

C++ Competitive Programming Library

DO NOT DISCLOSE OR DISTRIBUTE

bfs.07 - Bernardo Flores Salmeron

1	Template	3			
2	Data Structures	3			
2.1	Arvore Binaria	3	4.2	Circle-Circle Intersection	16
2.2	Bit2D	4	4.3	Closest Pair Of Points	17
2.3	Merge Sort Tree (K-Esimo Maior Elemento Num Intervalo, Valores Maiores Que K Num Intervalo,	5	4.4	Condicao De Existencia De Um Triangulo	17
2.4	Mos Algorithm	5	4.5	Convex Hull	18
2.5	Ordenacao De Estruturas (Pq, Etc)	6	4.6	Cross Product	18
2.6	Ordered Set (Policy Based Data Structures)	6	4.7	Distance Point Segment	18
2.7	Sqrt Decomposition	6	4.8	Line-Line Intersection	18
2.8	Bit	6	4.9	Line-Point Distance	18
2.9	Bit (Range Update)	7	4.10	Point Inside Convex Polygon - Log(N)	19
2.10	Counting Inversions (Minimum Number Of Adjacent Swaps To Sort Array)	8	4.11	Point Inside Polygon	20
2.11	Persistent Segment Tree	8	4.12	Points Inside And In Boundary Polygon	20
2.12	Segment Tree	9	4.13	Polygon Area (3D)	21
2.13	Segment Tree 2D	11	4.14	Polygon Area	21
2.14	Segment Tree Polynomial	11	4.15	Segment-Segment Intersection	21
2.15	Sparse Table	12	4.16	Upper And Lower Hull	22
3	Dp	13	4.17	Struct Point And Line	22
3.1	Achar Maior Palindromo	13	5	Graphs	23
3.2	Catalan	13	5.1	Checa Grafo Bipartido	23
3.3	Digit Dp	13	5.2	Ciclo Grafo	23
3.4	Longest Common Subsequence	14	5.3	Diametro Em Arvore	23
3.5	Longest Common Substring	14	5.4	Floyd Warshall	23
3.6	Longest Increasing Subsequence 2D (Not Sorted)	14	5.5	Ford Fulkersson (Maximum Flow)	24
3.7	Longest Increasing Subsequence 2D (Sorted)	15	5.6	Pontes Num Grafo	24
3.8	Longest Increasing Subsequence	15	5.7	Pontos De Articulaao	25
3.9	Subset Sum Com Bitset	16	5.8	Scc (Kosaraju)	25
3.10	Coin Change Problem	16	5.9	All Eulerian Path Or Tour	26
3.11	Knapsack	16	5.10	Bellman Ford	28
4	Geometry	16	5.11	De Bruijn Sequence	28
4.1	Centro De Massa De Um Poligono	16	5.12	Dijkstra + Dij Graph	29
			5.13	Dinic (Max Flow)	30
			5.14	Functional Graph	31
			5.15	Hld	33

5.16	Kruskal + Dsu	34	7.4	Combinacao Ncr	41
5.17	Lca	34	7.5	Compressao De Pontos	41
5.18	Maximum Path Unweighted Graph	36	7.6	Equacao Diofantina	41
5.19	Number Of Different Spanning Trees In A Complete Graph	37	7.7	Euclides Estendido	41
5.20	Number Of Ways To Make A Graph Connected	37	7.8	Euler Totient	42
5.21	Pruffer Decode	37	7.9	Fatoracao Multiplas Queries	42
5.22	Pruffer Encode	37	7.10	Fatoracao Simples	42
5.23	Pruffer Properties	37	7.11	Inclusao-Exclusao	42
5.24	Remove All Bridges From Graph	38	7.12	Numero De Fatores	42
5.25	Shortest Cycle In A Graph	38	7.13	Pollard Rho (Find A Divisor)	43
5.26	Topological Sort	38	7.14	Precomputar Combinacao Ncr	43
5.27	Tree Distance	38	7.15	Teorema Chines Do Resto	43
6	Language Stuff	39	7.16	Binary Exponentiation	44
6.1	Binary String To Int	39	7.17	Divisors	44
6.2	Climits	39	7.18	Matrix Exponentiation	44
6.3	Checagem Brute Force Com Solucao	39	7.19	Modular Inverse	44
6.4	Checagem De Bits	39	7.20	Sieve + Segmented Sieve	45
6.5	Checagem E Tranformacao De Caractere	39	8	Miscellaneous	45
6.6	Conta Digitos 1 Ate N	39	8.1	2-Sat	45
6.7	Escrita Em Arquivo	39	8.2	3Sum Problem	46
6.8	Gcd	39	8.3	Fibonacci Matrix Exponentiation	46
6.9	Hipotenusa	39	8.4	Infix To Prefix	47
6.10	Int To Binary String	39	8.5	Interval Scheduling	47
6.11	Int To String	39	8.6	Kadane (Maior Soma Num Vetor)	47
6.12	Leitura De Arquivo	39	8.7	Kadane 2D	47
6.13	Max E Min Element Num Vetor	39	8.8	Oito Rainhas	48
6.14	Permutacao	39	8.9	Sliding Window Minimum	48
6.15	Printf De Uma String	39	8.10	Torre De Hanoi	48
6.16	Remove Repeticoes Continuas Num Vetor	40	8.11	Kadane (Segment Tree)	49
6.17	Rotate (Left)	40	8.12	Point Compression	49
6.18	Rotate (Right)	40	9	Strings	49
6.19	Scanf De Uma String	40	9.1	Kmp	49
6.20	Split Function	40	9.2	Suffix Array	50
6.21	String To Long Long	40	9.3	Trie - Maximum Xor Sum	51
6.22	Substring	40	9.4	Trie - Maximum Xor Two Elements	51
6.23	Width	40	9.5	Z-Function	51
6.24	Check Overflow	40	9.6	Aho Corasick	51
6.25	Readint	40	9.7	Hashing	52
7	Math	40	9.8	Lcs K Strings	53
7.1	Bell Numbers	40	9.9	Lexicographically Smallest Rotation	54
7.2	Checagem De Primalidade	41	9.10	Manacher (Longest Palindrome)	54
7.3	Combinacao Ncr Mod Primo	41	9.11	Suffix Array	55

9.12 Suffix Array Pessoa	56
9.13 Suffix Array With Additional Memory	58
9.14 Trie	60

1. Template

```

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

2. Data Structures

2.1. Arvore Binaria

```

1 // C program to demonstrate delete operation in binary search tree
2 #include<stdio.h>
3 #include<stdlib.h>
4
5 struct node {
6     int key;
7     struct node *left, *right;
8 };
9
10 // A utility function to create a new BST node
11 struct node *newNode(int item) {
12     struct node *temp = (struct node *)malloc(sizeof(struct node));
13     temp->key = item;
14     temp->left = temp->right = NULL;
15     return temp;
16 }
17
18 // A utility function to do inorder traversal of BST
19 void inorder(struct node *root) {
20     if (root != NULL) {
21         inorder(root->left);
22         printf("%d ", root->key);
23         inorder(root->right);
24     }
25 }
26
27 /* A utility function to insert a new node with given key in BST */
28 struct node* insert(struct node* node, int key) {
29     /* If the tree is empty, return a new node */
30     if (node == NULL) return newNode(key);
31
32     /* Otherwise, recur down the tree */
33     if (key < node->key)
34         node->left = insert(node->left, key);
35     else
36         node->right = insert(node->right, key);
```

```

37  /* return the (unchanged) node pointer */
38  return node;
39  }
40
41  /* Given a non-empty binary search tree, return the node with minimum
42  key value found in that tree. Note that the entire tree does not
43  need to be searched. */
44  struct node * minValueNode(struct node* node) {
45      struct node* current = node;
46
47      /* loop down to find the leftmost leaf */
48      while (current->left != NULL)
49          current = current->left;
50
51      return current;
52  }
53
54  /* Given a binary search tree and a key, this function deletes the key
55  and returns the new root */
56  struct node* deleteNode(struct node* root, int key) {
57      // base case
58      if (root == NULL) return root;
59
60      // If the key to be deleted is smaller than the root's key,
61      // then it lies in left subtree
62      if (key < root->key)
63          root->left = deleteNode(root->left, key);
64
65      // If the key to be deleted is greater than the root's key,
66      // then it lies in right subtree
67      else if (key > root->key)
68          root->right = deleteNode(root->right, key);
69
70      // if key is same as root's key, then This is the node
71      // to be deleted
72      else {
73          // node with only one child or no child
74          if (root->left == NULL) {
75              struct node *temp = root->right;
76              free(root);
77              return temp;
78          } else if (root->right == NULL) {
79              struct node *temp = root->left;
80              free(root);
81              return temp;
82          }
83
84          // node with two children: Get the inorder successor (smallest
85          // in the right subtree)
86          struct node* temp = minValueNode(root->right);
87
88          // Copy the inorder successor's content to this node
89          root->key = temp->key;
90
91          // Delete the inorder successor
92          root->right = deleteNode(root->right, temp->key);
93      }
94      return root;
95  }
96
97  // Driver Program to test above functions
98  int main() {
99      /* Let us create following BST
100      50
101

```

```

102      /      \
103     30      70
104    /  \    /  \
105   20  40  60  80 */
106  struct node *root = NULL;
107  root = insert(root, 50);
108  root = insert(root, 30);
109  root = insert(root, 20);
110  root = insert(root, 40);
111  root = insert(root, 70);
112  root = insert(root, 60);
113  root = insert(root, 80);
114
115  printf("Inorder traversal of the given tree \n");
116  inorder(root);
117
118  printf("\nDelete 20\n");
119  root = deleteNode(root, 20);
120  printf("Inorder traversal of the modified tree \n");
121  inorder(root);
122
123  printf("\nDelete 30\n");
124  root = deleteNode(root, 30);
125  printf("Inorder traversal of the modified tree \n");
126  inorder(root);
127
128  printf("\nDelete 50\n");
129  root = deleteNode(root, 50);
130  printf("Inorder traversal of the modified tree \n");
131  inorder(root);
132
133  return 0;
134  }

```

2.2. 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 contains 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     /// Updates point (x, y).
73     ///
74     /// Time Complexity: O(log(n) + log(m))
75     void update(const int x, const int y, const int delta) {
76         assert(0 < x); assert(x < this->n);
77         assert(0 < y); assert(y < this->m);
78
79         bit_update(x, y, delta);
80     }
81 };

```

2.3. 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 }

```

2.4. Mos Algorithm

```

1 struct Tree {
2     int l, r, ind;
3 };
4 Tree query[311111];
5 int arr[311111];
6 int freq[1111111];
7 int ans[311111];
8 int block = sqrt(n), cont = 0;
9
10 bool cmp(Tree a, Tree b) {
11     if(a.l/block == b.l/block)
12         return a.r < b.r;
13     return a.l/block < b.l/block;
14 }
15
16 void add(int pos) {
17     freq[arr[pos]]++;
18     if(freq[arr[pos]] == 1) {
19         cont++;
20     }
21 }
22
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     }
36 }

```

```

33     freq[arr[i]] = 0;
34 }
35
36 int m;  cin >> m;
37
38 for(int i = 0; i < m; i++) {
39     cin >> query[i].l >> query[i].r;
40     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 }

```

2.5. Ordenacao De Estruturas (Pq, Etc)

```

1 struct cmp {
2     bool operator(ii a, ii b) {
3         //ordena primeiro pelo first(decrecente), dps pelo second(crescente)
4         if(a.first == b.first)
5             return a.second < b.second;
6         return a.first > b.first;
7     }
8 }
9 Ex: pq<ii,vector<ii>,cmp> fila;

```

2.6. Ordered Set (Policy Based Data Structures)

```

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 typedef tree<
9 int,
10 null_type,
11 less<int>,
12 rb_tree_tag,
13 tree_order_statistics_node_update>
14 ordered_set;
15
16 ordered_set X;
17 X.insert(1); X.insert(2);
18 X.insert(4); X.insert(8);

```

```

19 X.insert(16);
20 // 1, 2, 4, 8, 16
21 // retorna o k-ésimo maior elemento a partir de 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 // retorna o número de itens estritamente menores que o número
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

```

2.7. Sqrt Decomposition

```

1 // Problem: Sum from 1 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 }

```

2.8. Bit

```

1 /// INDEX THE ARRAY BY 1!!!
2 class BIT {
3 private:
4     vector<int> bit;
5     int n;
6
7 private:
8     int low(const int i) { return (i & (-i)); }
9
10    // point update
11    void bit_update(int i, const int delta) {
12        while (i <= this->n) {
13            this->bit[i] += delta;
14            i += this->low(i);
15        }
16    }
17
18    // point query
19    int bit_query(int i) {
20        int sum = 0;
21        while (i > 0) {

```

```

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

```

2.9. Bit (Range Update)

```

1 /// INDEX THE ARRAY BY 1!!!
2 class BIT {
3 private:
4     vector<int> bit1;
5     vector<int> bit2;
6     int n;
7
8 private:
9     int low(int i) {
10         return (i & (-i));
11     }
12
13 // point update
14 void update(int i, const int delta, vector<int> &bit) {
15     while(i <= this->n) {

```

```

16         bit[i] += delta;
17         i += this->low(i);
18     }
19 }
20
21 // point query
22 int query(const int i, const vector<int> &bit) {
23     int sum = 0;
24     while(i > 0) {
25         sum += bit[i];
26         i -= this->low(i);
27     }
28     return sum;
29 }
30
31 // build the bit
32 void build(const vector<int> &arr) {
33     // OBS: BIT IS INDEXED FROM 1
34     // THE USE OF 1-BASED ARRAY IS MANDATORY
35     assert(arr.front() == 0);
36     this->n = (int)arr.size() - 1;
37     this->bit1.resize(arr.size(), 0);
38     this->bit2.resize(arr.size(), 0);
39
40     for(int i = 1; i <= this->n; i++)
41         this->update(i, arr[i]);
42 }
43
44 public:
45 BIT(const vector<int> &arr) {
46     this->build(arr);
47 }
48
49 BIT(const int n) {
50     // OBS: BIT IS INDEXED FROM 1
51     // THE USE OF 1-BASED ARRAY IS MANDATORY
52     this->n = n;
53     this->bit1.resize(n + 1, 0);
54     this->bit2.resize(n + 1, 0);
55 }
56
57 // range update
58 void update(const int l, const int r, const int delta) {
59     assert(1 <= l); assert(l <= r); assert(r <= this->n);
60     this->update(l, delta, this->bit1);
61     this->update(r + 1, -delta, this->bit1);
62     this->update(l, delta * (l - 1), this->bit2);
63     this->update(r + 1, -delta * r, this->bit2);
64 }
65
66 // point update
67 void update(const int i, const int delta) {
68     assert(1 <= i); assert(i <= this->n);
69     this->update(i, i, delta);
70 }
71
72 // range query
73 int query(const int l, const int r) {
74     assert(1 <= l); assert(l <= r); assert(r <= this->n);
75     return this->query(r) - this->query(l - 1);
76 }
77
78 // point query
79 int query(const int i) {
80     assert(1 <= i); assert(i <= this->n);

```

```

81     return (this->query(i, this->bit1) * i) - this->query(i, this->bit2);
82 }
83 };
84
85 // TESTS
86 // signed main()
87 // {
88
89 //     vector<int> input = {0,1,2,3,4,5,6,7};
90
91 //     BIT ft(input);
92
93 //     assert (1 == ft.query(1));
94 //     assert (3 == ft.query(2));
95 //     assert (6 == ft.query(3));
96 //     assert (10 == ft.query(4));
97 //     assert (15 == ft.query(5));
98 //     assert (21 == ft.query(6));
99 //     assert (28 == ft.query(7));
100 //     assert (12 == ft.query(3,5));
101 //     assert (21 == ft.query(1,6));
102 //     assert (28 == ft.query(1,7));
103 // }

```

2.10. 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 }

```

2.11. Persistent Segment Tree

```

1 class Persistent_Seg_Tree {
2     struct Node {
3         int val;
4         Node *left, *right;
5         Node() {}
6         Node(int v, Node *l, Node *r) : val(v), left(l), right(r) {}
7     };
8     #define NEUTRAL_NODE Node(0, nullptr, nullptr);
9     Node _NEUTRAL_NODE = Node(0, nullptr, nullptr);
10
11 public:
12     int merge_nodes(const int x, const int y) { return x + y; }
13
14 private:
15     int n;
16     vector<Node *> version;
17
18 public:
19     Persistent_Seg_Tree() { this->n = -1; }
20     /// Builds version[0] with the values in the array.
21     ///

```

```

22 /// Time complexity: O(n)
23 Node *pst_build(Node *node, const int l, const int r,
24               const vector<int> &arr) {
25     node = new NEUTRAL_NODE;
26     if (l == r) {
27         node->val = arr[l];
28         return node;
29     }
30
31     int mid = (l + r) / 2;
32     node->left = pst_build(node->left, l, mid, arr);
33     node->right = pst_build(node->right, mid + 1, r, arr);
34     node->val = merge_nodes(node->left->val, node->right->val);
35     return node;
36 }

```

/// Builds version[0] with 0.

```

37
38
39
40 /// Time complexity: O(n)
41 Node *pst_build_empty(Node *node, const int l, const int r) {
42     node = new NEUTRAL_NODE;
43     if (l == r)
44         return node;
45
46     int mid = (l + r) / 2;
47     node->left = pst_build_empty(node->left, l, mid);
48     node->right = pst_build_empty(node->right, mid + 1, r);
49     node->val = merge_nodes(node->left->val, node->right->val);
50     return node;
51 }

```

```

52
53 Node *pst_update(Node *cur_tree, Node *prev_tree, const int l, const int r,
54               const int idx, const int delta) {
55     if (l > idx || r < idx) {
56         if (cur_tree != nullptr)
57             return cur_tree;
58         return prev_tree;
59     }
60
61     if (cur_tree == nullptr)
62         cur_tree = new Node(prev_tree->val, prev_tree->left, prev_tree->right);
63     else
64         cur_tree = new Node(cur_tree->val, cur_tree->left, cur_tree->right);
65
66     if (l == r) {
67         cur_tree->val += delta;
68         return cur_tree;
69     }
70
71     int mid = (l + r) / 2;
72     cur_tree->left =
73         pst_update(cur_tree->left, prev_tree->left, l, mid, idx, delta);
74     cur_tree->right =
75         pst_update(cur_tree->right, prev_tree->right, mid + 1, r, idx,
76         delta);
77     cur_tree->val = merge_nodes(cur_tree->left->val, cur_tree->right->val);
78     return cur_tree;
79 }

```

```

80 int pst_query(Node *node, const int l, const int r, const int i,
81             const int j) {
82     if (l > j || r < i)
83         return _NEUTRAL_NODE.val;
84
85     if (i <= l && r <= j)

```



```

86     return node->val;
87
88     int mid = (l + r) / 2;
89     return merge_nodes(pst_query(node->left, l, mid, i, j),
90                       pst_query(node->right, mid + 1, r, i, j));
91 }
92
93 public:
94 Persistent_Seg_Tree(const int n, const int number_of_versions) {
95     this->n = n;
96     version.resize(number_of_versions);
97     this->version[0] = this->pst_build_empty(this->version[0], 0, this->n -
98     1);
99
100    /// Constructor that allows to pass initial values to the leafs.
101    Persistent_Seg_Tree(const vector<int> &arr, const int number_of_versions) {
102        this->n = arr.size();
103        version.resize(number_of_versions);
104        this->version[0] = this->pst_build(this->version[0], 0, this->n - 1,
105        arr);
106    }
107
108    /// Links the root of a version to a previous version.
109    ///
110    /// Time Complexity: O(1)
111    void link(const int version, const int prev_version) {
112        assert(this->n > -1);
113        assert(0 <= prev_version);
114        assert(prev_version <= version);
115        assert(version < this->version.size());
116        this->version[version] = this->version[prev_version];
117    }
118
119    /// Updates an index in cur_tree based on prev_tree with a delta.
120    ///
121    /// Time Complexity: O(log(n))
122    void update(const int cur_version, const int prev_version, const int idx,
123    const int delta) {
124        assert(this->n > -1);
125        assert(0 <= prev_version);
126        assert(prev_version <= cur_version);
127        assert(cur_version < this->version.size());
128        this->version[cur_version] = this->pst_update(this->version[cur_version],
129        this->version[prev_version],
130        0, this->n - 1, idx,
131        delta);
132    }
133
134    /// Query from l to r.
135    ///
136    /// Time Complexity: O(log(n))
137    int query(const int version, const int l, const int r) {
138        assert(this->n > -1);
139        assert(this->version[version] != nullptr);
140        assert(0 <= l);
141        assert(l <= r);
142        assert(r < this->n);
143        return this->pst_query(this->version[version], 0, this->n - 1, l, r);
144    }
145 };

```

2.12. Segment Tree

```

1  class Seg_Tree {
2  public:
3      struct Node {
4          int val, lazy;
5
6          Node() {}
7          Node(const int val, const int lazy) : val(val), lazy(lazy) {}
8      };
9
10     private:
11     // // range sum
12     // Node NEUTRAL_NODE = Node(0, 0);
13     // Node merge_nodes(const Node &x, const Node &y) {
14     //     return Node(x.val + y.val, 0);
15     // }
16     // void apply_lazy(const int l, const int r, const int pos) {
17     //     tree[pos].val += (r - l + 1) * tree[pos].lazy;
18     // }
19
20     // // RMQ max
21     // Node NEUTRAL_NODE = Node(-INF, 0);
22     // Node merge_nodes(const Node &x, const Node &y) {
23     //     return Node(max(x.val, y.val), 0);
24     // }
25     // void apply_lazy(const int l, const int r, const int pos) {
26     //     tree[pos].val += tree[pos].lazy;
27     // }
28
29     // // RMQ min
30     // Node NEUTRAL_NODE = Node(INF, 0);
31     // Node merge_nodes(const Node &x, const Node &y) {
32     //     return Node(min(x.val, y.val), 0);
33     // }
34     // void apply_lazy(const int l, const int r, const int pos) {
35     //     tree[pos].val += tree[pos].lazy;
36     // }
37
38     // XOR
39     // Only works with point updates
40     // Node NEUTRAL_NODE = Node(0, 0);
41     // Node merge_nodes(const Node &x, const Node &y) {
42     //     return Node(x.val ^ y.val, 0);
43     // }
44     // void apply_lazy(const int l, const int r, const int pos) {}
45
46     private:
47     int n;
48
49     public:
50     vector<Node> tree;
51
52     private:
53     void st_propagate(const int l, const int r, const int pos) {
54         if (tree[pos].lazy != 0) {
55             apply_lazy(l, r, pos);
56             if (l != r) {
57                 tree[2 * pos + 1].lazy += tree[pos].lazy;
58                 tree[2 * pos + 2].lazy += tree[pos].lazy;
59             }
60             tree[pos].lazy = 0;
61         }
62     }
63
64     Node st_build(const int l, const int r, const vector<int> &arr,

```

```

65         const int pos) {
66     if (l == r)
67         return tree[pos] = Node(arr[l], 0);
68
69     int mid = (l + r) / 2;
70     return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
71                                   st_build(mid + 1, r, arr, 2 * pos + 2));
72 }
73
74 int st_get_first(const int l, const int r, const int v, const int pos) {
75     st_propagate(l, r, pos);
76
77     // Needs RMQ MAX
78     // Replace to <= for greater or equal or (with RMQ MIN) > for smaller or
79     // equal or >= for smaller
80     if (tree[pos].val < v)
81         return -1;
82
83     if (l == r)
84         return l;
85
86     int mid = (l + r) / 2;
87     int aux = st_get_first(l, mid, v, 2 * pos + 1);
88     if (aux != -1)
89         return aux;
90     return st_get_first(mid + 1, r, v, 2 * pos + 2);
91 }
92
93 Node st_query(const int l, const int r, const int i, const int j,
94              const int pos) {
95     st_propagate(l, r, pos);
96
97     if (l > r || l > j || r < i)
98         return NEUTRAL_NODE;
99
100    if (i <= l && r <= j)
101        return tree[pos];
102
103    int mid = (l + r) / 2;
104    return merge_nodes(st_query(l, mid, i, j, 2 * pos + 1),
105                      st_query(mid + 1, r, i, j, 2 * pos + 2));
106 }
107
108 // it adds a number delta to the range from i to j
109 Node st_update(const int l, const int r, const int i, const int j,
110              const int delta, const int pos) {
111     st_propagate(l, r, pos);
112
113     if (l > r || l > j || r < i)
114         return tree[pos];
115
116     if (i <= l && r <= j) {
117         tree[pos].lazy = delta;
118         st_propagate(l, r, pos);
119         return tree[pos];
120     }
121
122     int mid = (l + r) / 2;
123     return tree[pos] =
124         merge_nodes(st_update(l, mid, i, j, delta, 2 * pos + 1),
125                   st_update(mid + 1, r, i, j, delta, 2 * pos + 2));
126 }
127
128 void build(const vector<int> &arr) {
129     this->n = arr.size();

```

```

130     this->tree.resize(4 * this->n);
131     this->st_build(0, this->n - 1, arr, 0);
132 }
133
134 public:
135     /// N equals to -1 means the Segment Tree hasn't been created yet.
136     Seg_Tree() : n(-1) {}
137
138     /// Constructor responsible initialize a tree with 0.
139     ///
140     /// Time Complexity O(n)
141     Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0, 0));
142     }
143
144     /// Constructor responsible for building the initial tree based on a
145     /// vector.
146     ///
147     /// Time Complexity O(n)
148     Seg_Tree(const vector<int> &arr) { this->build(arr); }
149
150     /// Returns the first index from left to right.
151     ///
152     /// Uncomment the line in the original funtion to get the proper element
153     /// that
154     /// may be: GREATER OR EQUAL, GREATER, SMALLER OR EQUAL, SMALLER.
155     ///
156     /// Time Complexity O(log n)
157     int get_first(const int v) {
158         assert(this->n >= 0);
159         return this->st_get_first(0, this->n - 1, v, 0);
160     }
161
162     /// Update at a single index.
163     ///
164     /// Time Complexity O(log n)
165     void update(const int idx, const int delta) {
166         assert(this->n >= 0);
167         assert(0 <= idx, assert(idx < this->n);
168         this->st_update(0, this->n - 1, idx, idx, delta, 0);
169     }
170
171     /// Range update from l to r.
172     ///
173     /// Time Complexity O(log n)
174     void update(const int l, const int r, const int delta) {
175         assert(this->n >= 0);
176         assert(0 <= l), assert(l <= r), assert(r < this->n);
177         this->st_update(0, this->n - 1, l, r, delta, 0);
178     }
179
180     /// Query at a single index.
181     ///
182     /// Time Complexity O(log n)
183     int query(const int idx) {
184         assert(this->n >= 0);
185         assert(0 <= idx), assert(idx < this->n);
186         return this->st_query(0, this->n - 1, idx, idx, 0).val;
187     }
188
189     /// Range query from l to r.
190     ///
191     /// Time Complexity O(log n)
192     int query(const int l, const int r) {
193         assert(this->n >= 0);
194         assert(0 <= l), assert(l <= r), assert(r < this->n);
195         return this->st_query(0, this->n - 1, l, r, 0).val;

```

```
192 }
193 };
```

2.13. Segment Tree 2D

```
1 // REQUIRES segment_tree.cpp!!
2 class Seg_Tree_2d {
3 private:
4 // // range sum
5 // int NEUTRAL_VALUE = 0;
6 // int merge_nodes(const int &x, const int &y) {
7 //     return x + y;
8 // }
9
10 // // RMQ max
11 // int NEUTRAL_VALUE = -INF;
12 // int merge_nodes(const int &x, const int &y) {
13 //     return max(x, y);
14 // }
15
16 // // RMQ min
17 // int NEUTRAL_VALUE = INF;
18 // int merge_nodes(const int &x, const int &y) {
19 //     return min(x, y);
20 // }
21
22 private:
23 int n, m;
24
25 public:
26 vector<Seg_Tree> tree;
27
28 private:
29 void st_build(const int l, const int r, const int pos, const
30 vector<vector<int>> &mat) {
31     if(l == r)
32         tree[pos] = Seg_Tree(mat[l]);
33     else {
34         int mid = (l + r) / 2;
35         st_build(l, mid, 2*pos + 1, mat);
36         st_build(mid + 1, r, 2*pos + 2, mat);
37         for(int i = 0; i < tree[2*pos + 1].tree.size(); i++)
38             tree[pos].tree[i].val = merge_nodes(tree[2*pos + 1].tree[i].val,
39                                                 tree[2*pos + 2].tree[i].val);
40     }
41 }
42
43 int st_query(const int l, const int r, const int x1, const int y1, const
44 int x2, const int y2, const int pos) {
45     if(l > x2 || r < x1)
46         return NEUTRAL_VALUE;
47
48     if(x1 <= l && r <= x2)
49         return tree[pos].query(y1, y2);
50
51     int mid = (l + r) / 2;
52     return merge_nodes(st_query(l, mid, x1, y1, x2, y2, 2*pos + 1),
53                       st_query(mid + 1, r, x1, y1, x2, y2, 2*pos + 2));
54 }
55
56 void st_update(const int l, const int r, const int x, const int y, const
57 int delta, const int pos) {
58     if(l > x || r < x)
59         return;
```

```
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->m);
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->m);
109     st_update(0, this->n - 1, x, y, delta, 0);
110 }
111 };
```

2.14. Segment Tree Polynomial

```
1 // Works for the polynomial f(x) = z1*x + z0
2 class Seg_Tree {
3 public:
4     struct Node {
5         int val, z1, z0;
6     }
```

```

7     Node() {}
8     Node(const int val, const int z1, const int z0)
9         : val(val), z1(z1), z0(z0) {}
10 };
11
12 private:
13     // range sum
14     Node NEUTRAL_NODE = Node(0, 0, 0);
15     Node merge_nodes(const Node &x, const Node &y) {
16         return Node(x.val + y.val, 0, 0);
17     }
18     void apply_lazy(const int l, const int r, const int pos) {
19         tree[pos].val += (r - l + 1) * tree[pos].z0;
20         tree[pos].val += (r - l) * (r - l + 1) / 2 * tree[pos].z1;
21     }
22
23 private:
24     int n;
25
26 public:
27     vector<Node> tree;
28
29 private:
30     void st_propagate(const int l, const int r, const int pos) {
31         if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {
32             apply_lazy(l, r, pos);
33             int mid = (l + r) / 2;
34             int sz_left = mid - l + 1;
35             if (l != r) {
36                 tree[2 * pos + 1].z0 += tree[pos].z0;
37                 tree[2 * pos + 1].z1 += tree[pos].z1;
38
39                 tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
40                 tree[2 * pos + 2].z1 += tree[pos].z1;
41             }
42             tree[pos].z0 = 0;
43             tree[pos].z1 = 0;
44         }
45     }
46
47     Node st_build(const int l, const int r, const vector<int> &arr,
48                 const int pos) {
49         if (l == r)
50             return tree[pos] = Node(arr[l], 0, 0);
51
52         int mid = (l + r) / 2;
53         return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
54                                       st_build(mid + 1, r, arr, 2 * pos + 2));
55     }
56
57     Node st_query(const int l, const int r, const int i, const int j,
58                 const int pos) {
59         st_propagate(l, r, pos);
60
61         if (l > r || l > j || r < i)
62             return NEUTRAL_NODE;
63
64         if (i <= l && r <= j)
65             return tree[pos];
66
67         int mid = (l + r) / 2;
68         return merge_nodes(st_query(l, mid, i, j, 2 * pos + 1),
69                           st_query(mid + 1, r, i, j, 2 * pos + 2));
70     }
71 }

```

```

72 // it adds a number delta to the range from i to j
73 Node st_update(const int l, const int r, const int i, const int j,
74               const int z1, const int z0, const int pos) {
75     st_propagate(l, r, pos);
76
77     if (l > r || l > j || r < i)
78         return tree[pos];
79
80     if (i <= l && r <= j) {
81         tree[pos].z0 = (l - i + 1) * z0;
82         tree[pos].z1 = z1;
83         st_propagate(l, r, pos);
84         return tree[pos];
85     }
86
87     int mid = (l + r) / 2;
88     return tree[pos] =
89         merge_nodes(st_update(l, mid, i, j, z1, z0, 2 * pos + 1),
90                   st_update(mid + 1, r, i, j, z1, z0, 2 * pos + 2));
91 }
92
93 public:
94     Seg_Tree() : n(-1) {}
95
96     Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0, 0)); }
97
98     Seg_Tree(const vector<int> &arr) { this->build(arr); }
99
100     void build(const vector<int> &arr) {
101         this->n = arr.size();
102         this->tree.resize(4 * this->n);
103         this->st_build(0, this->n - 1, arr, 0);
104     }
105
106     /// Index update of a polynomial  $f(x) = z1 \cdot x + z0$ 
107     ///
108     /// Time Complexity  $O(\log n)$ 
109     void update(const int i, const int z1, const int z0) {
110         assert(this->n >= 0);
111         assert(0 <= i), assert(i < this->n);
112         this->st_update(0, this->n - 1, i, i, z1, z0, 0);
113     }
114
115     /// Range update of a polynomial  $f(x) = z1 \cdot x + z0$  from l to r
116     ///
117     /// Time Complexity  $O(\log n)$ 
118     void update(const int l, const int r, const int z1, const int z0) {
119         assert(this->n >= 0);
120         assert(0 <= l), assert(l <= r), assert(r < this->n);
121         this->st_update(0, this->n - 1, l, r, z1, z0, 0);
122     }
123
124     /// Range sum query from l to r
125     ///
126     /// Time Complexity  $O(\log n)$ 
127     int query(const int l, const int r) {
128         assert(this->n >= 0);
129         assert(0 <= l), assert(l <= r), assert(r < this->n);
130         return this->st_query(0, this->n - 1, l, r, 0).val;
131     }
132 };

```

```

1 // RMQ min implementation
2 class Sparse_Table {
3 private:
4     int n;
5     vector<vector<int>> > table;
6     vector<int> lg;
7
8     /// lg[i] represents the log2(i)
9     void build_log_array() {
10         lg.resize(this->n + 1);
11
12         for(int i = 2; i <= this->n; i++)
13             lg[i] = lg[i/2] + 1;
14     }
15
16     /// Time Complexity: O(n*log(n))
17     /// Space Complexity: O(n*log(n))
18     void build_sparse_table(const vector<int> &arr) {
19
20         table.resize(lg[this->n] + 1, vector<int>(this->n));
21
22         table[0] = arr;
23         int pow2 = 1;
24
25         for(int i = 1; i < table.size(); i++) {
26             int lastsz = this->n - pow2 + 1;
27             for(int j = 0; j + pow2 < lastsz; j++) {
28                 table[i][j] = merge(table[i-1][j], table[i-1][j+pow2]);
29             }
30             pow2 <= 1;
31         }
32     }
33
34     int merge(const int &l, const int &r) {
35         return min(l, r);
36     }
37
38 public:
39     Sparse_Table(const vector<int> &arr) {
40         this->n = arr.size();
41
42         this->build_log_array();
43         this->build_sparse_table(arr);
44     }
45
46     void print() {
47         int pow2 = 1;
48         for(int i = 0; i < table.size(); i++) {
49             int sz = (int)(table.front().size()) - pow2 + 1;
50             for(int j = 0; j < sz; j++) {
51                 cout << table[i][j] << " \n"[(j+1) == sz];
52             }
53             pow2 <= 1;
54         }
55     }
56
57     /// Query of a range from l to r.
58     ///
59     /// Time Complexity: O(1)
60     /// Space Complexity: O(1)
61     int query(int l, int r) {
62         assert(l <= r);
63         assert(0 <= l && r <= this->n - 1);

```

```

65     int lgg = lg[(r - l + 1)];
66     return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
67 }
68 };
69

```

3. Dp

3.1. Achar Maior Palindromo

1 Fazer LCS da string com o reverso

3.2. 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 }

```

3.3. Digit Dp

```

1 /// How many numbers x are there in the range a to b, where the digit d
2   occurs exactly k times in x?
3 vector<int> num;
4 int a, b, d, k;
5 int DP[12][12][2];
6 /// DP[p][c][f] = Number of valid numbers <= b from this state
7 /// p = current position from left side (zero based)
8 /// c = number of times we have placed the digit d so far

```

```

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 }

```

3.4. 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     string ans;
19     int i = n, j = m;
20     while(i > 0 && j > 0) {
21         if(s[i] == t[j])
22             ans += s[i], i--, j--;
23         else if(mat[i][j - 1] > mat[i - 1][j])
24             j--;
25         else
26             i--;
27     }
28
29     reverse(ans.begin(), ans.end());
30     return ans;
31 }
32

```

3.5. 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 }

```

3.6. Longest Increasing Subsequence 2D (Not Sorted)

```

1  set<ii> s[(int)2e6];
2  bool check(ii par, int ind) {
3
4      auto it = s[ind].lower_bound(ii(par.ff, -INF));
5      if(it == s[ind].begin())
6          return false;
7
8      it--;
9
10     if(it->ss < par.ss)
11         return true;
12     return false;

```

```

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

```

3.7. Longest Increasing Subsequence 2D (Sorted)

```

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

```

```

21 for(int i = 1; i < n; i++) {
22
23     ii x = arr[i];
24
25     int l = 1, r = maior;
26     int ansbb = 0;
27     while(l <= r) {
28         int mid = (l+r)/2;
29         if(check(x, mid)) {
30             l = mid + 1;
31             ansbb = mid;
32         } else {
33             r = mid - 1;
34         }
35     }
36
37     // inserting in list
38     auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
39     while(it != s[ansbb+1].end() && it->ss >= x.ss)
40         it = s[ansbb+1].erase(it);
41
42     it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
43     if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
44        it->ss <= x.ss)
45         continue;
46     s[ansbb+1].insert(arr[i]);
47
48     maior = max(maior, ansbb + 1);
49 }
50 return maior;
51 }
52

```

3.8. Longest Increasing Subsequence

```

1 int lis(vector<int> &arr){
2     int n = arr.size();
3     vector<int> lis;
4     for(int i = 0; i < n; i++){
5         int l = 0, r = (int)lis.size() - 1;
6         int ansj = -1;
7         while(l <= r){
8             int mid = (l+r)/2;
9             // OBS: PARA >= TROCAR SINAL EMBAIXO POR <=
10             if(arr[i] < lis[mid]){
11                 r = mid - 1;
12                 ansj = mid;
13             }
14             else l = mid + 1;
15         }
16         if(ansj == -1){
17             // se arr[i] e maior que todos
18             lis.push_back(arr[i]);
19         }
20         else {
21             lis[ansj] = arr[i];
22         }
23     }
24
25     return lis.size();
26 }

```

3.9. 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 }

```

3.10. 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. Knapsack

```

1 int dp[2001][2001];
2 int moc(int q,int p,vector<ii> vec) {
3     for(int i = 1; i <= q; i++)
4     {
5         for(int j = 1; j <= p; j++) {
6             if(j >= vec[i-1].ff)
7                 dp[i][j] = max(dp[i-1][j],vec[i-1].ss + dp[i-1][j-vec[i-1].ff]);
8             else
9                 dp[i][j] = dp[i-1][j];
10        }
11    }
12    return dp[q][p];
13 }
14 int main(int argc, char *argv[])
15 {
16     int p,q;
17     vector<ii> vec;
18     cin >> p >> q;
19     int x,y;
20     for(int i = 0; i < q; i++) {
21         cin >> x >> y;
22         vec.push_back(make_pair(x,y));
23     }

```

```

24     for(int i = 0; i <= p; i++)
25         dp[0][i] = 0;
26     for(int i = 1; i <= q; i++)
27         dp[i][0] = 0;
28     sort(vec.begin(),vec.end());
29     cout << moc(q,p,vec) << endl;
30 }

```

4. Geometry

4.1. Centro De Massa De Um Poligono

```

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

```

4.2. 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 }

```

4.3. 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 }

```

4.4. Condicao De Existencia De Um Triangulo

```

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

```

4.5. 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 }

```

4.6. Cross Product

```

1 // Outra forma de produto vetorial
2 // reta ab,ac se for zero e colinear
3 // se for < 0 então 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

```

4.7. 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 }

```

4.8. 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 }

```

4.9. 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.

```

```

28 // pto nao esta mais proximo de uma das bordas do segmento
29 // Ex:
30 //      |
31 //      | (Ângulo Reto)
32 //
33 // cord x do pto na reta = (x1 + t * diffX)
34 // cord y do pto na reta = (y1 + t * diffY)
35 *ptox = (x1 + t * diffX), *ptoy = (y1 + t * diffY);
36 diffX = pointX - (x1 + t * diffX);
37 diffY = pointY - (y1 + t * diffY);
38 }
39 //returning shortest distance
40 return sqrt(diffX * diffX + diffY * diffY);
41 }

```

4.10. Point Inside Convex Polygon - Log(N)

```

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

```

```

48     }
49     H.resize(k);
50     /* o último ponto do vetor é igual ao primeiro, atente para isso
51        as vezes é necessário mudar */
52
53     int j = 1;
54     lower.pb(H.front());
55     while (H[j].x >= H[j-1].x) {
56         lower.pb(H[j++]);
57     }
58
59     int l = H.size()-1;
60     while (l >= j) {
61         upper.pb(H[l--]);
62     }
63     upper.pb(H[l--]);
64
65     return H;
66 }
67
68 bool insidePolygon(pto p, vector<pto> &arr) {
69
70     if (pair<double,double>(p.x, p.y) == pair<double,double>(lower[0].x,
71         lower[0].y))
72         return true;
73
74     pto lo = {p.x, -(double)INF};
75     pto hi = {p.x, (double)INF};
76     auto itl = lower_bound(lower.begin(), lower.end(), lo);
77     auto itu = lower_bound(upper.begin(), upper.end(), lo);
78
79     if (itl == lower.begin() || itu == upper.begin()) {
80         auto it = lower_bound(arr.begin(), arr.end(), lo);
81         auto it2 = lower_bound(arr.begin(), arr.end(), hi);
82         it2--;
83         if (it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y
84             <= it2->y)
85             return true;
86         return false;
87     }
88
89     if (itl == lower.end() || itu == upper.end()) {
90         return false;
91     }
92
93     auto ol = itl, ou = itu;
94     ol--, ou--;
95     if (cross(*ol, *itl, p) >= 0 && cross(*ou, *itu, p) <= 0)
96         return true;
97
98     auto it = lower_bound(arr.begin(), arr.end(), lo);
99     auto it2 = lower_bound(arr.begin(), arr.end(), hi);
100     it2--;
101     if (it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y <=
102         it2->y)
103         return true;
104     return false;
105 }
106
107 signed main () {
108     ios_base::sync_with_stdio(false);
109     cin.tie(NULL);

```

```

110 double n, m, k;
111
112 cin >> n >> m >> k;
113
114 vector<pto> arr(n);
115
116 for(pto &x: arr) {
117     cin >> x.x >> x.y;
118 }
119
120 convex_hull(arr);
121
122 pto p;
123
124 int c = 0;
125 while(m--) {
126     cin >> p.x >> p.y;
127     cout << (insidePolygon(p, arr) ? "dentro" : "fora") << endl;
128 }
129
130 }

```

4.11. 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 ímpar
   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 - p.x) -
29             (q.x - p.x) * (r.y - p.y);
30
31     if (val == 0) return 0; // colinear
32     return (val > 0) ? 1 : 2; // clock or counterclock wise
33 }
34
35 // The function that returns true if line segment 'p1q1'
36 // and 'p2q2' intersect.
37 bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
38     // Find the four orientations needed for general and

```

```

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

```

4.12. 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 }

```

4.13. 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 }

```

4.14. Polygon Area

```

1
2 double polygonArea(vector<pto> &arr, int n) {
3     int area = 0;
4     // N = quantidade de pontos no polígono e armazenados em p;
5     // OBS: VALE PARA CONVEXO E NÃO CONVEXO
6     for(int i = 0; i < n; i++){
7         area += cross(arr[i], arr[(i+1)%n]);
8     }
9     return (double)abs(area/2.0);
10 }

```

4.15. Segment-Segment Intersection

```

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

```

```

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

```

4.16. Upper And Lower Hull

```

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

```

```

33         k--;
34         H[k++] = P[i];
35     }
36     H.resize(k);
37     /* o último ponto do vetor é igual ao primeiro, atente para isso
38        as vezes é necessário mudar */
39
40     int j = 1;
41     lower.pb(H.front());
42     while(H[j].x >= H[j-1].x) {
43         lower.pb(H[j++]);
44     }
45
46     int l = H.size()-1;
47     while(l >= j) {
48         upper.pb(H[l--]);
49     }
50     upper.pb(H[l--]);
51
52     return H;
53 }

```

4.17. 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. Checa Grafo Bipartido

```

1 bool isBipartite(int src, int V){
2
3     int colorArr[V + 1];
4     memset(colorArr, -1, sizeof(colorArr));
5     colorArr[src] = 1;
6
7     queue<int> q; q.push(src);
8
9     while (!q.empty()) {
10         int u = q.front(); q.pop();
11
12         // Find all non-colored adjacent vertices
13         for (auto it = adj[u].begin(); it != adj[u].end(); it++) {
14             //Return false if there is a self-loop
15             if (u == *it)
16                 return false;
17             // An edge from u to v exists and destination v is not colored
18
19             if (colorArr[*it] == -1) {
20                 // Assign alternate color to this adjacent v of u
21                 colorArr[*it] = 1 - colorArr[u];
22                 q.push(*it);
23             }
24             // An edge from u to v exists and destination v is colored with same
25             // color as u
26             else if (colorArr[*it] == colorArr[u])
27                 return false;
28         }
29         // If we reach here, then all adjacent vertices can be colored with
30         // alternate color
31         return true;
32     }
33 }

```

5.2. Ciclo Grafo

```

1 int n;
2 vector<vector<int>> adj;
3 vector<char> color;
4 vector<int> parent;

```

```

5 int cycle_start, cycle_end;
6
7 bool dfs(int v) {
8     color[v] = 1;
9     for (int u : adj[v]) {
10         if (color[u] == 0) {
11             parent[u] = v;
12             if (dfs(u))
13                 return true;
14         } else if (color[u] == 1) {
15             cycle_end = v;
16             cycle_start = u;
17             return true;
18         }
19     }
20     color[v] = 2;
21     return false;
22 }
23
24 void find_cycle() {
25     color.assign(n, 0);
26     parent.assign(n, -1);
27     cycle_start = -1;
28
29     for (int v = 0; v < n; v++) {
30         if (dfs(v))
31             break;
32     }
33
34     if (cycle_start == -1) {
35         cout << "Acyclic" << endl;
36     } else {
37         vector<int> cycle;
38         cycle.push_back(cycle_start);
39         for (int v = cycle_end; v != cycle_start; v = parent[v])
40             cycle.push_back(v);
41         cycle.push_back(cycle_start);
42         reverse(cycle.begin(), cycle.end());
43
44         cout << "Cycle found: ";
45         for (int v : cycle)
46             cout << v << " ";
47         cout << endl;
48     }
49 }

```

5.3. Diametro Em Arvore

1 Calcula qual o vértice a mais distante de um qualquer vértice X e do vértice A calcula-se o vértice B mais distante dele.

5.4. Floyd Warshall

```

1 // OBS: ZERAR adj[i][i] sempre
2 for(int i = 0; i < n; i++)
3     adj[i][i] = 0;
4
5 for(int k = 0; k < n; k++) {
6     for(int i = 0; i < n; i++) {
7         for(int j = 0; j < n; j++) {
8             adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
9         }
10     }
11 }

```

11 }

5.5. Ford Fulkersson (Maximum Flow)

```

1  int rGraph[2000][2000];
2  int graph[2000][2000];
3
4  int V;
5  bool bfs(int s, int t, int parent[]) {
6      bool visited[V];
7      memset(visited, 0, sizeof(visited));
8
9      // Create a queue, enqueue source vertex and mark source vertex
10     // as visited
11     queue<int> q;
12     q.push(s);
13     visited[s] = true;
14     parent[s] = -1;
15
16     // Standard BFS Loop
17     while (!q.empty()) {
18         int u = q.front();
19         q.pop();
20
21         for (int v=0; v<V; v++) {
22             if (visited[v]==false && rGraph[u][v] > 0) {
23                 q.push(v);
24                 parent[v] = u;
25                 visited[v] = true;
26             }
27         }
28     }
29     // If we reached sink in BFS starting from source, then return true, else
30     // false
31     return (visited[t] == true);
32 }
33 // Returns the maximum flow from s to t in the given graph
34 int fordFulkerson(int s, int t) {
35     int u, v;
36     // Create a residual graph and fill the residual graph with given
37     // capacities in the original graph as residual capacities in residual
38     // graph residual capacity of edge from i to j (if there is an edge. If
39     // rGraph[i][j] is 0, then there is not)
40     for (u = 0; u < V; u++)
41         for (v = 0; v < V; v++)
42             rGraph[u][v] = graph[u][v];
43
44     int parent[V]; // This array is filled by BFS and to store path
45
46     int max_flow = 0; // There is no flow initially
47
48     // Augment the flow while there is path from source to sink
49     while (bfs(s, t, parent)) {
50         // Find minimum residual capacity of the edges along the path filled by
51         // BFS. Or we can say find the maximum flow through the path found.
52         int path_flow = INT_MAX;
53         for (v=t; v!=s; v=parent[v]) {
54             u = parent[v];
55             path_flow = min(path_flow, rGraph[u][v]);
56         }
57         // update residual capacities of the edges and reverse edges
58         // along the path

```

```

56     for (v=t; v != s; v=parent[v]) {
57         u = parent[v];
58         rGraph[u][v] -= path_flow;
59         rGraph[v][u] += path_flow;
60     }
61
62     // Add path flow to overall flow
63     max_flow += path_flow;
64 }
65
66 // Return the overall flow
67 return max_flow;
68 }
69
70 // PRINT THE FLOW AFTER RUNNING THE ALGORITHM
71 void print(int n) {
72     for(int i = 1; i <= m; i++) {
73         for(int j = m+1; j <= m*2; j++) {
74             cout << "flow from i(left) to j(right) is " << graph[i][j] -
75                 rGraph[i][j] << endl;
76         }
77     }
78 }
79
80 void addEdge(int l, int r, int n, int x) {
81     graph[l][r+n] = x;
82 }
83
84 void addEdgeSource(int l, int x) {
85     graph[0][l] = x;
86 }
87
88 void addEdgeSink(int r, int n, int x) {
89     graph[r+n][V-1] = x;
90 }

```

5.6. Pontes Num Grafo

```

1  //SE TIRA-LAS O GRAFO FICA DESCONEXO
2  // OBS: PRESTAR ATENCAO EM SELF-LOOPS, é MELHOR NÃO ADICIONA-LOS
3  // SO FUNCIONA EM GRAFO NÃO DIRECIONADO
4  int t=1;
5  vector<int> T((int)2e6,0); //Tempo necessário para chegar naquele vértice na
6  // dfs
7  vector<int> adj[(int)2e6];
8  vector<int> Low((int)2e6); // Tempo "mínimo" para chegar naquele vértice na
9  // dfs
10 vector<int> ciclo((int)2e6, false);
11 vector<ii> bridges;
12 void dfs(int u, int p){
13     Low[u] = T[u] = t;
14     t++;
15     for(auto v : adj[u]){
16         if(v==p){
17             //checa arestas paralelas
18             p=-1;
19             continue;
20         }
21         //se ele ainda não foi visited
22         else if(T[v]==0){
23             dfs(v,u);
24             Low[u]=min(Low[u], Low[v]);
25             if(Low[v]>T[u]) {
26                 bridges.pb(ii(min(u,v), (max(u,v))));
27             }
28         }
29     }
30 }

```



```

25     // ponte de u para v
26     }
27     }
28     else
29         Low[u]=min(Low[u], T[v]);
30     ciclo[u] |= (T[u]>=Low[v]);
31     //checa se o vértice u faz parte de um ciclo
32     }
33 }
34
35 void clear() {
36
37     for(int i = 0; i <= n; i++) {
38         T[i] = 0, Low[i] = 0, adj[i].clear(), ciclo[i] = false;
39     }
40     bridges.clear();
41 }
42
43
44 signed main () {
45
46     for(int i = 0; i < n; i++)
47         if(T[i] == 0)
48             dfs(i, -1);
49
50     sort(bridges.begin(), bridges.end());
51
52     cout << (int)bridges.size() << endl;
53     for(int i = 0; i < bridges.size(); i++) {
54         cout << bridges[i].ff << " - " << bridges[i].ss << endl;
55     }
56     cout << endl;
57
58     clear();
59
60 }

```

5.7. Pontos De Articulação

```

1 // SE TIRAR TAIS VERTICES O GRAFO FICA DESCONEXO
2
3 vector<bool> ap(100000,false);
4 vector<int> low(100000,0), T(100000,0);
5 int tempo = 1;
6 list<int> adj[100000];
7
8 void artPoint(int u, int p) {
9
10     low[u] = T[u] = tempo++;
11     int children = 0;
12
13     for(int v: adj[u]) {
14
15         // cuidado com arestas paralelas
16         // se tiver nao podemos fazer assim
17
18         if(T[v] == 0) {
19
20             children++;
21             artPoint(v,u);
22             low[u] = min(low[v],low[u]);
23
24             if(p == -1 && children > 1) {
25                 ap[u] = true;

```

```

26     }
27
28     if(p != -1 && low[v] > T[u])
29         ap[u] = true;
30     } else if(v != p)
31         low[u] = min(low[u], T[v]);
32
33     }
34 }
35
36 int main() {
37
38     for(int i = 0; i < n; i++)
39         if(T[i] == 0)
40             artPoint(i,-1);
41 }

```

5.8. 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 };

```

5.9. 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         }
30     }
31 }

```

```

28
29     if(src == -1 && dest == -1)
30         return make_pair(true, pair<int, int>(src, dest));
31     else if(src != -1 && dest != -1)
32         return make_pair(true, pair<int, int>(src, dest));
33
34     return make_pair(false, pair<int, int>());
35 }
36
37 /// Builds the path/tour for directed graphs.
38 void build(const int u, vector<int> &tour, vector<vector<edge>> &adj,
39     vector<bool> &used) {
40     while(!adj[u].empty()) {
41         const edge e = adj[u].back();
42         if(!used[e.id]) {
43             used[e.id] = true;
44             adj[u].pop_back();
45             build(e.v, tour, adj, used);
46         } else
47             adj[u].pop_back();
48     }
49     tour.push_back(u);
50 }
51
52 /// Auxiliary function to build the eulerian tour/path.
53 vector<int> set_build(vector<vector<edge>> &adj, const int E, const int
54     first) {
55     vector<int> path;
56     vector<bool> used(E + 3);
57
58     build(first, path, adj, used);
59
60     for(int i = 0; i < adj.size(); i++)
61         // if there are some remaining edges, it's not possible to build the
62         // tour.
63         if(adj[i].size())
64             return vector<int>();
65
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 ///
74 /// Time complexity: O(V * (log V) + E)
75 bool has_euler_tour_directed(const vector<vector<edge>> &adj, const
76     vector<int> &in_degree) {
77     const pair<bool, pair<int, int>> aux = detail::check_both_directed(adj,
78         in_degree);
79     const bool valid = aux.first;
80     const int src = aux.second.first;
81     const int dest = aux.second.second;
82     return (valid && src == -1 && dest == -1);
83 }
84
85 /// A directed graph has an eulerian path/tour if has:
86 /// - One vertex v such that out_degree[v] - in_degree[v] == 1
87 /// - One vertex v such that in_degree[v] - out_degree[v] == 1
88 /// - The remaining vertices v such that in_degree[v] == out_degree[v]
89 /// or
90 /// - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR

```

```

87  ///
88  /// Returns a boolean value that indicates whether there's a path or not.
89  /// If there's a valid path it also returns two numbers: the source and
    the destination.
90  /// If the source and destination can be an arbitrary vertex it will
    return the pair (-1, -1)
91  /// for the source and destination (it means the contains an eulerian
    tour).
92  ///
93  /// Time complexity: O(V + E)
94  pair<bool, pair<int, int>> has_euler_path_directed(const
    vector<vector<edge>> &adj, const vector<int> &in_degree) {
95      return detail::check_both_directed(adj, in_degree);
96  }
97
98  /// Returns the euler path. If the graph doesn't have an euler path it
    returns an empty vector.
99  ///
100  /// Time Complexity: O(V + E) for directed, O(V * log(V) + E) for
    undirected.
101  /// Time Complexity: O(adj.size() + sum(adj[i].size()))
102  vector<int> get_euler_path_directed(const int E, vector<vector<edge>>
    &adj, const vector<int> &in_degree) {
103      const pair<bool, pair<int, int>> aux = has_euler_path_directed(adj,
        in_degree);
104      const bool valid = aux.first;
105      const int src = aux.second.first;
106      const int dest = aux.second.second;
107
108      if(!valid)
109          return vector<int>();
110
111      int first;
112      if(src != -1)
113          first = src;
114      else {
115          first = 0;
116          while(adj[first].empty())
117              first++;
118      }
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
    returns an empty vector.
124  ///
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>>
    &adj, const vector<int> &in_degree) {
128      const bool valid = has_euler_tour_directed(adj, in_degree);
129
130      if(!valid)
131          return vector<int>();
132
133      int first = 0;
134      while(adj[first].empty())
135          first++;
136
137      return detail::set_build(adj, E, first);
138  }
139
140  /// The graph has a tour that passes to every edge exactly once and gets
    // back to the first edge on the tour.
141

```

```

142  ///
143  /// A graph with an euler path has zero odd degree vertex.
144  ///
145  /// Time Complexity: O(V)
146  bool has_euler_tour_undirected(const vector<int> &degree) {
147      for(int i = 0; i < degree.size(); i++)
148          if(degree[i] & 1)
149              return false;
150      return true;
151  }
152
153  /// The graph has a path that passes to every edge exactly once.
154  /// It doesn't necessarily gets back to the beginning.
155  ///
156  /// A graph with an euler path has two or zero (tour) odd degree vertices.
157  ///
158  /// Returns a pair with the startpoint/endpoint of the path.
159  ///
160  /// Time Complexity: O(V)
161  pair<bool, pair<int, int>> has_euler_path_undirected(const vector<int>
    &degree) {
162      vector<int> odd_degree;
163      for(int i = 0; i < degree.size(); i++)
164          if(degree[i] & 1)
165              odd_degree.pb(i);
166
167      if(odd_degree.size() == 0)
168          return make_pair(true, make_pair(-1, -1));
169      else if (odd_degree.size() == 2)
170          return make_pair(true, make_pair(odd_degree.front(),
            odd_degree.back()));
171      else
172          return make_pair(false, pair<int, int>());
173  }
174
175  vector<int> get_euler_tour_undirected(const int E, const vector<int>
    &degree, vector<vector<edge>> &adj) {
176      if(!has_euler_tour_undirected(degree))
177          return vector<int>();
178
179      int first = 0;
180      while(adj[first].empty())
181          first++;
182
183      return detail::set_build(adj, E, first);
184  }
185
186  /// Returns the euler tour. If the graph doesn't have an euler tour it
    returns an empty vector.
187  ///
188  /// Time Complexity: O(V + E)
189  /// Time Complexity: O(adj.size() + sum(adj[i].size()))
190  vector<int> get_euler_path_undirected(const int E, const vector<int>
    &degree, vector<vector<edge>> &adj) {
191      auto aux = has_euler_path_undirected(degree);
192      const bool valid = aux.first;
193      const int x = aux.second.first;
194      const int y = aux.second.second;
195
196      if(!valid)
197          return vector<int>();
198
199      int first;
200      if(x != -1) {
201          first = x;

```

```

202     adj[x].emplace_back(y, E + 1);
203     adj[y].emplace_back(x, E + 1);
204 } else {
205     first = 0;
206     while(adj[first].empty())
207         first++;
208 }
209
210 vector<int> ans = detail::set_build(adj, E, first);
211 reverse(ans.begin(), ans.end());
212 if(x != -1)
213     ans.pop_back();
214 return ans;
215 }
216 };

```

5.10. 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)
16 /// Space Complexity: O(n)
17 bool bellman_ford(vector<edge> &edges, int src, int n) {
18     // n = qtd of vertices, E = qtd de arestas
19
20     // To calculate the shortest path uncomment the line below
21     // vector<int> dist(n, INF);
22
23     // To check cycles uncomment the line below
24     // vector<int> dist(n, 0);
25
26     vector<int> pai(n, -1);
27     int E = edges.size();
28
29     dist[src] = 0;
30     // Relax all edges n - 1 times.
31     // A simple shortest path from src to any other vertex can have at-most n
32     // - 1 edges.
33     for (int i = 1; i <= n - 1; i++) {
34         for (int j = 0; j < E; j++) {
35             int u = edges[j].src;
36             int v = edges[j].dest;
37             int weight = edges[j].weight;
38             if (dist[u] != INF && dist[u] + weight < dist[v]) {
39                 dist[v] = dist[u] + weight;
40                 pai[v] = u;
41             }
42         }
43     }
44
45     // Check for NEGATIVE-WEIGHT CYCLES.

```

```

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

```

5.11. 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 // k
5 // characters in A and adds that character to the starting string. For
6 // example,
7 // if n=3 and k=2, then we construct the following graph:
8 //
9 //      - 1 -> (01) - 1 ->
10 //      ^ |
11 // 0 -> (00) 1 0 (11) <- 1
12 //      | v
13 //      <- 0 - (10) <- 0 -
14
15 // The node '01' is connected to node '11' through edge '1', as adding '1' to
16 // '01' (and removing the first character) gives us '11'.
17
18 // We can observe that every node in this graph has equal in-degree and
19 // out-degree, which means that a Eulerian circuit exists in this graph.
20
21 namespace graph {
22 namespace detail {
23 // Finding an valid eulerian path
24 void dfs(const string &node, const string &alphabet, set<string> &vis,
25         string &edges_order) {
26     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 }; // namespace detail
35
36 // Returns a string in which every string of the alphabet of size n appears
37 // in
38 // the resulting string exactly once.
39 //
40 // Time Complexity: O(alphabet.size() ^ n * log2(alphabet.size() ^ n))
41 string de_bruijn(const int n, const string &alphabet) {
42     set<string> vis;
43     string edges_order;
44
45     string starting_node = string(n - 1, alphabet.front());
46     detail::dfs(starting_node, alphabet, vis, edges_order);
47
48     return edges_order + starting_node;
49 }
50 }; // namespace graph

```

5.12. Dijkstra + Dij Graph

```

1 class Dijkstra {
2     private:
3         int src, dest;
4         int n;
5
6     private:
7         int calculate(vector<vector<ii>> &adj) {
8             dist.resize(this->n + 1, INF);
9             parent.resize(this->n + 1);
10            vector<int> vis(this->n + 1, 0);
11
12            for(int i = 0; i <= this->n; i++)
13                parent[i].pb(i);
14
15            priority_queue<ii, vector<ii>, greater<ii>> pq;
16            pq.push(make_pair(0, this->src));
17            dist[this->src] = 0;
18
19            while(!pq.empty()) {
20                int u = pq.top().ss;
21                pq.pop();
22                if(vis[u])
23                    continue;
24                vis[u] = true;
25
26                for (ii x: adj[u]) {
27                    int v = x.ff;
28                    int w = x.ss;
29
30                    if(dist[u] + w < dist[v]) {
31                        parent[v].clear();
32                        parent[v].pb(u);
33                        dist[v] = dist[u] + w;
34                        pq.push(ii(dist[v], v));

```

```

35                }
36                else if(dist[u] + w == dist[v]) {
37                    parent[v].pb(u);
38                }
39            }
40        }
41
42        return dist[dest];
43    }
44
45    // Use a vector vis in a DFS on the dijkstra graph
46    vector<vector<int>> gen_dij_graph() {
47        vector<vector<int>> dijkstra_graph(this->n + 1);
48        vector<bool> vis(this->n + 1);
49
50        queue<int> q;
51        q.push(this->dest);
52        while(!q.empty()) {
53            int v = q.front();
54            q.pop();
55
56            for(int u: parent[v]) {
57                if(u == v)
58                    continue;
59                dijkstra_graph[u].pb(v);
60                if(!vis[u]) {
61                    q.push(u);
62                    vis[u] = true;
63                }
64            }
65        }
66        return dijkstra_graph;
67    }
68
69    public:
70        int min_path;
71        vector<int> dist;
72        vector<vector<int>> parent;
73        vector<vector<int>> dij_graph;
74
75        Dijkstra(int n, int src, int dest, vector<vector<ii>> &adj) {
76            this->n = n;
77            this->src = src;
78            this->dest = dest;
79            this->min_path = this->calculate(adj);
80            // Generates the dijkstra graph with the parent vector
81            this->dij_graph = this->gen_dij_graph();
82            d_graph = this->dij_graph;
83        }
84
85        // Returns a path with minimum costs and a minimum length.
86        vector<int> get_min_path() {
87            vector<int> path;
88            vector<int> pai(this->n + 1, -1);
89            vector<int> d(this->n + 1, INF);
90
91            queue<int> q;
92            q.push(this->dest);
93            d[this->dest] = 0;
94
95            while(!q.empty()) {
96                int v = q.front();
97                q.pop();
98
99                for(int u: parent[v]) {

```

```

100     if(u == v)
101         continue;
102     if(d[v] + 1 < d[u]) {
103         d[u] = d[v] + 1;
104         pai[u] = v;
105         q.push(u);
106     }
107 }
108 }
109
110 int cur = this->src;
111 while(cur != -1) {
112     path.pb(cur);
113     cur = pai[cur];
114 }
115
116 return path;
117 }
118 };

```

5.13. Dinic (Max Flow)

```

1 // Created by Ubiratan Neto
2
3 struct Dinic {
4
5     struct FlowEdge {
6         int v, rev, c, cap;
7         bool is_rev;
8         FlowEdge() {}
9         FlowEdge(int v, int c, int cap, int rev, bool is_rev) : v(v), c(c),
10             cap(cap), rev(rev), is_rev(is_rev) {}
11     };
12
13     vector<vector<FlowEdge>> adj;
14     vector<int> level, used;
15     int src, snk, V;
16     int sz;
17     int max_flow;
18     bool calculated;
19     Dinic() {}
20     Dinic(int n) {
21         calculated = false;
22         src = 0;
23         snk = n+1;
24         adj.resize(n+2, vector<FlowEdge>());
25         level.resize(n+2);
26         used.resize(n+2);
27         sz = n+2;
28         V = n+2;
29         max_flow = 0;
30     }
31
32     void add_edge(int u, int v, int c) {
33         int id1 = adj[u].size();
34         int id2 = adj[v].size();
35         adj[u].pb(FlowEdge(v, c, c, id2, false));
36         adj[v].pb(FlowEdge(u, 0, 0, id1, true));
37     }
38
39     void add_to_src(int v, int c) {
40         adj[src].pb(FlowEdge(v, c, c, -1, false));
41     }

```

```

42 void add_to_snk(int u, int c) {
43     adj[u].pb(FlowEdge(snk, c, c, -1, false));
44 }
45
46 bool bfs() {
47     for(int i=0; i<sz; i++) {
48         level[i] = -1;
49     }
50
51     level[src] = 0;
52     queue<int> q; q.push(src);
53
54     while(!q.empty()) {
55         int cur = q.front();
56         q.pop();
57         for(FlowEdge e : adj[cur]) {
58             if(level[e.v] == -1 && e.c > 0) {
59                 level[e.v] = level[cur]+1;
60                 q.push(e.v);
61             }
62         }
63     }
64
65     return (level[snk] == -1 ? false : true);
66 }
67
68 int send_flow(int u, int flow) {
69     if(u == snk) return flow;
70
71     for(int &i = used[u]; i<adj[u].size(); i++) {
72         FlowEdge &e = adj[u][i];
73
74         if(level[u]+1 != level[e.v] || e.c <= 0) continue;
75
76         int new_flow = min(flow, e.c);
77         int adjusted_flow = send_flow(e.v, new_flow);
78
79         if(adjusted_flow > 0) {
80             e.c -= adjusted_flow;
81             if(e.rev != -1) adj[e.v][e.rev].c += adjusted_flow;
82             return adjusted_flow;
83         }
84     }
85
86     return 0;
87 }
88
89 int calculate() {
90     if(src == snk) {max_flow = -1; return -1;} //not sure if needed
91
92     max_flow = 0;
93
94     while(bfs()) {
95         for(int i=0; i<sz; i++) used[i] = 0;
96         while(int inc = send_flow(src, INF)) max_flow += inc;
97     }
98     calculated = true;
99
100     return max_flow;
101 }
102
103
104 vector<ii> mincut(vector<vector<int>> &mat_adj) {
105     assert(calculated);
106     int mat[sz][sz];

```

```

107  memset(mat, 0, sizeof mat);
108  for(int i = 0; i < V; i++)
109      for(FlowEdge x: adj[i])
110          mat[i][x.v] += x.c;
111
112  vector<bool> vis(sz);
113  queue<int> q;
114  q.push(src);
115  vis[src] = true;
116  while(!q.empty()){
117      int u = q.front();
118      q.pop();
119      for(int v = 0; v < sz; v++) {
120          if(mat[u][v] > 0 && !vis[v]) {
121              q.push(v);
122              vis[v] = true;
123          }
124      }
125  }
126
127  vector<ii> cut;
128  for(int i = 0; i < sz; i++)
129      for(int j = 0; j < sz; j++)
130          if(vis[i] && !vis[j])
131              // if there's an edge from i to j.
132              if(mat_adj[i][j] > 0)
133                  cut.emplace_back(i, j);
134
135  return cut;
136 }
137
138 vector<ii> min_edge_cover(){
139     bool covered[sz];
140     for(int i=0; i<sz; i++) covered[i] = false;
141     vector< ii > edge_cover;
142     for(int i=1; i<sz-1; i++){
143         for(FlowEdge e : adj[i]){
144             if(e.cap == 0 || e.v > sz-2) continue;
145             if(e.c == 0){
146                 edge_cover.pb(ii(i, e.v));
147                 covered[i] = true;
148                 covered[e.v] = true;
149                 break;
150             }
151         }
152     }
153     for(int i=1; i<sz-1; i++){
154         for(FlowEdge e : adj[i]){
155             if(e.cap == 0 || e.v > sz-2) continue;
156             if(e.c == 0) continue;
157             if(!covered[i] || !covered[e.v]){
158                 edge_cover.pb(ii(i, e.v));
159                 covered[i] = true;
160                 covered[e.v] = true;
161             }
162         }
163     }
164     return edge_cover;
165 }
166
167 vector<vector<int>> allFlow() {
168     assert(calculated);
169     vector<vector<int>> ret(V, vector<int>(V, 0));
170
171     for(int i = 0; i < V; i++) {

```

```

172         for(FlowEdge x: adj[i]) {
173             if(x.is_rev)
174                 continue;
175             // flow from vertex i to x.v
176             ret[i][x.v] += x.cap - x.c;
177         }
178     }
179
180     // for(int i = 0; i < V; i++) {
181     //     for(int j = 0; j < V; j++) {
182     //         cout << ret[i][j] << ' ';
183     //     }
184     //     cout << endl;
185     // }
186
187     return ret;
188 }
189
190 void dfs_build_path(int u, vector<int> &path, vector<vector<int>>
191                    &mat_flow, vector<vector<int>> &ans, vector<vector<int>> &adj) {
192     path.pb(u);
193
194     if(u == this->snk) {
195         ans.pb(path);
196         return;
197     }
198     for(int v: adj[u]) {
199         if(mat_flow[u][v]) {
200             mat_flow[u][v]--;
201             dfs_build_path(v, path, mat_flow, ans, adj);
202             return;
203         }
204     }
205 }
206
207 vector<vector<int>> get_all_paths(vector<vector<int>> &adj) {
208     assert(calculated);
209
210     vector<vector<int>> mat_flow = allFlow();
211     vector<vector<int>> ans;
212     ans.reserve(max_flow);
213
214     for(int i = 0; i < max_flow; i++) {
215         vector<int> path;
216         path.reserve(V);
217         dfs_build_path(this->src, path, mat_flow, ans, adj);
218     }
219
220     return ans;
221 }
222 };

```

5.14. Functional Graph

```

1  // Based on:
2     http://maratona.ic.unicamp.br/MaratonaVerao2020/lecture-b/20200122.pdf
3
4  class Functional_Graph {
5      // FOR DIRECTED GRAPH
6  private:
7      void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis) {
8          int id_cycle = cycle_cnt++;
9          int cur_id = 0;

```

```

9     this->first[id_cycle] = u;
10
11     while(!vis[u]) {
12         vis[u] = true;
13
14         this->cycle[id_cycle].push_back(u);
15
16         this->in_cycle[u] = true;
17         this->cycle_id[u] = id_cycle;
18         this->id_in_cycle[u] = cur_id;
19         this->near_in_cycle[u] = u;
20         this->id_near_cycle[u] = id_cycle;
21         this->cycle_dist[u] = 0;
22
23         u = nxt[u];
24         cur_id++;
25     }
26 }
27
28 // Time Complexity: O(V)
29 void build(int n, int indexed_from, vector<int> &nxt, vector<int>
    &in_degree) {
30     queue<int> q;
31     vector<bool> vis(n + indexed_from);
32     for(int i = indexed_from; i < n + indexed_from; i++) {
33         if(in_degree[i] == 0) {
34             q.push(i);
35             vis[i] = true;
36         }
37     }
38
39     vector<int> process_order;
40     process_order.reserve(n + indexed_from);
41     while(!q.empty()) {
42         int u = q.front();
43         q.pop();
44
45         process_order.push_back(u);
46
47         if(--in_degree[nxt[u]] == 0) {
48             q.push(nxt[u]);
49             vis[nxt[u]] = true;
50         }
51     }
52
53     int cycle_cnt = 0;
54     for(int i = indexed_from; i < n + indexed_from; i++)
55         if(!vis[i])
56             compute_cycle(i, nxt, vis);
57
58     for(int i = (int)process_order.size() - 1; i >= 0; i--) {
59         int u = process_order[i];
60
61         this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
62         this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
63         this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
64     }
65 }
66
67 void allocate(int n, int indexed_from) {
68     this->cycle.resize(n + indexed_from);
69     this->first.resize(n + indexed_from);
70
71     this->in_cycle.resize(n + indexed_from, false);
72     this->cycle_id.resize(n + indexed_from, -1);

```

```

73     this->id_in_cycle.resize(n + indexed_from, -1);
74     this->near_in_cycle.resize(n + indexed_from);
75     this->id_near_cycle.resize(n + indexed_from);
76     this->cycle_dist.resize(n + indexed_from);
77 }
78
79 public:
80     Functional_Graph(int n, int indexed_from, vector<int> &nxt, vector<int>
        &in_degree) {
81         this->allocate(n, indexed_from);
82         this->build(n, indexed_from, nxt, in_degree);
83     }
84
85     // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
86
87     // number of cycles
88     int cycle_cnt = 0;
89     // Vertices present in the i-th cycle.
90     vector<vector<int>> cycle;
91     // first vertex of the i-th cycle
92     vector<int> first;
93
94     // The i-th vertex is present in any cycle?
95     vector<bool> in_cycle;
96     // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
        cycle.
97     vector<int> cycle_id;
98     // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
        belong to any cycle.
99     vector<int> id_in_cycle;
100    // Represents the id of the nearest vertex present in a cycle.
101    vector<int> near_in_cycle;
102    // Represents the id of the nearest cycle.
103    vector<int> id_near_cycle;
104    // Distance to the nearest cycle.
105    vector<int> cycle_dist;
106    // Represent the id of the component of the vertex.
107    // Equal to id_near_cycle
108    vector<int> &comp = id_near_cycle;
109 };
110
111 class Functional_Graph {
112     // FOR UNDIRECTED GRAPH
113 private:
114     void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis,
        vector<vector<int>> &adj) {
115         int id_cycle = cycle_cnt++;
116         int cur_id = 0;
117         this->first[id_cycle] = u;
118
119         while(!vis[u]) {
120             vis[u] = true;
121
122             this->cycle[id_cycle].push_back(u);
123             nxt[u] = find_nxt(u, vis, adj);
124             if(nxt[u] == -1)
125                 nxt[u] = this->first[id_cycle];
126
127             this->in_cycle[u] = true;
128             this->cycle_id[u] = id_cycle;
129             this->id_in_cycle[u] = cur_id;
130             this->near_in_cycle[u] = u;
131             this->id_near_cycle[u] = id_cycle;
132             this->cycle_dist[u] = 0;
133         }

```



```

134     u = nxt[u];
135     cur_id++;
136 }
137 }
138
139 int find_nxt(int u, vector<bool> &vis, vector<vector<int>> &adj) {
140     for(int v: adj[u])
141         if(!vis[v])
142             return v;
143     return -1;
144 }
145
146 // Time Complexity: O(V + E)
147 void build(int n, int indexed_from, vector<int> &degree,
148           vector<vector<int>> &adj) {
149     queue<int> q;
150     vector<bool> vis(n + indexed_from, false);
151     vector<int> nxt(n + indexed_from);
152     for(int i = indexed_from; i < n + indexed_from; i++) {
153         if(adj[i].size() == 1) {
154             q.push(i);
155             vis[i] = true;
156         }
157     }
158     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 public:
199     Functional_Graph(int n, int indexed_from, vector<int> degree,
200                     vector<vector<int>> &adj) {
201         this->allocate(n, indexed_from);
202         this->build(n, indexed_from, degree, adj);
203     }
204
205     // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
206
207     // number of cycles
208     int cycle_cnt = 0;
209     // Vertices present in the i-th cycle.
210     vector<vector<int>> cycle;
211     // first vertex of the i-th cycle
212     vector<int> first;
213
214     // The i-th vertex is present in any cycle?
215     vector<bool> in_cycle;
216     // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
217     // cycle.
218     vector<int> cycle_id;
219     // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
220     // belong to any cycle.
221     vector<int> id_in_cycle;
222     // Represents the id of the nearest vertex present in a cycle.
223     vector<int> near_in_cycle;
224     // Represents the id of the nearest cycle.
225     vector<int> id_near_cycle;
226     // Distance to the nearest cycle.
227     vector<int> cycle_dist;
228     // Represent the id of the component of the vertex.
229     // Equal to id_near_cycle
230     vector<int> &comp = id_near_cycle;
231 };

```

5.15. Hld

```

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

```

```

26
27     int maxx = -1, idx = -1;
28     for(const int v: adj[u]) {
29         if(v == p)
30             continue;
31         if(sz[v] > maxx) {
32             maxx = sz[v];
33             idx = v;
34         }
35     }
36
37     if(idx != -1)
38         this->dfs(idx, id, u, adj);
39
40     for(const int v: adj[u]) {
41         if(v == idx || v == p)
42             continue;
43         this->dfs(v, this->number_of_chains++, u, adj);
44     }
45 }
46
47 public:
48     /// Builds the paths.
49     ///
50     /// Time Complexity: O(n)
51     HLD(const vector<vector<int>> &adj) {
52         this->n = adj.size();
53         this->chain_head.resize(this->n + 1, -1);
54         this->id_in_chain.resize(this->n + 1, -1);
55         this->chain_id.resize(this->n + 1, -1);
56         this->sz.resize(this->n + 1);
57         this->chain_size.resize(this->n + 1);
58         this->parent.resize(this->n + 1, -1);
59         this->get_sz(1, -1, adj);
60         this->dfs(1, 0, -1, adj);
61     }
62
63     // the chains are indexed from 0
64     int number_of_chains = 1;
65     // topmost node of the chain
66     vector<int> chain_head;
67     // id of the i-th node in his chain
68     vector<int> id_in_chain;
69     // id of the chain that the i-th node belongs
70     vector<int> chain_id;
71     // size of the i-th chain
72     vector<int> chain_size;
73     // parent of the i-th node, -1 for root
74     vector<int> parent;
75 };

```

5.16. Kruskal + Dsu

```

1 class DSU {
2
3 public:
4
5     vector<int> root;
6     vector<int> sz;
7
8     DSU(int n) {
9         this->root.resize(n + 1);
10        iota(this->root.begin(), this->root.begin() + n + 1, 0);
11        this->sz.resize(n + 1, 1);

```

```

12    }
13
14    int Find(int x) {
15        if(this->root[x] == x)
16            return x;
17        return this->root[x] = this->Find(this->root[x]);
18    }
19
20    bool Union(int p, int q) {
21
22        p = this->Find(p), q = this->Find(q);
23
24        if(p == q)
25            return false;
26
27        if(this->sz[p] > this->sz[q]) {
28            this->root[q] = p;
29            this->sz[p] += this->sz[q];
30        } else {
31            this->root[p] = q;
32            this->sz[q] += this->sz[p];
33        }
34
35        return true;
36    }
37
38 };
39
40 struct edge {
41     int u, v, w;
42     edge() {}
43     edge(int u, int v, int w) : u(u), v(v), w(w) {}
44
45     bool operator<(const edge &a) const {
46         return w < a.w;
47     }
48 };
49
50 int kruskal(int n, vector<edge>& edges) {
51
52     DSU dsu(n);
53
54     sort(edges.begin(), edges.end());
55
56     int weight = 0;
57     for(int i = 0; i < (int)edges.size(); i++) {
58         if(dsu.Union(edges[i].u, edges[i].v)) {
59             weight += edges[i].w;
60         }
61     }
62
63     // returns weight of mst
64     return weight;
65 }

```

5.17. Lca

```

1 // #define DIST
2 // #define COST
3 /// UNCOMMENT ALSO THE LINE BELOW FOR COST!
4
5 class LCA {
6 private:
7     int n;

```

```

8 // INDEXED from 0 or 1??
9 int indexed_from;
10 /// Store all log2 from 1 to n
11 vector<int> lg;
12 // level of the i-th node (height)
13 vector<int> level;
14 // matrix to store the ancestors of each node in power of 2 levels
15 vector<vector<int>> anc;
16
17 #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 // int NEUTRAL_VALUE = INF; // MIN COST
24 // int combine(const int a, const int b) {return min(a, b);}
25 vector<vector<int>> cost;
26 #endif
27
28 private:
29 void allocate() {
30     // initializes a matrix [n][lg n] with -1
31     this->build_log_array();
32     this->anc.resize(n + 1, vector<int>(lg[n] + 1, -1));
33     this->level.resize(n + 1, -1);
34
35     #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
46     for(int i = 2; i <= this->n; i++)
47         this->lg[i] = this->lg[i/2] + 1;
48 }
49
50 void build_anc() {
51     for(int j = 1; j < anc.front().size(); j++)
52         for(int i = 0; i < anc.size(); i++)
53             if(this->anc[i][j - 1] != -1) {
54                 this->anc[i][j] = this->anc[this->anc[i][j - 1]][j - 1];
55                 #ifdef COST
56                 this->cost[i][j] = combine(this->cost[i][j - 1],
57                 this->cost[anc[i][j - 1]][j - 1]);
58                 #endif
59             }
60 }
61
62 void build_weighted(const vector<vector<pair<int, int>>> &adj) {
63     this->dfs_LCA_weighted(this->indexed_from, -1, 1, 0, adj);
64
65     this->build_anc();
66 }
67
68 void dfs_LCA_weighted(const int u, const int p, const int l, const int d,
69 const vector<vector<pair<int, int>>> &adj) {
70     this->level[u] = l;
71     this->anc[u][0] = p;
72     #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
88     this->build_anc();
89 }
90
91 void dfs_LCA_unweighted(const int u, const int p, const int l, const int
92 d, const vector<vector<int>> &adj) {
93     this->level[u] = l;
94     this->anc[u][0] = p;
95     #ifdef DIST
96     this->dist[u] = d;
97     #endif
98
99     for(const int v: adj[u]) {
100         if(v == p)
101             continue;
102         this->dfs_LCA_unweighted(v, u, l + 1, d + 1, adj);
103     }
104 }
105
106 // go up k levels from x
107 int lca_go_up(int x, int k) {
108     for(int i = 0; k > 0; i++, k >>= 1)
109         if(k & 1) {
110             x = this->anc[x][i];
111             if(x == -1)
112                 return -1;
113         }
114     return x;
115 }
116
117 #ifdef COST
118 /// Query between the an ancestor of v (p) and v. It returns the
119 /// max/min edge between them.
120 int lca_query_cost_in_line(int v, int p) {
121     assert(this->level[v] >= this->level[p]);
122
123     int k = this->level[v] - this->level[p];
124     int ans = NEUTRAL_VALUE;
125
126     for(int i = 0; k > 0; i++, k >>= 1)
127         if(k & 1) {
128             ans = combine(ans, this->cost[v][i]);
129             v = this->anc[v][i];
130         }
131     return ans;
132 }
133 #endif
134

```

```

135
136 int get_lca(int a, int b) {
137     // a is below b
138     if(this->level[b] > this->level[a])
139         swap(a,b);
140
141     const int logg = lg[this->level[a]];
142
143     // putting a and b in the same level
144     for(int i = logg; i >= 0; i--)
145         if(this->level[a] - (1 << i) >= this->level[b])
146             a = this->anc[a][i];
147
148     if(a == b)
149         return a;
150
151     for(int i = logg; i >= 0; i--)
152         if(this->anc[a][i] != -1 && this->anc[a][i] != this->anc[b][i]) {
153             a = this->anc[a][i];
154             b = this->anc[b][i];
155         }
156
157     return anc[a][0];
158 }
159
160 public:
161     /// Builds an weighted graph.
162     ///
163     /// Time Complexity: O(n*log(n))
164     explicit LCA(const vector<vector<pair<int, int>>> &adj, const int
165         indexed_from) {
166         this->n = adj.size();
167         this->indexed_from = indexed_from;
168         this->allocate();
169
170         this->build_weighted(adj);
171     }
172
173     /// Builds an unweighted graph.
174     ///
175     /// Time Complexity: O(n*log(n))
176     explicit LCA(const vector<vector<int>> &adj, const int indexed_from) {
177         this->n = adj.size();
178         this->indexed_from = indexed_from;
179         this->allocate();
180
181         this->build_unweighted(adj);
182     }
183
184     /// Goes up k levels from v. If it passes the root, returns -1.
185     ///
186     /// Time Complexity: O(log(k))
187     int go_up(const int v, const int k) {
188         assert(indexed_from <= v); assert(v < this->n + indexed_from);
189
190         return this->lca_go_up(v, k);
191     }
192
193     /// Returns the parent of v in the LCA dfs from l.
194     ///
195     /// Time Complexity: O(1)
196     int parent(int v) {
197         assert(indexed_from <= v); assert(v < this->n + indexed_from);
198
199         return this->anc[v][0];

```

```

199     }
200
201     /// Returns the LCA of a and b.
202     ///
203     /// Time Complexity: O(log(n))
204     int query_lca(const int a, const int b) {
205         assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n +
206             indexed_from);
207
208         return this->get_lca(a, b);
209     }
210
211     #ifndef DIST
212     /// Returns the distance from a to b. When the graph is unweighted, it is
213     /// considered
214     /// 1 as the weight of the edges.
215     ///
216     /// Time Complexity: O(log(n))
217     int query_dist(const int a, const int b) {
218         assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n +
219             indexed_from);
220
221         return this->dist[a] + this->dist[b] - 2*this->dist[this->get_lca(a, b)];
222     }
223     #endif
224
225     #ifndef COST
226     /// Returns the max/min weight edge from a to b.
227     ///
228     /// Time Complexity: O(log(n))
229     int query_cost(const int a, const int b) {
230         assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n +
231             indexed_from);
232
233         const int l = this->query_lca(a, b);
234         return combine(this->lca_query_cost_in_line(a, l),
235             this->lca_query_cost_in_line(b, l));
236     }
237     #endif
238 };

```

5.18. Maximum Path Unweighted Graph

```

1  /// Returns the maximum path between the vertices 0 and n - 1 in a
2  unweighted graph.
3  ///
4  /// Time Complexity: O(V + E)
5  int maximum_path(int n) {
6      vector<int> top_order = topological_sort(n);
7      vector<int> pai(n, -1);
8      if(top_order.empty())
9          return -1;
10
11      vector<int> dp(n);
12      dp[0] = 1;
13      for(int u: top_order)
14          for(int v: adj[u])
15              if(dp[u] && dp[u] + 1 > dp[v]) {
16                  dp[v] = dp[u] + 1;
17                  pai[v] = u;
18              }
19
20      if(dp[n - 1] == 0)
21          return -1;

```

```

21 vector<int> path;
22 int cur = n - 1;
23 while (cur != -1) {
24     path.pb(cur);
25     cur = pai[cur];
26 }
27 reverse(path.begin(), path.end());
28
29 // cout << path.size() << endl;
30 // for(int x: path) {
31 //     cout << x + 1 << ' ';
32 // }
33 // cout << endl;
34
35 return dp[n - 1];
36 }

```

5.19. Number Of Different Spanning Trees In A Complete Graph

```

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

```

5.20. Number Of Ways To Make A Graph Connected

```

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

```

5.21. 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 }

```

5.22. 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 }

```

5.23. Pruffer Properties

- * After constructing the Prüfer code two vertices will remain. One of them is the highest vertex $n-1$, but nothing **else** can be said about the other one.
- * Each vertex appears in the Prüfer code exactly a fixed number of times - its degree minus one. This can be easily checked, since the degree will get smaller every time we record its label in the code, **and** we remove it once the degree is 1. For the two remaining vertices **this** fact is also **true**.

5.24. 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.

5.25. Shortest Cycle In A Graph

```

1 int bfs(int vt) {
2
3     vector<int> dist(MAXN, INF);
4     queue<pair<int, int>> q;
5
6     q.emplace(vt, -1);
7     dist[vt] = 0;
8
9     int ans = INF;
10    while (!q.empty()) {
11        pair<int, int> aux = q.front();
12        int u = aux.first, p = aux.second;
13        q.pop();
14
15        for (int v : adj[u]) {
16            if (v == p)
17                continue;
18            if (dist[v] < INF)
19                ans = min(ans, dist[u] + dist[v] + 1);
20            else {
21                dist[v] = dist[u] + 1;
22                q.emplace(v, u);
23            }
24        }
25    }
26
27    return ans;
28 }
29
30 /// Returns the shortest cycle in the graph
31 ///
32 /// Time Complexity: O(V^2)
33 int get_girth(int n) {
34     int ans = INF;
35     for (int u = 1; u <= n; u++)
36         ans = min(ans, bfs(u));
37     return ans;
38 }

```

5.26. Topological Sort

```

1 /// INDEXED BY ZERO
2 ///
3 /// Time Complexity: O(n)
4 vector<int> topological_sort(int n) {
5     vector<int> in_degree(n, 0);
6
7     for(int u = 0; u < n; u++)
8         for(int v: adj[u])
9             in_degree[v]++;
10
11     queue<int> q;
12     for(int i = 0; i < n; i++)
13         if(in_degree[i] == 0)
14             q.push(i);

```

```

15
16     int cnt = 0;
17     vector<int> top_order;
18     while(!q.empty()) {
19         int u = q.front();
20         q.pop();
21
22         top_order.push_back(u);
23         cnt++;
24
25         for(int v: adj[u])
26             if(--in_degree[v] == 0)
27                 q.push(v);
28     }
29
30     if(cnt != n) {
31         cerr << "There exists a cycle in the graph" << endl;
32         return vector<int>();
33     }
34
35     return top_order;
36 }

```

5.27. Tree Distance

```

1 vector<pair<int, int>> sub(MAXN, pair<int, int>(0, 0));
2
3 void subu(int u, int p) {
4     for (const pair<int, int> x : adj[u]) {
5         int v = x.first, w = x.second;
6         if (v == p)
7             continue;
8         subu(v, u);
9         if (sub[v].first + w > sub[u].first) {
10             swap(sub[u].first, sub[u].second);
11             sub[u].first = sub[v].first + w;
12         } else if (sub[v].first + w > sub[u].second) {
13             sub[u].second = sub[v].first + w;
14         }
15     }
16 }
17
18 /// Contains the maximum distance to the node i
19 vector<int> ans(MAXN);
20
21 void dfs(int u, int d, int p) {
22     ans[u] = max(d, sub[u].first);
23     for (const pair<int, int> x : adj[u]) {
24         int v = x.first, w = x.second;
25         if (v == p)
26             continue;
27         if (sub[v].first + w == ans[u]) {
28             dfs(v, max(d, sub[u].second) + w, u);
29         } else {
30             dfs(v, ans[u] + w, u);
31         }
32     }
33 }
34
35 // Returns the maximum tree distance
36 int solve() {
37     subu(0, -1);
38     dfs(0, 0, -1);
39     return *max_element(ans.begin(), ans.end());

```

40 }

6. Language Stuff

6.1. Binary String To Int

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

6.2. 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
```

6.3. Checagem Brute Force Com Solucao

```
1 $ g++ -std=c++11 gen.cpp && ./a.out > gen.out && g++ -std=c++11 brute.cpp &&
  ./a.out < gen.in > brute.out && g++ -std=c++11 sol.cpp && ./a.out <
  gen.in > sol.out && diff brute.out sol.out
```

6.4. Checagem De Bits

```
1 // OBS: SO FUNCIONA PARA INT (NAO FUNCIONA COM LONG LONG)
2 __builtin_popcount(int) -> Número de bits ativos;
3 __builtin_ctz(int) -> Número de zeros à direita
4 __builtin_clz(int) -> Número de zeros à esquerda
5 __builtin_parity(int) -> Retorna se a quantidade de uns é ímpar(1) ou par(0)
```

6.5. 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
```

6.6. 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 }
```

6.7. Escrita Em Arquivo

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

6.8. Gcd

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

6.9. Hipotenusa

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

6.10. 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
```

6.11. Int To String

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

6.12. Leitura De Arquivo

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

6.13. 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
```

6.14. 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));
```

6.15. Printf De Uma String

```
1 char buffer [50];
2 int n, a=5, b=3;
3 n=sprintf (buffer, "%d plus %d is %d", a, b, a+b);
4 printf ("%s] is a string %d chars long\n",buffer,n);
5 // Output:
6 // [5 plus 3 is 8] is a string 13 chars long
```

6.16. 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}

```

6.17. 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

```

6.18. 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

```

6.19. 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

```

6.20. Split Function

```

1 // SEPARA STRING POR UM DELIMITADOR
2 // EX: str=A-B-C split -> x = {A,B,C}
3 vector<string> split(const string &s, char delim) {
4     stringstream ss(s);
5     string item;
6     vector<string> tokens;
7     while (getline(ss, item, delim)) {
8         tokens.push_back(item);
9     }
10    return tokens;
11 }
12 int main () {
13     vector<string> x = split("cap-one-best-opinion-language", '-');
14     // x = {cap,one,best,opinion,language};
15 }

```

6.21. 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);

```

6.22. 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"

```

6.23. Width

```

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

```

6.24. Check Overflow

```

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

```

6.25. Readint

```

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

```

7. Math

7.1. Bell Numbers


```

1 int bellNumber(int n) {
2     int bell[n+1][n+1];
3     bell[0][0] = 1;
4     for (int i=1; i<=n; i++) {
5         // Explicitly fill for j = 0
6         bell[i][0] = bell[i-1][i-1];
7
8         // Fill for remaining values of j
9         for (int j=1; j<=i; j++)
10            bell[i][j] = bell[i-1][j-1] + bell[i][j-1];
11     }
12     return bell[n][0];
13 }

```

7.2. Checagem De Primalidade

```

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

```

7.3. Combinacao Ncr Mod Primo

```

1 inv[1] = 1;
2 for(int i = 2; i < m; ++i)
3     inv[i] = (m - (m/i) * inv[m%i] % m) % m;
4
5 factorial[0] = 1;
6 for (int i = 1; i <= MAXN; i++) {
7     factorial[i] = factorial[i - 1] * i % m;
8 }
9
10 int binomial_coefficient(int n, int k) {
11     return factorial[n] * inverse(factorial[k]) % m * inverse(factorial[n -
12     k]) % m;
13 }

```

7.4. Combinacao Ncr

```

1 // Returns value of Binomial Coefficient C(n, k)
2 int binomialCoeff(int n, int k) {
3     int res = 1;
4     // Since C(n, k) = C(n, n-k)
5     if (k > n - k)
6         k = n - k;
7     // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
8     for (int i = 0; i < k; ++i) {
9         res *= (n - i);
10        res /= (i + 1);
11    }
12    return res;
13 }

```

7.5. Compressao De Pontos

```

1 map<int, int> rev;
2 for(int x : arr) {
3     sl.insert(x);
4 }
5 vector<int> aux;
6 for(int x : sl) aux.pb(x);
7 for(int i=0; i<=n; i++){
8     int id = lower_bound(aux.begin(), aux.end(), arr[i]) - aux.begin();
9     rev[id] = arr[i];
10    arr[i] = id;
11 }

```

7.6. Equacao Diofantina

```

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

```

7.7. Euclides Estendido

```

1 int gcd(x,y);
2
3 //Ax + By = gcd(A,B)
4
5 void extendedEuclidian(int a,int b){
6
7     if(b==0){
8         gcd=a;
9         x=1;
10        y=0;
11    } else{
12        extendedEuclidian(b, a%b);
13
14        int temp = x;
15        x=y;
16        y = temp - (a/b)*y;
17    }
18 }

```

7.8. Euler Totient

```

1 int phi(int n) {
2     int result = n;
3     for (int i = 2; i * i <= n; i++) {
4         if (n % i == 0) {
5             while (n % i == 0)
6                 n /= i;
7             result -= result / i;
8         }
9     }
10
11     if (n > 1)
12         result -= result / n;
13     return result;
14 }

```

7.9. Fatoracao Multiplas Queries

```

1 //stor smallest prime factor for every num
2 int spf[MAXN];
3 // Calculating SPF (Smallest Prime Factor) for every number till MAXN.
4 // Time Complexity : O(nloglogn)
5 void sieve() {
6     spf[1] = 1;
7     for (int i=2; i<MAXN; i++)
8         // marking smallest prime factor for every number to be itself.
9         spf[i] = i;
10
11     // separatelyMarking spf for every even
12     // number as 2
13     for (int i=4; i<MAXN; i+=2)
14         spf[i] = 2;
15
16     for (int i=3; i*i<MAXN; i++) {
17         // checking if i is prime
18         if (spf[i] == i) {
19             // marking SPF for all numbers divisible by i
20             for (int j=i*i; j<MAXN; j+=i)
21                 // marking spf[j] if it is not previously marked
22                 if (spf[j]==0)
23                     spf[j] = i;
24         }
25     }
26 }
27 // A O(log n) function returning primefactorization
28 // by dividing by smallest prime factor at every step
29 vector<int> getFactorization(int x) {
30     vector<int> ret;
31     while (x != 1) {
32         ret.push_back(spf[x]);
33         x = x / spf[x];
34     }
35     return ret;
36 }

```

7.10. Fatoracao Simples

```

1 map<int, int> primeFactors(int n) {
2     set<int> ret;
3     while (n%2 == 0) {
4         m[2]++;
5         n = n/2;

```

```

6     }
7
8     int sq = sqrt(n);
9     for (int i = 3; i <= sq; i = i+2) {
10         while (n%i == 0) {
11             m[i]++;
12             n = n/i;
13         }
14         /* OBS1
15         IF(N < 1E7)
16             FATORE COM SPF
17         */
18     }
19
20     if (n > 2)
21         m[n]++;
22
23     return ret;
24 }

```

7.11. Inclusao-Exclusao

```

1 // |A ∪ B ∪ C| = |A| + |B| + |C| - |A ∩ B| - |A ∩ C| - |B ∩ C| + |A ∩ B ∩ C|
2
3 // EXEMPLO: Quantos números de 1 a 10^9 são múltiplos de 42, 54, 137 ou 201?
4
5 int f(vector<int> arr, int LIMIT) {
6
7     int n = arr.size();
8     int c = 0;
9
10    for (int mask = 1; mask < (1<<n); mask++) {
11        int lcm = 1;
12        for (int i = 0; i < n; i++)
13            if (mask & (1<<i))
14                lcm = lcm * arr[i] / __gcd(lcm, arr[i]);
15        // se o numero de conjuntos a unir for impar entao soma
16        if (__builtin_popcount(mask) % 2 == 1)
17            c += LIMIT / lcm;
18        else // se nao subtrai
19            c -= LIMIT / lcm;
20    }
21
22    return LIMIT - c;
23
24 }

```

7.12. Numero De Fatores

```

1 int calcFat(vector<int> fatores) {
2     int x = fatores[1];
3     auto lo = lower_bound(fatores.begin(), fatores.end(), x);
4     auto up = upper_bound(fatores.begin(), fatores.end(), x);
5     int fat = 1;
6     while (up != fatores.end()) {
7         fat *= (up - lo + 1);
8         lo = lower_bound(fatores.begin(), fatores.end(), *up);
9         up = upper_bound(fatores.begin(), fatores.end(), *up);
10    }
11    fat *= (up - lo + 1);
12    return fat;
13 }

```

7.13. Pollard Rho (Find A Divisor)

```

1  /* Function to calculate (base^exponent)%modulus */
2  int modular_pow(int base, int exponent,
3                  int modulus) {
4      /* initialize result */
5      int result = 1;
6      while (exponent > 0) {
7          /* if y is odd, multiply base with result */
8          if (exponent & 1)
9              result = (result * base) % modulus;
10         /* exponent = exponent/2 */
11         exponent = exponent >> 1;
12         /* base = base * base */
13         base = (base * base) % modulus;
14     }
15     return result;
16 }
17
18 /* method to return prime divisor for n */
19 int PollardRho(int n) {
20     /* initialize random seed */
21     srand (time(NULL));
22
23     /* no prime divisor for 1 */
24     if (n==1) return n;
25
26     /* even number means one of the divisors is 2 */
27     if (n % 2 == 0) return 2;
28
29     /* we will pick from the range [2, N) */
30     int x = (rand()%(n-2))+2;
31     int y = x;
32
33     /* the constant in f(x).
34     * Algorithm can be re-run with a different c
35     * if it throws failure for a composite. */
36     int c = (rand()%(n-1))+1;
37
38     /* Initialize candidate divisor (or result) */
39     int d = 1;
40
41     /* until the prime factor isn't obtained.
42     If n is prime, return n */
43     while (d==1) {
44         /* Tortoise Move: x(i+1) = f(x(i)) */
45         x = (modular_pow(x, 2, n) + c + n)%n;
46
47         /* Hare Move: y(i+1) = f(f(y(i))) */
48         y = (modular_pow(y, 2, n) + c + n)%n;
49         y = (modular_pow(y, 2, n) + c + n)%n;
50
51         /* check gcd of |x-y| and n */
52         d = __gcd(abs(x-y), n);
53
54         /* retry if the algorithm fails to find prime factor
55         * with chosen x and c */
56         if (d==n) return PollardRho(n);
57     }
58
59     return d;
60 }
61
62 /* driver function */
63 signed main() {

```

```

64     int n = 12;
65     printf("One of the divisors for %lld is %lld.",
66           n, PollardRho(n));
67     return 0;
68 }

```

7.14. Precomputar Combinacao Ncr

```

1  int C[1123][1123];
2
3  int mod(int n) {return n%((int)1e9+7);}
4
5  int nCr(int n, int k) {
6      for(int i = 0; i <= n; i++) {
7          for(int j = 0; j <= min(i,k); j++) {
8              if(j == 0 || j == i) {
9                  C[i][j] = 1;
10             } else {
11                 C[i][j]=mod(C[i-1][j-1]+C[i-1][j]);
12             }
13         }
14     }
15 }

```

7.15. Teorema Chines Do Resto

```

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     // debug(x1);
26
27     return x1;
28 }
29
30 // k is size of num[] and rem[]. Returns the smallest
31 // number x such that:
32 // x % num[0] = rem[0],
33 // x % num[1] = rem[1],
34 // .....
35 // x % num[k-2] = rem[k-1]
36 // Assumption: Numbers in num[] are pairwise coprimes
37 // (gcd for every pair is 1)
38 int findMinX(int num[], int rem[], 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 // Initialize result
44 int result = 0;
45
46 // Apply above formula
47 for (int i = 0; i < k; i++){
48     int pp = prod / num[i];
49     // debug(pp);
50     int iv = inv(pp, num[i]);
51     if(iv == INF)
52         return INF;
53     result+=rem[i]*inv(pp,num[i])*pp;
54 }
55
56 // IF IS NOT VALID RETURN INF
57 return (result % prod == 0 ? INF: result % prod);
58 }

```

7.16. Binary Exponentiation

```

1 int power(const int x, const int p, const int MOD = ((int)1e9 + 7)) {
2     if(p == 0)
3         return 1%MOD;
4     if(p == 1)
5         return x%MOD;
6     int res = power(x, p/2, MOD);
7     res = (long long)res*res%MOD;
8     if(p&1)
9         res = (long long)res*x%MOD;
10    return res;
11 }

```

7.17. Divisors

```

1 // OBS: EACH NUMBER HAS AT MOST  $\sqrt[3]{N}$  DIVISORS
2 vector<int> divisors(int n) {
3     vector<int> ans;
4     for (int i=1; i * i <= n; i++) {
5         if (n%i==0) {
6             // If divisors are equal, print only one
7             if (n/i == i)
8                 ans.pb(i);
9             else // Otherwise print both
10                ans.pb(i), ans.pb(n/i);
11         }
12     }
13     return ans;
14 }

```

7.18. Matrix Exponentiation

```

1 namespace matrix {
2 #define Matrix vector<vector<int>>
3 const int MOD = 1e9 + 7;
4
5 /// Creates an n x n identity matrix.
6 ///
7 /// Time Complexity: O(n*n)

```

```

8 Matrix identity(const int n) {
9     assert(n > 0);
10
11     Matrix mat_identity(n, vector<int>(n, 0));
12
13     for (int i = 0; i < n; i++)
14         mat_identity[i][i] = 1;
15
16     return mat_identity;
17 }
18
19 /// Multiplies matrices a and b.
20 ///
21 /// Time Complexity: O(mat.size() ^ 3)
22 Matrix mult(const Matrix &a, const Matrix &b) {
23     assert(a.front().size() == b.size());
24
25     Matrix ans(a.size(), vector<int>(b.front().size(), 0));
26     for (int i = 0; i < ans.size(); i++)
27         for (int j = 0; j < ans.front().size(); j++)
28             for (int k = 0; k < a.front().size(); k++)
29                 ans[i][j] = (ans[i][j] + a[i][k] * b[k][j]) % MOD;
30
31     return ans;
32 }
33
34 /// Exponentiates the matrix mat to the power of p.
35 ///
36 /// Time Complexity: O((mat.size() ^ 3) * log2(p))
37 Matrix expo(Matrix &mat, int p) {
38     assert(p >= 0);
39
40     Matrix ans = identity(mat.size());
41     Matrix cur_power;
42     cur_power.swap(mat);
43
44     while (p) {
45         if (p & 1)
46             ans = mult(ans, cur_power);
47
48         cur_power = mult(cur_power, cur_power);
49         p >>= 1;
50     }
51
52     return ans;
53 }
54 }; // namespace matrix

```

7.19. Modular Inverse

```

1 int gcd(int a, int b) {
2     if (a == 0)
3         return b;
4     return gcd(b%a, a);
5 }
6
7 int power(int x, int p, int MOD) {
8     if(p == 0)
9         return 1%MOD;
10    if(p == 1)
11        return x%MOD;
12    int res = power(x, p/2, MOD);
13    res = (long long)res*res%MOD;
14    if(p&1)

```

```

15     res = (long long)res*x%MOD;
16     return res;
17 }
18
19 // A*B = x (mod m)
20 // B = x * A^(-1)
21 // Function to find modular inverse of a under modulo m
22 // Assumption: m is prime
23 int modInverse(int a, int mod) {
24     int g = gcd(a, mod);
25     if (g != 1)
26         return -1;
27     else
28         // If a and m are relatively prime, then modulo inverse
29         // is a^(m-2) mod m
30         return power(a, mod-2, mod);
31 }

```

7.20. Sieve + Segmented Sieve

```

1 const int MAXN = 1e6;
2
3 /// Contains all the primes in the segments
4 vector<int> segPrimes;
5 bitset<MAXN+5> primesInSeg;
6
7 /// smallest prime factor
8 int spf[MAXN+5];
9
10 vector<int> primes;
11 bitset<MAXN+5> isPrime;
12
13 void sieve(int n = MAXN + 2) {
14
15     for(int i = 0; i <= n; i++)
16         spf[i] = i;
17
18     isPrime.set();
19     for(int i = 2; i <= n; i++) {
20         if(!isPrime[i])
21             continue;
22
23         for(int j = i*i; j <= n; j+=i) {
24             isPrime[j] = false;
25             spf[j] = min(i, spf[j]);
26         }
27         primes.pb(i);
28     }
29 }
30
31 vector<int> getFactorization(int x) {
32     vector<int> ret;
33     while (x != 1) {
34         ret.push_back(spf[x]);
35         x = x / spf[x];
36     }
37     return ret;
38 }
39
40 /// Gets all primes from l to r
41 void segSieve(int l, int r) {
42     // primes from l to r
43     // transferred to 0..(l-r)
44     segPrimes.clear();

```

```

45     primesInSeg.set();
46     int sq = sqrt(r) + 5;
47
48     for(int p: primes) {
49         if(p > sq)
50             break;
51
52         for(int i = l - l%p; i <= r; i += p) {
53             if(i - l < 0)
54                 continue;
55
56             // if i is less than 1e6, it could be checked in the
57             // array of the sieve
58             if(i >= (int)1e6 || !isPrime[i])
59                 primesInSeg[i-l] = false;
60         }
61     }
62
63     for(int i = 0; i < r-l+1; i++) {
64         if(primesInSeg[i])
65             segPrimes.pb(i+l);
66     }
67 }

```

8. Miscellaneous

8.1. 2-Sat

```

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     int id = 0;
52     while (s.empty() == false) {
53         int v = s.top();
54         s.pop();
55
56         if (comp[v] == -1)
57             dfs_trans(v, id++);
58     }
59 }
60
61 public:
62     // number of the component of the i-th vertex
63     // it's always indexed from 0
64     vector<int> comp;
65     // the i-th vector contains the vertices that belong to the i-th scc
66     // it's always indexed from 0
67     vector<vector<int>> scc;
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 };
79
80 // OBS: INDEXED FROM 0
81 class SAT {
82
83 private:
84     vector<vector<int>> adj;
85     int n;
86
87 public:
88     vector<bool> ans;
89
90     SAT(int n) {
91         this->n = n;
92         adj.resize(2 * n);
93         ans.resize(n);
94     }
95
96     // (X v Y) = (X -> ~Y) & (~X -> Y)
97 void add_or(int x, bool pos_x, int y, bool pos_y) {
98     assert(0 <= x);
99     assert(x < n);
100    assert(0 <= y);
101    assert(x < n);

```

```

102    adj[(x << 1) ^ pos_x].pb((y << 1) ^ (pos_y ^ 1));
103    adj[(y << 1) ^ pos_y].pb((x << 1) ^ (pos_x ^ 1));
104 }
105
106 // (X xor Y) = (X v Y) & (~X v ~Y)
107 // for this function the result is always 0 1 or 1 0
108 void add_xor(int x, bool pos_x, int y, bool pos_y) {
109     assert(0 <= x);
110     assert(x < n);
111     assert(0 <= y);
112     assert(x < n);
113     add_or(x, y, pos_x, pos_y);
114     add_or(x, y, pos_x ^ 1, pos_y ^ 1);
115 }
116
117 bool check() {
118     SCC scc(2 * n, 0, adj);
119
120     for (int i = 0; i < n; i++) {
121         if (scc.comp[(i << 1) | 1] == scc.comp[(i << 1) | 0])
122             return false;
123         ans[i] = (scc.comp[(i << 1) | 1] < scc.comp[(i << 1) | 0]);
124     }
125
126     return true;
127 }
128 };

```

8.2. 3Sum Problem

```

1 // vetor arr e valor x, a soma de três valores desse vetor deve ser igual a x
2
3 bool sum3(int arr[], int x, int n) {
4     sort(arr, arr + n);
5     for(int i = 0; i < n-2; i++) {
6         int l = i+1, r = n-1;
7         /* 2SUM problem -> ponteiro que aponta para o primeiro e ultimo da
           sequencia e caso a soma for menor do que x adianta em uma casa o
           ponteiro da esquerda caso seja maior diminui em uma casa o ponteiro da
           direita */
8         while(l < r) {
9             if(arr[i] + arr[l] + arr[r] == x) {
10                 return true;
11             } else if(arr[i] + arr[l] + arr[r] < x)
12                 l++;
13             else
14                 r--;
15         }
16     }
17     return false;
18 }

```

8.3. Fibonacci Matrix Exponentiation

```

1 int fib (int n) {
2     long long fib[2][2]= {{1,1},{1,0}};
3     int ret[2][2]= {{1,0},{0,1}};
4     int tmp[2][2]= {{0,0},{0,0}};
5     int i,j,k;
6     while(n) {
7         if(n&1) {
8             memset(tmp,0,sizeof tmp);
9             for(i=0; i<2; i++)

```

```

10     for(j=0; j<2; j++)
11         for(k=0; k<2; k++)
12             tmp[i][j]=(tmp[i][j]+ret[i][k]*fib[k][j]);
13     for(i=0; i<2; i++)
14         for(j=0; j<2; j++)
15             ret[i][j]=tmp[i][j];
16 }
17 memset(tmp,0,sizeof tmp);
18 for(i=0; i<2; i++)
19     for(j=0; j<2; j++)
20         for(k=0; k<2; k++)
21             tmp[i][j]=(tmp[i][j]+fib[i][k]*fib[k][j]);
22 for(i=0; i<2; i++)
23     for(j=0; j<2; j++)
24         fib[i][j]=tmp[i][j];
25 n/=2;
26 }
27 return (ret[0][1]);
28 }

```

8.4. Infix To Prefix

```

1 int main() {
2     map<char,int> prec;
3     stack<char> op;
4
5     string postfix;
6     string infix;
7     cin >> infix;
8
9     prec['+'] = prec['-'] = 1;
10    prec['*'] = prec['/'] = 2;
11    prec['^'] = 3;
12    for(int i = 0; i < infix.length(); i++) {
13        char x = infix[i];
14        if('0' <= x && x <= '9') {
15            for(i;i < infix.length() && ('0' <= infix[i] && infix[i] <= '9');i++)
16                postfix += infix[i];
17            i--;
18        } else if(('a' <= x && x <= 'z') || ('A' <= x && x <= 'Z')) {
19            postfix += x;
20        } else if (x == '(')
21            op.push('(');
22        else if(x == ')') {
23            while(!op.empty() && op.top() != '(') {
24                postfix += op.top();
25                op.pop();
26            }
27            op.pop();
28        } else {
29            while(!op.empty() && prec[op.top()] >= prec[x]) {
30                postfix += op.top();
31                op.pop();
32            }
33            op.push(x);
34        }
35    }
36    while(!op.empty()) {
37        postfix += op.top();
38        op.pop();
39    }
40    cout << postfix << endl;
41 }

```

8.5. 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.

8.6. Kadane (Maior Soma Num Vetor)

```

1 int kadane(int arr[], int l) {
2
3     int soma, total;
4     soma = total = arr[0];
5
6     for(int i = 1; i < l; i++) {
7         soma = max(arr[i], arr[i] +soma);
8         if(soma > total)
9             total = soma;
10    }
11    return total;
12
13 }

```

8.7. Kadane 2D

```

1
2 // Program to find maximum sum subarray in a given 2D array
3 #include <stdio.h>
4 #include <string.h>
5 #include <limits.h>
6 int mat[1001][1001]
7 int ROW = 1000, COL = 1000;
8
9 // Implementation of Kadane's algorithm for 1D array. The function
10 // returns the maximum sum and stores starting and ending indexes of the
11 // maximum sum subarray at addresses pointed by start and finish pointers
12 // respectively.
13 int kadane(int* arr, int* start, int* finish, int n) {
14     // initialize sum, maxSum and
15     int sum = 0, maxSum = INT_MIN, i;
16
17     // Just some initial value to check for all negative values case
18     *finish = -1;
19
20     // local variable
21     int local_start = 0;
22
23     for (i = 0; i < n; ++i) {
24         sum += arr[i];
25         if (sum < 0) {
26             sum = 0;
27             local_start = i+1;
28         }
29         else if (sum > maxSum){
30             maxSum = sum;
31             *start = local_start;
32             *finish = i;
33         }
34     }
35
36     // There is at-least one non-negative number
37     if (*finish != -1)
38         return maxSum;
39 }

```

```

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

```

8.8. 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 }

```

8.9. Sliding Window Minimum

```

1 // mínimo num vetor arr de arr[0] ... arr[k-1], arr[1] ... arr[k], arr[2]
2 // ... arr[k+1]
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(arr[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 }

```

8.10. 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 }

```

8.11. Kadane (Segment Tree)

```

1 struct Node {
2     int pref, suf, tot, best;
3     Node () {}
4     Node(int pref, int suf, int tot, int best) : pref(pref), suf(suf),
5         tot(tot), best(best) {}
6 };
7
8 const int MAXN = 2E5 + 10;
9 Node tree[5*MAXN];
10 int arr[MAXN];
11
12 Node query(const int l, const int r, const int i, const int j, const int
13     pos) {
14     if(l > r || l > j || r < i)
15         return Node(-INF, -INF, -INF, -INF);
16
17     if(i <= l && r <= j)
18         return Node(tree[pos].pref, tree[pos].suf, tree[pos].tot,
19             tree[pos].best);
20
21     int mid = (l + r) / 2;
22     Node left = query(l, mid, i, j, 2*pos+1), right = query(mid+1, r, i, j, 2*pos+2);
23     Node x;
24     x.pref = max({left.pref, left.tot, left.tot + right.pref});
25     x.suf = max({right.suf, right.tot, right.tot + left.suf});
26     x.tot = left.tot + right.tot;
27     x.best = max({left.best, right.best, left.suf + right.pref});
28     return x;
29 }
30
31 // Update arr[idx] to v
32 // ITS NOT DELTA!!!
33 void update(int l, int r, const int idx, const int v, const int pos) {
34     if(l > r || l > idx || r < idx)
35         return;
36
37     if(l == idx && r == idx) {
38         tree[pos] = Node(v, v, v, v);
39         return;
40     }
41
42     int mid = (l + r)/2;
43     update(l, mid, idx, v, 2*pos+1); update(mid+1, r, idx, v, 2*pos+2);
44     l = 2*pos+1, r = 2*pos+2;
45     tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
46         tree[r].pref});
47     tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
48     tree[pos].tot = tree[l].tot + tree[r].tot;

```

```

46     tree[pos].best = max({tree[l].best, tree[r].best, tree[l].suf +
47         tree[r].pref});
48 }
49
50 void build(int l, int r, const int pos) {
51     if(l == r) {
52         tree[pos] = Node(arr[l], arr[l], arr[l], arr[l]);
53         return;
54     }
55
56     int mid = (l + r)/2;
57     build(l, mid, 2*pos+1); build(mid+1, r, 2*pos+2);
58     l = 2*pos+1, r = 2*pos+2;
59     tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
60         tree[r].pref});
61     tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
62     tree[pos].tot = tree[l].tot + tree[r].tot;
63     tree[pos].best = max({tree[l].best, tree[r].best, tree[l].suf +
64         tree[r].pref});
65 }

```

8.12. 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 }

```

9. Strings

9.1. Kmp

```

1 vector<int> pi(const string &s) {
2     int n = s.size();
3     vector<int> pi(n);
4
5     int l = 0, r = 1;
6     while (r < n) {
7         if (s[l] == s[r]) {
8             l++;
9             pi[r] = l;
10            r++;
11        } else {
12            if (l == 0) {
13                pi[r] = 0;
14                r++;
15            } else
16                l = pi[l - 1];
17        }

```

```

18     }
19     return pi;
20 }
21
22 // returns the index of first occurrence of a pat in a txt
23 int kmp(const string &txt, const string &pat) {
24     int n = txt.size(), m = pat.size();
25
26     int t = 0, p = 0;
27     vector<int> pi_pat = pi(pat);
28     // vector<int> occ;
29
30     while (t < n) {
31
32         if (txt[t] == pat[p]) {
33             t++, p++;
34             if (p == m) {
35                 return t - m;
36                 // if you want to continue searching
37                 // occ.pb(t - m);
38                 // p = pi_pat[p - 1];
39             }
40         } else {
41             if (p == 0) {
42                 t++;
43             } else {
44                 p = pi_pat[p - 1];
45             }
46         }
47     }
48     return -1;
49     // return occ;
50 }

```

9.2. Suffix Array

```

1 // Created by Ubiratan Neto
2
3 /*
4
5 0 banana          Suf,    lcp
6 1 anana           Sort the Suffixes    3 ana    1
7 2 nana            ----->            1 anana   3
8 3 ana             alphabetically       0 banana  0
9 4 na              4 na                 0
10 5 a               2 nana                2
11
12 lcp = number of characters equal prefi
13 */
14
15 struct SuffixArray {
16
17     vector<int> rnk,tmp,sa, sa_aux, lcp, pot, sp[22];
18
19     int block, n;
20
21     string s;
22
23     SuffixArray() {}
24
25     SuffixArray(string t){
26         s = t;
27         n = t.size();
28         rnk.resize(n+1);

```

```

29         for(int i=0; i<22; i++) sp[i].resize(n+1);
30         pot.resize(n+1);
31         tmp.resize(max((int)257, n+1));
32         // sa stores index of first char of suffix
33         sa.resize(n+1);
34         sa_aux.resize(n+1);
35         // lcp stores value between the string and next string
36         lcp.resize(n+1);
37         block = 0;
38     }
39
40     bool suffixcmp(int i, int j){
41         if(rnk[i] != rnk[j]) return rnk[i] < rnk[j];
42         i+=block, j+=block;
43         i%=n;
44         j%=n;
45         return rnk[i] < rnk[j];
46     }
47
48     void suffixSort(int MAX_VAL){
49         for(int i=0; i<=MAX_VAL; i++) tmp[i] = 0;
50         for(int i=0; i<n; i++) tmp[rnk[i]]++;
51         for(int i=1; i<=MAX_VAL; i++) tmp[i] += tmp[i-1];
52         for(int i = n-1; i>=0; i--){
53             int aux = sa[i]-block;
54             aux%=n;
55             if(aux < 0) aux+=n;
56             sa_aux[--tmp[rnk[aux]]] = aux;
57         }
58         for(int i=0; i<n; i++) sa[i] = sa_aux[i];
59         tmp[0] = 0;
60         for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
61         for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
62     }
63
64     void calculate(){
65         s+='\\0';
66         n++;
67         for(int i=0; i<n; i++){
68             sa[i] = i;
69             rnk[i] = s[i];
70             tmp[i] = 0;
71         }
72         suffixSort(256);
73         block = 1;
74         while(tmp[n-1] != n-1){
75             suffixSort(tmp[n-1]);
76             block*=2;
77         }
78         for(int i=0; i<n-1; i++) sa[i] = sa[i+1];
79         n--;
80         tmp[0] = 0;
81         for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
82         for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
83         s.pop_back();
84         sa.pop_back();
85     }
86
87     void calculate_lcp(){
88         int last = 0;
89         for(int i=0; i<n; i++){
90             if(rnk[i] == n-1) continue;
91             int x = rnk[i];
92             lcp[x] = max((int)0,last-1);

```

```

93     while(sa[x] + lcp[x] < n && sa[x+1] + lcp[x] < n && s[sa[x]+lcp[x]] ==
94           s[sa[x+1]+lcp[x]]){
95         lcp[x]++;
96     }
97     last = lcp[x];
98 }
99
100 void build_lcp_table() {
101     int k = 0;
102     for(int j = 0; (1<<j) <= 2*n; j++) {
103         for(; k <= n && k < (1<<j); k++) {
104             pot[k] = j-1;
105         }
106     }
107     for(int i=0; i<n; i++){
108         sp[0][i] = lcp[i];
109     }
110     for(int i = 1; (1<<i) <= n; i++) {
111         for(int j = 0; j+(1<<i) <= n; j++) {
112             sp[i][j] = min(sp[i-1][j], sp[i-1][j+(1<<(i-1))]);
113         }
114     }
115 }
116
117 // to find lcp of two different suffixes starting at x and y
118 int query_lcp(int x, int y){
119     if(x == y) return n - x;
120     if(rnk[x] > rnk[y]) swap(x,y);
121     int l = rnk[x], r = rnk[y]-1;
122     return min(sp[pot[r-l+1]][l], sp[pot[r-l+1]][r-(1LL<<pot[r-l+1])+1]);
123 }
124
125 // needs calculate and calculate lcp first
126 int number_of_substrings(){
127     int ans = n - sa[0];
128     for(int i=0; i<n-1; i++){
129         int length = n - sa[i+1];
130         ans += length - lcp[i];
131     }
132     return ans;
133 }
134
135 // needs calculate and calculate lcp first
136 int lcs(string &x, string &y) {
137
138     string s = x + "#" + y;
139
140     int n = (int)s.size() - 1;
141     int ans = 0;
142     for(int i = 0; i < n - 1; i++) {
143         int ida = sa[i];
144         int idb = sa[i+1];
145         if(ida < x.size() && idb > x.size() || ida > x.size() && idb <
146           x.size()) {
147             ans = max(ans, lcp[i]);
148         }
149     }
150 }
151
152 };

```

9.3. 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

```

9.4. Trie - Maximum Xor Two Elements

```

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

```

9.5. Z-Function

```

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

```

9.6. 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 vector<vector<pair<int, int>>> aho_find_occurrences(const string &text) {
79     vector<vector<pair<int, int>>> ans(arr.size());
80     Trie::Node *cur = trie.root();
81
82     for (int i = 0; i < text.size(); ++i) {
83         cur = find_fail_node(cur, text[i]);
84         for (Out_Node *node = out_node[cur->id]; node != nullptr;
85              node = node->next)
86             for (const int idx : node->str_idx)
87                 ans[idx].emplace_back(i - (int)arr[idx].size() + 1, i);
88     }
89     return ans;
90 }
91
92 public:
93     /// Constructor that builds the trie and the failures.
94     ///
95     /// Time Complexity: O(number of characters in arr)
96     Aho(const vector<string> &arr) : arr(arr) { this->build_trie(); }
97
98     /// Searches in text for all occurrences of all strings in array arr.
99     ///
100     /// Time Complexity: O(text.size() + number of characters in arr)
101     vector<vector<pair<int, int>>> find_occurrences(const string &text) {
102         return this->aho_find_occurrences(text);
103     }
104 }
105

```

9.7. Hashing

```

1 // OBS: CHOOSE THE OFFSET AND THE PRIMES BELOW!!
2 class Hash {
3     /// Prime numbers to be used in mod operations
4     /// OBS: if you change m's size, please change the return type of both
5     /// query
6     /// and _query methods.
7     vector<int> m = {1000000007, 1000000009};
8
9     // Case the alphabet goes from 'a' to 'z'.
10    static constexpr int OFFSET = 'a';
11    // Choose primes greater than the size of the alphabet.
12    vector<int> prime = {31, 37};
13
14    // Case the alphabet goes from 'A' to 'z'.
15    // constexpr int OFFSET = 'A';
16    // Choose primes greater than the size of the alphabet.
17    // vector<int> prime = {61, 67};
18
19    vector<vector<int>> hash_table;
20    vector<vector<int>> pot;
21    // size of the string
22    int n;

```

```

23 private:
24     int mod(int n, int m) {
25         n %= m;
26         if (n < 0)
27             n += m;
28         return n;
29     }
30
31     /// Time Complexity: O(1)
32     pair<int, int> _query(const int l, const int r) {
33         vector<int> ans(m.size());
34
35         if (l == 0) {
36             for (int i = 0; i < m.size(); i++)
37                 ans[i] = hash_table[i][r];
38         } else {
39             for (int i = 0; i < m.size(); i++)
40                 ans[i] =
41                     mod((hash_table[i][r] - hash_table[i][l - 1] * pot[i][r - 1 +
42                         1])), m[i]);
43         }
44
45         return {ans.front(), ans.back()};
46     }
47
48     /// Builds the hash table and the pot table.
49     ///
50     /// Time Complexity: O(n)
51     void build(string &s) {
52         pot.resize(m.size(), vector<int>(this->n));
53         hash_table.resize(m.size(), vector<int>(this->n));
54         // Remapping the string
55         for (char &c : s)
56             c -= OFFSET;
57
58         for (int i = 0; i < m.size(); i++) {
59             hash_table[i][0] = s[0];
60             pot[i][0] = 1;
61             for (int j = 1; j < this->n; j++) {
62                 hash_table[i][j] = (s[j] + hash_table[i][j - 1] * prime[i]) % m[i];
63                 pot[i][j] = (pot[i][j - 1] * prime[i]) % m[i];
64             }
65         }
66     }
67
68 public:
69     /// Constructor that is responsible for building the hash table and pot
70     /// table.
71     ///
72     /// Time Complexity: O(n)
73     Hash(string s) {
74         assert(m.size() == prime.size());
75         this->n = s.size();
76
77         build(s);
78     }
79
80     /// Returns the hash from l to r.
81     ///
82     /// Time Complexity: O(1) -> Actually O(number_of_primes)
83     pair<int, int> query(const int l, const int r) {
84         assert(0 <= l), assert(l <= r), assert(r < this->n);
85         return _query(l, r);
86     }

```

```

86 };

```

9.8. Lcs K Strings

```

1 // Make the change below in SuffixArray code.
2 int MaximumNumberOfStrings;
3
4 void build_suffix_array() {
5     vector<pair<Rank, int>> ranks(this->n + 1);
6     vector<int> arr;
7
8     for (int i = 1, separators = 0; i <= n; i++)
9         if (this->s[i] > 0) {
10             ranks[i] = pair<Rank, int>(Rank((int)this->s[i] +
11                 MaximumNumberOfStrings, 0), i);
12             this->s[i] += MaximumNumberOfStrings;
13         } else {
14             ranks[i] = pair<Rank, int>(Rank(separators, 0), i);
15             this->s[i] = separators;
16             separators++;
17         }
18
19     RadixSort::sort_pairs(ranks, 256 + MaximumNumberOfStrings);
20     ...
21 }
22
23 /// Program to find the LCS between k different strings.
24 ///
25 /// Time Complexity: O(n*log(n))
26 /// Space Complexity: O(n*log(n))
27 int main() {
28     int n;
29
30     cin >> n;
31
32     MaximumNumberOfStrings = n;
33
34     vector<string> arr(n);
35
36     int sum = 0;
37     for (string &x: arr) {
38         cin >> x;
39         sum += x.size() + 1;
40     }
41
42     string concat;
43     vector<int> ind(sum + 1);
44     int cnt = 0;
45     for (string &x: arr) {
46         if (concat.size())
47             concat += (char)cnt;
48         concat += x;
49     }
50
51     cnt = 0;
52     for (int i = 0; i < concat.size(); i++) {
53         ind[i + 1] = cnt;
54         if (concat[i] < MaximumNumberOfStrings)
55             cnt++;
56     }
57
58     Suffix_Array say(concat);
59     vector<int> sa = say.get_suffix_array();
60     Sparse_Table spt(say.get_lcp());

```

```

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

```

9.9. Lexicographically Smallest Rotation

```

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

```

9.10. 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>
    &lps) {
18     for(int i = 0; i < lps.size(); i++) {
19         if(lps[i] == max_len) {
20             string ans;
21             int cnt = max_len / 2;
22             int io = i - 1;
23             while(cnt) {
24                 if(str[io] != '#') {
25                     ans += str[io];
26                     cnt--;
27                 }
28                 io--;
29             }
30             reverse(ans.begin(), ans.end());
31             if(str[i] != '#')
32                 ans += str[i];
33             cnt = max_len / 2;
34             io = i + 1;
35             while(cnt) {
36                 if(str[io] != '#') {
37                     ans += str[io];
38                     cnt--;
39                 }
40                 io++;
41             }
42             return ans;
43         }
44     }
45 }
46
47 /// Returns a pair containing the size of the longest palindrome and the
48 /// first occurrence of it.
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
55     //char as the center
56     vector<int> lps(len);
57     int c = 0; //stores the center of the longest palindromic substring until
58     //now
59     int r = 0; //stores the right boundary of the longest palindromic
60     //substring until now
61     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
64     //current longest palindrome at center c
65     //if it is, then take r - i as lps[i]
66     //else take lps[mirror] as lps[i]
67     if(i < r)
68         lps[i] = min(r - i, lps[mirror]);
69
70     //expand at i
71     int a = i + (1 + lps[i]);
72     int b = i - (1 + lps[i]);
73     while(a < len && b >= 0 && str[a] == str[b]) {
74         lps[i]++;
75         a++;
76         b--;
77     }
78
79     //check if the expanded palindrome at i is expanding beyond the right
80     //boundary of current longest palindrome at center c
81     //if it is, the new center is i
82     if(i + lps[i] > r) {
83         c = i;
84         r = i + lps[i];
85
86         if(lps[i] > max_len) //update max_len
87             max_len = lps[i];
88     }
89
90     return make_pair(max_len, get_best(max_len, str, lps));
91 }

```

9.11. Suffix Array

```

1 namespace RadixSort {
2     /// Sorts the array arr stably in ascending order.
3     ///
4     /// Time Complexity: O(n + max_element)
5     /// Space Complexity: O(n + max_element)
6     template <typename T>
7     void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),
8             int begin = 0) {
9         const int n = arr.size();
10        vector<T> new_order(n);
11        vector<int> count(max_element + 1, 0);
12
13        for (int i = begin; i < n; i++)
14            count[get_key(arr[i])]++;
15
16        for (int i = 1; i <= max_element; i++)
17            count[i] += count[i - 1];
18
19        for (int i = n - 1; i >= begin; i--) {
20            new_order[count[get_key(arr[i])] - (begin == 0)] = arr[i];
21            count[get_key(arr[i])]--;
22        }
23
24        arr.swap(new_order);
25    }
26
27    /// Sorts an array by their pair of ranks stably in ascending order.

```

```

28 template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
29     // Sort by the second rank
30     RadixSort::sort<T>(
31         arr, rank_size, [](T &item) { return item.first.second; }, 0ll);
32
33     // Sort by the first rank
34     RadixSort::sort<T>(
35         arr, rank_size, [](T &item) { return item.first.first; }, 0ll);
36 }
37 } // namespace RadixSort
38
39 /// It is indexed by 0.
40 /// Let the given string be "banana".
41 ///
42 /// 0 banana          5 a
43 /// 1 anana          Sort the Suffixes  3 ana
44 /// 2 nana          ----->          1 anana
45 /// 3 ana            alphabetically    0 banana
46 /// 4 na              4 na
47 /// 5 a              2 nana
48 /// So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
49 ///
50 /// LCP
51 ///
52 /// 1 a
53 /// 3 ana
54 /// 0 anana
55 /// 0 banana
56 /// 2 na
57 /// 0 nana (The last position will always be zero)
58 ///
59 /// So the LCP for "banana" is {1, 3, 0, 0, 2, 0}
60 ///
61 class Suffix_Array {
62 private:
63     string s;
64     int n;
65
66     typedef pair<int, int> Rank;
67
68 public:
69     Suffix_Array(string &s) {
70         this->n = s.size();
71         this->s = s;
72         // little optimization, remove the line above
73         // this->s.swap(s);
74
75         this->sa = build_suffix_array();
76         this->lcp = build_lcp();
77     }
78
79 private:
80     /// The vector containing the ranks will be present at ret
81     void build_ranks(const vector<pair<Rank, int>> &ranks, vector<int> &ret) {
82         ret[ranks[0].second] = 1;
83         for (int i = 1; i < n; i++) {
84             // If their rank are equal, than its position should be the same.
85             if (ranks[i - 1].first == ranks[i].first)
86                 ret[ranks[i].second] = ret[ranks[i - 1].second];
87             else
88                 ret[ranks[i].second] = ret[ranks[i - 1].second] + 1;
89         }
90     }
91
92     /// Builds the Suffix Array for the string s.

```

```

93  ///
94  /// Time Complexity: O(n*log(n))
95  /// Space Complexity: O(n)
96  vector<int> build_suffix_array() {
97      // This tuple below represents the rank and the index associated with it.
98      vector<pair<Rank, int>> ranks(this->n);
99      vector<int> arr(this->n);
100
101      for (int i = 0; i < n; i++)
102          ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);
103
104      RadixSort::sort_pairs(ranks, 256);
105      build_ranks(ranks, arr);
106
107      {
108          int jump = 1;
109          int max_rank = arr[ranks.back().second];
110          // It will be compared intervals a pair of intervals (i, jump-1), (i +
111          // jump, i + 2*jump - 1). The variable jump is always a power of 2.
112          while (max_rank != this->n) {
113              for (int i = 0; i < this->n; i++) {
114                  ranks[i].first.first = arr[i];
115                  ranks[i].first.second = (i + jump < this->n ? arr[i + jump] : 0);
116                  ranks[i].second = i;
117              }
118
119              RadixSort::sort_pairs(ranks, n);
120              build_ranks(ranks, arr);
121
122              max_rank = arr[ranks.back().second];
123              jump *= 2;
124          }
125      }
126
127      vector<int> sa(this->n);
128      for (int i = 0; i < this->n; i++)
129          sa[arr[i] - 1] = i;
130      return sa;
131  }
132
133  /// Builds the lcp (Longest Common Prefix) array for the string s.
134  /// A value lcp[i] indicates length of the longest common prefix of the
135  /// suffixes indexed by i and i + 1. Implementation of the Kasai's
136  /// Algorithm.
137  ///
138  /// Time Complexity: O(n)
139  /// Space Complexity: O(n)
140  vector<int> build_lcp() {
141      lcp.resize(n, 0);
142      vector<int> inverse_suffix(this->n);
143
144      for (int i = 0; i < this->n; i++)
145          inverse_suffix[sa[i]] = i;
146
147      int k = 0;
148
149      for (int i = 0; i < this->n; i++) {
150          if (inverse_suffix[i] == this->n - 1) {
151              k = 0;
152              continue;
153          }
154
155          int j = sa[inverse_suffix[i] + 1];
156          while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])

```

```

157         k++;
158         lcp[inverse_suffix[i]] = k;
159     }
160
161     if (k > 0)
162         k--;
163 }
164
165 return lcp;
166 }
167
168 public:
169     vector<int> sa;
170     vector<int> lcp;
171
172     /// LCS of two strings A and B.
173     ///
174     /// The string s must be initialized in the constructor as the string (A +
175     /// '$' + B).
176     ///
177     /// The string A starts at index 1 and ends at index (separator - 1).
178     /// The string B starts at index (separator + 1) and ends at the end of the
179     /// string.
180     ///
181     /// Time Complexity: O(n)
182     /// Space Complexity: O(1)
183     int lcs(int separator) {
184         assert(!isalpha(this->s[separator] && !isdigit(this->s[separator]]));
185
186         int ans = 0;
187
188         for (int i = 0; i + 1 < this->sa.size(); i++) {
189             int left = this->sa[i];
190             int right = this->sa[i + 1];
191
192             if ((left < separator && right > separator) ||
193                 (left > separator && right < separator))
194                 ans = max(ans, lcp[i]);
195         }
196
197         return ans;
198     }
199 };

```

9.12. Suffix Array Pessoa

```

1  // OBS: Suffix Array build code imported from:
2  //
3  // https://github.com/gabrielpessoa1/Biblioteca-Maratona/blob/master/code/String/SuffixArray.cpp
4  // Because it's faster.
5  /// It is indexed by 0.
6  /// Let the given string be "banana".
7  ///
8  /// 0 banana          5 a
9  /// 1 anana          Sort the Suffixes      3 ana
10 /// 2 nana            ----->              1 anana
11 /// 3 ana            alphabetically         0 banana
12 /// 4 na              4 na
13 /// 5 a              2 nana
14 /// So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
15 ///
16 /// LCP

```



```

17 ///
18 /// 1 a
19 /// 3 ana
20 /// 0 anana
21 /// 0 banana
22 /// 2 na
23 /// 0 nana (The last position will always be zero)
24 ///
25 /// So the LCP for "banana" is {1, 3, 0, 0, 2, 0}
26 ///
27 class Suffix_Array {
28 private:
29     string s;
30     int n;
31
32     typedef pair<int, int> Rank;
33
34 public:
35     Suffix_Array(string &s) {
36         this->n = s.size();
37         this->s = s;
38         // little optimization, remove the line above
39         // this->s.swap(s);
40
41         this->sa = build_suffix_array();
42         this->lcp = build_lcp();
43     }
44
45 private:
46     /// Builds the Suffix Array for the string s.
47     ///
48     /// Time Complexity: O(n*log(n))
49     /// Space Complexity: O(n)
50     vector<int> build_suffix_array() {
51         int n = this->s.size(), c = 0;
52         vector<int> temp(n), posBucket(n), bucket(n), bpos(n), out(n);
53         for (int i = 0; i < n; i++)
54             out[i] = i;
55         sort(out.begin(), out.end(),
56             [&](int a, int b) { return this->s[a] < this->s[b]; });
57         for (int i = 0; i < n; i++) {
58             bucket[i] = c;
59             if (i + 1 == n || this->s[out[i]] != this->s[out[i + 1]])
60                 c++;
61         }
62         for (int h = 1; h < n && c < n; h <= 1) {
63             for (int i = 0; i < n; i++)
64                 posBucket[out[i]] = bucket[i];
65             for (int i = n - 1; i >= 0; i--)
66                 bpos[bucket[i]] = i;
67             for (int i = 0; i < n; i++) {
68                 if (out[i] >= n - h)
69                     temp[bpos[bucket[i]]++] = out[i];
70             }
71             for (int i = 0; i < n; i++) {
72                 if (out[i] >= h)
73                     temp[bpos[posBucket[out[i] - h]]++] = out[i] - h;
74             }
75             c = 0;
76             for (int i = 0; i + 1 < n; i++) {
77                 int a = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h) ||
78                     (posBucket[temp[i + 1] + h] != posBucket[temp[i] + h]);
79                 bucket[i] = c;
80                 c += a;
81             }

```

```

82         bucket[n - 1] = c++;
83         temp.swap(out);
84     }
85     return out;
86 }
87
88 /// Builds the lcp (Longest Common Prefix) array for the string s.
89 /// A value lcp[i] indicates length of the longest common prefix of the
90 /// suffixes indexed by i and i + 1. Implementation of the Kasai's
    Algorithm.
91 ///
92 /// Time Complexity: O(n)
93 /// Space Complexity: O(n)
94 vector<int> build_lcp() {
95     lcp.resize(n, 0);
96     vector<int> inverse_suffix(this->n);
97
98     for (int i = 0; i < this->n; i++)
99         inverse_suffix[sa[i]] = i;
100
101     int k = 0;
102
103     for (int i = 0; i < this->n; i++) {
104         if (inverse_suffix[i] == this->n - 1) {
105             k = 0;
106             continue;
107         }
108
109         int j = sa[inverse_suffix[i] + 1];
110
111         while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
112             k++;
113
114         lcp[inverse_suffix[i]] = k;
115
116         if (k > 0)
117             k--;
118     }
119
120     return lcp;
121 }
122
123 public:
124     vector<int> sa;
125     vector<int> lcp;
126
127     /// LCS of two strings A and B.
128     ///
129     /// The string s must be initialized in the constructor as the string (A +
    ',$'
130     /// + B).
131     ///
132     /// The string A starts at index 1 and ends at index (separator - 1).
133     /// The string B starts at index (separator + 1) and ends at the end of the
    /// string.
134     ///
135     /// Time Complexity: O(n)
136     /// Space Complexity: O(1)
137     int lcs(int separator) {
138         assert(!isalpha(this->s[separator] && !isdigit(this->s[separator]]));
139
140         int ans = 0;
141
142         for (int i = 0; i + 1 < this->sa.size(); i++) {
143             int left = this->sa[i];

```

```

145     int right = this->sa[i + 1];
146
147     if ((left < separator && right > separator) ||
148         (left > separator && right < separator))
149         ans = max(ans, lcp[i]);
150 }
151
152 return ans;
153 }
154 };

```

9.13. Suffix Array With Additional Memory

```

1 namespace RadixSort {
2     /// Sorts the array arr stably in ascending order.
3     ///
4     /// Time Complexity: O(n + max_element)
5     /// Space Complexity: O(n + max_element)
6     template <typename T>
7     void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),
8               int begin = 0) {
9         const int n = arr.size();
10        vector<T> new_order(n);
11        vector<int> count(max_element + 1, 0);
12
13        for (int i = begin; i < n; i++)
14            count[get_key(arr[i])]++;
15
16        for (int i = 1; i <= max_element; i++)
17            count[i] += count[i - 1];
18
19        for (int i = n - 1; i >= begin; i--) {
20            new_order[count[get_key(arr[i])]] = arr[i];
21            count[get_key(arr[i])]--;
22        }
23
24        arr = new_order;
25    }
26
27    /// Sorts an array by their pair of ranks stably in ascending order.
28    template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
29        /// Sort by the second rank
30        RadixSort::sort<T>(
31            arr, rank_size, [](T &item) { return item.first.second; }, 1ll);
32
33        /// Sort by the first rank
34        RadixSort::sort<T>(
35            arr, rank_size, [](T &item) { return item.first.first; }, 1ll);
36    }
37 } // namespace RadixSort
38
39 /// It is indexed by 1.
40 class Suffix_Array {
41 private:
42     string s;
43     int n;
44
45     typedef pair<int, int> Rank;
46     vector<int> suffix_array;
47     vector<int> lcp;
48
49     vector<vector<int>>> rank_table;
50     vector<int> log_array;
51

```

```

52 public:
53     Suffix_Array(const string &s) {
54         this->n = s.size();
55         this->s = "#" + s;
56
57         build_log_array();
58         build_suffix_array();
59         lcp = build_lcp();
60     }
61
62 private:
63     vector<int> build_ranks(const vector<pair<Rank, int>> &ranks) {
64         vector<int> arr(this->n + 1);
65
66         arr[ranks[1].second] = 1;
67         for (int i = 2; i <= n; i++) {
68             /// If their rank are equal, than its position should be the same.
69             if (ranks[i - 1].first == ranks[i].first)
70                 arr[ranks[i].second] = arr[ranks[i - 1].second];
71             else
72                 arr[ranks[i].second] = arr[ranks[i - 1].second] + 1;
73         }
74
75         return arr;
76     }
77
78     /// Builds the Suffix Array for the string s.
79     ///
80     /// Time Complexity: O(n*log(n))
81     /// Space Complexity: O(n*log(n))
82     void build_suffix_array() {
83         /// This tuple below represents the rank and the index associated with it.
84         vector<pair<Rank, int>> ranks(this->n + 1);
85         vector<int> arr;
86
87         int rank_table_size = 0;
88         this->rank_table.resize(log_array[this->n] + 2);
89
90         for (int i = 1; i <= this->n; i++)
91             ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);
92
93         /// Inserting only the ranks in the table.
94         transform(ranks.begin(), ranks.end(),
95                 back_inserter(rank_table[rank_table_size++]),
96                 [](pair<Rank, int> &pair) { return pair.first.first; });
97
98         RadixSort::sort_pairs(ranks, 256);
99         arr = build_ranks(ranks);
100
101     {
102         int jump = 1;
103         int max_rank = arr[ranks.back().second];
104
105         /// It will be compared intervals a pair of intervals (i, jump-1), (i +
106         /// jump, i + 2*jump - 1). The variable jump is always a power of 2.
107         while (jump < n) {
108             for (int i = 1; i <= this->n; i++) {
109                 ranks[i].first.first = arr[i];
110                 ranks[i].first.second = (i + jump <= this->n ? arr[i + jump] : 0);
111                 ranks[i].second = i;
112             }
113
114             /// Inserting only the ranks in the table.
115             transform(ranks.begin(), ranks.end(),
116                     back_inserter(rank_table[rank_table_size++]),

```

```

117         [(pair<Rank, int> &pair) { return pair.first.first; }]);
118
119     RadixSort::sort_pairs(ranks, n);
120
121     arr = build_ranks(ranks);
122
123     max_rank = arr[ranks.back().second];
124     jump *= 2;
125 }
126
127 for (int i = 1; i <= n; i++) {
128     ranks[i].first.first = arr[i];
129     ranks[i].first.second = (i + jump <= this->n ? arr[i + jump] : 0);
130     ranks[i].second = i;
131 }
132
133 // Inserting only the ranks in the table.
134 transform(ranks.begin(), ranks.end(),
135           back_inserter(rank_table[rank_table_size++]),
136           [(pair<Rank, int> &pair) { return pair.first.first; }]);
137 }
138
139 this->suffix_array.resize(this->n + 1);
140 for (int i = 1; i <= this->n; i++)
141     this->suffix_array[arr[i]] = i;
142 }
143
144 /// Builds the lcp (Longest Common Prefix) array for the string s.
145 /// A value lcp[i] indicates length of the longest common prefix of the
146 /// suffixes indexed by i and i + 1. Implementation of the Kasai's
147 /// Algorithm.
148 ///
149 /// Time Complexity: O(n)
150 /// Space Complexity: O(n)
151 vector<int> build_lcp() {
152     vector<int> lcp(this->n + 1, 0);
153     vector<int> inverse_suffix(this->n + 1, 0);
154
155     for (int i = 1; i <= n; i++)
156         inverse_suffix[suffix_array[i]] = i;
157
158     int k = 0;
159
160     for (int i = 1; i <= n; i++) {
161         if (inverse_suffix[i] == n) {
162             k = 0;
163             continue;
164         }
165
166         int j = suffix_array[inverse_suffix[i] + 1];
167
168         while (i + k <= this->n && j + k <= this->n && s[i + k] == s[j + k])
169             k++;
170
171         lcp[inverse_suffix[i]] = k;
172
173         if (k > 0)
174             k--;
175     }
176
177     return lcp;
178 }
179
180 void build_log_array() {
181     log_array.resize(this->n + 1, 0);

```

```

181
182     for (int i = 2; i <= this->n; i++)
183         log_array[i] = log_array[i / 2] + 1;
184 }
185
186 public:
187     const vector<int> &get_suffix_array() { return suffix_array; }
188
189     const vector<int> &get_lcp() { return lcp; }
190
191     /// LCS of two strings A and B.
192     ///
193     /// The string s must be initialized in the constructor as the string (A +
194     /// '$' + B).
195     ///
196     /// The string A starts at index 1 and ends at index (separator - 1).
197     /// The string B starts at index (separator + 1) and ends at the end of the
198     /// string.
199     ///
200     /// Time Complexity: O(n)
201     /// Space Complexity: O(1)
202     int lcs(int separator) {
203         separator++;
204         assert(!isalpha(this->s[separator] && !isdigit(this->s[separator])));
205
206         int ans = 0;
207
208         for (int i = 1; i < this->n - 1; i++) {
209             int left = this->suffix_array[i];
210             int right = this->suffix_array[i + 1];
211
212             if ((left < separator && right > separator) ||
213                 (left > separator && right < separator))
214                 ans = max(ans, lcp[i]);
215         }
216
217         return ans;
218     }
219
220     /// Compares two substrings beginning at indexes i and j of a fixed length.
221     ///
222     /// OBS: Necessary build rank_table (uncomment build_suffix_array) and
223     /// build
224     /// log_array.
225     ///
226     /// Time Complexity: O(1)
227     /// Space Complexity: O(1)
228     int compare(const int i, const int j, const int length) {
229         assert(1 <= i && i <= this->n && 1 <= j && j <= this->n);
230         assert(!this->log_array.empty() && !this->rank_table.empty());
231         assert(i + length - 1 <= this->n && j + length - 1 <= this->n);
232
233         // Greatest k such that 2^k <= 1
234         const int k = this->log_array[length];
235
236         const int jump = length - (1 << k);
237
238         const pair<int, int> iRank = {
239             this->rank_table[k][i],
240             (i + jump <= this->n ? this->rank_table[k][i + jump] : -1)};
241         const pair<int, int> jRank = {
242             this->rank_table[k][j],
243             (j + jump <= this->n ? this->rank_table[k][j + jump] : -1)};
244     }

```

```

244     return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
245 }
246 };

```

9.14. Trie

```

1  class Trie {
2  private:
3      static const int INT_LEN = 31;
4      // static const int INT_LEN = 63;
5
6  public:
7      struct Node {
8          map<char, Node *> next;
9          int id;
10         // cnt counts the number of words which pass in that node
11         int cnt = 0;
12         // word counts the number of words ending at that node
13         int word_cnt = 0;
14
15         Node(const int x) : id(x) {}
16     };
17
18 private:
19     int trie_size = 0;
20     // contains the next id to be used in a node
21     int node_cnt = 0;
22     Node *trie_root = this->make_node();
23
24 private:
25     Node *make_node() { return new Node(node_cnt++); }
26
27     int trie_insert(const string &s) {
28         Node *aux = this->root();
29         for (const char c : s) {
30             if (!aux->next.count(c))
31                 aux->next[c] = this->make_node();
32             aux = aux->next[c];
33             ++aux->cnt;
34         }
35         ++aux->word_cnt;
36         ++this->trie_size;
37         return aux->id;
38     }
39
40     void trie_erase(const string &s) {
41         Node *aux = this->root();
42         for (const char c : s) {
43             Node *last = aux;
44             aux = aux->next[c];
45             --aux->cnt;
46             if (aux->cnt == 0) {
47                 last->next.erase(c);
48                 aux = nullptr;
49                 break;
50             }
51         }
52         if (aux != nullptr)
53             --aux->word_cnt;
54         --this->trie_size;
55     }
56
57     int trie_count(const string &s) {
58         Node *aux = this->root();

```

```

59     for (const char c : s) {
60         if (aux->next.count(c))
61             aux = aux->next[c];
62         else
63             return 0;
64     }
65     return aux->word_cnt;
66 }
67
68 int trie_query_xor_max(const string &s) {
69     Node *aux = this->root();
70     int ans = 0;
71     for (const char c : s) {
72         const char inv = (c == '0' ? '1' : '0');
73         if (aux->next.count(inv)) {
74             ans = (ans << 1ll) | (inv - '0');
75             aux = aux->next[inv];
76         } else {
77             ans = (ans << 1ll) | (c - '0');
78             aux = aux->next[c];
79         }
80     }
81     return ans;
82 }
83
84 public:
85     Trie() {}
86
87     Node *root() { return this->trie_root; }
88
89     int size() { return this->trie_size; }
90
91     /// Returns the number of nodes present in the trie.
92     int node_count() { return this->node_cnt; }
93
94     /// Inserts s in the trie.
95     ///
96     /// Returns the id of the last character of the string in the trie.
97     ///
98     /// Time Complexity: O(s.size())
99     int insert(const string &s) { return this->trie_insert(s); }
100
101     /// Inserts the binary representation of x in the trie.
102     ///
103     /// Time Complexity: O(log x)
104     int insert(const int x) {
105         assert(x >= 0);
106         // converting x to binary representation
107         return this->trie_insert(bitset<INT_LEN>(x).to_string());
108     }
109
110     /// Removes the string s from the trie.
111     ///
112     /// Time Complexity: O(s.size())
113     void erase(const string &s) { this->trie_erase(s); }
114
115     /// Removes the binary representation of x from the trie.
116     ///
117     /// Time Complexity: O(log x)
118     void erase(const int x) {
119         assert(x >= 0);
120         // converting x to binary representation
121         this->trie_erase(bitset<INT_LEN>(x).to_string());
122     }
123

```

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