C++ Competitive Programming Library

DO NOT DISCLOSE OR DISTRIBUTE

bfs.07 - Bernardo Flores Salmeron

1. Template

```
#include <bits/stdc++.h>

using namespace std;

#define INF (1ll << 62)
#define pb push_back
#define ii pair<int,int>
#define OK cerr <<"OK"<< endl
#define debug(x) cerr << #x " = " << (x) << endl
#define ff first
#define ss second
#define int long long
#define tt tuple<int, int, int>
#define endl '\n'

signed main () {

  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

}
```

2. Data Structures

2.1. Bit2D

```
// INDEX BY ONE ALWAYS!!!
class BIT_2D {
 private:
  // row, column
  int n, m;
  vector<vector<int>> tree;

 private:
  // Returns an integer which constains only the least significant bit.
  int low(int i) {
    return i & (-i);
  }

  void bit_update(const int x, const int y, const int delta) {
    for(int i = x; i < n; i += low(i))
      for(int j = y; j < m; j += low(j))
        this->tree[i][j] += delta;
  }

  int bit_query(const int x, const int y) {
    int ans = 0;
    for(int i = x; i > 0; i -= low(i))
      for(int j = y; j > 0; j -= low(j))
        ans += this->tree[i][j];

    return ans;
  }

 public:
  // put the size of the array without 1 indexing.
  /// Time Complexity: O(n * m)
  BIT_2D(int n, int m) {
    this->n = n + 1;
    this->m = m + 1;

    this->tree.resize(n, vector<int>(m, 0));
```

```
  }

  /// Time Complexity: O(n * m * (log(n) + log(m)))
  BIT_2D(const vector<vector<int>> &mat) {
    // Check if it is 1 index.
    assert(mat[0][0] == 0);
    this->n = mat.size();
    this->m = mat.front().size();

    this->tree.resize(n, vector<int>(m, 0));
    for(int i = 1; i < n; i++)
      for(int j = 1; j < m; j++)
        update(i, j, mat[i][j]);
  }

  /// Query from (1, 1) to (x, y).
  ///
  /// Time Complexity: O(log(n) + log(m))
  int prefix_query(const int x, const int y) {
    assert(0 < x); assert(x < this->n);
    assert(0 < y); assert(y < this->m);

    return bit_query(x, y);
  }

  /// Query from (x1, y1) to (x2, y2).
  ///
  /// Time Complexity: O(log(n) + log(m))
  int query(const int x1, const int y1, const int x2, const int y2) {
    assert(0 < x1); assert(x1 <= x2); assert(x2 < this->n);
    assert(0 < y1); assert(y1 <= y2); assert(y2 < this->m);

    return bit_query(x2, y2) - bit_query(x1 - 1, y2) - bit_query(x2, y1 - 1)
    + bit_query(x1 - 1, y1 - 1);
  }

  /// Updates point (x, y).
  ///
  /// Time Complexity: O(log(n) + log(m))
  void update(const int x, const int y, const int delta) {
    assert(0 < x); assert(x < this->n);
    assert(0 < y); assert(y < this->m);

    bit_update(x, y, delta);
  }
};
```

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

```
// retornar a qtd de números maiores q um numero k numa array de i...j
struct Tree {
  vector<int> vet;
};
Tree tree[4*(int)3e4];
int arr[(int)5e4];

int query(int l,int r, int i, int j, int k, int pos) {
  if(l > j || r < i)
    return 0;

  if(i <= l && r <= j) {
    auto it = upper_bound(tree[pos].vet.begin(), tree[pos].vet.end(), k);
    return tree[pos].vet.end()-it;
  }

  int mid = (l+r)>>1;
  return query(l, mid, i, j, k, 2*pos+1) + query(mid+1,r,i,j,k,2*pos+2);
}

void build(int l, int r, int pos) {

  if(l == r) {
    tree[pos].vet.pb(arr[l]);
    return;
  }

  int mid = (l+r)>>1;
  build(l, mid, 2*pos+1);
  build(mid + 1, r, 2*pos+2);

  merge(tree[2*pos+1].vet.begin(), tree[2*pos+1].vet.end(),
    tree[2*pos+2].vet.begin(), tree[2*pos+2].vet.end(),
    back_inserter(tree[pos].vet));
}
```

2.3. Mos Algorithm

```
struct Tree {
  int l, r, ind;
};
Tree query[311111];
int arr[311111];
int freq[1111111];
int ans[311111];
int block = sqrt(n), cont = 0;

bool cmp(Tree a, Tree b) {
  if(a.l/block == b.l/block)
    return a.r < b.r;
  return a.l/block < b.l/block;
}

void add(int pos) {
  freq[arr[pos]]++;
  if(freq[arr[pos]] == 1) {
    cont++;
  }
}
void del(int pos) {
  freq[arr[pos]]--;
  if(freq[arr[pos]] == 0)
    cont--;
}
int main () {
  int n; cin >> n;
  block = sqrt(n);

  for(int i = 0; i < n; i++) {
    cin >> arr[i];
    freq[arr[i]] = 0;
  }

  int m;  cin >> m;

  for(int i = 0; i < m; i++) {
    cin >> query[i].l >> query[i].r;
    query[i].l--, query[i].r--;
```

```
    query[i].ind = i;
  }
  sort(query, query + m, cmp);

  int s,e;
  s = e = query[0].l;
  add(s);
  for(int i = 0; i < m; i++) {
    while(s > query[i].l)
      add(--s);
    while(s < query[i].l)
      del(s++);
    while(e < query[i].r)
      add(++e);
    while(e > query[i].r)
      del(e--);
    ans[query[i].ind] = cont;

  }
  for(int i = 0; i < m; i++)
    cout << ans[i] << endl;
}
```

2.4. Sqrt Decomposition

```
// Problem: Sum from l to r
// Ver MO'S ALGORITHM
// -------------------------------------
int getId(int indx,int blockSZ) {
    return indx/blockSZ;
}
void init(int sz) {
    for(int i=0; i<=sz; i++)
  BLOCK[i]=inf;
}
int query(int left, int right) {
int startBlockIndex=left/sqrt;
int endIBlockIndex = right / sqrt;
int sum = 0;
for (int i = startBlockIndex + 1; i < endIBlockIndex; i++) {
        sum += blockSums[i];
    }
for(i=left...(startBlockIndex*BLOCK_SIZE-1))
  sum += a[i];
for(j = endIBlockIndex*BLOCK_SIZE ... right)
  sum += a[i];
}
```

2.5. Bit

```
/// INDEX THE ARRAY BY 1!!!
class BIT {
private:
  vector<int> bit;
  int n;

private:
  int low(const int i) { return (i & (-i)); }

  // point update
  void bit_update(int i, const int delta) {
    while (i <= this->n) {
      this->bit[i] += delta;
      i += this->low(i);
    }
  }

  // point query
  int bit_query(int i) {
    int sum = 0;
    while (i > 0) {
      sum += bit[i];
      i -= this->low(i);
    }
    return sum;
  }

public:
  BIT(const vector<int> &arr) { this->build(arr); }

  BIT(const int n) {
    // OBS: BIT IS INDEXED FROM 1
    // THE USE OF 1-BASED ARRAY IS RECOMMENDED
    this->n = n;
    this->bit.resize(n + 1, 0);
  }

  // build the bit
  void build(const vector<int> &arr) {
    // OBS: BIT IS INDEXED FROM 1
    // THE USE OF 1-BASED ARRAY IS RECOMMENDED
    assert(arr.front() == 0);
    this->n = (int)arr.size() - 1;
    this->bit.resize(arr.size(), 0);

    for (int i = 1; i <= this->n; i++)
      this->bit_update(i, arr[i]);
  }

  // point update
  void update(const int i, const int delta) {
    assert(1 <= i), assert(i <= this->n);
    this->bit_update(i, delta);
  }

  // point query
  int query(const int i) {
    assert(1 <= i), assert(i <= this->n);
    return this->bit_query(i);
  }

  // range query
  int query(const int l, const int r) {
    assert(1 <= l), assert(l <= r), assert(r <= this->n);
    return this->bit_query(r) - this->bit_query(l - 1);
  }
};
```

2.6. Bit (Range Update)

```
/// INDEX THE ARRAY BY 1!!!
class BIT {
private:
  vector<int> bit1;
  vector<int> bit2;
  int n;

```

```cpp
private:
  int low(int i) { return (i & (-i)); }

  // point update
  void update(int i, const int delta, vector<int> &bit) {
    while (i <= this->n) {
      bit[i] += delta;
      i += this->low(i);
    }
  }

  // point query
  int query(int i, const vector<int> &bit) {
    int sum = 0;
    while (i > 0) {
      sum += bit[i];
      i -= this->low(i);
    }
    return sum;
  }

  // build the bit
  void build(const vector<int> &arr) {
    // OBS: BIT IS INDEXED FROM 1
    // THE USE OF 1-BASED ARRAY IS MANDATORY
    assert(arr.front() == 0);
    this->n = (int)arr.size() - 1;
    this->bit1.resize(arr.size(), 0);
    this->bit2.resize(arr.size(), 0);

    for (int i = 1; i <= this->n; i++)
      this->update(i, arr[i]);
  }

public:
  BIT(const vector<int> &arr) { this->build(arr); }

  BIT(const int n) {
    // OBS: BIT IS INDEXED FROM 1
    // THE USAGE OF 1-INDEXED ARRAY IS MANDATORY
    this->n = n;
    this->bit1.resize(n + 1, 0);
    this->bit2.resize(n + 1, 0);
  }

  // range update
  void update(const int l, const int r, const int delta) {
    assert(1 <= l), assert(l <= r), assert(r <= this->n);
    this->update(l, delta, this->bit1);
    this->update(r + 1, -delta, this->bit1);
    this->update(l, delta * (l - 1), this->bit2);
    this->update(r + 1, -delta * r, this->bit2);
  }

  // point update
  void update(const int i, const int delta) {
    assert(1 <= i), assert(i <= this->n);
    this->update(i, i, delta);
  }

  // range query
  int query(const int l, const int r) {
    assert(1 <= l), assert(l <= r), assert(r <= this->n);
    return this->query(r) - this->query(l - 1);
  }

  // point prefix query
  int query(const int i) {
    assert(i <= this->n);
    return (this->query(i, this->bit1) * i) - this->query(i, this->bit2);
  }
};

// TESTS
// signed main()
// {

//  vector<int> input = {0,1,2,3,4,5,6,7};

//  BIT ft(input);

//  assert (1 == ft.query(1));
//  assert (3 == ft.query(2));
//  assert (6 == ft.query(3));
//  assert (10 == ft.query(4));
//  assert (15 == ft.query(5));
//  assert (21 == ft.query(6));
//  assert (28 == ft.query(7));
//  assert (12 == ft.query(3,5));
//  assert (21 == ft.query(1,6));
//  assert (28 == ft.query(1,7));
// }
```

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

```cpp
// REQUIRES bit.cpp!!
// REQUIRES point_compresion.cpp!!
int count_inversions(vector<int> &arr) {
  arr = compress(arr);
  int ans = 0;
  BIT bit(arr.size());
  for (int i = arr.size() - 1; i > 0; --i) {
    ans += bit.query(arr[i] - 1);
    bit.update(arr[i], 1);
  }
  return ans;
}
```

2.8. Ordered Set

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

template <typename T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag,
     tree_order_statistics_node_update>;

ordered_set<int> X;
X.insert(1);
X.insert(2);
X.insert(4);
X.insert(8);
```

```cpp
X.insert(16);

// 1, 2, 4, 8, 16
// returns the k-th greatest element from 0
cout << *X.find_by_order(1) << endl;            // 2
cout << *X.find_by_order(2) << endl;            // 4
cout << *X.find_by_order(4) << endl;            // 16
cout << (end(X) == X.find_by_order(6)) << endl; // true

// returns the number of items strictly less than a number
cout << X.order_of_key(-5) << endl;  // 0
cout << X.order_of_key(1) << endl;   // 0
cout << X.order_of_key(3) << endl;   // 2
cout << X.order_of_key(4) << endl;   // 2
cout << X.order_of_key(400) << endl; // 5
```

2.9. Persistent Segment Tree

```cpp
class Persistent_Seg_Tree {
  struct Node {
    int val;
    Node *left, *right;
    Node(const int v) : val(v), left(nullptr), right(nullptr) {}
  };

private:
  const Node NEUTRAL_NODE = Node(0);
  int merge_nodes(const int x, const int y) { return x + y; }

private:
  const int n;
  vector<Node *> version = {nullptr};

public:
  /// Builds version[0] with the values in the array.
  ///
  /// Time complexity: O(n)
  Node *build(Node *node, const int l, const int r, const vector<int> &arr) {
    node = new Node(NEUTRAL_NODE);
    if (l == r) {
      node->val = arr[l];
      return node;
    }

    const int mid = (l + r) / 2;
    node->left = build(node->left, l, mid, arr);
    node->right = build(node->right, mid + 1, r, arr);
    node->val = merge_nodes(node->left->val, node->right->val);
    return node;
  }

  Node *_update(Node *cur_tree, Node *prev_tree, const int l, const int r,
                const int idx, const int delta) {
    if (l > idx || r < idx)
      return cur_tree != nullptr ? cur_tree : prev_tree;

    if (cur_tree == nullptr && prev_tree == nullptr)
      cur_tree = new Node(NEUTRAL_NODE);
    else
      cur_tree = new Node(cur_tree == nullptr ? *prev_tree : *cur_tree);

    if (l == r) {
      cur_tree->val += delta;
      return cur_tree;
    }

    const int mid = (l + r) / 2;
    cur_tree->left =
        _update(cur_tree->left, prev_tree ? prev_tree->left : nullptr, l, mid,
                idx, delta);
    cur_tree->right =
        _update(cur_tree->right, prev_tree ? prev_tree->right : nullptr,
                mid + 1, r, idx, delta);
    cur_tree->val =
        merge_nodes(cur_tree->left ? cur_tree->left->val : NEUTRAL_NODE.val,
                    cur_tree->right ? cur_tree->right->val : NEUTRAL_NODE.val);
    return cur_tree;
  }

  int _query(Node *node, const int l, const int r, const int i, const int j) {
    if (node == nullptr || l > j || r < i)
      return NEUTRAL_NODE.val;

    if (i <= l && r <= j)
      return node->val;

    int mid = (l + r) / 2;
    return merge_nodes(_query(node->left, l, mid, i, j),
                       _query(node->right, mid + 1, r, i, j));
  }

  void create_version(const int v) {
    if (v >= this->version.size())
      version.resize(v + 1);
  }

public:
  Persistent_Seg_Tree() : n(-1) {}

  /// Constructor that initializes the segment tree empty. It’s allowed to query
  /// from 0 to MAXN - 1.
  ///
  /// Time Complexity: O(1)
  Persistent_Seg_Tree(const int MAXN) : n(MAXN) {}

  /// Constructor that allows to pass initial values to the leafs. It’s allowed
  /// to query from 0 to n - 1.
  ///
  /// Time Complexity: O(n)
  Persistent_Seg_Tree(const vector<int> &arr) : n(arr.size()) {
    this->version[0] = this->build(this->version[0], 0, this->n - 1, arr);
  }

  /// Links the root of a version to a previous version.
  ///
  /// Time Complexity: O(1)
  void link(const int version, const int prev_version) {
    assert(this->n > -1);
    assert(0 <= prev_version), assert(prev_version <= version);
    this->create_version(version);
    this->version[version] = this->version[prev_version];
  }

  /// Updates an index in cur_tree based on prev_tree with a delta.
```

```
  ///
  /// Time Complexity: O(log(n))
  void update(const int cur_version, const int prev_version, const int idx,
              const int delta) {
    assert(this->n > -1);
    assert(0 <= prev_version), assert(prev_version <= cur_version);
    this->create_version(cur_version);
    this->version[cur_version] =
        this->_update(this->version[cur_version],
    this->version[prev_version],
                      0, this->n - 1, idx, delta);
  }

  /// Query from l to r.
  ///
  /// Time Complexity: O(log(n))
  int query(const int version, const int l, const int r) {
    assert(this->n > -1);
    assert(0 <= l), assert(l <= r), assert(r < this->n);
    return this->_query(this->version[version], 0, this->n - 1, l, r);
  }
};
```

2.10. Segment Tree

```
class Seg_Tree {
public:
  struct Node {
    int val, lazy;

    Node() {}
    Node(const int val) : val(val), lazy(0) {}
  };

private:
  // // Range Sum
  // Node NEUTRAL_NODE = Node(0);
  // Node merge_nodes(const Node &x, const Node &y) {
  //   return Node(x.val + y.val);
  //   ;
  // }
  // void apply_lazy(const int l, const int r, const int pos) {
  //   // for set change this to =
  //   tree[pos].val += (r - l + 1) * tree[pos].lazy;
  // }

  // // RMQ Max
  // Node NEUTRAL_NODE = Node(-INF);
  // Node merge_nodes(const Node &x, const Node &y) {
  //   return Node(max(x.val, y.val));
  // }
  // void apply_lazy(const int l, const int r, const int pos) {
  //   tree[pos].val += tree[pos].lazy;
  // }

  // // RMQ Min
  // Node NEUTRAL_NODE = Node(INF);
  // Node merge_nodes(const Node &x, const Node &y) {
  //   return Node(min(x.val, y.val));
  // }
  // void apply_lazy(const int l, const int r, const int pos) {
  //   tree[pos].val += tree[pos].lazy;
  // }

  // // XOR
  // // Only works with point updates
  // Node NEUTRAL_NODE = Node(0);
  // Node merge_nodes(const Node &x, const Node &y) {
  //   return Node(x.val ^ y.val);
  //   ;
  // }
  // void apply_lazy(const int l, const int r, const int pos) {}

private:
  int n;

public:
  vector<Node> tree;

private:
  void propagate(const int l, const int r, const int pos) {
    if (tree[pos].lazy != 0) {
      apply_lazy(l, r, pos);
      if (l != r) {
        // for set change this to =
        tree[2 * pos + 1].lazy += tree[pos].lazy;
        tree[2 * pos + 2].lazy += tree[pos].lazy;
      }
      tree[pos].lazy = 0;
    }
  }

  Node _build(const int l, const int r, const vector<int> &arr, const int
    pos) {
    if (l == r)
      return tree[pos] = Node(arr[l]);

    int mid = (l + r) / 2;
    return tree[pos] = merge_nodes(_build(l, mid, arr, 2 * pos + 1),
                                   _build(mid + 1, r, arr, 2 * pos + 2));
  }

  int _get_first(const int l, const int r, const int i, const int j,
                 const int v, const int pos) {
    propagate(l, r, pos);

    if (l > r || l > j || r < i)
      return -1;
    // Needs RMQ MAX
    // Replace to <= for greater or (with RMQ MIN) > for smaller or
    // equal or >= for smaller
    if (tree[pos].val < v)
      return -1;

    if (l == r)
      return l;

    int mid = (l + r) / 2;
    int aux = _get_first(l, mid, i, j, v, 2 * pos + 1);
    if (aux != -1)
      return aux;
    return _get_first(mid + 1, r, i, j, v, 2 * pos + 2);
  }

  Node _query(const int l, const int r, const int i, const int j,
              const int pos) {
    propagate(l, r, pos);

    if (l > r || l > j || r < i)
```

```
      return NEUTRAL_NODE;

    if (i <= l && r <= j)
      return tree[pos];

    int mid = (l + r) / 2;
    return merge_nodes(_query(l, mid, i, j, 2 * pos + 1),
                       _query(mid + 1, r, i, j, 2 * pos + 2));
  }

  // It adds a number delta to the range from i to j
  Node _update(const int l, const int r, const int i, const int j,
               const int delta, const int pos) {
    propagate(l, r, pos);

    if (l > r || l > j || r < i)
      return tree[pos];

    if (i <= l && r <= j) {
      tree[pos].lazy = delta;
      propagate(l, r, pos);
      return tree[pos];
    }

    int mid = (l + r) / 2;
    return tree[pos] =
               merge_nodes(_update(l, mid, i, j, delta, 2 * pos + 1),
                           _update(mid + 1, r, i, j, delta, 2 * pos + 2));
  }

  void build(const vector<int> &arr) {
    this->tree.resize(4 * this->n);
    this->_build(0, this->n - 1, arr, 0);
  }

public:
  /// N equals to -1 means the Segment Tree hasn't been created yet.
  Seg_Tree() : n(-1) {}

  /// Constructor responsible for initializing a tree with 0.
  ///
  /// Time Complexity O(n)
  Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0)); }

  /// Constructor responsible for building the initial tree based on a vector.
  ///
  /// Time Complexity O(n)
  Seg_Tree(const vector<int> &arr) : n(arr.size()) { this->build(arr); }

  /// Returns the first index from i to j compared to v.
  /// Uncomment the line in the original function to get the proper element that
  /// may be: GREATER OR EQUAL, GREATER, SMALLER OR EQUAL, SMALLER.
  ///
  /// Time Complexity O(log n)
  int get_first(const int i, const int j, const int v) {
    assert(this->n >= 0);
    return this->_get_first(0, this->n - 1, i, j, v, 0);
  }

  /// Update at a single index.
  ///
  /// Time Complexity O(log n)
  void update(const int idx, const int delta) {
    assert(this->n >= 0);
    assert(0 <= idx), assert(idx < this->n);
    this->_update(0, this->n - 1, idx, idx, delta, 0);
  }

  /// Range update from l to r.
  ///
  /// Time Complexity O(log n)
  void update(const int l, const int r, const int delta) {
    assert(this->n >= 0);
    assert(0 <= l), assert(l <= r), assert(r < this->n);
    this->_update(0, this->n - 1, l, r, delta, 0);
  }

  /// Query at a single index.
  ///
  /// Time Complexity O(log n)
  int query(const int idx) {
    assert(this->n >= 0);
    assert(0 <= idx), assert(idx < this->n);
    return this->_query(0, this->n - 1, idx, idx, 0).val;
  }

  /// Range query from l to r.
  ///
  /// Time Complexity O(log n)
  int query(const int l, const int r) {
    assert(this->n >= 0);
    assert(0 <= l), assert(l <= r), assert(r < this->n);
    return this->_query(0, this->n - 1, l, r, 0).val;
  }
};
```

2.11. Segment Tree 2D

```
// REQUIRES segment_tree.cpp!!
class Seg_Tree_2d {
 private:
  // // range sum
  // int NEUTRAL_VALUE = 0;
  // int merge_nodes(const int &x, const int &y) {
  //   return x + y;
  // }

  // // RMQ max
  // int NEUTRAL_VALUE = -INF;
  // int merge_nodes(const int &x, const int &y) {
  //   return max(x, y);
  // }

  // // RMQ min
  // int NEUTRAL_VALUE = INF;
  // int merge_nodes(const int &x, const int &y) {
  //   return min(x, y);
  // }

 private:
  int n, m;

 public:
  vector<Seg_Tree> tree;

 private:
```

```
  void st_build(const int l, const int r, const int pos, const
    vector<vector<int>> &mat) {
    if(l == r)
      tree[pos] = Seg_Tree(mat[l]);
    else {
      int mid = (l + r) / 2;
      st_build(l, mid, 2*pos + 1, mat);
      st_build(mid + 1, r, 2*pos + 2, mat);
      for(int i = 0; i < tree[2*pos + 1].tree.size(); i++)
        tree[pos].tree[i].val = merge_nodes(tree[2*pos + 1].tree[i].val,
                                            tree[2*pos + 2].tree[i].val);
    }
  }

  int st_query(const int l, const int r, const int x1, const int y1, const
    int x2, const int y2, const int pos) {
    if(l > x2 || r < x1)
      return NEUTRAL_VALUE;

    if(x1 <= l && r <= x2)
      return tree[pos].query(y1, y2);

    int mid = (l + r) / 2;
    return merge_nodes(st_query(l, mid, x1, y1, x2, y2, 2*pos + 1),
                       st_query(mid + 1, r, x1, y1, x2, y2, 2*pos + 2));
  }

  void st_update(const int l, const int r, const int x, const int y, const
    int delta, const int pos) {
    if(l > x || r < x)
      return;

    // Only supports point updates.
    if(l == r) {
      tree[pos].update(y, delta);
      return;
    }

    int mid = (l + r) / 2;
    st_update(l, mid, x, y, delta, 2*pos + 1);
    st_update(mid + 1, r, x, y, delta, 2*pos + 2);
    tree[pos].update(y, delta);
  }

 public:
  Seg_Tree_2d() {
    this->n = -1;
    this->m = -1;
  }

  Seg_Tree_2d(const int n, const int m) {
    this->n = n;
    this->m = m;
    // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
    assert(m < 10000);
    tree.resize(4 * n, Seg_Tree(m));
  }

  Seg_Tree_2d(const int n, const int m, const vector<vector<int>> &mat) {
    this->n = n;
    this->m = m;
    // MAY TLE IN BUILD, TEST IT OR UPDATE EACH NODE MANUALLY!
    assert(m < 10000);
    tree.resize(4 * n, Seg_Tree(m));
    st_build(0, n - 1, 0, mat);
  }

  // Query from (x1, y1) to (x2, y2).
  //
  // Time complexity: O((log n) * (log m))
  int query(const int x1, const int y1, const int x2, const int y2) {
    assert(this->n > -1);
    assert(0 <= x1); assert(x1 <= x2); assert(x2 < this->n);
    assert(0 <= y1); assert(y1 <= y2); assert(y2 < this->n);
    return st_query(0, this->n - 1, x1, y1, x2, y2, 0);
  }

  // Point updates on position (x, y).
  //
  // Time complexity: O((log n) * (log m))
  void update(const int x, const int y, const int delta) {
    assert(0 <= x); assert(x < this->n);
    assert(0 <= y); assert(y < this->n);
    st_update(0, this->n - 1, x, y, delta, 0);
  }
};
```

2.12. Segment Tree Polynomial

```
/// Works for the polynomial f(x) = z1*x + z0
class Seg_Tree {
public:
  struct Node {
    int val, z1, z0;

    Node() {}
    Node(const int val, const int z1, const int z0)
        : val(val), z1(z1), z0(z0) {}
  };

private:
  // range sum
  Node NEUTRAL_NODE = Node(0, 0, 0);
  Node merge_nodes(const Node &x, const Node &y) {
    return Node(x.val + y.val, 0, 0);
  }
  void apply_lazy(const int l, const int r, const int pos) {
    tree[pos].val += (r - l + 1) * tree[pos].z0;
    tree[pos].val += (r - l) * (r - l + 1) / 2 * tree[pos].z1;
  }

private:
  int n;

public:
  vector<Node> tree;

private:
  void st_propagate(const int l, const int r, const int pos) {
    if (tree[pos].z0 != 0 || tree[pos].z1 != 0) {
      apply_lazy(l, r, pos);
      int mid = (l + r) / 2;
      int sz_left = mid - l + 1;
      if (l != r) {
        tree[2 * pos + 1].z0 += tree[pos].z0;
        tree[2 * pos + 1].z1 += tree[pos].z1;

        tree[2 * pos + 2].z0 += tree[pos].z0 + sz_left * tree[pos].z1;
        tree[2 * pos + 2].z1 += tree[pos].z1;
```

```cpp
      }
      tree[pos].z0 = 0;
      tree[pos].z1 = 0;
    }
  }

  Node st_build(const int l, const int r, const vector<int> &arr,
                const int pos) {
    if (l == r)
      return tree[pos] = Node(arr[l], 0, 0);

    int mid = (l + r) / 2;
    return tree[pos] = merge_nodes(st_build(l, mid, arr, 2 * pos + 1),
                                   st_build(mid + 1, r, arr, 2 * pos + 2));
  }

  Node st_query(const int l, const int r, const int i, const int j,
                const int pos) {
    st_propagate(l, r, pos);

    if (l > r || l > j || r < i)
      return NEUTRAL_NODE;

    if (i <= l && r <= j)
      return tree[pos];

    int mid = (l + r) / 2;
    return merge_nodes(st_query(l, mid, i, j, 2 * pos + 1),
                       st_query(mid + 1, r, i, j, 2 * pos + 2));
  }

  // it adds a number delta to the range from i to j
  Node st_update(const int l, const int r, const int i, const int j,
                 const int z1, const int z0, const int pos) {
    st_propagate(l, r, pos);

    if (l > r || l > j || r < i)
      return tree[pos];

    if (i <= l && r <= j) {
      tree[pos].z0 = (l - i + 1) * z0;
      tree[pos].z1 = z1;
      st_propagate(l, r, pos);
      return tree[pos];
    }

    int mid = (l + r) / 2;
    return tree[pos] =
               merge_nodes(st_update(l, mid, i, j, z1, z0, 2 * pos + 1),
                           st_update(mid + 1, r, i, j, z1, z0, 2 * pos + 2));
  }

public:
  Seg_Tree() : n(-1) {}

  Seg_Tree(const int n) : n(n) { this->tree.resize(4 * this->n, Node(0, 0));
    }

  Seg_Tree(const vector<int> &arr) { this->build(arr); }

  void build(const vector<int> &arr) {
    this->n = arr.size();
    this->tree.resize(4 * this->n);
    this->st_build(0, this->n - 1, arr, 0);
  }

  /// Index update of a polynomial f(x) = z1*x + z0
  ///
  /// Time Complexity O(log n)
  void update(const int i, const int z1, const int z0) {
    assert(this->n >= 0);
    assert(0 <= i), assert(i < this->n);
    this->st_update(0, this->n - 1, i, i, z1, z0, 0);
  }

  /// Range update of a polynomial f(x) = z1*x + z0 from l to r
  ///
  /// Time Complexity O(log n)
  void update(const int l, const int r, const int z1, const int z0) {
    assert(this->n >= 0);
    assert(0 <= l), assert(l <= r), assert(r < this->n);
    this->st_update(0, this->n - 1, l, r, z1, z0, 0);
  }

  /// Range sum query from l to r
  ///
  /// Time Complexity O(log n)
  int query(const int l, const int r) {
    assert(this->n >= 0);
    assert(0 <= l), assert(l <= r), assert(r < this->n);
    return this->st_query(0, this->n - 1, l, r, 0).val;
  }
};
```

2.13. Sparse Table

```cpp
class Sparse_Table {
private:
  /// Sparse table min
  int merge(const int l, const int r) { return min(l, r); }
  /// Sparse table max
  // int merge(const int l, const int r) { return max(l, r); }

private:
  int n;
  vector<vector<int>> table;
  vector<int> lg;

private:
  /// lg[i] represents the log2(i)
  void build_log_array() {
    lg.resize(this->n + 1);
    for (int i = 2; i <= this->n; i++)
      lg[i] = lg[i / 2] + 1;
  }

  /// Time Complexity: O(n*log(n))
  void build_sparse_table(const vector<int> &arr) {
    table.resize(lg[this->n] + 1, vector<int>(this->n));

    table[0] = arr;
    int pow2 = 1;
    for (int i = 1; i < table.size(); i++) {
      int lastsz = this->n - pow2 + 1;
      for (int j = 0; j + pow2 < lastsz; j++) {
        table[i][j] = merge(table[i - 1][j], table[i - 1][j + pow2]);
      }
      pow2 <<= 1;
    }
```

```cpp
  }

public:
  /// Constructor that builds the log array and the sparse table.
  ///
  /// Time Complexity: O(n*log(n))
  Sparse_Table(const vector<int> &arr) : n(arr.size()) {
    this->build_log_array();
    this->build_sparse_table(arr);
  }

  void print() {
    int pow2 = 1;
    for (int i = 0; i < table.size(); i++) {
      int sz = (int)(table.front().size()) - pow2 + 1;
      for (int j = 0; j < sz; j++) {
        cout << table[i][j] << " \n"[(j + 1) == sz];
      }
      pow2 <<= 1;
    }
  }

  /// Range query from l to r.
  ///
  /// Time Complexity: O(1)
  int query(const int l, const int r) {
    assert(l <= r);
    assert(0 <= l && r <= this->n - 1);

    int lgg = lg[r - l + 1];
    return merge(table[lgg][l], table[lgg][r - (1 << lgg) + 1]);
  }
};
```

2.14. Treap

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

// #define REVERSE
// #define LAZY
class Treap {
public:
  struct Node {
    Node *left = nullptr, *right = nullptr, *par = nullptr;
    // Priority to be used in the treap
    const int rank;
    int size = 1, val;
    // Contains the result of the range query between the node and its
    children.
    int ans;
#ifdef LAZY
    int lazy = 0;
#endif
#ifdef REVERSE
    bool rev = false;
#endif

    Node(const int val) : val(val), ans(val), rank(rng()) {}
    Node(const int val, const int rank) : val(val), ans(val), rank(rank) {}
  };

private:
  vector<Node *> nodes;
  int _size = 0;
  Node *root = nullptr;

private:
  // // Range Sum
  // void merge_nodes(Node *node) {
  //   node->ans = node->val;
  //   if (node->left)
  //     node->ans += node->left->ans;
  //   if (node->right)
  //     node->ans += node->right->ans;
  // }

  // #ifdef LAZY
  //   void apply_lazy(Node *node) {
  //     node->val += node->lazy;
  //     node->ans += node->lazy * get_size(node);
  //   }
  // #endif

  // // RMQ Min
  // void merge_nodes(Node *node) {
  //   node->ans = node->val;
  //   if (node->left)
  //     node->ans = min(node->ans, node->left->ans);
  //   if (node->right)
  //     node->ans = min(node->ans, node->right->ans);
  // }

  // #ifdef LAZY
  //   void apply_lazy(Node *node) {
  //     node->val += node->lazy;
  //     node->ans += node->lazy;
  //   }
  // #endif

  // // RMQ Max
  // void merge_nodes(Node *node) {
  //   node->ans = node->val;
  //   if (node->left)
  //     node->ans = max(node->ans, node->left->ans);
  //   if (node->right)
  //     node->ans = max(node->ans, node->right->ans);
  // }

  // #ifdef LAZY
  //   void apply_lazy(Node *node) {
  //     node->val += node->lazy;
  //     node->ans += node->lazy;
  //   }
  // #endif

  int get_size(const Node *node) { return node ? node->size : 0; }

  void update_size(Node *node) {
    if (node)
      node->size = 1 + get_size(node->left) + get_size(node->right);
  }

#ifdef REVERSE
  void propagate_reverse(Node *node) {
    if (node && node->rev) {
      swap(node->left, node->right);
      if (node->left)
        node->left->rev ^= 1;
      if (node->right)
```

```
        node->right->rev ^= 1;
      node->rev = 0;
    }
  }
#endif

#ifdef LAZY
  void propagate_lazy(Node *node) {
    if (node && node->lazy != 0) {
      apply_lazy(node);
      if (node->left)
        node->left->lazy += node->lazy;
      if (node->right)
        node->right->lazy += node->lazy;
      node->lazy = 0;
    }
  }
#endif

  void update_node(Node *node) {
    if (node) {
      update_size(node);
#ifdef LAZY
      propagate_lazy(node->left);
      propagate_lazy(node->right);
#endif
#ifdef REVERSE
      propagate_reverse(node);
#endif
      merge_nodes(node);
    }
  }

  /// Splits the treap into to different treaps that contains nodes with indexes
  /// <= pos ans indexes > pos. The nodes l and r contains, in the end, these
  /// two different treaps.
  void split(Node *node, Node *&l, Node *&r, const int pos, Node *pl = nullptr,
             Node *pr = nullptr) {
    if (!node)
      l = r = nullptr;
    else {
#ifdef LAZY
      propagate_lazy(node);
#endif
#ifdef REVERSE
      propagate_reverse(node);
#endif
      if (get_size(node->left) <= pos) {
        node->par = pr;
        split(node->right, node->right, r, pos - get_size(node->left) - 1, pl,
              node);
        l = node;
      } else {
        node->par = pl;
        split(node->left, l, node->left, pos, node, pr);
        r = node;
      }
    }
    update_node(node);
  }

  /// Merges to treaps (l and r) into a single one based on the rank of each
  /// node.
  void merge(Node *&node, Node *l, Node *r, Node *par = nullptr) {
#ifdef LAZY
    propagate_lazy(l), propagate_lazy(r);
#endif
#ifdef REVERSE
    propagate_reverse(l), propagate_reverse(r);
#endif
    if (l == nullptr || r == nullptr)
      node = (l == nullptr ? r : l);
    else if (l->rank > r->rank) {
      merge(l->right, l->right, r, l);
      node = l;
    } else {
      merge(r->left, l, r->left, r);
      node = r;
    }
    if (node)
      node->par = par;
    update_node(node);
  }

  Node *build(const int l, const int r, const vector<int> &arr,
              vector<int> &rand) {
    if (l > r)
      return nullptr;

    const int mid = (l + r) / 2;
    Node *node = new Node(arr[mid], rand.back());
    rand.pop_back();
    node->right = build(mid + 1, r, arr, rand);
    node->left = build(l, mid - 1, arr, rand);
    update_node(node);

    return node;
  }

  int _get_ith(const int idx) {
    int ans = 0;
    Node *cur = nodes[idx], *prev = nullptr;
    while (cur) {
      if (cur == nodes[idx] || prev == cur->right)
        ans += 1 + get_size(cur->left);
      prev = cur;
      cur = cur->par;
    }
    return ans - 1;
  }

  vector<int> gen_rand(const int n) {
    vector<int> ans(n);
    for (int &x : ans)
      x = rng();
    sort(ans.begin(), ans.end());
    return ans;
  }

  Node *_query(const int l, const int r) {
    Node *L, *M, *R;
    split(this->root, L, M, l - 1);
    split(M, M, R, r - l);
    Node *ret = new Node(*M);
    merge(L, L, M);
    merge(root, L, R);
    return ret;
```

```
  }

  void _update(const int l, const int r, const int delta) {
    Node *L, *M, *R;
    split(this->root, L, M, l - 1);
    split(M, M, R, r - l);

    Node *node = M;
#ifdef LAZY
    node->lazy = delta;
    propagate_lazy(node);
#else
    node->val += delta;
#endif

    merge(L, L, M);
    merge(root, L, R);
  }

  void _insert(const int pos, Node *node) {
    this->_size += node->size;
    Node *L, *R;
    split(this->root, L, R, pos - 1);
    merge(L, L, node);
    merge(this->root, L, R);
  }

  Node *_erase(const int l, const int r) {
    Node *L, *M, *R;
    split(this->root, L, M, l - 1);
    split(M, M, R, r - l);
    merge(root, L, R);
    this->_size -= r - l + 1;
    return M;
  }

  void _move(const int l, const int r, const int new_pos) {
    Node *node = _erase(l, r);
    _insert(new_pos, node);
  }

#ifdef REVERSE
  void _reverse(const int l, const int r) {
    Node *L, *M, *R;
    split(this->root, L, M, l - 1);
    split(M, M, R, r - l);

    Node *node = M;
    node->rev = true;
    propagate_reverse(node);

    merge(L, L, M);
    merge(root, L, R);
  }
#endif

public:
  Treap() {}

  /// Constructor that initializes the treap based on an array.
  ///
  /// Time Complexity: O(n)
  Treap(const vector<int> &arr) : _size(arr.size()) {
    vector<int> r = gen_rand(arr.size());
    this->root = build(0, (int)arr.size() - 1, arr, r);
  }

  int size() { return _size; }

  /// Moves the subarray [l, r] to the position starting at new_pos.
  ///
  /// Time Complexity: O(log n)
  void move(const int l, const int r, const int new_pos) {
    assert(0 <= new_pos), assert(new_pos <= _size - (r - l + 1));
    _move(l, r, new_pos);
  }

  /// Moves the subarray [l, r] to the back of the array.
  ///
  /// Time Complexity: O(log n)
  void move_back(const int l, const int r) {
    assert(0 <= l), assert(l <= r), assert(r < _size);
    _move(l, r, _size - (r - l + 1));
  }

  /// Moves the subarray [l, r] to the front of the array.
  ///
  /// Time Complexity: O(log n)
  void move_front(const int l, const int r) {
    assert(0 <= l), assert(l <= r), assert(r < _size);
    _move(l, r, 0);
  }

#ifdef REVERSE
  /// Reverses the subarray [l, r].
  ///
  /// Time Complexity: O(log n)
  void reverse(const int l, const int r) {
    assert(0 <= l), assert(l <= r), assert(r < _size);
    _reverse(l, r);
  }
#endif

  /// Erases the subarray [l, r].
  ///
  /// Time Complexity: O(log n)
  void erase(const int l, const int r) {
    assert(0 <= l), assert(l <= r), assert(r < _size);
    _erase(l, r);
  }

  /// Inserts the value val at the position pos.
  ///
  /// Time Complexity: O(log n)
  void insert(const int pos, const int val) {
    assert(pos <= _size);
    nodes.emplace_back(new Node(val));
    _insert(pos, nodes.back());
  }

  /// Returns the index of the i-th added node.
  ///
  /// Time Complexity: O(log n)
  int get_ith(const int idx) {
    assert(0 <= idx), assert(idx < nodes.size());
    return _get_ith(idx);
  }

  /// Sums the delta value to the position pos.
  ///
```

```
  /// Time Complexity: O(log n)
  void update(const int pos, const int delta) {
    assert(0 <= pos), assert(pos < _size);
    _update(pos, pos, delta);
  }

#ifdef LAZY
  /// Sums the delta value to the subarray [l, r].
  ///
  /// Time Complexity: O(log n)
  void update(const int l, const int r, const int delta) {
    assert(0 <= l), assert(l <= r), assert(r < _size);
    _update(l, r, delta);
  }
#endif

  /// Query at a single index.
  ///
  /// Time Complexity: O(log n)
  int query(const int pos) {
    assert(0 <= pos), assert(pos < _size);
    return _query(pos, pos)->ans;
  }

  /// Range query from l to r.
  ///
  /// Time Complexity: O(log n)
  int query(const int l, const int r) {
    assert(0 <= l), assert(l <= r), assert(r < _size);
    return _query(l, r)->ans;
  }
};
```

3. Dp

3.1. Achar Maior Palindromo

```
Fazer LCS da string com o reverso
```

3.2. Digit Dp

```
/// How many numbers x are there in the range a to b, where the digit d occurs exactly k times in x?
vector<int> num;
int a, b, d, k;
int DP[12][12][2];
/// DP[p][c][f] = Number of valid numbers <= b from this state
/// p = current position from left side (zero based)
/// c = number of times we have placed the digit d so far
/// f = the number we are building has already become smaller than b? [0 = no, 1 = yes]

int call(int pos, int cnt, int f){
  if(cnt > k) return 0;

  if(pos == num.size()){
    if(cnt == k) return 1;
    return 0;
  }

  if(DP[pos][cnt][f] != -1) return DP[pos][cnt][f];
  int res = 0;
  int lim = (f ? 9 : num[pos]);

  /// Try to place all the valid digits such that the number doesn't exceed b
  for(int dgt = 0; dgt<=LMT; dgt++){
    int nf = f;
    int ncnt = cnt;
    if(f == 0 && dgt < LMT) nf = 1; /// The number is getting smaller at this position
    if(dgt == d) ncnt++;
    if(ncnt <= k) res += call(pos+1, ncnt, nf);
  }

  return DP[pos][cnt][f] = res;
}

int solve(int b){
  num.clear();
  while(b>0){
    num.push_back(b%10);
    b/=10;
  }
  reverse(num.begin(), num.end());
  /// Stored all the digits of b in num for simplicity

  memset(DP, -1, sizeof(DP));
  int res = call(0, 0, 0);
  return res;
}

int main () {

  cin >> a >> b >> d >> k;
  int res = solve(b) - solve(a-1);
  cout << res << endl;

  return 0;
}
```

3.3. Longest Common Subsequence

```
string lcs(string &s, string &t) {

  int n = s.size(), m = t.size();

  s.insert(s.begin(), '#');
  t.insert(t.begin(), '$');

  vector<vector<int>> mat(n + 1, vector<int>(m + 1, 0));

  for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= m; j++) {
      if(s[i] == t[j])
        mat[i][j] = mat[i - 1][j - 1] + 1;
      else
        mat[i][j] = max(mat[i - 1][j], mat[i][j - 1]);
    }
  }

  string ans;
  int i = n, j = m;
  while(i > 0 && j > 0) {
    if(s[i] == t[j])
      ans += s[i], i--, j--;
    else if(mat[i][j - 1] > mat[i - 1][j])
      j--;
```

```
    else
      i--;
  }

  reverse(ans.begin(), ans.end());
  return ans;
}
```

3.4. Longest Common Substring

```
int LCSubStr(char *X, char *Y, int m, int n) {
  // Create a table to store lengths of longest common suffixes of
  // substrings.   Notethat LCSuff[i][j] contains length of longest
  // common suffix of X[0..i-1] and Y[0..j-1]. The first row and
  // first column entries have no logical meaning, they are used only
  // for simplicity of program
  int LCSuff[m+1][n+1];
  int result = 0;  // To store length of the longest common substring

  /* Following steps build LCSuff[m+1][n+1] in bottom up fashion. */
  for (int i=0; i<=m; i++) {
    for (int j=0; j<=n; j++) {
      if (i == 0 || j == 0)
        LCSuff[i][j] = 0;

      else if (X[i-1] == Y[j-1]) {
        LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
        result = max(result, LCSuff[i][j]);
      }
      else LCSuff[i][j] = 0;
    }
  }
  return result;
}
```

3.5. Longest Increasing Subsequence 2D (Not Sorted)

```
set<ii> s[(int)2e6];
bool check(ii par, int ind) {

  auto it = s[ind].lower_bound(ii(par.ff, -INF));
  if(it == s[ind].begin())
    return false;

  it--;

  if(it->ss < par.ss)
    return true;
  return false;
}

int lis2d(vector<ii> &arr) {

  int n = arr.size();
  s[1].insert(arr[0]);

  int maior = 1;
  for(int i = 1; i < n; i++) {

    ii x = arr[i];

    int l = 1, r = maior;
    int ansbb = 0;
    while(l <= r) {
      int mid = (l+r)/2;
      if(check(x, mid)) {
        l = mid + 1;
        ansbb = mid;
      } else {
        r = mid - 1;
      }
    }

    // inserting in list
    auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    while(it != s[ansbb+1].end() && it->ss >= x.ss)
      it = s[ansbb+1].erase(it);

    it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
    it->ss <= x.ss)
      continue;
    s[ansbb+1].insert(arr[i]);

    maior = max(maior, ansbb + 1);
  }

  return maior;

}
```

3.6. Longest Increasing Subsequence 2D (Sorted)

```
set<ii> s[(int)2e6];
bool check(ii par, int ind) {

  auto it = s[ind].lower_bound(ii(par.ff, -INF));
  if(it == s[ind].begin())
    return false;

  it--;

  if(it->ss < par.ss)
    return true;
  return false;
}

int lis2d(vector<ii> &arr) {

  int n = arr.size();
  s[1].insert(arr[0]);

  int maior = 1;
  for(int i = 1; i < n; i++) {

    ii x = arr[i];

    int l = 1, r = maior;
    int ansbb = 0;
    while(l <= r) {
      int mid = (l+r)/2;
      if(check(x, mid)) {
        l = mid + 1;
        ansbb = mid;
      } else {
        r = mid - 1;
      }
```

```
    }

    // inserting in list
    auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    while(it != s[ansbb+1].end() && it->ss >= x.ss)
      it = s[ansbb+1].erase(it);

    it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
    it->ss <= x.ss)
      continue;
    s[ansbb+1].insert(arr[i]);

    maior = max(maior, ansbb + 1);
  }

  return maior;
}
```

3.7. Longest Increasing Subsequence

```
int lis(vector<int> &arr){
  int n = arr.size();
  vector<int> lis;
  for(int i = 0; i < n; i++){
    int l = 0, r = (int)lis.size() - 1;
    int ansj = -1;
    while(l <= r){
      int mid = (l+r)/2;
      // OBS: PARA >= TROCAR SINAL EMBAIXO POR <=
      if(arr[i] < lis[mid]){
        r = mid - 1;
        ansj = mid;
      }
      else l = mid + 1;
    }
    if(ansj == -1){
      // se arr[i] e maior que todos
      lis.push_back(arr[i]);
    }
    else {
      lis[ansj] = arr[i];
    }
  }

  return lis.size();
}
```

3.8. Subset Sum Com Bitset

```
bitset<312345> bit;
int arr[112345];
void subsetSum(int n) {
  bit.reset();
  bit.set(0);
  for(int i = 0; i < n; i++) {
    bit |= (bit << arr[i]);
  }
}
```

3.9. Catalan

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

3.10. Catalan

```
// The first few Catalan numbers for n = 0, 1, 2, 3, ...
// are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
// Formula Recursiva:
// cat(0) = 0
// cat(n+1) = somatorio(i from 0 to n)(cat(i)*cat(n-i))
//
// Using Binomial Coefficient
// We can also use the below formula to find nth catalan number in O(n) time.
// Formula acima

// Returns value of Binomial Coefficient C(n, k)

int binomialCoeff(int n, int k) {
  int res = 1;

  // Since C(n, k) = C(n, n-k)
  if (k > n - k)
    k = n - k;

  // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
  for (int i = 0; i < k; ++i) {
      res *= (n - i);
      res /= (i + 1);
  }

    return res;
}
// A Binomial coefficient based function to find nth catalan
// number in O(n) time
int catalan(int n) {
    // Calculate value of 2nCn
    int c = binomialCoeff(2*n, n);

    // return 2nCn/(n+1)
    return c/(n+1);
}
```

3.11. Coin Change Problem

```
// função que recebe o valor de troco N, o número de moedas disponíveis M,
// e um vetor com as moedas disponíveis arr
// essa função deve retornar o número mínimo de moedas,
// de acordo com a solução com Programação Dinamica.
int num_moedas(int N, int M, int arr[]) {
  int dp[N+1];
  // caso base
  dp[0] = 0;
  // sub-problemas
  for(int i=1; i<=N; i++) {
  // é comum atribuir um valor alto, que concerteza
  // é maior que qualquer uma das próximas possibilidades,
```

```
  // sendo assim substituido
  dp[i] = 1000000;
  for(int j=0; j<M; j++) {
    if(i-arr[j] >= 0) {
      dp[i] = min(dp[i], dp[i-arr[j]]+1);
    }
  }
  }
  // solução
  return dp[N];
}
```

3.12. Knapsack

```
int dp[2001][2001];
int moc(int q,int p,vector<ii> vec) {
  for(int i = 1; i <= q; i++)
  {
    for(int j = 1; j <= p;j++) {
      if(j >= vec[i-1].ff)
        dp[i][j] = max(dp[i-1][j],vec[i-1].ss + dp[i-1][j-vec[i-1].ff]);
      else
        dp[i][j] = dp[i-1][j];
    }
  }
  return dp[q][p];
}
int main(int argc, char *argv[])
{
  int p,q;
  vector<ii> vec;
  cin >> p >> q;
  int x,y;
  for(int i = 0; i < q; i++) {
       cin >> x >> y;
       vec.push_back(make_pair(x,y));
  }
  for(int i = 0; i <= p; i++)
    dp[0][i] = 0;
  for(int i = 1; i <= q; i++)
    dp[i][0] = 0;
  sort(vec.begin(),vec.end());
  cout << moc(q,p,vec) << endl;
}
```

4. Geometry

4.1. Centro De Massa De Um Poligono

```
double area = 0;
pto c;

c.x = c.y = 0;
for(int i = 0; i < n ; i++) {
  double aux = (arr[i].x * arr[i+1].y) - (arr[i].y * arr[i+1].x); // shoelace
  area += aux;
  c.x += aux*(arr[i].x + arr[i+1].x);
  c.y += aux*(arr[i].y + arr[i+1].y);
}

c.x /= (3.0*area);
c.y /= (3.0*area);

cout  << c.x <<  ' ' << c.y << endl;
```

4.2. Closest Pair Of Points

```
struct Point {
  int x, y;
};
int compareX(const void *a,const void *b){
  Point *p1 = (Point *)a,  *p2 = (Point *)b;
  return (p1->x - p2->x);
}
int compareY(const void *a,const void *b) {
  Point *p1 = (Point *)a,*p2 =(Point *)b;
  return (p1->y - p2->y);
}
float dist(Point p1, Point p2) {
  return sqrt((p1.x- p2.x)*(p1.x- p2.x) +(p1.y - p2.y)*(p1.y - p2.y));
}
float bruteForce(Point P[], int n){
  float min = FLT_MAX;
  for (int i = 0; i < n; ++i)
    for (int j = i+1; j < n; ++j)
      if (dist(P[i], P[j]) < min)
        min = dist(P[i], P[j]);
  return min;
}
float min(float x, float y) {
  return (x < y)? x : y;
}
float stripClosest(Point strip[], int size, float d) {
  float min = d;
  for (int i = 0; i < size; ++i)
    for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
      if (dist(strip[i],strip[j]) < min)
        min = dist(strip[i], strip[j]);
  return min;
}
float closestUtil(Point Px[], Point Py[], int n){
  if (n <= 3)
    return bruteForce(Px, n);
  int mid = n/2;
  Point midPoint = Px[mid];
  Point Pyl[mid+1];
  Point Pyr[n-mid-1];
  int li = 0, ri = 0;
  for (int i = 0; i < n; i++)
    if (Py[i].x <= midPoint.x)
      Pyl[li++] = Py[i];
    else
      Pyr[ri++] = Py[i];

  float dl = closestUtil(Px, Pyl, mid);
  float dr = closestUtil(Px + mid, Pyr, n-mid);
  float d = min(dl, dr);
  Point strip[n];
  int j = 0;
  for (int i = 0; i < n; i++)
    if (abs(Py[i].x - midPoint.x) < d)
      strip[j] = Py[i], j++;
  return min(d, stripClosest(strip, j, d));
}

float closest(Point P[], int n) {
  Point Px[n];
```

```
  Point Py[n];
  for (int i = 0; i < n; i++) {
    Px[i] = P[i];
    Py[i] = P[i];
  }
  qsort(Px, n, sizeof(Point), compareX);
  qsort(Py, n, sizeof(Point), compareY);
  return closestUtil(Px, Py, n);
}
```

4.3. Condicao De Existencia De Um Triangulo

```

    | b - c | < a < b + c
    | a - c | < b < a + c
    | a - b | < c < a + b

Para a < b < c, basta checar
    a + b > c

OBS: Para um conjunto n >= 100 sempre exite um triângulo válido, pois a
    sequência de triângulos não válidos seguem a sequência de Fibonacci e
    Fib(100) > 2^64
```

4.4. Convex Hull

```
// Asymptotic complexity: O(n log n).
struct pto {
  double x, y;
  bool operator <(const pto &p) const {
    return x < p.x || (x == p.x && y < p.y);
    /* a impressao será em prioridade por mais a esquerda, mais
       abaixo, e antihorário pelo cross abaixo */
  }
};

double cross(const pto &O, const pto &A, const pto &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<pto> convex_hull(vector<pto> P) {
  int n = P.size(), k = 0;
  vector<pto> H(2 * n);
  // Sort points lexicographically
  sort(P.begin(), P.end());
  // Build lower hull
  for (int i = 0; i < n; ++i) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  // Build upper hull
  for (int i = n - 2, t = k + 1; i >= 0; i--) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  H.resize(k);
  /* o último ponto do vetor é igual ao primeiro, atente para isso
  as vezes é necessário mudar */
  return H;
}
```

4.5. Cross Product

```
// Outra forma de produto vetorial
// reta ab,ac se for zero e colinear
// se for < 0 entao antiHorario, > 0 horario
bool ehcol(pto a,pto b,pto c) {
  return ((b.y-a.y)*(c.x-a.x) - (b.x-a.x)*(c.y-a.y));
}
---------------------------------------
//Produto vetorial AB x AC, se for zero e colinear
int cross(pto A, pto B, pto C){
  pto AB, AC;
  AB.x = B.x-A.x;
  AB.y = B.y-A.y;
  AC.x = C.x-A.x;
  AC.y = C.y-A.y;
  int cross = AB.x*AC.y-AB.y * AC.x;
  return cross;
}

// OBS: DEFINE ÁREA DE QUADRILÁTERO FORMADO PELAS RETAS, A ÁREA DO TRIÂNGULO
   É A METADE
```

4.6. Distance Point Segment

```
// use struct point and line
double dist_point_segment(const Point p, const Point s, const Point t) {
    if(sgn(dot(p-s, t-s)) < 0)
      return (p-s).norm();
    if(sgn(dot(p-t, s-t)) < 0)
      return (p-t).norm();
    return abs(det(s-p, t-p) / dist(s, t));
}
```

4.7. Line-Line Intersection

```
// Intersecção de retas Ax + By = C    dados pontos (x1,y1) e (x2,y2)
A = y2-y1
B = x1-x2
C = A*x1+B*y1
//Retas definidas pelas equações:
A1x + B1y = C1
A2x + B2y = C2
//Encontrar x e y resolvendo o sistema
double det = A1*B2 - A2*B1;
if(det == 0){
  //Lines are parallel
}else{
  double x = (B2*C1 - B1*C2)/det;
  double y = (A1*C2 - A2*C1)/det;
}
```

4.8. Line-Point Distance

```
double ptoReta(double x1, double y1, double x2,double y2,double pointX,
    double pointY, double *ptox,double *ptoy){
  double diffX = x2 - x1;
```

```
  double diffY = y2 - y1;
  if ((diffX == 0) && (diffY == 0)) {
    diffX = pointX - x1;
    diffY = pointY - y1;
    //se os dois sao pontos
    return hypot(pointX - x1,pointY - y1);
  }
  double t = ((pointX - x1) * diffX + (pointY - y1) * diffY) /
              (diffX * diffX + diffY * diffY);
  if (t < 0) {
    //point is nearest to the first point i.e x1 and y1
    // Ex:      _______________    .
    // cord do pto na reta = pto inicial(x1,y1);
    *ptox = x1, *ptoy = y1;
    diffX = pointX - x1;
    diffY = pointY - y1;
  } else if (t > 1) {
    //point is nearest to the end point i.e x2 and y2
    // Ex : .    _________________
    // cord do pto na reta = pto final(x2,y2);
    *ptox = x2, *ptoy = y2;
    diffX = pointX - x2;
    diffY = pointY - y2;
  } else  {
      //if perpendicular line intersect the line segment.
      // pto nao esta mais proximo de uma das bordas do segmento
      // Ex:               .
      //                   |
      //                   |(Ângulo Reto)
      //________________________________________
      // cord x do pto na reta = (x1 + t * diffX)
      // cord y do pto na reta = (y1 + t * diffY)
    *ptox = (x1 + t * diffX), *ptoy = (y1 + t * diffY);
    diffX = pointX - (x1 + t * diffX);
    diffY = pointY - (y1 + t * diffY);
  }
  //returning shortest distance
  return sqrt(diffX * diffX + diffY * diffY);
}
```

4.9. Point Inside Convex Polygon - Log(N)

```
#include <bits/stdc++.h>

using namespace std;

#define INF 1e18
#define pb push_back
#define ii pair<int,int>
#define OK cout<<"OK"<<endl
#define debug(x) cout << #x " = " << (x) << endl
#define ff first
#define ss second
#define int long long

struct pto {
  double x, y;
  bool operator <(const pto &p) const {
    return x < p.x || (x == p.x && y < p.y);
    /* a impressao será em prioridade por mais a esquerda, mais
       abaixo, e antihorário pelo cross abaixo */
  }
};
double cross(const pto &O, const pto &A, const pto &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<pto> lower, upper;

vector<pto> convex_hull(vector<pto> &P) {
  int n = P.size(), k = 0;
  vector<pto> H(2 * n);
  // Sort points lexicographically
  sort(P.begin(), P.end());
  // Build lower hull
  for (int i = 0; i < n; ++i) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  // Build upper hull
  for (int i = n - 2, t = k + 1; i >= 0; i--) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  H.resize(k);
  /* o último ponto do vetor é igual ao primeiro, atente para isso
  as vezes é necessário mudar */

  int j = 1;
  lower.pb(H.front());
  while(H[j].x >= H[j-1].x) {
    lower.pb(H[j++]);
  }

  int l = H.size()-1;
  while(l >= j) {
    upper.pb(H[l--]);
  }
  upper.pb(H[l--]);

  return H;
}

bool insidePolygon(pto p, vector<pto> &arr) {

  if(pair<double,double>(p.x, p.y) == pair<double,double>(lower[0].x, lower[0].y))
    return true;

  pto lo = {p.x, -(double)INF};
  pto hi = {p.x, (double)INF};
  auto itl = lower_bound(lower.begin(), lower.end(), lo);
  auto itu = lower_bound(upper.begin(), upper.end(), lo);

  if(itl == lower.begin() || itu == upper.begin()) {
    auto it  = lower_bound(arr.begin(), arr.end(), lo);
    auto it2 = lower_bound(arr.begin(), arr.end(), hi);
    it2--;
    if(it2 >= it && p.x == it-> x && it->x == it2->x && it->y <= p.y && p.y <= it2->y)
      return true;
    return false;
  }
```

```
  if(itl == lower.end() || itu == upper.end())  {
    return false;
  }

  auto ol = itl, ou = itu;
  ol--, ou--;
  if(cross(*ol, *itl, p) >= 0 && cross(*ou, *itu, p) <= 0)
    return true;

  auto it  = lower_bound(arr.begin(), arr.end(), lo);
  auto it2 = lower_bound(arr.begin(), arr.end(), hi);
  it2--;
  if(it2 >= it && p.x == it-> x && it->x == it2->x && it->y <= p.y && p.y <= it2->y)
    return true;

  return false;

}

signed main () {

  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  double n, m, k;

  cin >> n >> m >> k;

  vector<pto> arr(n);

  for(pto &x: arr) {
    cin >> x.x >> x.y;
  }

  convex_hull(arr);

  pto p;

  int c = 0;
  while(m--) {
    cin >> p.x >> p.y;
    cout << (insidePolygon(p, arr) ? "dentro" : "fora") << endl;
  }

}
```

4.10. Point Inside Polygon

```

/* 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 */

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct pto {
    int x, y;
    pto() {}
    pto(int x, int y) : x(x), y(y) {}
};

// Given three colinear ptos p, q, r, the function checks if
// pto q lies on line segment 'pr'
bool onSegment(pto p, pto q, pto r) {
  if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
      q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
    return true;
  return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(pto p, pto q, pto r) {
  int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);

  if (val == 0) return 0;  // colinear
  return (val > 0)? 1: 2; // clock or counterclock wise
}

// The function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
  // Find the four orientations needed for general and
  // special cases
  int o1 = orientation(p1, q1, p2);
  int o2 = orientation(p1, q1, q2);
  int o3 = orientation(p2, q2, p1);
  int o4 = orientation(p2, q2, q1);

  // General case
  if (o1 != o2 && o3 != o4)
    return true;

  // Special Cases
  // p1, q1 and p2 are colinear and p2 lies on segment p1q1
  if (o1 == 0 && onSegment(p1, p2, q1)) return true;

  // p1, q1 and p2 are colinear and q2 lies on segment p1q1
  if (o2 == 0 && onSegment(p1, q2, q1)) return true;

  // p2, q2 and p1 are colinear and p1 lies on segment p2q2
  if (o3 == 0 && onSegment(p2, p1, q2)) return true;

   // p2, q2 and q1 are colinear and q1 lies on segment p2q2
  if (o4 == 0 && onSegment(p2, q1, q2)) return true;

  return false; // Doesn't fall in any of the above cases
}

// Returns true if the pto p lies inside the polygon[] with n vertices
bool isInside(pto polygon[], int n, pto p) {
  // There must be at least 3 vertices in polygon[]
  if (n < 3)  return false;

  // Create a pto for line segment from p to infinite
  pto extreme = pto(INF, p.y);

  // Count intersections of the above line with sides of polygon
  int count = 0, i = 0;
  do {
    int next = (i+1)%n;

    // Check if the line segment from 'p' to 'extreme' intersects
```

```
    // with the line segment from 'polygon[i]' to 'polygon[next]'
    if (doIntersect(polygon[i], polygon[next], p, extreme)) {
      // If the pto 'p' is colinear with line segment 'i-next',
      // then check if it lies on segment. If it lies, return true,
      // otherwise false
      if (orientation(polygon[i], p, polygon[next]) == 0)
        return onSegment(polygon[i], p, polygon[next]);

      count++;
    }
    i = next;
  } while (i != 0);

  // Return true if count is odd, false otherwise
  return count&1;  // Same as (count%2 == 1)
}
```

4.11. Points Inside And In Boundary Polygon

```
int cross(pto a, pto b) {
  return a.x * b.y - b.x * a.y;
}

int boundaryCount(pto a, pto b) {
  if(a.x == b.x)
    return abs(a.y-b.y)-1;
  if(a.y == b.y)
    return abs(a.x-b.x)-1;
  return _gcd(abs(a.x-b.x), abs(a.y-b.y))-1;
}

int totalBoundaryPolygon(vector<pto> &arr, int n) {

  int boundPoint = n;
  for(int i = 0; i < n; i++) {
    boundPoint += boundaryCount(arr[i], arr[(i+1)%n]);
  }
  return boundPoint;
}

int polygonArea2(vector<pto> &arr, int n) {
  int area = 0;
  // N = quantidade de pontos no polígono e armazenados em p;
  // OBS: VALE PARA CONVEXO E NÃO CONVEXO
  for(int i = 0; i<n; i++){
    area += cross(arr[i], arr[(i+1)%n]);
  }
  return abs(area);
}

int internalCount(vector<pto> &arr, int n) {

  int area_2 = polygonArea2(arr, n);
  int boundPoints = totalBoundaryPolygon(arr,n);
  return (area_2 - boundPoints + 2)/2;
}
```

4.12. Polygon Area (3D)

```
#include <bits/stdc++.h>

using namespace std;

struct point{
  double x,y,z;
  void operator=(const point & b){
    x = b.x;
    y = b.y;
    z = b.z;
  }
};

point cross(point a, point b){
  point ret;
  ret.x = a.y*b.z - b.y*a.z;
  ret.y = a.z*b.x - a.x*b.z;
  ret.z = a.x*b.y - a.y*b.x;
  return ret;
}

int main(){
  int num;
  cin >> num;
  point v[num];
  for(int i=0; i<num; i++) cin >> v[i].x >> v[i].y >> v[i].z;

  point cur;
  cur.x = 0, cur.y = 0, cur.z = 0;

  for(int i=0; i<num; i++){
    point res = cross(v[i], v[(i+1)%num]);
    cur.x += res.x;
    cur.y += res.y;
    cur.z += res.z;
  }

  double ans = sqrt(cur.x*cur.x + cur.y*cur.y + cur.z*cur.z);

  double area = abs(ans);

  cout << fixed << setprecision(9) << area/2. << endl;
}
```

4.13. Polygon Area

```
double polygonArea(vector<int> &X, vector<int> &Y, int n) {
  int area = 0;
  int j = n - 1;
  for (int i = 0; i < n; i++) {
    area += (X[j] + X[i]) * (Y[j] - Y[i]);
    j = i;
  }
  return abs(area / 2.0);
}
```

4.14. Segment-Segment Intersection

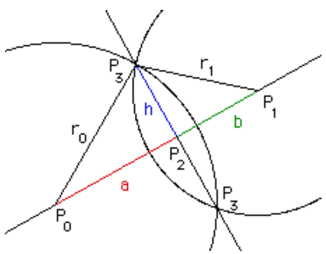
```
// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
int onSegment(Point p, Point q, Point r) {
  if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y <= max(p.y, r.y)
     && q.y >= min(p.y, r.y))
    return true;
  return false;
}
/* PODE SER RETIRADO
```

```
int onSegmentNotBorda(Point p, Point q, Point r) {
    if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) && q.y <= max(p.y, r.y)
     && q.y >= min(p.y, r.y))
         return true;
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y < max(p.y, r.y)
     && q.y > min(p.y, r.y))
         return true;
    return false;
}
*/
// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r) {
  int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);
  if (val == 0) return 0;  // colinear
  return (val > 0)? 1: 2; // clock or counterclock wise
}
// The main function that returns true if line segment 'p1p2'
// and 'q1q2' intersect.
int doIntersect(Point p1, Point p2, Point q1, Point q2) {
  // Find the four orientations needed for general and
  // special cases
  int o1 = orientation(p1, p2, q1);
  int o2 = orientation(p1, p2, q2);
  int o3 = orientation(q1, q2, p1);
  int o4 = orientation(q1, q2, p2);

  // General case
  if (o1 != o2 && o3 != o4) return 2;

/* PODE SER RETIRADO
  if(o1 == o2 && o2 == o3 && o3 == o4 && o4 == 0) {
    //INTERCEPTAM EM RETA
    if(onSegmentNotBorda(p1,q1,p2) || onSegmentNotBorda(p1,q2,p2)) return 1;
    if(onSegmentNotBorda(q1,p1,q2) || onSegmentNotBorda(q1,p2,q2)) return 1;
  }
*/
  // Special Cases (INTERCEPTAM EM PONTO)
  // p1, p2 and q1 are colinear and q1 lies on segment p1p2
  if (o1 == 0 && onSegment(p1, q1, p2)) return 2;
  // p1, p2 and q1 are colinear and q2 lies on segment p1p2
  if (o2 == 0 && onSegment(p1, q2, p2)) return 2;
  // q1, q2 and p1 are colinear and p1 lies on segment q1q2
  if (o3 == 0 && onSegment(q1, p1, q2)) return 2;
   // q1, q2 and p2 are colinear and p2 lies on segment q1q2
  if (o4 == 0 && onSegment(q1, p2, q2)) return 2;
  return false; // Doesn't fall in any of the above cases
}
// OBS: SE (C2/A2 == C1/A1) SÃO COLINEARES
```

4.15. Upper And Lower Hull

```
struct pto {
  double x, y;
  bool operator <(const pto &p) const {
    return x < p.x || (x == p.x && y < p.y);
    /* a impressao será em prioridade por mais a esquerda, mais
       abaixo, e antihorário pelo cross abaixo */
  }
};
double cross(const pto &O, const pto &A, const pto &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<pto> lower, upper;

vector<pto> convex_hull(vector<pto> &P) {
  int n = P.size(), k = 0;
  vector<pto> H(2 * n);
  // Sort points lexicographically
  sort(P.begin(), P.end());
  // Build lower hull
  for (int i = 0; i < n; ++i) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  // Build upper hull
  for (int i = n - 2, t = k + 1; i >= 0; i--) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  H.resize(k);
  /* o último ponto do vetor é igual ao primeiro, atente para isso
  as vezes é necessário mudar */

  int j = 1;
  lower.pb(H.front());
  while(H[j].x >= H[j-1].x) {
    lower.pb(H[j++]);
  }

  int l = H.size()-1;
  while(l >= j) {
    upper.pb(H[l--]);
  }
  upper.pb(H[l--]);

  return H;
}
```

4.16. Circle Circle Intersection



4.17. Circle Circle Intersection

```
/* circle_circle_intersection() *
 * Determine the points where 2 circles in a common plane intersect.
 *
 * int circle_circle_intersection(
 *                                  // center and radius of 1st circle
 *                                  double x0, double y0, double r0,
 *                                  // center and radius of 2nd circle
 *                                  double x1, double y1, double r1,
 *                                  // 1st intersection point
 *                                  double *xi, double *yi,
 *                                  // 2nd intersection point
 *                                  double *xi_prime, double *yi_prime)
 *
 * This is a public domain work. 3/26/2005 Tim Voght
 *
 */

int circle_circle_intersection(double x0, double y0, double r0, double x1,
                               double y1, double r1, double *xi, double *yi,
                               double *xi_prime, double *yi_prime) {
  double a, dx, dy, d, h, rx, ry;
  double x2, y2;

  /* dx and dy are the vertical and horizontal distances between
   * the circle centers.
   */
  dx = x1 - x0;
  dy = y1 - y0;

  /* Determine the straight-line distance between the centers. */
  // d = sqrt((dy*dy) + (dx*dx));
  d = hypot(dx, dy); // Suggested by Keith Briggs

  /* Check for solvability. */
  if (d > (r0 + r1)) {
    /* no solution. circles do not intersect. */
    return 0;
  }
  if (d < fabs(r0 - r1)) {
    /* no solution. one circle is contained in the other */
    return 0;
  }

  /* 'point 2' is the point where the line through the circle
   * intersection points crosses the line between the circle
   * centers.
   */

  /* Determine the distance from point 0 to point 2. */
  a = ((r0 * r0) - (r1 * r1) + (d * d)) / (2.0 * d);

  /* Determine the coordinates of point 2. */
  x2 = x0 + (dx * a / d);
  y2 = y0 + (dy * a / d);

  /* Determine the distance from point 2 to either of the
   * intersection points.
   */
  h = sqrt((r0 * r0) - (a * a));

  /* Now determine the offsets of the intersection points from
   * point 2.
   */
  rx = -dy * (h / d);
  ry = dx * (h / d);

  /* Determine the absolute intersection points. */
  *xi = x2 + rx;
  *xi_prime = x2 - rx;
  *yi = y2 + ry;
  *yi_prime = y2 - ry;

  return 1;
}
```

4.18. Struct Point And Line

```
int sgn(double x) {
    if(abs(x) < 1e-8)   return 0;
    return x > 0 ? 1 : -1;
}
inline double sqr(double x) {   return x * x;   }

struct Point {
    double x, y, z;
    Point() {};
    Point(double a, double b): x(a), y(b) {};
    Point (double x, double y, double z): x(x), y(y), z(z) {}

    void input() {  scanf(" %lf %lf", &x, &y);  };
    friend Point operator+(const Point &a, const Point &b) {
        return Point(a.x + b.x, a.y + b.y);
    }
    friend Point operator-(const Point &a, const Point &b) {
        return Point(a.x - b.x, a.y - b.y);
    }

    bool operator !=(const Point& a) const {
        return (x != a.x || y != a.y);
    }

    bool operator <(const Point &a) const{
      if(x == a.x)
        return y < a.y;
      return x < a.x;
    }

    double norm() {
        return sqrt(sqr(x) + sqr(y));
    }
};
double det(const Point &a, const Point &b) {
    return a.x * b.y - a.y * b.x;
}
double dot(const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y;
}
double dist(const Point &a, const Point &b) {
    return (a-b).norm();
}


struct Line {
    Point a, b;
    Line() {}
    Line(Point x, Point y): a(x), b(y) {};
};
```

```
double dis_point_segment(const Point p, const Point s, const Point t) {
    if(sgn(dot(p-s, t-s)) < 0)
      return (p-s).norm();
    if(sgn(dot(p-t, s-t)) < 0)
      return (p-t).norm();
    return abs(det(s-p, t-p) / dist(s, t));
}
```

5. Graphs

5.1. All Eulerian Path Or Tour

```
struct edge {
  int v, id;
  edge() {}
  edge(int v, int id) : v(v), id(id) {}
};

// The undirected + path and directed + tour wasn't tested in a problem.
// TEST AGAIN BEFORE SUBMITTING IT!
namespace graph {
  // Namespace which auxiliary funcions are defined.
  namespace detail {
    pair<bool, pair<int, int>> check_both_directed(const vector<vector<edge>> &adj, const vector<int> &in_degree) {
      // source and destination
      int src = -1, dest = -1;
      // adj[i].size() represents the out degree of an vertex
      for(int i = 0; i < adj.size(); i++) {
        if((int)adj[i].size() - in_degree[i] == 1) {
          if(src != -1)
            return make_pair(false, pair<int, int>());
          src = i;
        } else if((int)adj[i].size() - in_degree[i] == -1) {
           if(dest != -1)
            return make_pair(false, pair<int, int>());
          dest = i;
        } else if(abs((int)adj[i].size() - in_degree[i]) > 1)
          return make_pair(false, pair<int, int>());
      }

      if(src == -1 && dest == -1)
        return make_pair(true, pair<int, int>(src, dest));
      else if(src != -1 && dest != -1)
        return make_pair(true, pair<int, int>(src, dest));

      return make_pair(false, pair<int, int>());
    }

    /// Builds the path/tour for directed graphs.
    void build(const int u, vector<int> &tour, vector<vector<edge>> &adj, vector<bool> &used) {
      while(!adj[u].empty()) {
        const edge e = adj[u].back();
        if(!used[e.id]) {
          used[e.id] = true;
          adj[u].pop_back();
          build(e.v, tour, adj, used);
        } else
          adj[u].pop_back();
      }

      tour.push_back(u);
    }

    /// Auxiliary function to build the eulerian tour/path.
    vector<int> set_build(vector<vector<edge>> &adj, const int E, const int first) {
      vector<int> path;
      vector<bool> used(E + 3);

      build(first, path, adj, used);

      for(int i = 0; i < adj.size(); i++)
        // if there are some remaining edges, it's not possible to build the tour.
        if(adj[i].size())
          return vector<int>();

      reverse(path.begin(), path.end());
      return path;
    }
  }

  /// All vertices v should have in_degree[v] == out_degree[v]. It must not contain a specific
  /// start and end vertices.
  ///
  /// Time complexity: O(V * (log V) + E)
  bool has_euler_tour_directed(const vector<vector<edge>> &adj, const vector<int> &in_degree) {
    const pair<bool, pair<int, int>> aux = detail::check_both_directed(adj, in_degree);
    const bool valid = aux.first;
    const int src = aux.second.first;
    const int dest = aux.second.second;
    return (valid && src == -1 && dest == -1);
  }

  /// A directed graph has an eulerian path/tour if has:
  /// - One vertex v such that out_degree[v] - in_degree[v] == 1
  /// - One vertex v such that in_degree[v] - out_degree[v] == 1
  /// - The remaining vertices v such that in_degree[v] == out_degree[v]
  /// or
  /// - All vertices v such that in_degree[v] - out_degree[v] == 0 -> TOUR
  ///
  /// Returns a boolean value that indicates whether there's a path or not.
  /// If there's a valid path it also returns two numbers: the source and the destination.
  /// If the source and destination can be an arbitrary vertex it will return the pair (-1, -1)
  /// for the source and destination (it means the contains an eulerian tour).
  ///
  /// Time complexity: O(V + E)
  pair<bool, pair<int, int>> has_euler_path_directed(const vector<vector<edge>> &adj, const vector<int> &in_degree) {
    return detail::check_both_directed(adj, in_degree);
  }

  /// Returns the euler path. If the graph doesn't have an euler path it returns an empty vector.
  ///
  /// Time Complexity: O(V + E) for directed, O(V * log(V) + E) for undirected.
  /// Time Complexity: O(adj.size() + sum(adj[i].size()))
  vector<int> get_euler_path_directed(const int E, vector<vector<edge>> &adj, const vector<int> &in_degree) {
```

```
    const pair<bool, pair<int, int>> aux = has_euler_path_directed(adj, in_degree);
    const bool valid = aux.first;
    const int src = aux.second.first;
    const int dest = aux.second.second;

    if(!valid)
      return vector<int>();

    int first;
    if(src != -1)
      first = src;
    else {
      first = 0;
      while(adj[first].empty())
        first++;
    }

    return detail::set_build(adj, E, first);
  }

  /// Returns the euler tour. If the graph doesn't have an euler tour it returns an empty vector.
  ///
  /// Time Complexity: O(V + E)
  /// Time Complexity: O(adj.size() + sum(adj[i].size()))
  vector<int> get_euler_tour_directed(const int E, vector<vector<edge>> &adj, const vector<int> &in_degree) {
    const bool valid = has_euler_tour_directed(adj, in_degree);

    if(!valid)
      return vector<int>();

    int first = 0;
    while(adj[first].empty())
      first++;

    return detail::set_build(adj, E, first);
  }

  // The graph has a tour that passes to every edge exactly once and gets
  // back to the first edge on the tour.
  //
  // A graph with an euler path has zero odd degree vertex.
  //
  // Time Complexity: O(V)
  bool has_euler_tour_undirected(const vector<int> &degree) {
    for(int i = 0; i < degree.size(); i++)
      if(degree[i] & 1)
        return false;
    return true;
  }

  // The graph has a path that passes to every edge exactly once.
  // It doesn't necessarely gets back to the beginning.
  //
  // A graph with an euler path has two or zero (tour) odd degree vertices.
  //
  // Returns a pair with the startpoint/endpoint of the path.
  //
  // Time Complexity: O(V)
  pair<bool, pair<int, int>> has_euler_path_undirected(const vector<int> &degree) {
    vector<int> odd_degree;
    for(int i = 0; i < degree.size(); i++)
      if(degree[i] & 1)
        odd_degree.pb(i);

    if(odd_degree.size() == 0)
      return make_pair(true, make_pair(-1, -1));
    else if (odd_degree.size() == 2)
      return make_pair(true, make_pair(odd_degree.front(), odd_degree.back()));
    else
      return make_pair(false, pair<int, int>());
  }

  vector<int> get_euler_tour_undirected(const int E, const vector<int> &degree, vector<vector<edge>> &adj) {
    if(!has_euler_tour_undirected(degree))
      return vector<int>();

    int first = 0;
    while(adj[first].empty())
      first++;

    return detail::set_build(adj, E, first);
  }

  /// Returns the euler tour. If the graph doesn't have an euler tour it returns an empty vector.
  ///
  /// Time Complexity: O(V + E)
  /// Time Complexity: O(adj.size() + sum(adj[i].size()))
  vector<int> get_euler_path_undirected(const int E, const vector<int> &degree, vector<vector<edge>> &adj) {
    auto aux = has_euler_path_undirected(degree);
    const bool valid = aux.first;
    const int x = aux.second.first;
    const int y = aux.second.second;

    if(!valid)
      return vector<int>();

    int first;
    if(x != -1) {
      first = x;
      adj[x].emplace_back(y, E + 1);
      adj[y].emplace_back(x, E + 1);
    } else {
      first = 0;
      while(adj[first].empty())
        first++;
    }

    vector<int> ans = detail::set_build(adj, E, first);
    reverse(ans.begin(), ans.end());
    if(x != -1)
      ans.pop_back();
    return ans;
  }
};
```

5.2. Articulation Points

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

```cpp

void dfs_ap(const int u, const int p, const vector<vector<int>> &adj) {
  low[u] = disc[u] = cur_time++;
  int children = 0;

  for (const int v : adj[u]) {
    // DO NOT ADD PARALLEL EDGES
    if (disc[v] == 0) {
      ++children;
      dfs_ap(v, u, adj);

      low[u] = min(low[v], low[u]);
      if (p == -1 && children > 1)
        ap.emplace(u);
      if (p != -1 && low[v] >= disc[u])
        ap.emplace(u);
    } else if (v != p)
      low[u] = min(low[u], disc[v]);
  }
}

void init_ap(const int n) {
  cur_time = 1;
  ap = unordered_set<int>();
  low = vector<int>(n, 0);
  disc = vector<int>(n, 0);
}

/// THE GRAPH MUST BE UNDIRECTED!
///
/// Returns the vertices in which their removal disconnects the graph.
///
/// Time Complexity: O(V + E)
vector<int> articulation_points(const int indexed_from,
                                const vector<vector<int>> &adj) {
  init_ap(adj.size());
  vector<int> ans;
  for (int u = indexed_from; u < adj.size(); ++u) {
    if (disc[u] == 0)
      dfs_ap(u, -1, adj);
    if (ap.count(u))
      ans.emplace_back(u);
  }
  return ans;
}
}; // namespace graph
```

5.3. Bellman Ford

```cpp
struct edge {
  int src, dest, weight;
  edge() {}
  edge(int src, int dest, int weight) : src(src), dest(dest), weight(weight) {}

  bool operator<(const edge &a) const {
    return weight < a.weight;
  }
};

/// Works to find the shortest path with negative edges.
/// Also detects cycles.
///
/// Time Complexity: O(n * e)
/// Space Complexity: O(n)
bool bellman_ford(vector<edge> &edges, int src, int n)  {
  // n = qtd of vertices, E = qtd de arestas

  // To calculate the shortest path uncomment the line below
  // vector<int> dist(n, INF);

  // To check cycles uncomment the line below
  // vector<int> dist(n, 0);

  vector<int> pai(n, -1);
  int E = edges.size();

  dist[src] = 0;
  // Relax all edges n - 1 times.
  // A simple shortest path from src to any other vertex can have at-most n - 1 edges.
  for (int i = 1; i <= n - 1; i++) {
    for (int j = 0; j < E; j++) {
      int u = edges[j].src;
      int v = edges[j].dest;
      int weight = edges[j].weight;
      if (dist[u] != INF && dist[u] + weight < dist[v]) {
        dist[v] = dist[u] + weight;
        pai[v] = u;
      }
    }
  }

  // Check for NEGATIVE-WEIGHT CYCLES.
  // The above step guarantees shortest distances if graph doesn't contain negative weight cycle.
  // If we get a shorter path, then there is a cycle.
  bool is_cycle = false;
  int vert_in_cycle;
  for (int i = 0; i < E; i++) {
    int u = edges[i].src;
    int v = edges[i].dest;
    int weight = edges[i].weight;
    if (dist[u] != INF && dist[u] + weight < dist[v]) {
      is_cycle = true;
      pai[v] = u;
      vert_in_cycle = v;
    }
  }

  if(is_cycle) {
    for(int i = 0; i < n; i++)
      vert_in_cycle = pai[vert_in_cycle];

    vector<int> cycle;
    for(int v = vert_in_cycle; (v != vert_in_cycle || cycle.size() <= 1) ; v = pai[v])
      cycle.pb(v);

    reverse(cycle.begin(), cycle.end());

    for(int x: cycle) {
      cout << x + 1 << ' ';
    }
    cout << cycle.front() + 1 << endl;
    return true;
  } else
    return false;
}
```

5.4. Bipartite Check

```
/// Time Complexity: O(V + E)
bool is_bipartite(const int src, const vector<vector<int>> &adj) {
  vector<int> color(adj.size(), -1);
  queue<int> q;

  color[src] = 1;
  q.emplace(src);
  while (!q.empty()) {
    const int u = q.front();
    q.pop();

    for (const int v : adj[u]) {
      if (color[v] == color[u])
        return false;
      else if (color[v] == -1) {
        color[v] = !color[u];
        q.emplace(v);
      }
    }
  }
  return true;
}
```

5.5. Bridges

```
namespace graph {
int cur_time = 1;
vector<pair<int, int>> bg;
vector<int> disc;
vector<int> low;
vector<int> cycle;

void dfs_bg(const int u, int p, const vector<vector<int>> &adj) {
  low[u] = disc[u] = cur_time++;
  for (const int v : adj[u]) {
    if (v == p) {
      // checks parallel edges
      // IT'S BETTER TO REMOVE THEM!
      p = -1;
      continue;
    } else if (disc[v] == 0) {
      dfs_bg(v, u, adj);
      low[u] = min(low[u], low[v]);
      if (low[v] > disc[u])
        bg.emplace_back(u, v);
    } else
      low[u] = min(low[u], disc[v]);
    // checks if the vertex u belongs to a cycle
    cycle[u] |= (disc[u] >= low[v]);
  }
}

void init_bg(const int n) {
  cur_time = 1;
  bg = vector<pair<int, int>>();
  disc = vector<int>(n, 0);
  low = vector<int>(n, 0);
  cycle = vector<int>(n, 0);
}

/// THE GRAPH MUST BE UNDIRECTED!
///
/// Returns the edges in which their removal disconnects the graph.
///
/// Time Complexity: O(V + E)
vector<pair<int, int>> bridges(const int indexed_from,
                               const vector<vector<int>> &adj) {
  init_bg(adj.size());
  for (int u = indexed_from; u < adj.size(); ++u)
    if (disc[u] == 0)
      dfs_bg(u, -1, adj);

  return bg;
}
} // namespace graph
```

5.6. Centroid Decomposition

```
class Centroid {
private:
  int it = 1, _vertex;
  vector<int> vis, used, sub, _parent;
  vector<vector<int>> _tree;

  int dfs(const int u, int &cnt, const vector<vector<int>> &adj) {
    vis[u] = it;
    ++cnt;
    sub[u] = 1;
    for (const int v : adj[u])
      if (vis[v] != it && !used[v])
        sub[u] += dfs(v, cnt, adj);
    return sub[u];
  }

  int find_centroid(const int u, const int cnt,
                    const vector<vector<int>> &adj) {
    vis[u] = it;

    bool valid = true;
    int max_sub = -1;
    for (const int v : adj[u]) {
      if (vis[v] == it || used[v])
        continue;
      if (sub[v] > cnt / 2)
        valid = false;
      if (max_sub == -1 || sub[v] > sub[max_sub])
        max_sub = v;
    }

    if (valid && cnt - sub[u] <= cnt / 2)
      return u;
    return find_centroid(max_sub, cnt, adj);
  }

  int find_centroid(const int u, const vector<vector<int>> &adj) {
    // counts the number of vertices
    int cnt = 0;

    // set up sizes and nodes in current subtree
    dfs(u, cnt, adj);
    ++it;

    const int ctd = find_centroid(u, cnt, adj);
```

```
    ++it;
    used[ctd] = true;
    return ctd;
  }

  int build_tree(const int u, const vector<vector<int>> &adj) {
    const int ctd = find_centroid(u, adj);

    for (const int v : adj[ctd]) {
      if (used[v])
        continue;
      const int ctd_v = build_tree(v, adj);
      _tree[ctd].emplace_back(ctd_v);
      _tree[ctd_v].emplace_back(ctd);
      _parent[ctd_v] = ctd;
    }

    return ctd;
  }

  void allocate(const int n) {
    vis.resize(n);
    _parent.resize(n, -1);
    sub.resize(n);
    used.resize(n);
    _tree.resize(n);
  }

public:
  /// Constructor that creates the centroid tree.
  ///
  /// Time Complexity: O(n * log(n))
  Centroid(const int root_idx, const vector<vector<int>> &adj) {
    allocate(adj.size());
    _vertex = build_tree(root_idx, adj);
  }

  /// Returns the centroid of the whole tree.
  int vertex() { return _vertex; }

  int parent(const int u) { return _parent[u]; }

  vector<vector<int>> tree() { return _tree; };
};
```

5.7. De Bruijn Sequence

```
// We can solve this problem by constructing a directed graph with
// k^(n-1) nodes with each node having k outgoing edges_order. Each node
// corresponds to a string of size n-1. Every edge corresponds to one of the k
// characters in A and adds that character to the starting string. For example,
// if n=3 and k=2, then we construct the following graph:

//             - 1 ->  (01)  - 1 ->
//            /          ^  |        \
// 0 -> (00)            1  0          (11) <- 1
//            \          |  v        /
//            <- 0 -   (10)  <- 0 -

// The node '01' is connected to node '11' through edge '1', as adding '1' to
// '01' (and removing the first character) gives us '11'.
//
// We can observe that every node in this graph has equal in-degree and
// out-degree, which means that a Eulerian circuit exists in this graph.

namespace graph {
namespace detail {
// Finding an valid eulerian path
void dfs(const string &node, const string &alphabet, set<string> &vis,
         string &edges_order) {
  for (char c : alphabet) {
    string nxt = node + c;
    if (vis.count(nxt))
      continue;

    vis.insert(nxt);
    nxt.erase(nxt.begin());
    dfs(nxt, alphabet, vis, edges_order);
    edges_order += c;
  }
}
}; // namespace detail

// Returns a string in which every string of the alphabet of size n appears in
// the resulting string exactly once.
//
// Time Complexity: O(alphabet.size() ^ n * log2(alphabet.size() ^ n))
string de_bruijn(const int n, const string &alphabet) {
  set<string> vis;
  string edges_order;

  string starting_node = string(n - 1, alphabet.front());
  detail::dfs(starting_node, alphabet, vis, edges_order);

  return edges_order + starting_node;
}
}; // namespace graph
```

5.8. Diameter In Tree

```
From any vertex, X find the furthermost vertex A from X. After that, return
    the distance from vertex A from the furthermost vertex B from A.
```

5.9. Dijkstra + Dij Graph

```
/// Works also with 1-indexed graphs.
class Dijkstra {
private:
  static constexpr int INF = 2e18;
  bool CREATE_GRAPH = false;
  int src;
  int n;
  vector<int> _dist;
  vector<vector<int>> parent;

private:
  void _compute(const int src, const vector<vector<pair<int, int>>> &adj) {
    _dist.resize(this->n, INF);
    vector<bool> vis(this->n, false);

    if (CREATE_GRAPH) {
      parent.resize(this->n);

      for (int i = 0; i < this->n; i++)
```

```
      parent[i].emplace_back(i);
    }

    priority_queue<pair<int, int>, vector<pair<int, int>>,
                   greater<pair<int, int>>>
        pq;
    pq.emplace(0, src);
    _dist[src] = 0;

    while (!pq.empty()) {
      int u = pq.top().second;
      pq.pop();
      if (vis[u])
        continue;
      vis[u] = true;

      for (const pair<int, int> &x : adj[u]) {
        int v = x.first, w = x.second;

        if (_dist[u] + w < _dist[v]) {
          _dist[v] = _dist[u] + w;
          pq.emplace(_dist[v], v);
          if (CREATE_GRAPH) {
            parent[v].clear();
            parent[v].emplace_back(u);
          }
        } else if (CREATE_GRAPH && _dist[u] + w == _dist[v]) {
          parent[v].emplace_back(u);
        }
      }
    }
  }

  vector<vector<int>> gen_dij_graph(const int dest) {
    vector<vector<int>> dijkstra_graph(this->n);
    vector<bool> vis(this->n, false);
    queue<int> q;

    q.emplace(dest);
    while (!q.empty()) {
      int v = q.front();
      q.pop();

      for (const int u : parent[v]) {
        if (u == v)
          continue;
        dijkstra_graph[u].emplace_back(v);
        if (!vis[u]) {
          q.emplace(u);
          vis[u] = true;
        }
      }
    }
    return dijkstra_graph;
  }

  vector<int> gen_min_path(const int dest) {
    vector<int> path, prev(this->n, -1), d(this->n, INF);
    queue<int> q;

    q.emplace(dest);
    d[dest] = 0;

    while (!q.empty()) {
      int v = q.front();
      q.pop();

      for (const int u : parent[v]) {
        if (u == v)
          continue;
        if (d[v] + 1 < d[u]) {
          d[u] = d[v] + 1;
          prev[u] = v;
          q.emplace(u);
        }
      }
    }

    int cur = this->src;
    while (cur != -1) {
      path.emplace_back(cur);
      cur = prev[cur];
    }

    return path;
  }

public:
  /// Allows creation of dijkstra graph and getting the minimum path.
  Dijkstra(const int src, const bool create_graph,
           const vector<vector<pair<int, int>>> &adj)
      : n(adj.size()), src(src), CREATE_GRAPH(create_graph) {
    this->_compute(src, adj);
  }

  /// Constructor that computes only the Dijkstra minimum path from src.
  ///
  /// Time Complexity: O(E log V)
  Dijkstra(const int src, const vector<vector<pair<int, int>>> &adj)
      : n(adj.size()), src(src) {
    this->_compute(src, adj);
  }

  /// Returns the Dijkstra graph of the graph.
  ///
  /// Time Complexity: O(V)
  vector<vector<int>> dij_graph(const int dest) {
    assert(CREATE_GRAPH);
    return gen_dij_graph(dest);
  }

  /// Returns the vertices present in a path from src to dest with
  /// minimum cost and a minimum length.
  ///
  /// Time Complexity: O(V)
  vector<int> min_path(const int dest) {
    assert(CREATE_GRAPH);
    return gen_min_path(dest);
  }

  /// Returns the distance from src to dest.
  int dist(const int dest) {
    assert(0 <= dest), assert(dest < n);
    return _dist[dest];
  }
};
```

5.10. Dinic

```cpp
class Dinic {
  struct Edge {
    const int v;
    // capacity (maximum flow) of the edge
    // if it is a reverse edge then its capacity should be equal to 0
    const int cap;
    // current flow of the graph
    int flow = 0;
    Edge(const int v, const int cap) : v(v), cap(cap) {}
  };

private:
  static constexpr int INF = 2e18;
  bool COMPUTED = false;
  int _max_flow;
  vector<Edge> edges;
  // holds the indexes of each edge present in each vertex.
  vector<vector<int>> adj;
  const int n;
  // src will be always 0 and sink n+1.
  const int src, sink;
  vector<int> level, ptr;

private:
  vector<vector<int>> _flow_table() {
    vector<vector<int>> table(n, vector<int>(n, 0));
    for (int u = 0; u <= sink; ++u)
      for (const int idx : adj[u])
        // checks if it's not a reverse edge
        if (!(idx & 1))
          table[u][edges[idx].v] += edges[idx].flow;
    return table;
  }

  /// Algorithm: Greedily all vertices from the matching will be added and,
  /// after that, edges in which one of the vertices is not covered will also be
  /// added to the answer.
  vector<pair<int, int>> _min_edge_cover() {
    vector<bool> covered(n, false);
    vector<pair<int, int>> ans;
    for (int u = 1; u < sink; ++u) {
      for (const int idx : adj[u]) {
        const Edge &e = edges[idx];
        // ignore if it is a reverse edge or an edge linked to the sink
        if (idx & 1 || e.v == sink)
          continue;
        if (e.flow == e.cap) {
          ans.emplace_back(u, e.v);
          covered[u] = covered[e.v] = true;
          break;
        }
      }
    }

    for (int u = 1; u < sink; ++u) {
      for (const int idx : adj[u]) {
        const Edge &e = edges[idx];
        if (idx & 1 || e.v == sink)
          continue;
        if (e.flow < e.cap && (!covered[u] || !covered[e.v])) {
          ans.emplace_back(u, e.v);
          covered[u] = covered[e.v] = true;
        }
      }
    }
    return ans;
  }

  /// Algorithm: Takes the complement of the vertex cover.
  vector<int> _max_ind_set(const int max_left) {
    const vector<int> mvc = _min_vertex_cover(max_left);
    vector<bool> contains(n);
    for (const int v : mvc)
      contains[v] = true;
    vector<int> ans;
    for (int i = 1; i < sink; ++i)
      if (!contains[i])
        ans.emplace_back(i);
    return ans;
  }

  void dfs_vc(const int u, vector<bool> &vis, const bool left,
              const vector<vector<int>> &paths) {
    vis[u] = true;
    for (const int idx : adj[u]) {
      const Edge &e = edges[idx];
      if (vis[e.v])
        continue;
      // saturated edges goes from right to left
      if (left && paths[u][e.v] == 0)
        dfs_vc(e.v, vis, left ^ 1, paths);
      // non-saturated edges goes from left to right
      else if (!left && paths[e.v][u] == 1)
        dfs_vc(e.v, vis, left ^ 1, paths);
    }
  }

  /// Algorithm: The edges that belong to the Matching M will go from right to
  /// left, all other edges will go from left to right. A DFS will be run
  /// starting at all left vertices that are not incident to edges in M. Some
  /// vertices of the graph will become visited during this DFS and some
  /// not-visited. To get minimum vertex cover all visited right
  /// vertices of M will be taken, and all not-visited left vertices of M.
  /// Source: codeforces.com/blog/entry/17534?#comment-223759
  vector<int> _min_vertex_cover(const int max_left) {
    vector<bool> vis(n, false), saturated(n, false);
    const auto paths = flow_table();

    for (int i = 1; i <= max_left; ++i) {
      for (int j = max_left + 1; j < sink; ++j)
        if (paths[i][j] > 0) {
          saturated[i] = saturated[j] = true;
          break;
        }
      if (!saturated[i] && !vis[i])
        dfs_vc(i, vis, 1, paths);
    }

    vector<int> ans;
    for (int i = 1; i <= max_left; ++i)
      if (saturated[i] && !vis[i])
        ans.emplace_back(i);

    for (int i = max_left + 1; i < sink; ++i)
      if (saturated[i] && vis[i])
        ans.emplace_back(i);

    return ans;
```

```
  }

  void dfs_build_path(const int u, vector<int> &path,
                      vector<vector<int>> &table, vector<vector<int>> &ans,
                      const vector<vector<int>> &adj) {
    path.emplace_back(u);

    if (u == sink) {
      ans.emplace_back(path);
      return;
    }

    for (const int v : adj[u]) {
      if (table[u][v]) {
        --table[u][v];
        dfs_build_path(v, path, table, ans, adj);
        return;
      }
    }
  }

  /// Algorithm: Run DFS's from the source and gets the paths when possible.
  vector<vector<int>> _compute_all_paths(const vector<vector<int>> &adj) {
    vector<vector<int>> table = flow_table();
    vector<vector<int>> ans;
    ans.reserve(_max_flow);

    for (int i = 0; i < _max_flow; i++) {
      vector<int> path;
      path.reserve(n);
      dfs_build_path(src, path, table, ans, adj);
    }

    return ans;
  }

  /// Algorithm: Find the set of vertices that are reachable from the source in
  /// the residual graph. All edges which are from a reachable vertex to
  /// non-reachable vertex are minimum cut edges.
  /// Source: geeksforgeeks.org/minimum-cut-in-a-directed-graph
  pair<int, vector<pair<int, int>>> _min_cut() {
    // checks if there's an edge from i to j.
    vector<vector<int>> mat_adj(n, vector<int>(n, 0));
    // checks if if the residual capacity is greater than 0
    vector<vector<bool>> residual(n, vector<bool>(n, 0));
    for (int u = 0; u <= sink; ++u)
      for (const int idx : adj[u])
        // checks if it's not a reverse edge
        if (!(idx & 1)) {
          mat_adj[u][edges[idx].v] = edges[idx].cap;
          // checks if its residual capacity is greater than zero.
          if (edges[idx].flow < edges[idx].cap)
            residual[u][edges[idx].v] = true;
        }

    vector<bool> vis(n);
    queue<int> q;

    q.emplace(src);
    vis[src] = true;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (int v = 0; v < n; ++v)
        if (residual[u][v] && !vis[v]) {
          q.emplace(v);
          vis[v] = true;
        }
    }

    int weight = 0;
    vector<pair<int, int>> cut;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        if (vis[i] && !vis[j])
          // if there's an edge from i to j.
          if (mat_adj[i][j] > 0) {
            weight += mat_adj[i][j];
            cut.emplace_back(i, j);
          }

    return make_pair(weight, cut);
  }

  void _add_edge(const int u, const int v, const int cap) {
    adj[u].emplace_back(edges.size());
    edges.emplace_back(v, cap);
    // adding reverse edge
    adj[v].emplace_back(edges.size());
    edges.emplace_back(u, 0);
  }

  bool bfs_flow() {
    queue<int> q;
    memset(level.data(), -1, sizeof(*level.data()) * level.size());
    q.emplace(src);
    level[src] = 0;
    while (!q.empty()) {
      const int u = q.front();
      q.pop();
      for (const int idx : adj[u]) {
        const Edge &e = edges[idx];
        if (e.cap == e.flow || level[e.v] != -1)
          continue;
        level[e.v] = level[u] + 1;
        q.emplace(e.v);
      }
    }
    return (level[sink] != -1);
  }

  int dfs_flow(const int u, const int cur_flow) {
    if (u == sink)
      return cur_flow;

    for (int &idx = ptr[u]; idx < adj[u].size(); ++idx) {
      Edge &e = edges[adj[u][idx]];
      if (level[u] + 1 != level[e.v] || e.cap == e.flow)
        continue;
      const int flow = dfs_flow(e.v, min(e.cap - e.flow, cur_flow));
      if (flow == 0)
        continue;
      e.flow += flow;
      edges[adj[u][idx] ^ 1].flow -= flow;
      return flow;
    }
    return 0;
  }
```

```
  int compute() {
    int ans = 0;
    while (bfs_flow()) {
      memset(ptr.data(), 0, sizeof(*ptr.data()) * ptr.size());
      while (const int cur = dfs_flow(src, INF))
        ans += cur;
    }
    return ans;
  }

  void check_computed() {
    if (!COMPUTED) {
      COMPUTED = true;
      this->_max_flow = compute();
    }
  }

public:
  /// Constructor that makes assignments and allocations.
  ///
  /// Time Complexity: O(V)
  Dinic(const int n) : n(n + 2), src(0), sink(n + 1) {
    assert(n >= 0);

    adj.resize(this->n);
    level.resize(this->n);
    ptr.resize(this->n);
  }

  /// Returns the edges from the minimum edge cover of the graph.
  /// A minimum edge cover represents a set of edges such that each vertex
  /// present in the graph is linked to at least one edge from this set.
  ///
  /// Time Complexity: O(V + E)
  vector<pair<int, int>> min_edge_cover() {
    this->check_computed();
    return this->_min_edge_cover();
  }

  /// Returns the maximum independent set for the graph.
  /// An independent set represents a set of vertices such that they're not
  /// adjacent to each other.
  /// It is equal to the complement of the minimum vertex cover.
  ///
  /// Time Complexity: O(V + E)
  vector<int> max_ind_set(const int max_left) {
    this->check_computed();
    return this->_max_ind_set(max_left);
  }

  /// Returns the minimum vertex cover of a bipartite graph.
  /// A minimum vertex cover represents a set of vertices such that each edge of
  /// the graph is incident to at least one vertex of the graph.
  /// Pass the maximum index of a vertex on the left side as an argument.
  ///
  /// Time Complexity: O(V + E)
  vector<int> min_vertex_cover(const int max_left) {
    this->check_computed();
    return this->_min_vertex_cover(max_left);
  }

  /// Computes all paths from src to sink.
  /// Add all edges from the original graph. Its weights should be equal to the
  /// number of edges between the vertices. Pass the adjacency list with
  /// repeated vertices if there are multiple edges.
  ///
  /// Time Complexity: O(max_flow*V + E)
  vector<vector<int>> compute_all_paths(const vector<vector<int>> &adj) {
    this->check_computed();
    return this->_compute_all_paths(adj);
  }

  /// Returns the weight and the edges present in the minimum cut of the graph.
  /// A minimum cut represents a set of edges with minimum weight such that
  /// after removing these edges, it disconnects the graph. If the graph is
  /// undirected you can safely add edges in both directions. It doesn't work
  /// with parallel edges, it's required to merge them.
  ///
  /// Time Complexity: O(V^2 + E)
  pair<int, vector<pair<int, int>>> min_cut() {
    this->check_computed();
    return this->_min_cut();
  }

  /// Returns a table with the flow values for each pair of vertices.
  ///
  /// Time Complexity: O(V^2 + E)
  vector<vector<int>> flow_table() {
    this->check_computed();
    return this->_flow_table();
  }

  /// Adds a directed edge between u and v and its reverse edge.
  ///
  /// Time Complexity: O(1);
  void add_to_sink(const int u, const int cap) {
    assert(!COMPUTED);
    assert(src <= u), assert(u < sink);
    this->_add_edge(u, sink, cap);
  }

  /// Adds a directed edge between u and v and its reverse edge.
  ///
  /// Time Complexity: O(1);
  void add_to_src(const int v, const int cap) {
    assert(!COMPUTED);
    assert(src < v), assert(v <= sink);
    this->_add_edge(src, v, cap);
  }

  /// Adds a directed edge between u and v and its reverse edge.
  ///
  /// Time Complexity: O(1);
  void add_edge(const int u, const int v, const int cap) {
    assert(!COMPUTED);
    assert(src <= u), assert(u <= sink);
    this->_add_edge(u, v, cap);
  }

  /// Computes the maximum flow for the network.
  ///
  /// Time Complexity: O(V^2*E) or O(E*sqrt(V)) for matching.
  int max_flow() {
    this->check_computed();
    return this->_max_flow;
  }
};
```

5.11. Dsu

```
class DSU {
public:
  vector<int> root;
  vector<int> sz;

  DSU(int n) {
    this->root.resize(n + 1);
    iota(this->root.begin(), this->root.begin() + n + 1, 0ll);
    this->sz.resize(n + 1, 1);
  }

  int Find(int x) {
    if (this->root[x] == x)
      return x;
    return this->root[x] = this->Find(this->root[x]);
  }

  bool Union(int p, int q) {
    p = this->Find(p), q = this->Find(q);

    if (p == q)
      return false;

    if (this->sz[p] > this->sz[q]) {
      this->root[q] = p;
      this->sz[p] += this->sz[q];
    } else {
      this->root[p] = q;
      this->sz[q] += this->sz[p];
    }

    return true;
  }
};
```

5.12. Floyd Warshall

```
/// Put n = n + 1 for 1 based.
void floyd_warshall(const int n) {
  // OBS: Always assign adj[i][i] = 0.
  for (int i = 0; i < n; i++)
    adj[i][i] = 0;

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
}
```

5.13. Functional Graph

```
// Based on:
    http://maratona.ic.unicamp.br/MaratonaVerao2020/lecture-b/20200122.pdf

class Functional_Graph {
 // FOR DIRECTED GRAPH
 private:
  void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis) {
    int id_cycle = cycle_cnt++;
    int cur_id = 0;
    this->first[id_cycle] = u;

    while(!vis[u]) {
      vis[u] = true;

      this->cycle[id_cycle].push_back(u);

      this->in_cycle[u] = true;
      this->cycle_id[u] = id_cycle;
      this->id_in_cycle[u] = cur_id;
      this->near_in_cycle[u] = u;
      this->id_near_cycle[u] = id_cycle;
      this->cycle_dist[u] = 0;

      u = nxt[u];
      cur_id++;
    }
  }

  // Time Complexity: O(V)
  void build(int n, int indexed_from, vector<int> &nxt, vector<int>
    &in_degree) {
    queue<int> q;
    vector<bool> vis(n + indexed_from);
    for(int i = indexed_from; i < n + indexed_from; i++) {
      if(in_degree[i] == 0) {
        q.push(i);
        vis[i] = true;
      }
    }

    vector<int> process_order;
    process_order.reserve(n + indexed_from);
    while(!q.empty()) {
      int u = q.front();
      q.pop();

      process_order.push_back(u);

      if(--in_degree[nxt[u]] == 0) {
        q.push(nxt[u]);
        vis[nxt[u]] = true;
      }
    }

    int cycle_cnt = 0;
    for(int i = indexed_from; i < n + indexed_from; i++)
      if(!vis[i])
        compute_cycle(i, nxt, vis);

    for(int i = (int)process_order.size() - 1; i >= 0; i--) {
      int u = process_order[i];

      this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
      this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
      this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
    }
  }

  void allocate(int n, int indexed_from) {
    this->cycle.resize(n + indexed_from);
    this->first.resize(n + indexed_from);
```

```cpp
    this->in_cycle.resize(n + indexed_from, false);
    this->cycle_id.resize(n + indexed_from, -1);
    this->id_in_cycle.resize(n + indexed_from, -1);
    this->near_in_cycle.resize(n + indexed_from);
    this->id_near_cycle.resize(n + indexed_from);
    this->cycle_dist.resize(n + indexed_from);
  }

 public:
  Functional_Graph(int n, int indexed_from, vector<int> &nxt, vector<int> &in_degree) {
    this->allocate(n, indexed_from);
    this->build(n, indexed_from, nxt, in_degree);
  }

  // THE CYCLES ARE ALWAYS INDEXED BY ZERO!

  // number of cycles
  int cycle_cnt = 0;
  // Vertices present in the i-th cycle.
  vector<vector<int>> cycle;
  // first vertex of the i-th cycle
  vector<int> first;

  // The i-th vertex is present in any cycle?
  vector<bool> in_cycle;
  // id of the cycle that the vertex belongs. -1 if it doesn't belong to any cycle.
  vector<int> cycle_id;
  // Represents the id of the cycle of the i-th vertex. -1 if it doesn't belong to any cycle.
  vector<int> id_in_cycle;
  // Represents the id of the nearest vertex present in a cycle.
  vector<int> near_in_cycle;
  // Represents the id of the nearest cycle.
  vector<int> id_near_cycle;
  // Distance to the nearest cycle.
  vector<int> cycle_dist;
  // Represent the id of the component of the vertex.
  // Equal to id_near_cycle
  vector<int> &comp = id_near_cycle;
};

class Functional_Graph {
 // FOR UNDIRECTED GRAPH
 private:
  void compute_cycle(int u, vector<int> &nxt, vector<bool> &vis, vector<vector<int>> &adj) {
    int id_cycle = cycle_cnt++;
    int cur_id = 0;
    this->first[id_cycle] = u;

    while(!vis[u]) {
      vis[u] = true;

      this->cycle[id_cycle].push_back(u);
      nxt[u] = find_nxt(u, vis, adj);
      if(nxt[u] == -1)
        nxt[u] = this->first[id_cycle];

      this->in_cycle[u] = true;
      this->cycle_id[u] = id_cycle;
      this->id_in_cycle[u] = cur_id;
      this->near_in_cycle[u] = u;
      this->id_near_cycle[u] = id_cycle;
      this->cycle_dist[u] = 0;

      u = nxt[u];
      cur_id++;
    }
  }

  int find_nxt(int u, vector<bool> &vis, vector<vector<int>> &adj) {
    for(int v: adj[u])
      if(!vis[v])
        return v;
    return -1;
  }

  // Time Complexity: O(V + E)
  void build(int n, int indexed_from, vector<int> &degree, vector<vector<int>> &adj) {
    queue<int> q;
    vector<bool> vis(n + indexed_from, false);
    vector<int> nxt(n + indexed_from);
    for(int i = indexed_from; i < n + indexed_from; i++) {
      if(adj[i].size() == 1) {
        q.push(i);
        vis[i] = true;
      }
    }

    vector<int> process_order;
    process_order.reserve(n + indexed_from);
    while(!q.empty()) {
      int u = q.front();
      q.pop();

      process_order.push_back(u);

      nxt[u] = find_nxt(u, vis, adj);
      if(--degree[nxt[u]] == 1) {
        q.push(nxt[u]);
        vis[nxt[u]] = true;
      }
    }

    int cycle_cnt = 0;
    for(int i = indexed_from; i < n + indexed_from; i++)
      if(!vis[i])
        compute_cycle(i, nxt, vis, adj);

    for(int i = (int)process_order.size() - 1; i >= 0; i--) {
      int u = process_order[i];

      this->near_in_cycle[u] = this->near_in_cycle[nxt[u]];
      this->id_near_cycle[u] = this->id_near_cycle[nxt[u]];
      this->cycle_dist[u] = this->cycle_dist[nxt[u]] + 1;
    }
  }

  void allocate(int n, int indexed_from) {
    this->cycle.resize(n + indexed_from);
    this->first.resize(n + indexed_from);

    this->in_cycle.resize(n + indexed_from, false);
    this->cycle_id.resize(n + indexed_from, -1);
    this->id_in_cycle.resize(n + indexed_from, -1);
    this->near_in_cycle.resize(n + indexed_from);
    this->id_near_cycle.resize(n + indexed_from);
```

```
    this->cycle_dist.resize(n + indexed_from);
  }

 public:
  Functional_Graph(int n, int indexed_from, vector<int> degree,
     vector<vector<int>> &adj) {
    this->allocate(n, indexed_from);
    this->build(n, indexed_from, degree, adj);
  }

  // THE CYCLES ARE ALWAYS INDEXED BY ZERO!

  // number of cycles
  int cycle_cnt = 0;
  // Vertices present in the i-th cycle.
  vector<vector<int>> cycle;
  // first vertex of the i-th cycle
  vector<int> first;

  // The i-th vertex is present in any cycle?
  vector<bool> in_cycle;
  // id of the cycle that the vertex belongs. -1 if it doesn't belong to any
     cycle.
  vector<int> cycle_id;
  // Represents the id of the cycle of the i-th vertex. -1 if it doesn't
     belong to any cycle.
  vector<int> id_in_cycle;
  // Represents the id of the nearest vertex present in a cycle.
  vector<int> near_in_cycle;
  // Represents the id of the nearest cycle.
  vector<int> id_near_cycle;
  // Distance to the nearest cycle.
  vector<int> cycle_dist;
  // Represent the id of the component of the vertex.
  // Equal to id_near_cycle
  vector<int> &comp = id_near_cycle;
};
```

5.14. Girth (Shortest Cycle In A Graph)

```
int bfs(const int src) {
  vector<int> dist(MAXN, INF);
  queue<pair<int, int>> q;

  q.emplace(src, -1);
  dist[src] = 0;

  int ans = INF;
  while (!q.empty()) {
    pair<int, int> aux = q.front();
    const int u = aux.first, p = aux.second;
    q.pop();

    for (const int v : adj[u]) {
      if (v == p)
        continue;
      if (dist[v] < INF)
        ans = min(ans, dist[u] + dist[v] + 1);
      else {
        dist[v] = dist[u] + 1;
        q.emplace(v, u);
      }
    }
  }

  return ans;
}

/// Returns the shortest cycle in the graph
///
/// Time Complexity: O(V^2)
int get_girth(const int n) {
  int ans = INF;
  for (int u = 1; u <= n; u++)
    ans = min(ans, bfs(u));
  return ans;
}
```

5.15. Hld

```
class HLD {
private:
  int n;
  // number of nodes below the i-th node
  vector<int> sz;

private:
  void allocate() {
    // this->id_in_tree.resize(this->n + 1, -1);
    this->chain_head.resize(this->n + 1, -1);
    this->chain_id.resize(this->n + 1, -1);
    this->sz.resize(this->n + 1);
    this->parent.resize(this->n + 1, -1);
    // this->id_in_chain.resize(this->n + 1, -1);
    // this->chain_size.resize(this->n + 1);
  }

  int get_sz(const int u, const int p, const vector<vector<int>> &adj) {
    this->sz[u] = 1;
    for (const int v : adj[u]) {
      if (v == p)
        continue;
      this->sz[u] += this->get_sz(v, u, adj);
    }
    return this->sz[u];
  }

  void dfs(const int u, const int id, const int p,
           const vector<vector<int>> &adj, int &nidx) {
    // this->id_in_tree[u] = nidx++;
    this->chain_id[u] = id;
    // this->id_in_chain[u] = chain_size[id]++;
    this->parent[u] = p;

    if (this->chain_head[id] == -1)
      this->chain_head[id] = u;

    int maxx = -1, idx = -1;
    for (const int v : adj[u]) {
      if (v == p)
        continue;
      if (sz[v] > maxx) {
        maxx = sz[v];
        idx = v;
      }
    }

    if (idx != -1)
```

```cpp
      this->dfs(idx, id, u, adj, nidx);

    for (const int v : adj[u]) {
      if (v == idx || v == p)
        continue;
      this->dfs(v, this->number_of_chains++, u, adj, nidx);
    }
  }

  void build(const int root_idx, const vector<vector<int>> &adj) {
    this->get_sz(root_idx, -1, adj);
    int nidx = 0;
    this->dfs(root_idx, 0, -1, adj, nidx);
  }

  // int _compute(const int u, Seg_Tree &st) {
  //   int ans = 0;
  //   for (int v = u; v != -1; v = parent[chain_head[chain_id[v]]]) {
  //     // change here
  //     ans += st.query(id_in_tree[chain_head[chain_id[v]]], id_in_tree[v]);
  //   }
  //   return ans;
  // }

public:
  /// Builds the chains.
  ///
  /// Time Complexity: O(n)
  HLD(const int root_idx, const vector<vector<int>> &adj) : n(adj.size()) {
    allocate();
    build(root_idx, adj);
  }

  /// Computes the paths using segment tree.
  /// Uncomment id_in_tree!!!
  ///
  /// Time Complexity: O(log^2(n))
  // int compute(const int u, Seg_Tree &st) { return _compute(u, st); }

  // TAKE CARE, YOU MAY GET MLE!!!
  // the chains are indexed from 0
  int number_of_chains = 1;
  // topmost node of the chain
  vector<int> chain_head;
  // id of the node based on the order of the dfs (indexed by 0)
  // vector<int> id_in_tree;
  // id of the i-th node in his chain
  // vector<int> id_in_chain;
  // id of the chain that the i-th node belongs
  vector<int> chain_id;
  // size of the i-th chain
  // vector<int> chain_size;
  // parent of the i-th node, -1 for root
  vector<int> parent;
};
```

5.16. Hungarian

```cpp
/// Returns a vector p of size n, where p[i] is the match for i
/// and the minimum cost.
///
/// Code copied from:
///
    github.com/gabrielpessoa1/Biblioteca-Maratona/blob/master/code/Graph/Hungarian.cpp
///
/// Time Complexity: O(n^2 * m)
pair<vector<int>, int> solve(const vector<vector<int>> &matrix) {
  const int n = matrix.size();
  if (n == 0)
    return {vector<int>(), 0};
  const int m = matrix[0].size();
  assert(n <= m);
  vector<int> u(n + 1, 0), v(m + 1, 0), p(m + 1, 0), way, minv;
  for (int i = 1; i <= n; i++) {
    vector<int> minv(m + 1, INF);
    vector<int> way(m + 1, 0);
    vector<bool> used(m + 1, 0);
    p[0] = i;
    int k0 = 0;
    do {
      used[k0] = 1;
      int i0 = p[k0], delta = INF, k1;
      for (int j = 1; j <= m; j++) {
        if (!used[j]) {
          const int cur = matrix[i0 - 1][j - 1] - u[i0] - v[j];
          if (cur < minv[j]) {
            minv[j] = cur;
            way[j] = k0;
          }
          if (minv[j] < delta) {
            delta = minv[j];
            k1 = j;
          }
        }
      }
      for (int j = 0; j <= m; j++) {
        if (used[j]) {
          u[p[j]] += delta;
          v[j] -= delta;
        } else {
          minv[j] -= delta;
        }
      }
      k0 = k1;
    } while (p[k0]);
    do {
      const int k1 = way[k0];
      p[k0] = p[k1];
      k0 = k1;
    } while (k0);
  }
  vector<int> ans(n, -1);
  for (int j = 1; j <= m; j++) {
    if (!p[j])
      continue;
    ans[p[j] - 1] = j - 1;
  }
  return {ans, -v[0]};
}
```

5.17. Kruskal

```cpp
/// Requires DSU.cpp
struct edge {
  int u, v, w;
  edge() {}
  edge(int u, int v, int w) : u(u), v(v), w(w) {}
```

```
  bool operator<(const edge &a) const { return w < a.w; }
};

/// Returns weight of the minimum spanning tree of the graph.
///
/// Time Complexity: O(V log V)
int kruskal(int n, vector<edge> &edges) {
  DSU dsu(n);
  sort(edges.begin(), edges.end());

  int weight = 0;
  for (int i = 0; i < edges.size(); i++) {
    if (dsu.Union(edges[i].u, edges[i].v)) {
      weight += edges[i].w;
    }
  }

  return weight;
}
```

5.18. Lca

```
// #define DIST
// #define COST
/// UNCOMMENT ALSO THE LINE BELOW FOR COST!

class LCA {
 private:
  int n;
  // INDEXED from 0 or 1??
  int indexed_from;
  /// Store all log2 from 1 to n
  vector<int> lg;
  // level of the i-th node (height)
  vector<int> level;
  // matrix to store the ancestors of each node in power of 2 levels
  vector<vector<int>> anc;

  #ifdef DIST
    vector<int> dist;
  #endif
  #ifdef COST
    // int NEUTRAL_VALUE = -INF; // MAX COST
    // int combine(const int a, const int b) {return max(a, b);}
    // int NEUTRAL_VALUE = INF; // MIN COST
    // int combine(const int a, const int b) {return min(a, b);}
    vector<vector<int>> cost;
  #endif

 private:
  void allocate() {
    // initializes a matrix [n][lg n] with -1
    this->build_log_array();
    this->anc.resize(n + 1, vector<int>(lg[n] + 1, -1));
    this->level.resize(n + 1, -1);

    #ifdef DIST
      this->dist.resize(n + 1, 0);
    #endif
    #ifdef COST
      this->cost.resize(n + 1, vector<int>(lg[n] + 1, NEUTRAL_VALUE));
    #endif
  }

  void build_log_array() {
    this->lg.resize(this->n + 1);

    for(int i = 2; i <= this->n; i++)
      this->lg[i] = this->lg[i/2] + 1;
  }

  void build_anc() {
    for(int j = 1; j < anc.front().size(); j++)
      for(int i = 0; i < anc.size(); i++)
        if(this->anc[i][j - 1] != -1) {
          this->anc[i][j] = this->anc[this->anc[i][j - 1]][j - 1];
          #ifdef COST
            this->cost[i][j] = combine(this->cost[i][j - 1], this->cost[anc[i][j - 1]][j - 1]);
          #endif
        }
  }

  void build_weighted(const vector<vector<pair<int, int>>> &adj) {
    this->dfs_LCA_weighted(this->indexed_from, -1, 1, 0, adj);

    this->build_anc();
  }

  void dfs_LCA_weighted(const int u, const int p, const int l, const int d, const vector<vector<pair<int, int>>> &adj) {
    this->level[u] = l;
    this->anc[u][0] = p;
    #ifdef DIST
      this->dist[u] = d;
    #endif

    for(const pair<int, int> &x: adj[u]) {
      int v = x.first, w = x.second;
      if(v == p)
        continue;
      #ifdef COST
        this->cost[v][0] = w;
      #endif
      this->dfs_LCA_weighted(v, u, l + 1, d + w, adj);
    }
  }

  void build_unweighted(const vector<vector<int>> &adj) {
    this->dfs_LCA_unweighted(this->indexed_from, -1, 1, 0, adj);

    this->build_anc();
  }

  void dfs_LCA_unweighted(const int u, const int p, const int l, const int d, const vector<vector<int>> &adj) {
    this->level[u] = l;
    this->anc[u][0] = p;
    #ifdef DIST
      this->dist[u] = d;
    #endif

    for(const int v: adj[u]) {
      if(v == p)
        continue;
      this->dfs_LCA_unweighted(v, u, l + 1, d + 1, adj);
    }
  }

```

```
  // go up k levels from x
  int lca_go_up(int x, int k) {
    for(int i = 0; k > 0; i++, k >>= 1)
      if(k & 1) {
        x = this->anc[x][i];
        if(x == -1)
          return -1;
      }

    return x;
  }

  #ifdef COST
  /// Query between the an ancestor of v (p) and v. It returns the
  /// max/min edge between them.
  int lca_query_cost_in_line(int v, int p) {
    assert(this->level[v] >= this->level[p]);

    int k = this->level[v] - this->level[p];
    int ans = NEUTRAL_VALUE;

    for(int i = 0; k > 0; i++, k >>= 1)
      if(k & 1) {
        ans = combine(ans, this->cost[v][i]);
        v = this->anc[v][i];
      }

    return ans;
  }
  #endif

  int get_lca(int a, int b) {
    // a is below b
    if(this->level[b] > this->level[a])
      swap(a,b);

    const int logg = lg[this->level[a]];

    // putting a and b in the same level
    for(int i = logg; i >= 0; i--)
      if(this->level[a] - (1 << i) >= this->level[b])
        a = this->anc[a][i];

    if(a == b)
      return a;

    for(int i = logg; i >= 0; i--)
      if(this->anc[a][i] != -1 && this->anc[a][i] != this->anc[b][i]) {
        a = this->anc[a][i];
        b = this->anc[b][i];
      }

    return anc[a][0];
  }

 public:
  /// Builds an weighted graph.
  ///
  /// Time Complexity: O(n*log(n))
  explicit LCA(const vector<vector<pair<int, int>>> &adj, const int indexed_from) {
    this->n = adj.size();
    this->indexed_from = indexed_from;
    this->allocate();

    this->build_weighted(adj);
  }

  /// Builds an unweighted graph.
  ///
  /// Time Complexity: O(n*log(n))
  explicit LCA(const vector<vector<int>> &adj, const int indexed_from) {
    this->n = adj.size();
    this->indexed_from = indexed_from;
    this->allocate();

    this->build_unweighted(adj);
  }

  /// Goes up k levels from v. If it passes the root, returns -1.
  ///
  /// Time Complexity: O(log(k))
  int go_up(const int v, const int k) {
    assert(indexed_from <= v); assert(v < this->n + indexed_from);

    return this->lca_go_up(v, k);
  }

  /// Returns the parent of v in the LCA dfs from 1.
  ///
  /// Time Complexity: O(1)
  int parent(int v) {
    assert(indexed_from <= v); assert(v < this->n + indexed_from);

    return this->anc[v][0];
  }

  /// Returns the LCA of a and b.
  ///
  /// Time Complexity: O(log(n))
  int query_lca(const int a, const int b) {
    assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n + indexed_from);

    return this->get_lca(a, b);
  }

  #ifdef DIST
  /// Returns the distance from a to b. When the graph is unweighted, it is considered
  /// 1 as the weight of the edges.
  ///
  /// Time Complexity: O(log(n))
  int query_dist(const int a, const int b) {
    assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n + indexed_from);

    return this->dist[a] + this->dist[b] - 2*this->dist[this->get_lca(a, b)];
  }
  #endif

  #ifdef COST
  /// Returns the max/min weight edge from a to b.
  ///
  /// Time Complexity: O(log(n))
  int query_cost(const int a, const int b) {
    assert(indexed_from <= min(a, b)); assert(max(a, b) < this->n + indexed_from);

    const int l = this->query_lca(a, b);
```

```
    return combine(this->lca_query_cost_in_line(a, l),
    this->lca_query_cost_in_line(b, l));
  }
  #endif
};
```

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

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

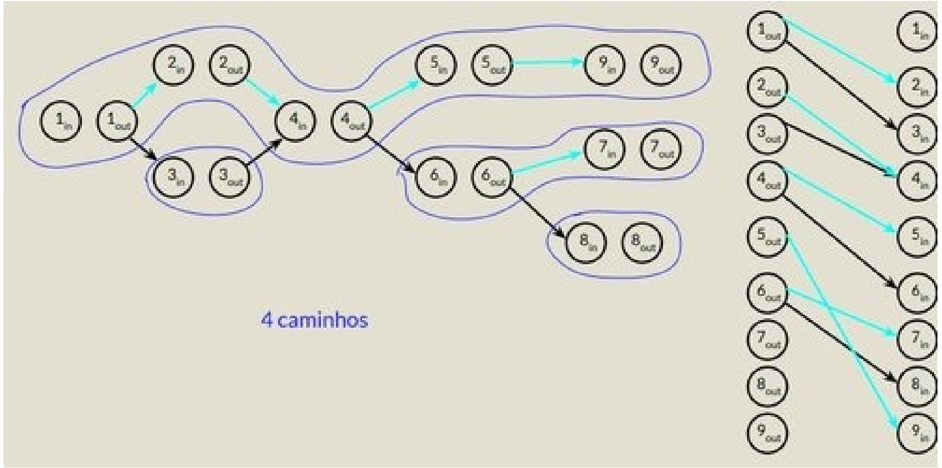
5.20. Maximum Path Unweighted Graph

```
/// Returns the maximum path between the vertices 0 and n - 1 in a
    unweighted graph.
///
/// Time Complexity: O(V + E)
int maximum_path(int n) {
  vector<int> top_order = topological_sort(n);
  vector<int> pai(n, -1);
  if(top_order.empty())
    return -1;

  vector<int> dp(n);
  dp[0] = 1;
  for(int u: top_order)
    for(int v: adj[u])
      if(dp[u] && dp[u] + 1 > dp[v]) {
        dp[v] = dp[u] + 1;
        pai[v] = u;
      }

  if(dp[n - 1] == 0)
    return -1;

  vector<int> path;
  int cur = n - 1;
  while(cur != -1) {
    path.pb(cur);
    cur = pai[cur];
  }
  reverse(path.begin(), path.end());

  // cout << path.size() << endl;
  // for(int x: path) {
  //   cout << x + 1 << ' ';
  // }
  // cout << endl;

  return dp[n - 1];
}
```

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

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

5.22. Minimum Path Cover In Dag



5.23. Minimum Path Cover In Dag

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

5.24. Number Of Different Spanning Trees In A Complete Graph

```
Cayley's formula

n ^ (n - 2)
```

5.25. Number Of Ways To Make A Graph Connected

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

5.26. Pruffer Decode

```
// IT MUST BE INDEXED BY 0.
/// Returns the adjacency matrix of the decoded tree.
///
/// Time Complexity: O(V)
vector<vector<int>> pruefer_decode(const vector<int> &code) {

  int n = code.size() + 2;
  vector<vector<int>> adj = vector<vector<int>>(n, vector<int>());
```

```
  vector<int> degree(n, 1);
  for (int x : code)
    degree[x]++;

  int ptr = 0;
  while (degree[ptr] > 1)
    ++ptr;

  int nxt = ptr;
  for (int u : code) {
    adj[u].push_back(nxt);
    adj[nxt].push_back(u);

    if (--degree[u] == 1 && u < ptr)
      nxt = u;
    else {
      while (degree[++ptr] > 1)
        ;
      nxt = ptr;
    }
  }
  adj[n - 1].push_back(nxt);
  adj[nxt].push_back(n - 1);

  return adj;
}
```

5.27. Pruffer Encode

```
void dfs(int v, const vector<vector<int>> &adj, vector<int> &parent) {
  for (int u : adj[v]) {
    if (u != parent[v]) {
      parent[u] = v;
      dfs(u, adj, parent);
    }
  }
}

// IT MUST BE INDEXED BY 0.
/// Returns prueffer code of the tree.
///
/// Time Complexity: O(V)
vector<int> pruefer_code(const vector<vector<int>> &adj) {
  int n = adj.size();
  vector<int> parent(n);
  parent[n - 1] = -1;
  dfs(n - 1, adj, parent);

  int ptr = -1;
  vector<int> degree(n);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1 && ptr == -1)
      ptr = i;
  }

  vector<int> code(n - 2);
  int leaf = ptr;
  for (int i = 0; i < n - 2; i++) {
    int next = parent[leaf];
    code[i] = next;
    if (--degree[next] == 1 && next < ptr)
      leaf = next;
    else {
      ptr++;
      while (degree[ptr] != 1)
        ptr++;
      leaf = ptr;
    }
  }

  return code;
}
```

5.28. Pruffer Properties

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

5.29. Remove All Bridges From Graph

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

5.30. Scc (Kosaraju)

```
class SCC {
 private:
  // number of vertices
  int n;
  // indicates whether it is indexed from 0 or 1
  int indexed_from;
  // reversed graph
  vector<vector<int>> trans;

 private:
  void dfs_trans(int u, int id) {
    comp[u] = id;
    scc[id].push_back(u);

    for (int v: trans[u])
      if (comp[v] == -1)
        dfs_trans(v, id);
  }

  void get_transpose(vector<vector<int>>& adj) {
    for (int u = indexed_from; u < this->n + indexed_from; u++)
      for(int v: adj[u])
        trans[v].push_back(u);
  }

  void dfs_fill_order(int u, stack<int> &s, vector<vector<int>>& adj) {
    comp[u] = true;

    for(int v: adj[u])
      if(!comp[v])
        dfs_fill_order(v, s, adj);

    s.push(u);
```

```cpp
  }

  // The main function that finds all SCCs
  void compute_SCC(vector<vector<int>>& adj) {

    stack<int> s;
    // Fill vertices in stack according to their finishing times
    for(int i = indexed_from; i < this->n + indexed_from; i++)
      if(!comp[i])
        dfs_fill_order(i, s, adj);

    // Create a reversed graph
    get_transpose(adj);

    fill(comp.begin(), comp.end(), -1);

    // Now process all vertices in order defined by stack
    while(s.empty() == false) {
      int v = s.top();
      s.pop();

      if(comp[v] == -1)
        dfs_trans(v, this->number_of_comp++);
    }
  }

 public:
  // number of the component of the i-th vertex
  // it's always indexed from 0
  vector<int> comp;
  // the i-th vector contains the vertices that belong to the i-th scc
  // it's always indexed from 0
  vector<vector<int>> scc;
  int number_of_comp = 0;

  SCC(int n, int indexed_from, vector<vector<int>>& adj) {
    this->n = n;
    this->indexed_from = indexed_from;
    comp.resize(n + 1);
    trans.resize(n + 1);
    scc.resize(n + 1);

    this->compute_SCC(adj);
  }
};
```

5.31. Topological Sort

```cpp
/// Time Complexity: O(V + E)
vector<int> topological_sort(const int indexed_from,
                             const vector<vector<int>> &adj) {
  const int n = adj.size();
  vector<int> in_degree(n, 0);

  for (int u = indexed_from; u < n; ++u)
    for (const int v : adj[u])
      in_degree[v]++;

  queue<int> q;
  for (int i = indexed_from; i < n; ++i)
    if (in_degree[i] == 0)
      q.emplace(i);

  int cnt = 0;
  vector<int> top_order;
  while (!q.empty()) {
    const int u = q.front();
    q.pop();

    top_order.emplace_back(u);
    ++cnt;

    for (const int v : adj[u])
      if (--in_degree[v] == 0)
        q.emplace(v);
  }

  if (cnt != n) {
    // There exists a cycle in the graph
    return vector<int>();
  }

  return top_order;
}
```

5.32. Tree Diameter

```cpp
namespace tree {
/// Returns a pair which contains the most distant vertex from src and the
/// distance.
pair<int, int> bfs(const int src, const vector<vector<int>> &adj) {
  queue<tuple<int, int, int>> q;
  q.emplace(0, src, -1);
  int furthest = src, dist = 0;
  while (!q.empty()) {
    int d, u, p;
    tie(d, u, p) = q.front();
    q.pop();
    if (d > dist) {
      furthest = u;
      dist = d;
    }
    for (int v : adj[u]) {
      if (v == p)
        continue;
      q.emplace(d + 1, v, u);
    }
  }
  return make_pair(furthest, dist);
}

/// Returns the length of the diameter and two vertices that belong to it.
///
/// Time Complexity: O(n)
tuple<int, int, int> diameter(const int root_idx,
                              const vector<vector<int>> &adj) {
  int ini = bfs(root_idx, adj).first, end, dist;
  tie(end, dist) = bfs(ini, adj);
  return {dist, ini, end};
}
}; // namespace tree
```

5.33. Tree Distance

```cpp
vector<pair<int, int>> sub(MAXN, pair<int, int>(0, 0));

void subu(int u, int p) {
```

```
  for (const pair<int, int> x : adj[u]) {
    int v = x.first, w = x.second;
    if (v == p)
      continue;
    subu(v, u);
    if (sub[v].first + w > sub[u].first) {
      swap(sub[u].first, sub[u].second);
      sub[u].first = sub[v].first + w;
    } else if (sub[v].first + w > sub[u].second) {
      sub[u].second = sub[v].first + w;
    }
  }
}

/// Contains the maximum distance to the node i
vector<int> ans(MAXN);

void dfs(int u, int d, int p) {
  ans[u] = max(d, sub[u].first);
  for (const pair<int, int> x : adj[u]) {
    int v = x.first, w = x.second;
    if (v == p)
      continue;
    if (sub[v].first + w == ans[u]) {
      dfs(v, max(d, sub[u].second) + w, u);
    } else {
      dfs(v, ans[u] + w, u);
    }
  }
}

// Returns the maximum tree distance
int solve() {
  subu(0, -1);
  dfs(0, 0, -1);
  return *max_element(ans.begin(), ans.end());
}
```

6. Language Stuff

6.1. Climits

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

6.2. Checagem E Tranformacao De Caractere

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

6.3. Conta Digitos 1 Ate N

```
int solve(int n) {

  int maxx = 9, minn = 1, dig = 1, ret = 0;

  for(int i = 1; i <= 17; i++) {
    int q = min(maxx, n);
    ret += max(0ll, (q - minn + 1) * dig);
    maxx = (maxx * 10 + 9), minn *= 10, dig++;
  }

  return ret;
}
```

6.4. Escrita Em Arquivo

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

6.5. Gcd

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

6.6. Hipotenusa

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

6.7. Int To Binary String

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

6.8. Int To String

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

6.9. Leitura De Arquivo

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

6.10. Max E Min Element Num Vetor

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

6.11. Permutacao

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

6.12. Remove Repeticoes Continuas Num Vetor

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

6.13. Rotate (Left)

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

6.14. Rotate (Right)

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

6.15. Scanf De Uma String

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

6.16. Split Function

```
// SEPARA STRING POR UM DELIMITADOR
// EX: str=A-B-C split -> x = {A,B,C}
vector<string> split(const string &s, char delim) {
    stringstream ss(s);
    string item;
    vector<string> tokens;
    while (getline(ss, item, delim)) {
        tokens.push_back(item);
    }
    return tokens;
}
int main () {
  vector<string> x = split("cap-one-best-opinion-language", '-');
// x = {cap,one,best,opinion,language};
}
```

6.17. String To Long Long

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

6.18. Substring

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

6.19. Width

```
cout << width(13);
cout << 100 << endl; // "     100      "
cout.fill('x');
cout.width(13);
cout << 100 << endl; // "xxxxx100xxxxx"
cout << right << 100 << endl; "xxxxxxx100"
```

6.20. Binary String To Int

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

6.21. Check

```
#!/bin/bash
g++ -std=c++17 gen.cpp -o gen
g++ -std=c++17 a.cpp -o a
g++ -std=c++17 brute.cpp -o brute

for((i=1;;i++)); do
  echo $i
  ./gen $i > in
  diff <(./a < in) <(./brute < in) || break
done

cat in
#sed -i 's/\r$//' filename  ----- remover \r do txt
```

6.22. Check Overflow

```
bool __builtin_add_overflow (type1 a, type2 b, type3 *res)
bool __builtin_sadd_overflow (int a, int b, int *res)
bool __builtin_saddl_overflow (long int a, long int b, long int *res)
bool __builtin_saddll_overflow (long long int a, long long int b, long long
    int *res)
bool __builtin_uadd_overflow (unsigned int a, unsigned int b, unsigned int
    *res)
bool __builtin_uaddl_overflow (unsigned long int a, unsigned long int b,
    unsigned long int *res)
bool __builtin_uaddll_overflow (unsigned long long int a, unsigned long long
    int b, unsigned long long int *res)

bool __builtin_sub_overflow (type1 a, type2 b, type3 *res)
bool __builtin_ssub_overflow (int a, int b, int *res)
bool __builtin_ssubl_overflow (long int a, long int b, long int *res)
bool __builtin_ssubll_overflow (long long int a, long long int b, long long
    int *res)
bool __builtin_usub_overflow (unsigned int a, unsigned int b, unsigned int
    *res)
bool __builtin_usubl_overflow (unsigned long int a, unsigned long int b,
    unsigned long int *res)
bool __builtin_usubll_overflow (unsigned long long int a, unsigned long long
    int b, unsigned long long int *res)

bool __builtin_mul_overflow (type1 a, type2 b, type3 *res)
bool __builtin_smul_overflow (int a, int b, int *res)
bool __builtin_smull_overflow (long int a, long int b, long int *res)
bool __builtin_smulll_overflow (long long int a, long long int b, long long
    int *res)
```

```
bool __builtin_umul_overflow (unsigned int a, unsigned int b, unsigned int
    *res)
bool __builtin_umull_overflow (unsigned long int a, unsigned long int b,
    unsigned long int *res)
bool __builtin_umulll_overflow (unsigned long long int a, unsigned long long
    int b, unsigned long long int *res)
```

6.23. Counting Bits

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

6.24. Random Numbers

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

6.25. Readint

```
int readInt() {
  int a = 0;
  char c;
  while (!(c >= '0' && c <= '9'))
    c = getchar();
  while (c >= '0' && c <= '9')
    a = 10 * a + (c - '0'), c = getchar();
  return a;
}
```

6.26. Time Measure

```
clock_t start = clock();

/* Execute the program */

clock_t end = clock();

double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
```

7. Math

7.1. Bell Numbers

```
/// Number of ways to partition a set.
/// For example, the set {a, b, c}.
/// It can be partitioned in five ways: {(a) (b) (c)},{(a, b), (c)},
/// {(a, c)(b)}, {(b, c), a}, {(a, b, c)}.
///
/// Time Complexity: O(n * n)
int bellNumber(int n) {
  int bell[n + 1][n + 1];
  bell[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    bell[i][0] = bell[i - 1][i - 1];

    for (int j = 1; j <= i; j++)
      bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
  }
  return bell[n][0];
}
```

7.2. Binary Exponentiation

```
int bin_pow(const int n, int p) {
  assert(p >= 0);
  int ans = 1;
  int cur_pow = n;

  while (p) {
    if (p & 1)
      ans = (ans * cur_pow) % MOD;

    cur_pow = (cur_pow * cur_pow) % MOD;
    p >>= 1;
  }

  return ans;
}
```

7.3. Chinese Remainder Theorem

```
int inv(int a, int m) {
  int m0 = m, t, q;
  int x0 = 0, x1 = 1;

  if (m == 1)
    return 0;

  // Apply extended Euclid Algorithm
  while (a > 1) {
    // q is quotient
    if (m == 0)
      return INF;
    q = a / m;
    t = m;
    // m is remainder now, process same as euclid's algo
    m = a % m, a = t;
    t = x0;
    x0 = x1 - q * x0;
    x1 = t;
  }

  // Make x1 positive
  if (x1 < 0)
    x1 += m0;

  return x1;
}
// k is size of num[] and rem[].  Returns the smallest
// number x such that:
//  x % num[0] = rem[0],
//  x % num[1] = rem[1],
//  ..................
//  x % num[k-2] = rem[k-1]
// Assumption: Numbers in num[] are pairwise coprimes
// (gcd for every pair is 1)
int findMinX(const vector<int> &num, const vector<int> &rem, const int k) {
  // Compute product of all numbers
  int prod = 1;
```

```
  for (int i = 0; i < k; i++)
    prod *= num[i];

  int result = 0;

  // Apply above formula
  for (int i = 0; i < k; i++) {
    int pp = prod / num[i];
    int iv = inv(pp, num[i]);
    if (iv == INF)
      return INF;
    result += rem[i] * inv(pp, num[i]) * pp;
  }

  // IF IS NOT VALID RETURN INF
  return (result % prod == 0 ? INF : result % prod);
}
```

7.4. Combinatorics

```
class Combinatorics {
private:
  static constexpr int MOD = 1e9 + 7;
  const int max_val;
  vector<int> _inv, _fat;

private:
  int mod(int x) {
    x %= MOD;
    if (x < 0)
      x += MOD;
    return x;
  }

  static int bin_pow(const int n, int p) {
    assert(p >= 0);
    int ans = 1;
    int cur_pow = n;

    while (p) {
      if (p & 1ll)
        ans = (ans * cur_pow) % MOD;

      cur_pow = (cur_pow * cur_pow) % MOD;
      p >>= 1ll;
    }

    return ans;
  }

  vector<int> build_inverse(const int max_val) {
    vector<int> inv(max_val + 1);
    inv[1] = 1;
    for (int i = 2; i <= max_val; ++i)
      inv[i] = mod(-MOD / i * inv[MOD % i]);
    return inv;
  }

  vector<int> build_fat(const int max_val) {
    vector<int> fat(max_val + 1);
    fat[0] = 1;
    for (int i = 1; i <= max_val; ++i)
      fat[i] = mod(i * fat[i - 1]);
    return fat;
  }

public:
  /// Builds both factorial and modular inverse array.
  ///
  /// Time Complexity: O(max_val)
  Combinatorics(const int max_val) : max_val(max_val) {
    assert(0 <= max_val), assert(max_val <= MOD);
    this->_inv = this->build_inverse(max_val);
    this->_fat = this->build_fat(max_val);
  }

  /// Returns the modular inverse of n % MOD.
  ///
  /// Time Complexity: O(log(MOD))
  static int inv_log(const int n) { return bin_pow(n, MOD - 2); }

  /// Returns the modular inverse of n % MOD.
  ///
  /// Time Complexity: O((n <= max_val ? 1 : log(MOD))
  int inv(const int n) {
    assert(0 <= n);
    if (n <= max_val)
      return this->_inv[n];
    else
      return inv_log(n);
  }

  /// Returns the factorial of n % MOD.
  int fat(const int n) {
    assert(0 <= n), assert(n <= max_val);
    return this->_fat[n];
  }

  /// Returns C(n, k) % MOD.
  ///
  /// Time Complexity: O(1)
  int choose(const int n, const int k) {
    assert(0 <= k), assert(k <= n), assert(n <= this->max_val);
    return mod(fat(n) * mod(inv(fat(k)) * inv(fat(n - k))));
  }
};
```

7.5. Diophantine Equation

```
int gcd(int a, int b, int &x, int &y) {
  if (a == 0) {
    x = 0;
    y = 1;
    return b;
  }
  int x1, y1;
  int d = gcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return d;
}

bool diophantine(int a, int b, int c, int &x0, int &y0, int &g) {
  g = gcd(abs(a), abs(b), x0, y0);
  if (c % g)
    return false;

  x0 *= c / g;
```

```
  y0 *= c / g;
  if (a < 0)
    x0 = -x0;
  if (b < 0)
    y0 = -y0;
  return true;
}
```

7.6. Divisors

```
/// OBS: Each number has at most ∛N divisors
/// THE NUMBERS ARE NOT SORTED!!!
///
/// Time Complexity: O(sqrt(n))
vector<int> divisors(int n) {
  vector<int> ans;
  for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
      if (n / i == i)
        ans.emplace_back(i);
      else
        ans.emplace_back(i), ans.emplace_back(n / i);
    }
  }
  // sort(ans.begin(), ans.end());
  return ans;
}
```

7.7. Euler Totient

```
/// Returns the amount of numbers less than or equal to n which are co-primes
/// to it.
int phi(int n) {
  int result = n;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      while (n % i == 0)
        n /= i;
      result -= result / i;
    }
  }

  if (n > 1)
    result -= result / n;
  return result;
}
```

7.8. Extended Euclidean

```
int gcd, x, y;

// Ax + By = gcd(A,B)

void extended_euclidian(const int a, const int b) {
  if (b == 0) {
    gcd = a;
    x = 1;
    y = 0;
  } else {
    extended_euclidian(b, a % b);
    const int temp = x;
    x = y;
    y = temp - (a / b) * y;
  }
}
```

7.9. Factorization

```
/// Factorizes a number.
///
/// Time Complexity: O(sqrt(n))
map<int, int> factorize(int n) {
  map<int, int> fat;
  while (n % 2 == 0) {
    ++fat[2];
    n /= 2;
  }

  for (int i = 3; i * i <= n; i += 2) {
    while (n % i == 0) {
      ++fat[i];
      n /= i;
    }
    /* OBS1
        IF(N < 1E7)
          you can optimize by factoring with SPF
    */
  }
  if (n > 2)
    ++fat[n];
  return fat;
}
```

7.10. Inclusion Exclusion

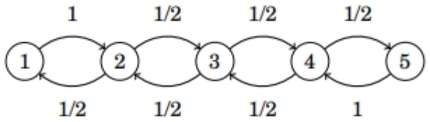
$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n}(-1)^{k+1}\left(\sum_{1\le i_1<\cdots<i_k\le n} |A_{i_1}\cap\cdots\cap A_{i_k}|\right)$$

7.11. Inclusion Exclusion

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

  for (int mask = 1; mask < (1ll << n); mask++) {
    int lcm = 1;
    for (int i = 0; i < n; i++)
      if (mask & (1ll << i))
        lcm = lcm * arr[i] / __gcd(lcm, arr[i]);
    // if the number of element is odd, then add
    if (__builtin_popcount_ll(mask) % 2 == 1)
      c += LIMIT / lcm;
    else // otherwise subtract
      c -= LIMIT / lcm;
```

```
  }

  return LIMIT - c;
}
```

7.12. Markov Chains



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

Probabily after moving 1 step from 1

7.13. Matrix Exponentiation

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

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

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

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

Fibonacci

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

7.14. Matrix Exponentiation

```
struct Matrix {
  static constexpr int MOD = 1e9 + 7;

  // static matrix, if it's created multiple times, it's recommended
  // to avoid TLE.
  static constexpr int MAXN = 4, MAXM = 4;
  array<array<int, MAXM>, MAXN> mat = {};
  int n, m;
  Matrix(const int n, const int m) : n(n), m(m) {}

  static int mod(int n) {
    n %= MOD;
    if (n < 0)
      n += MOD;
    return n;
  }

  /// Creates a n x n identity matrix.
  ///
  /// Time Complexity: O(n*n)
  Matrix identity() {
    assert(n == m);
    Matrix mat_identity(n, m);
    for (int i = 0; i < n; ++i)
      mat_identity.mat[i][i] = 1;
    return mat_identity;
  }

  /// Multiplies matrices mat and other.
  ///
  /// Time Complexity: O(mat.size() ^ 3)
  Matrix operator*(const Matrix &other) const {
    assert(m == other.n);
    Matrix ans(n, other.m);
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        for (int k = 0; k < m; ++k)
          ans.mat[i][j] = mod(ans.mat[i][j] + mat[i][k] * other.mat[k][j]);
    return ans;
  }

  /// Exponentiates the matrix mat to the power of p.
  ///
  /// Time Complexity: O((mat.size() ^ 3) * log2(p))
  Matrix expo(int p) {
    assert(p >= 0);
    Matrix ans = identity(), cur_power(n, m);
    cur_power.mat = mat;
    while (p) {
      if (p & 1)
        ans = ans * cur_power;

      cur_power = cur_power * cur_power;
      p >>= 1;
    }
    return ans;
  }
};
```

7.15. Pollard Rho (Find A Divisor)

```
// Requires binary_exponentiation.cpp

```

```
/// Returns a prime divisor for n.
///
/// Expected Time Complexity: O(n1/4)
int pollard_rho(const int n) {
  srand(time(NULL));

  /* no prime divisor for 1 */
  if (n == 1)
    return n;

  if (n % 2 == 0)
    return 2;

  /* we will pick from the range [2, N) */
  int x = (rand() % (n - 2)) + 2;
  int y = x;

  /* the constant in f(x).
   * Algorithm can be re-run with a different c
   * if it throws failure for a composite. */
  int c = (rand() % (n - 1)) + 1;

  /* Initialize candidate divisor (or result) */
  int d = 1;

  /* until the prime factor isn't obtained.
  If n is prime, return n */
  while (d == 1) {
    /* Tortoise Move: x(i+1) = f(x(i)) */
    x = (modular_pow(x, 2, n) + c + n) % n;

    /* Hare Move: y(i+1) = f(f(y(i))) */
    y = (modular_pow(y, 2, n) + c + n) % n;
    y = (modular_pow(y, 2, n) + c + n) % n;

    d = __gcd(abs(x - y), n);

    /* retry if the algorithm fails to find prime factor
     * with chosen x and c */
    if (d == n)
      return pollard_rho(n);
  }

  return d;
}
```

7.16. Primality Check

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

7.17. Primes

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

7.18. Sieve + Segmented Sieve

```
const int MAXN = 1e6;

/// Contains all the primes in the segments
vector<int> segPrimes;
bitset<MAXN + 5> primesInSeg;

/// smallest prime factor
int spf[MAXN + 5];

vector<int> primes;
bitset<MAXN + 5> isPrime;

void sieve(int n = MAXN + 2) {

  for (int i = 0; i <= n; i++)
    spf[i] = i;

  isPrime.set();
  for (int i = 2; i <= n; i++) {
    if (!isPrime[i])
      continue;

    for (int j = i * i; j <= n; j += i) {
      isPrime[j] = false;
      spf[j] = min(i, spf[j]);
    }
    primes.emplace_back(i);
  }
}

vector<int> getFactorization(int x) {
  vector<int> ret;
  while (x != 1) {
    ret.emplace_back(spf[x]);
    x = x / spf[x];
  }
  return ret;
}

/// Gets all primes from l to r
void segSieve(int l, int r) {
```

```
  // primes from l to r
  // transferred to 0..(l-r)
  segPrimes.clear();
  primesInSeg.set();
  int sq = sqrt(r) + 5;

  for (int p : primes) {
    if (p > sq)
      break;

    for (int i = l - l % p; i <= r; i += p) {
      if (i - l < 0)
        continue;

      // if i is less than 1e6, it could be checked in the
      // array of the sieve
      if (i >= (int)1e6 || !isPrime[i])
        primesInSeg[i - l] = false;
    }
  }

  for (int i = 0; i < r - l + 1; i++) {
    if (primesInSeg[i])
      segPrimes.emplace_back(i + l);
  }
}
```

7.19. Stars And Bars

I. positive integers x_i

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

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

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

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

II. non-negative integers x_i

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

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

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

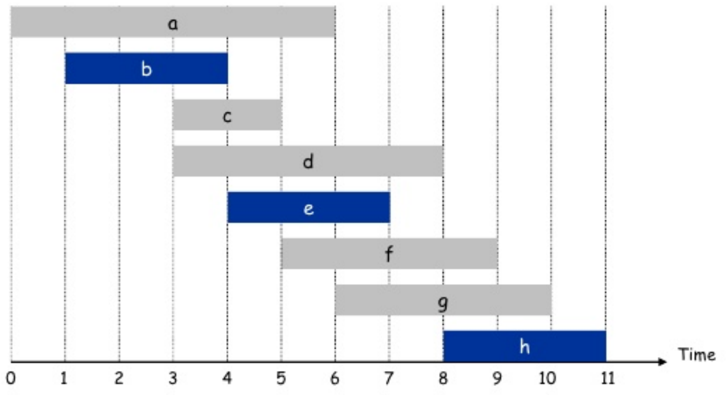
$$\binom{22+4-1}{22} = \binom{25}{22} = \frac{25!}{22!3!} = 2300$$

8. Miscellaneous

8.1. 2-Sat

```
// REQUIRES SCC code

// OBS: INDEXED FROM 0
class SAT {

private:
  vector<vector<int>> adj;
  int n;

public:
  vector<bool> ans;

  SAT(int n) {
    this->n = n;
    adj.resize(2 * n);
    ans.resize(n);
  }

  //  (X v Y)  = (X -> ~Y) & (~X -> Y)
  void add_or(int x, bool pos_x, int y, bool pos_y) {
    assert(0 <= x), assert(x < n);
    assert(0 <= y), assert(y < n);
    adj[(x << 1) ^ pos_x].pb((y << 1) ^ (pos_y ^ 1));
    adj[(y << 1) ^ pos_y].pb((x << 1) ^ (pos_x ^ 1));
  }

  // (X xor Y) = (X v Y) & (~X v ~Y)
  // for this function the result is always 0 1 or 1 0
  void add_xor(int x, bool pos_x, int y, bool pos_y) {
    assert(0 <= x), assert(x < n);
    assert(0 <= y), assert(y < n);
    add_or(x, y, pos_x, pos_y);
    add_or(x, y, pos_x ^ 1, pos_y ^ 1);
  }

  bool check() {
    SCC scc(2 * n, 0, adj);

    for (int i = 0; i < n; i++) {
      if (scc.comp[(i << 1) | 1] == scc.comp[(i << 1) | 0])
        return false;
      ans[i] = (scc.comp[(i << 1) | 1] < scc.comp[(i << 1) | 0]);
    }

    return true;
  }
};
```

8.2. Interval Scheduling



8.3. Interval Scheduling

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

8.4. Oito Rainhas

```cpp
#define N 4
bool isSafe(int mat[N][N],int row,int col) {
  for(int i = row - 1; i >= 0; i--)
    if(mat[i][col])
      return false;
  for(int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--,j--)
    if(mat[i][j])
      return false;
  for(int i = row - 1, j = col + 1; i >= 0 && j < N; i--,j++)
    if(mat[i][j])
      return false;
  return true;
}
// inicialmente a matriz esta zerada
int queen(int mat[N][N], int row  = 0) {
  if(row >= N) {
    for(int i = 0; i < N; i++) {
      for(int j = 0; j < N; j++) {
        cout << mat[i][j] << ' ';
      }
      cout << endl;
    }
    cout << endl << endl;
    return false;
  }
  for(int i = 0; i < N; i++) {
    if(isSafe(mat,row,i)) {
      mat[row][i] = 1;
      if(queen(mat,row+1))
        return true;
      mat[row][i] = 0;
    }
  }
  return false;
}
```

8.5. Sliding Window Minimum

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

void swma(vector<int> arr, int k) {
  deque<ii> window;
  for(int i = 0; i < arr.size(); i++) {
    while(!window.empty() && window.back().ff > arr[i])
      window.pop_back();
    window.pb(ii(arr[i],i));
    while(window.front().ss <= i - k)
      window.pop_front();

  if(i >= k)
    cout << ' ';
  if(i - k + 1 >= 0)
    cout << window.front().ff;
  }
}
```

8.6. Torre De Hanoi

```c
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
  if (n == 1) {
    printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
    return;
  }
  towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
  printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
  towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main() {
  int n = 4; // Number of disks
  towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
  return 0;
}
```

8.7. Counting Frequency Of Digits From 1 To K

```python
def check(k):
  ans = [0] * 10
  for d in range(1, 10):
    pot = 10
    last = 1
    for i in range(20):
      v = (k // pot * last) + min(max(0, ((k % pot) - (last * d)) + 1), last)
      ans[d] += v
      pot *= 10
      last *= 10

  return ans
```

8.8. Infix To Postfix

```
/// Infix Expression | Prefix Expression | Postfix Expression
///      A + B       |     + A B         |       A B +
///    A + B * C     |    + A * B C      |     A B C * +
/// Time Complexity: O(n)
int infix_to_postfix(const string &infix) {
  map<char, int> prec;
  stack<char> op;
  string postfix;

  prec['+'] = prec['-'] = 1;
  prec['*'] = prec['/'] = 2;
  prec['^'] = 3;
  for (int i = 0; i < infix.size(); ++i) {
    char c = infix[i];
    if (is_digit(c)) {
      while (i < infix.size() && isdigit(infix[i])) {
        postfix += infix[i];
        ++i;
      }
      --i;
    } else if (isalpha(c))
      postfix += c;
    else if (c == '(')
      op.push('(');
    else if (c == ')') {
      while (!op.empty() && op.top() != '(') {
        postfix += op.top();
        op.pop();
      }
      op.pop();
    } else {
      while (!op.empty() && prec[op.top()] >= prec[c]) {
        postfix += op.top();
        op.pop();
      }
      op.push(c);
    }
  }
  while (!op.empty()) {
    postfix += op.top();
    op.pop();
  }
  return postfix;
}
```

8.9. Kadane

```
/// Returns the maximum contiguous sum in the array.
///
/// Time Complexity: O(n)
int kadane(vector<int> &arr) {
  if (arr.empty())
    return 0;
  int sum, tot;
  sum = tot = arr[0];

  for (int i = 1; i < arr.size(); i++) {
    sum = max(arr[i], arr[i] + sum);
    if (sum > tot)
      tot = sum;
  }
  return tot;
}
```

8.10. Kadane (Segment Tree)

```
struct Node {
  int pref, suf, tot, best;
  Node () {}
  Node(int pref, int suf, int tot, int best) : pref(pref), suf(suf),
    tot(tot), best(best) {}
};

const int MAXN = 2E5 + 10;
Node tree[5*MAXN];
int arr[MAXN];

Node query(const int l, const int r, const int i, const int j, const int
    pos) {

  if(l > r || l > j || r < i)
    return Node(-INF, -INF, -INF, -INF);

  if(i <= l && r <= j)
    return Node(tree[pos].pref, tree[pos].suf, tree[pos].tot,
    tree[pos].best);

  int mid = (l + r) / 2;
  Node left = query(l,mid,i,j,2*pos+1), right = query(mid+1,r,i,j,2*pos+2);
  Node x;
  x.pref = max({left.pref, left.tot, left.tot + right.pref});
  x.suf = max({right.suf, right.tot, right.tot + left.suf});
  x.tot = left.tot + right.tot;
  x.best = max({left.best,right.best, left.suf + right.pref});
  return x;
}

// Update arr[idx] to v
// ITS NOT DELTA!!!
void update(int l, int r, const int idx, const int v, const int pos) {
  if(l > r || l > idx || r < idx)
    return;

  if(l == idx && r == idx) {
    tree[pos] = Node(v, v, v, v);
    return;
  }

  int mid = (l + r)/2;
  update(l,mid,idx,v,2*pos+1); update(mid+1,r,idx,v,2*pos+2);
  l = 2*pos+1, r = 2*pos+2;
  tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
    tree[r].pref});
  tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
  tree[pos].tot = tree[l].tot + tree[r].tot;
  tree[pos].best = max({tree[l].best,tree[r].best, tree[l].suf +
    tree[r].pref});
}

void build(int l, int r, const int pos) {

  if(l == r) {
    tree[pos] = Node(arr[l], arr[l], arr[l], arr[l]);
    return;
  }

```

```
  int mid = (l + r)/2;
  build(l,mid,2*pos+1); build(mid+1,r,2*pos+2);
  l = 2*pos+1, r = 2*pos+2;
  tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
    tree[r].pref});
  tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
  tree[pos].tot = tree[l].tot + tree[r].tot;
  tree[pos].best = max({tree[l].best,tree[r].best, tree[l].suf +
    tree[r].pref});
}
```

8.11. Kadane 2D

```

// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
int mat[1001][1001]
int ROW = 1000, COL = 1000;

// Implementation of Kadane's algorithm for 1D array. The function
// returns the maximum sum and stores starting and ending indexes of the
// maximum sum subarray at addresses pointed by start and finish pointers
// respectively.
int kadane(int* arr, int* start, int* finish, int n) {
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i) {
        sum += arr[i];
        if (sum < 0) {
            sum = 0;
            local_start = i+1;
        }
        else if (sum > maxSum){
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

     // There is at-least one non-negative number
    if (*finish != -1)
        return maxSum;

    // Special Case: When all numbers in arr[] are negative
    maxSum = arr[0];
    *start = *finish = 0;

    // Find the maximum element in array
    for (i = 1; i < n; i++) {
        if (arr[i] > maxSum) {
            maxSum = arr[i];
            *start = *finish = i;
        }
    }
    return maxSum;
}

// The main function that finds maximum sum rectangle in mat[][]
int findMaxSum() {
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left) {
        // Initialize all elements of temp as 0
        for(int i = 0; i < ROW; i++)
            temp[i] = 0;

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right) {
           // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += mat[i][right];

            // Find the maximum sum subarray in temp[]. The kadane()
            // function also sets values of start and finish.  So 'sum' is
            // sum of rectangle between (start, left) and (finish, right)
            //  which is the maximum sum with boundary columns strictly as
            //  left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then
            // update maxSum and other output values
            if (sum > maxSum) {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

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

8.12. Largest Area In Histogram

```
/// Time Complexity: O(n)
int largest_area_in_histogram(vector<int> &arr) {
  arr.emplace_back(0);

  stack<int> s;
  int ans = 0;
  for (int i = 0; i < arr.size(); ++i) {
    while (!s.empty() && arr[s.top()] >= arr[i]) {
      int height = arr[s.top()];
      s.pop();
      int l = (s.empty() ? 0 : s.top() + 1);
      // creates a rectangle from l to i - 1
      ans = max(ans, height * (i - l));
    }
```

```
    s.emplace(i);
  }
  return ans;
}
```

8.13. Point Compression

```
// map<int, int> rev;

/// Compress points in the array arr to the range [0..n-1].
///
/// Time Complexity: O(n log n)
vector<int> compress(vector<int> &arr) {
  vector<int> aux = arr;
  sort(aux.begin(), aux.end());
  aux.erase(unique(aux.begin(), aux.end()), aux.end());

  for (size_t i = 0; i < arr.size(); i++) {
    int id = lower_bound(aux.begin(), aux.end(), arr[i]) - aux.begin();
    // rev[id] = arr[i];
    arr[i] = id;
  }
  return arr;
}
```

8.14. Ternary Search

```
/// Returns the index in the array which contains the minimum element. In case
/// of draw, it returns the first occurrence. The array should, first, decrease,
/// then increase.
///
/// Time Complexity: O(log3(n))
int ternary_search(const vector<int> &arr) {
  int l = 0, r = (int)arr.size() - 1;
  while (r - l > 2) {
    int lc = l + (r - l) / 3;
    int rc = r - (r - l) / 3;
    // the function f(x) returns the element on the position x
    if (f(lc) > f(rc))
      // the function is going down, then the middle is on the right.
      l = lc;
    else
      r = rc;
  }
  // the range [l, r] contains the minimum element.

  int minn = f(l), idx = l;
  for (int i = l + 1; i <= r; ++i)
    if (f(i) < minn) {
      idx = i;
      minn = f(i);
    }

  return idx;
}
```

9. Strings

9.1. Trie - Maximum Xor Sum

```
// XOR(L,R) = XOR(1,L-1) ^ XOR(1,R)
ans= pre = 0
Trie.insert(0)
for i=1 to N:
    pre = pre XOR a[i]
    Trie.insert(pre)
    ans=max(ans, Trie.query(pre))
print ans

// a funcao query é a mesma da maximum xor between two elements
```

9.2. Trie - Maximum Xor Two Elements

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

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

9.3. Z-Function

```
// What is Z Array?
// For a string str[0..n-1], Z array is of same length as string.
// An element Z[i] of Z array stores length of the longest substring
// starting from str[i] which is also a prefix of str[0..n-1]. The
// first entry of Z array is meaning less as complete string is always
// prefix of itself.
// Example:
// Index
// 0   1   2   3   4   5   6   7   8   9  10  11
// Text
// a   a   b   c   a   a   b   x   a   a   a   z
// Z values
// X   1   0   0   3   1   0   0   2   2   1   0
// More Examples:
// str  = "aaaaaa"
// Z[]  = {x, 5, 4, 3, 2, 1}

// str = "aabaacd"
// Z[] = {x, 1, 0, 2, 1, 0, 0}

// str = "abababab"
// Z[] = {x, 0, 6, 0, 4, 0, 2, 0}

vector<int> z_function(const string &s) {
  vector<int> z(s.size());
  int l = -1, r = -1;
  for (int i = 1; i < s.size(); ++i) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < s.size() && s[i + z[i]] == s[z[i]])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  return z;
}
```

9.4. Aho Corasick

```
/// REQUIRES trie.cpp

```

```cpp
class Aho {
private:
  // node of the output list
  struct Out_Node {
    vector<int> str_idx;
    Out_Node *next = nullptr;
  };

  vector<Trie::Node *> fail;
  Trie trie;
  // list of nodes of output
  vector<Out_Node *> out_node;
  const vector<string> arr;

  /// Time Complexity: O(number of characters in arr)
  void build_trie() {
    const int n = arr.size();
    int node_cnt = 1;

    for (int i = 0; i < n; ++i)
      node_cnt += arr[i].size();

    out_node.reserve(node_cnt);
    for (int i = 0; i < node_cnt; ++i)
      out_node.push_back(new Out_Node());

    fail.resize(node_cnt);
    for (int i = 0; i < n; ++i) {
      const int id = trie.insert(arr[i]);
      out_node[id]->str_idx.push_back(i);
    }

    this->build_failures();
  }

  /// Returns the fail node of cur.
  Trie::Node *find_fail_node(Trie::Node *cur, char c) {
    while (cur != this->trie.root() && !cur->next.count(c))
      cur = fail[cur->id];
    // if cur is pointing to the root node and c is not a child
    if (!cur->next.count(c))
      return trie.root();
    return cur->next[c];
  }

  /// Time Complexity: O(number of characters in arr)
  void build_failures() {
    queue<const Trie::Node *> q;

    fail[trie.root()->id] = trie.root();
    for (const pair<char, Trie::Node *> v : trie.root()->next) {
      q.emplace(v.second);
      fail[v.second->id] = trie.root();
      out_node[v.second->id]->next = out_node[trie.root()->id];
    }

    while (!q.empty()) {
      const Trie::Node *u = q.front();
      q.pop();

      for (const pair<char, Trie::Node *> x : u->next) {
        const char c = x.first;
        const Trie::Node *v = x.second;
        Trie::Node *fail_node = find_fail_node(fail[u->id], c);
        fail[v->id] = fail_node;

        if (!out_node[fail_node->id]->str_idx.empty())
          out_node[v->id]->next = out_node[fail_node->id];
        else
          out_node[v->id]->next = out_node[fail_node->id]->next;

        q.emplace(v);
      }
    }
  }

  vector<vector<pair<int, int>>> aho_find_occurrences(const string &text) {
    vector<vector<pair<int, int>>> ans(arr.size());
    Trie::Node *cur = trie.root();

    for (int i = 0; i < text.size(); ++i) {
      cur = find_fail_node(cur, text[i]);
      for (Out_Node *node = out_node[cur->id]; node != nullptr;
           node = node->next)
        for (const int idx : node->str_idx)
          ans[idx].emplace_back(i - (int)arr[idx].size() + 1, i);
    }
    return ans;
  }

public:
  /// Constructor that builds the trie and the failures.
  ///
  /// Time Complexity: O(number of characters in arr)
  Aho(const vector<string> &arr) : arr(arr) { this->build_trie(); }

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

9.5. Hashing

```cpp
// Global vector used in the class.
vector<int> hash_base;

class Hash {
  /// Prime numbers to be used in mod operations
  const vector<int> m = {1000000007, 1000000009};

  vector<vector<int>> hash_table;
  vector<vector<int>> pot;
  // size of the string
  const int n;

private:
  static int mod(int n, int m) {
    n %= m;
    if (n < 0)
      n += m;
    return n;
  }

  /// Time Complexity: O(1)
  pair<int, int> hash_query(const int l, const int r) {
    vector<int> ans(m.size());
```

```
    if (l == 0) {
      for (int i = 0; i < m.size(); i++)
        ans[i] = hash_table[i][r];
    } else {
      for (int i = 0; i < m.size(); i++)
        ans[i] =
            mod((hash_table[i][r] - hash_table[i][l - 1] * pot[i][r - l +
    1]),
                m[i]);
    }

    return {ans.front(), ans.back()};
  }

  /// Time Complexity: O(m.size())
  void build_base() {
    if (!hash_base.empty())
      return;
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> distribution(CHAR_MAX, INT_MAX);
    hash_base.resize(m.size());
    for (int i = 0; i < hash_base.size(); ++i)
      hash_base[i] = distribution(gen);
  }

  /// Time Complexity: O(n)
  void build_table(const string &s) {
    pot.resize(m.size(), vector<int>(this->n));
    hash_table.resize(m.size(), vector<int>(this->n));

    for (int i = 0; i < m.size(); i++) {
      pot[i][0] = 1;
      hash_table[i][0] = s[0];
      for (int j = 1; j < this->n; j++) {
        hash_table[i][j] =
            mod(s[j] + hash_table[i][j - 1] * hash_base[i], m[i]);
        pot[i][j] = mod(pot[i][j - 1] * hash_base[i], m[i]);
      }
    }
  }

public:
  /// Constructor thats builds the hash and pot tables and the hash_base
    vector.
  ///
  /// Time Complexity: O(n)
  Hash(const string &s) : n(s.size()) {
    build_base();
    build_table(s);
  }

  /// Returns the hash from l to r.
  ///
  /// Time Complexity: O(1) -> Actually O(number_of_primes)
  pair<int, int> query(const int l, const int r) {
    assert(0 <= l), assert(l <= r), assert(r < this->n);
    return hash_query(l, r);
  }
};
```

9.6. Kmp

```
/// Builds the pi array for the KMP algorithm.
///
/// Time Complexity: O(n)
vector<int> pi(const string &pat) {
  vector<int> ans(pat.size() + 1, -1);
  int i = 0, j = -1;
  while (i < pat.size()) {
    while (j >= 0 && pat[i] != pat[j])
      j = ans[j];
    ++i, ++j;
    ans[i] = j;
  }
  return ans;
}

/// Returns the occurrences of a pattern in a text.
///
/// Time Complexity: O(n + m)
vector<int> kmp(const string &txt, const string &pat) {
  vector<int> p = pi(pat);
  vector<int> ans;

  for (int i = 0, j = 0; i < txt.size(); ++i) {
    while (j >= 0 && pat[j] != txt[i])
      j = p[j];
    if (++j == pat.size()) {
      ans.emplace_back(i);
      j = p[j];
    }
  }
  return ans;
}
```

9.7. Lcs K Strings

```
// Make the change below in SuffixArray code.
int MaximumNumberOfStrings;

void build_suffix_array() {
  vector<pair<Rank, int>> ranks(this->n + 1);
  vector<int> arr;

  for (int i = 1, separators = 0; i <= n; i++)
    if(this->s[i] > 0) {
      ranks[i] = pair<Rank, int>(Rank((int)this->s[i] +
    MaximumNumberOfStrings, 0), i);
      this->s[i] += MaximumNumberOfStrings;
    } else {
      ranks[i] = pair<Rank, int>(Rank(separators, 0), i);
      this->s[i] = separators;
      separators++;
    }

  RadixSort::sort_pairs(ranks, 256 + MaximumNumberOfStrings);
  ...
}

/// Program to find the LCS between k different strings.
///
/// Time Complexity: O(n*log(n))
/// Space Complexity: O(n*log(n))
int main() {
  int n;

```

```cpp
  cin >> n;

  MaximumNumberOfStrings = n;

  vector<string> arr(n);

  int sum = 0;
  for(string &x: arr) {
    cin >> x;
    sum += x.size() + 1;
  }

  string concat;
  vector<int> ind(sum + 1);
  int cnt = 0;
  for(string &x: arr) {
    if(concat.size())
      concat += (char)cnt;
    concat += x;
  }

  cnt = 0;
  for(int i = 0; i < concat.size(); i++) {
    ind[i + 1] = cnt;
    if(concat[i] < MaximumNumberOfStrings)
      cnt++;
  }

  Suffix_Array say(concat);
  vector<int> sa = say.get_suffix_array();
  Sparse_Table spt(say.get_lcp());

  vector<int> freq(n);
  int cnt1 = 0;

  /// Ignore separators
  int i = n, j = n - 1;
  int ans = 0;

  while(true) {

    if(cnt1 == n) {

      ans = max(ans, spt.query(i, j - 1));

      int idx = ind[sa[i]];
      freq[idx]--;
      if(freq[idx] == 0)
        cnt1--;
      i++;
    } else if(j == (int)sa.size() - 1)
      break;
    else {
      j++;
      int idx = ind[sa[j]];
      freq[idx]++;
      if(freq[idx] == 1)
        cnt1++;
    }
  }

  cout << ans << endl;
}
```

9.8. Lexicographically Smallest Rotation

```cpp
int booth(string &s) {
  s += s;
  int n = s.size();

  vector<int> f(n, -1);
  int k = 0;
  for(int j = 1; j < n; j++) {
    int sj = s[j];
    int i = f[j - k - 1];
    while(i != -1 && sj != s[k + i + 1]) {
      if(sj < s[k + i + 1])
        k = j - i - 1;
      i = f[i];
    }
    if(sj != s[k + i + 1]) {
      if(sj < s[k])
        k = j;
      f[j - k] = -1;
    }
    else
      f[j - k] = i + 1;
  }
  return k;
}
```

9.9. Manacher (Longest Palindrome)

```cpp
// https://medium.com/hackernoon/manachers-algorithm-explained-longest-palindromic-s

/// Create a string containing '#' characters between any two characters.
string get_modified_string(string &s){
  string ret;
  for(int i = 0; i < s.size(); i++){
    ret.push_back('#');
    ret.push_back(s[i]);
  }
  ret.push_back('#');
  return ret;
}

/// Returns the first occurence of the longest palindrome based on the lps array.
///
/// Time Complexity: O(n)
string get_best(const int max_len, const string &str, const vector<int> &lps) {
  for(int i = 0; i < lps.size(); i++) {
    if(lps[i] == max_len) {
      string ans;
      int cnt = max_len / 2;
      int io = i - 1;
      while(cnt) {
        if(str[io] != '#') {
          ans += str[io];
          cnt--;
        }
        io--;
      }
      reverse(ans.begin(), ans.end());
      if(str[i] != '#')
        ans += str[i];
```

```cpp
      cnt = max_len / 2;
      io = i + 1;
      while(cnt) {
        if(str[io] != '#') {
          ans += str[io];
          cnt--;
        }
        io++;
      }
      return ans;
    }
  }
}

/// Returns a pair containing the size of the longest palindrome and the first occurence of it.
///
/// Time Complexity: O(n)
pair<int, string> manacher(string &s) {
  int n = s.size();
  string str = get_modified_string(s);
  int len = (2 * n) + 1;
  //the i-th index contains the longest palindromic substring with the i-th char as the center
  vector<int> lps(len);
  int c = 0; //stores the center of the longest palindromic substring until now
  int r = 0; //stores the right boundary of the longest palindromic substring until now
  int max_len = 0;
  for(int i = 0; i < len; i++) {
    //get mirror index of i
    int mirror = (2 * c) - i;

    //see if the mirror of i is expanding beyond the left boundary of current longest palindrome at center c
    //if it is, then take r - i as lps[i]
    //else take lps[mirror] as lps[i]
    if(i < r)
      lps[i] = min(r - i, lps[mirror]);

    //expand at i
    int a = i + (1 + lps[i]);
    int b = i - (1 + lps[i]);
    while(a < len && b >= 0 && str[a] == str[b]) {
      lps[i]++;
      a++;
      b--;
    }

    //check if the expanded palindrome at i is expanding beyond the right boundary of current longest palindrome at center c
    //if it is, the new center is i
    if(i + lps[i] > r) {
      c = i;
      r = i + lps[i];

      if(lps[i] > max_len) //update max_len
        max_len = lps[i];
    }
  }

  return make_pair(max_len, get_best(max_len, str, lps));
}
```

9.10. Suffix Array

```cpp
// To use the compare method use the macro below.
#define BUILD_TABLE

namespace RadixSort {
/// Sorts the array arr stably in ascending order.
///
/// Time Complexity: O(n + max_element)
/// Space Complexity: O(n + max_element)
template <typename T>
void sort(vector<T> &arr, const int max_element, int (*get_key)(T &),
          const int begin = 0) {
  const int n = arr.size();
  vector<T> new_order(n);
  vector<int> count(max_element + 1, 0);

  for (int i = begin; i < n; ++i)
    ++count[get_key(arr[i])];

  for (int i = 1; i <= max_element; ++i)
    count[i] += count[i - 1];

  for (int i = n - 1; i >= begin; --i) {
    new_order[count[get_key(arr[i])] - (begin == 0)] = arr[i];
    --count[get_key(arr[i])];
  }

  arr = move(new_order);
}

/// Sorts an array by their pair of ranks stably in ascending order.
template <typename T> void sort_pairs(vector<T> &arr, const int rank_size) {
  // sort by the second rank
  RadixSort::sort<T>(
      arr, rank_size, [](T &item) { return item.first.second; }, 0);

  // sort by the first rank
  RadixSort::sort<T>(
      arr, rank_size, [](T &item) { return item.first.first; }, 0);
}
} // namespace RadixSort

/// It is indexed by 0.
/// Let the given string be "banana".
///
/// 0 banana                          5 a
/// 1 anana     Sort the Suffixes     3 ana
/// 2 nana      ---------------->     1 anana
/// 3 ana        alphabetically       0 banana
/// 4 na                              4 na
/// 5 a                               2 nana
/// So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
///
/// LCP
/// 1 a
/// 3 ana
/// 0 anana
/// 0 banana
/// 2 na
/// 0 nana (The last position will always be zero)
/// So the LCP for "banana" is {1, 3, 0, 0, 2, 0}
class Suffix_Array {
private:
  const string s;
```

```
  const int n;

  typedef pair<int, int> Rank;
#ifdef BUILD_TABLE
  vector<vector<int>> rank_table;
  const vector<int> log_array = build_log_array();
#endif

public:
  Suffix_Array(const string &s) : n(s.size()), s(s) {}

private:
  vector<int> build_log_array() {
    vector<int> log_array(this->n + 1, 0);
    for (int i = 2; i <= this->n; ++i)
      log_array[i] = log_array[i / 2] + 1;
    return log_array;
  }

  static void build_ranks(const vector<pair<Rank, int>> &ranks,
                          vector<int> &ret) {
    // The vector containing the ranks will be present at ret
    ret[ranks[0].second] = 1;
    for (int i = 1; i < ranks.size(); ++i) {
      // if their rank are equal, than their position should be the same
      if (ranks[i - 1].first == ranks[i].first)
        ret[ranks[i].second] = ret[ranks[i - 1].second];
      else
        ret[ranks[i].second] = ret[ranks[i - 1].second] + 1;
    }
  }

  /// Time Complexity: O(n*log(n))
  vector<int> build_suffix_array() {
    // the tuple below represents the rank and the index associated with it
    vector<pair<Rank, int>> ranks(this->n);
    vector<int> arr(this->n);

    for (int i = 0; i < n; ++i)
      ranks[i] = pair<Rank, int>(Rank(s[i], 0), i);

#ifdef BUILD_TABLE
    int rank_table_size = 0;
    this->rank_table.resize(log_array[this->n] + 2);
#endif
    RadixSort::sort_pairs(ranks, 256);
    build_ranks(ranks, arr);

    {
      int jump = 1;
      int max_rank = arr[ranks.back().second];

      // it will be compared intervals a pair of intervals (i, jump-1), (i +
      // jump, i + 2*jump - 1). The variable jump is always a power of 2
#ifdef BUILD_TABLE
      while (jump / 2 < this->n) {
#else
      while (max_rank != this->n) {
#endif
        for (int i = 0; i < this->n; ++i) {
          ranks[i].first.first = arr[i];
          ranks[i].first.second = (i + jump < this->n ? arr[i + jump] : 0);
          ranks[i].second = i;
        }

#ifdef BUILD_TABLE
        // inserting only the ranks in the table
        transform(ranks.begin(), ranks.end(),
                  back_inserter(rank_table[rank_table_size++]),
                  [](pair<Rank, int> &pair) { return pair.first.first; });
#endif
        RadixSort::sort_pairs(ranks, n);
        build_ranks(ranks, arr);

        max_rank = arr[ranks.back().second];
        jump *= 2;
      }
    }

    vector<int> sa(this->n);
    for (int i = 0; i < this->n; ++i)
      sa[arr[i] - 1] = i;
    return sa;
  }

  /// Builds the lcp (Longest Common Prefix) array for the string s.
  /// A value lcp[i] indicates length of the longest common prefix of the
  /// suffixes indexed by i and i + 1. Implementation of the Kasai's
    Algorithm.
  ///
  /// Time Complexity: O(n)
  vector<int> build_lcp() {
    vector<int> lcp(this->n, 0);
    vector<int> inverse_suffix(this->n);

    for (int i = 0; i < this->n; ++i)
      inverse_suffix[sa[i]] = i;

    for (int i = 0, k = 0; i < this->n; ++i) {
      if (inverse_suffix[i] == this->n - 1) {
        k = 0;
      } else {
        int j = sa[inverse_suffix[i] + 1];
        while (i + k < this->n && j + k < this->n && s[i + k] == s[j + k])
          ++k;

        lcp[inverse_suffix[i]] = k;

        if (k > 0)
          --k;
      }
    }

    return lcp;
  }

  int _lcs(const int separator) {
    int ans = 0;
    for (int i = 0; i + 1 < this->sa.size(); ++i) {
      const int left = this->sa[i];
      const int right = this->sa[i + 1];
      if ((left < separator && right > separator) ||
          (left > separator && right < separator))
        ans = max(ans, lcp[i]);
    }
    return ans;
  }

#ifdef BUILD_TABLE
  int _compare(const int i, const int j, const int length) {
```

```
    const int k = this->log_array[length]; // floor log2(length)
    const int jump = length - (1ll << k);

    const pair<int, int> iRank = {
        this->rank_table[k][i],
        (i + jump < this->n ? this->rank_table[k][i + jump] : -1)};
    const pair<int, int> jRank = {
        this->rank_table[k][j],
        (j + jump < this->n ? this->rank_table[k][j + jump] : -1)};
    return iRank == jRank ? 0 : iRank < jRank ? -1 : 1;
  }
#endif

public:
  const vector<int> sa = build_suffix_array();
  const vector<int> lcp = build_lcp();

  /// LCS of two strings A and B. The string s must be initialized in the
  /// constructor as the string (A + '$' + B).
  /// The string A starts at index 1 and ends at index (separator - 1).
  /// The string B starts at index (separator + 1) and ends at the end of the
  /// string.
  ///
  /// Time Complexity: O(n)
  int lcs(const int separator) {
    assert(!isalpha(this->s[separator]) && !isdigit(this->s[separator]));
    return _lcs(separator);
  }

#ifdef BUILD_TABLE
  /// Compares two substrings beginning at indexes i and j of a fixed length.
  ///
  /// Time Complexity: O(1)
  int compare(const int i, const int j, const int length) {
    assert(0 <= i && i < this->n && 0 <= j && j < this->n);
    assert(i + length - 1 < this->n && j + length - 1 < this->n);
    return _compare(i, j, length);
  }
#endif
};
```

9.11. Suffix Array Pessoa

```
// OBS: Suffix Array build code imported from:
// https://github.com/gabrielpessoa1/Biblioteca-Maratona/
//                 blob/master/code/String/SuffixArray.cpp
// Because it's faster.
// Swap the method below with the one in "suffix_array.cpp"

vector<int> build_suffix_array() {
  int n = this->s.size(), c = 0;
  vector<int> temp(n), posBucket(n), bucket(n), bpos(n), out(n);
  for (int i = 0; i < n; i++)
    out[i] = i;
  sort(out.begin(), out.end(),
       [&](int a, int b) { return this->s[a] < this->s[b]; });
  for (int i = 0; i < n; i++) {
    bucket[i] = c;
    if (i + 1 == n || this->s[out[i]] != this->s[out[i + 1]])
      c++;
  }
  for (int h = 1; h < n && c < n; h <<= 1) {
    for (int i = 0; i < n; i++)
      posBucket[out[i]] = bucket[i];
    for (int i = n - 1; i >= 0; i--)
      bpos[bucket[i]] = i;
    for (int i = 0; i < n; i++) {
      if (out[i] >= n - h)
        temp[bpos[bucket[i]]++] = out[i];
    }
    for (int i = 0; i < n; i++) {
      if (out[i] >= h)
        temp[bpos[posBucket[out[i] - h]]++] = out[i] - h;
    }
    c = 0;
    for (int i = 0; i + 1 < n; i++) {
      int a = (bucket[i] != bucket[i + 1]) || (temp[i] >= n - h) ||
              (posBucket[temp[i + 1] + h] != posBucket[temp[i] + h]);
      bucket[i] = c;
      c += a;
    }
    bucket[n - 1] = c++;
    temp.swap(out);
  }
  return out;
}
```

9.12. Trie

```
class Trie {
private:
  static const int INT_LEN = 31;
  // static const int INT_LEN = 63;

public:
  struct Node {
    map<char, Node *> next;
    int id;
    // cnt counts the number of words which pass in that node
    int cnt = 0;
    // word counts the number of words ending at that node
    int word_cnt = 0;

    Node(const int x) : id(x) {}
  };

private:
  int trie_size = 0;
  // contains the next id to be used in a node
  int node_cnt = 0;
  Node *trie_root = this->make_node();

private:
  Node *make_node() { return new Node(node_cnt++); }

  int trie_insert(const string &s) {
    Node *aux = this->root();
    for (const char c : s) {
      if (!aux->next.count(c))
        aux->next[c] = this->make_node();
      aux = aux->next[c];
      ++aux->cnt;
    }
    ++aux->word_cnt;
    ++this->trie_size;
    return aux->id;
  }

```

```cpp
  void trie_erase(const string &s) {
    Node *aux = this->root();
    for (const char c : s) {
      Node *last = aux;
      aux = aux->next[c];
      --aux->cnt;
      if (aux->cnt == 0) {
        last->next.erase(c);
        aux = nullptr;
        break;
      }
    }
    if (aux != nullptr)
      --aux->word_cnt;
    --this->trie_size;
  }

  int trie_count(const string &s) {
    Node *aux = this->root();
    for (const char c : s) {
      if (aux->next.count(c))
        aux = aux->next[c];
      else
        return 0;
    }
    return aux->word_cnt;
  }

  int trie_query_xor_max(const string &s) {
    Node *aux = this->root();
    int ans = 0;
    for (const char c : s) {
      const char inv = (c == '0' ? '1' : '0');
      if (aux->next.count(inv)) {
        ans = (ans << 1ll) | (inv - '0');
        aux = aux->next[inv];
      } else {
        ans = (ans << 1ll) | (c - '0');
        aux = aux->next[c];
      }
    }
    return ans;
  }

public:
  Trie() {}

  Node *root() { return this->trie_root; }

  int size() { return this->trie_size; }

  /// Returns the number of nodes present in the trie.
  int node_count() { return this->node_cnt; }

  /// Inserts s in the trie.
  ///
  /// Returns the id of the last character of the string in the trie.
  ///
  /// Time Complexity: O(s.size())
  int insert(const string &s) { return this->trie_insert(s); }

  /// Inserts the binary representation of x in the trie.
  ///
  /// Time Complexity: O(log x)
  int insert(const int x) {
    assert(x >= 0);
    // converting x to binary representation
    return this->trie_insert(bitset<INT_LEN>(x).to_string());
  }

  /// Removes the string s from the trie.
  ///
  /// Time Complexity: O(s.size())
  void erase(const string &s) { this->trie_erase(s); }

  /// Removes the binary representation of x from the trie.
  ///
  /// Time Complexity: O(log x)
  void erase(const int x) {
    assert(x >= 0);
    // converting x to binary representation
    this->trie_erase(bitset<INT_LEN>(x).to_string());
  }

  /// Returns the number of maximum xor sum with x present in the trie.
  ///
  /// Time Complexity: O(log x)
  int query_xor_max(const int x) {
    assert(x >= 0);
    // converting x to binary representation
    return this->trie_query_xor_max(bitset<INT_LEN>(x).to_string());
  }

  /// Returns the number of strings equal to s present in the trie.
  ///
  /// Time Complexity: O(s.size())
  int count(const string &s) { return this->trie_count(s); }
};
```