

C++ Competitive Programming Library

DO NOT DISCLOSE OR DISTRIBUTE

bfs.07 - Bernardo Flores Salmeron

| | | | | | |
|----------|---|-----------|----------|--|-----------|
| 1 | Template | 3 | 4 | Geometry | 21 |
| 2 | Data Structures | 3 | | | |
| 2.1 | Bit2D | 3 | 4.1 | Centro De Massa De Um Poligono | 21 |
| 2.2 | Merge Sort Tree (K-Esimo Maior Elemento Num Intervalo, Valores Maiores Que K Num Intervalo, | 4 | 4.2 | Closest Pair Of Points | 21 |
| 2.3 | Mos Algorithm | 4 | 4.3 | Condicao De Existencia De Um Triangulo | 22 |
| 2.4 | Sqrt Decomposition | 5 | 4.4 | Convex Hull | 22 |
| 2.5 | Bit | 5 | 4.5 | Cross Product | 22 |
| 2.6 | Bit (Range Update) | 6 | 4.6 | Distance Point Segment | 22 |
| 2.7 | Counting Inversions (Minimum Number Of Adjacent Swaps To Sort Array) | 6 | 4.7 | Line-Line Intersection | 22 |
| 2.8 | Ordered Set | 6 | 4.8 | Line-Point Distance | 22 |
| 2.9 | Persistent Segment Tree | 7 | 4.9 | Point Inside Convex Polygon - Log(N) | 23 |
| 2.10 | Segment Tree | 8 | 4.10 | Point Inside Polygon | 24 |
| 2.11 | Segment Tree 2D | 9 | 4.11 | Points Inside And In Boundary Polygon | 25 |
| 2.12 | Segment Tree Beats | 10 | 4.12 | Polygon Area (3D) | 25 |
| 2.13 | Segment Tree Polynomial | 13 | 4.13 | Polygon Area | 25 |
| 2.14 | Sparse Table | 14 | 4.14 | Segment-Segment Intersection | 25 |
| 2.15 | Treap | 14 | 4.15 | Upper And Lower Hull | 26 |
| 3 | Dp | 18 | 4.16 | Circle Circle Intersection | 26 |
| 3.1 | Achar Maior Palindromo | 18 | 4.17 | Circle Circle Intersection | 27 |
| 3.2 | Digit Dp | 18 | 4.18 | Struct Point And Line | 27 |
| 3.3 | Longest Common Subsequence | 18 | 5 | Graphs | 28 |
| 3.4 | Longest Common Substring | 18 | 5.1 | All Eulerian Path Or Tour | 28 |
| 3.5 | Longest Increasing Subsequence 2D (Not Sorted) | 19 | 5.2 | Articulation Points | 29 |
| 3.6 | Longest Increasing Subsequence 2D (Sorted) | 19 | 5.3 | Bellman Ford | 30 |
| 3.7 | Subset Sum Com Bitset | 19 | 5.4 | Bipartite Check | 31 |
| 3.8 | Catalan | 20 | 5.5 | Block Cut Tree | 31 |
| 3.9 | Catalan | 20 | 5.6 | Bridges | 32 |
| 3.10 | Coin Change Problem | 20 | 5.7 | Centroid | 32 |
| 3.11 | Edit Distance | 20 | 5.8 | Centroid Decomposition | 32 |
| 3.12 | Knapsack | 20 | 5.9 | Cycle Detection | 33 |
| 3.13 | Lis | 21 | 5.10 | De Bruijn Sequence | 33 |
| | | | 5.11 | Diameter In Tree | 34 |
| | | | 5.12 | Dijkstra + Dij Graph | 34 |

| | | | | | |
|-------------------------|---|-----------|------------------------|--|-----------|
| 5.13 | Dinic | 35 | 6.14 | Rotate (Right) | 50 |
| 5.14 | Dsu | 38 | 6.15 | Scanf De Uma String | 50 |
| 5.15 | Dsu On Tree | 38 | 6.16 | Split Function | 50 |
| 5.16 | Floyd Warshall | 39 | 6.17 | String To Long Long | 51 |
| 5.17 | Functional Graph | 39 | 6.18 | Substring | 51 |
| 5.18 | Girth (Shortest Cycle In A Graph) | 41 | 6.19 | Width | 51 |
| 5.19 | Hld | 41 | 6.20 | Binary String To Int | 51 |
| 5.20 | Hungarian | 42 | 6.21 | Check | 51 |
| 5.21 | Kuhn | 43 | 6.22 | Check Overflow | 51 |
| 5.22 | Lca | 43 | 6.23 | Counting Bits | 51 |
| 5.23 | Maximum Independent Set (Set Of Vertices That Arent Directly Connected) | 45 | 6.24 | Print Int128 T | 51 |
| 5.24 | Maximum Path Unweighted Graph | 45 | 6.25 | Random Numbers | 52 |
| 5.25 | Min Cost Flow | 45 | 6.26 | Readint | 52 |
| 5.26 | Minimum Edge Cover (Set Of Edges That Are Adjacent To All Vertices) | 46 | 6.27 | Time Measure | 52 |
| 5.27 | Minimum Path Cover In Dag | 46 | 7 Math | | 52 |
| 5.28 | Minimum Path Cover In Dag | 46 | 7.1 | Bell Numbers | 52 |
| 5.29 | Mst | 46 | 7.2 | Binary Exponentiation | 52 |
| 5.30 | Number Of Different Spanning Trees In A Complete Graph | 47 | 7.3 | Chinese Remainder Theorem | 52 |
| 5.31 | Number Of Ways To Make A Graph Connected | 47 | 7.4 | Combinatorics | 53 |
| 5.32 | Pruffer Decode | 47 | 7.5 | Diophantine Equation | 53 |
| 5.33 | Pruffer Encode | 47 | 7.6 | Divisors | 53 |
| 5.34 | Pruffer Properties | 47 | 7.7 | Euler Totient | 54 |
| 5.35 | Remove All Bridges From Graph | 48 | 7.8 | Extended Euclidean | 54 |
| 5.36 | Scc (Kosaraju) | 48 | 7.9 | Factorization | 54 |
| 5.37 | Topological Sort | 48 | 7.10 | Inclusion Exclusion | 54 |
| 5.38 | Tree Diameter | 48 | 7.11 | Inclusion Exclusion | 54 |
| 5.39 | Tree Distance | 49 | 7.12 | Markov Chains | 55 |
| 5.40 | Tree Isomorphism | 49 | 7.13 | Matrix Exponentiation | 55 |
| 6 Language Stuff | | 50 | 7.14 | Matrix Exponentiation | 55 |
| 6.1 | Climits | 50 | 7.15 | Pollard Rho (Find A Divisor) | 55 |
| 6.2 | Checagem E Transformacao De Caractere | 50 | 7.16 | Polynomial Convolution | 56 |
| 6.3 | Conta Digitos 1 Ate N | 50 | 7.17 | Primality Check | 56 |
| 6.4 | Escrita Em Arquivo | 50 | 7.18 | Primes | 56 |
| 6.5 | Gcd | 50 | 7.19 | Sieve + Segmented Sieve | 56 |
| 6.6 | Hipotenusa | 50 | 7.20 | Stars And Bars | 57 |
| 6.7 | Int To Binary String | 50 | 8 Miscellaneous | | 57 |
| 6.8 | Int To String | 50 | 8.1 | 2-Sat | 57 |
| 6.9 | Leitura De Arquivo | 50 | 8.2 | Interval Scheduling | 58 |
| 6.10 | Max E Min Element Num Vektor | 50 | 8.3 | Interval Scheduling | 58 |
| 6.11 | Permutacao | 50 | 8.4 | Oito Rainhas | 58 |
| 6.12 | Remove Repeticoes Continuas Num Vektor | 50 | 8.5 | Sliding Window Minimum | 58 |
| 6.13 | Rotate (Left) | 50 | 8.6 | Torre De Hanoi | 58 |

| | | |
|-----------|--|-----------|
| 8.7 | Counting Frequency Of Digits From 1 To K | 59 |
| 8.8 | Infix To Postfix | 59 |
| 8.9 | Kadane | 59 |
| 8.10 | Kadane (Segment Tree) | 59 |
| 8.11 | Kadane 2D | 60 |
| 8.12 | Largest Area In Histogram | 61 |
| 8.13 | Modular Integer | 61 |
| 8.14 | Point Compression | 62 |
| 8.15 | Ternary Search | 62 |
| 9 | Stress Testing | 62 |
| 9.1 | Check | 62 |
| 9.2 | Gen | 62 |
| 9.3 | Run | 64 |
| 10 | Strings | 64 |
| 10.1 | Trie - Maximum Xor Sum | 64 |
| 10.2 | Trie - Maximum Xor Two Elements | 64 |
| 10.3 | Z-Function | 64 |
| 10.4 | Aho Corasick | 64 |
| 10.5 | Hashing | 65 |
| 10.6 | Kmp | 66 |
| 10.7 | Lcs K Strings | 66 |
| 10.8 | Lexicographically Smallest Rotation | 67 |
| 10.9 | Manacher (Longest Palindrome) | 67 |
| 10.10 | Suffix Array | 68 |
| 10.11 | Suffix Array Pessoa | 70 |
| 10.12 | Trie | 70 |

1. Template

```

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

2. Data Structures

2.1. Bit2D

```

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

```

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

```

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

```

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

```

```

15     }
16
17     int mid = (l+r)>>1;
18     return query(l, mid, i, j, k, 2*pos+1) + query(mid+1, r, i, j, k, 2*pos+2);
19 }
20
21 void build(int l, int r, int pos) {
22
23     if(l == r) {
24         tree[pos].vet.pb(arr[l]);
25         return;
26     }
27
28     int mid = (l+r)>>1;
29     build(l, mid, 2*pos+1);
30     build(mid + 1, r, 2*pos+2);
31
32     merge(tree[2*pos+1].vet.begin(), tree[2*pos+1].vet.end(),
33           tree[2*pos+2].vet.begin(), tree[2*pos+2].vet.end(),
34           back_inserter(tree[pos].vet));
35 }

```

2.3. Mos Algorithm

```

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

```

```

41     query[i].ind = i;
42 }
43 sort(query, query + m, cmp);
44
45 int s,e;
46 s = e = query[0].l;
47 add(s);
48 for(int i = 0; i < m; i++) {
49     while(s > query[i].l)
50         add(--s);
51     while(s < query[i].l)
52         del(s++);
53     while(e < query[i].r)
54         add(++e);
55     while(e > query[i].r)
56         del(e--);
57     ans[query[i].ind] = cont;
58 }
59
60 for(int i = 0; i < m; i++)
61     cout << ans[i] << endl;
62 }

```

2.4. Sqrt Decomposition

```

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

```

2.5. Bit

```

1 /// INDEX THE ARRAY BY 1!!!
2 class BIT {
3 private:
4     vector<int> bit;
5     int n;
6
7 private:
8     int low(const int i) { return i & (-i); }
9
10    // Point update
11    void bit_update(int i, const int delta) {
12        while (i <= n) {
13            bit[i] += delta;

```

```

14        i += low(i);
15    }
16 }
17
18 // Prefix query
19 int bit_query(int i) {
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     bit.resize(arr.size(), 0);
35
36     for (int i = 1; i <= n; i++)
37         bit_update(i, arr[i]);
38 }
39
40 public:
41     /// Constructor responsible for initializing the tree with 0's.
42     ///
43     /// Time Complexity: O(n log n)
44     BIT(const vector<int> &arr) { build(arr); }
45
46     /// Constructor responsible for building the tree based on a vector.
47     ///
48     /// Time Complexity O(n)
49     BIT(const int n) {
50         // OBS: BIT IS INDEXED FROM 1
51         // THE USAGE OF 1-BASED ARRAY IS MANDATORY
52         this->n = n;
53         bit.resize(n + 1, 0);
54     }
55
56     /// Update at a single index.
57     ///
58     /// Time Complexity O(log n)
59     void update(const int i, const int delta) {
60         assert(1 <= i), assert(i <= n);
61         bit_update(i, delta);
62     }
63
64     /// Prefix query from 1 to i.
65     ///
66     /// Time Complexity O(log n)
67     int prefix_query(const int i) {
68         assert(1 <= i), assert(i <= n);
69         return bit_query(i);
70     }
71
72     /// Query at a single index.
73     ///
74     /// Time Complexity O(log n)
75     int query(const int idx) {
76         assert(1 <= idx), assert(idx <= this->n);
77         return bit_query(idx) - bit_query(idx - 1);
78     }

```

```

79
80 /// Range query from l to r.
81 ///
82 /// Time Complexity O(log n)
83 int query(const int l, const int r) {
84     assert(l <= 1), assert(l <= r), assert(r <= n);
85     return bit_query(r) - bit_query(l - 1);
86 }
87 };

```

2.6. Bit (Range Update)

```

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

```

```

53     this->n = n;
54     bit1.resize(n + 1, 0);
55     bit2.resize(n + 1, 0);
56 }
57
58 /// Range update from l to r.
59 ///
60 /// Time Complexity O(log n)
61 void update(const int l, const int r, const int delta) {
62     assert(l <= 1), assert(l <= r), assert(r <= n);
63     update(l, delta, bit1);
64     update(r + 1, -delta, bit1);
65     update(l, delta * (1 - 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 <= 1), assert(l <= r), assert(r <= n);
82     return query(r) - query(l - 1);
83 }
84
85 /// Prefix query from 1 to i.
86 ///
87 /// Time Complexity O(log n)
88 int query(const int i) {
89     assert(i <= n);
90     return (query(i, bit1) * i) - query(i, bit2);
91 }
92 };

```

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

```

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

```

2.8. Ordered Set

```

1  #include <bits/stdc++.h>
2  #include <ext/pb_ds/assoc_container.hpp>
3  #include <ext/pb_ds/trie_policy.hpp>
4

```

```

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

```

2.9. Persistent Segment Tree

```

1 class Persistent_Seg_Tree {
2     struct Node {
3         int val;
4         Node *left, *right;
5         Node(const int v) : val(v), left(nullptr), right(nullptr) {}
6     };
7
8     private:
9         const Node NEUTRAL_NODE = Node(0);
10        int merge_nodes(const int x, const int y) { return x + y; }
11
12    private:
13        const int n;
14        vector<Node*> version = {nullptr};
15
16    public:
17        /// Builds version[0] with the values in the array.
18        ///
19        /// Time complexity: O(n)
20        Node *build(Node *node, const int l, const int r, const vector<int> &arr) {
21            node = new Node(NEUTRAL_NODE);
22            if (l == r) {
23                node->val = arr[l];
24                return node;
25            }
26
27            const int mid = (l + r) / 2;
28            node->left = build(node->left, l, mid, arr);
29            node->right = build(node->right, mid + 1, r, arr);
30            node->val = merge_nodes(node->left->val, node->right->val);
31            return node;
32        }
33

```

```

34 Node *_update(Node *cur_tree, Node *prev_tree, const int l, const int r,
35              const int idx, const int delta) {
36     if (l > idx || r < idx)
37         return cur_tree != nullptr ? cur_tree : prev_tree;
38
39     if (cur_tree == nullptr && prev_tree == nullptr)
40         cur_tree = new Node(NEUTRAL_NODE);
41     else
42         cur_tree = new Node(cur_tree == nullptr ? *prev_tree : *cur_tree);
43
44     if (l == r) {
45         cur_tree->val += delta;
46         return cur_tree;
47     }
48
49     const int mid = (l + r) / 2;
50     cur_tree->left =
51         _update(cur_tree->left, prev_tree ? prev_tree->left : nullptr, l,
52               mid, 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) {}
83
84 /// Constructor that initializes the segment tree empty. It's allowed to
85 /// query
86 /// from 0 to MAXN - 1.
87 ///
88 /// Time Complexity: O(1)
89 Persistent_Seg_Tree(const int MAXN) : n(MAXN) {}
90
91 /// Constructor that allows to pass initial values to the leafs. It's
92 /// allowed
93 /// to query from 0 to n - 1.
94 ///
95 /// Time Complexity: O(n)
96 Persistent_Seg_Tree(const vector<int> &arr) : n(arr.size()) {
97     this->version[0] = this->build(this->version[0], 0, this->n - 1, arr);
98 }

```

```

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

```

2.10. Segment Tree

```

1 class Seg_Tree {
2 public:
3     struct Node {
4         int val, lazy;
5
6         Node() {}
7         Node(const int val) : val(val), lazy(0) {}
8     };
9
10 private:
11     /// // Range Sum
12     /// Node NEUTRAL_NODE = Node(0);
13     /// Node merge_nodes(const Node &x, const Node &y) {
14     ///     return Node(x.val + y.val);
15     /// };
16     ///
17     /// void apply_lazy(const int l, const int r, const int pos) {
18     ///     // for set change this to =
19     ///     tree[pos].val += (r - l + 1) * tree[pos].lazy;
20     /// }
21
22     /// // RMQ Max
23     /// Node NEUTRAL_NODE = Node(-INF);
24     /// Node merge_nodes(const Node &x, const Node &y) {
25     ///     return Node(max(x.val, y.val));
26     /// }

```

```

27 // void apply_lazy(const int l, const int r, const int pos) {
28 //     tree[pos].val += tree[pos].lazy;
29 // }
30
31 // // RMQ Min
32 // Node NEUTRAL_NODE = Node(INF);
33 // Node merge_nodes(const Node &x, const Node &y) {
34 //     return Node(min(x.val, y.val));
35 // }
36 // void apply_lazy(const int l, const int r, const int pos) {
37 //     tree[pos].val += tree[pos].lazy;
38 // }
39
40 // // XOR
41 // // Only works with point updates
42 // Node NEUTRAL_NODE = Node(0);
43 // Node merge_nodes(const Node &x, const Node &y) {
44 //     return Node(x.val ^ y.val);
45 // };
46 // }
47 // void apply_lazy(const int l, const int r, const int pos) {}
48
49 private:
50     int n;
51
52 public:
53     vector<Node> tree;
54
55 private:
56     void propagate(const int l, const int r, const int pos) {
57         if (tree[pos].lazy != 0) {
58             apply_lazy(l, r, pos);
59             if (l != r) {
60                 // for set change this to =
61                 tree[2 * pos + 1].lazy += tree[pos].lazy;
62                 tree[2 * pos + 2].lazy += tree[pos].lazy;
63             }
64             tree[pos].lazy = 0;
65         }
66     }
67
68     Node _build(const int l, const int r, const vector<int> &arr, const int
69                pos) {
70         if (l == r)
71             return tree[pos] = Node(arr[l]);
72
73         int mid = (l + r) / 2;
74         return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
75                                         _build(mid + 1, r, arr, 2 * pos + 2));
76     }
77
78     int _get_first(const int l, const int r, const int i, const int j,
79                   const int v, const int pos) {
80         propagate(l, r, pos);
81
82         if (l > r || l > j || r < i)
83             return -1;
84         // Needs RMQ MAX
85         // Replace to <= for greater or (with RMQ MIN) > for smaller or
86         // equal or >= for smaller
87         if (tree[pos].val < v)
88             return -1;
89
90         if (l == r)
91             return l;

```



```

91     int mid = (l + r) / 2;
92     int aux = _get_first(l, mid, i, j, v, 2 * pos + 1);
93     if (aux != -1)
94         return aux;
95     return _get_first(mid + 1, r, i, j, v, 2 * pos + 2);
96 }
97
98 Node _query(const int l, const int r, const int i, const int j,
99             const int pos) {
100     propagate(l, r, pos);
101
102     if (l > r || l > j || r < i)
103         return NEUTRAL_NODE;
104
105     if (i <= l && r <= j)
106         return tree[pos];
107
108     int mid = (l + r) / 2;
109     return merge_nodes(_query(l, mid, i, j, 2 * pos + 1),
110                       _query(mid + 1, r, i, j, 2 * pos + 2));
111 }
112
113 // It adds a number delta to the range from i to j
114 Node _update(const int l, const int r, const int i, const int j,
115             const int delta, const int pos) {
116     propagate(l, r, pos);
117
118     if (l > r || l > j || r < i)
119         return tree[pos];
120
121     if (i <= l && r <= j) {
122         tree[pos].lazy = delta;
123         propagate(l, r, pos);
124         return tree[pos];
125     }
126
127     int mid = (l + r) / 2;
128     return tree[pos] =
129         merge_nodes(_update(l, mid, i, j, delta, 2 * pos + 1),
130                   _update(mid + 1, r, i, j, delta, 2 * pos + 2));
131 }
132
133 void build(const vector<int> &arr) {
134     this->tree.resize(4 * this->n);
135     this->_build(0, this->n - 1, arr, 0);
136 }
137
138 public:
139     /// N equals to -1 means the Segment Tree hasn't been created yet.
140     Seg_Tree() : n(-1) {}
141
142     /// Constructor responsible for initializing the tree with val.
143     ///
144     /// Time Complexity O(n)
145     Seg_Tree(const int n, const int val = 0) : n(n) {
146         this->tree.resize(4 * this->n, Node(val));
147     }
148
149     /// Constructor responsible for building the tree based on a vector.
150     ///
151     /// Time Complexity O(n)
152     Seg_Tree(const vector<int> &arr) : n(arr.size()) { this->build(arr); }
153
154     /// Returns the first index from i to j compared to v.
155

```

```

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 };

```

2.11. Segment Tree 2D

```

1 // REQUIRES segment_tree.cpp!!
2 class Seg_Tree_2d {
3 private:
4     /// // range sum
5     /// int NEUTRAL_VALUE = 0;
6     /// int merge_nodes(const int &x, const int &y) {
7     ///     return x + y;
8     /// }
9
10    /// // RMQ max
11    /// int NEUTRAL_VALUE = -INF;
12    /// int merge_nodes(const int &x, const int &y) {
13    ///     return max(x, y);
14    /// }
15

```

```

16 // // RMQ min
17 // int NEUTRAL_VALUE = INF;
18 // int merge_nodes(const int &x, const int &y) {
19 //     return min(x, y);
20 // }
21
22 private:
23     int n, m;
24
25 public:
26     vector<Seg_Tree> tree;
27
28 private:
29     void st_build(const int l, const int r, const int pos, const
        vector<vector<int>> &mat) {
30         if(l == r)
31             tree[pos] = Seg_Tree(mat[l]);
32         else {
33             int mid = (l + r) / 2;
34             st_build(l, mid, 2*pos + 1, mat);
35             st_build(mid + 1, r, 2*pos + 2, mat);
36             for(int i = 0; i < tree[2*pos + 1].tree.size(); i++)
37                 tree[pos].tree[i].val = merge_nodes(tree[2*pos + 1].tree[i].val,
38                                                         tree[2*pos + 2].tree[i].val);
39         }
40     }
41
42     int st_query(const int l, const int r, const int x1, const int y1, const
        int x2, const int y2, const int pos) {
43         if(l > x2 || r < x1)
44             return NEUTRAL_VALUE;
45
46         if(x1 <= l && r <= x2)
47             return tree[pos].query(y1, y2);
48
49         int mid = (l + r) / 2;
50         return merge_nodes(st_query(l, mid, x1, y1, x2, y2, 2*pos + 1),
51                             st_query(mid + 1, r, x1, y1, x2, y2, 2*pos + 2));
52     }
53
54     void st_update(const int l, const int r, const int x, const int y, const
        int delta, const int pos) {
55         if(l > x || r < x)
56             return;
57
58         // Only supports point updates.
59         if(l == r) {
60             tree[pos].update(y, delta);
61             return;
62         }
63
64         int mid = (l + r) / 2;
65         st_update(l, mid, x, y, delta, 2*pos + 1);
66         st_update(mid + 1, r, x, y, delta, 2*pos + 2);
67         tree[pos].update(y, delta);
68     }
69
70 public:
71     Seg_Tree_2d() {
72         this->n = -1;
73         this->m = -1;
74     }
75
76     Seg_Tree_2d(const int n, const int m) {
77         this->n = n;

```

```

78         this->m = m;
79         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
80         assert(m < 10000);
81         tree.resize(4 * n, Seg_Tree(m));
82     }
83
84     Seg_Tree_2d(const int n, const int m, const vector<vector<int>> &mat) {
85         this->n = n;
86         this->m = m;
87         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
88         assert(m < 10000);
89         tree.resize(4 * n, Seg_Tree(m));
90         st_build(0, n - 1, 0, mat);
91     }
92
93     // Query from (x1, y1) to (x2, y2).
94     //
95     // Time complexity: O((log n) * (log m))
96     int query(const int x1, const int y1, const int x2, const int y2) {
97         assert(this->n > -1);
98         assert(0 <= x1); assert(x1 <= x2); assert(x2 < this->n);
99         assert(0 <= y1); assert(y1 <= y2); assert(y2 < this->n);
100         return st_query(0, this->n - 1, x1, y1, x2, y2, 0);
101     }
102
103     // Point updates on position (x, y).
104     //
105     // Time complexity: O((log n) * (log m))
106     void update(const int x, const int y, const int delta) {
107         assert(0 <= x); assert(x < this->n);
108         assert(0 <= y); assert(y < this->n);
109         st_update(0, this->n - 1, x, y, delta, 0);
110     }
111 };

```

2.12. Segment Tree 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
148         // max update
149         if (tree[c1].minn < node.minn)
150             apply_update_max(c1, node.minn);
151         if (tree[c2].minn < node.minn)
152             apply_update_max(c2, node.minn);
153         #endif
154     }
155
156     Node _build(const int l, const int r, const vector<int> &arr, const int
157         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,
const int v, const int pos) {
181     propagate(l, r, pos);
182
183     if (l > r || l > j || r < i)
184         return tree[pos];
185
186     if (i <= l && r <= j) {
187         apply_update_sum(l, r, pos, v);
188         return tree[pos];
189     }
190
191     int mid = (l + r) / 2;
192     return tree[pos] =
193         merge_nodes(_update_sum(l, mid, i, j, v, 2 * pos + 1),
194                   _update_sum(mid + 1, r, i, j, v, 2 * pos + 2));
195 }
196 #endif
197
198 #ifdef MIN_UPDATE
199 Node _update_min(const int l, const int r, const int i, const int j,
const int x, const int pos) {
200     propagate(l, r, pos);
201
202     if (l > r || l > j || r < i || tree[pos].maxx <= x)
203         return tree[pos];
204
205     if (i <= l && r <= j && tree[pos].smaxx < x) {
206         apply_update_min(pos, x);
207         return tree[pos];
208     }
209
210     const int mid = (l + r) / 2;
211     return tree[pos] =
212         merge_nodes(_update_min(l, mid, i, j, x, 2 * pos + 1),
213                   _update_min(mid + 1, r, i, j, x, 2 * pos + 2));
214 }
215 #endif
216
217 #ifdef MAX_UPDATE

```

```

220 Node _update_max(const int l, const int r, const int i, const int j,
const int x, const int pos) {
221     propagate(l, r, pos);
222
223     if (l > r || l > j || r < i || tree[pos].minn >= x)
224         return tree[pos];
225
226     if (i <= l && r <= j && tree[pos].sminn > x) {
227         apply_update_max(pos, x);
228         return tree[pos];
229     }
230
231     const int mid = (l + r) / 2;
232     return tree[pos] =
233         merge_nodes(_update_max(l, mid, i, j, x, 2 * pos + 1),
234                   _update_max(mid + 1, r, i, j, x, 2 * pos + 2));
235 }
236 #endif
237
238 void build(const vector<int> &arr) {
239     this->tree.resize(4 * this->n);
240     this->_build(0, this->n - 1, arr, 0);
241 }
242
243 public:
244     /// N equals to -1 means the Segment Tree hasn't been created yet.
245     Seg_Tree Beats() : n(-1) {}
246
247     /// Constructor responsible for initializing the tree with 0's.
248     ///
249     /// Time Complexity O(n)
250     Seg_Tree Beats(const int n) : n(n) {
251         this->tree.resize(4 * this->n, Node(0));
252     }
253
254     /// Constructor responsible for building the tree based on a vector.
255     ///
256     /// Time Complexity O(n)
257     Seg_Tree Beats(const vector<int> &arr) : n(arr.size()) { this->build(arr);
258     }
259
260     #ifdef ADD_UPDATE
261     /// Range update from l to r.
262     /// Type: for i in range [l, r] do a[i] += x
263     void update_sum(const int l, const int r, const int x) {
264         assert(this->n >= 0);
265         assert(0 <= l), assert(l <= r), assert(r < this->n);
266         this->_update_sum(0, this->n - 1, l, r, x, 0);
267     }
268     #endif
269
270     #ifdef MIN_UPDATE
271     /// Range update from l to r.
272     /// Type: for i in range [l, r] do a[i] = min(a[i], x)
273     void update_min(const int l, const int r, const int x) {
274         assert(this->n >= 0);
275         assert(0 <= l), assert(l <= r), assert(r < this->n);
276         this->_update_min(0, this->n - 1, l, r, x, 0);
277     }
278     #endif
279
280     #ifdef MAX_UPDATE
281     /// Range update from l to r.
282     /// Type: for i in range [l, r] do a[i] = max(a[i], x)
283     void update_max(const int l, const int r, const int x) {

```

```

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

```

2.13. 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 private:
23     int n;
24
25 public:
26     vector<Node> tree;
27
28 private:
29     void st_propagate(const int l, const int r, const int pos) {
30         if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {
31             apply_lazy(l, r, pos);
32             int mid = (l + r) / 2;
33             int sz_left = mid - l + 1;
34             if (l != r) {
35                 tree[2 * pos + 1].z0 += tree[pos].z0;
36                 tree[2 * pos + 1].z1 += tree[pos].z1;
37
38                 tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
39                 tree[2 * pos + 2].z1 += tree[pos].z1;
40             }
41             tree[pos].z0 = 0;
42             tree[pos].z1 = 0;
43         }
44     }
45
46     Node st_build(const int l, const int r, const vector<int> &arr,
47                  const int pos) {
48         if (l == r)
49             return tree[pos] = Node(arr[l], 0, 0);
50
51         int mid = (l + r) / 2;
52         return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
53                                       st_build(mid + 1, r, arr, 2 * pos + 2));
54     }
55
56     Node st_query(const int l, const int r, const int i, const int j,
57                  const int pos) {
58         st_propagate(l, r, pos);
59
60         if (l > r || l > j || r < i)
61             return NEUTRAL_NODE;
62
63         if (i <= l && r <= j)
64             return tree[pos];
65
66         int mid = (l + r) / 2;
67         return merge_nodes(st_query(l, mid, i, j, 2 * pos + 1),
68                           st_query(mid + 1, r, i, j, 2 * pos + 2));
69     }
70
71     // it adds a number delta to the range from i to j
72     Node st_update(const int l, const int r, const int i, const int j,
73                   const int z1, const int z0, const int pos) {
74         st_propagate(l, r, pos);
75
76         if (l > r || l > j || r < i)
77             return tree[pos];
78
79         if (i <= l && r <= j) {
80             tree[pos].z0 = (l - i + 1) * z0;
81             tree[pos].z1 = z1;
82             st_propagate(l, r, pos);
83             return tree[pos];
84         }
85     }
86

```

```

87     int mid = (l + r) / 2;
88     return tree[pos] =
89         merge_nodes(st_update(l, mid, i, j, z1, z0, 2 * pos + 1),
90                     st_update(mid + 1, r, i, j, z1, z0, 2 * pos + 2));
91 }
92
93 public:
94     Seg_Tree() : n(-1) {}
95
96     Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0, 0)); }
97
98     Seg_Tree(const vector<int> &arr) { this->build(arr); }
99
100 void build(const vector<int> &arr) {
101     this->n = arr.size();
102     this->tree.resize(4 * this->n);
103     this->st_build(0, this->n - 1, arr, 0);
104 }
105
106 /// Index update of a polynomial  $f(x) = z1 \cdot x + z0$ 
107 ///
108 /// Time Complexity  $O(\log n)$ 
109 void update(const int i, const int z1, const int z0) {
110     assert(this->n >= 0);
111     assert(0 <= i), assert(i < this->n);
112     this->st_update(0, this->n - 1, i, i, z1, z0, 0);
113 }
114
115 /// Range update of a polynomial  $f(x) = z1 \cdot x + z0$  from l to r
116 ///
117 /// Time Complexity  $O(\log n)$ 
118 void update(const int l, const int r, const int z1, const int z0) {
119     assert(this->n >= 0);
120     assert(0 <= l), assert(l <= r), assert(r < this->n);
121     this->st_update(0, this->n - 1, l, r, z1, z0, 0);
122 }
123
124 /// Range sum query from l to r
125 ///
126 /// Time Complexity  $O(\log n)$ 
127 int query(const int l, const int r) {
128     assert(this->n >= 0);
129     assert(0 <= l), assert(l <= r), assert(r < this->n);
130     return this->st_query(0, this->n - 1, l, r, 0).val;
131 }
132 };

```

2.14. Sparse Table

```

1 class Sparse_Table {
2 private:
3     /// Sparse table min
4     // int merge(const int l, const int r) { return min(l, r); }
5     /// Sparse table max
6     // int merge(const int l, const int r) { return max(l, r); }
7
8 private:
9     int n;
10    vector<vector<int>> table;
11    vector<int> lg;
12
13 private:
14    /// lg[i] represents the log2(i)

```

```

15 void build_log_array() {
16     lg.resize(this->n + 1);
17     for (int i = 2; i <= this->n; i++)
18         lg[i] = lg[i / 2] + 1;
19 }
20
21 /// Time Complexity:  $O(n \cdot \log(n))$ 
22 void build_sparse_table(const vector<int> &arr) {
23     table.resize(lg[this->n] + 1, vector<int>(this->n));
24
25     table[0] = arr;
26     int pow2 = 1;
27     for (int i = 1; i < table.size(); i++) {
28         const int lastsz = this->n - pow2 + 1;
29         for (int j = 0; j + pow2 < lastsz; j++)
30             table[i][j] = merge(table[i - 1][j], table[i - 1][j + pow2]);
31         pow2 <= 1;
32     }
33 }
34
35 public:
36     /// Constructor that builds the log array and the sparse table.
37     ///
38     /// Time Complexity:  $O(n \cdot \log(n))$ 
39     Sparse_Table(const vector<int> &arr) : n(arr.size()) {
40         this->build_log_array();
41         this->build_sparse_table(arr);
42     }
43
44 void print() {
45     int pow2 = 1;
46     for (int i = 0; i < table.size(); i++) {
47         const int sz = (int)(table.front().size()) - pow2 + 1;
48         for (int j = 0; j < sz; j++)
49             cout << table[i][j] << " \n"[(j + 1) == sz];
50         pow2 <= 1;
51     }
52 }
53
54 /// Range query from l to r.
55 ///
56 /// Time Complexity:  $O(1)$ 
57 int query(const int l, const int r) {
58     assert(0 <= l), assert(l <= r), assert(r < this->n);
59     int lgg = lg[r - l + 1];
60     return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
61 }
62 };

```

2.15. Treap

```

1 // clang-format off
2 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
3 // #define REVERSE
4 // #define LAZY
5 class Treap {
6 public:
7     struct Node {
8         Node *left = nullptr, *right = nullptr, *par = nullptr;
9         // Priority to be used in the treap
10        const int rank;
11        int size = 1, val;
12        // Contains the result of the range query between the node and its children.

```

```

13     int ans;
14     #ifdef LAZY
15     int lazy = 0;
16     #endif
17     #ifdef REVERSE
18     bool rev = false;
19     #endif
20
21     Node(const int val) : val(val), ans(val), rank(rng()) {}
22     Node(const int val, const int rank) : val(val), ans(val), rank(rank) {}
23 };
24
25 private:
26     vector<Node *> nodes;
27     int _size = 0;
28     Node *root = nullptr;
29
30 private:
31     // // Range Sum
32     // void merge_nodes(Node *node) {
33     //     node->ans = node->val;
34     //     if (node->left)
35     //         node->ans += node->left->ans;
36     //     if (node->right)
37     //         node->ans += node->right->ans;
38     // }
39
40     // #ifdef LAZY
41     // void apply_lazy(Node *node) {
42     //     node->val += node->lazy;
43     //     node->ans += node->lazy * get_size(node);
44     // }
45     // #endif
46
47     // // RMQ Min
48     // void merge_nodes(Node *node) {
49     //     node->ans = node->val;
50     //     if (node->left)
51     //         node->ans = min(node->ans, node->left->ans);
52     //     if (node->right)
53     //         node->ans = min(node->ans, node->right->ans);
54     // }
55
56     // #ifdef LAZY
57     // void apply_lazy(Node *node) {
58     //     node->val += node->lazy;
59     //     node->ans += node->lazy;
60     // }
61     // #endif
62
63     // // RMQ Max
64     // void merge_nodes(Node *node) {
65     //     node->ans = node->val;
66     //     if (node->left)
67     //         node->ans = max(node->ans, node->left->ans);
68     //     if (node->right)
69     //         node->ans = max(node->ans, node->right->ans);
70     // }
71
72     // #ifdef LAZY
73     // void apply_lazy(Node *node) {
74     //     node->val += node->lazy;
75     //     node->ans += node->lazy;
76     // }
77     // #endif

```

```

78
79     #ifdef REVERSE
80     void apply_reverse(Node *node) {
81         swap(node->left, node->right);
82         // write other operations here
83     }
84     #endif
85
86     int get_size(const Node *node) { return node ? node->size : 0; }
87
88     void update_size(Node *node) {
89         if (node)
90             node->size = 1 + get_size(node->left) + get_size(node->right);
91     }
92
93     void print(Node *node) {
94         if (!node)
95             return;
96         if (node->left) {
97             cerr << "left" << endl;
98             print(node->left);
99         }
100         cerr << node->val << endl;
101         cerr << endl;
102         if (node->right) {
103             cerr << "right" << endl;
104             print(node->right);
105         }
106     }
107
108     #ifdef REVERSE
109     void propagate_reverse(Node *node) {
110         if (node && node->rev) {
111             apply_reverse(node);
112             if (node->left)
113                 node->left->rev ^= 1;
114             if (node->right)
115                 node->right->rev ^= 1;
116             node->rev = 0;
117         }
118     }
119     #endif
120
121     #ifdef LAZY
122     void propagate_lazy(Node *node) {
123         if (node && node->lazy != 0) {
124             apply_lazy(node);
125             if (node->left)
126                 node->left->lazy += node->lazy;
127             if (node->right)
128                 node->right->lazy += node->lazy;
129             node->lazy = 0;
130         }
131     }
132     #endif
133
134     void update_node(Node *node) {
135         if (node) {
136             update_size(node);
137             #ifdef LAZY
138             propagate_lazy(node->left);
139             propagate_lazy(node->right);
140             #endif
141             #ifdef REVERSE
142             propagate_reverse(node->left);

```

```

143     propagate_reverse(node->right);
144     #endif
145     merge_nodes(node);
146 }
147 }
148
149 /// Splits the treap into to different treaps that contains nodes with
150 indexes
151 /// <= pos ans indexes > pos. The nodes l and r contains, in the end, these
152 /// two different treaps.
153 void split(Node *node, Node *&l, Node *&r, const int pos, Node *pl =
154     nullptr, Node *pr = nullptr) {
155     if (!node)
156         l = r = nullptr;
157     else {
158         #ifdef LAZY
159             propagate_lazy(node);
160         #endif
161         #ifdef REVERSE
162             propagate_reverse(node);
163         #endif
164         if (get_size(node->left) <= pos) {
165             node->par = pr;
166             split(node->right, node->right, r, pos - get_size(node->left) - 1,
167                 pl, node);
168             l = node;
169         } else {
170             node->par = pl;
171             split(node->left, l, node->left, pos, node, pr);
172             r = node;
173         }
174     }
175     update_node(node);
176 }
177
178 /// Merges to treaps (l and r) into a single one based on the rank of each
179 /// node.
180 void merge(Node *&node, Node *l, Node *r, Node *par = nullptr) {
181     #ifdef LAZY
182         propagate_lazy(l), propagate_lazy(r);
183     #endif
184     #ifdef REVERSE
185         propagate_reverse(l), propagate_reverse(r);
186     #endif
187     if (l == nullptr || r == nullptr)
188         node = (l == nullptr ? r : l);
189     else if (l->rank > r->rank) {
190         merge(l->right, l->right, r, l);
191         node = l;
192     } else {
193         merge(r->left, l, r->left, r);
194         node = r;
195     }
196     if (node)
197         node->par = par;
198     update_node(node);
199 }
200
201 Node *build(const int l, const int r, const vector<int> &arr,
202     vector<int> &rand) {
203     if (l > r)
204         return nullptr;

```

```

205     const int mid = (l + r) / 2;
206     Node *node = new Node(arr[mid], rand.back());
207     rand.pop_back();
208     node->right = build(mid + 1, r, arr, rand);
209     node->left = build(l, mid - 1, arr, rand);
210     update_node(node);
211
212     return node;
213 }
214
215 int _get_ith(const int idx) {
216     int ans = 0;
217     Node *cur = nodes[idx], *prev = nullptr;
218     while (cur) {
219         if (cur == nodes[idx] || prev == cur->right)
220             ans += 1 + get_size(cur->left);
221         prev = cur;
222         cur = cur->par;
223     }
224     return ans - 1;
225 }
226
227 vector<int> gen_rand(const int n) {
228     vector<int> ans(n);
229     for (int &x : ans)
230         x = rng();
231     sort(ans.begin(), ans.end());
232     return ans;
233 }
234
235 Node *_query(const int l, const int r) {
236     Node *L, *M, *R;
237     split(this->root, L, M, l - 1);
238     split(M, M, R, r - 1);
239     Node *ret = new Node(*M);
240     merge(L, L, M);
241     merge(root, L, R);
242     return ret;
243 }
244
245 void _update(const int l, const int r, const int delta) {
246     Node *L, *M, *R;
247     split(this->root, L, M, l - 1);
248     split(M, M, R, r - 1);
249
250     Node *node = M;
251     #ifdef LAZY
252         node->lazy = delta;
253         propagate_lazy(node);
254     #else
255         node->val += delta;
256     #endif
257
258     merge(L, L, M);
259     merge(root, L, R);
260 }
261
262 void _insert(const int pos, Node *node) {
263     this->_size += node->size;
264     Node *L, *R;
265     split(this->root, L, R, pos - 1);
266     merge(L, L, node);
267     merge(this->root, L, R);
268 }
269

```



```

270 Node *_erase(const int l, const int r) {
271     Node *L, *M, *R;
272     split(this->root, L, M, l - 1);
273     split(M, M, R, r - l);
274     merge(root, L, R);
275     this->_size -= r - l + 1;
276     return M;
277 }
278
279 void _move(const int l, const int r, const int new_pos) {
280     Node *node = _erase(l, r);
281     _insert(new_pos, node);
282 }
283
284 #ifdef REVERSE
285 void _reverse(const int l, const int r) {
286     Node *L, *M, *R;
287     split(this->root, L, M, l - 1);
288     split(M, M, R, r - l);
289
290     Node *node = M;
291     node->rev ^= true;
292
293     merge(L, L, M);
294     merge(root, L, R);
295 }
296 #endif
297
298 public:
299     Treap() {}
300
301     /// Constructor that initializes the treap based on an array.
302     ///
303     /// Time Complexity: O(n)
304     Treap(const vector<int> &arr) : _size(arr.size()) {
305         vector<int> r = gen_rand(arr.size());
306         this->root = build(0, (int)arr.size() - 1, arr, r);
307     }
308
309     int size() { return _size; }
310
311     /// Moves the subarray [l, r] to the position starting at new_pos.
312     /// new_pos represents the position BEFORE the subarray is deleted!!!
313     ///
314     /// Time Complexity: O(log n)
315     void move(const int l, const int r, int new_pos) {
316         assert(0 <= new_pos), assert(new_pos <= _size);
317         if(new_pos > l)
318             // after erase the index will be different if new_pos > l
319             new_pos -= r - l + 1;
320         _move(l, r, new_pos);
321     }
322
323     /// Moves the subarray [l, r] to the back of the array.
324     ///
325     /// Time Complexity: O(log n)
326     void move_back(const int l, const int r) {
327         assert(0 <= l), assert(l <= r), assert(r < _size);
328         move(l, r, _size);
329     }
330
331     /// Moves the subarray [l, r] to the front of the array.
332     ///
333     /// Time Complexity: O(log n)
334     void move_front(const int l, const int r) {

```

```

335         assert(0 <= l), assert(l <= r), assert(r < _size);
336         move(l, r, 0);
337     }
338
339     #ifndef REVERSE
340     /// Reverses the subarray [l, r].
341     ///
342     /// Time Complexity: O(log n)
343     void reverse(const int l, const int r) {
344         assert(0 <= l), assert(l <= r), assert(r < _size);
345         _reverse(l, r);
346     }
347     #endif
348
349     /// Erases the subarray [l, r].
350     ///
351     /// Time Complexity: O(log n)
352     void erase(const int l, const int r) {
353         assert(0 <= l), assert(l <= r), assert(r < _size);
354         _erase(l, r);
355     }
356
357     /// Inserts the value val at the position pos.
358     ///
359     /// Time Complexity: O(log n)
360     void insert(const int pos, const int val) {
361         assert(pos <= _size);
362         nodes.emplace_back(new Node(val));
363         _insert(pos, nodes.back());
364     }
365
366     /// Returns the index of the i-th added node.
367     ///
368     /// Time Complexity: O(log n)
369     int get_ith(const int idx) {
370         assert(0 <= idx), assert(idx < nodes.size());
371         return _get_ith(idx);
372     }
373
374     /// Sums the delta value to the position pos.
375     ///
376     /// Time Complexity: O(log n)
377     void update(const int pos, const int delta) {
378         assert(0 <= pos), assert(pos < _size);
379         _update(pos, pos, delta);
380     }
381
382     #ifndef LAZY
383     /// Sums the delta value to the subarray [l, r].
384     ///
385     /// Time Complexity: O(log n)
386     void update(const int l, const int r, const int delta) {
387         assert(0 <= l), assert(l <= r), assert(r < _size);
388         _update(l, r, delta);
389     }
390     #endif
391
392     /// Query at a single index.
393     ///
394     /// Time Complexity: O(log n)
395     int query(const int pos) {
396         assert(0 <= pos), assert(pos < _size);
397         return _query(pos, pos)->ans;
398     }
399

```

```

400 /// Range query from l to r.
401 ///
402 /// Time Complexity: O(log n)
403 int query(const int l, const int r) {
404     assert(0 <= l), assert(l <= r), assert(r < _size);
405     return _query(l, r)->ans;
406 }
407 };
408 // clang-format on

```

3. Dp

3.1. Achar Maior Palindromo

1 Fazer LCS da string com o reverso

3.2. Digit Dp

```

1 /// How many numbers x are there in the range a to b, where the digit d
  occurs exactly k times in x?
2 vector<int> num;
3 int a, b, d, k;
4 int DP[12][12][2];
5 /// DP[p][c][f] = Number of valid numbers <= b from this state
6 /// p = current position from left side (zero based)
7 /// c = number of times we have placed the digit d so far
8 /// f = the number we are building has already become smaller than b? [0 =
  no, 1 = yes]
9
10 int call(int pos, int cnt, int f){
11     if(cnt > k) return 0;
12
13     if(pos == num.size()){
14         if(cnt == k) return 1;
15         return 0;
16     }
17
18     if(DP[pos][cnt][f] != -1) return DP[pos][cnt][f];
19     int res = 0;
20     int lim = (f ? 9 : num[pos]);
21
22     /// Try to place all the valid digits such that the number doesn't exceed b
23     for(int dgt = 0; dgt <= LMT; dgt++){
24         int nf = f;
25         int ncnt = cnt;
26         if(f == 0 && dgt < LMT) nf = 1; /// The number is getting smaller at
  this position
27         if(dgt == d) ncnt++;
28         if(ncnt <= k) res += call(pos+1, ncnt, nf);
29     }
30
31     return DP[pos][cnt][f] = res;
32 }
33
34 int solve(int b){
35     num.clear();
36     while(b>0){
37         num.push_back(b%10);
38         b/=10;
39     }
40     reverse(num.begin(), num.end());
41     /// Stored all the digits of b in num for simplicity
42

```

```

43 memset(DP, -1, sizeof(DP));
44 int res = call(0, 0, 0);
45 return res;
46 }
47
48 int main () {
49
50     cin >> a >> b >> d >> k;
51     int res = solve(b) - solve(a-1);
52     cout << res << endl;
53
54     return 0;
55 }

```

3.3. Longest Common Subsequence

```

1 string lcs(string &s, string &t) {
2
3     int n = s.size(), m = t.size();
4
5     s.insert(s.begin(), '#');
6     t.insert(t.begin(), '$');
7
8     vector<vector<int>> mat(n + 1, vector<int>(m + 1, 0));
9
10    for(int i = 1; i <= n; i++) {
11        for(int j = 1; j <= m; j++) {
12            if(s[i] == t[j])
13                mat[i][j] = mat[i - 1][j - 1] + 1;
14            else
15                mat[i][j] = max(mat[i - 1][j], mat[i][j - 1]);
16        }
17    }
18
19    string ans;
20    int i = n, j = m;
21    while(i > 0 && j > 0) {
22        if(s[i] == t[j])
23            ans += s[i], i--, j--;
24        else if(mat[i][j - 1] > mat[i - 1][j])
25            j--;
26        else
27            i--;
28    }
29
30    reverse(ans.begin(), ans.end());
31    return ans;
32 }

```

3.4. Longest Common Substring

```

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

```

```

13     if (i == 0 || j == 0)
14         LCSuff[i][j] = 0;
15
16     else if (X[i-1] == Y[j-1]) {
17         LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
18         result = max(result, LCSuff[i][j]);
19     }
20     else LCSuff[i][j] = 0;
21 }
22 }
23 return result;
24 }

```

3.5. Longest Increasing Subsequence 2D (Not Sorted)

```

1 set<ii> s[(int)2e6];
2 bool check(ii par, int ind) {
3
4     auto it = s[ind].lower_bound(ii(par.ff, -INF));
5     if(it == s[ind].begin())
6         return false;
7
8     it--;
9
10    if(it->ss < par.ss)
11        return true;
12    return false;
13 }
14
15 int lis2d(vector<ii> &arr) {
16
17     int n = arr.size();
18     s[1].insert(arr[0]);
19
20     int maior = 1;
21     for(int i = 1; i < n; i++) {
22
23         ii x = arr[i];
24
25         int l = 1, r = maior;
26         int ansbb = 0;
27         while(l <= r) {
28             int mid = (l+r)/2;
29             if(check(x, mid)) {
30                 l = mid + 1;
31                 ansbb = mid;
32             } else {
33                 r = mid - 1;
34             }
35         }
36
37         // inserting in list
38         auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
39         while(it != s[ansbb+1].end() && it->ss >= x.ss)
40             it = s[ansbb+1].erase(it);
41
42         it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
43         if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
44            it->ss <= x.ss)
45             continue;
46         s[ansbb+1].insert(arr[i]);
47
48         maior = max(maior, ansbb + 1);
49     }
50 }

```

```

49
50     return maior;
51
52 }

```

3.6. Longest Increasing Subsequence 2D (Sorted)

```

1 set<ii> s[(int)2e6];
2 bool check(ii par, int ind) {
3
4     auto it = s[ind].lower_bound(ii(par.ff, -INF));
5     if(it == s[ind].begin())
6         return false;
7
8     it--;
9
10    if(it->ss < par.ss)
11        return true;
12    return false;
13 }
14
15 int lis2d(vector<ii> &arr) {
16
17     int n = arr.size();
18     s[1].insert(arr[0]);
19
20     int maior = 1;
21     for(int i = 1; i < n; i++) {
22
23         ii x = arr[i];
24
25         int l = 1, r = maior;
26         int ansbb = 0;
27         while(l <= r) {
28             int mid = (l+r)/2;
29             if(check(x, mid)) {
30                 l = mid + 1;
31                 ansbb = mid;
32             } else {
33                 r = mid - 1;
34             }
35         }
36
37         // inserting in list
38         auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
39         while(it != s[ansbb+1].end() && it->ss >= x.ss)
40             it = s[ansbb+1].erase(it);
41
42         it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
43         if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
44            it->ss <= x.ss)
45             continue;
46         s[ansbb+1].insert(arr[i]);
47
48         maior = max(maior, ansbb + 1);
49     }
50
51     return maior;
52 }

```

3.7. Subset Sum Com Bitset

```

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

```

3.8. Catalan

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

3.9. Catalan

```

1 // The first few Catalan numbers for n = 0, 1, 2, 3, ...
2 // are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
3 // Formula Recursiva:
4 // cat(0) = 0
5 // cat(n+1) = somatorio(i from 0 to n) (cat(i)*cat(n-i))
6 //
7 // Using Binomial Coefficient
8 // We can also use the below formula to find nth catalan number in O(n) time.
9 // Formula acima
10
11 // Returns value of Binomial Coefficient C(n, k)
12
13 int binomialCoeff(int n, int k) {
14     int res = 1;
15
16     // Since C(n, k) = C(n, n-k)
17     if (k > n - k)
18         k = n - k;
19
20     // Calculate value of [n*(n-1)*...*(n-k+1)] / [k*(k-1)*...*1]
21     for (int i = 0; i < k; ++i) {
22         res *= (n - i);
23         res /= (i + 1);
24     }
25
26     return res;
27 }
28 // A Binomial coefficient based function to find nth catalan
29 // number in O(n) time
30 int catalan(int n) {
31     // Calculate value of 2nCn
32     int c = binomialCoeff(2*n, n);
33
34     // return 2nCn/(n+1)
35     return c/(n+1);
36 }

```

3.10. Coin Change Problem

```

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

```

3.11. Edit Distance

```

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

```

3.12. Knapsack

```

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

```

```

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

```

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

```

4. Geometry

4.1. Centro De Massa De Um Poligono

```

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

```

```

13 c.y /= (3.0*area);
14
15 cout << c.x << ' ' << c.y << endl;

```

4.2. Closest Pair Of Points

```

1 struct Point {
2     int x, y;
3 };
4 int compareX(const void *a,const void *b){
5     Point *p1 = (Point *)a, *p2 = (Point *)b;
6     return (p1->x - p2->x);
7 }
8 int compareY(const void *a,const void *b) {
9     Point *p1 = (Point *)a,*p2 =(Point *)b;
10    return (p1->y - p2->y);
11 }
12 float dist(Point p1, Point p2) {
13     return sqrt((p1.x- p2.x)*(p1.x- p2.x) +(p1.y - p2.y)*(p1.y - p2.y));
14 }
15 float bruteForce(Point P[], int n){
16     float min = FLT_MAX;
17     for (int i = 0; i < n; ++i)
18         for (int j = i+1; j < n; ++j)
19             if (dist(P[i], P[j]) < min)
20                 min = dist(P[i], P[j]);
21     return min;
22 }
23 float min(float x, float y) {
24     return (x < y)? x : y;
25 }
26 float stripClosest(Point strip[], int size, float d) {
27     float min = d;
28     for (int i = 0; i < size; ++i)
29         for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
30             if (dist(strip[i],strip[j]) < min)
31                 min = dist(strip[i], strip[j]);
32     return min;
33 }
34 float closestUtil(Point Px[], Point Py[], int n){
35     if (n <= 3)
36         return bruteForce(Px, n);
37     int mid = n/2;
38     Point midPoint = Px[mid];
39     Point Pyl[mid+1];
40     Point Pyr[n-mid-1];
41     int li = 0, ri = 0;
42     for (int i = 0; i < n; i++)
43         if (Py[i].x <= midPoint.x)
44             Pyl[li++] = Py[i];
45         else
46             Pyr[ri++] = Py[i];
47
48     float dl = closestUtil(Px, Pyl, mid);
49     float dr = closestUtil(Px + mid, Pyr, n-mid);
50     float d = min(dl, dr);
51     Point strip[n];
52     int j = 0;
53     for (int i = 0; i < n; i++)
54         if (abs(Py[i].x - midPoint.x) < d)
55             strip[j] = Py[i], j++;
56     return min(d, stripClosest(strip, j, d));
57 }
58

```

```

59 float closest(Point P[], int n) {
60     Point Px[n];
61     Point Py[n];
62     for (int i = 0; i < n; i++) {
63         Px[i] = P[i];
64         Py[i] = P[i];
65     }
66     qsort(Px, n, sizeof(Point), compareX);
67     qsort(Py, n, sizeof(Point), compareY);
68     return closestUtil(Px, Py, n);
69 }

```

4.3. Condicao De Existencia De Um Triangulo

```

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

```

4.4. Convex Hull

```

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

```

```

36 H.resize(k);
37 /* o último ponto do vetor é igual ao primeiro, atente para isso
38    as vezes é necessário mudar */
39 return H;
40 }

```

4.5. Cross Product

```

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

```

4.6. Distance Point Segment

```

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

```

4.7. Line-Line Intersection

```

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

```

4.8. Line-Point Distance

```

1 double ptoReta(double x1, double y1, double x2, double y2, double pointX,
2   double pointY, double *ptox, double *ptoy) {
3   double diffX = x2 - x1;
4   double diffY = y2 - y1;
5   if ((diffX == 0) && (diffY == 0)) {
6     diffX = pointX - x1;
7     diffY = pointY - y1;
8     //se os dois sao pontos
9     return hypot(pointX - x1, pointY - y1);
10  }
11  double t = ((pointX - x1) * diffX + (pointY - y1) * diffY) /
12    (diffX * diffX + diffY * diffY);
13  if (t < 0) {
14    //point is nearest to the first point i.e x1 and y1
15    // Ex: .
16    // cord do pto na reta = pto inicial(x1,y1);
17    *ptox = x1, *ptoy = y1;
18    diffX = pointX - x1;
19    diffY = pointY - y1;
20  } else if (t > 1) {
21    //point is nearest to the end point i.e x2 and y2
22    // Ex: .
23    // cord do pto na reta = pto final(x2,y2);
24    *ptox = x2, *ptoy = y2;
25    diffX = pointX - x2;
26    diffY = pointY - y2;
27  } else {
28    //if perpendicular line intersect the line segment.
29    // pto nao esta mais proximo de uma das bordas do segmento
30    // Ex: .
31    // |
32    // | (Ângulo Reto)
33    // |
34    // cord x do pto na reta = (x1 + t * diffX)
35    // cord y do pto na reta = (y1 + t * diffY)
36    *ptox = (x1 + t * diffX), *ptoy = (y1 + t * diffY);
37    diffX = pointX - (x1 + t * diffX);
38    diffY = pointY - (y1 + t * diffY);
39  }
40  //returning shortest distance
41  return sqrt(diffX * diffX + diffY * diffY);
42 }

```

4.9. Point Inside Convex Polygon - Log(N)

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define INF 1e18
6 #define pb push_back
7 #define ii pair<int,int>
8 #define OK cout<<"OK"<<endl
9 #define debug(x) cout << #x " = " << (x) << endl
10 #define ff first
11 #define ss second
12 #define int long long
13
14 struct pto {
15   double x, y;
16   bool operator <(const pto &p) const {
17     return x < p.x || (x == p.x && y < p.y);
18     /* a impressao será em prioridade por mais a esquerda, mais

```

```

19   abaixo, e antihorário pelo cross abaixo */
20   }
21 };
22 double cross(const pto &O, const pto &A, const pto &B) {
23   return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
24 }
25
26 vector<pto> lower, upper;
27
28 vector<pto> convex_hull(vector<pto> &P) {
29   int n = P.size(), k = 0;
30   vector<pto> H(2 * n);
31   // Sort points lexicographically
32   sort(P.begin(), P.end());
33   // Build lower hull
34   for (int i = 0; i < n; ++i) {
35     // esse <= 0 representa sentido anti-horario, caso deseje mudar
36     // trocar por >= 0
37     while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
38       k--;
39     H[k++] = P[i];
40   }
41   // Build upper hull
42   for (int i = n - 2, t = k + 1; i >= 0; i--) {
43     // esse <= 0 representa sentido anti-horario, caso deseje mudar
44     // trocar por >= 0
45     while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
46       k--;
47     H[k++] = P[i];
48   }
49   H.resize(k);
50   /* o último ponto do vetor é igual ao primeiro, atente para isso
51   as vezes é necessário mudar */
52
53   int j = 1;
54   lower.pb(H.front());
55   while(H[j].x >= H[j-1].x) {
56     lower.pb(H[j++]);
57   }
58
59   int l = H.size()-1;
60   while(l >= j) {
61     upper.pb(H[l--]);
62   }
63   upper.pb(H[l--]);
64
65   return H;
66 }
67
68 bool insidePolygon(pto p, vector<pto> &arr) {
69
70   if(pair<double,double>(p.x, p.y) == pair<double,double>(lower[0].x,
71     lower[0].y))
72     return true;
73
74   pto lo = {p.x, -(double)INF};
75   pto hi = {p.x, (double)INF};
76   auto itl = lower_bound(lower.begin(), lower.end(), lo);
77   auto itu = lower_bound(upper.begin(), upper.end(), lo);
78
79   if(itl == lower.begin() || itu == upper.begin()) {
80     auto it = lower_bound(arr.begin(), arr.end(), lo);
81     auto it2 = lower_bound(arr.begin(), arr.end(), hi);
82     it2--;

```

```

82     if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y
      <= it2->y)
83         return true;
84     return false;
85 }
86 if(itl == lower.end() || itu == upper.end()) {
87     return false;
88 }
89
90 auto ol = itl, ou = itu;
91 ol--, ou--;
92 if(cross(*ol, *itl, p) >= 0 && cross(*ou, *itu, p) <= 0)
93     return true;
94
95 auto it = lower_bound(arr.begin(), arr.end(), lo);
96 auto it2 = lower_bound(arr.begin(), arr.end(), hi);
97 it2--;
98 if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y <=
      it2->y)
99     return true;
100
101 return false;
102
103 }
104
105 signed main () {
106     ios_base::sync_with_stdio(false);
107     cin.tie(NULL);
108
109     double n, m, k;
110
111     cin >> n >> m >> k;
112
113     vector<pto> arr(n);
114
115     for(pto &x: arr) {
116         cin >> x.x >> x.y;
117     }
118
119     convex_hull(arr);
120
121     pto p;
122
123     int c = 0;
124     while(m--) {
125         cin >> p.x >> p.y;
126         cout << (insidePolygon(p, arr) ? "dentro" : "fora") << endl;
127     }
128
129 }
130

```

4.10. Point Inside Polygon

```

1  /* Traça-se uma reta do ponto até um outro ponto qualquer fora do triangulo
2  e checa o número de interseção com a borda do polígono se este for impar
   então está dentro se não está fora */
3
4  // Define Infinite (Using INT_MAX caused overflow problems)
5  #define INF 10000
6
7  struct pto {
8      int x, y;

```

```

9      pto() {}
10     pto(int x, int y) : x(x), y(y) {}
11 };
12
13 // Given three colinear ptos p, q, r, the function checks if
14 // pto q lies on line segment 'pr'
15 bool onSegment(pto p, pto q, pto r) {
16     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
17         q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
18         return true;
19     return false;
20 }
21
22 // To find orientation of ordered triplet (p, q, r).
23 // The function returns following values
24 // 0 --> p, q and r are colinear
25 // 1 --> Clockwise
26 // 2 --> Counterclockwise
27 int orientation(pto p, pto q, pto r) {
28     int val = (q.y - p.y) * (r.x - q.x) -
29             (q.x - p.x) * (r.y - q.y);
30
31     if (val == 0) return 0; // colinear
32     return (val > 0)? 1: 2; // clock or counterclock wise
33 }
34
35 // The function that returns true if line segment 'p1q1'
36 // and 'p2q2' intersect.
37 bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
38     // Find the four orientations needed for general and
39     // special cases
40     int o1 = orientation(p1, q1, p2);
41     int o2 = orientation(p1, q1, q2);
42     int o3 = orientation(p2, q2, p1);
43     int o4 = orientation(p2, q2, q1);
44
45     // General case
46     if (o1 != o2 && o3 != o4)
47         return true;
48
49     // Special Cases
50     // p1, q1 and p2 are colinear and p2 lies on segment p1q1
51     if (o1 == 0 && onSegment(p1, p2, q1)) return true;
52
53     // p1, q1 and p2 are colinear and q2 lies on segment p1q1
54     if (o2 == 0 && onSegment(p1, q2, q1)) return true;
55
56     // p2, q2 and p1 are colinear and p1 lies on segment p2q2
57     if (o3 == 0 && onSegment(p2, p1, q2)) return true;
58
59     // p2, q2 and q1 are colinear and q1 lies on segment p2q2
60     if (o4 == 0 && onSegment(p2, q1, q2)) return true;
61
62     return false; // Doesn't fall in any of the above cases
63 }
64
65 // Returns true if the pto p lies inside the polygon[] with n vertices
66 bool isInside(pto polygon[], int n, pto p) {
67     // There must be at least 3 vertices in polygon[]
68     if (n < 3) return false;
69
70     // Create a pto for line segment from p to infinite
71     pto extreme = pto(INF, p.y);
72
73     // Count intersections of the above line with sides of polygon

```



```

74 int count = 0, i = 0;
75 do {
76     int next = (i+1)%n;
77
78     // Check if the line segment from 'p' to 'extreme' intersects
79     // with the line segment from 'polygon[i]' to 'polygon[next]'
80     if (doIntersect(polygon[i], polygon[next], p, extreme)) {
81         // If the pto 'p' is colinear with line segment 'i-next',
82         // then check if it lies on segment. If it lies, return true,
83         // otherwise false
84         if (orientation(polygon[i], p, polygon[next]) == 0)
85             return onSegment(polygon[i], p, polygon[next]);
86
87         count++;
88     }
89     i = next;
90 } while (i != 0);
91
92 // Return true if count is odd, false otherwise
93 return count&1; // Same as (count%2 == 1)
94 }

```

4.11. Points Inside And In Boundary Polygon

```

1 int cross(pto a, pto b) {
2     return a.x * b.y - b.x * a.y;
3 }
4
5 int boundaryCount(pto a, pto b) {
6     if(a.x == b.x)
7         return abs(a.y-b.y)-1;
8     if(a.y == b.y)
9         return abs(a.x-b.x)-1;
10    return _gcd(abs(a.x-b.x), abs(a.y-b.y))-1;
11 }
12
13 int totalBoundaryPolygon(vector<pto> &arr, int n) {
14
15     int boundPoint = n;
16     for(int i = 0; i < n; i++) {
17         boundPoint += boundaryCount(arr[i], arr[(i+1)%n]);
18     }
19     return boundPoint;
20 }
21
22 int polygonArea2(vector<pto> &arr, int n) {
23     int area = 0;
24     // N = quantidade de pontos no polígono e armazenados em p;
25     // OBS: VALE PARA CONVEXO E NÃO CONVEXO
26     for(int i = 0; i < n; i++){
27         area += cross(arr[i], arr[(i+1)%n]);
28     }
29     return abs(area);
30 }
31
32 int internalCount(vector<pto> &arr, int n) {
33
34     int area_2 = polygonArea2(arr, n);
35     int boundPoints = totalBoundaryPolygon(arr,n);
36     return (area_2 - boundPoints + 2)/2;
37 }

```

4.12. Polygon Area (3D)

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 struct point{
6     double x,y,z;
7     void operator=(const point & b){
8         x = b.x;
9         y = b.y;
10        z = b.z;
11    }
12 };
13
14 point cross(point a, point b){
15     point ret;
16     ret.x = a.y*b.z - b.y*a.z;
17     ret.y = a.z*b.x - a.x*b.z;
18     ret.z = a.x*b.y - a.y*b.x;
19     return ret;
20 }
21
22 int main(){
23     int num;
24     cin >> num;
25     point v[num];
26     for(int i=0; i<num; i++) cin >> v[i].x >> v[i].y >> v[i].z;
27
28     point cur;
29     cur.x = 0, cur.y = 0, cur.z = 0;
30
31     for(int i=0; i<num; i++){
32         point res = cross(v[i], v[(i+1)%num]);
33         cur.x += res.x;
34         cur.y += res.y;
35         cur.z += res.z;
36     }
37
38     double ans = sqrt(cur.x*cur.x + cur.y*cur.y + cur.z*cur.z);
39
40     double area = abs(ans);
41
42     cout << fixed << setprecision(9) << area/2. << endl;
43 }

```

4.13. Polygon Area

```

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

```

4.14. Segment-Segment Intersection

```

1 // Given three colinear points p, q, r, the function checks if
2 // point q lies on line segment 'pr'
3 int onSegment(Point p, Point q, Point r) {
4     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y <= max(p.y, r.y)
5         && q.y >= min(p.y, r.y))

```

```

5     return true;
6     return false;
7 }
8 /* PODE SER RETIRADO
9 int onSegmentNotBorda(Point p, Point q, Point r) {
10     if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) && q.y <= max(p.y, r.y)
        && q.y >= min(p.y, r.y))
11         return true;
12     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y < max(p.y, r.y)
        && q.y > min(p.y, r.y))
13         return true;
14     return false;
15 }
16 */
17 // To find orientation of ordered triplet (p, q, r).
18 // The function returns following values
19 // 0 --> p, q and r are colinear
20 // 1 --> Clockwise
21 // 2 --> Counterclockwise
22 int orientation(Point p, Point q, Point r) {
23     int val = (q.y - p.y) * (r.x - q.x) -
24             (q.x - p.x) * (r.y - q.y);
25     if (val == 0) return 0; // colinear
26     return (val > 0) ? 1 : 2; // clock or counterclock wise
27 }
28 // The main function that returns true if line segment 'p1p2'
29 // and 'q1q2' intersect.
30 int doIntersect(Point p1, Point p2, Point q1, Point q2) {
31     // Find the four orientations needed for general and
32     // special cases
33     int o1 = orientation(p1, p2, q1);
34     int o2 = orientation(p1, p2, q2);
35     int o3 = orientation(q1, q2, p1);
36     int o4 = orientation(q1, q2, p2);
37
38     // General case
39     if (o1 != o2 && o3 != o4) return 2;
40
41     /* PODE SER RETIRADO
42     if (o1 == o2 && o2 == o3 && o3 == o4 && o4 == 0) {
43         //INTERCEPTAM EM RETA
44         if (onSegmentNotBorda(p1, q1, p2) || onSegmentNotBorda(p1, q2, p2)) return 1;
45         if (onSegmentNotBorda(q1, p1, q2) || onSegmentNotBorda(q1, p2, q2)) return 1;
46     }
47     */
48     // Special Cases (INTERCEPTAM EM PONTO)
49     // p1, p2 and q1 are colinear and q1 lies on segment p1p2
50     if (o1 == 0 && onSegment(p1, q1, p2)) return 2;
51     // p1, p2 and q1 are colinear and q2 lies on segment p1p2
52     if (o2 == 0 && onSegment(p1, q2, p2)) return 2;
53     // q1, q2 and p1 are colinear and p1 lies on segment q1q2
54     if (o3 == 0 && onSegment(q1, p1, q2)) return 2;
55     // q1, q2 and p2 are colinear and p2 lies on segment q1q2
56     if (o4 == 0 && onSegment(q1, p2, q2)) return 2;
57     return false; // Doesn't fall in any of the above cases
58 }
59 // OBS: SE (C2/A2 == C1/A1) SÃO COLINEARES

```

4.15. Upper And Lower Hull

```

1 struct pto {
2     double x, y;
3     bool operator <(const pto &p) const {
4         return x < p.x || (x == p.x && y < p.y);

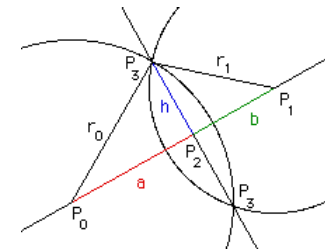
```

```

5     /* a impressao será em prioridade por mais a esquerda, mais
6        abaixo, e anti-horário pelo cross abaixo */
7 }
8 };
9 double cross(const pto &O, const pto &A, const pto &B) {
10     return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
11 }
12
13 vector<pto> lower, upper;
14
15 vector<pto> convex_hull(vector<pto> &P) {
16     int n = P.size(), k = 0;
17     vector<pto> H(2 * n);
18     // Sort points lexicographically
19     sort(P.begin(), P.end());
20     // Build lower hull
21     for (int i = 0; i < n; ++i) {
22         // esse <= 0 representa sentido anti-horário, caso deseje mudar
23         // trocar por >= 0
24         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
25             k--;
26         H[k++] = P[i];
27     }
28     // Build upper hull
29     for (int i = n - 2; i >= 0; i--) {
30         // esse <= 0 representa sentido anti-horário, caso deseje mudar
31         // trocar por >= 0
32         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
33             k--;
34         H[k++] = P[i];
35     }
36     H.resize(k);
37     /* o último ponto do vetor é igual ao primeiro, atente para isso
38        as vezes é necessário mudar */
39
40     int j = 1;
41     lower.pb(H.front());
42     while (H[j].x >= H[j - 1].x) {
43         lower.pb(H[j++]);
44     }
45
46     int l = H.size() - 1;
47     while (l >= j) {
48         upper.pb(H[l--]);
49     }
50     upper.pb(H[l--]);
51
52     return H;
53 }

```

4.16. Circle Circle Intersection



4.17. Circle Circle Intersection

```

1  /* circle_circle_intersection() *
2  * Determine the points where 2 circles in a common plane intersect.
3  *
4  * int circle_circle_intersection(
5  * // center and radius of 1st circle
6  * double x0, double y0, double r0,
7  * // center and radius of 2nd circle
8  * double x1, double y1, double r1,
9  * // 1st intersection point
10 * double *xi, double *yi,
11 * // 2nd intersection point
12 * double *xi_prime, double *yi_prime)
13 *
14 * This is a public domain work. 3/26/2005 Tim Voght
15 *
16 */
17
18 int circle_circle_intersection(double x0, double y0, double r0, double x1,
19                             double y1, double r1, double *xi, double *yi,
20                             double *xi_prime, double *yi_prime) {
21     double a, dx, dy, d, h, rx, ry;
22     double x2, y2;
23
24     /* dx and dy are the vertical and horizontal distances between
25     * the circle centers.
26     */
27     dx = x1 - x0;
28     dy = y1 - y0;
29
30     /* Determine the straight-line distance between the centers. */
31     // d = sqrt((dy*dy) + (dx*dx));
32     d = hypot(dx, dy); // Suggested by Keith Briggs
33
34     /* Check for solvability. */
35     if (d > (r0 + r1)) {
36         /* no solution. circles do not intersect. */
37         return 0;
38     }
39     if (d < fabs(r0 - r1)) {
40         /* no solution. one circle is contained in the other */
41         return 0;
42     }
43
44     /* 'point 2' is the point where the line through the circle
45     * intersection points crosses the line between the circle
46     * centers.
47     */
48
49     /* Determine the distance from point 0 to point 2. */
50     a = ((r0 * r0) - (r1 * r1) + (d * d)) / (2.0 * d);
51
52     /* Determine the coordinates of point 2. */
53     x2 = x0 + (dx * a / d);
54     y2 = y0 + (dy * a / d);
55
56     /* Determine the distance from point 2 to either of the
57     * intersection points.
58     */
59     h = sqrt((r0 * r0) - (a * a));
60
61     /* Now determine the offsets of the intersection points from
62     * point 2.
63     */

```

```

64     rx = -dy * (h / d);
65     ry = dx * (h / d);
66
67     /* Determine the absolute intersection points. */
68     *xi = x2 + rx;
69     *xi_prime = x2 - rx;
70     *yi = y2 + ry;
71     *yi_prime = y2 - ry;
72
73     return 1;
74 }

```

4.18. Struct Point And Line

```

1  int sgn(double x) {
2      if(abs(x) < 1e-8) return 0;
3      return x > 0 ? 1 : -1;
4  }
5  inline double sqr(double x) { return x * x; }
6
7  struct Point {
8      double x, y, z;
9      Point() {}
10     Point(double a, double b): x(a), y(b) {};
11     Point (double x, double y, double z): x(x), y(y), z(z) {}
12
13     void input() { scanf("%lf %lf", &x, &y); };
14     friend Point operator+(const Point &a, const Point &b) {
15         return Point(a.x + b.x, a.y + b.y);
16     }
17     friend Point operator-(const Point &a, const Point &b) {
18         return Point(a.x - b.x, a.y - b.y);
19     }
20
21     bool operator !=(const Point& a) const {
22         return (x != a.x || y != a.y);
23     }
24
25     bool operator <(const Point &a) const{
26         if(x == a.x)
27             return y < a.y;
28         return x < a.x;
29     }
30
31     double norm() {
32         return sqrt(sqr(x) + sqr(y));
33     }
34 };
35 double det(const Point &a, const Point &b) {
36     return a.x * b.y - a.y * b.x;
37 }
38 double dot(const Point &a, const Point &b) {
39     return a.x * b.x + a.y * b.y;
40 }
41 double dist(const Point &a, const Point &b) {
42     return (a-b).norm();
43 }
44
45 struct Line {
46     Point a, b;
47     Line() {}
48     Line(Point x, Point y): a(x), b(y) {};
49 };
50

```

```

51 double dis_point_segment(const Point p, const Point s, const Point t) {
52     if(sgn(dot(p-s, t-s)) < 0)
53         return (p-s).norm();
54     if(sgn(dot(p-t, s-t)) < 0)
55         return (p-t).norm();
56     return abs(det(s-p, t-p) / dist(s, t));
57 }
58

```

5. Graphs

5.1. All Eulerian Path Or Tour

```

1 struct edge {
2     int v, id;
3     edge() {}
4     edge(int v, int id) : v(v), id(id) {}
5 };
6
7 // The undirected + path and directed + tour wasn't tested in a problem.
8 // TEST AGAIN BEFORE SUBMITTING IT!
9 namespace graph {
10     // Namespace which auxiliary functions are defined.
11     namespace detail {
12         pair<bool, pair<int, int>> check_both_directed(const
13             vector<vector<edge>> &adj, const vector<int> &in_degree) {
14             // source and destination
15             int src = -1, dest = -1;
16             // adj[i].size() represents the out degree of an vertex
17             for(int i = 0; i < adj.size(); i++) {
18                 if((int)adj[i].size() - in_degree[i] == 1) {
19                     if(src != -1)
20                         return make_pair(false, pair<int, int>());
21                     src = i;
22                 } else if((int)adj[i].size() - in_degree[i] == -1) {
23                     if(dest != -1)
24                         return make_pair(false, pair<int, int>());
25                     dest = i;
26                 } else if(abs((int)adj[i].size() - in_degree[i]) > 1)
27                     return make_pair(false, pair<int, int>());
28             }
29             if(src == -1 && dest == -1)
30                 return make_pair(true, pair<int, int>(src, dest));
31             else if(src != -1 && dest != -1)
32                 return make_pair(true, pair<int, int>(src, dest));
33             return make_pair(false, pair<int, int>());
34         }
35     }
36
37     /// Builds the path/tour for directed graphs.
38     void build(const int u, vector<int> &tour, vector<vector<edge>> &adj,
39         vector<bool> &used) {
40         while(!adj[u].empty()) {
41             const edge e = adj[u].back();
42             if(!used[e.id]) {
43                 used[e.id] = true;
44                 adj[u].pop_back();
45                 build(e.v, tour, adj, used);
46             } else
47                 adj[u].pop_back();
48         }
49         tour.push_back(u);
50

```

```

51     }
52     /// Auxiliary function to build the eulerian tour/path.
53     vector<int> set_build(vector<vector<edge>> &adj, const int E, const int
54         first) {
55         vector<int> path;
56         vector<bool> used(E + 3);
57
58         build(first, path, adj, used);
59
60         for(int i = 0; i < adj.size(); i++)
61             // if there are some remaining edges, it's not possible to build the
62             tour.
63             if(adj[i].size())
64                 return vector<int>();
65
66         reverse(path.begin(), path.end());
67         return path;
68     }
69
70     /// All vertices v should have in_degree[v] == out_degree[v]. It must not
71     contain a specific
72     start and end vertices.
73     /// Time complexity: O(V * (log V) + E)
74     bool has_euler_tour_directed(const vector<vector<edge>> &adj, const
75         vector<int> &in_degree) {
76         const pair<bool, pair<int, int>> aux = detail::check_both_directed(adj,
77             in_degree);
78         const bool valid = aux.first;
79         const int src = aux.second.first;
80         const int dest = aux.second.second;
81         return (valid && src == -1 && dest == -1);
82     }
83
84     /// A directed graph has an eulerian path/tour if has:
85     /// - One vertex v such that out_degree[v] - in_degree[v] == 1
86     /// - One vertex v such that in_degree[v] - out_degree[v] == 1
87     /// - The remaining vertices v such that in_degree[v] == out_degree[v]
88     /// or
89     /// - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR
90     /// Returns a boolean value that indicates whether there's a path or not.
91     /// If there's a valid path it also returns two numbers: the source and
92     the destination.
93     /// If the source and destination can be an arbitrary vertex it will
94     return the pair (-1, -1)
95     /// for the source and destination (it means the contains an eulerian
96     tour).
97     /// Time complexity: O(V + E)
98     pair<bool, pair<int, int>> has_euler_path_directed(const
99         vector<vector<edge>> &adj, const vector<int> &in_degree) {
100         return detail::check_both_directed(adj, in_degree);
101     }
102
103     /// Returns the euler path. If the graph doesn't have an euler path it
104     returns an empty vector.
105     /// Time Complexity: O(V + E) for directed, O(V * log(V) + E) for
106     undirected.
107     /// Time Complexity: O(adj.size() + sum(adj[i].size()))
108     vector<int> get_euler_path_directed(const int E, vector<vector<edge>>
109         &adj, const vector<int> &in_degree) {
110

```

```

103     const pair<bool, pair<int, int>> aux = has_euler_path_directed(adj,
104         in_degree);
105     const bool valid = aux.first;
106     const int src = aux.second.first;
107     const int dest = aux.second.second;
108
109     if(!valid)
110         return vector<int>();
111
112     int first;
113     if(src != -1)
114         first = src;
115     else {
116         first = 0;
117         while(adj[first].empty())
118             first++;
119     }
120     return detail::set_build(adj, E, first);
121 }
122
123 /// Returns the euler tour. If the graph doesn't have an euler tour it
124 /// returns an empty vector.
125 /// Time Complexity: O(V + E)
126 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
127 vector<int> get_euler_tour_directed(const int E, vector<vector<edge>>
128     &adj, const vector<int> &in_degree) {
129     const bool valid = has_euler_tour_directed(adj, in_degree);
130
131     if(!valid)
132         return vector<int>();
133
134     int first = 0;
135     while(adj[first].empty())
136         first++;
137
138     return detail::set_build(adj, E, first);
139 }
140
141 // The graph has a tour that passes to every edge exactly once and gets
142 // back to the first edge on the tour.
143 // A graph with an euler path has zero odd degree vertex.
144 //
145 // Time Complexity: O(V)
146 bool has_euler_tour_undirected(const vector<int> &degree) {
147     for(int i = 0; i < degree.size(); i++)
148         if(degree[i] & 1)
149             return false;
150     return true;
151 }
152
153 // The graph has a path that passes to every edge exactly once.
154 // It doesn't necessarily gets back to the beginning.
155 //
156 // A graph with an euler path has two or zero (tour) odd degree vertices.
157 //
158 // Returns a pair with the startpoint/endpoint of the path.
159 //
160 // Time Complexity: O(V)
161 pair<bool, pair<int, int>> has_euler_path_undirected(const vector<int>
162     &degree) {
163     vector<int> odd_degree;
164     for(int i = 0; i < degree.size(); i++)

```

```

164         if(degree[i] & 1)
165             odd_degree.pb(i);
166
167     if(odd_degree.size() == 0)
168         return make_pair(true, make_pair(-1, -1));
169     else if (odd_degree.size() == 2)
170         return make_pair(true, make_pair(odd_degree.front(),
171             odd_degree.back()));
172     else
173         return make_pair(false, pair<int, int>());
174 }
175
176 vector<int> get_euler_tour_undirected(const int E, const vector<int>
177     &degree, vector<vector<edge>> &adj) {
178     if(!has_euler_tour_undirected(degree))
179         return vector<int>();
180
181     int first = 0;
182     while(adj[first].empty())
183         first++;
184
185     return detail::set_build(adj, E, first);
186 }
187
188 /// Returns the euler tour. If the graph doesn't have an euler tour it
189 /// returns an empty vector.
190 /// Time Complexity: O(V + E)
191 /// Time Complexity: O(adj.size() + sum(adj[i].size()))
192 vector<int> get_euler_path_undirected(const int E, const vector<int>
193     &degree, vector<vector<edge>> &adj) {
194     auto aux = has_euler_path_undirected(degree);
195     const bool valid = aux.first;
196     const int x = aux.second.first;
197     const int y = aux.second.second;
198
199     if(!valid)
200         return vector<int>();
201
202     int first;
203     if(x != -1) {
204         first = x;
205         adj[x].emplace_back(y, E + 1);
206         adj[y].emplace_back(x, E + 1);
207     } else {
208         first = 0;
209         while(adj[first].empty())
210             first++;
211     }
212
213     vector<int> ans = detail::set_build(adj, E, first);
214     reverse(ans.begin(), ans.end());
215     if(x != -1)
216         ans.pop_back();
217     return ans;
218 }

```

5.2. Articulation Points

```

1 namespace graph {
2 unordered_set<int> ap;
3 vector<int> low, disc;
4 int cur_time = 1;

```

```

5
6 void dfs_ap(const int u, const int p, const vector<vector<int>> &adj) {
7     low[u] = disc[u] = cur_time++;
8     int children = 0;
9
10    for (const int v : adj[u]) {
11        // DO NOT ADD PARALLEL EDGES
12        if (disc[v] == 0) {
13            ++children;
14            dfs_ap(v, u, adj);
15
16            low[u] = min(low[v], low[u]);
17            if (p == -1 && children > 1)
18                ap.emplace(u);
19            if (p != -1 && low[v] >= disc[u])
20                ap.emplace(u);
21        } else if (v != p)
22            low[u] = min(low[u], disc[v]);
23    }
24 }
25
26 void init_ap(const int n) {
27     cur_time = 1;
28     ap = unordered_set<int>();
29     low = vector<int>(n, 0);
30     disc = vector<int>(n, 0);
31 }
32
33 /// THE GRAPH MUST BE UNDIRECTED!
34 ///
35 /// Returns the vertices in which their removal disconnects the graph.
36 ///
37 /// Time Complexity: O(V + E)
38 vector<int> articulation_points(const int indexed_from,
39                               const vector<vector<int>> &adj) {
40     init_ap(adj.size());
41     vector<int> ans;
42     for (int u = indexed_from; u < adj.size(); ++u) {
43         if (disc[u] == 0)
44             dfs_ap(u, -1, adj);
45         if (ap.count(u))
46             ans.emplace_back(u);
47     }
48     return ans;
49 }
50 }; // namespace graph

```

5.3. Bellman Ford

```

1 struct edge {
2     int src, dest, weight;
3     edge() {}
4     edge(int src, int dest, int weight) : src(src), dest(dest), weight(weight)
5     {}
6
7     bool operator<(const edge &a) const {
8         return weight < a.weight;
9     }
10 };
11
12 /// Works to find the shortest path with negative edges.
13 /// Also detects cycles.
14 ///
15 /// Time Complexity: O(n * e)

```

```

15 /// Space Complexity: O(n)
16 bool bellman_ford(vector<edge> &edges, int src, int n) {
17     // n = qtd of vertices, E = qtd de arestas
18
19     // To calculate the shortest path uncomment the line below
20     // vector<int> dist(n, INF);
21
22     // To check cycles uncomment the line below
23     // vector<int> dist(n, 0);
24
25     vector<int> pai(n, -1);
26     int E = edges.size();
27
28     dist[src] = 0;
29     // Relax all edges n - 1 times.
30     // A simple shortest path from src to any other vertex can have at-most n
31     // - 1 edges.
32     for (int i = 1; i <= n - 1; i++) {
33         for (int j = 0; j < E; j++) {
34             int u = edges[j].src;
35             int v = edges[j].dest;
36             int weight = edges[j].weight;
37             if (dist[u] != INF && dist[u] + weight < dist[v]) {
38                 dist[v] = dist[u] + weight;
39                 pai[v] = u;
40             }
41         }
42     }
43
44     // Check for NEGATIVE-WEIGHT CYCLES.
45     // The above step guarantees shortest distances if graph doesn't contain
46     // negative weight cycle.
47     // If we get a shorter path, then there is a cycle.
48     bool is_cycle = false;
49     int vert_in_cycle;
50     for (int i = 0; i < E; i++) {
51         int u = edges[i].src;
52         int v = edges[i].dest;
53         int weight = edges[i].weight;
54         if (dist[u] != INF && dist[u] + weight < dist[v]) {
55             is_cycle = true;
56             pai[v] = u;
57             vert_in_cycle = v;
58         }
59     }
60
61     if(is_cycle) {
62         for(int i = 0; i < n; i++)
63             vert_in_cycle = pai[vert_in_cycle];
64
65         vector<int> cycle;
66         for(int v = vert_in_cycle; (v != vert_in_cycle || cycle.size() <= 1) ; v
67             = pai[v])
68             cycle.pb(v);
69
70         reverse(cycle.begin(), cycle.end());
71
72         for(int x: cycle) {
73             cout << x + 1 << ' ';
74         }
75         cout << cycle.front() + 1 << endl;
76         return true;
77     } else
78         return false;
79 }

```

5.4. Bipartite Check

```

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

```

5.5. Block Cut Tree

```

1  // based on kokosha's implementation.
2  /// INDEXED FROM ZERO!!!!
3  class BCT {
4      vector<vector<pair<int, int>>> adj;
5      vector<pair<int, int>> edges;
6      /// Stores the edges in the i-th component.
7      vector<vector<int>> comps;
8      /// Stores the vertices in the i-th component.
9      vector<vector<int>> vert_in_comp;
10     int cur_time = 0;
11     vector<int> disc, conv;
12     vector<vector<int>> adj_bct;
13     const int n;
14
15     /// Finds the biconnected components.
16     int dfs(const int x, const int p, stack<int> &st) {
17         int low = disc[x] = ++cur_time;
18         for (const pair<int, int> &e : adj[x]) {
19             const int v = e.first, idx = e.second;
20             if (idx != p) {
21                 if (!disc[v]) { // if haven't passed
22                     st.emplace(idx); // disc[x] < low -> bridge
23                     const int low_at = dfs(v, idx, st);
24                     low = min(low, low_at);
25                     if (disc[x] <= low_at) {
26                         comps.emplace_back();
27                         vector<int> &tmp = comps.back();
28                         for (int y = -1; y != idx; st.pop())
29                             tmp.emplace_back(y = st.top());
30                     }
31                 } else if (disc[v] < disc[x]) // back_edge
32                     low = min(low, disc[v]), st.emplace(idx);
33             }
34         }
35     }

```

```

35     return low;
36 }
37
38 /// Splits the graph into biconnected components.
39 void split() {
40     adj_bct.resize(n + edges.size() + 1);
41     stack<int> st;
42     for (int i = 0; i < n; ++i)
43         if (!disc[i])
44             dfs(i, -1, st);
45
46     vector<bool> in(n);
47     for (const vector<int> &comp : comps) {
48         vert_in_comp.emplace_back();
49         for (const int e : comp) {
50             const int u = edges[e].first, v = edges[e].second;
51             if (!in[u])
52                 in[u] = 1, vert_in_comp.back().emplace_back(u);
53             if (!in[v])
54                 in[v] = 1, vert_in_comp.back().emplace_back(v);
55         }
56         for (const int e : comp)
57             in[edges[e].first] = in[edges[e].second] = 0;
58     }
59 }
60
61 /// Algorithm: It compresses the biconnected components into one vertex.
62     Then
63     /// it creates a bipartite graph with the original vertices on the left and
64     /// the bcc's on the right. After that, it connects with an edge the i-th
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 void init() {
88     disc.resize(n);
89     conv.resize(n);
90     adj.resize(n);
91 }
92
93 public:
94     /// Pass the number of vertices to the constructor.
95     BCT(const int n) : n(n) { init(); }
96
97     /// Adds an bidirectional edge.
98     void add_edge(const int u, const int v) {

```

```

97     assert(0 <= min(u, v)), assert(max(u, v) < n), assert(u != v);
98     adj[u].emplace_back(v, edges.size());
99     adj[v].emplace_back(u, edges.size());
100     edges.emplace_back(u, v);
101 }
102
103 /// Returns the bct tree. It builds the tree if it's not computed.
104 ///
105 /// Time Complexity: O(n + m)
106 vector<vector<int>>> tree() {
107     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. 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 }

```

5.10. 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'.

```

```

15 //
16 // We can observe that every node in this graph has equal in-degree and
17 // out-degree, which means that a Eulerian circuit exists in this graph.
18
19 namespace graph {
20 namespace detail {
21 // Finding an valid eulerian path
22 void dfs(const string &node, const string &alphabet, set<string> &vis,
23         string &edges_order) {
24     for (char c : alphabet) {
25         string nxt = node + c;
26         if (vis.count(nxt))
27             continue;
28
29         vis.insert(nxt);
30         nxt.erase(nxt.begin());
31         dfs(nxt, alphabet, vis, edges_order);
32         edges_order += c;
33     }
34 }
35 }; // namespace detail
36
37 // Returns a string in which every string of the alphabet of size n appears
38 // in
39 // the resulting string exactly once.
40 // Time Complexity:  $O(\text{alphabet.size()} \wedge n \star \log_2(\text{alphabet.size()} \wedge n))$ 
41 string de_bruijn(const int n, const string &alphabet) {
42     set<string> vis;
43     string edges_order;
44
45     string starting_node = string(n - 1, alphabet.front());
46     detail::dfs(starting_node, alphabet, vis, edges_order);
47
48     return edges_order + starting_node;
49 }
50 }; // namespace graph

```

5.11. Diameter In Tree

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

5.12. Dijkstra + Dij Graph

```

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

```

```

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

```

```

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

```

5.13. 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 graph
8          int flow = 0;
9          Edge(const int v, const int cap) : v(v), cap(cap) {}
10     };
11
12     private:
13         static constexpr int INF = (sizeof(int) == 4 ? 1e9 : 2e18) + 1e5;
14         bool COMPUTED = false;
15         int _max_flow;
16         vector<Edge> edges;
17         // holds the indexes of each edge present in each vertex.
18         vector<vector<int>> adj;
19         const int n;
20         // src will be always 0 and sink n+1.
21         const int src, sink;
22         vector<int> level, ptr;
23
24     private:
25         vector<vector<int>> _flow_table() {
26             vector<vector<int>> table(n, vector<int>(n, 0));
27             for (int u = 0; u <= sink; ++u)
28                 for (const int idx : adj[u])
29                     // checks if it's not a reverse edge
30                     if (!(idx & 1))
31                         table[u][edges[idx].v] += edges[idx].flow;
32             return table;
33         }
34
35         /// Algorithm: Greedily all vertices from the matching will be added and,
36         /// after that, edges in which one of the vertices is not covered will
37         /// also be
38         /// added to the answer.
39         vector<pair<int, int>> _min_edge_cover() {
40             vector<bool> covered(n, false);
41             vector<pair<int, int>> ans;
42             for (int u = 1; u < sink; ++u) {
43                 for (const int idx : adj[u]) {
44                     const Edge &e = edges[idx];
45                     // ignore if it is a reverse edge or an edge linked to the sink
46                     if (idx & 1 || e.v == sink)
47                         continue;
48                     if (e.flow == e.cap) {
49                         ans.emplace_back(u, e.v);
50                         covered[u] = covered[e.v] = true;
51                         break;
52                     }
53                 }
54             }
55
56             for (int u = 1; u < sink; ++u) {
57                 for (const int idx : adj[u]) {
58                     const Edge &e = edges[idx];
59                     if (idx & 1 || e.v == sink)
60                         continue;
61                     if (e.flow < e.cap && (!covered[u] || !covered[e.v])) {
62                         ans.emplace_back(u, e.v);
63                         covered[u] = covered[e.v] = true;
64                     }
65                 }
66             }
67         }
68     };

```

```

64     }
65     }
66     return ans;
67 }
68
69 /// Algorithm: Takes the complement of the vertex cover.
70 vector<int> _max_ind_set(const int max_left) {
71     const vector<int> mvc = _min_vertex_cover(max_left);
72     vector<bool> contains(n);
73     for (const int v : mvc)
74         contains[v] = true;
75     vector<int> ans;
76     for (int i = 1; i < sink; ++i)
77         if (!contains[i])
78             ans.emplace_back(i);
79     return ans;
80 }
81
82 void dfs_vc(const int u, vector<bool> &vis, const bool left,
83            const vector<vector<int>> &paths) {
84     vis[u] = true;
85     for (const int idx : adj[u]) {
86         const Edge &e = edges[idx];
87         if (vis[e.v])
88             continue;
89         // saturated edges goes from right to left
90         if (left && paths[u][e.v] == 0)
91             dfs_vc(e.v, vis, left ^ 1, paths);
92         // non-saturated edges goes from left to right
93         else if (!left && paths[e.v][u] == 1)
94             dfs_vc(e.v, vis, left ^ 1, paths);
95     }
96 }
97
98 /// Algorithm: The edges that belong to the Matching M will go from right
99 to
100 /// left, all other edges will go from left to right. A DFS will be run
101 /// starting at all left vertices that are not incident to edges in M. Some
102 /// vertices of the graph will become visited during this DFS and some
103 /// not-visited. To get minimum vertex cover all visited right
104 /// vertices of M will be taken, and all not-visited left vertices of M.
105 /// Source: codeforces.com/blog/entry/17534?comment=223759
106 vector<int> _min_vertex_cover(const int max_left) {
107     vector<bool> vis(n, false), saturated(n, false);
108     const auto paths = flow_table();
109
110     for (int i = 1; i <= max_left; ++i) {
111         for (int j = max_left + 1; j < sink; ++j)
112             if (paths[i][j] > 0) {
113                 saturated[i] = saturated[j] = true;
114                 break;
115             }
116         if (!saturated[i] && !vis[i])
117             dfs_vc(i, vis, 1, paths);
118     }
119
120     vector<int> ans;
121     for (int i = 1; i <= max_left; ++i)
122         if (saturated[i] && !vis[i])
123             ans.emplace_back(i);
124
125     for (int i = max_left + 1; i < sink; ++i)
126         if (saturated[i] && vis[i])
127             ans.emplace_back(i);

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

5.14. Dsu

```

1 class DSU {
2     vector<int> root, sz;
3
4 public:
5     DSU(const int n) {
6         root.resize(n + 1);
7         iota(root.begin(), root.begin() + n + 1, 0ll);
8         sz.resize(n + 1, 1);
9     }
10
11     /// Returns the id of the set in which the element x belongs.
12     ///
13     /// Time Complexity:  $O(1)$ 
14     int Find(const int x) {
15         if (root[x] == x)
16             return x;
17         return root[x] = Find(root[x]);
18     }
19
20     /// Unites two sets in which p and q belong.
21     /// Returns false if they already belong to the same set.
22     ///
23     /// Time Complexity:  $O(1)$ 
24     bool Union(int p, int q) {
25         p = Find(p), q = Find(q);
26         if (p == q)
27             return false;
28
29         if (sz[p] < sz[q])
30             swap(p, q);
31
32         root[q] = p;
33         sz[p] += sz[q];
34         return true;
35     }
36 };

```

5.15. Dsu On Tree

```

1 /// Problem: What's the level of the subtree of u which contains the most
2 /// of nodes? In case of tie, choose the level with small number.
3
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     // brute force all subtrees other than idx
59     add(u, p, l, idx, 1);
60     // the answer of u is the level ans. As it is relative to the input tree we
61     // need to subtract it to the current level of u
62     q[u] = ans - l;
63     if (keep == 0) {
64         // removing the calculated answer for the subtree, if it doesn't belong
65         // to
66         // the biggest subtree of it's parent (keep = 0)
67         add(u, p, l, -1, -1);
68         // clearing the answer
69         maxx = 0, ans = 0;
70     }
71 }
72
73 // MODIFY TO WORK WITH DISCONNECTED GRAPHS!!!
74 // Time Complexity: O(n log n)

```

```

75 void precalculate() {
76     sz = sub_sz(1, adj);
77     dfs(1, -1, 0, 0);
78 }

```

5.16. 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 }

```

5.17. Functional Graph

```

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

```

```

41 while(!q.empty()) {
42     int u = q.front();
43     q.pop();
44
45     process_order.push_back(u);
46
47     if(--in_degree[nxt[u]] == 0) {
48         q.push(nxt[u]);
49         vis[nxt[u]] = true;
50     }
51 }
52
53 int cycle_cnt = 0;
54 for(int i = indexed_from; i < n + indexed_from; i++)
55     if(!vis[i])
56         compute_cycle(i, nxt, vis);
57
58 for(int i = (int)process_order.size() - 1; i >= 0; i--) {
59     int u = process_order[i];
60
61     this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
62     this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
63     this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
64 }
65 }
66
67 void allocate(int n, int indexed_from) {
68     this->cycle.resize(n + indexed_from);
69     this->first.resize(n + indexed_from);
70
71     this->in_cycle.resize(n + indexed_from, false);
72     this->cycle_id.resize(n + indexed_from, -1);
73     this->id_in_cycle.resize(n + indexed_from, -1);
74     this->near_in_cycle.resize(n + indexed_from);
75     this->id_near_cycle.resize(n + indexed_from);
76     this->cycle_dist.resize(n + indexed_from);
77 }
78
79 public:
80 Functional_Graph(int n, int indexed_from, vector<int> &nxt, vector<int>
    &in_degree) {
81     this->allocate(n, indexed_from);
82     this->build(n, indexed_from, nxt, in_degree);
83 }
84
85 // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
86
87 // number of cycles
88 int cycle_cnt = 0;
89 // Vertices present in the i-th cycle.
90 vector<vector<int>> cycle;
91 // first vertex of the i-th cycle
92 vector<int> first;
93
94 // The i-th vertex is present in any cycle?
95 vector<bool> in_cycle;
96 // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
    cycle.
97 vector<int> cycle_id;
98 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
    belong to any cycle.
99 vector<int> id_in_cycle;
100 // Represents the id of the nearest vertex present in a cycle.
101 vector<int> near_in_cycle;
102 // Represents the id of the nearest cycle.

```

```

103 vector<int> id_near_cycle;
104 // Distance to the nearest cycle.
105 vector<int> cycle_dist;
106 // Represent the id of the component of the vertex.
107 // Equal to id_near_cycle
108 vector<int> &comp = id_near_cycle;
109 };
110
111 class Functional_Graph {
112     // FOR UNDIRECTED GRAPH
113 private:
114     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis,
        vector<vector<int>> &adj) {
115         int id_cycle = cycle_cnt++;
116         int cur_id = 0;
117         this->first[id_cycle] = u;
118
119         while(!vis[u]) {
120             vis[u] = true;
121
122             this->cycle[id_cycle].push_back(u);
123             nxt[u] = find_nxt(u, vis, adj);
124             if(nxt[u] == -1)
125                 nxt[u] = this->first[id_cycle];
126
127             this->in_cycle[u] = true;
128             this->cycle_id[u] = id_cycle;
129             this->id_in_cycle[u] = cur_id;
130             this->near_in_cycle[u] = u;
131             this->id_near_cycle[u] = id_cycle;
132             this->cycle_dist[u] = 0;
133
134             u = nxt[u];
135             cur_id++;
136         }
137     }
138
139     int find_nxt(int u, vector<bool> &vis, vector<vector<int>> &adj) {
140         for(int v: adj[u])
141             if(!vis[v])
142                 return v;
143         return -1;
144     }
145
146     // Time Complexity: O(V + E)
147     void build(int n, int indexed_from, vector<int> &degree,
        vector<vector<int>> &adj) {
148         queue<int> q;
149         vector<bool> vis(n + indexed_from, false);
150         vector<int> nxt(n + indexed_from);
151         for(int i = indexed_from; i < n + indexed_from; i++) {
152             if(adj[i].size() == 1) {
153                 q.push(i);
154                 vis[i] = true;
155             }
156         }
157
158         vector<int> process_order;
159         process_order.reserve(n + indexed_from);
160         while(!q.empty()) {
161             int u = q.front();
162             q.pop();
163
164             process_order.push_back(u);
165

```



```

166     nxt[u] = find_nxt(u, vis, adj);
167     if(--degree[nxt[u]] == 1) {
168         q.push(nxt[u]);
169         vis[nxt[u]] = true;
170     }
171 }
172
173 int cycle_cnt = 0;
174 for(int i = indexed_from; i < n + indexed_from; i++)
175     if(!vis[i])
176         compute_cycle(i, nxt, vis, adj);
177
178 for(int i = (int)process_order.size() - 1; i >= 0; i--) {
179     int u = process_order[i];
180
181     this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
182     this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
183     this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
184 }
185 }
186
187 void allocate(int n, int indexed_from) {
188     this->cycle.resize(n + indexed_from);
189     this->first.resize(n + indexed_from);
190
191     this->in_cycle.resize(n + indexed_from, false);
192     this->cycle_id.resize(n + indexed_from, -1);
193     this->id_in_cycle.resize(n + indexed_from, -1);
194     this->near_in_cycle.resize(n + indexed_from);
195     this->id_near_cycle.resize(n + indexed_from);
196     this->cycle_dist.resize(n + indexed_from);
197 }
198
199 public:
200 Functional_Graph(int n, int indexed_from, vector<int> degree,
201                 vector<vector<int>> &adj) {
202     this->allocate(n, indexed_from);
203     this->build(n, indexed_from, degree, adj);
204 }
205
206 // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
207
208 // number of cycles
209 int cycle_cnt = 0;
210 // Vertices present in the i-th cycle.
211 vector<vector<int>> cycle;
212 // first vertex of the i-th cycle
213 vector<int> first;
214
215 // The i-th vertex is present in any cycle?
216 vector<bool> in_cycle;
217 // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
218 // cycle.
219 vector<int> cycle_id;
220 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
221 // belong to any cycle.
222 vector<int> id_in_cycle;
223 // Represents the id of the nearest vertex present in a cycle.
224 vector<int> near_in_cycle;
225 // Represents the id of the nearest cycle.
226 vector<int> id_near_cycle;
227 // Distance to the nearest cycle.
228 vector<int> cycle_dist;
229 // Represent the id of the component of the vertex.
230 // Equal to id_near_cycle

```

```

228     vector<int> &comp = id_near_cycle;
229 };

```

5.18. 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 }

```

5.19. Hld

```

1 class HLD {
2 private:
3     int n;
4     // number of nodes below the i-th node
5     vector<int> sz;
6
7 private:
8     void allocate() {
9         // this->id_in_tree.resize(this->n + 1, -1);
10        this->chain_head.resize(this->n + 1, -1);
11        this->chain_id.resize(this->n + 1, -1);
12        this->sz.resize(this->n + 1);
13        this->parent.resize(this->n + 1, -1);
14        // this->id_in_chain.resize(this->n + 1, -1);
15        // this->chain_size.resize(this->n + 1);
16    }
17
18     int get_sz(const int u, const int p, const vector<vector<int>> &adj) {

```

```

19     this->sz[u] = 1;
20     for (const int v : adj[u]) {
21         if (v == p)
22             continue;
23         this->sz[u] += this->get_sz(v, u, adj);
24     }
25     return this->sz[u];
26 }
27
28 void dfs(const int u, const int id, const int p,
29         const vector<vector<int>> &adj, int &nidx) {
30     // this->id_in_tree[u] = nidx++;
31     this->chain_id[u] = id;
32     // this->id_in_chain[u] = chain_size[id]++;
33     this->parent[u] = p;
34
35     if (this->chain_head[id] == -1)
36         this->chain_head[id] = u;
37
38     int maxx = -1, idx = -1;
39     for (const int v : adj[u]) {
40         if (v == p)
41             continue;
42         if (sz[v] > maxx) {
43             maxx = sz[v];
44             idx = v;
45         }
46     }
47
48     if (idx != -1)
49         this->dfs(idx, id, u, adj, nidx);
50
51     for (const int v : adj[u]) {
52         if (v == idx || v == p)
53             continue;
54         this->dfs(v, this->number_of_chains++, u, adj, nidx);
55     }
56 }
57
58 void build(const int root_idx, const vector<vector<int>> &adj) {
59     this->get_sz(root_idx, -1, adj);
60     int nidx = 0;
61     this->dfs(root_idx, 0, -1, adj, nidx);
62 }
63
64 // int _compute(const int u, Seg_Tree &st) {
65 //     int ans = 0;
66 //     for (int v = u; v != -1; v = parent[chain_head[chain_id[v]]]) {
67 //         // change here
68 //         ans += st.query(id_in_tree[chain_head[chain_id[v]]], id_in_tree[v]);
69 //     }
70 //     return ans;
71 // }
72
73 public:
74     /// Builds the chains.
75     ///
76     /// Time Complexity: O(n)
77     HLD(const int root_idx, const vector<vector<int>> &adj) : n(adj.size()) {
78         allocate();
79         build(root_idx, adj);
80     }
81
82     /// Computes the paths using segment tree.
83     /// Uncomment id_in_tree!!!

```

```

84     ///
85     /// Time Complexity: O(log^2(n))
86     /// int compute(const int u, Seg_Tree &st) { return _compute(u, st); }
87
88     // TAKE CARE, YOU MAY GET MLE!!!
89     // the chains are indexed from 0
90     int number_of_chains = 1;
91     // topmost node of the chain
92     vector<int> chain_head;
93     // id of the node based on the order of the dfs (indexed by 0)
94     vector<int> id_in_tree;
95     // id of the i-th node in his chain
96     vector<int> id_in_chain;
97     // id of the chain that the i-th node belongs
98     vector<int> chain_id;
99     // size of the i-th chain
100    vector<int> chain_size;
101    // parent of the i-th node, -1 for root
102    vector<int> parent;
103 };

```

5.20. 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, kl;
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                         kl = 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 }

```

5.21. Kuhn

```

1  /// Created by viniciustht
2  struct Kuhn {
3      vector<vector<int>> adj;
4      vector<int> matchA, matchB, marcB;
5      int n, m;
6      bool matched = false;
7      Kuhn(int n, int m) : n(n), m(m) {
8          adj.resize(n, vector<int>());
9          matchA.resize(n);
10         matchB = marcB = vector<int>(m);
11     }
12     void add_edge(int u, int v) {
13         adj[u].emplace_back(v);
14         matched = false;
15     }
16     bool dfs(int u) {
17         for (int &v : adj[u]) {
18             if (marcB[v]) // || w > mid) // use with binary search
19                 continue;
20             marcB[v] = 1;
21             if (matchB[v] == -1 or dfs(matchB[v])) {
22                 matchB[v] = u;
23                 matchA[u] = v;
24                 return true;
25             }
26         }
27         return false;
28     }
29     int matching() {
30         memset(matchA.data(), -1, sizeof(int) * n);
31         memset(matchB.data(), -1, sizeof(int) * m);
32         // shuffle(adj.begin(), adj.end(), rng); // se o grafo pode ser esparso
33         // for (auto v : adj)
34         //     shuffle(v.begin(), v.end(), rng);
35         int res = 0;
36         bool aux = true;
37         while (aux) {
38             memset(marcB.data(), 0, sizeof(int) * m);
39             aux = false;
40             for (int i = 0; i < n; i++) {

```

```

42         if (matchA[i] != -1)
43             continue;
44         if (dfs(i)) {
45             res++;
46             aux = true;
47         }
48     }
49     matched = true;
50     return res;
51 }
52 void print_matching() {
53     if (!matched)
54         matching();
55     for (int i = 0; i < n; i++)
56         if (matchA[i] != -1)
57             cerr << i + 1 << " " << matchA[i] + 1 << endl;
58 }
59 };
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 #ifdef DIST
204 /// Returns the distance from a to b. When the graph is unweighted, it is
205 /// considered 1 as the weight of the edges.
206 ///
207 /// Time Complexity: O(log(n))
208 int query_dist(const int a, const int b) {
209     assert(indexed_from <= min(a, b)),
210     assert(max(a, b) < this->n + indexed_from);
211     return this->dist[a] + this->dist[b] - 2 * this->dist[this->get_lca(a,
212     b)];
213 }
214 #endif
215
216 #ifdef 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 // clang-format on

```

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

```
1 |IS_maximal| = |V| - MAXIMUM_MATCHING
```

5.24. Maximum Path Unweighted Graph

```

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

```

5.25. Min Cost Flow

```

1 /// Code copied from:
2 ///
3     github.com/kth-competitive-programming/kactl/blob/master/content/graph/MinCostMax
4 #include <bits/extc++.h> /// include-line, keep-include
5
6 // #define all(x) begin(x), end(x)
7 // typedef pair<int, int> ii;
8 // typedef vector<int> vi;
9 typedef vector<ll> VL;
10 typedef long long ll;
11 #define sz(x) (int)(x).size()
12 #define rep(i, a, b) for (int i = a; i < (b); ++i)
13
14 const ll INF = numeric_limits<ll>::max() / 4;
15
16 // clang-format off
17 struct MCMF {
18     int N;
19     vector<vi> ed, red;
20     vector<VL> cap, flow, cost;
21     vi seen;

```

```

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

```

```

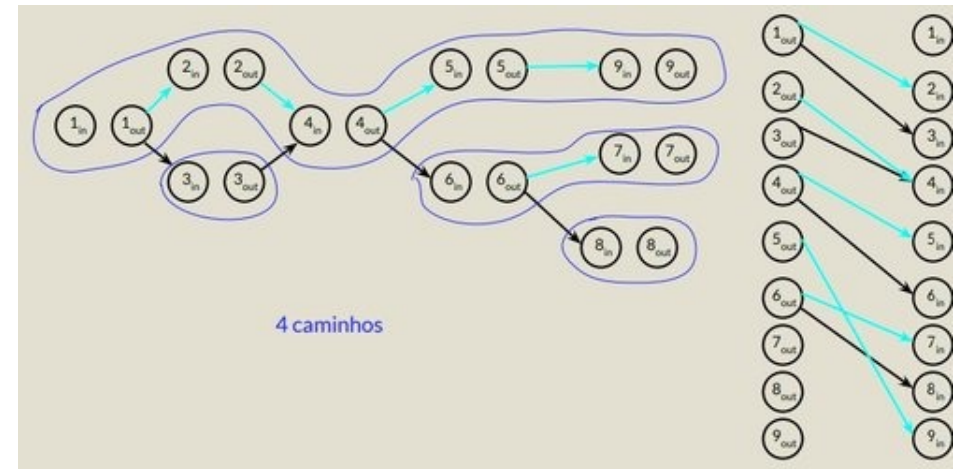
86     for (int to : ed[i]) if (cap[i][to])
87         if ((v = pi[i] + cost[i][to]) < pi[to])
88             pi[to] = v, ch = 1;
89     assert(it >= 0); // negative cost cycle
90 }
91 };
92 // clang-format on

```

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

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

5.27. Minimum Path Cover In Dag



5.28. 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.

5.29. 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 }

```

5.30. Number Of Different Spanning Trees In A Complete Graph

```

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

```

5.31. Number Of Ways To Make A Graph Connected

```

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

```

5.32. 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 }

```

5.33. Pruffer Encode

```

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

```

5.34. 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**.

5.35. 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.

5.36. Scc (Kosaraju)

```

1 class SCC {
2     private:
3         // number of vertices
4         int n;
5         // indicates whether it is indexed from 0 or 1
6         int indexed_from;
7         // reversed graph
8         vector<vector<int>> trans;
9
10    private:
11    void dfs_trans(int u, int id) {
12        comp[u] = id;
13        scc[id].push_back(u);
14
15        for (int v: trans[u])
16            if (comp[v] == -1)
17                dfs_trans(v, id);
18    }
19
20    void get_transpose(vector<vector<int>>& adj) {
21        for (int u = indexed_from; u < this->n + indexed_from; u++)
22            for (int v: adj[u])
23                trans[v].push_back(u);
24    }
25
26    void dfs_fill_order(int u, stack<int> &s, vector<vector<int>>& adj) {
27        comp[u] = true;
28
29        for (int v: adj[u])
30            if (!comp[v])
31                dfs_fill_order(v, s, adj);
32
33        s.push(u);
34    }
35
36    // The main function that finds all SCCs
37    void compute_SCC(vector<vector<int>>& adj) {
38
39        stack<int> s;
40        // Fill vertices in stack according to their finishing times
41        for (int i = indexed_from; i < this->n + indexed_from; i++)
42            if (!comp[i])
43                dfs_fill_order(i, s, adj);
44
45        // Create a reversed graph
46        get_transpose(adj);
47
48        fill(comp.begin(), comp.end(), -1);
49
50        // Now process all vertices in order defined by stack
51        while (s.empty() == false) {
52            int v = s.top();
53            s.pop();
54
55            if (comp[v] == -1)
56                dfs_trans(v, this->number_of_comp++);

```

```

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

```

5.37. Topological Sort

```

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

```

5.38. 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

```

5.39. Tree Distance

```

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

```

```

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

```

5.40. Tree Isomorphism

```

1 /// THE VALUES OF THE VERTICES MUST BELONG FROM 1 TO N.
2 namespace tree {
3   mt1937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
4
5   vector<uint64_t> base;
6   uint64_t build(const int u, const int p, const vector<vector<int>> &adj,
7                 const int level = 0) {
8     if (level == base.size())
9       base.emplace_back(rng());
10    uint64_t hsh = 1;
11    vector<uint64_t> child;
12    for (const int v : adj[u])
13      if (v != p)
14        child.emplace_back(build(v, u, adj, level + 1));
15    sort(child.begin(), child.end());
16    for (const uint64_t x : child)
17      hsh = hsh * base[level] + x;
18    return hsh;
19  }
20
21  /// Returns whether two rooted trees are isomorphic or not.
22  ///
23  /// Time Complexity: O(n)
24  bool same(const int root_1, const vector<vector<int>> &adj1, const int
25           root_2,
26           const vector<vector<int>> &adj2) {
27    if (adj1.size() != adj2.size())
28      return false;
29    return build(root_1, -1, adj1) == build(root_2, -1, adj2);
30  }
31
32  /// Returns whether two non-rooted trees are isomorphic or not.
33  /// REQUIRES centroid.cpp!!!
34  ///
35  /// Time Complexity: O(n)
36  bool same(const int n, const int indexed_from, const vector<vector<int>>
37           &adj1,
38           const vector<vector<int>> &adj2) {
39    vector<int> c1 = centroid(n, indexed_from, adj1),
40              c2 = centroid(n, indexed_from, adj2);
41    for (const int v : c2)
42      if (same(c1.front(), adj1, v, adj2))
43        return true;
44    return false;
45  }
46 } // namespace tree

```

6. Language Stuff

6.1. Climits

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

6.2. Checagem E Transformacao De Caractere

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

6.3. Conta Digitos 1 Ate N

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

6.4. Escrita Em Arquivo

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

6.5. Gcd

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

6.6. Hipotenusa

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

6.7. Int To Binary String

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

6.8. Int To String

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

6.9. Leitura De Arquivo

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

6.10. Max E Min Element Num Vetor

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

6.11. Permutacao

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

6.12. Remove Repeticoes Continuas Num Vetor

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

6.13. Rotate (Left)

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

6.14. Rotate (Right)

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

6.15. Scanf De Uma String

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

6.16. Split Function

```
1 /// Splits a string into a vector. A separator can be specified
2 /// EX: str=A-B-C -> split -> x = {A,B,C}
3 vector<string> split(const string &s, char separator = ' ') {
4     stringstream ss(s);
5     string item;
6     vector<string> tokens;
7     while (getline(ss, item, separator))
8         tokens.emplace_back(item);
9     return tokens;
10 }
```

```

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

```

6.17. String To Long Long

```

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

```

6.18. Substring

```

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

```

6.19. Width

```

1 cout << width(13);
2 cout << 100 << endl; // "      100      "
3 cout.fill('x');
4 cout.width(13);
5 cout << 100 << endl; // "xxxxx100xxxxx"
6 cout << right << 100 << endl; "xxxxxxxx100"

```

6.20. Binary String To Int

```

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

```

6.21. Check

```

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

```

6.22. Check Overflow

```

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

```

```

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

```

6.23. Counting Bits

```

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

```

6.24. 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 }

```

6.25. Random Numbers

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

6.26. 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 }
```

6.27. Time Measure

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

7. Math

7.1. Bell Numbers

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

7.2. Binary Exponentiation

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

```
12     }
13
14     return ans;
15 }
```

7.3. Chinese Remainder Theorem

```
1 int inv(int a, int m) {
2     int m0 = m, t, q;
3     int x0 = 0, x1 = 1;
4
5     if (m == 1)
6         return 0;
7
8     // Apply extended Euclid Algorithm
9     while (a > 1) {
10        // q is quotient
11        if (m == 0)
12            return INF;
13        q = a / m;
14        t = m;
15        // m is remainder now, process same as euclid's algo
16        m = a % m, a = t;
17        t = x0;
18        x0 = x1 - q * x0;
19        x1 = t;
20    }
21
22    // Make x1 positive
23    if (x1 < 0)
24        x1 += m0;
25
26    return x1;
27 }
28
29 // k is size of num[] and rem[]. Returns the smallest
30 // number x such that:
31 // x % num[0] = rem[0],
32 // x % num[1] = rem[1],
33 // .....
34 // x % num[k-2] = rem[k-1]
35 // Assumption: Numbers in num[] are pairwise coprimes
36 // (gcd for every pair is 1)
37 int findMinX(const vector<int> &num, const vector<int> &rem, const int k) {
38     // Compute product of all numbers
39     int prod = 1;
40     for (int i = 0; i < k; i++)
41         prod *= num[i];
42
43     int result = 0;
44
45     // Apply above formula
46     for (int i = 0; i < k; i++) {
47         int pp = prod / num[i];
48         int iv = inv(pp, num[i]);
49         if (iv == INF)
50             return INF;
51         result += rem[i] * inv(pp, num[i]) * pp;
52     }
53
54     // IF IS NOT VALID RETURN INF
55     return (result % prod == 0 ? INF : result % prod);
56 }
```

7.4. Combinatorics

```

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

```

```

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

```

7.5. Diophantine Equation

```

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

```

7.6. Divisors

```

1 /// OBS: Each number has at most  $\sqrt[3]{N}$  divisors
2 /// THE NUMBERS ARE NOT SORTED!!!
3 ///
4 /// Time Complexity: O(sqrt(n))
5 vector<int> divisors(int n) {
6     vector<int> ans;
7     for (int i = 1; i * i <= n; i++) {
8         if (n % i == 0) {

```

```

9     if (n / i == i)
10         ans.emplace_back(i);
11     else
12         ans.emplace_back(i), ans.emplace_back(n / i);
13     }
14 }
15 // sort(ans.begin(), ans.end());
16 return ans;
17 }

```

7.7. Euler Totient

```

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

```

7.8. Extended Euclidean

```

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

```

7.9. Factorization

```

1 /// Factorizes a number.
2 ///
3 /// Time Complexity: O(sqrt(n))
4 map<int, int> factorize(int n) {
5     map<int, int> fat;
6     while (n % 2 == 0) {
7         ++fat[2];
8         n /= 2;
9     }
10
11     for (int i = 3; i * i <= n; i += 2) {
12         while (n % i == 0) {

```

```

13         ++fat[i];
14         n /= i;
15     }
16     /* OBS1
17        IF (N < 1E7)
18            you can optimize by factoring with SPF
19        */
20 }
21 if (n > 2)
22     ++fat[n];
23 return fat;
24 }

```

7.10. Inclusion Exclusion

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

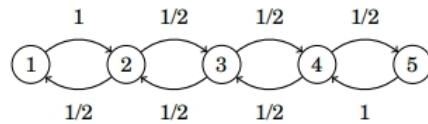
7.11. Inclusion Exclusion

```

1 // |A ∪ B ∪ C| = |A| + |B| + |C| - |A ∩ B| - |A ∩ C| - |B ∩ C| + |A ∩ B ∩ C|
2 // EXAMPLE: How many numbers from 1 to 10^9 are multiple of 42, 54, 137 or
3 // 201?
4 int f(const vector<int> &arr, const int LIMIT) {
5     int n = arr.size();
6     int c = 0;
7
8     for (int mask = 1; mask < (1ll << n); mask++) {
9         int lcm = 1;
10        for (int i = 0; i < n; i++)
11            if (mask & (1ll << i))
12                lcm = lcm * arr[i] / __gcd(lcm, arr[i]);
13        // if the number of element is odd, then add
14        if (__builtin_popcount_ll(mask) % 2 == 1)
15            c += LIMIT / lcm;
16        else // otherwise subtract
17            c -= LIMIT / lcm;
18    }
19    return LIMIT - c;
20 }

```

7.12. Markov Chains



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

Probably after moving 1 step from 1

7.13. Matrix Exponentiation

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

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

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

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

Fibonacci

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

7.14. Matrix Exponentiation

```
1 // USE #define int long long!!!
2 struct Matrix {
3     static constexpr int MOD = 1e9 + 7;
4
5     // static matrix, if it's created multiple times, it's recommended
```

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

7.15. Pollard Rho (Find A Divisor)

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

```

7.16. 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 }

```

7.17. 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 }

```

7.18. 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

```

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

```

7.20. Stars And Bars

I. positive integers x_i

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

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

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

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

II. non-negative integers x_i

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

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

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

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

8. Miscellaneous

8.1. 2-Sat

```

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

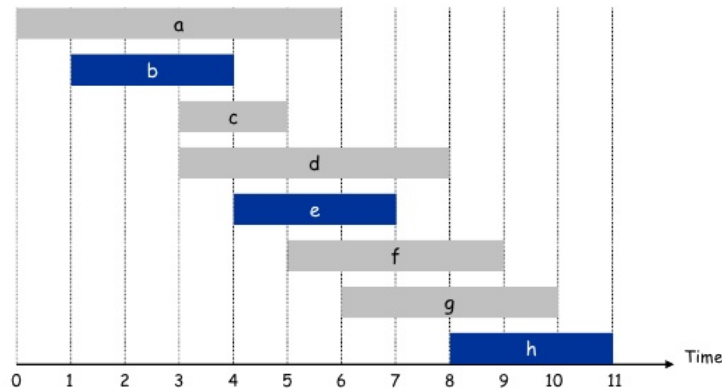
```

```

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

```

8.2. Interval Scheduling



8.3. Interval Scheduling

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

8.4. Oito Rainhas

```

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

```

```

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

```

8.5. Sliding Window Minimum

```

1 // minimo num vetor arr de arr[0] ... arr[k-1], arr[l] ... arr[k], arr[2]
    ... arr[k+1]
2
3 void swma(vector<int> arr, int k) {
4     deque<int> window;
5     for (int i = 0; i < arr.size(); i++) {
6         while (!window.empty() && window.back().ff > arr[i])
7             window.pop_back();
8         window.pb(ii(arr[i], i));
9         while (window.front().ss <= i - k)
10             window.pop_front();
11
12         if (i >= k)
13             cout << ' ';
14         if (i - k + 1 >= 0)
15             cout << window.front().ff;
16     }
17 }

```

8.6. Torre De Hanoi

```

1 #include <stdio.h>
2
3 // C recursive function to solve tower of hanoi puzzle
4 void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
5     if (n == 1) {
6         printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
7         return;
8     }

```

```

9   towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
10  printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
11  towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
12  }
13
14  int main() {
15      int n = 4; // Number of disks
16      towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
17      return 0;
18  }

```

8.7. Counting Frequency Of Digits From 1 To K

```

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

```

8.8. Infix To Postfix

```

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

```

```

36         op.push(c);
37     }
38 }
39 while (!op.empty()) {
40     postfix += op.top();
41     op.pop();
42 }
43 return postfix;
44 }

```

8.9. Kadane

```

1  /// Returns the maximum contiguous sum in the array.
2  ///
3  /// Time Complexity: O(n)
4  int kadane(vector<int> &arr) {
5      if (arr.empty())
6          return 0;
7      int sum, tot;
8      sum = tot = arr[0];
9
10     for (int i = 1; i < arr.size(); i++) {
11         sum = max(arr[i], arr[i] + sum);
12         if (sum > tot)
13             tot = sum;
14     }
15     return tot;
16 }

```

8.10. Kadane (Segment Tree)

```

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

```

```

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

```

8.11. Kadane 2D

```

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

```

```

24         sum += arr[i];
25         if (sum < 0) {
26             sum = 0;
27             local_start = i+1;
28         }
29         else if (sum > maxSum){
30             maxSum = sum;
31             *start = local_start;
32             *finish = i;
33         }
34     }
35
36     // There is at-least one non-negative number
37     if (*finish != -1)
38         return maxSum;
39
40     // Special Case: When all numbers in arr[] are negative
41     maxSum = arr[0];
42     *start = *finish = 0;
43
44     // Find the maximum element in array
45     for (i = 1; i < n; i++) {
46         if (arr[i] > maxSum) {
47             maxSum = arr[i];
48             *start = *finish = i;
49         }
50     }
51     return maxSum;
52 }
53
54 // The main function that finds maximum sum rectangle in mat[][]
55 int findMaxSum() {
56     // Variables to store the final output
57     int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;
58
59     int left, right, i;
60     int temp[ROW], sum, start, finish;
61
62     // Set the left column
63     for (left = 0; left < COL; ++left) {
64         // Initialize all elements of temp as 0
65         for(int i = 0; i < ROW; i++)
66             temp[i] = 0;
67
68         // Set the right column for the left column set by outer loop
69         for (right = left; right < COL; ++right) {
70             // Calculate sum between current left and right for every row 'i'
71             for (i = 0; i < ROW; ++i)
72                 temp[i] += mat[i][right];
73
74             // Find the maximum sum subarray in temp[]. The kadane()
75             // function also sets values of start and finish. So 'sum' is
76             // sum of rectangle between (start, left) and (finish, right)
77             // which is the maximum sum with boundary columns strictly as
78             // left and right.
79             sum = kadane(temp, &start, &finish, ROW);
80
81             // Compare sum with maximum sum so far. If sum is more, then
82             // update maxSum and other output values
83             if (sum > maxSum) {
84                 maxSum = sum;
85                 finalLeft = left;
86                 finalRight = right;
87                 finalTop = start;
88                 finalBottom = finish;

```

```

89     }
90     }
91 }
92
93 return maxSum;
94 // Print final values
95 printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
96 printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
97 printf("Max sum is: %d\n", maxSum);
98 }

```

8.12. Largest Area In Histogram

```

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

```

8.13. Modular Integer

```

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

```

```

28
29 template <typename T> modular(const T value) {
30     if (value >= 0)
31         this->value = ((uint)value < M ? value : (uint)value % M);
32     else {
33         uint abs_value = (-(uint)value) % M;
34         this->value = (abs_value == 0 ? 0 : M - abs_value);
35     }
36 }
37
38 template <typename T> explicit operator T() const { return value; }
39
40 modular operator-() const { return modular(value == 0 ? 0 : M - value); }
41
42 modular &operator+=(const modular &rhs) {
43     if (rhs.value >= M - value)
44         value = rhs.value - (M - value);
45     else
46         value += rhs.value;
47     return *this;
48 }
49
50 modular &operator-=(const modular &rhs) {
51     if (rhs.value > value)
52         value = M - (rhs.value - value);
53     else
54         value -= rhs.value;
55     return *this;
56 }
57
58 modular &operator*=(const modular &rhs) {
59     value = (uint64_t)value * rhs.value % M;
60     return *this;
61 }
62
63 modular &operator/=(const modular &rhs) { return *this *= inverse(rhs); }
64
65 /// Computes pow(b, e) % M.
66 ///
67 /// Time Complexity: O(log(e)).
68 friend modular exp(modular b, uint e) {
69     modular res = 1;
70     for (; e > 0; e >= 1) {
71         if (e & 1)
72             res *= b;
73         b *= b;
74     }
75     return res;
76 }
77
78 /// Computes the modular multiplicative inverse of a with mod M.
79 ///
80 /// Time Complexity: O(log(a)).
81 friend modular inverse(const modular &a) {
82     assert(a.value > 0);
83     auto aux = extended_gcd(a.value, M);
84     assert(get<0>(aux) == 1); // a and M must be coprimes.
85     return modular(get<1>(aux));
86 }
87
88 friend modular operator+(modular lhs, const modular &rhs) {
89     return lhs += rhs;
90 }
91
92 friend modular operator-(modular lhs, const modular &rhs) {

```

```

93     return lhs -= rhs;
94 }
95
96 friend modular operator*(modular lhs, const modular &rhs) {
97     return lhs *= rhs;
98 }
99
100 friend modular operator/(modular lhs, const modular &rhs) {
101     return lhs /= rhs;
102 }
103
104 friend bool operator==(const modular &lhs, const modular &rhs) {
105     return lhs.value == rhs.value;
106 }
107
108 friend bool operator!=(const modular &lhs, const modular &rhs) {
109     return !(lhs == rhs);
110 }
111
112 friend string to_string(const modular &a) { return to_string(a.value); }
113
114 friend ostream &operator<<(ostream &lhs, const modular &rhs) {
115     return lhs << to_string(rhs);
116 }
117 };
118
119 using mint = modular<MOD>;

```

8.14. Point Compression

```

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

```

8.15. Ternary Search

```

1 /// Returns the index in the array which contains the minimum element. In
2 case
3 /// of draw, it returns the first occurrence. The array should, first,
4 decrease,
5 /// then increase.
6 ///
7 /// Time Complexity: O(log3(n))
8 int ternary_search(const vector<int> &arr) {
9     int l = 0, r = (int)arr.size() - 1;
10     while (r - l > 2) {
11         int lc = l + (r - l) / 3;
12         int rc = r - (r - l) / 3;
13         // the function f(x) returns the element on the position x

```

```

12     if (f(lc) > f(rc))
13         // the function is going down, then the middle is on the right.
14         l = rc;
15     else
16         r = rc;
17 }
18 // the range [l, r] contains the minimum element.
19
20 int minn = f(l), idx = l;
21 for (int i = l + 1; i <= r; ++i)
22     if (f(i) < minn) {
23         idx = i;
24         minn = f(i);
25     }
26
27 return idx;
28 }

```

9. 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 l + n;
126 }
127
128 vector<int> vec(const int n, const int l, const int r) {
129     vector<int> arr(n);
130     for (int &x : arr)
131         x = num(l, r);
132     return arr;
133 }
134
135 vector<vector<int>> matrix(const int n, const int m, const int l, const int
    r) {
136     vector<vector<int>> mt;
137     for (int i = 0; i < n; ++i)
138         mt.emplace_back(vec(m, l, r));
139     return mt;
140 }
141
142 string str(const int n, const int c = 26) {

```

```

143 string ans;
144 for (int i = 0; i < n; ++i)
145     ans += char(rng() % c + 'a');
146 return ans;
147 }
148
149 string str01(const int n) {
150     string ans;
151     for (int i = 0; i < n; ++i) {
152         ans += char(rng() % 2 + '0');
153     }
154     return ans;
155 }
156
157 string spaced_str(const int n, const int size, const int c = 26) {
158     string ans;
159     for (int i = 0; i < size; ++i) {
160         if (i)
161             ans += ' ';
162         ans += str(n, c);
163     }
164     return ans;
165 }

```

9.3. Run

```

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

```

10. Strings

10.1. Trie - Maximum Xor Sum

```

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

```

10.2. Trie - Maximum Xor Two Elements

1 1. Dada uma trie de números binários e um numero X, tente achar o número máximo que resultante da operação XOR

2

3 Ex: Para o número 10(=(1010)2), o número que resulta no **xor** máximo é (0101)2, tente acha-lo na trie.

10.3. Z-Function

```

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

```

10.4. Aho Corasick

```

1 /// REQUIRES trie.cpp
2
3 class Aho {
4 private:
5     // node of the output list
6     struct Out_Node {
7         vector<int> str_idx;
8         Out_Node *next = nullptr;
9     };
10
11     vector<Trie::Node *> fail;
12     Trie trie;
13     // list of nodes of output
14     vector<Out_Node *> out_node;
15     const vector<string> arr;
16 }

```



```

17  /// Time Complexity: O(number of characters in arr)
18  void build_trie() {
19      const int n = arr.size();
20      int node_cnt = 1;
21
22      for (int i = 0; i < n; ++i)
23          node_cnt += arr[i].size();
24
25      out_node.reserve(node_cnt);
26      for (int i = 0; i < node_cnt; ++i)
27          out_node.push_back(new Out_Node());
28
29      fail.resize(node_cnt);
30      for (int i = 0; i < n; ++i) {
31          const int id = trie.insert(arr[i]);
32          out_node[id]->str_idx.push_back(i);
33      }
34
35      this->build_failures();
36  }
37
38  /// Returns the fail node of cur.
39  Trie::Node *find_fail_node(Trie::Node *cur, char c) {
40      while (cur != this->trie.root() && !cur->next.count(c))
41          cur = fail[cur->id];
42      // if cur is pointing to the root node and c is not a child
43      if (!cur->next.count(c))
44          return trie.root();
45      return cur->next[c];
46  }
47
48  /// Time Complexity: O(number of characters in arr)
49  void build_failures() {
50      queue<const Trie::Node *> q;
51
52      fail[trie.root()->id] = trie.root();
53      for (const pair<char, Trie::Node *> v : trie.root()->next) {
54          q.emplace(v.second);
55          fail[v.second->id] = trie.root();
56          out_node[v.second->id]->next = out_node[trie.root()->id];
57      }
58
59      while (!q.empty()) {
60          const Trie::Node *u = q.front();
61          q.pop();
62
63          for (const pair<char, Trie::Node *> x : u->next) {
64              const char c = x.first;
65              const Trie::Node *v = x.second;
66              Trie::Node *fail_node = find_fail_node(fail[u->id], c);
67              fail[v->id] = fail_node;
68
69              if (!out_node[fail_node->id]->str_idx.empty())
70                  out_node[v->id]->next = out_node[fail_node->id];
71              else
72                  out_node[v->id]->next = out_node[fail_node->id]->next;
73
74              q.emplace(v);
75          }
76      }
77  }
78
79  vector<vector<pair<int, int>>> aho_find_occurrences(const string &text) {
80      vector<vector<pair<int, int>>> ans(arr.size());
81      Trie::Node *cur = trie.root();

```

```

82
83      for (int i = 0; i < text.size(); ++i) {
84          cur = find_fail_node(cur, text[i]);
85          for (Out_Node *node = out_node[cur->id]; node != nullptr;
86               node = node->next)
87              for (const int idx : node->str_idx)
88                  ans[idx].emplace_back(i - (int)arr[idx].size() + 1, i);
89      }
90      return ans;
91  }
92
93 public:
94     /// Constructor that builds the trie and the failures.
95     ///
96     /// Time Complexity: O(number of characters in arr)
97     Aho(const vector<string> &arr) : arr(arr) { this->build_trie(); }
98
99     /// Searches in text for all occurrences of all strings in array arr.
100    ///
101    /// Time Complexity: O(text.size() + number of characters in arr)
102    vector<vector<pair<int, int>>> find_occurrences(const string &text) {
103        return this->aho_find_occurrences(text);
104    }
105    };

```

10.5. Hashing

```

1  // Global vector used in the class.
2  vector<int> hash_base;
3
4  class Hash {
5      /// Prime numbers to be used in mod operations
6      const vector<int> m = {1000000007, 1000000009};
7
8      vector<vector<int>> hash_table;
9      vector<vector<int>> pot;
10     // size of the string
11     const int n;
12
13 private:
14     static int mod(int n, int m) {
15         n %= m;
16         if (n < 0)
17             n += m;
18         return n;
19     }
20
21     /// Time Complexity: O(1)
22     pair<int, int> hash_query(const int l, const int r) {
23         vector<int> ans(m.size());
24
25         if (l == 0) {
26             for (int i = 0; i < m.size(); i++)
27                 ans[i] = hash_table[i][r];
28         } else {
29             for (int i = 0; i < m.size(); i++)
30                 ans[i] =
31                     mod((hash_table[i][r] - hash_table[i][l - 1] * pot[i][r - l +
32                     1])),
33                     m[i]);
34         }
35
36         return {ans.front(), ans.back()};
37     }

```

```

37 // Time Complexity: O(m.size())
38 void build_base() {
39     if (!hash_base.empty())
40         return;
41     random_device rd;
42     mt19937 gen(rd());
43     uniform_int_distribution<int> distribution(CHAR_MAX, INT_MAX);
44     hash_base.resize(m.size());
45     for (int i = 0; i < hash_base.size(); ++i)
46         hash_base[i] = distribution(gen);
47 }
48
49 // Time Complexity: O(n)
50 void build_table(const string &s) {
51     pot.resize(m.size(), vector<int>(this->n));
52     hash_table.resize(m.size(), vector<int>(this->n));
53
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 // Time Complexity: O(n)
69 Hash(const string &s) : n(s.size()) {
70     build_base();
71     build_table(s);
72 }
73
74 // Returns the hash from l to r.
75 // Time Complexity: O(1) -> Actually O(number_of_primes)
76 pair<int, int> query(const int l, const int r) {
77     assert(0 <= l), assert(1 <= r), assert(r < this->n);
78     return hash_query(l, r);
79 }
80 };
81
82

```

10.6. Kmp

```

1 // Builds the pi array for the KMP algorithm.
2 // Time Complexity: O(n)
3 vector<int> pi(const string &pat) {
4     vector<int> ans(pat.size() + 1, -1);
5     int i = 0, j = -1;
6     while (i < pat.size()) {
7         while (j >= 0 && pat[i] != pat[j])
8             j = ans[j];
9         ++i, ++j;
10        ans[i] = j;
11    }
12    return ans;
13 }
14

```

```

15 // Returns the occurrences of a pattern in a text.
16 // Time Complexity: O(n + m)
17 vector<int> kmp(const string &txt, const string &pat) {
18     vector<int> p = pi(pat);
19     vector<int> ans;
20
21     for (int i = 0, j = 0; i < txt.size(); ++i) {
22         while (j >= 0 && pat[j] != txt[i])
23             j = p[j];
24         if (++j == pat.size()) {
25             ans.emplace_back(i);
26             j = p[j];
27         }
28     }
29     return ans;
30 }
31
32

```

10.7. Lcs K Strings

```

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

```

```

43 int cnt = 0;
44 for(string &x: arr) {
45     if(concat.size())
46         concat += (char)cnt;
47     concat += x;
48 }
49
50 cnt = 0;
51 for(int i = 0; i < concat.size(); i++) {
52     ind[i + 1] = cnt;
53     if(concat[i] < MaximumNumberOfStrings)
54         cnt++;
55 }
56
57 Suffix_Array say(concat);
58 vector<int> sa = say.get_suffix_array();
59 Sparse_Table spt(say.get_lcp());
60
61 vector<int> freq(n);
62 int cnt1 = 0;
63
64 /// Ignore separators
65 int i = n, j = n - 1;
66 int ans = 0;
67
68 while(true) {
69
70     if(cnt1 == n) {
71
72         ans = max(ans, spt.query(i, j - 1));
73
74         int idx = ind[sa[i]];
75         freq[idx]--;
76         if(freq[idx] == 0)
77             cnt1--;
78         i++;
79     } else if(j == (int)sa.size() - 1)
80         break;
81     else {
82         j++;
83         int idx = ind[sa[j]];
84         freq[idx]++;
85         if(freq[idx] == 1)
86             cnt1++;
87     }
88 }
89
90 cout << ans << endl;
91 }

```

10.8. Lexicographically Smallest Rotation

```

1 int booth(string &s) {
2     s += s;
3     int n = s.size();
4
5     vector<int> f(n, -1);
6     int k = 0;
7     for(int j = 1; j < n; j++) {
8         int sj = s[j];
9         int i = f[j - k - 1];
10        while(i != -1 && sj != s[k + i + 1]) {
11            if(sj < s[k + i + 1])
12                k = j - i - 1;

```

```

13         i = f[i];
14     }
15     if(sj != s[k + i + 1]) {
16         if(sj < s[k])
17             k = j;
18         f[j - k] = -1;
19     }
20     else
21         f[j - k] = i + 1;
22 }
23 return k;
24 }

```

10.9. Manacher (Longest Palindrome)

```

1 // https://medium.com/hackernoon/manachers-algorithm-explained-longest-palindromic-s
2
3 /// Create a string containing '#' characters between any two characters.
4 string get_modified_string(string &s) {
5     string ret;
6     for(int i = 0; i < s.size(); i++){
7         ret.push_back('#');
8         ret.push_back(s[i]);
9     }
10    ret.push_back('#');
11    return ret;
12 }
13
14 /// Returns the first occurrence of the longest palindrome based on the lps
15    array.
16 /// Time Complexity: O(n)
17 string get_best(const int max_len, const string &str, const vector<int>
18    &lps) {
19     for(int i = 0; i < lps.size(); i++) {
20         if(lps[i] == max_len) {
21             string ans;
22             int cnt = max_len / 2;
23             int io = i - 1;
24             while(cnt) {
25                 if(str[io] != '#') {
26                     ans += str[io];
27                     cnt--;
28                 }
29                 io--;
30             }
31             reverse(ans.begin(), ans.end());
32             if(str[i] != '#')
33                 ans += str[i];
34             cnt = max_len / 2;
35             io = i + 1;
36             while(cnt) {
37                 if(str[io] != '#') {
38                     ans += str[io];
39                     cnt--;
40                 }
41                 io++;
42             }
43             return ans;
44         }
45     }
46 }

```

```

47 /// Returns a pair containing the size of the longest palindrome and the
    first occurrence of it.
48 ///
49 /// Time Complexity: O(n)
50 pair<int, string> manacher(string &s) {
51     int n = s.size();
52     string str = get_modified_string(s);
53     int len = (2 * n) + 1;
54     //the i-th index contains the longest palindromic substring with the i-th
        char as the center
55     vector<int> lps(len);
56     int c = 0; //stores the center of the longest palindromic substring until
        now
57     int r = 0; //stores the right boundary of the longest palindromic
        substring until now
58     int max_len = 0;
59     for(int i = 0; i < len; i++) {
60         //get mirror index of i
61         int mirror = (2 * c) - i;
62
63         //see if the mirror of i is expanding beyond the left boundary of
        current longest palindrome at center c
64         //if it is, then take r - i as lps[i]
65         //else take lps[mirror] as lps[i]
66         if(i < r)
67             lps[i] = min(r - i, lps[mirror]);
68
69         //expand at i
70         int a = i + (1 + lps[i]);
71         int b = i - (1 + lps[i]);
72         while(a < len && b >= 0 && str[a] == str[b]) {
73             lps[i]++;
74             a++;
75             b--;
76         }
77
78         //check if the expanded palindrome at i is expanding beyond the right
        boundary of current longest palindrome at center c
79         //if it is, the new center is i
80         if(i + lps[i] > r) {
81             c = i;
82             r = i + lps[i];
83
84             if(lps[i] > max_len) //update max_len
85                 max_len = lps[i];
86         }
87     }
88
89     return make_pair(max_len, get_best(max_len, str, lps));
90 }

```

10.10. Suffix Array

```

1 // To use the compare method use the macro below.
2 // #define BUILD_TABLE
3
4 namespace RadixSort {
5     /// Sorts the array arr stably in ascending order.
6     ///
7     /// Time Complexity: O(n + max_element)
8     /// Space Complexity: O(n + max_element)
9     template <typename T>
10     void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),
11             const int begin = 0) {

```

```

12     const int n = arr.size();
13     vector<T> new_order(n);
14     vector<int> count(max_element + 1, 0);
15
16     for (int i = begin; i < n; ++i)
17         ++count[get_key(arr[i])];
18
19     for (int i = 1; i <= max_element; ++i)
20         count[i] += count[i - 1];
21
22     for (int i = n - 1; i >= begin; --i) {
23         new_order[count[get_key(arr[i])] - (begin == 0)] = arr[i];
24         --count[get_key(arr[i])];
25     }
26
27     arr = move(new_order);
28 }
29
30 /// Sorts an array by their pair of ranks stably in ascending order.
31 template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
32     // sort by the second rank
33     RadixSort::sort<T>(
34         arr, rank_size, [](T &item) { return item.first.second; }, 0);
35
36     // sort by the first rank
37     RadixSort::sort<T>(
38         arr, rank_size, [](T &item) { return item.first.first; }, 0);
39 }
40 } // namespace RadixSort
41
42 // clang-format off
43 /// It is indexed by 0.
44 /// Let the given string be "banana".
45 ///
46 /// 0 banana          5 a
47 /// 1 anana          Sort the Suffixes  3 ana
48 /// 2 nana           ----->          1 anana
49 /// 3 ana            alphabetically    0 banana
50 /// 4 na              4 na
51 /// 5 a              2 nana
52 /// So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
53 ///
54 /// LCP
55 /// 1 a
56 /// 3 ana
57 /// 0 anana
58 /// 0 banana
59 /// 2 na
60 /// 0 nana (The last position will always be zero)
61 /// So the LCP for "banana" is {1, 3, 0, 0, 2, 0}
62 class Suffix_Array {
63 private:
64     const string s;
65     const int n;
66
67     typedef pair<int, int> Rank;
68
69     #ifdef BUILD_TABLE
70     vector<vector<int>> rank_table;
71     const vector<int> log_array = build_log_array();
72     #endif
73 public:
74     Suffix_Array(const string &s) : n(s.size()), s(s) {}
75
76 private:

```

```

77 vector<int> build_log_array() {
78     vector<int> log_array(this->n + 1, 0);
79     for (int i = 2; i <= this->n; ++i)
80         log_array[i] = log_array[i / 2] + 1;
81     return log_array;
82 }
83
84 static void build_ranks(const vector<pair<Rank, int>> &ranks,
85                        vector<int> &ret) {
86     // The vector containing the ranks will be present at ret
87     ret[ranks[0].second] = 1;
88     for (int i = 1; i < ranks.size(); ++i) {
89         // if their rank are equal, than their position should be the same
90         if (ranks[i - 1].first == ranks[i].first)
91             ret[ranks[i].second] = ret[ranks[i - 1].second];
92         else
93             ret[ranks[i].second] = ret[ranks[i - 1].second] + 1;
94     }
95 }
96
97 /// Time Complexity: O(n*log(n))
98 vector<int> build_suffix_array() {
99     // the tuple below represents the rank and the index associated with it
100    vector<pair<Rank, int>> ranks(this->n);
101    vector<int> arr(this->n);
102
103    for (int i = 0; i < n; ++i)
104        ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);
105
106    #ifdef BUILD_TABLE
107        int rank_table_size = 0;
108        this->rank_table.resize(log_array[this->n] + 2);
109    #endif
110    RadixSort::sort_pairs(ranks, 256);
111    build_ranks(ranks, arr);
112
113    {
114        int jump = 1;
115        int max_rank = arr[ranks.back().second];
116
117        // it will be compared intervals a pair of intervals (i, jump-1), (i +
118        // jump, i + 2*jump - 1). The variable jump is always a power of 2
119        #ifdef BUILD_TABLE
120            while (jump / 2 < this->n) {
121                #else
122                while (max_rank != this->n) {
123                    #endif
124                    for (int i = 0; i < this->n; ++i) {
125                        ranks[i].first.first = arr[i];
126                        ranks[i].first.second = (i + jump < this->n ? arr[i + jump] : 0);
127                        ranks[i].second = i;
128                    }
129
130                    #ifdef BUILD_TABLE
131                    // inserting only the ranks in the table
132                    transform(ranks.begin(), ranks.end(),
133                            back_inserter(rank_table[rank_table_size++]),
134                            [](pair<Rank, int> &pair) { return pair.first.first; });
135                    #endif
136                    RadixSort::sort_pairs(ranks, n);
137                    build_ranks(ranks, arr);
138
139                    max_rank = arr[ranks.back().second];
140                    jump *= 2;
141                }
142            }

```

```

142     }
143
144     vector<int> sa(this->n);
145     for (int i = 0; i < this->n; ++i)
146         sa[arr[i] - 1] = i;
147     return sa;
148 }
149
150 /// Builds the lcp (Longest Common Prefix) array for the string s.
151 /// A value lcp[i] indicates length of the longest common prefix of the
152 /// suffixes indexed by i and i + 1. Implementation of the Kasai's
153 /// Algorithm.
154 /// Time Complexity: O(n)
155 vector<int> build_lcp() {
156     vector<int> lcp(this->n, 0);
157     vector<int> inverse_suffix(this->n);
158
159     for (int i = 0; i < this->n; ++i)
160         inverse_suffix[sa[i]] = i;
161
162     for (int i = 0, k = 0; i < this->n; ++i) {
163         if (inverse_suffix[i] == this->n - 1) {
164             k = 0;
165         } else {
166             int j = sa[inverse_suffix[i] + 1];
167             while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
168                 ++k;
169
170             lcp[inverse_suffix[i]] = k;
171
172             if (k > 0)
173                 --k;
174         }
175     }
176
177     return lcp;
178 }
179
180 int _lcs(const int separator) {
181     int ans = 0;
182     for (int i = 0; i + 1 < this->sa.size(); ++i) {
183         const int left = this->sa[i];
184         const int right = this->sa[i + 1];
185         if ((left < separator && right > separator) ||
186             (left > separator && right < separator))
187             ans = max(ans, lcp[i]);
188     }
189     return ans;
190 }
191
192 #ifdef BUILD_TABLE
193 int _compare(const int i, const int j, const int length) {
194     const int k = this->log_array[length]; // floor log2(length)
195     const int jump = length - (1ll << k);
196
197     const pair<int, int> iRank = {
198         this->rank_table[k][i],
199         (i + jump < this->n ? this->rank_table[k][i + jump] : -1)};
200     const pair<int, int> jRank = {
201         this->rank_table[k][j],
202         (j + jump < this->n ? this->rank_table[k][j + jump] : -1)};
203     return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
204 }
205 #endif

```

```

206 public:
207     const vector<int> sa = build_suffix_array();
208     const vector<int> lcp = build_lcp();
209
210     /// LCS of two strings A and B. The string s must be initialized in the
211     /// constructor as the string (A + '$' + B).
212     /// The string A starts at index 1 and ends at index (separator - 1).
213     /// The string B starts at index (separator + 1) and ends at the end of the
214     /// string.
215     ///
216     /// Time Complexity: O(n)
217     int lcs(const int separator) {
218         assert(!isalpha(this->s[separator]) && !isdigit(this->s[separator]));
219         return _lcs(separator);
220     }
221
222     #ifdef BUILD_TABLE
223     /// Compares two substrings beginning at indexes i and j of a fixed length.
224     ///
225     /// Time Complexity: O(1)
226     int compare(const int i, const int j, const int length) {
227         assert(0 <= i && i < this->n && 0 <= j && j < this->n);
228         assert(i + length - 1 < this->n && j + length - 1 < this->n);
229         return _compare(i, j, length);
230     }
231     #endif
232 };
233 // clang-format on
234

```

10.11. Suffix Array Pessoa

```

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

```

```

33     for (int i = 0; i + 1 < n; i++) {
34         int a = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h) ||
35             (posBucket[temp[i + 1] + h] != posBucket[temp[i] + h]);
36         bucket[i] = c;
37         c += a;
38     }
39     bucket[n - 1] = c++;
40     temp.swap(out);
41 }
42 return out;
43 }

```

10.12. Trie

```

1 class Trie {
2 private:
3     static const int INT_LEN = 31;
4     // static const int INT_LEN = 63;
5
6 public:
7     struct Node {
8         map<char, Node*> next;
9         int id;
10        // cnt counts the number of words which pass in that node
11        int cnt = 0;
12        // word counts the number of words ending at that node
13        int word_cnt = 0;
14
15        Node(const int x) : id(x) {}
16    };
17
18 private:
19     int trie_size = 0;
20     // contains the next id to be used in a node
21     int node_cnt = 0;
22     Node *trie_root = this->make_node();
23
24 private:
25     Node *make_node() { return new Node(node_cnt++); }
26
27     int trie_insert(const string &s) {
28         Node *aux = this->root();
29         for (const char c : s) {
30             if (!aux->next.count(c))
31                 aux->next[c] = this->make_node();
32             aux = aux->next[c];
33             ++aux->cnt;
34         }
35         ++aux->word_cnt;
36         ++this->trie_size;
37         return aux->id;
38     }
39
40     void trie_erase(const string &s) {
41         Node *aux = this->root();
42         for (const char c : s) {
43             Node *last = aux;
44             aux = aux->next[c];
45             --aux->cnt;
46             if (aux->cnt == 0) {
47                 last->next.erase(c);
48                 aux = nullptr;
49                 break;
50             }
51         }
52     }
53

```

```

51     }
52     if (aux != nullptr)
53         --aux->word_cnt;
54     --this->trie_size;
55 }
56
57 int trie_count(const string &s) {
58     Node *aux = this->root();
59     for (const char c : s) {
60         if (aux->next.count(c))
61             aux = aux->next[c];
62         else
63             return 0;
64     }
65     return aux->word_cnt;
66 }
67
68 int trie_query_xor_max(const string &s) {
69     Node *aux = this->root();
70     int ans = 0;
71     for (const char c : s) {
72         const char inv = (c == '0' ? '1' : '0');
73         if (aux->next.count(inv)) {
74             ans = (ans << 111) | (inv - '0');
75             aux = aux->next[inv];
76         } else {
77             ans = (ans << 111) | (c - '0');
78             aux = aux->next[c];
79         }
80     }
81     return ans;
82 }
83
84 public:
85     Trie() {}
86
87     Node *root() { return this->trie_root; }
88
89     int size() { return this->trie_size; }
90
91     /// Returns the number of nodes present in the trie.
92     int node_count() { return this->node_cnt; }
93
94     /// Inserts s in the trie.
95     ///
96     /// Returns the id of the last character of the string in the trie.
97     ///
98     /// Time Complexity: O(s.size())
99     int insert(const string &s) { return this->trie_insert(s); }
100
101     /// Inserts the binary representation of x in the trie.
102     ///
103     /// Time Complexity: O(log x)
104     int insert(const int x) {
105         assert(x >= 0);
106         // converting x to binary representation
107         return this->trie_insert(bitset<INT_LEN>(x).to_string());
108     }
109
110     /// Removes the string s from the trie.
111     ///
112     /// Time Complexity: O(s.size())
113     void erase(const string &s) { this->trie_erase(s); }
114
115     /// Removes the binary representation of x from the trie.

```

```

116     ///
117     /// Time Complexity: O(log x)
118     void erase(const int x) {
119         assert(x >= 0);
120         // converting x to binary representation
121         this->trie_erase(bitset<INT_LEN>(x).to_string());
122     }
123
124     /// Returns the number of maximum xor sum with x present in the trie.
125     ///
126     /// Time Complexity: O(log x)
127     int query_xor_max(const int x) {
128         assert(x >= 0);
129         // converting x to binary representation
130         return this->trie_query_xor_max(bitset<INT_LEN>(x).to_string());
131     }
132
133     /// Returns the number of strings equal to s present in the trie.
134     ///
135     /// Time Complexity: O(s.size())
136     int count(const string &s) { return this->trie_count(s); }
137 };

```