (p != n - 1 && i != s)
56             return 0;
57     }
58     return 1;
59 }
60 typedef ull u64;
61 typedef unsigned int u32;
62
63 typedef __uint128_t u128;
64 // typedef __int128_t i128;
65 typedef long long i64;
66 typedef unsigned long long u64;
67
68 u64 hi(u128 x) { return (x >> 64); }
69 u64 lo(u128 x) { return (x << 64) >> 64; }
70 struct Mont {
71     Mont(u64 n) : mod(n) {
72         inv = n;
73         rep(i, 0, 6) inv *= 2 - n * inv;
74         r2 = -n % n;
75         rep(i, 0, 4) if ((r2 <= 1) >= mod) r2 -= mod;
76         rep(i, 0, 5) r2 = mul(r2, r2);
77     }
78     u64 reduce(u128 x) const {
79         u64 y = hi(x) - hi(u128(lo(x) * inv) * mod);
80         return i64(y) < 0 ? y + mod : y;
81     }
82     u64 reduce(u64 x) const { return reduce(x); }
83     u64 init(u64 n) const { return reduce(u128(n) * r2); }
84     u64 mul(u64 a, u64 b) const { return reduce(u128(a) * b); }
85     u64 mod, inv, r2;
86 };
87
88 ull pollard(ull n) {
89     if (n == 9)
90         return 3;
91     if (n == 25)
92         return 5;
93     if (n == 49)
94         return 7;

```

```

95     if (n == 323)
96         return 17;
97     Mont mont(n);
98     auto f = [n, &mont](ull x) { return mont.mul(x, x) + 1; };
99     ull x = 0, y = 0, t = 0, prd = 2, i = 1, q;
100    while (t++ % 32 || gcd(prd, n) == 1) {
101        if (x == y)
102            x = ++i, y = f(x);
103        if ((q = mont.mul(prd, max(x, y) - min(x, y)))
104            prd = q;
105            x = f(x), y = f(f(y));
106        }
107        return gcd(prd, n);
108    }
109
110 unordered_set<ll> primes;
111 unordered_set<ll> seen;
112 set<ll> prm;
113 void factor(ull n) {
114     if (n <= 1 || seen.count(n))
115         return;
116     seen.insert(n);
117     if (isPrime(n)) {
118         primes.insert(n);
119         prm.insert(n);
120     } else {
121         ull x = pollard(n);
122         factor(x), factor(n / x);
123     }
124 }
125 signed main() {
126     ull x;
127     // Factorizes 3e4 numbers in less than 1 sec in my PC.
128     for (int i = 0; i < 30000; i++) {
129         prm.clear();
130         seen.clear();
131         cin >> x;
132         factor(x);
133         // for (ll y : prm) {
134         //     cout << y << " ";
135         //     while (x % y == 0)
136         //         x /= y;
137         // }
138         // cout << endl;
139         // assert(x == 1);
140     }
141     cout << endl;
142 }

```

7.19. Pollard Rho (Find A Divisor)

```

1 // Requires binary_exponentiation.cpp
2
3 /// Returns a prime divisor for n.
4 ///
5 /// Expected Time Complexity: O(n1/4)
6 int pollard_rho(const int n) {
7     srand(time(NULL));
8
9     /* no prime divisor for 1 */
10    if (n == 1)
11        return n;
12
13    if (n % 2 == 0)

```

```

14     return 2;
15
16     /* we will pick from the range [2, N) */
17     int x = (rand() % (n - 2)) + 2;
18     int y = x;
19
20     /* the constant in f(x).
21     * Algorithm can be re-run with a different c
22     * if it throws failure for a composite. */
23     int c = (rand() % (n - 1)) + 1;
24
25     /* Initialize candidate divisor (or result) */
26     int d = 1;
27
28     /* until the prime factor isn't obtained.
29     If n is prime, return n */
30     while (d == 1) {
31         /* Tortoise Move: x(i+1) = f(x(i)) */
32         x = (modular_pow(x, 2, n) + c + n) % n;
33
34         /* Hare Move: y(i+1) = f(f(y(i))) */
35         y = (modular_pow(y, 2, n) + c + n) % n;
36         y = (modular_pow(y, 2, n) + c + n) % n;
37
38         d = __gcd(abs(x - y), n);
39
40         /* retry if the algorithm fails to find prime factor
41         * with chosen x and c */
42         if (d == n)
43             return pollard_rho(n);
44     }
45
46     return d;
47 }

```

7.20. Polynomial Convolution

```

1  /// Returns the resulting polynomial after convolution of polynomials a and
2  b.
3  ///
4  /// Time Complexity: O(a.size() * b.size())
5  vector<int> convolution(const vector<int> &a, const vector<int> &b) {
6      const int n = a.size(), m = b.size();
7      vector<int> ans(n + m - 1);
8      for (int i = 0; i < n; ++i)
9          for (int j = 0; j < m; ++j)
10             ans[i + j] += a[i] * b[j];
11     return ans;
12 }

```

7.21. Primality Check

```

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

```

```

12     return false;
13     return true;
14 }

```

7.22. Primes

```

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

```

7.23. Sieve + Segmented Sieve

```

1  const int MAXN = 1e6;
2
3  /// Contains all the primes in the segments
4  vector<int> segPrimes;
5  bitset<MAXN + 5> primesInSeg;
6
7  /// smallest prime factor
8  vector<int> spf(MAXN + 5);
9
10 vector<int> primes;
11 bitset<MAXN + 5> isPrime;
12
13 void sieve(int n = MAXN + 2) {
14     iota(sp.begin(), sp.end(), 0ll);
15     isPrime.set();
16     for (int64_t i = 2; i <= n; i++) {
17         if (isPrime[i]) {
18             for (int64_t j = i * i; j <= n; j += i) {
19                 isPrime[j] = false;
20                 spf[j] = min(i, int64_t(spf[j]));
21             }
22             primes.emplace_back(i);
23         }
24     }
25 }
26
27 vector<int> getFactorization(int x) {
28     vector<int> ret;
29     while (x != 1) {
30         ret.emplace_back(spf[x]);
31         x = x / spf[x];
32     }
33     return ret;
34 }
35
36 /// Gets all primes from 1 to r

```

```

37 void segSieve(int l, int r) {
38     // primes from l to r
39     // transferred to 0..(l-r)
40     segPrimes.clear();
41     primesInSeg.set();
42     int sq = sqrt(r) + 5;
43
44     for (int p : primes) {
45         if (p > sq)
46             break;
47
48         for (int i = l - l % p; i <= r; i += p) {
49             if (i - l < 0)
50                 continue;
51
52             // if i is less than 1e6, it could be checked in the
53             // array of the sieve
54             if (i >= (int)1e6 || !isPrime[i])
55                 primesInSeg[i - l] = false;
56         }
57     }
58
59     for (int i = 0; i < r - l + 1; i++) {
60         if (primesInSeg[i])
61             segPrimes.emplace_back(i + l);
62     }
63 }

```

7.24. Stars And Bars

I. positive integers x_i

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

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

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

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

II. non-negative integers x_i

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

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

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

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

8. Miscellaneous

8.1. 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

```

8.2. Counting Number Of Digits Up To N

```

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

```



```

7   }
8   return ret;
9 }

```

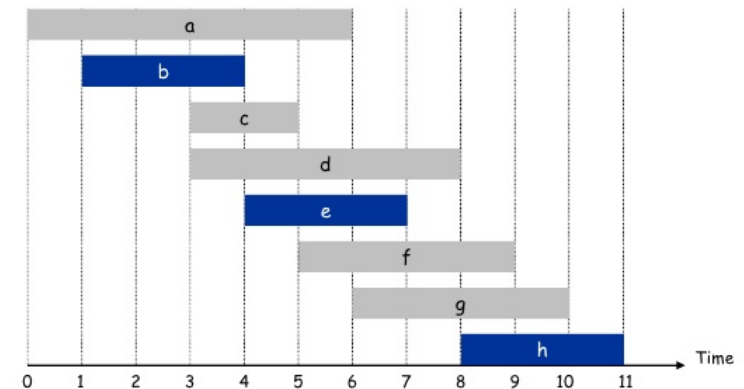
8.3. Infix To Postfix

```

1  /// Code copied from:
2  /// https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/
3  /// Infix Expression | Prefix Expression | Postfix Expression
4  ///   A + B       |   + A B       |   A B +
5  ///   A + B * C    |   + A * B C    |   A B C * +
6  /// Time Complexity: O(n)
7  int infix_to_postfix(const string &infix) {
8      map<char, int> prec;
9      stack<char> op;
10     string postfix;
11
12     prec['+'] = prec['-'] = 1;
13     prec['*'] = prec['/'] = 2;
14     prec['^'] = 3;
15     for (int i = 0; i < infix.size(); ++i) {
16         char c = infix[i];
17         if (is_digit(c)) {
18             while (i < infix.size() && isdigit(infix[i])) {
19                 postfix += infix[i];
20                 ++i;
21             }
22             --i;
23         } else if (isalpha(c))
24             postfix += c;
25         else if (c == '(')
26             op.push('(');
27         else if (c == ')') {
28             while (!op.empty() && op.top() != '(') {
29                 postfix += op.top();
30                 op.pop();
31             }
32             op.pop();
33         } else {
34             while (!op.empty() && prec[op.top()] >= prec[c]) {
35                 postfix += op.top();
36                 op.pop();
37             }
38             op.push(c);
39         }
40     }
41     while (!op.empty()) {
42         postfix += op.top();
43         op.pop();
44     }
45     return postfix;
46 }

```

8.4. Interval Scheduling



8.5. Interval Scheduling

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

8.6. Iterate Over Subsets Of Mask

```

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

```

8.7. 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 }

```

8.8. Kadane (Segment Tree)

```

1 struct Seg_Tree {
2     struct Node {
3         int pref, suf, tot, best;
4         Node() {}
5     }
6 };

```

```

5   Node(int pref, int suf, int tot, int best)
6       : pref(pref), suf(suf), tot(tot), best(best) {}
7   };
8
9   int n;
10  vector<Node> tree;
11  vi arr;
12
13  Seg_Tree(vi &arr) : n(arr.size()), arr(arr) {
14      tree.resize(4 * n);
15      build(0, n - 1, 0);
16  }
17
18  Node query(const int l, const int r, const int i, const int j,
19             const int pos) {
20      if (l > r || l > j || r < i)
21          return Node(-INF, -INF, -INF, -INF);
22
23      if (i <= l && r <= j)
24          return Node(tree[pos].pref, tree[pos].suf, tree[pos].tot,
25                      tree[pos].best);
26
27      int mid = (l + r) / 2;
28      Node left = query(l, mid, i, j, 2 * pos + 1),
29              right = query(mid + 1, r, i, j, 2 * pos + 2);
30      Node x;
31      x.pref = max({left.pref, left.tot, left.tot + right.pref});
32      x.suf = max({right.suf, right.tot, right.tot + left.suf});
33      x.tot = left.tot + right.tot;
34      x.best = max({left.best, right.best, left.suf + right.pref});
35      return x;
36  }
37
38  // Update arr[idx] to v
39  // ITS NOT DELTA!!!
40  void update(int l, int r, const int idx, const int v, const int pos) {
41      if (l > r || l > idx || r < idx)
42          return;
43
44      if (l == idx && r == idx) {
45          tree[pos] = Node(v, v, v, v);
46          return;
47      }
48
49      int mid = (l + r) / 2;
50      update(l, mid, idx, v, 2 * pos + 1);
51      update(mid + 1, r, idx, v, 2 * pos + 2);
52      l = 2 * pos + 1, r = 2 * pos + 2;
53      tree[pos].pref =
54          max({tree[l].pref, tree[l].tot, tree[l].tot + tree[r].pref});
55      tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot +
56                          tree[l].suf});
57      tree[pos].tot = tree[l].tot + tree[r].tot;
58      tree[pos].best =
59          max({tree[l].best, tree[r].best, tree[l].suf + tree[r].pref});
60
61  void build(int l, int r, const int pos) {
62      if (l == r) {
63          tree[pos] = Node(arr[l], arr[l], arr[l], arr[l]);
64          return;
65      }
66
67      int mid = (l + r) / 2;
68      build(l, mid, 2 * pos + 1);

```

```

68      build(mid + 1, r, 2 * pos + 2);
69      l = 2 * pos + 1, r = 2 * pos + 2;
70      tree[pos].pref =
71          max({tree[l].pref, tree[l].tot, tree[l].tot + tree[r].pref});
72      tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot +
73                          tree[l].suf});
74      tree[pos].tot = tree[l].tot + tree[r].tot;
75      tree[pos].best =
76          max({tree[l].best, tree[r].best, tree[l].suf + tree[r].pref});
77  }
78  };

```

8.9. Kadane 2D

```

1  /// Code copied from:
2  /// https://www.geeksforgeeks.org/maximum-sum-rectangle-in-a-2d-matrix-dp-27/
3  /// Program to find maximum sum subarray in a given 2D array
4  const int ROW = 1001, COL = 1001;
5  int mat[ROW][COL];
6
7  // Implementation of Kadane's algorithm for 1D array. The function
8  // returns the maximum sum and stores starting and ending indexes of the
9  // maximum sum subarray at addresses pointed by start and finish pointers
10 // respectively.
11 int kadane(int *arr, int *start, int *finish, int n) {
12     // initialize sum, maxSum and
13     int sum = 0, maxSum = INT_MIN, i;
14
15     // Just some initial value to check for all negative values case
16     *finish = -1;
17
18     // local variable
19     int local_start = 0;
20
21     for (i = 0; i < n; ++i) {
22         sum += arr[i];
23         if (sum < 0) {
24             sum = 0;
25             local_start = i + 1;
26         } else if (sum > maxSum) {
27             maxSum = sum;
28             *start = local_start;
29             *finish = i;
30         }
31     }
32
33     // There is at-least one non-negative number
34     if (*finish != -1)
35         return maxSum;
36
37     // Special Case: When all numbers in arr[] are negative
38     maxSum = arr[0];
39     *start = *finish = 0;
40
41     // Find the maximum element in array
42     for (i = 1; i < n; i++) {
43         if (arr[i] > maxSum) {
44             maxSum = arr[i];
45             *start = *finish = i;
46         }
47     }
48     return maxSum;
49 }
50

```

```

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

```

8.10. 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 }

```

8.11. Point Compression

```

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

```

8.12. 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 /// Time Complexity: O(log3(n))
7 int ternary_search(const vector<int> &arr) {
8     int l = 0, r = (int)arr.size() - 1;
9     while (r - l > 2) {
10         int lc = l + (r - l) / 3;
11         int rc = r - (r - l) / 3;
12         // the function f(x) returns the element on the position x
13         if (f(lc) > f(rc))
14             // the function is going down, then the middle is on the right.
15             l = rc;
16         else
17             r = lc;
18     }
19     // the range [l, r] contains the minimum element.
20
21     int minn = f(l), idx = l;
22     for (int i = l + 1; i <= r; ++i)
23         if (f(i) < minn) {
24             idx = i;
25             minn = f(i);
26         }
27     return idx;
28 }

```

8.13. Tower Of Hanoi

```

1 /// Code copied from:
2 /// https://www.geeksforgeeks.org/recursive-tower-hanoi-using-4-pegs-rods/
3 void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
4     if (n == 1) {

```

```

5     printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
6     return;
7 }
8 towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
9 printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
10 towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
11 }
12
13 int main() {
14     int n = 4; // Number of disks
15     towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
16 }

```

8.14. Two Sat

```

1 // OBS: INDEXED FROM 0
2 // USE POS_X = 1 FOR POSITIVE CLAUSES AND 0 FOR NEGATIVE. OTHERWISE THE FINAL
3 // ANSWER ARRAY WILL BE FLIPPED.
4 class Two_Sat {
5 private:
6     vector<vector<int>> adj;
7     int n;
8
9 public:
10     Two_Sat(const int n) : n(n) {
11         adj.resize(2 * n);
12         ans.resize(n);
13     }
14
15     // (X v Y) = (~X -> Y) & (~Y -> X)
16     void add_or(const int x, const bool pos_x, const int y, const bool pos_y) {
17         assert(0 <= x), assert(x < n), assert(0 <= y), assert(y < n);
18         adj[(x << 1) ^ (pos_x ^ 1)].emplace_back((y << 1) ^ pos_y);
19         adj[(y << 1) ^ (pos_y ^ 1)].emplace_back((x << 1) ^ pos_x);
20     }
21
22     // (X xor Y) = (X v Y) & (~X v ~Y)
23     // for this operation the result is always 0 1 or 1 0
24     void add_xor(const int x, const bool pos_x, const int y, const bool pos_y)
25     {
26         assert(0 <= x), assert(x < n), assert(0 <= y), assert(y < n);
27         add_or(x, pos_x, y, pos_y);
28         add_or(x, pos_x ^ 1, y, pos_y ^ 1);
29     }
30
31     vector<bool> ans;
32     // Checks whether the system is feasible or not. If it's feasible, it
33     // stores
34     // a satisfiable answer in the array 'ans'.
35     //
36     // Time Complexity: O(n)
37     bool check() {
38         SCC scc(2 * n, 0, adj);
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         return true;
45     }
46 };

```

9. Stress Testing

9.1. Check

```

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

```

9.2. Gen

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define eb emplace_back
6 #define ii pair<int, int>
7 #define OK (cerr << "OK" << endl)
8 #define debug(x) cerr << #x " = " << (x) << endl
9 #define ff first
10 #define ss second
11 #define int long long
12 #define tt tuple<int, int, int>
13 #define all(x) x.begin(), x.end()
14 #define vi vector<int>
15 #define vii vector<pair<int, int>>
16 #define vvi vector<vector<int>>
17 #define vvii vector<vector<pair<int, int>>>
18 #define Matrix(n, m, v) vector<vector<int>>(n, vector<int>(m, v))
19 #define endl '\n'
20
21 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
22
23 // Generates a string of (n) characters from 'a' to 'a' + (c)
24 string str(const int n, const int c);
25 // Generates (size) strings of (n) characters from 'a' to 'a' + (c)
26 string spaced_str(const int n, const int size, const int c);
27 // Generates a string of (n) 01 characters.
28 string str01(const int n);
29 // Generates a number in the range [l, r].
30 int num(const int l, const int r);
31 // Generates a vector of (n) numbers in the range [l, r].
32 vector<int> vec(const int n, const int l, const int r);
33 // Generates a matrix of (n x m) numbers in the range [l, r].

```

```

34 vector<vector<int>> matrix(const int n, const int m, const int l, const int
    r);
35 // Generates a tree with n vertices
36 vector<pair<int, int>> tree(const int n);
37 // Generates a forest with n vertices.
38 vector<pair<int, int>> forest(const int n);
39 // Generates a connected graph with n vertices.
40 vector<pair<int, int>> connected_graph(const int n);
41 // Generates a graph with n vertices.
42 vector<pair<int, int>> graph(const int n);
43
44 signed main() {
45     int t = num(1, 1);
46     // cout << t << endl;
47     while (t--) {
48         int n = num(1, 2e5);
49         int m = num(1, 2e5);
50         cout << n << endl;
51     }
52 }
53
54 vector<pair<int, int>> tree(const int n) {
55     const int root = num(1, n);
56     vector<int> v1, v2;
57     v1.emplace_back(root);
58     for (int i = 1; i <= n; ++i)
59         if (i != root)
60             v2.emplace_back(i);
61     random_shuffle(all(v2));
62     vector<pair<int, int>> edges;
63     while (!v2.empty()) {
64         const int idx = num(0, (int)v1.size() - 1);
65         edges.emplace_back(v1[idx], v2.back());
66         v1.emplace_back(v2.back());
67         v2.pop_back();
68     }
69     return edges;
70 }
71
72 vector<pair<int, int>> forest(const int n) {
73     int val = n;
74     vector<pair<int, int>> edges;
75     int oft = 0;
76     while (val > 0) {
77         const int cur = num(1, val);
78         auto e = tree(cur);
79         for (auto [u, v] : e)
80             edges.emplace_back(u + oft, v + oft);
81         val -= cur;
82         oft += cur;
83     }
84     return edges;
85 }
86
87 vector<pair<int, int>> connected_graph(const int n) {
88     auto e = tree(n);
89     set<pair<int, int>> s(e.begin(), e.end());
90     const int ERROR = n;
91     int q = num(0, max(0ll, (n - 1) * (n - 2)) / 2 + ERROR);
92     while (q--) {
93         int u = num(1, n), v = num(1, n);
94         if (u == v || s.count(make_pair(u, v)) || s.count(make_pair(v, u)))
95             continue;
96         e.emplace_back(u, v);
97         s.emplace(u, v);

```

```

98     }
99     return e;
100 }
101
102 vector<pair<int, int>> graph(const int n) {
103     int q = num(0, n * (n - 1) / 2);
104     set<pair<int, int>> s;
105     while (q--) {
106         int u = num(1, n), v = num(1, n);
107         if (u == v)
108             continue;
109         if (u > v)
110             swap(u, v);
111         s.emplace(u, v);
112     }
113     vector<pair<int, int>> edges;
114     for (auto [u, v] : s) {
115         if (rng() % 2)
116             swap(u, v);
117         edges.emplace(u, v);
118     }
119     return edges;
120 }
121
122 int num(const int l, const int r) {
123     int sz = r - l + 1;
124     int n = rng() % sz;
125     return n + l;
126 }
127
128 vector<int> vec(const int n, const int l, const int r) {
129     vector<int> arr(n);
130     for (int &x : arr)
131         x = num(l, r);
132     return arr;
133 }
134
135 vector<vector<int>> matrix(const int n, const int m, const int l, const int
    r) {
136     vector<vector<int>> mt;
137     for (int i = 0; i < n; ++i)
138         mt.emplace_back(vec(m, l, r));
139     return mt;
140 }
141
142 string str(const int n, const int c = 26) {
143     string ans;
144     for (int i = 0; i < n; ++i)
145         ans += char(rng() % c + 'a');
146     return ans;
147 }
148
149 string str01(const int n) {
150     string ans;
151     for (int i = 0; i < n; ++i) {
152         ans += char(rng() % 2 + '0');
153     }
154     return ans;
155 }
156
157 string spaced_str(const int n, const int size, const int c = 26) {
158     string ans;
159     for (int i = 0; i < size; ++i) {
160         if (i)
161             ans += ' ';

```

```

162     ans += str(n, c);
163 }
164 return ans;
165 }

```

9.3. Run

```

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

```

10. Strings

10.1. 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     this->build_failures();
35 }
36
37
38 /// Returns the fail node of cur.
39 Trie::Node *find_fail_node(Trie::Node *cur, char c) {
40     while (cur != this->trie.root() && !cur->next.count(c))
41         cur = fail[cur->id];
42     // if cur is pointing to the root node and c is not a child
43     if (!cur->next.count(c))
44         return trie.root();
45     return cur->next[c];
46 }
47
48 /// Time Complexity: O(number of characters in arr)
49 void build_failures() {
50     queue<const Trie::Node*> q;
51
52     fail[trie.root()->id] = trie.root();
53     for (const pair<char, Trie::Node*> v : trie.root()->next) {
54         q.emplace(v.second);
55         fail[v.second->id] = trie.root();
56         out_node[v.second->id]->next = out_node[trie.root()->id];
57     }
58
59     while (!q.empty()) {
60         const Trie::Node *u = q.front();
61         q.pop();
62
63         for (const pair<char, Trie::Node*> x : u->next) {
64             const char c = x.first;
65             const Trie::Node *v = x.second;
66             Trie::Node *fail_node = find_fail_node(fail[u->id], c);
67             fail[v->id] = fail_node;
68
69             if (!out_node[fail_node->id]->str_idx.empty())
70                 out_node[v->id]->next = out_node[fail_node->id];
71             else
72                 out_node[v->id]->next = out_node[fail_node->id]->next;
73
74             q.emplace(v);
75         }
76     }
77 }
78
79 vector<vector<pair<int, int>>> aho_find_occurrences(const string &text) {
80     vector<vector<pair<int, int>>> ans(arr.size());
81     Trie::Node *cur = trie.root();
82
83     for (int i = 0; i < text.size(); ++i) {
84         cur = find_fail_node(cur, text[i]);
85         for (Out_Node *node = out_node[cur->id]; node != nullptr;
86              node = node->next)
87             for (const int idx : node->str_idx)
88                 ans[idx].emplace_back(i - (int)arr[idx].size() + 1, i);
89     }
90     return ans;
91 }
92
93 public:
94     /// Constructor that builds the trie and the failures.
95     ///
96     /// Time Complexity: O(number of characters in arr)
97     Aho(const vector<string> &arr) : arr(arr) { this->build_trie(); }
98 }

```

```

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

```

10.2. 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));

```

```

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

```

10.3. Kmp

```

1  /// Builds the pi array for the KMP algorithm.
2  ///
3  /// Time Complexity: O(n)
4  vector<int> pi(const string &pat) {
5      vector<int> ans(pat.size() + 1, -1);
6      int i = 0, j = -1;
7      while (i < pat.size()) {
8          while (j >= 0 && pat[i] != pat[j])
9              j = ans[j];
10         ++i, ++j;
11         ans[i] = j;
12     }
13     return ans;
14 }
15
16 /// Returns the occurrences of a pattern in a text.
17 ///
18 /// Time Complexity: O(n + m)
19 vector<int> kmp(const string &txt, const string &pat) {
20     vector<int> p = pi(pat);
21     vector<int> ans;
22
23     for (int i = 0, j = 0; i < txt.size(); ++i) {
24         while (j >= 0 && pat[j] != txt[i])
25             j = p[j];
26         if (++j == pat.size()) {
27             ans.emplace_back(i);
28             j = p[j];
29         }
30     }
31     return ans;

```

32 }

10.4. Lcs K Strings

```

1 // Make the change below in SuffixArray code.
2 int MaximumNumberOfStrings;
3
4 /// Program to find the LCS between k different strings.
5 ///
6 /// Time Complexity: O(n*log(n))
7 /// Space Complexity: O(n*log(n))
8 int main() {
9     int n;
10    cin >> n;
11    MaximumNumberOfStrings = n;
12
13    vector<string> arr(n);
14
15    int sum = 0;
16    for (string &x : arr) {
17        cin >> x;
18        sum += x.size() + 1;
19    }
20
21    string concat;
22    vector<int> ind(sum + 1);
23    int cnt = 0;
24    for (string &x : arr) {
25        if (concat.size())
26            concat += (char)cnt;
27        concat += x;
28    }
29
30    cnt = 0;
31    for (int i = 0; i < concat.size(); i++) {
32        ind[i + 1] = cnt;
33        if (concat[i] < MaximumNumberOfStrings)
34            cnt++;
35    }
36
37    Suffix_Array say(concat);
38    vector<int> sa = say.get_suffix_array();
39    Sparse_Table spt(say.get_lcp());
40
41    vector<int> freq(n);
42    int cnt1 = 0;
43
44    /// Ignore separators
45    int i = n, j = n - 1;
46    int ans = 0;
47
48    while (true) {
49        if (cnt1 == n) {
50            ans = max(ans, spt.query(i, j - 1));
51            int idx = ind[sa[i]];
52            freq[idx]--;
53            if (freq[idx] == 0)
54                cnt1--;
55            i++;
56        } else if (j == (int)sa.size() - 1)
57            break;
58        else {
59            j++;
60            int idx = ind[sa[j]];

```

```

61        freq[idx]++;
62        if (freq[idx] == 1)
63            cnt1++;
64    }
65
66    cout << ans << endl;
67
68 }

```

10.5. Lexicographically Smallest Rotation

```

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

```

10.6. Manacher (Longest Palindrome)

```

1 /// Copied from:
2 ///
3     https://medium.com/hackernoon/manachers-algorithm-explained-longest-palindromic-s
4
5 /// Create a string containing '#' characters between any two characters.
6 string get_modified_string(string &s) {
7     string ret;
8     for (int i = 0; i < s.size(); i++) {
9         ret.push_back('#');
10        ret.push_back(s[i]);
11    }
12    ret.push_back('#');
13    return ret;
14
15 /// Returns the first occurrence of the longest palindrome based on the lps
16 /// array.
17 ///
18 /// Time Complexity: O(n)
19 string get_best(const int max_len, const string &str, const vector<int>
    &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
49 /// Returns a pair containing the size of the longest palindrome and the
50 /// occurrence of it.
51 ///
52 /// Time Complexity: O(n)
53 pair<int, string> manacher(string &s) {
54     int n = s.size();
55     string str = get_modified_string(s);
56     int len = (2 * n) + 1;
57     // the i-th index contains the longest palindromic substring with the i-th
58     // char as the center
59     vector<int> lps(len);
60     int c = 0; // stores the center of the longest palindromic substring until
61     now
62     int r = 0; // stores the right boundary of the longest palindromic
63     substring
64     // until now
65     int max_len = 0;
66     for (int i = 0; i < len; i++) {
67         // get mirror index of i
68         int mirror = (2 * c) - i;
69
70         // see if the mirror of i is expanding beyond the left boundary of
71         // current
72         // longest palindrome at center c if it is, then take r - i as lps[i]
73         else
74             // take lps[mirror] as lps[i]
75             if (i < r)
76                 lps[i] = min(r - i, lps[mirror]);
77
78         // expand at i
79         int a = i + (1 + lps[i]);
80         int b = i - (1 + lps[i]);
81         while (a < len && b >= 0 && str[a] == str[b]) {
82             lps[i]++;
83             a++;
84             b--;
85         }
86
87         // check if the expanded palindrome at i is expanding beyond the right

```

```

84     // boundary of current longest palindrome at center c if it is, the new
85     // center is i
86     if (i + lps[i] > r) {
87         c = i;
88         r = i + lps[i];
89
90         if (lps[i] > max_len) // update max_len
91             max_len = lps[i];
92     }
93 }
94
95 return make_pair(max_len, get_best(max_len, str, lps));
96 }

```

10.7. Suffix Array

```

1 // #define LCP
2 // clang-format off
3 class Suffix_Array {
4 private:
5     const string s;
6     const int n;
7
8 private:
9     /// OBS: Suffix Array build code imported from:
10    ///
11    /// https://github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/String/SuffixArray.cpp
12    /// Time Complexity: O(n*(log n))
13    vector<int> build_suffix_array() {
14        int c = 0;
15        vector<int> temp(n), posBucket(n), bucket(n), bpos(n), out(n);
16        for (int i = 0; i < n; i++)
17            out[i] = i;
18        sort(out.begin(), out.end(),
19             [&](int a, int b) { return this->s[a] < this->s[b]; });
20        for (int i = 0; i < n; i++) {
21            bucket[i] = c;
22            if (i + 1 == n || this->s[out[i]] != this->s[out[i + 1]])
23                c++;
24        }
25        for (int h = 1; h < n && c < n; h <= 1) {
26            for (int i = 0; i < n; i++)
27                posBucket[out[i]] = bucket[i];
28            for (int i = n - 1; i >= 0; i--)
29                bpos[bucket[i]] = i;
30            for (int i = 0; i < n; i++)
31                if (out[i] >= n - h)
32                    temp[bpos[bucket[i]]++] = out[i];
33            for (int i = 0; i < n; i++)
34                if (out[i] >= h)
35                    temp[bpos[posBucket[out[i] - h]]++] = out[i] - h;
36            c = 0;
37            for (int i = 0; i + 1 < n; i++) {
38                const int tmp = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h) ||
39                    (posBucket[temp[i + 1] + h] != posBucket[temp[i] +
40                    h]);
41                bucket[i] = c;
42                c += tmp;
43            }
44            bucket[n - 1] = c++;
45            temp.swap(out);
46        }
47        return out;
48    }

```

```

47 }
48
49 vector<int> build_inverse_suffix() {
50     vector<int> inverse_suffix(this->n);
51     for (int i = 0; i < this->n; ++i)
52         inverse_suffix[sa[i]] = i;
53     return inverse_suffix;
54 }
55
56 #ifdef LCP
57 /// Builds the lcp (Longest Common Prefix) array for the string s.
58 /// A value lcp[i] indicates length of the longest common prefix of the
59 /// suffixes indexed by i and i + 1. Implementation of the Kasai's
    Algorithm.
60 ///
61 /// Time Complexity: O(n)
62 vector<int> build_lcp() {
63     vector<int> lcp(this->n, 0);
64     for (int i = 0, k = 0; i < this->n; ++i) {
65         if (inverse_suffix[i] == this->n - 1)
66             k = 0;
67         else {
68             const int j = sa[inverse_suffix[i] + 1];
69             while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
70                 ++k;
71             lcp[inverse_suffix[i]] = k;
72             k -= k > 0;
73         }
74     }
75     return lcp;
76 }
77
78 int _lcs(const int separator) {
79     int ans = 0;
80     for (int i = 0; i + 1 < this->sa.size(); ++i) {
81         const int left = this->sa[i], right = this->sa[i + 1];
82         if ((left < separator && right > separator) ||
83             (left > separator && right < separator))
84             ans = max(ans, lcp[i]);
85     }
86     return ans;
87 }
88 #endif
89
90 /// Returns the minimum index, in the range [l, r], in which after advance
    i
91 /// positions the character c is present.
92 ///
93 /// Time Complexity: O(log n)
94 int lower(const char c, const int i, int l, int r) {
95     int ans = -1;
96     while (l <= r) {
97         int mid = (l + r) / 2;
98         if (sa[mid] + i < s.size() && s[sa[mid] + i] >= c) {
99             ans = mid;
100             r = mid - 1;
101         } else
102             l = mid + 1;
103     }
104     return ans;
105 };
106
107 /// Returns the maximum index in the range [l, r], such that after advance
    i
108 /// positions the character c is present.

```

```

109 ///
110 /// Time Complexity: O(log n)
111 int upper(const char c, const int i, int l, int r) {
112     int ans = -1;
113     while (l <= r) {
114         int mid = (l + r) / 2;
115         if (sa[mid] + i >= s.size() || s[sa[mid] + i] <= c) {
116             ans = mid;
117             l = mid + 1;
118         } else
119             r = mid - 1;
120     }
121     return ans;
122 };
123
124
125 public:
126     Suffix_Array(const string &s) : n(s.size()), s(s) {}
127
128     const vector<int> sa = build_suffix_array();
129     /// Position of the i-th character in suffix array.
130     const vector<int> inverse_suffix = build_inverse_suffix();
131     #ifdef LCP
132     const vector<int> lcp = build_lcp();
133
134     /// LCS of two strings A and B. The string s must be initialized in the
135     /// constructor as the string (A + '$' + B).
136     /// The string A starts at index 1 and ends at index (separator - 1).
137     /// The string B starts at index (separator + 1) and ends at the end of the
138     /// string.
139     ///
140     /// Time Complexity: O(n)
141     int lcs(const int separator) {
142         assert(!isalpha(this->s[separator]) && !isdigit(this->s[separator]));
143         return _lcs(separator);
144     }
145     #endif
146
147     void print() {
148         for(int i = 0; i < n; ++i)
149             cerr << s.substr(sa[i]) << endl;
150     }
151
152     /// Returns the range, inside the range [l, r], in which after advance i
153     /// positions the character c is present.
154     ///
155     /// Time Complexity: O(log n)
156     pair<int, int> range(const char c, const int i, int l, int r) {
157         l = lower(c, i, l, r), r = upper(c, i, l, r);
158         return min(l, r) == -1 ? pair<int, int>(-1, -1) : pair<int, int>(l, r);
159     }
160 };
161 // clang-format on

```

10.8. Suffix Array Mine

```

1 // clang-format off
2 namespace RadixSort {
3     /// Sorts the array arr stably in ascending order.
4     ///
5     /// Time Complexity: O(n + max_element)
6     /// Space Complexity: O(n + max_element)
7     template <typename T>
8     void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),

```

```

9      const int begin = 0) {
10  const int n = arr.size();
11  vector<T> new_order(n);
12  vector<int> count(max_element + 1, 0);
13
14  for (int i = begin; i < n; ++i)
15      ++count[get_key(arr[i])];
16
17  for (int i = 1; i <= max_element; ++i)
18      count[i] += count[i - 1];
19
20  for (int i = n - 1; i >= begin; --i) {
21      new_order[count[get_key(arr[i])] - (begin == 0)] = arr[i];
22      --count[get_key(arr[i])];
23  }
24
25  arr = move(new_order);
26 }
27
28 /// Sorts an array by their pair of ranks stably in ascending order.
29 template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
30     // sort by the second rank
31     RadixSort::sort<T>(
32         arr, rank_size, [](T &item) { return item.first.second; }, 0);
33
34     // sort by the first rank
35     RadixSort::sort<T>(
36         arr, rank_size, [](T &item) { return item.first.first; }, 0);
37 }
38 // namespace RadixSort
39
40 class Suffix_Array {
41     typedef pair<int, int> Rank;
42
43     vector<vector<int>>> rank_table;
44     const vector<int> log_array = build_log_array();
45
46     vector<int> build_log_array() {
47         vector<int> log_array(this->n + 1, 0);
48         for (int i = 2; i <= this->n; ++i)
49             log_array[i] = log_array[i / 2] + 1;
50         return log_array;
51     }
52
53     /// Time Complexity: O(n*log(n))
54     vector<int> build_suffix_array() {
55         // the tuple below represents the rank and the index associated with it
56         vector<pair<Rank, int>>> ranks(this->n);
57         vector<int> arr(this->n);
58
59         for (int i = 0; i < n; ++i)
60             ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);
61
62         #ifdef BUILD_TABLE
63         int rank_table_size = 0;
64         this->rank_table.resize(log_array[this->n] + 2);
65         #endif
66         RadixSort::sort_pairs(ranks, 256);
67         build_ranks(ranks, arr);
68
69         {
70             int jump = 1;
71             int max_rank = arr[ranks.back().second];
72
73             // it will be compared intervals a pair of intervals (i, jump-1), (i +

```

```

74         // jump, i + 2*jump - 1). The variable jump is always a power of 2
75         #ifdef BUILD_TABLE
76         while (jump / 2 < this->n) {
77             #else
78             while (max_rank != this->n) {
79                 #endif
80                 for (int i = 0; i < this->n; ++i) {
81                     ranks[i].first.first = arr[i];
82                     ranks[i].first.second = (i + jump < this->n ? arr[i + jump] : 0);
83                     ranks[i].second = i;
84                 }
85
86                 #ifdef BUILD_TABLE
87                 // inserting only the ranks in the table
88                 transform(ranks.begin(), ranks.end(),
89                     back_inserter(rank_table[rank_table_size++]),
90                     [](pair<Rank, int> &pair) { return pair.first.first; });
91                 #endif
92                 RadixSort::sort_pairs(ranks, n);
93                 build_ranks(ranks, arr);
94
95                 max_rank = arr[ranks.back().second];
96                 jump *= 2;
97             }
98         }
99
100     vector<int> sa(this->n);
101     for (int i = 0; i < this->n; ++i)
102         sa[arr[i] - 1] = i;
103     return sa;
104 }
105
106 int _compare(const int i, const int j, const int length) {
107     const int k = this->log_array[length]; // floor log2(length)
108     const int jump = length - (1ll << k);
109
110     const pair<int, int> iRank = {
111         this->rank_table[k][i],
112         (i + jump < this->n ? this->rank_table[k][i + jump] : -1)};
113     const pair<int, int> jRank = {
114         this->rank_table[k][j],
115         (j + jump < this->n ? this->rank_table[k][j + jump] : -1)};
116     return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
117 }
118
119 /// Compares two substrings beginning at indexes i and j of a fixed length.
120 ///
121 /// Time Complexity: O(1)
122 int compare(const int i, const int j, const int length) {
123     assert(0 <= i && i < this->n && 0 <= j && j < this->n);
124     assert(i + length - 1 < this->n && j + length - 1 < this->n);
125     return _compare(i, j, length);
126 }
127 };
128 // clang-format on

```

10.9. Suffix Automaton

```

1 class Suffix_Automaton {
2 private:
3     struct state {
4         map<char, int> next;
5         /// Length of the current substring which is the longest in the ith
6         class.

```

```

6   /// The range of substring lengths of this class is the following:
7   /// [st[st[u].link].len + 1, len].
8   const int len;
9   /// Contains a link to the state containing the longest suffix of the
10  /// current class which isn't present in it.
11  int link;
12  /// Contains the index of the last position of the first substring.
13  const int first_pos;
14  /// Whether the ith node is terminal or not.
15  bool is_terminal = false;
16
17  state(const map<char, int> next, const int len, const int link,
18        const int first_pos)
19      : next(next), len(len), link(link), first_pos(first_pos) {}
20 };
21 vector<state> st;
22 int last = 0;
23
24 /// Time Complexity: O(n*log(alphabet_size))
25 void build(const string &s) {
26     st.emplace_back(map<char, int>(), 0, -1, -1);
27
28     for (int i = 0; i < s.size(); ++i) {
29         st.emplace_back(map<char, int>(), i + 1, 0, i);
30         const int cur = (int)st.size() - 1;
31
32         int link = last;
33         while (link >= 0 && !st[link].next.count(s[i])) {
34             st[link].next[s[i]] = cur;
35             link = st[link].link;
36         }
37
38         if (link != -1) {
39             const int q = st[link].next[s[i]];
40             if (st[link].len + 1 == st[q].len) {
41                 st[cur].link = q;
42             } else {
43                 st.emplace_back(st[q].next, st[link].len + 1, st[q].link,
44                                st[q].first_pos);
45                 const int qq = (int)st.size() - 1;
46                 st[q].link = st[cur].link = qq;
47                 while (link >= 0) {
48                     auto it = st[link].next.find(s[i]);
49                     if (it == st[link].next.end() || it->second != q)
50                         break;
51                     it->second = qq;
52                     link = st[link].link;
53                 }
54             }
55         }
56         last = cur;
57     }
58 }
59
60 void find_terminals() {
61     int p = last;
62     while (p > 0) {
63         st[p].is_terminal = true;
64         p = st[p].link;
65     }
66 }
67
68 vector<int> dp_ocur;
69 int _ocur(const int idx) {
70     int &ret = dp_ocur[idx];

```

```

71     if (~ret)
72         return ret;
73     ret = st[idx].is_terminal;
74     for (const pair<char, int> &p : st[idx].next)
75         ret += _ocur(p.second);
76     return ret;
77 }
78
79 public:
80     Suffix_Automaton(const string &s) {
81         st.reserve(2 * s.size());
82         build(s);
83         find_terminals();
84     }
85
86     int size() { return st.size(); }
87
88     int link(const int idx) { return st[idx].link; }
89
90     int len(const int idx) { return st[idx].len; }
91
92     int first_pos(const int idx) { return st[idx].first_pos; }
93
94     /// Returns the next state from state cur with character c.
95     /// Returns -1 if this state doesn't exists.
96     int next(const int cur, const char c) {
97         auto it = st[cur].next.find(c);
98         return it == st[cur].next.end() ? -1 : it->second;
99     }
100
101     void print() {
102         cerr << "Terminals" << endl;
103         for (int i = 0; i < st.size(); ++i)
104             if (st[i].is_terminal)
105                 cerr << i << ' ';
106         cerr << endl;
107         cerr << "Edges" << endl;
108         for (int i = 0; i < st.size(); ++i)
109             for (auto [a, b] : st[i].next)
110                 cerr << i << ' ' << b << ' ' << a << endl;
111     }
112
113     /// Returns the number of occurrences of the pattern ending at state idx.
114     ///
115     /// Time Complexity: O(n), amortized for q queries.
116     int ocur(const int idx) {
117         if (dp_ocur.empty())
118             dp_ocur.resize(st.size(), -1);
119         return _ocur(idx);
120     }
121
122     /// Returns the state in which the pattern s ends.
123     ///
124     /// Time complexity: O(s.size())
125     int find(const string &s) {
126         int cur = 0;
127         for (char c : s) {
128             auto it = st[cur].next.find(c);
129             if (it == st[cur].next.end())
130                 return -1;
131             cur = it->second;
132         }
133         return cur;
134     }
135 };

```

```

136 /// OUTPUT ALL OCCURRENCES
137 /// To output all occurrences build the inverse_link adjacency list
138 ///
139 /// for (int i = 1; i < st.size(); ++i)
140 ///     inverse_link[st[i].link].emplace_back(i);
141 ///
142 /// Then take all occurrences from state cur (where the substring ends)
143 ///
144 /// void output_all_occurrences(int cur, int pat_length) {
145 ///     occ.emplace_back(st[cur].first_pos - pat_length + 1);
146 ///     for (const int u : inverse_link[cur])
147 ///         output_all_occurrences(u, pat_length);
148 /// }
149 ///
150 /// Take care and remove all duplicates after that
151 ///
152 /// sort(occ.begin(), occ.end())
153 /// occ.resize(unique(occ.begin(), occ.end()) - occ.begin())
154 ///
155 /// LAST POSITION
156 /// Since the link of the current state represents the longest suffix of the
157 /// current class which is not present in it, we can calculate the last
158 /// position of the current class using dp.
159 ///
160 /// vector<int> dp_last_pos;
161 /// vector<vector<int>> inverse_link;
162 /// void pre_compute() {
163 ///     dp_last_pos.resize(st.size(), -1);
164 ///     inverse_link.resize(st.size());
165 ///     for (int i = 1; i < st.size(); ++i)
166 ///         inverse_link[st[i].link].emplace_back(i);
167 /// }
168 ///
169 /// int last_pos(const int u) {
170 ///     int &ret = dp_last_pos[u];
171 ///     if (~ret)
172 ///         return ret;
173 ///     ret = st[u].first_pos;
174 ///     for (const int v : inverse_link[u])
175 ///         ret = max(ret, last_pos(v));
176 ///     return ret;
177 /// }
178

```

10.10. 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 };

```

```

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

```

10.11. 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:

```