

# C++ Competitive Programming Library

\*\*\*DO NOT DISCLOSE OR DISTRIBUTE\*\*\*

bfs.07 - Bernardo Flores Salmeron

|          |   |           |          |  |           |
|----------|---|-----------|----------|--|-----------|
| <b>1</b> | <b>Template</b>   | <b>3</b>  |          |  |           |
| <b>2</b> | <b>Data Structures</b>  | <b>3</b>  |          |  |           |
| 2.1      | Bit2D . . . . .   | 3         | 4.3      | Condicao De Existencia De Um Triangulo . . . . . | 15        |
| 2.2      | Merge Sort Tree (K-Esimo Maior Elemento Num Intervalo, Valores Maiores Que K Num Intervalo, . . . . . | 3         | 4.4      | Convex Hull . . . . .                            | 15        |
| 2.3      | Mos Algorithm . . . . .   | 4         | 4.5      | Cross Product . . . . .                          | 16        |
| 2.4      | Sqrt Decomposition . . . . .  | 4         | 4.6      | Distance Point Segment . . . . .                 | 16        |
| 2.5      | Bit . . . . .   | 4         | 4.7      | Line-Line Intersection . . . . .                 | 16        |
| 2.6      | Bit (Range Update) . . . . .  | 5         | 4.8      | Line-Point Distance . . . . .                    | 16        |
| 2.7      | Counting Inversions (Minimum Number Of Adjacent Swaps To Sort Array) . . . . .                        | 6         | 4.9      | Point Inside Convex Polygon - Log(N) . . . . .   | 16        |
| 2.8      | Ordered Set . . . . .   | 6         | 4.10     | Point Inside Polygon . . . . .                   | 17        |
| 2.9      | Persistent Segment Tree . . . . .   | 6         | 4.11     | Points Inside And In Boundary Polygon . . . . .  | 18        |
| 2.10     | Segment Tree . . . . .  | 7         | 4.12     | Polygon Area (3D) . . . . .                      | 19        |
| 2.11     | Segment Tree 2D . . . . .   | 9         | 4.13     | Polygon Area . . . . .                           | 19        |
| 2.12     | Segment Tree Polynomial . . . . .   | 10        | 4.14     | Segment-Segment Intersection . . . . .           | 19        |
| 2.13     | Sparse Table . . . . .  | 11        | 4.15     | Upper And Lower Hull . . . . .                   | 20        |
| <b>3</b> | <b>Dp</b>   | <b>11</b> | 4.16     | Circle Circle Intersection . . . . .             | 20        |
| 3.1      | Achar Maior Palindromo . . . . .  | 11        | 4.17     | Circle Circle Intersection . . . . .             | 20        |
| 3.2      | Digit Dp . . . . .  | 11        | 4.18     | Struct Point And Line . . . . .                  | 21        |
| 3.3      | Longest Common Subsequence . . . . .  | 12        | <b>5</b> | <b>Graphs</b>                                    | <b>21</b> |
| 3.4      | Longest Common Substring . . . . .  | 12        | 5.1      | Checa Grafo Bipartido . . . . .                  | 21        |
| 3.5      | Longest Increasing Subsequence 2D (Not Sorted) . . . . .  | 12        | 5.2      | Ciclo Grafo . . . . .                            | 22        |
| 3.6      | Longest Increasing Subsequence 2D (Sorted) . . . . .  | 13        | 5.3      | Diametro Em Arvore . . . . .                     | 22        |
| 3.7      | Longest Increasing Subsequence . . . . .  | 13        | 5.4      | Ford Fulkersson (Maximum Flow) . . . . .         | 22        |
| 3.8      | Subset Sum Com Bitset . . . . .   | 14        | 5.5      | Pontes Num Grafo . . . . .                       | 23        |
| 3.9      | Catalan . . . . .   | 14        | 5.6      | Pontos De Articulacao . . . . .                  | 23        |
| 3.10     | Catalan . . . . .   | 14        | 5.7      | Scc (Kosaraju) . . . . .                         | 24        |
| 3.11     | Coin Change Problem . . . . .   | 14        | 5.8      | All Eulerian Path Or Tour . . . . .              | 24        |
| 3.12     | Knapsack . . . . .  | 14        | 5.9      | Bellman Ford . . . . .                           | 26        |
| <b>4</b> | <b>Geometry</b>   | <b>14</b> | 5.10     | De Bruijn Sequence . . . . .                     | 27        |
| 4.1      | Centro De Massa De Um Poligono . . . . .  | 14        | 5.11     | Dijkstra + Dij Graph . . . . .                   | 27        |
| 4.2      | Closest Pair Of Points . . . . .  | 15        | 5.12     | Dinic (Max Flow) . . . . .                       | 28        |
|          |   |           | 5.13     | Floyd Warshall . . . . .                         | 30        |
|          |   |           | 5.14     | Functional Graph . . . . .                       | 30        |
|          |   |           | 5.15     | Hld . . . . .                                    | 32        |

|          |   |           |          |                                     |           |
|----------|---|-----------|----------|-------------------------------------|-----------|
| 5.16     | Kruskal + Dsu   | 33        | 7.1      | Bell Numbers                        | 40        |
| 5.17     | Lca   | 33        | 7.2      | Binary Exponentiation               | 40        |
| 5.18     | Maximum Independent Set (Set Of Vertices That Arent Directly Connected) | 35        | 7.3      | Chinese Remainder Theorem           | 40        |
| 5.19     | Maximum Path Unweighted Graph   | 35        | 7.4      | Combinatorics                       | 40        |
| 5.20     | Minimum Edge Cover (Set Of Edges That Are Adjacent To All Vertices)     | 36        | 7.5      | Diophantine Equation                | 41        |
| 5.21     | Minimum Path Cover In Dag   | 36        | 7.6      | Divisors                            | 41        |
| 5.22     | Minimum Path Cover In Dag   | 36        | 7.7      | Euler Totient                       | 42        |
| 5.23     | Number Of Different Spanning Trees In A Complete Graph                  | 36        | 7.8      | Extended Euclidean                  | 42        |
| 5.24     | Number Of Ways To Make A Graph Connected                                | 36        | 7.9      | Factorization                       | 42        |
| 5.25     | Pruffer Decode  | 36        | 7.10     | Inclusion Exclusion                 | 42        |
| 5.26     | Pruffer Encode  | 36        | 7.11     | Inclusion Exclusion                 | 42        |
| 5.27     | Pruffer Properties  | 37        | 7.12     | Matrix Exponentiation               | 42        |
| 5.28     | Remove All Bridges From Graph   | 37        | 7.13     | Pollard Rho (Find A Divisor)        | 43        |
| 5.29     | Shortest Cycle In A Graph   | 37        | 7.14     | Primality Check                     | 43        |
| 5.30     | Topological Sort  | 37        | 7.15     | Sieve + Segmented Sieve             | 43        |
| 5.31     | Tree Distance   | 37        | <b>8</b> | <b>Miscellaneous</b>                | <b>44</b> |
| <b>6</b> | <b>Language Stuff</b>   | <b>38</b> | 8.1      | 2-Sat                               | 44        |
| 6.1      | Binary String To Int  | 38        | 8.2      | Infix To Prefix                     | 44        |
| 6.2      | Climits   | 38        | 8.3      | Interval Scheduling                 | 45        |
| 6.3      | Checagem Brute Force Com Solucao  | 38        | 8.4      | Interval Scheduling                 | 45        |
| 6.4      | Checagem De Bits  | 38        | 8.5      | Kadane (Maior Soma Num Vetor)       | 45        |
| 6.5      | Checagem E Transformacao De Caractere                                   | 38        | 8.6      | Kadane 2D                           | 45        |
| 6.6      | Conta Digitos 1 Ate N   | 38        | 8.7      | Oito Rainhas                        | 46        |
| 6.7      | Escrita Em Arquivo  | 38        | 8.8      | Sliding Window Minimum              | 46        |
| 6.8      | Gcd   | 38        | 8.9      | Torre De Hanoi                      | 46        |
| 6.9      | Hipotenusa  | 38        | 8.10     | Kadane (Segment Tree)               | 46        |
| 6.10     | Int To Binary String  | 38        | 8.11     | Largest Area In Histogram           | 47        |
| 6.11     | Int To String   | 38        | 8.12     | Point Compression                   | 47        |
| 6.12     | Leitura De Arquivo  | 39        | 8.13     | Ternary Search                      | 47        |
| 6.13     | Max E Min Element Num Vetor   | 39        | <b>9</b> | <b>Strings</b>                      | <b>48</b> |
| 6.14     | Permutacao  | 39        | 9.1      | Kmp                                 | 48        |
| 6.15     | Remove Repeticoes Continuas Num Vetor                                   | 39        | 9.2      | Trie - Maximum Xor Sum              | 48        |
| 6.16     | Rotate (Left)   | 39        | 9.3      | Trie - Maximum Xor Two Elements     | 48        |
| 6.17     | Rotate (Right)  | 39        | 9.4      | Z-Function                          | 48        |
| 6.18     | Scanf De Uma String   | 39        | 9.5      | Aho Corasick                        | 48        |
| 6.19     | Split Function  | 39        | 9.6      | Hashing                             | 49        |
| 6.20     | String To Long Long   | 39        | 9.7      | Lcs K Strings                       | 50        |
| 6.21     | Substring   | 39        | 9.8      | Lexicographically Smallest Rotation | 51        |
| 6.22     | Width   | 39        | 9.9      | Manacher (Longest Palindrome)       | 51        |
| 6.23     | Check Overflow  | 39        | 9.10     | Suffix Array                        | 52        |
| 6.24     | Readint   | 40        | 9.11     | Suffix Array Pessoa                 | 53        |
| <b>7</b> | <b>Math</b>   | <b>40</b> | 9.12     | Suffix Array With Additional Memory | 55        |

## 9.13 Trie . . . . . 56

## 1. Template

```

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

```

## 2. Data Structures

## 2.1. Bit2D

```

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

```

```

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

```

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

```

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

```

```

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

```

## 2.3. Mos Algorithm

```

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

```

```

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

```

## 2.4. Sqrt Decomposition

```

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

```

## 2.5. Bit

```

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

```

```

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

```

## 2.6. Bit (Range Update)

```

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

```

```

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

```

```

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

```

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

```

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

```

## 2.8. Ordered Set

```

1 #include <bits/stdc++.h>
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/trie_policy.hpp>
4
5 using namespace std;
6 using namespace __gnu_pbds;
7
8 template <typename T>
9 using ordered_set =
10   tree<T, null_type, less<T>, rb_tree_tag,
11     tree_order_statistics_node_update>;
12
13 ordered_set<int> X;
14 X.insert(1);
15 X.insert(2);
16 X.insert(4);
17 X.insert(8);

```

```

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

```

## 2.9. Persistent Segment Tree

```

1  class Persistent_Seg_Tree {
2      struct Node {
3          int val;
4          Node *left, *right;
5          Node() {}
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      }
37
38      /// Builds version[0] with 0.
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
81      int pst_query(Node *node, const int l, const int r, const int i,
82                   const int j) {
83          if (l > j || r < i)
84              return _NEUTRAL_NODE.val;
85
86          if (i <= l && r <= j)
87              return node->val;
88
89          int mid = (l + r) / 2;
90          return merge_nodes(pst_query(node->left, l, mid, i, j),
91                          pst_query(node->right, mid + 1, r, i, j));
92      }
93
94  public:
95      Persistent_Seg_Tree(const int n, const int number_of_versions) {
96          this->n = n;
97          version.resize(number_of_versions);
98          this->version[0] = this->pst_build_empty(this->version[0], 0, this->n -
99          1);
100
101          /// Constructor that allows to pass initial values to the leafs.
102          Persistent_Seg_Tree(const vector<int> &arr, const int number_of_versions) {
103              this->n = arr.size();
104              version.resize(number_of_versions);
105              this->version[0] = this->pst_build(this->version[0], 0, this->n - 1,
106              arr);
107
108          /// Links the root of a version to a previous version.
109          ///

```

```

109 /// Time Complexity: O(1)
110 void link(const int version, const int prev_version) {
111     assert(this->n > -1);
112     assert(0 <= prev_version);
113     assert(prev_version <= version);
114     assert(version < this->version.size());
115     this->version[version] = this->version[prev_version];
116 }
117
118 /// Updates an index in cur_tree based on prev_tree with a delta.
119 /// Time Complexity: O(log(n))
120 void update(const int cur_version, const int prev_version, const int idx,
121            const int delta) {
122     assert(this->n > -1);
123     assert(0 <= prev_version);
124     assert(prev_version <= cur_version);
125     assert(cur_version < this->version.size());
126     this->version[cur_version] = this->pst_update(this->version[cur_version],
127     this->version[prev_version],
128     delta);
129 }
130
131 /// Query from l to r.
132 /// Time Complexity: O(log(n))
133 int query(const int version, const int l, const int r) {
134     assert(this->n > -1);
135     assert(this->version[version] != nullptr);
136     assert(0 <= l);
137     assert(l <= r);
138     assert(r < this->n);
139     return this->pst_query(this->version[version], 0, this->n - 1, l, r);
140 }
141 };
142
143

```

## 2.10. Segment Tree

```

1 class Seg_Tree {
2 public:
3     struct Node {
4         int val, lazy;
5
6         Node() {}
7         Node(const int val, 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     /// Constructor responsible for building the initial tree based on a
144     /// vector.
145     ///
146     /// Time Complexity O(n)
147     Seg_Tree(const vector<int> &arr) { this->build(arr); }
148
149     /// Returns the first index from left to right.
150     /// Uncomment the line in the original funtion to get the proper element
151     /// that
152     /// may be: GREATER OR EQUAL, GREATER, SMALLER OR EQUAL, SMALLER.
153     ///

```

```

152     /// Time Complexity O(log n)
153     int get_first(const int v) {
154         assert(this->n >= 0);
155         return this->st_get_first(0, this->n - 1, v, 0);
156     }
157
158     /// Update at a single index.
159     ///
160     /// Time Complexity O(log n)
161     void update(const int idx, const int delta) {
162         assert(this->n >= 0);
163         assert(0 <= idx), assert(idx < this->n);
164         this->st_update(0, this->n - 1, idx, idx, delta, 0);
165     }
166
167     /// Range update from l to r.
168     ///
169     /// Time Complexity O(log n)
170     void update(const int l, const int r, const int delta) {
171         assert(this->n >= 0);
172         assert(0 <= l), assert(l <= r), assert(r < this->n);
173         this->st_update(0, this->n - 1, l, r, delta, 0);
174     }
175
176     /// Query at a single index.
177     ///
178     /// Time Complexity O(log n)
179     int query(const int idx) {
180         assert(this->n >= 0);
181         assert(0 <= idx), assert(idx < this->n);
182         return this->st_query(0, this->n - 1, idx, idx, 0).val;
183     }
184
185     /// Range query from l to r.
186     ///
187     /// Time Complexity O(log n)
188     int query(const int l, const int r) {
189         assert(this->n >= 0);
190         assert(0 <= l), assert(l <= r), assert(r < this->n);
191         return this->st_query(0, this->n - 1, l, r, 0).val;
192     }
193 };

```

## 2.11. Segment Tree 2D

```

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

```

```

20 // }
21
22 private:
23     int n, m;
24
25 public:
26     vector<Seg_Tree> tree;
27
28 private:
29     void st_build(const int l, const int r, const int pos, const
30         vector<vector<int>> &mat) {
31         if(l == r)
32             tree[pos] = Seg_Tree(mat[l]);
33         else {
34             int mid = (l + r) / 2;
35             st_build(l, mid, 2*pos + 1, mat);
36             st_build(mid + 1, r, 2*pos + 2, mat);
37             for(int i = 0; i < tree[2*pos + 1].tree.size(); i++)
38                 tree[pos].tree[i].val = merge_nodes(tree[2*pos + 1].tree[i].val,
39                                                         tree[2*pos + 2].tree[i].val);
40         }
41     }
42
43     int st_query(const int l, const int r, const int x1, const int y1, const
44         int x2, const int y2, const int pos) {
45         if(l > x2 || r < x1)
46             return NEUTRAL_VALUE;
47
48         if(x1 <= l && r <= x2)
49             return tree[pos].query(y1, y2);
50
51         int mid = (l + r) / 2;
52         return merge_nodes(st_query(l, mid, x1, y1, x2, y2, 2*pos + 1),
53                             st_query(mid + 1, r, x1, y1, x2, y2, 2*pos + 2));
54     }
55
56     void st_update(const int l, const int r, const int x, const int y, const
57         int delta, const int pos) {
58         if(l > x || r < x)
59             return;
60
61         // Only supports point updates.
62         if(l == r) {
63             tree[pos].update(y, delta);
64             return;
65         }
66
67         int mid = (l + r) / 2;
68         st_update(l, mid, x, y, delta, 2*pos + 1);
69         st_update(mid + 1, r, x, y, delta, 2*pos + 2);
70         tree[pos].update(y, delta);
71     }
72
73 public:
74     Seg_Tree_2d() {
75         this->n = -1;
76         this->m = -1;
77     }
78
79     Seg_Tree_2d(const int n, const int m) {
80         this->n = n;
81         this->m = m;
82         // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
83         assert(m < 10000);
84         tree.resize(4 * n, Seg_Tree(m));

```

```

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 };

```

## 2.12. Segment Tree Polynomial

```

1 // Works for the polynomial f(x) = z1*x + z0
2 class Seg_Tree {
3 public:
4     struct Node {
5         int val, z1, z0;
6
7         Node() {}
8         Node(const int val, const int z1, const int z0) {
9             : val(val), z1(z1), z0(z0) {}
10     };
11
12 private:
13     // range sum
14     Node NEUTRAL_NODE = Node(0, 0, 0);
15     Node merge_nodes(const Node &x, const Node &y) {
16         return Node(x.val + y.val, 0, 0);
17     }
18
19     void apply_lazy(const int l, const int r, const int pos) {
20         tree[pos].val += (r - l + 1) * tree[pos].z0;
21         tree[pos].val += (r - l) * (r - l + 1) / 2 * tree[pos].z1;
22     }
23
24 private:
25     int n;
26
27 public:
28     vector<Node> tree;
29
30 private:
31     void st_propagate(const int l, const int r, const int pos) {
32         if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {

```

```

32     apply_lazy(l, r, pos);
33     int mid = (l + r) / 2;
34     int sz_left = mid - l + 1;
35     if (l != r) {
36         tree[2 * pos + 1].z0 += tree[pos].z0;
37         tree[2 * pos + 1].z1 += tree[pos].z1;
38
39         tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
40         tree[2 * pos + 2].z1 += tree[pos].z1;
41     }
42     tree[pos].z0 = 0;
43     tree[pos].z1 = 0;
44 }
45 }
46
47 Node st_build(const int l, const int r, const vector<int> &arr,
48              const int pos) {
49     if (l == r)
50         return tree[pos] = Node(arr[l], 0, 0);
51
52     int mid = (l + r) / 2;
53     return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
54                                   st_build(mid + 1, r, arr, 2 * pos + 2));
55 }
56
57 Node st_query(const int l, const int r, const int i, const int j,
58              const int pos) {
59     st_propagate(l, r, pos);
60
61     if (l > r || l > j || r < i)
62         return NEUTRAL_NODE;
63
64     if (i <= l && r <= j)
65         return tree[pos];
66
67     int mid = (l + r) / 2;
68     return merge_nodes(st_query(l, mid, i, j, 2 * pos + 1),
69                       st_query(mid + 1, r, i, j, 2 * pos + 2));
70 }
71
72 // it adds a number delta to the range from i to j
73 Node st_update(const int l, const int r, const int i, const int j,
74               const int z1, const int z0, const int pos) {
75     st_propagate(l, r, pos);
76
77     if (l > r || l > j || r < i)
78         return tree[pos];
79
80     if (i <= l && r <= j) {
81         tree[pos].z0 = (l - i + 1) * z0;
82         tree[pos].z1 = z1;
83         st_propagate(l, r, pos);
84         return tree[pos];
85     }
86
87     int mid = (l + r) / 2;
88     return tree[pos] =
89         merge_nodes(st_update(l, mid, i, j, z1, z0, 2 * pos + 1),
90                   st_update(mid + 1, r, i, j, z1, z0, 2 * pos + 2));
91 }
92
93 public:
94     Seg_Tree() : n(-1) {}
95

```

```

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

```

## 2.13. Sparse Table

```

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:  $O(n \cdot \log(n))$ 
17     /// Space Complexity:  $O(n \cdot \log(n))$ 
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
35 int merge(const int &l, const int &r) {
36     return min(l, r);
37 }
38
39 public:
40 Sparse_Table(const vector<int> &arr) {
41     this->n = arr.size();
42
43     this->build_log_array();
44     this->build_sparse_table(arr);
45 }
46
47 void print() {
48     int pow2 = 1;
49     for(int i = 0; i < table.size(); i++) {
50         int sz = (int)(table.front().size()) - pow2 + 1;
51         for(int j = 0; j < sz; j++) {
52             cout << table[i][j] << " \n"[(j+1) == sz];
53         }
54         pow2 <= 1;
55     }
56 }
57
58 /// Query of a range from l to r.
59 ///
60 /// Time Complexity: O(1)
61 /// Space Complexity: O(1)
62 int query(int l, int r) {
63     assert(l <= r);
64     assert(0 <= l && r <= this->n - 1);
65
66     int lgg = lg[(r - l + 1)];
67     return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
68 }
69 };

```

### 3. Dp

#### 3.1. Achar Maior Palindromo

1 Fazer LCS da string com o reverso

#### 3.2. Digit Dp

```

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

```

```

8 /// f = the number we are building has already become smaller than b? [0 =
  no, 1 = yes]
9
10 int call(int pos, int cnt, int f){
11     if(cnt > k) return 0;
12
13     if(pos == num.size()){
14         if(cnt == k) return 1;
15         return 0;
16     }
17
18     if(DP[pos][cnt][f] != -1) return DP[pos][cnt][f];
19     int res = 0;
20     int lim = (f ? 9 : num[pos]);
21
22     /// Try to place all the valid digits such that the number doesn't exceed b
23     for(int dgt = 0; dgt <= LMT; dgt++){
24         int nf = f;
25         int ncnt = cnt;
26         if(f == 0 && dgt < LMT) nf = 1; /// The number is getting smaller at
           this position
27         if(dgt == d) ncnt++;
28         if(ncnt <= k) res += call(pos+1, ncnt, nf);
29     }
30
31     return DP[pos][cnt][f] = res;
32 }
33
34 int solve(int b){
35     num.clear();
36     while(b>0){
37         num.push_back(b%10);
38         b/=10;
39     }
40     reverse(num.begin(), num.end());
41     /// Stored all the digits of b in num for simplicity
42
43     memset(DP, -1, sizeof(DP));
44     int res = call(0, 0, 0);
45     return res;
46 }
47
48 int main () {
49
50     cin >> a >> b >> d >> k;
51     int res = solve(b) - solve(a-1);
52     cout << res << endl;
53
54     return 0;
55 }

```

#### 3.3. Longest Common Subsequence

```

1 string lcs(string &s, string &t) {
2
3     int n = s.size(), m = t.size();
4
5     s.insert(s.begin(), '#');
6     t.insert(t.begin(), '$');
7
8     vector<vector<int>> mat(n + 1, vector<int>(m + 1, 0));
9
10    for(int i = 1; i <= n; i++) {
11        for(int j = 1; j <= m; j++) {

```

```

12     if(s[i] == t[j])
13         mat[i][j] = mat[i - 1][j - 1] + 1;
14     else
15         mat[i][j] = max(mat[i - 1][j], mat[i][j - 1]);
16     }
17 }
18
19 string ans;
20 int i = n, j = m;
21 while(i > 0 && j > 0) {
22     if(s[i] == t[j])
23         ans += s[i], i--, j--;
24     else if(mat[i][j - 1] > mat[i - 1][j])
25         j--;
26     else
27         i--;
28 }
29 reverse(ans.begin(), ans.end());
30 return ans;
31 }
32

```

### 3.4. Longest Common Substring

```

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

```

### 3.5. Longest Increasing Subsequence 2D (Not Sorted)

```

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

```

```

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

```

### 3.6. Longest Increasing Subsequence 2D (Sorted)

```

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

```

```

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

```

### 3.7. 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    return lis.size();
25
26 }

```

### 3.8. Subset Sum Com Bitset

```

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

```

### 3.9. Catalan

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

### 3.10. Catalan

```

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

```

### 3.11. Coin Change Problem

```

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

```

### 3.12. Knapsack

```

1 int dp[2001][2001];
2 int moc(int q,int p,vector<ii> vec) {
3     for(int i = 1; i <= q; i++)
4     {
5         for(int j = 1; j <= p; j++) {
6             if(j >= vec[i-1].ff)
7                 dp[i][j] = max(dp[i-1][j],vec[i-1].ss + dp[i-1][j-vec[i-1].ff]);
8             else
9                 dp[i][j] = dp[i-1][j];
10        }
11    }
12    return dp[q][p];
13 }
14 int main(int argc, char *argv[])
15 {
16     int p,q;
17     vector<ii> vec;
18     cin >> p >> q;
19     int x,y;
20     for(int i = 0; i < q; i++) {
21         cin >> x >> y;
22         vec.push_back(make_pair(x,y));
23     }
24     for(int i = 0; i <= p; i++)
25         dp[0][i] = 0;
26     for(int i = 1; i <= q; i++)
27         dp[i][0] = 0;
28     sort(vec.begin(),vec.end());
29     cout << moc(q,p,vec) << endl;
30 }

```

## 4. Geometry

### 4.1. Centro De Massa De Um Poligono

```

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

```

### 4.2. Closest Pair Of Points

```

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

```

```

46     Pyr[ri++] = Py[i];
47
48     float dl = closestUtil(Px, Py1, mid);
49     float dr = closestUtil(Px + mid, Pyr, n-mid);
50     float d = min(dl, dr);
51     Point strip[n];
52     int j = 0;
53     for (int i = 0; i < n; i++)
54         if (abs(Py[i].x - midPoint.x) < d)
55             strip[j] = Py[i], j++;
56     return min(d, stripClosest(strip, j, d));
57 }
58
59 float closest(Point P[], int n) {
60     Point Px[n];
61     Point Py[n];
62     for (int i = 0; i < n; i++) {
63         Px[i] = P[i];
64         Py[i] = P[i];
65     }
66     qsort(Px, n, sizeof(Point), compareX);
67     qsort(Py, n, sizeof(Point), compareY);
68     return closestUtil(Px, Py, n);
69 }

```

#### 4.3. Condicao De Existencia De Um Triangulo

```

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

```

#### 4.4. Convex Hull

```

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

```

```

23         // trocar por >= 0
24         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
25             k--;
26         H[k++] = P[i];
27     }
28     // Build upper hull
29     for (int i = n - 2, t = k + 1; i >= 0; i--) {
30         // esse <= 0 representa sentido anti-horario, caso deseje mudar
31         // trocar por >= 0
32         while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
33             k--;
34         H[k++] = P[i];
35     }
36     H.resize(k);
37     /* o último ponto do vetor é igual ao primeiro, atente para isso
38        as vezes é necessário mudar */
39     return H;
40 }

```

#### 4.5. Cross Product

```

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

```

#### 4.6. Distance Point Segment

```

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

```

#### 4.7. Line-Line Intersection

```

1 // Intersecção de retas Ax + By = C dados pontos (x1,y1) e (x2,y2)
2 A = y2-y1
3 B = x1-x2
4 C = A*x1+B*y1
5 //Retas definidas pelas equações:
6 A1x + B1y = C1
7 A2x + B2y = C2

```



```

8 //Encontrar x e y resolvendo o sistema
9 double det = A1*B2 - A2*B1;
10 if(det == 0){
11     //Lines are parallel
12 }else{
13     double x = (B2*C1 - B1*C2)/det;
14     double y = (A1*C2 - A2*C1)/det;
15 }

```

#### 4.8. Line-Point Distance

```

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

```

#### 4.9. Point Inside Convex Polygon - Log(N)

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define INF 1e18
6 #define pb push_back
7 #define ii pair<int,int>

```

```

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

```

```

72 pto lo = {p.x, -(double)INF};
73 pto hi = {p.x, (double)INF};
74 auto itl = lower_bound(lower.begin(), lower.end(), lo);
75 auto itu = lower_bound(upper.begin(), upper.end(), lo);
76
77 if(itl == lower.begin() || itu == upper.begin()) {
78     auto it = lower_bound(arr.begin(), arr.end(), lo);
79     auto it2 = lower_bound(arr.begin(), arr.end(), hi);
80     it2--;
81     if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y
82     <= it2->y)
83         return true;
84     return false;
85 }
86 if(itl == lower.end() || itu == upper.end()) {
87     return false;
88 }
89
90 auto ol = itl, ou = itu;
91 ol--, ou--;
92 if(cross(*ol, *itl, p) >= 0 && cross(*ou, *itu, p) <= 0)
93     return true;
94
95 auto it = lower_bound(arr.begin(), arr.end(), lo);
96 auto it2 = lower_bound(arr.begin(), arr.end(), hi);
97 it2--;
98 if(it2 >= it && p.x == it->x && it->x == it2->x && it->y <= p.y && p.y <=
99     it2->y)
100     return true;
101 return false;
102 }
103
104 signed main () {
105     ios_base::sync_with_stdio(false);
106     cin.tie(NULL);
107
108     double n, m, k;
109
110     cin >> n >> m >> k;
111
112     vector<pto> arr(n);
113
114     for(pto &x: arr) {
115         cin >> x.x >> x.y;
116     }
117
118     convex_hull(arr);
119
120     pto p;
121
122     int c = 0;
123     while(m--) {
124         cin >> p.x >> p.y;
125         cout << (insidePolygon(p, arr) ? "dentro" : "fora") << endl;
126     }
127 }
128
129
130

```

## 4.10. Point Inside Polygon

```

1
2 /* Traça-se uma reta do ponto até um outro ponto qualquer fora do triangulo
   e checa o número de interseção com a borda do polígono se este for 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 - q.x) -
29             (q.x - p.x) * (r.y - q.y);
30
31     if (val == 0) return 0; // colinear
32     return (val > 0) ? 1 : 2; // clock or counterclock wise
33 }
34
35 // The function that returns true if line segment 'p1q1'
36 // and 'p2q2' intersect.
37 bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
38     // Find the four orientations needed for general and
39     // special cases
40     int o1 = orientation(p1, q1, p2);
41     int o2 = orientation(p1, q1, q2);
42     int o3 = orientation(p2, q2, p1);
43     int o4 = orientation(p2, q2, q1);
44
45     // General case
46     if (o1 != o2 && o3 != o4)
47         return true;
48
49     // Special Cases
50     // p1, q1 and p2 are colinear and p2 lies on segment p1q1
51     if (o1 == 0 && onSegment(p1, p2, q1)) return true;
52
53     // p1, q1 and p2 are colinear and q2 lies on segment p1q1
54     if (o2 == 0 && onSegment(p1, q2, q1)) return true;
55
56     // p2, q2 and p1 are colinear and p1 lies on segment p2q2
57     if (o3 == 0 && onSegment(p2, p1, q2)) return true;
58
59     // p2, q2 and q1 are colinear and q1 lies on segment p2q2
60     if (o4 == 0 && onSegment(p2, q1, q2)) return true;
61
62     return false; // Doesn't fall in any of the above cases
63 }

```

```

64 // Returns true if the pto p lies inside the polygon[] with n vertices
65 bool isInside(pto polygon[], int n, pto p) {
66     // There must be at least 3 vertices in polygon[]
67     if (n < 3) return false;
68     // Create a pto for line segment from p to infinite
69     pto extreme = pto(INF, p.y);
70     // Count intersections of the above line with sides of polygon
71     int count = 0, i = 0;
72     do {
73         int next = (i+1)%n;
74         // Check if the line segment from 'p' to 'extreme' intersects
75         // with the line segment from 'polygon[i]' to 'polygon[next]'
76         if (doIntersect(polygon[i], polygon[next], p, extreme)) {
77             // If the pto 'p' is colinear with line segment 'i-next',
78             // then check if it lies on segment. If it lies, return true,
79             // otherwise false
80             if (orientation(polygon[i], p, polygon[next]) == 0)
81                 return onSegment(polygon[i], p, polygon[next]);
82             count++;
83         }
84         i = next;
85     } while (i != 0);
86     // Return true if count is odd, false otherwise
87     return count%2 == 1; // Same as (count%2 == 1)
88 }

```

#### 4.11. Points Inside And In Boundary Polygon

```

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

```

```

31 int internalCount(vector<pto> &arr, int n) {
32
33     int area_2 = polygonArea2(arr, n);
34     int boundPoints = totalBoundaryPolygon(arr,n);
35     return (area_2 - boundPoints + 2)/2;
36 }

```

#### 4.12. Polygon Area (3D)

```

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

```

#### 4.13. Polygon Area

```

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

```

```

8     }
9     return (double)abs(area/2.0);
10 }

```

#### 4.14. Segment-Segment Intersection

```

1 // Given three colinear points p, q, r, the function checks if
2 // point q lies on line segment 'pr'
3 int onSegment(Point p, Point q, Point r) {
4     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y <= max(p.y, r.y)
5         && q.y >= min(p.y, r.y))
6         return true;
7     return false;
8 }
9 /* PODE SER RETIRADO
10 int onSegmentNotBorda(Point p, Point q, Point r) {
11     if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) && q.y <= max(p.y, r.y)
12         && q.y >= min(p.y, r.y))
13         return true;
14     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y < max(p.y, r.y)
15         && q.y > min(p.y, r.y))
16         return true;
17     return false;
18 }
19 */
20 // To find orientation of ordered triplet (p, q, r).
21 // The function returns following values
22 // 0 --> p, q and r are colinear
23 // 1 --> Clockwise
24 // 2 --> Counterclockwise
25 int orientation(Point p, Point q, Point r) {
26     int val = (q.y - p.y) * (r.x - q.x) -
27             (q.x - p.x) * (r.y - q.y);
28     if (val == 0) return 0; // colinear
29     return (val > 0)? 1: 2; // clock or counterclock wise
30 }
31 // The main function that returns true if line segment 'p1p2'
32 // and 'q1q2' intersect.
33 int doIntersect(Point p1, Point p2, Point q1, Point q2) {
34     // Find the four orientations needed for general and
35     // special cases
36     int o1 = orientation(p1, p2, q1);
37     int o2 = orientation(p1, p2, q2);
38     int o3 = orientation(q1, q2, p1);
39     int o4 = orientation(q1, q2, p2);
40
41     // General case
42     if (o1 != o2 && o3 != o4) return 2;
43
44     /* PODE SER RETIRADO
45     if(o1 == o2 && o2 == o3 && o3 == o4 && o4 == 0) {
46         //INTERCEPTAM EM RETA
47         if(onSegmentNotBorda(p1,q1,p2) || onSegmentNotBorda(p1,q2,p2)) return 1;
48         if(onSegmentNotBorda(q1,p1,q2) || onSegmentNotBorda(q1,p2,q2)) return 1;
49     }
50     */
51     // Special Cases (INTERCEPTAM EM PONTO)
52     // p1, p2 and q1 are colinear and q1 lies on segment p1p2
53     if (o1 == 0 && onSegment(p1, q1, p2)) return 2;
54     // p1, p2 and q1 are colinear and q2 lies on segment p1p2
55     if (o1 == 0 && onSegment(p1, q2, p2)) return 2;
56     // q1, q2 and p1 are colinear and p1 lies on segment q1q2
57     if (o3 == 0 && onSegment(q1, p1, q2)) return 2;
58     // q1, q2 and p2 are colinear and p2 lies on segment q1q2

```

```

56     if (o4 == 0 && onSegment(q1, p2, q2)) return 2;
57     return false; // Doesn't fall in any of the above cases
58 }
59 // OBS: SE (C2/A2 == C1/A1) SÃO COLINEARES

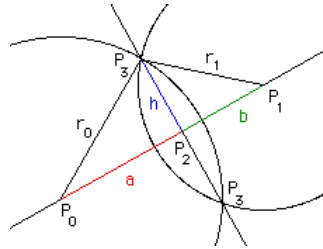
```

#### 4.15. Upper And Lower Hull

```

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

```



## 4.16. Circle Circle Intersection

## 4.17. Circle Circle Intersection

```

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

```

```

48
49     /* Determine the distance from point 0 to point 2. */
50     a = ((r0 * r0) - (r1 * r1) + (d * d)) / (2.0 * d);
51
52     /* Determine the coordinates of point 2. */
53     x2 = x0 + (dx * a / d);
54     y2 = y0 + (dy * a / d);
55
56     /* Determine the distance from point 2 to either of the
57      * intersection points.
58      */
59     h = sqrt((r0 * r0) - (a * a));
60
61     /* Now determine the offsets of the intersection points from
62      * point 2.
63      */
64     rx = -dy * (h / d);
65     ry = dx * (h / d);
66
67     /* Determine the absolute intersection points. */
68     *xi = x2 + rx;
69     *xi_prime = x2 - rx;
70     *yi = y2 + ry;
71     *yi_prime = y2 - ry;
72
73     return 1;
74 }

```

## 4.18. Struct Point And Line

```

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

```

```

35 double det(const Point &a, const Point &b) {
36     return a.x * b.y - a.y * b.x;
37 }
38 double dot(const Point &a, const Point &b) {
39     return a.x * b.x + a.y * b.y;
40 }
41 double dist(const Point &a, const Point &b) {
42     return (a-b).norm();
43 }
44
45 struct Line {
46     Point a, b;
47     Line() {}
48     Line(Point x, Point y): a(x), b(y) {};
49 };
50
51 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. 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
34 // Returns the maximum flow from s to t in the given graph
35 int fordFulkerson(int s, int t) {
36     int u, v;
37     // Create a residual graph and fill the residual graph with given
38     // capacities in the original graph as residual capacities in residual
39     // graph residual capacity of edge from i to j (if there is an edge. If
40     // rGraph[i][j] is 0, then there is not)
41     for (u = 0; u < V; u++)
42         for (v = 0; v < V; v++)
43             rGraph[u][v] = graph[u][v];
44
45     int parent[V]; // This array is filled by BFS and to store path
46
47     int max_flow = 0; // There is no flow initially
48
49     // Augment the flow while there is path from source to sink
50     while (bfs(s, t, parent)) {
51         // Find minimum residual capacity of the edges along the path filled by
52         // BFS. Or we can say find the maximum flow through the path found.
53         int path_flow = INT_MAX;
54         for (v=t; v!=s; v=parent[v]) {
55             u = parent[v];
56             path_flow = min(path_flow, rGraph[u][v]);
57         }
58
59         // update residual capacities of the edges and reverse edges
60         // along the path
61         for (v=t; v!=s; v=parent[v]) {
62             u = parent[v];
63             rGraph[u][v] -= path_flow;
64             rGraph[v][u] += path_flow;
65         }
66
67         // Add path flow to overall flow
68         max_flow += path_flow;
69     }
70 }

```

```

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

```

### 5.5. 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                 // ponte de u para v
28             }
29         }
30         else
31             Low[u]=min(Low[u], T[v]);
32         ciclo[u] |= (T[u]>=Low[v]);
33         //checa se o vértice u faz parte de um ciclo
34     }
35 }

```

```

34 void clear() {
35
36     for(int i = 0; i <= n; i++) {
37         T[i] = 0, Low[i] = 0, adj[i].clear(), ciclo[i] = false;
38     }
39     bridges.clear();
40 }
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 }

```

## 5.6. 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 int main() {
36
37     for(int i = 0; i < n; i++)
38         if(T[i] == 0)
39             artPoint(i, -1);
40
41 }

```

## 5.7. 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 };

```

## 5.8. 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             if(src == -1 && dest == -1)
31                 return make_pair(true, pair<int, int>(src, dest));
32             else if(src != -1 && dest != -1)
33                 return make_pair(true, pair<int, int>(src, dest));
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     /// Time complexity: O(V * (log V) + E)
74     bool has_euler_tour_directed(const vector<vector<edge>> &adj, const
75         vector<int> &in_degree) {
76         const pair<bool, pair<int, int>> aux = detail::check_both_directed(adj,
77             in_degree);
78         const bool valid = aux.first;
79         const int src = aux.second.first;
80         const int dest = aux.second.second;
81         return (valid && src == -1 && dest == -1);
82     }
83
84     /// A directed graph has an eulerian path/tour if has:
85     /// - One vertex v such that out_degree[v] - in_degree[v] == 1
86     /// - One vertex v such that in_degree[v] - out_degree[v] == 1
87     /// - The remaining vertices v such that in_degree[v] == out_degree[v]
88     /// or
89     /// - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR
90
91     /// Returns a boolean value that indicates whether there's a path or not.
92     /// If there's a valid path it also returns two numbers: the source and
93     /// the destination.
94     /// If the source and destination can be an arbitrary vertex it will
95     /// return the pair (-1, -1)
96     /// for the source and destination (it means the contains an eulerian
97     /// tour).
98     ///

```

```

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
141  // back to the first edge on the tour.
142  //
143  // A graph with an euler path has zero odd degree vertex.
144  //
145  // Time Complexity: O(V)
146  bool has_euler_tour_undirected(const vector<int> &degree) {
147      for(int i = 0; i < degree.size(); i++)
148          if(degree[i] & 1)
149              return false;
150      return true;

```

```

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

```

### 5.9. Bellman Ford

```

1 struct edge {
2     int src, dest, weight;
3     edge() {}
4     edge(int src, int dest, int weight) : src(src), dest(dest), weight(weight)
5     {}
6     bool operator<(const edge &a) const {
7         return weight < a.weight;
8     }
9 };
10
11 /// Works to find the shortest path with negative edges.
12 /// Also detects cycles.
13 ///
14 /// Time Complexity: O(n * e)
15 /// Space Complexity: O(n)
16 bool bellman_ford(vector<edge> &edges, int src, int n) {
17     // n = qtd of vertices, E = qtd de arestas
18
19     // To calculate the shortest path uncomment the line below
20     // vector<int> dist(n, INF);
21
22     // To check cycles uncomment the line below
23     // vector<int> dist(n, 0);
24
25     vector<int> pai(n, -1);
26     int E = edges.size();
27
28     dist[src] = 0;
29     // Relax all edges n - 1 times.
30     // A simple shortest path from src to any other vertex can have at-most n
31     // - 1 edges.
32     for (int i = 1; i <= n - 1; i++) {
33         for (int j = 0; j < E; j++) {
34             int u = edges[j].src;
35             int v = edges[j].dest;
36             int weight = edges[j].weight;
37             if (dist[u] != INF && dist[u] + weight < dist[v]) {
38                 dist[v] = dist[u] + weight;
39                 pai[v] = u;
40             }
41         }
42     }
43
44     // Check for NEGATIVE-WEIGHT CYCLES.
45     // The above step guarantees shortest distances if graph doesn't contain
46     // negative weight cycle.
47     // If we get a shorter path, then there is a cycle.
48     bool is_cycle = false;
49     int vert_in_cycle;
50     for (int i = 0; i < E; i++) {
51         int u = edges[i].src;
52         int v = edges[i].dest;
53         int weight = edges[i].weight;
54         if (dist[u] != INF && dist[u] + weight < dist[v]) {

```

```

53         is_cycle = true;
54         pai[v] = u;
55         vert_in_cycle = v;
56     }
57 }
58
59 if(is_cycle) {
60     for(int i = 0; i < n; i++)
61         vert_in_cycle = pai[vert_in_cycle];
62
63     vector<int> cycle;
64     for(int v = vert_in_cycle; (v != vert_in_cycle || cycle.size() <= 1) ; v
65         = pai[v])
66         cycle.pb(v);
67
68     reverse(cycle.begin(), cycle.end());
69
70     for(int x: cycle) {
71         cout << x + 1 << ' ';
72     }
73     cout << cycle.front() + 1 << endl;
74     return true;
75 } else
76     return false;

```

### 5.10. De Bruijn Sequence

```

1 // We can solve this problem by constructing a directed graph with
2 //  $k^{(n-1)}$  nodes with each node having k outgoing edges_order. Each node
3 // corresponds to a string of size n-1. Every edge corresponds to one of the
4 // characters in A and adds that character to the starting string. For
5 // example,
6 // if n=3 and k=2, then we construct the following graph:
7 //
8 //      - 1 -> (01) - 1 ->
9 //      /      ^ |      \
10 // 0 -> (00)  1 0      (11) <- 1
11 //      \      | v      /
12 //      <- 0 - (10) <- 0 -
13
14 // The node '01' is connected to node '11' through edge '1', as adding '1' to
15 // '01' (and removing the first character) gives us '11'.
16
17 // We can observe that every node in this graph has equal in-degree and
18 // out-degree, which means that a Eulerian circuit exists in this graph.
19
20 namespace graph {
21 namespace detail {
22 // Finding an valid eulerian path
23 void dfs(const string &node, const string &alphabet, set<string> &vis,
24         string &edges_order) {
25     for (char c : alphabet) {
26         string nxt = node + c;
27         if (vis.count(nxt))
28             continue;
29
30         vis.insert(nxt);
31         nxt.erase(nxt.begin());
32         dfs(nxt, alphabet, vis, edges_order);
33         edges_order += c;
34     }
35 }

```

```

35 }; // namespace detail
36
37 // Returns a string in which every string of the alphabet of size n appears
38 // in
39 // the resulting string exactly once.
40 //
41 // Time Complexity:  $O(\text{alphabet.size() } ^ n \star \log_2(\text{alphabet.size() } ^ n))$ 
42 string de_bruijn(const int n, const string &alphabet) {
43     set<string> vis;
44     string edges_order;
45
46     string starting_node = string(n - 1, alphabet.front());
47     detail::dfs(starting_node, alphabet, vis, edges_order);
48
49     return edges_order + starting_node;
50 }; // namespace graph

```

### 5.11. Dijkstra + Dij Graph

```

1 /// Works with 1-indexed graphs.
2 class Dijkstra {
3 private:
4     static constexpr int INF = 2e18;
5     bool CREATE_GRAPH = false;
6     int src;
7     int n;
8     vector<int> _dist;
9     vector<vector<int>> parent;
10
11 private:
12     void _compute(const int src, const vector<vector<pair<int, int>>> &adj) {
13         _dist.resize(this->n, INF);
14         vector<bool> vis(this->n, false);
15
16         if (CREATE_GRAPH) {
17             parent.resize(this->n);
18
19             for (int i = 0; i < this->n; i++)
20                 parent[i].emplace_back(i);
21         }
22
23         priority_queue<pair<int, int>, vector<pair<int, int>>,
24             greater<pair<int, int>>>
25             pq;
26         pq.emplace(0, src);
27         _dist[src] = 0;
28
29         while (!pq.empty()) {
30             int u = pq.top().second;
31             pq.pop();
32             if (vis[u])
33                 continue;
34             vis[u] = true;
35
36             for (const pair<int, int> &x : adj[u]) {
37                 int v = x.first;
38                 int w = x.second;
39
40                 if (_dist[u] + w < _dist[v]) {
41                     _dist[v] = _dist[u] + w;
42                     pq.emplace(_dist[v], v);
43                     if (CREATE_GRAPH) {
44                         parent[v].clear();

```

```

45         parent[v].emplace_back(u);
46     }
47     } else if (CREATE_GRAPH && _dist[u] + w == _dist[v]) {
48         parent[v].emplace_back(u);
49     }
50 }
51 }
52 }
53
54 vector<vector<int>> gen_dij_graph(const int dest) {
55     vector<vector<int>> dijkstra_graph(this->n);
56     vector<bool> vis(this->n, false);
57     queue<int> q;
58
59     q.emplace(dest);
60     while (!q.empty()) {
61         int v = q.front();
62         q.pop();
63
64         for (const int u : parent[v]) {
65             if (u == v)
66                 continue;
67             dijkstra_graph[u].emplace_back(v);
68             if (!vis[u]) {
69                 q.emplace(u);
70                 vis[u] = true;
71             }
72         }
73     }
74     return dijkstra_graph;
75 }
76
77 vector<int> gen_min_path(const int dest) {
78     vector<int> path;
79     vector<int> prev(this->n, -1);
80     vector<int> d(this->n, INF);
81     queue<int> q;
82
83     q.emplace(dest);
84     d[dest] = 0;
85
86     while (!q.empty()) {
87         int v = q.front();
88         q.pop();
89
90         for (const int u : parent[v]) {
91             if (u == v)
92                 continue;
93             if (d[v] + 1 < d[u]) {
94                 d[u] = d[v] + 1;
95                 prev[u] = v;
96                 q.emplace(u);
97             }
98         }
99     }
100
101     int cur = this->src;
102     while (cur != -1) {
103         path.emplace_back(cur);
104         cur = prev[cur];
105     }
106
107     return path;
108 }
109

```

```

110 public:
111     /// Allows creation of dijkstra graph and getting the minimum path.
112     Dijkstra(const int src, const bool create_graph,
113              const vector<vector<pair<int, int>>> &adj)
114         : n(adj.size()), src(src), CREATE_GRAPH(create_graph) {
115         this->_compute(src, adj);
116     }
117
118     /// Constructor that computes only the Dijkstra minimum path from src.
119     ///
120     /// Time Complexity: O(E log V)
121     Dijkstra(const int src, const vector<vector<pair<int, int>>> &adj)
122         : n(adj.size()), src(src) {
123         this->_compute(src, adj);
124     }
125
126     /// Returns the Dijkstra graph of the graph.
127     ///
128     /// Time Complexity: O(V)
129     vector<vector<int>> dij_graph(const int dest) {
130         assert(CREATE_GRAPH);
131         return gen_dij_graph(dest);
132     }
133
134     /// Returns the vertices present in a path from src to dest with
135     /// minimum cost and a minimum length.
136     ///
137     /// Time Complexity: O(V)
138     vector<int> min_path(const int dest) {
139         assert(CREATE_GRAPH);
140         return gen_min_path(dest);
141     }
142
143     /// Returns the distance from src to dest.
144     int dist(const int dest) {
145         assert(0 <= dest, assert(dest < n);
146         return _dist[dest];
147     }
148 };

```

## 5.12. 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)
10             : v(v), c(c), 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].emplace_back(FlowEdge(v, c, c, id2, false));
36         adj[v].emplace_back(FlowEdge(u, 0, 0, id1, true));
37     }
38
39     void add_to_src(int v, int c) {
40         adj[src].emplace_back(FlowEdge(v, c, c, -1, false));
41     }
42
43     void add_to_snk(int u, int c) {
44         adj[u].emplace_back(FlowEdge(snk, c, c, -1, false));
45     }
46
47     bool bfs() {
48         for (int i = 0; i < sz; i++) {
49             level[i] = -1;
50         }
51
52         level[src] = 0;
53         queue<int> q;
54         q.push(src);
55
56         while (!q.empty()) {
57             int cur = q.front();
58             q.pop();
59             for (FlowEdge e : adj[cur]) {
60                 if (level[e.v] == -1 && e.c > 0) {
61                     level[e.v] = level[cur] + 1;
62                     q.push(e.v);
63                 }
64             }
65         }
66
67         return (level[snk] == -1 ? false : true);
68     }
69
70     int send_flow(int u, int flow) {
71         if (u == snk)
72             return flow;
73
74         for (int &i = used[u]; i < adj[u].size(); i++) {
75             FlowEdge &e = adj[u][i];
76
77             if (level[u] + 1 != level[e.v] || e.c <= 0)
78                 continue;
79
80             int new_flow = min(flow, e.c);
81             int adjusted_flow = send_flow(e.v, new_flow);
82
83             if (adjusted_flow > 0) {
84                 e.c -= adjusted_flow;
85                 if (e.rev != -1)
86                     adj[e.v][e.rev].c += adjusted_flow;
87                 return adjusted_flow;

```

```

88     }
89 }
90
91 return 0;
92 }
93
94 int calculate() {
95     if (src == snk) {
96         max_flow = -1;
97         return -1;
98     } // not sure if needed
99
100     max_flow = 0;
101
102     while (bfs()) {
103         for (int i = 0; i < sz; i++)
104             used[i] = 0;
105         while (int inc = send_flow(src, INF))
106             max_flow += inc;
107     }
108     calculated = true;
109
110     return max_flow;
111 }
112
113 vector<pair<int, int>> mincut(vector<vector<int>> &mat_adj) {
114     assert(calculated);
115     int mat[sz][sz];
116     memset(mat, 0, sizeof mat);
117     for (int i = 0; i < V; i++)
118         for (FlowEdge x : adj[i])
119             mat[i][x.v] += x.c;
120
121     vector<bool> vis(sz);
122     queue<int> q;
123     q.push(src);
124     vis[src] = true;
125     while (!q.empty()) {
126         int u = q.front();
127         q.pop();
128         for (int v = 0; v < sz; v++) {
129             if (mat[u][v] > 0 && !vis[v]) {
130                 q.push(v);
131                 vis[v] = true;
132             }
133         }
134     }
135
136     vector<pair<int, int>> cut;
137     for (int i = 0; i < sz; i++)
138         for (int j = 0; j < sz; j++)
139             if (vis[i] && !vis[j])
140                 // if there's an edge from i to j.
141                 if (mat_adj[i][j] > 0)
142                     cut.emplace_back(i, j);
143
144     return cut;
145 }
146
147 vector<pair<int, int>> min_edge_cover() {
148     bool covered[sz];
149     for (int i = 0; i < sz; i++)
150         covered[i] = false;
151     vector<pair<int, int>> edge_cover;
152     for (int i = 1; i < sz - 1; i++) {

```

```

153         for (FlowEdge e : adj[i]) {
154             if (e.cap == 0 || e.v > sz - 2)
155                 continue;
156             if (e.c == 0) {
157                 edge_cover.emplace_back(pair<int, int>(i, e.v));
158                 covered[i] = true;
159                 covered[e.v] = true;
160                 break;
161             }
162         }
163     }
164     for (int i = 1; i < sz - 1; i++) {
165         for (FlowEdge e : adj[i]) {
166             if (e.cap == 0 || e.v > sz - 2)
167                 continue;
168             if (e.c == 0)
169                 continue;
170             if (!covered[i] || !covered[e.v]) {
171                 edge_cover.emplace_back(pair<int, int>(i, e.v));
172                 covered[i] = true;
173                 covered[e.v] = true;
174             }
175         }
176     }
177     return edge_cover;
178 }
179
180 vector<vector<int>> allFlow() {
181     assert(calculated);
182     vector<vector<int>> ret(V, vector<int>(V, 0));
183
184     for (int i = 0; i < V; i++) {
185         for (FlowEdge x : adj[i]) {
186             if (x.is_rev)
187                 continue;
188             // flow from vertex i to x.v
189             ret[i][x.v] += x.cap - x.c;
190         }
191     }
192
193     // for(int i = 0; i < V; i++) {
194     //     for(int j = 0; j < V; j++) {
195     //         cout << ret[i][j] << ' ';
196     //     }
197     //     cout << endl;
198     // }
199
200     return ret;
201 }
202
203 void dfs_build_path(int u, vector<int> &path, vector<vector<int>>
204     &mat_flow,
205     vector<vector<int>> &ans, vector<vector<int>> &adj) {
206     path.emplace_back(u);
207
208     if (u == this->snk) {
209         ans.emplace_back(path);
210         return;
211     }
212
213     for (int v : adj[u]) {
214         if (mat_flow[u][v]) {
215             mat_flow[u][v]--;
216             dfs_build_path(v, path, mat_flow, ans, adj);
217             return;

```

```

217     }
218 }
219 }
220
221 vector<vector<int>> get_all_paths(vector<vector<int>> &adj) {
222     assert(calculated);
223
224     vector<vector<int>> mat_flow = allFlow();
225     vector<vector<int>> ans;
226     ans.reserve(max_flow);
227
228     for (int i = 0; i < max_flow; i++) {
229         vector<int> path;
230         path.reserve(V);
231         dfs_build_path(this->src, path, mat_flow, ans, adj);
232     }
233
234     return ans;
235 }
236 };

```

### 5.13. Floyd Warshall

```

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

```

### 5.14. Functional Graph

```

1  // Based on:
2  // http://maratona.ic.unicamp.br/MaratonaVerao2020/lecture-b/20200122.pdf
3  class Functional_Graph {
4      // FOR DIRECTED GRAPH
5      private:
6          void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis) {
7              int id_cycle = cycle_cnt++;
8              int cur_id = 0;
9              this->first[id_cycle] = u;
10
11              while(!vis[u]) {
12                  vis[u] = true;
13
14                  this->cycle[id_cycle].push_back(u);
15
16                  this->in_cycle[u] = true;
17                  this->cycle_id[u] = id_cycle;
18                  this->id_in_cycle[u] = cur_id;
19                  this->near_in_cycle[u] = u;
20                  this->id_near_cycle[u] = id_cycle;
21                  this->cycle_dist[u] = 0;
22
23                  u = nxt[u];
24                  cur_id++;
25              }

```

```

26     }
27
28     // Time Complexity: O(V)
29     void build(int n, int indexed_from, vector<int> &nxt, vector<int>
30         &in_degree) {
31         queue<int> q;
32         vector<bool> vis(n + indexed_from);
33         for (int i = indexed_from; i < n + indexed_from; i++) {
34             if(in_degree[i] == 0) {
35                 q.push(i);
36                 vis[i] = true;
37             }
38         }
39
40         vector<int> process_order;
41         process_order.reserve(n + indexed_from);
42         while(!q.empty()) {
43             int u = q.front();
44             q.pop();
45
46             process_order.push_back(u);
47
48             if(--in_degree[nxt[u]] == 0) {
49                 q.push(nxt[u]);
50                 vis[nxt[u]] = true;
51             }
52         }
53
54         int cycle_cnt = 0;
55         for (int i = indexed_from; i < n + indexed_from; i++)
56             if(!vis[i])
57                 compute_cycle(i, nxt, vis);
58
59         for (int i = (int)process_order.size() - 1; i >= 0; i--) {
60             int u = process_order[i];
61
62             this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
63             this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
64             this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
65         }
66
67         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>
81             &in_degree) {
82             this->allocate(n, indexed_from);
83             this->build(n, indexed_from, nxt, in_degree);
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
97 // cycle.
98 vector<int> cycle_id;
99 // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
100 // belong to any cycle.
101 vector<int> id_in_cycle;
102 // Represents the id of the nearest vertex present in a cycle.
103 vector<int> near_in_cycle;
104 // Represents the id of the nearest cycle.
105 vector<int> id_near_cycle;
106 // Distance to the nearest cycle.
107 vector<int> cycle_dist;
108 // Represent the id of the component of the vertex.
109 // Equal to id_near_cycle
110 vector<int> &comp = id_near_cycle;
111 };
112
113 class Functional_Graph {
114 // FOR UNDIRECTED GRAPH
115 private:
116 void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis,
117 vector<vector<int>>> &adj) {
118     int id_cycle = cycle_cnt++;
119     int cur_id = 0;
120     this->first[id_cycle] = u;
121
122     while(!vis[u]) {
123         vis[u] = true;
124
125         this->cycle[id_cycle].push_back(u);
126         nxt[u] = find_nxt(u, vis, adj);
127         if(nxt[u] == -1)
128             nxt[u] = this->first[id_cycle];
129
130         this->in_cycle[u] = true;
131         this->cycle_id[u] = id_cycle;
132         this->id_in_cycle[u] = cur_id;
133         this->near_in_cycle[u] = u;
134         this->id_near_cycle[u] = id_cycle;
135         this->cycle_dist[u] = 0;
136
137         u = nxt[u];
138         cur_id++;
139     }
140 }
141
142 int find_nxt(int u, vector<bool> &vis, vector<vector<int>>> &adj) {
143     for(int v: adj[u])
144         if(!vis[v])
145             return v;
146     return -1;
147 }
148
149 // Time Complexity: O(V + E)
150 void build(int n, int indexed_from, vector<int> &degree,
151 vector<vector<int>>> &adj) {
152     queue<int> q;
153     vector<bool> vis(n + indexed_from, false);

```

```

154 vector<int> nxt(n + indexed_from);
155 for(int i = indexed_from; i < n + indexed_from; i++) {
156     if(adj[i].size() == 1) {
157         q.push(i);
158         vis[i] = true;
159     }
160 }
161
162 vector<int> process_order;
163 process_order.reserve(n + indexed_from);
164 while(!q.empty()) {
165     int u = q.front();
166     q.pop();
167
168     process_order.push_back(u);
169
170     nxt[u] = find_nxt(u, vis, adj);
171     if(--degree[nxt[u]] == 1) {
172         q.push(nxt[u]);
173         vis[nxt[u]] = true;
174     }
175 }
176
177 int cycle_cnt = 0;
178 for(int i = indexed_from; i < n + indexed_from; i++)
179     if(!vis[i])
180         compute_cycle(i, nxt, vis, adj);
181
182 for(int i = (int)process_order.size() - 1; i >= 0; i--) {
183     int u = process_order[i];
184
185     this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
186     this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
187     this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
188 }
189
190 void allocate(int n, int indexed_from) {
191     this->cycle.resize(n + indexed_from);
192     this->first.resize(n + indexed_from);
193
194     this->in_cycle.resize(n + indexed_from, false);
195     this->cycle_id.resize(n + indexed_from, -1);
196     this->id_in_cycle.resize(n + indexed_from, -1);
197     this->near_in_cycle.resize(n + indexed_from);
198     this->id_near_cycle.resize(n + indexed_from);
199     this->cycle_dist.resize(n + indexed_from);
200 }
201
202 public:
203 Functional_Graph(int n, int indexed_from, vector<int> degree,
204 vector<vector<int>>> &adj) {
205     this->allocate(n, indexed_from);
206     this->build(n, indexed_from, degree, adj);
207 }
208
209 // THE CYCLES ARE ALWAYS INDEXED BY ZERO!
210
211 // number of cycles
212 int cycle_cnt = 0;
213 // Vertices present in the i-th cycle.
214 vector<vector<int>>> cycle;
215 // first vertex of the i-th cycle
216 vector<int> first;
217

```



```

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

```

### 5.15. 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,
19         const vector<vector<int>>& 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     int maxx = -1, idx = -1;
29     for (const int v : adj[u]) {
30         if (v == p)
31             continue;
32         if (sz[v] > maxx) {
33             maxx = sz[v];
34             idx = v;
35         }
36     }
37
38     if (idx != -1)
39         this->dfs(idx, id, u, adj);
40
41     for (const int v : adj[u]) {
42         if (v == idx || v == p)
43             continue;

```

```

44         this->dfs(v, this->number_of_chains++, u, adj);
45     }
46 }
47
48 public:
49     /// Builds the chains.
50     ///
51     /// Time Complexity: O(n)
52     HLD(const int root_idx, const vector<vector<int>>& adj) {
53         this->n = adj.size();
54         this->chain_head.resize(this->n + 1, -1);
55         this->id_in_chain.resize(this->n + 1, -1);
56         this->chain_id.resize(this->n + 1, -1);
57         this->sz.resize(this->n + 1);
58         this->chain_size.resize(this->n + 1);
59         this->parent.resize(this->n + 1, -1);
60         this->get_sz(root_idx, -1, adj);
61         this->dfs(root_idx, 0, -1, adj);
62     }
63
64     // the chains are indexed from 0
65     int number_of_chains = 1;
66     // topmost node of the chain
67     vector<int> chain_head;
68     // id of the i-th node in his chain
69     vector<int> id_in_chain;
70     // id of the chain that the i-th node belongs
71     vector<int> chain_id;
72     // size of the i-th chain
73     vector<int> chain_size;
74     // parent of the i-th node, -1 for root
75     vector<int> parent;
76 };

```

### 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     DSU dsu(n);
52
53     sort(edges.begin(), edges.end());
54
55     int weight = 0;
56     for(int i = 0; i < (int)edges.size(); i++) {
57         if(dsu.Union(edges[i].u, edges[i].v)) {
58             weight += edges[i].w;
59         }
60     }
61
62     // returns weight of mst
63     return weight;
64 }
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

```

```

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
73         this->dist[u] = d;
74     #endif
75
76     for(const pair<int, int> &x: adj[u]) {
77         int v = x.first, w = x.second;
78         if(v == p)
79             continue;
80         #ifdef COST
81             this->cost[v][0] = w;
82         #endif
83         this->dfs_LCA_weighted(v, u, l + 1, d + w, adj);
84     }
85 }
86
87 void build_unweighted(const vector<vector<int>> &adj) {
88     this->dfs_LCA_unweighted(this->indexed_from, -1, 1, 0, adj);
89 }
90

```

```

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     // go up k levels from x
106     int lca_go_up(int x, int k) {
107         for(int i = 0; k > 0; i++, k >>= 1)
108             if(k & 1) {
109                 x = this->anc[x][i];
110                 if(x == -1)
111                     return -1;
112             }
113
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
132             return ans;
133         }
134     #endif
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 1.
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];
200     }
201
202     /// Returns the LCA of a and b.
203     ///
204     /// Time Complexity: O(log(n))
205     int query_lca(const int a, const int b) {
206         assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n +
207 indexed_from);
208
209         return this->get_lca(a, b);
210     }
211
212     #ifdef DIST
213         /// Returns the distance from a to b. When the graph is unweighted, it is
214         /// considered
215         /// 1 as the weight of the edges.
216         ///

```

```

214 /// Time Complexity: O(log(n))
215 int query_dist(const int a, const int b) {
216     assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n +
        indexed_from);
217
218     return this->dist[a] + this->dist[b] - 2*this->dist[this->get_lca(a, b)];
219 }
220 #endif
221
222 #ifdef COST
223 /// Returns the max/min weight edge from a to b.
224 ///
225 /// Time Complexity: O(log(n))
226 int query_cost(const int a, const int b) {
227     assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n +
        indexed_from);
228
229     const int l = this->query_lca(a, b);
230     return combine(this->lca_query_cost_in_line(a, l),
        this->lca_query_cost_in_line(b, l));
231 }
232 #endif
233 };

```

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

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

5.19. Maximum Path Unweighted Graph

```

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

```

```

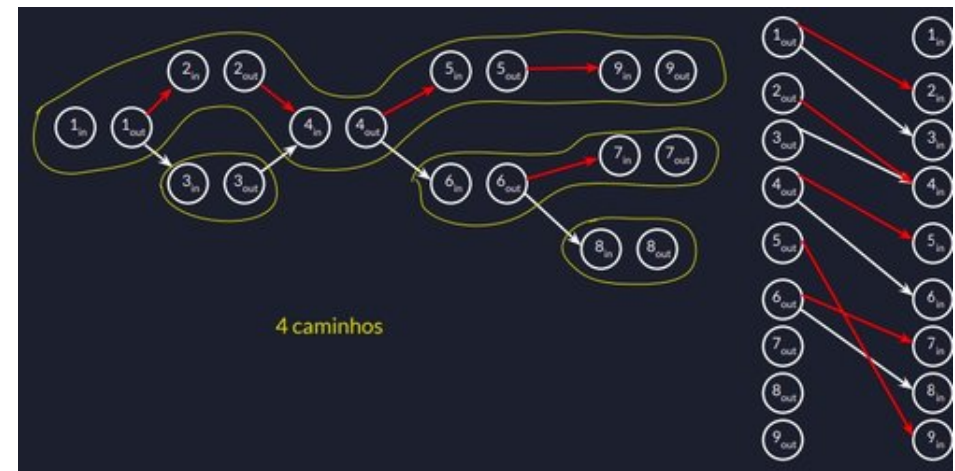
32 // cout << x + 1 << ' ';
33 // }
34 // cout << endl;
35
36 return dp[n - 1];
37 }

```

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

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

5.21. Minimum Path Cover In Dag



5.22. Minimum Path Cover In Dag

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

5.23. Number Of Different Spanning Trees In A Complete Graph

```

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

```

5.24. Number Of Ways To Make A Graph Connected

```

1 s_{1} * s_{2} * s_{3} * (...) * s_{k} * (n ^ (k - 2))
2 n = number of vertices

```

```

3 s_[i] = size of the i-th connected component
4 k = number of connected components

```

### 5.25. 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 }

```

### 5.26. 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 }

```

### 5.27. Pruffer Properties

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

### 5.28. 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.

### 5.29. 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 }

```

### 5.30. 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 }

```

### 5.31. 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. 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}

```

## 6.16. 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

```

## 6.17. 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

```

## 6.18. 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

```

## 6.19. 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 }

```

## 6.20. 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);

```

## 6.21. 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"

```

## 6.22. 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"

```

## 6.23. 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)

```

## 6.24. Readint

```

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

```

## 7. Math

### 7.1. Bell Numbers

```

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

```

### 7.2. Binary Exponentiation

```

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

```

### 7.3. Chinese Remainder Theorem

```

1 int inv(int a, int m) {
2     int m0 = m, t, q;

```



```

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

```

#### 7.4. Combinatorics

```

1  class Combinatorics {
2  private:
3      static constexpr int MOD = 1e9 + 7;
4      const int max_val;
5      vector<int> _inv, _fat;
6
7  private:
8      int mod(int x) {

```

```

9      x %= MOD;
10     if (x < 0)
11         x += MOD;
12     return x;
13 }
14
15 static int bin_pow(const int n, int p) {
16     assert(p >= 0);
17     int ans = 1;
18     int cur_pow = n;
19
20     while (p) {
21         if (p & 1ll)
22             ans = (ans * cur_pow) % MOD;
23
24         cur_pow = (cur_pow * cur_pow) % MOD;
25         p >>= 1ll;
26     }
27
28     return ans;
29 }
30
31 vector<int> build_inverse(const int max_val) {
32     vector<int> inv(max_val + 1);
33     inv[1] = 1;
34     for (int i = 2; i <= max_val; ++i)
35         inv[i] = mod(-MOD / i * inv[MOD % i]);
36     return inv;
37 }
38
39 vector<int> build_fat(const int max_val) {
40     vector<int> fat(max_val + 1);
41     fat[0] = 1;
42     for (int i = 1; i <= max_val; ++i)
43         fat[i] = mod(i * fat[i - 1]);
44     return fat;
45 }
46
47 public:
48     /// Builds both factorial and modular inverse array.
49     ///
50     /// Time Complexity: O(max_val)
51     Combinatorics(const int max_val) : max_val(max_val) {
52         assert(0 <= max_val), assert(max_val <= MOD);
53         this->_inv = this->build_inverse(max_val);
54         this->_fat = this->build_fat(max_val);
55     }
56
57     /// Returns the modular inverse of n % MOD.
58     ///
59     /// Time Complexity: O(log(MOD))
60     static int inv_log(const int n) { return bin_pow(n, MOD - 2); }
61
62     /// Returns the modular inverse of n % MOD.
63     ///
64     /// Time Complexity: O((n <= max_val ? 1 : log(MOD)))
65     int inv(const int n) {
66         assert(0 <= n);
67         if (n <= max_val)
68             return this->_inv[n];
69         else
70             return inv_log(n);
71     }
72
73     /// Returns the factorial of n % MOD.

```

```

74 int fat(const int n) {
75     assert(0 <= n), assert(n <= max_val);
76     return this->_fat[n];
77 }
78
79 /// Returns C(n, k) % MOD.
80 ///
81 /// Time Complexity: O(1)
82 int choose(const int n, const int k) {
83     assert(0 <= k), assert(k <= n), assert(n <= this->max_val);
84     return mod(fat(n) * mod(inv(fat(k)) * inv(fat(n - k))));
85 }
86 };

```

### 7.5. Diophantine Equation

```

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

```

### 7.6. Divisors

```

1 /// OBS: Each number has at most  $\sqrt[3]{N}$  divisors
2 /// THE NUMBERS ARE NOT SORTED!!!
3 vector<int> divisors(int n) {
4     vector<int> ans;
5     for (int i = 1; i * i <= n; i++) {
6         if (n % i == 0) {
7             // If divisors are equal, print only one
8             if (n / i == i)
9                 ans.emplace_back(i);
10            else // Otherwise print both
11                ans.emplace_back(i), ans.emplace_back(n / i);
12        }
13    }
14    return ans;
15 }

```

### 7.7. Euler Totient

```

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

```

### 7.8. Extended Euclidean

```

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

```

### 7.9. Factorization

```

1 map<int, int> primeFactors(int n) {
2     set<int> ret;
3     while (n % 2 == 0) {
4         ++m[2];
5         n /= 2;
6     }
7
8     for (int i = 3; i * i <= n; i += 2) {
9         while (n % i == 0) {
10            m[i]++;
11            n = n / i;
12        }
13        /* OBS1
14           IF (N < 1E7)
15              you can optimize by factoring with SPF
16        */
17    }
18
19    if (n > 2)
20        ++m[n];
21
22    return ret;
23 }

```

## 7.10. Inclusion Exclusion

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

## 7.11. Inclusion Exclusion

```

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

```

## 7.12. 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

```

## 7.13. Pollard Rho (Find A Divisor)

```

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

```

```

33
34  /* Hare Move: y(i+1) = f(f(y(i))) */
35  y = (modular_pow(y, 2, n) + c + n) % n;
36  y = (modular_pow(y, 2, n) + c + n) % n;
37
38  d = __gcd(abs(x - y), n);
39
40  /* retry if the algorithm fails to find prime factor
41   * with chosen x and c */
42  if (d == n)
43      return pollard_rho(n);
44  }
45
46  return d;
47 }

```

#### 7.14. Primality Check

```

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

```

#### 7.15. Sieve + Segmented Sieve

```

1 const int MAXN = 1e6;
2
3 // Contains all the primes in the segments
4 vector<int> segPrimes;
5 bitset<MAXN + 5> primesInSeg;
6
7 // smallest prime factor
8 int spf[MAXN + 5];
9
10 vector<int> primes;
11 bitset<MAXN + 5> isPrime;
12
13 void sieve(int n = MAXN + 2) {
14
15     for (int i = 0; i <= n; i++)
16         spf[i] = i;
17
18     isPrime.set();
19     for (int i = 2; i <= n; i++) {
20         if (!isPrime[i])
21             continue;
22
23         for (int j = i * i; j <= n; j += i) {
24             isPrime[j] = false;
25             spf[j] = min(i, spf[j]);
26         }
27         primes.emplace_back(i);
28     }
29 }

```

```

29 }
30
31 vector<int> getFactorization(int x) {
32     vector<int> ret;
33     while (x != 1) {
34         ret.emplace_back(spf[x]);
35         x = x / spf[x];
36     }
37     return ret;
38 }
39
40 /// Gets all primes from l to r
41 void segSieve(int l, int r) {
42     // primes from l to r
43     // transferred to 0..(l-r)
44     segPrimes.clear();
45     primesInSeg.set();
46     int sq = sqrt(r) + 5;
47
48     for (int p : primes) {
49         if (p > sq)
50             break;
51
52         for (int i = l - l % p; i <= r; i += p) {
53             if (i - l < 0)
54                 continue;
55
56             // if i is less than 1e6, it could be checked in the
57             // array of the sieve
58             if (i >= (int)1e6 || !isPrime[i])
59                 primesInSeg[i - l] = false;
60         }
61     }
62
63     for (int i = 0; i < r - l + 1; i++) {
64         if (primesInSeg[i])
65             segPrimes.emplace_back(i + l);
66     }
67 }

```

## 8. Miscellaneous

### 8.1. 2-Sat

```

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

```

```

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

```

## 8.2. 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 }

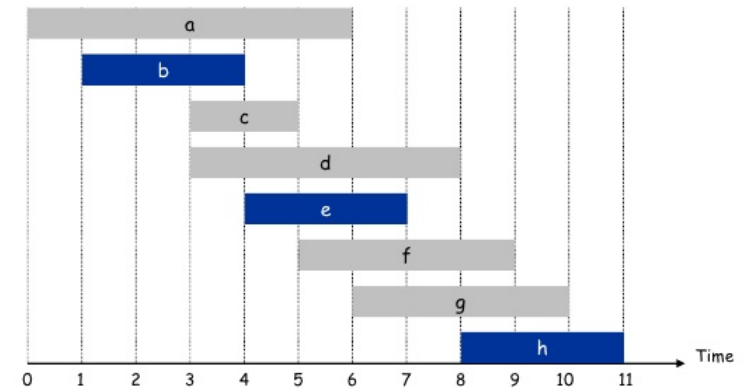
```

```

35 }
36 while(!op.empty()) {
37     postfix += op.top();
38     op.pop();
39 }
40 cout << postfix << endl;
41 }

```

## 8.3. Interval Scheduling



## 8.4. 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.

## 8.5. 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 }

```

## 8.6. Kadane 2D

```

1 // Program to find maximum sum subarray in a given 2D array
2 #include <stdio.h>
3 #include <string.h>
4 #include <limits.h>
5

```

```

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

```

## 8.7. 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 }

```

### 8.8. Sliding Window Minimum

```

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

```

### 8.9. 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 }

```

### 8.10. Kadane (Segment Tree)

```

1 struct Node {
2     int pref, suf, tot, best;
3     Node () {}
4     Node(int pref, int suf, int tot, int best) : pref(pref), suf(suf),
5         tot(tot), best(best) {}
6 };
7
8 const int MAXN = 2E5 + 10;
9 Node tree[5*MAXN];
10 int arr[MAXN];
11
12 Node query(const int l, const int r, const int i, const int j, const int
13 pos) {
14     if(l > r || l > j || r < i)

```

```

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

```

### 8.11. Largest Area In Histogram

```

1 // Time Complexity: O(n)
2 int largest_area_in_histogram(vector<int> &arr) {
3     arr.emplace_back(0);
4
5     stack<int> s;
6     int ans = 0;

```

```

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

```

### 8.12. 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 }

```

### 8.13. Ternary Search

```

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

```

## 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. 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

```

### 9.3. 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.

```

### 9.4. 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 }

```

### 9.5. 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     }

```

```

76     }
77 }
78
79 vector<vector<pair<int, int>>> aho_find_occurrences(const string &text) {
80     vector<vector<pair<int, int>>> ans(arr.size());
81     Trie::Node *cur = trie.root();
82
83     for (int i = 0; i < text.size(); ++i) {
84         cur = find_fail_node(cur, text[i]);
85         for (Out_Node *node = out_node[cur->id]; node != nullptr;
86              node = node->next)
87             for (const int idx : node->str_idx)
88                 ans[idx].emplace_back(i - (int)arr[idx].size() + 1, i);
89     }
90     return ans;
91 }
92
93 public:
94     /// Constructor that builds the trie and the failures.
95     ///
96     /// Time Complexity: O(number of characters in arr)
97     Aho(const vector<string> &arr) : arr(arr) { this->build_trie(); }
98
99     /// Searches in text for all occurrences of all strings in array arr.
100    ///
101    /// Time Complexity: O(text.size() + number of characters in arr)
102    vector<vector<pair<int, int>>> find_occurrences(const string &text) {
103        return this->aho_find_occurrences(text);
104    }
105 };

```

## 9.6. 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 = {10000000007, 10000000009};
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                     l]),
43                     m[i]);
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     }
87 };

```

## 9.7. Lcs K Strings

```

1 // Make the change below in SuffixArray code.
2 int MaximumNumberOfStrings;
3

```

```

4 void build_suffix_array() {
5     vector<pair<Rank, int>> ranks(this->n + 1);
6     vector<int> arr;
7
8     for (int i = 1, separators = 0; i <= n; i++)
9         if(this->s[i] > 0) {
10             ranks[i] = pair<Rank, int>(Rank((int)this->s[i] +
11                 MaximumNumberOfStrings, 0), i);
12             this->s[i] += MaximumNumberOfStrings;
13         } else {
14             ranks[i] = pair<Rank, int>(Rank(separators, 0), i);
15             this->s[i] = separators;
16             separators++;
17         }
18     RadixSort::sort_pairs(ranks, 256 + MaximumNumberOfStrings);
19     ...
20 }
21
22 /// Program to find the LCS between k different strings.
23 ///
24 /// Time Complexity: O(n*log(n))
25 /// Space Complexity: O(n*log(n))
26 int main() {
27     int n;
28
29     cin >> n;
30
31     MaximumNumberOfStrings = n;
32
33     vector<string> arr(n);
34
35     int sum = 0;
36     for(string &x: arr) {
37         cin >> x;
38         sum += x.size() + 1;
39     }
40
41     string concat;
42     vector<int> ind(sum + 1);
43     int cnt = 0;
44     for(string &x: arr) {
45         if(concat.size())
46             concat += (char)cnt;
47         concat += x;
48     }
49
50     cnt = 0;
51     for(int i = 0; i < concat.size(); i++) {
52         ind[i + 1] = cnt;
53         if(concat[i] < MaximumNumberOfStrings)
54             cnt++;
55     }
56
57     Suffix_Array say(concat);
58     vector<int> sa = say.get_suffix_array();
59     Sparse_Table spt(say.get_lcp());
60
61     vector<int> freq(n);
62     int cnt1 = 0;
63
64     /// Ignore separators
65     int i = n, j = n - 1;
66     int ans = 0;
67

```

```

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

```

## 9.8. Lexicographically Smallest Rotation

```

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

```

## 9.9. Manacher (Longest Palindrome)

```

1 // 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 }

```

```

9     }
10    ret.push_back('#');
11    return ret;
12 }
13
14 /// Returns the first occurrence of the longest palindrome based on the lps
15 array.
16 /// Time Complexity: O(n)
17 string get_best(const int max_len, const string &str, const vector<int>
18 &lps) {
19     for(int i = 0; i < lps.size(); i++) {
20         if(lps[i] == max_len) {
21             string ans;
22             int cnt = max_len / 2;
23             int io = i - 1;
24             while(cnt) {
25                 if(str[io] != '#') {
26                     ans += str[io];
27                     cnt--;
28                 }
29                 io--;
30             }
31             reverse(ans.begin(), ans.end());
32             if(str[i] != '#')
33                 ans += str[i];
34             cnt = max_len / 2;
35             io = i + 1;
36             while(cnt) {
37                 if(str[io] != '#') {
38                     ans += str[io];
39                     cnt--;
40                 }
41                 io++;
42             }
43             return ans;
44         }
45     }
46 }
47 /// Returns a pair containing the size of the longest palindrome and the
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;
62     for(int i = 0; i < len; i++) {
63         //get mirror index of i
64         int mirror = (2 * c) - i;
65
66         //see if the mirror of i is expanding beyond the left boundary of
67         current longest palindrome at center c
68         //if it is, then take r - i as lps[i]
69         //else take lps[mirror] as lps[i]
70         if(i < r)

```

```

67     lps[i] = min(r - i, lps[mirror]);
68
69     //expand at i
70     int a = i + (1 + lps[i]);
71     int b = i - (1 + lps[i]);
72     while(a < len && b >= 0 && str[a] == str[b]) {
73         lps[i]++;
74         a++;
75         b--;
76     }
77
78     //check if the expanded palindrome at i is expanding beyond the right
79     boundary of current longest palindrome at center c
80     //if it is, the new center is i
81     if(i + lps[i] > r) {
82         c = i;
83         r = i + lps[i];
84
85         if(lps[i] > max_len) //update max_len
86             max_len = lps[i];
87     }
88
89     return make_pair(max_len, get_best(max_len, str, lps));
90 }

```

## 9.10. Suffix Array

```

1 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
157         while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
158             k++;
159
160         lcp[inverse_suffix[i]] = k;
161
162         if (k > 0)
163             k--;
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     /// 'S' + 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 };

```

### 9.11. Suffix Array Pessoa

```

1  // OBS: Suffix Array build code imported from:
2  //
3  // https://github.com/gabrielpessoal/Biblioteca-Maratona/blob/master/code/Strings/SuffixArray.cpp
4  // Because it's faster.
5
6  /// It is indexed by 0.
7  /// Let the given string be "banana".
8  ///
9  /// 0 banana          5 a
10  /// 1 anana           Sort the Suffixes 3 ana
11  /// 2 nana            ----->         1 anana
12  /// 3 ana             alphabetically    0 banana
13  /// 4 na              4 na
14  /// 5 a               2 nana
15  /// So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
16  ///
17  /// LCP
18  ///
19  /// 1 a
20  /// 3 ana
21  /// 0 anana
22  /// 0 banana
23  /// 2 na
24  /// 0 nana (The last position will always be zero)
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];
144             int right = this->sa[i + 1];
145
146             if ((left < separator && right > separator) ||
147                 (left > separator && right < separator))
148                 ans = max(ans, lcp[i]);
149         }
150
151         return ans;
152     }

```

```

153 }
154 };

```

## 9.12. 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; }, 111);
32
33        // Sort by the first rank
34        RadixSort::sort<T>(
35            arr, rank_size, [](T &item) { return item.first.first; }, 111);
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
140     this->suffix_array.resize(this->n + 1);
141     for (int i = 1; i <= this->n; i++)
142         this->suffix_array[arr[i]] = i;
143
144     }
145
146     /// Builds the lcp (Longest Common Prefix) array for the string s.
147     /// A value lcp[i] indicates length of the longest common prefix of the
148     /// suffixes indexed by i and i + 1. Implementation of the Kasai's
149     /// Algorithm.
150     ///
151     /// Time Complexity: O(n)
152     /// Space Complexity: O(n)
153     vector<int> build_lcp() {
154         vector<int> lcp(this->n + 1, 0);
155         vector<int> inverse_suffix(this->n + 1, 0);
156
157         for (int i = 1; i <= n; i++)
158             inverse_suffix[suffix_array[i]] = i;
159
160         int k = 0;
161
162         for (int i = 1; i <= n; i++) {
163             if (inverse_suffix[i] == n) {
164                 k = 0;
165                 continue;
166             }
167
168             int j = suffix_array[inverse_suffix[i] + 1];
169
170             while (i + k <= this->n && j + k <= this->n && s[i + k] == s[j + k])
171                 k++;
172
173             lcp[inverse_suffix[i]] = k;
174
175             if (k > 0)
176                 k--;
177         }
178
179         return lcp;
180     }
181
182     void build_log_array() {
183         log_array.resize(this->n + 1, 0);
184
185         for (int i = 2; i <= this->n; i++)
186             log_array[i] = log_array[i / 2] + 1;
187     }
188
189 public:
190     const vector<int> &get_suffix_array() { return suffix_array; }

```



```

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 /// 's'
195 /// + B).
196 ///
197 /// The string A starts at index 1 and ends at index (separator - 1).
198 /// The string B starts at index (separator + 1) and ends at the end of the
199 /// string.
200 ///
201 /// Time Complexity: O(n)
202 /// Space Complexity: O(1)
203 int lcs(int separator) {
204     separator++;
205     assert(!isalpha(this->s[separator] && !isdigit(this->s[separator]]));
206
207     int ans = 0;
208
209     for (int i = 1; i < this->n - 1; i++) {
210         int left = this->suffix_array[i];
211         int right = this->suffix_array[i + 1];
212
213         if ((left < separator && right > separator) ||
214             (left > separator && right < separator))
215             ans = max(ans, lcp[i]);
216     }
217
218     return ans;
219 }
220
221 /// Compares two substrings beginning at indexes i and j of a fixed length.
222 ///
223 /// OBS: Necessary build rank_table (uncomment build_suffix_array) and
224 /// build
225 /// log_array.
226 ///
227 /// Time Complexity: O(1)
228 /// Space Complexity: O(1)
229 int compare(const int i, const int j, const int length) {
230     assert(1 <= i && i <= this->n && 1 <= j && j <= this->n);
231     assert(!this->log_array.empty() && !this->rank_table.empty());
232     assert(i + length - 1 <= this->n && j + length - 1 <= this->n);
233
234     // Greatest k such that 2^k <= l
235     const int k = this->log_array[length];
236
237     const int jump = length - (1 << k);
238
239     const pair<int, int> iRank = {
240         this->rank_table[k][i],
241         (i + jump <= this->n ? this->rank_table[k][i + jump] : -1)};
242     const pair<int, int> jRank = {
243         this->rank_table[k][j],
244         (j + jump <= this->n ? this->rank_table[k][j + jump] : -1)};
245
246     return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
247 }
248 };

```

### 9.13. Trie

```

1 class Trie {

```

```

2 private:
3     static const int INT_LEN = 31;
4     // static const int INT_LEN = 63;
5
6 public:
7     struct Node {
8         map<char, Node *> next;
9         int id;
10        // cnt counts the number of words which pass in that node
11        int cnt = 0;
12        // word counts the number of words ending at that node
13        int word_cnt = 0;
14
15        Node(const int x) : id(x) {}
16    };
17
18 private:
19     int trie_size = 0;
20     // contains the next id to be used in a node
21     int node_cnt = 0;
22     Node *trie_root = this->make_node();
23
24 private:
25     Node *make_node() { return new Node(node_cnt++); }
26
27     int trie_insert(const string &s) {
28         Node *aux = this->root();
29         for (const char c : s) {
30             if (!aux->next.count(c))
31                 aux->next[c] = this->make_node();
32             aux = aux->next[c];
33             ++aux->cnt;
34         }
35         ++aux->word_cnt;
36         ++this->trie_size;
37         return aux->id;
38     }
39
40     void trie_erase(const string &s) {
41         Node *aux = this->root();
42         for (const char c : s) {
43             Node *last = aux;
44             aux = aux->next[c];
45             --aux->cnt;
46             if (aux->cnt == 0) {
47                 last->next.erase(c);
48                 aux = nullptr;
49                 break;
50             }
51         }
52         if (aux != nullptr)
53             --aux->word_cnt;
54         --this->trie_size;
55     }
56
57     int trie_count(const string &s) {
58         Node *aux = this->root();
59         for (const char c : s) {
60             if (aux->next.count(c))
61                 aux = aux->next[c];
62             else
63                 return 0;
64         }
65         return aux->word_cnt;
66     }

```

```

67
68 int trie_query_xor_max(const string &s) {
69     Node *aux = this->root();
70     int ans = 0;
71     for (const char c : s) {
72         const char inv = (c == '0' ? '1' : '0');
73         if (aux->next.count(inv)) {
74             ans = (ans << 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 };

```