BRUTE
UDESC

Eliton Machado da Silva, Enzo de Almeida Rodrigues, Eric Grochowicz,
João Vitor Frölich, João Marcos de Oliveira e Rafael Granza de Mello

3 de janeiro de 2024

Índice

1 Estruturas-de-Dados

1.1 Operation Queue

Fila que armazena o resultado do operatório dos itens.

* Complexidade de tempo (Push): O(1) * Complexidade de tempo (Pop): O(1)

```
template <typename T> struct op_queue {
    stack<pair<T, T>> s1, s2;
    T result;
    T op(T a, T b) {
        return a; // TODO: op to compare
        // min(a, b);
        // gcd(a, b);
        // lca(a, b);
    }
    T get() {
        if (s1.empty() || s2.empty()) {
            return result = s1.empty() ? s2.top().second : s1.top().second;
        } else {
            return result = op(s1.top().second, s2.top().second);
        }
    }
    void add(T element) {
        result = s1.empty() ? element : op(element, s1.top().second);
        s1.push({element, result});
    }
    void remove() {
        if (s2.empty()) {
            while (!s1.empty()) {
                T elem = s1.top().first;
                s1.pop();
                T result = s2.empty() ? elem : op(elem, s2.top().second);
                s2.push({elem, result});
            }
        }
        T remove_elem = s2.top().first;
        s2.pop();
    }
};
```

1.2 Interval Tree

Por Rafael Granza de Mello

Capaz de retornar todos os intervalos que intersectam [L, R]. **L e R inclusos** Contém funções insert(L, R, ID), erase(L, R, ID) , overlaps(L, R) e find(L, R, ID). É necessário inserir e apagar indicando tanto os limites quanto o ID do intervalo.

- Complexidade de tempo: O(N * log(N)).

Podem ser usadas as operações em Set:

- *insert*() - *erase*() - *upper_bound*() - etc

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

```

```
5  struct interval {
6      long long lo, hi, id;
7      bool operator<(const interval &i) const {
8          return lo < i.lo || (lo == i.lo && hi < i.hi) || (lo == i.lo && hi == i.hi
               && id < i.id);
9      }
10 };
11 template <class CNI, class NI, class Cmp_Fn, class Allocator> struct
       intervals_node_update {
12     typedef long long metadata_type;
13     int sz = 0;
14     virtual CNI node_begin() const = 0;
15     virtual CNI node_end() const = 0;
16
17     inline vector<int> overlaps(const long long l, const long long r) {
18         queue<CNI> q;
19         q.push(node_begin());
20         vector<int> vec;
21         while (!q.empty()) {
22             CNI it = q.front();
23             q.pop();
24             if (it == node_end()) { continue; }
25             if (r >= (*it)->lo && l <= (*it)->hi) { vec.push_back((*it)->id); }
26             CNI l_it = it.get_l_child();
27             long long l_max = (l_it == node_end()) ? -INF : l_it.get_metadata();
28             if (l_max >= l) { q.push(l_it); }
29             if ((*it)->lo <= r) { q.push(it.get_r_child()); }
30         }
31         return vec;
32     }
33
34     inline void operator()(NI it, CNI end_it) {
35         const long long l_max = (it.get_l_child() == end_it) ? -INF :
               it.get_l_child().get_metadata();
36         const long long r_max = (it.get_r_child() == end_it) ? -INF :
               it.get_r_child().get_metadata();
37         const_cast<long long &>(it.get_metadata()) = max((*it)->hi, max(l_max,
               r_max));
38     }
39 };
40 typedef tree<interval, null_type, less<interval>, rb_tree_tag,
       intervals_node_update> interval_tree;
```

1.3 Segment Tree

[Seg Tree](seg_tree.cpp) Implementação padrão de Seg Tree

- Complexidade de tempo (Pré-processamento): O(N) - Complexidade de tempo (Consulta em intervalo): O(log(N)) - Complexidade de tempo (Update em ponto): O(log(N)) - Complexidade de espaço: 4 *N = O(N)

[Seg Tree Lazy](seg_tree_lazy.cpp) Implementação padrão de Seg Tree com lazy update

- Complexidade de tempo (Pré-processamento): O(N) - Complexidade de tempo (Consulta em intervalo): O(log(N)) - Complexidade de tempo (Update em ponto): O(log(N)) - Complexidade de tempo (Update em intervalo): O(log(N)) - Complexidade de espaço: 2 *4 *N = O(N)

[Sparse Seg Tree](seg_tree_sparse.cpp) Seg Tree Esparsa:

- Complexidade de tempo (Pré-processamento): O(1) - Complexidade de tempo (Consulta em inter-

valo): O(log(N)) - Complexidade de tempo (Update em ponto): O(log(N))

[Persistent Seg Tree](seg_tree_persistent.cpp) Seg Tree Esparsa com histórico de Updates:

- Complexidade de tempo (Pré-processamento): O(N *log(N)) - Complexidade de tempo (Consulta em intervalo): O(log(N)) - Complexidade de tempo (Update em ponto): O(log(N)) - **Para fazer consulta em um tempo específico basta indicar o tempo na query**

[Seg Tree Beats](seg_tree_beats.cpp) Seg Tree que suporta update de maximo e query de soma

- Complexidade de tempo (Pré-processamento): O(N) - Complexidade de tempo (Consulta em intervalo): O(log(N)) - Complexidade de tempo (Update em ponto): O(log(N)) - Complexidade de tempo (Update em intervalo): O(log(N)) - Complexidade de espaço: 2 *4 *N = O(N)

[Seg Tree Beats Max and Sum update](seg_tree_beats_max_and_sum_update.cpp) Seg Tree que suporta update de maximo, update de soma e query de soma. Utiliza uma fila de lazy para diferenciar os updates

- Complexidade de tempo (Pré-processamento): O(N) - Complexidade de tempo (Consulta em intervalo): O(log(N)) - Complexidade de tempo (Update em ponto): O(log(N)) - Complexidade de tempo (Update em intervalo): O(log(N)) - Complexidade de espaço: 2 *4 *N = O(N)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define INF 1e9

struct Node {
    int m1 = INF, m2 = INF, cont = 0, lazy = 0;
    ll soma = 0;

    void set(int v) {
        m1 = v;
        cont = 1;
        soma = v;
    }

    void merge(Node a, Node b) {
        m1 = min(a.m1, b.m1);
        m2 = INF;
        if (a.m1 != b.m1) { m2 = min(m2, max(a.m1, b.m1)); }
        if (a.m2 != m1) { m2 = min(m2, a.m2); }
        if (b.m2 != m1) { m2 = min(m2, b.m2); }
        cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
        soma = a.soma + b.soma;
    }

    void print() { printf("%d %d %d %lld %d\n", m1, m2, cont, soma, lazy); }
};

int n, q;
vector<Node> tree;

int le(int n) { return 2 * n + 1; }
int ri(int n) { return 2 * n + 2; }

void push(int n, int esq, int dir) {
    if (tree[n].lazy <= tree[n].m1) { return; }
    tree[n].soma += (ll)abs(tree[n].m1 - tree[n].lazy) * tree[n].cont;
    tree[n].m1 = tree[n].lazy;
    if (esq != dir) {
```

```
41          tree[le(n)].lazy = max(tree[le(n)].lazy, tree[n].lazy);
42          tree[ri(n)].lazy = max(tree[ri(n)].lazy, tree[n].lazy);
43      }
44      tree[n].lazy = 0;
45  }
46
47  void build(int n, int esq, int dir, vector<int> &v) {
48      if (esq == dir) {
49          tree[n].set(v[esq]);
50      } else {
51          int mid = (esq + dir) / 2;
52          build(le(n), esq, mid, v);
53          build(ri(n), mid + 1, dir, v);
54          tree[n].merge(tree[le(n)], tree[ri(n)]);
55      }
56  }
57  void build(vector<int> &v) { build(0, 0, n - 1, v); }
58
59  // ai = max(ai, mi) em [l, r]
60  void update(int n, int esq, int dir, int l, int r, int mi) {
61      push(n, esq, dir);
62      if (esq > r || dir < l || mi <= tree[n].m1) { return; }
63      if (l <= esq && dir <= r && mi < tree[n].m2) {
64          tree[n].lazy = mi;
65          push(n, esq, dir);
66      } else {
67          int mid = (esq + dir) / 2;
68          update(le(n), esq, mid, l, r, mi);
69          update(ri(n), mid + 1, dir, l, r, mi);
70          tree[n].merge(tree[le(n)], tree[ri(n)]);
71      }
72  }
73  void update(int l, int r, int mi) { update(0, 0, n - 1, l, r, mi); }
74
75  // soma de [l, r]
76  int query(int n, int esq, int dir, int l, int r) {
77      push(n, esq, dir);
78      if (esq > r || dir < l) { return 0; }
79      if (l <= esq && dir <= r) { return tree[n].soma; }
80      int mid = (esq + dir) / 2;
81      return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir, l, r);
82  }
83  int query(int l, int r) { return query(0, 0, n - 1, l, r); }
84
85  int main() {
86      cin >> n;
87      tree.assign(4 * n, Node());
88  }
```

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define ll long long
5  #define INF 1e9
6  #define fi first
7  #define se second
8
9  typedef pair<int, int> ii;
10
11 struct Node {
12     int m1 = INF, m2 = INF, cont = 0;
```

```
13      ll soma = 0;
14      queue<ii> lazy;
15
16      void set(int v) {
17          m1 = v;
18          cont = 1;
19          soma = v;
20      }
21
22      void merge(Node a, Node b) {
23          m1 = min(a.m1, b.m1);
24          m2 = INF;
25          if (a.m1 != b.m1) { m2 = min(m2, max(a.m1, b.m1)); }
26          if (a.m2 != m1) { m2 = min(m2, a.m2); }
27          if (b.m2 != m1) { m2 = min(m2, b.m2); }
28          cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
29          soma = a.soma + b.soma;
30      }
31
32      void print() { printf("%d %d %d %lld\n", m1, m2, cont, soma); }
33  };
34
35  int n, q;
36  vector<Node> tree;
37
38  int le(int n) { return 2 * n + 1; }
39  int ri(int n) { return 2 * n + 2; }
40
41  void push(int n, int esq, int dir) {
42      while (!tree[n].lazy.empty()) {
43          ii p = tree[n].lazy.front();
44          tree[n].lazy.pop();
45          int op = p.fi, v = p.se;
46          if (op == 0) {
47              if (v <= tree[n].m1) { continue; }
48              tree[n].soma += (ll)abs(tree[n].m1 - v) * tree[n].cont;
49              tree[n].m1 = v;
50              if (esq != dir) {
51                  tree[le(n)].lazy.push({0, v});
52                  tree[ri(n)].lazy.push({0, v});
53              }
54          } else if (op == 1) {
55              tree[n].soma += v * (dir - esq + 1);
56              tree[n].m1 += v;
57              tree[n].m2 += v;
58              if (esq != dir) {
59                  tree[le(n)].lazy.push({1, v});
60                  tree[ri(n)].lazy.push({1, v});
61              }
62          }
63      }
64  }
65
66  void build(int n, int esq, int dir, vector<int> &v) {
67      if (esq == dir) {
68          tree[n].set(v[esq]);
69      } else {
70          int mid = (esq + dir) / 2;
71          build(le(n), esq, mid, v);
72          build(ri(n), mid + 1, dir, v);
73          tree[n].merge(tree[le(n)], tree[ri(n)]);
```

```
74      }
75  }
76  void build(vector<int> &v) { build(0, 0, n - 1, v); }
77
78  // ai = max(ai, mi) em [l, r]
79  void update(int n, int esq, int dir, int l, int r, int mi) {
80      push(n, esq, dir);
81      if (esq > r || dir < l || mi <= tree[n].m1) { return; }
82      if (l <= esq && dir <= r && mi < tree[n].m2) {
83          tree[n].soma += (ll)abs(tree[n].m1 - mi) * tree[n].cont;
84          tree[n].m1 = mi;
85          if (esq != dir) {
86              tree[le(n)].lazy.push({0, mi});
87              tree[ri(n)].lazy.push({0, mi});
88          }
89      } else {
90          int mid = (esq + dir) / 2;
91          update(le(n), esq, mid, l, r, mi);
92          update(ri(n), mid + 1, dir, l, r, mi);
93          tree[n].merge(tree[le(n)], tree[ri(n)]);
94      }
95  }
96  void update(int l, int r, int mi) { update(0, 0, n - 1, l, r, mi); }
97
98  // soma v em [l, r]
99  void upsoma(int n, int esq, int dir, int l, int r, int v) {
100     push(n, esq, dir);
101     if (esq > r || dir < l) { return; }
102     if (l <= esq && dir <= r) {
103         tree[n].soma += v * (dir - esq + 1);
104         tree[n].m1 += v;
105         tree[n].m2 += v;
106         if (esq != dir) {
107             tree[le(n)].lazy.push({1, v});
108             tree[ri(n)].lazy.push({1, v});
109         }
110     } else {
111         int mid = (esq + dir) / 2;
112         upsoma(le(n), esq, mid, l, r, v);
113         upsoma(ri(n), mid + 1, dir, l, r, v);
114         tree[n].merge(tree[le(n)], tree[ri(n)]);
115     }
116 }
117 void upsoma(int l, int r, int v) { upsoma(0, 0, n - 1, l, r, v); }
118
119 // soma de [l, r]
120 int query(int n, int esq, int dir, int l, int r) {
121     push(n, esq, dir);
122     if (esq > r || dir < l) { return 0; }
123     if (l <= esq && dir <= r) { return tree[n].soma; }
124     int mid = (esq + dir) / 2;
125     return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir, l, r);
126 }
127 int query(int l, int r) { return query(0, 0, n - 1, l, r); }
128
129 int main() {
130     cin >> n;
131     tree.assign(4 * n, Node());
132     build(v);
133 }
```

```cpp
const int SEGMAX = 8e6 + 5; // should be Q * log(DIR-ESQ+1)
const ll ESQ = 0, DIR = 1e9 + 7;

struct seg {
    ll tree[SEGMAX];
    int R[SEGMAX], L[SEGMAX], ptr = 2; // 0 is NULL; 1 is First Root
    ll op(ll a, ll b) { return (a + b) % MOD; }
    int le(int i) {
        if (L[i] == 0) { L[i] = ptr++; }
        return L[i];
    }
    int ri(int i) {
        if (R[i] == 0) { R[i] = ptr++; }
        return R[i];
    }
    ll query(ll l, ll r, int n = 1, ll esq = ESQ, ll dir = DIR) {
        if (r < esq || dir < l) { return 0; }
        if (l <= esq && dir <= r) { return tree[n]; }
        ll mid = (esq + dir) / 2;
        return op(query(l, r, le(n), esq, mid), query(l, r, ri(n), mid + 1, dir));
    }
    void update(ll x, ll v, int n = 1, ll esq = ESQ, ll dir = DIR) {
        if (esq == dir) {
            tree[n] = (tree[n] + v) % MOD;
        } else {
            ll mid = (esq + dir) / 2;
            if (x <= mid) {
                update(x, v, le(n), esq, mid);
            } else {
                update(x, v, ri(n), mid + 1, dir);
            }
            tree[n] = op(tree[le(n)], tree[ri(n)]);
        }
    }
};
```

```cpp
const int MAX = 2505;

int n, m, mat[MAX][MAX], tree[4 * MAX][4 * MAX];

int le(int x) { return 2 * x + 1; }
int ri(int x) { return 2 * x + 2; }

void build_y(int nx, int lx, int rx, int ny, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx) {
            tree[nx][ny] = mat[lx][ly];
        } else {
            tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
        }
    } else {
        int my = (ly + ry) / 2;
        build_y(nx, lx, rx, le(ny), ly, my);
        build_y(nx, lx, rx, ri(ny), my + 1, ry);
        tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
    }
}
void build_x(int nx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x(le(nx), lx, mx);
```

```
26          build_x(ri(nx), mx + 1, rx);
27      }
28      build_y(nx, lx, rx, 0, 0, m - 1);
29  }
30  void build() { build_x(0, 0, n - 1); }
31
32  void update_y(int nx, int lx, int rx, int ny, int ly, int ry, int x, int y, int v)
        {
33      if (ly == ry) {
34          if (lx == rx) {
35              tree[nx][ny] = v;
36          } else {
37              tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
38          }
39      } else {
40          int my = (ly + ry) / 2;
41          if (y <= my) {
42              update_y(nx, lx, rx, le(ny), ly, my, x, y, v);
43          } else {
44              update_y(nx, lx, rx, ri(ny), my + 1, ry, x, y, v);
45          }
46          tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
47      }
48  }
49  void update_x(int nx, int lx, int rx, int x, int y, int v) {
50      if (lx != rx) {
51          int mx = (lx + rx) / 2;
52          if (x <= mx) {
53              update_x(le(nx), lx, mx, x, y, v);
54          } else {
55              update_x(ri(nx), mx + 1, rx, x, y, v);
56          }
57      }
58      update_y(nx, lx, rx, 0, 0, m - 1, x, y, v);
59  }
60  void update(int x, int y, int v) { update_x(0, 0, n - 1, x, y, v); }
61
62  int sum_y(int nx, int ny, int ly, int ry, int qly, int qry) {
63      if (ry < qly || ly > qry) { return 0; }
64      if (qly <= ly && ry <= qry) { return tree[nx][ny]; }
65      int my = (ly + ry) / 2;
66      return sum_y(nx, le(ny), ly, my, qly, qry) + sum_y(nx, ri(ny), my + 1, ry,
            qly, qry);
67  }
68  int sum_x(int nx, int lx, int rx, int qlx, int qrx, int qly, int qry) {
69      if (rx < qlx || lx > qrx) { return 0; }
70      if (qlx <= lx && rx <= qrx) { return sum_y(nx, 0, 0, m - 1, qly, qry); }
71      int mx = (lx + rx) / 2;
72      return sum_x(le(nx), lx, mx, qlx, qrx, qly, qry) + sum_x(ri(nx), mx + 1, rx,
            qlx, qrx, qly, qry);
73  }
74  int sum(int lx, int rx, int ly, int ry) { return sum_x(0, 0, n - 1, lx, rx, ly,
        ry); }
```

```
1   namespace seg {
2       const int MAX = 2e5 + 5;
3       int n;
4       ll tree[4 * MAX];
5       ll merge(ll a, ll b) { return a + b; }
6       int le(int n) { return 2 * n + 1; }
7       int ri(int n) { return 2 * n + 2; }
```

```
8       void build(int n, int esq, int dir, const vector<ll> &v) {
9           if (esq == dir) {
10              tree[n] = v[esq];
11          } else {
12              int mid = (esq + dir) / 2;
13              build(le(n), esq, mid, v);
14              build(ri(n), mid + 1, dir, v);
15              tree[n] = merge(tree[le(n)], tree[ri(n)]);
16          }
17      }
18      void build(const vector<ll> &v) {
19          n = v.size();
20          build(0, 0, n - 1, v);
21      }
22      ll query(int n, int esq, int dir, int l, int r) {
23          if (esq > r || dir < l) { return 0; }
24          if (l <= esq && dir <= r) { return tree[n]; }
25          int mid = (esq + dir) / 2;
26          return merge(query(le(n), esq, mid, l, r), query(ri(n), mid + 1, dir, l,
                r));
27      }
28      ll query(int l, int r) { return query(0, 0, n - 1, l, r); }
29      void update(int n, int esq, int dir, int x, ll v) {
30          if (esq > x || dir < x) { return; }
31          if (esq == dir) {
32              tree[n] = v;
33          } else {
34              int mid = (esq + dir) / 2;
35              if (x <= mid) {
36                  update(le(n), esq, mid, x, v);
37              } else {
38                  update(ri(n), mid + 1, dir, x, v);
39              }
40              tree[n] = merge(tree[le(n)], tree[ri(n)]);
41          }
42      }
43      void update(int x, ll v) { update(0, 0, n - 1, x, v); }
44  }
```

```
1   namespace seg {
2       const int MAX = 1e5 + 5;
3       int n;
4       ll tree[4 * MAX];
5       ll merge(ll a, ll b) { return max(a, b); }
6       int le(int n) { return 2 * n + 1; }
7       int ri(int n) { return 2 * n + 2; }
8       void build(int n, int esq, int dir, const vector<ll> &v) {
9           if (esq == dir) {
10              tree[n] = v[esq];
11          } else {
12              int mid = (esq + dir) / 2;
13              build(le(n), esq, mid, v);
14              build(ri(n), mid + 1, dir, v);
15              tree[n] = merge(tree[le(n)], tree[ri(n)]);
16          }
17      }
18      void build(const vector<ll> &v) {
19          n = v.size();
20          build(0, 0, n - 1, v);
21      }
22      // find fist index greater than k in [l, r]
```

```
23      ll query(int n, int esq, int dir, int l, int r, ll k) {
24          if (esq > r || dir < l) { return -1; }
25          if (l <= esq && dir <= r) {
26              if (tree[n] < k) { return -1; }
27              while (esq != dir) {
28                  int mid = (esq + dir) / 2;
29                  if (tree[le(n)] >= k) {
30                      n = le(n), dir = mid;
31                  } else {
32                      n = ri(n), esq = mid + 1;
33                  }
34              }
35              return esq;
36          }
37          int mid = (esq + dir) / 2;
38          int res = query(le(n), esq, mid, l, r, k);
39          if (res != -1) { return res; }
40          return query(ri(n), mid + 1, dir, l, r, k);
41      }
42      ll query(int l, int r, ll k) { return query(0, 0, n - 1, l, r, k); }
43      void update(int n, int esq, int dir, int x, ll v) {
44          if (esq > x || dir < x) { return; }
45          if (esq == dir) {
46              tree[n] = v;
47          } else {
48              int mid = (esq + dir) / 2;
49              if (x <= mid) {
50                  update(le(n), esq, mid, x, v);
51              } else {
52                  update(ri(n), mid + 1, dir, x, v);
53              }
54              tree[n] = merge(tree[le(n)], tree[ri(n)]);
55          }
56      }
57      void update(int x, ll v) { update(0, 0, n - 1, x, v); }
58  }
```

```
1   struct SegTree {
2       int n;
3       vector<int> tree;
4
5       SegTree(int n) : n(n) { tree.assign(4 * n, 0); }
6
7       int le(int n) { return 2 * n + 1; }
8       int ri(int n) { return 2 * n + 2; }
9
10      int query(int n, int esq, int dir, int l, int r) {
11          if (esq > r || dir < l) { return 0; }
12          if (l <= esq && dir <= r) { return tree[n]; }
13          int mid = (esq + dir) / 2;
14          return max(query(le(n), esq, mid, l, r), query(ri(n), mid + 1, dir, l, r));
15      }
16      int query(int l, int r) { return query(0, 0, n - 1, l, r); }
17
18      void update(int n, int esq, int dir, int x, int v) {
19          if (esq > x || dir < x) { return; }
20          if (esq == dir) {
21              tree[n] = v;
22          } else {
23              int mid = (esq + dir) / 2;
24              if (x <= mid) {
```

```
25                  update(le(n), esq, mid, x, v);
26              } else {
27                  update(ri(n), mid + 1, dir, x, v);
28              }
29              tree[n] = max(tree[le(n)], tree[ri(n)]);
30          }
31      }
32      void update(int x, int v) { update(0, 0, n - 1, x, v); }
33  };
```

```
1   namespace seg {
2       const int MAX = 1e5 + 5;
3       struct node {
4           ll pref, suff, sum, best;
5       };
6       node new_node(ll v) { return node{v, v, v, v}; }
7       const node NEUTRAL = {0, 0, 0, 0};
8       node tree[4 * MAX];
9       node merge(node a, node b) {
10          ll pref = max(a.pref, a.sum + b.pref);
11          ll suff = max(b.suff, b.sum + a.suff);
12          ll sum = a.sum + b.sum;
13          ll best = max(a.suff + b.pref, max(a.best, b.best));
14          return node{pref, suff, sum, best};
15      }
16
17      int n;
18      int le(int n) { return 2 * n + 1; }
19      int ri(int n) { return 2 * n + 2; }
20      void build(int n, int esq, int dir, const vector<ll> &v) {
21          if (esq == dir) {
22              tree[n] = new_node(v[esq]);
23          } else {
24              int mid = (esq + dir) / 2;
25              build(le(n), esq, mid, v);
26              build(ri(n), mid + 1, dir, v);
27              tree[n] = merge(tree[le(n)], tree[ri(n)]);
28          }
29      }
30      void build(const vector<ll> &v) {
31          n = v.size();
32          build(0, 0, n - 1, v);
33      }
34      node query(int n, int esq, int dir, int l, int r) {
35          if (esq > r || dir < l) { return NEUTRAL; }
36          if (l <= esq && dir <= r) { return tree[n]; }
37          int mid = (esq + dir) / 2;
38          return merge(query(le(n), esq, mid, l, r), query(ri(n), mid + 1, dir, l,
                r));
39      }
40      ll query(int l, int r) { return query(0, 0, n - 1, l, r).best; }
41      void update(int n, int esq, int dir, int x, ll v) {
42          if (esq > x || dir < x) { return; }
43          if (esq == dir) {
44              tree[n] = new_node(v);
45          } else {
46              int mid = (esq + dir) / 2;
47              if (x <= mid) {
48                  update(le(n), esq, mid, x, v);
49              } else {
50                  update(ri(n), mid + 1, dir, x, v);
```

```
            }
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void update(int x, ll v) { update(0, 0, n - 1, x, v); }
}
```

```
namespace seg {
    const int MAX = 2e5 + 5;
    const ll NEUTRAL = 0; // merge(a, neutral) = a
    ll merge(ll a, ll b) { return a + b; }
    int sz; // size of the array
    ll tree[4 * MAX], lazy[4 * MAX];
    int le(int n) { return 2 * n + 1; }
    int ri(int n) { return 2 * n + 2; }
    void push(int n, int esq, int dir) {
        if (lazy[n] == 0) { return; }
        tree[n] += lazy[n] * (dir - esq + 1);
        if (esq != dir) {
            lazy[le(n)] += lazy[n];
            lazy[ri(n)] += lazy[n];
        }
        lazy[n] = 0;
    }
    void build(span<const ll> v, int n, int esq, int dir) {
        if (esq == dir) {
            tree[n] = v[esq];
        } else {
            int mid = (esq + dir) / 2;
            build(v, le(n), esq, mid);
            build(v, ri(n), mid + 1, dir);
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
    void build(span<const ll> v) {
        sz = v.size();
        build(v, 0, 0, sz - 1);
    }
    ll query(int l, int r, int n = 0, int esq = 0, int dir = sz - 1) {
        push(n, esq, dir);
        if (esq > r || dir < l) { return NEUTRAL; }
        if (l <= esq && dir <= r) { return tree[n]; }
        int mid = (esq + dir) / 2;
        return merge(query(l, r, le(n), esq, mid), query(l, r, ri(n), mid + 1,
            dir));
    }
    void update(int l, int r, ll v, int n = 0, int esq = 0, int dir = sz - 1) {
        push(n, esq, dir);
        if (esq > r || dir < l) { return; }
        if (l <= esq && dir <= r) {
            lazy[n] += v;
            push(n, esq, dir);
        } else {
            int mid = (esq + dir) / 2;
            update(l, r, v, le(n), esq, mid);
            update(l, r, v, ri(n), mid + 1, dir);
            tree[n] = merge(tree[le(n)], tree[ri(n)]);
        }
    }
}
```

```cpp
namespace seg {
    const ll ESQ = 0, DIR = 1e9 + 7;
    struct node {
        ll v = 0;
        node *l = NULL, *r = NULL;
        node() { }
        node(ll v) : v(v) { }
        node(node *l, node *r) : l(l), r(r) { v = l->v + r->v; }
        void apply() {
            if (l == NULL) { l = new node(); }
            if (r == NULL) { r = new node(); }
        }
    };
    vector<node *> roots;
    void build() { roots.push_back(new node()); }
    void push(node *n, int esq, int dir) {
        if (esq != dir) { n->apply(); }
    }
    // sum v on x
    node *update(node *n, int esq, int dir, int x, int v) {
        push(n, esq, dir);
        if (esq == dir) { return new node(n->v + v); }
        int mid = (esq + dir) / 2;
        if (x <= mid) {
            return new node(update(n->l, esq, mid, x, v), n->r);
        } else {
            return new node(n->l, update(n->r, mid + 1, dir, x, v));
        }
    }
    int update(int root, int pos, int val) {
        node *novo = update(roots[root], ESQ, DIR, pos, val);
        roots.push_back(novo);
        return roots.size() - 1;
    }
    // sum in [L, R]
    ll query(node *n, int esq, int dir, int l, int r) {
        push(n, esq, dir);
        if (esq > r || dir < l) { return 0; }
        if (l <= esq && dir <= r) { return n->v; }
        int mid = (esq + dir) / 2;
        return query(n->l, esq, mid, l, r) + query(n->r, mid + 1, dir, l, r);
    }
    ll query(int root, int l, int r) { return query(roots[root], ESQ, DIR, l, r); }
    // kth min number in [L, R] (l_root can not be 0)
    int kth(node *L, node *R, int esq, int dir, int k) {
        push(L, esq, dir);
        push(R, esq, dir);
        if (esq == dir) { return esq; }
        int mid = (esq + dir) / 2;
        int cont = R->l->v - L->l->v;
        if (cont >= k) {
            return kth(L->l, R->l, esq, mid, k);
        } else {
            return kth(L->r, R->r, mid + 1, dir, k - cont);
        }
    }
    int kth(int l_root, int r_root, int k) { return kth(roots[l_root - 1],
        roots[r_root], ESQ, DIR, k); }
};
```

1.4 Disjoint Sparse Table

Resolve Query de range para qualquer operação associativa em **O(1)**.

Pré-processamento em **O(Nlog(N))**

```
struct dst {
    const int neutral = 1;
#define comp(a, b) (a | b)
    vector<vector<int>> t;
    dst(vector<int> v) {
        int n, k, sz = v.size();
        for (n = 1, k = 0; n < sz; n <<= 1, k++)
            ;
        t.assign(k, vector<int>(n));
        for (int i = 0; i < n; i++) { t[0][i] = i < sz ? v[i] : neutral; }
        for (int j = 0, len = 1; j <= k; j++, len <<= 1) {
            for (int s = len; s < n; s += (len << 1)) {
                t[j][s] = v[s];
                t[j][s - 1] = v[s - 1];
                for (int i = 1; i < len; i++) {
                    t[j][s + i] = comp(t[j][s + i - 1], v[s + i]);
                    t[j][s - 1 - i] = comp(v[s - 1 - i], t[j][s - i]);
                }
            }
        }
    }
    int query(int l, int r) {
        if (l == r) { return t[0][r]; }
        int i = 31 - __builtin_clz(l ^ r);
        return comp(t[i][l], t[i][r]);
    }
};
```

1.5 Operation Stack

Pilha que armazena o resultado do operatório dos itens.

* Complexidade de tempo (Push): O(1) * Complexidade de tempo (Pop): O(1)

```
template <typename T> struct op_stack {
    stack<pair<T, T>> st;
    T result;
    T op(T a, T b) {
        return a; // TODO: op to compare
        // min(a, b);
        // gcd(a, b);
        // lca(a, b);
    }
    T get() { return result = st.top().second; }
    void add(T element) {
        result = st.empty() ? element : op(element, st.top().second);
        st.push({element, result});
    }
    void remove() {
        T removed_element = st.top().first;
        st.pop();
    }
};
```

1.6 Fenwick Tree

Consultas e atualizações de soma em intervalo.

O vetor precisa obrigatoriamente estar indexado em 1.

* Complexidade de tempo (Pre-processamento): $O(N * log(N))$ * Complexidade de tempo (Consulta em intervalo): $O(log(N))$ * Complexidade de tempo (Update em ponto): $O(log(N))$ * Complexidade de espaço: 2 * N = $O(N)$

```
1  struct FenwickTree {
2      int n;
3      vector<int> tree;
4      FenwickTree(int n) : n(n) { tree.assign(n, 0); }
5      FenwickTree(vector<int> v) : FenwickTree(v.size()) {
6          for (size_t i = 1; i < v.size(); i++) { update(i, v[i]); }
7      }
8      int lsONE(int x) { return x & (-x); }
9      int query(int x) {
10         int soma = 0;
11         for (; x > 0; x -= lsONE(x)) { soma += tree[x]; }
12         return soma;
13     }
14     int query(int l, int r) { return query(r) - query(l - 1); }
15     void update(int x, int v) {
16         for (; x < n; x += lsONE(x)) { tree[x] += v; }
17     }
18 };
```

1.7 Disjoint Set Union

[DSU Simples](dsu.cpp) Estrutura que trata conjuntos. Verifica se dois itens pertencem a um mesmo grupo.

- Complexidade de tempo: O(1) amortizado.

Une grupos.

- Complexidade de tempo: O(1) amortizado.

[DSU Bipartido](bipartite_dsu.cpp) DSU para grafo bipartido, é possível verificar se uma aresta é possível antes de adicioná-la. Para todas as operações:

- Complexidade de tempo: O(1) amortizado.

[DSU com Rollback](rollback_dsu.cpp) Desfaz as últimas K uniões

- Complexidade de tempo: O(K).

É possivel usar um checkpoint, bastando chamar **rollback()** para ir até o último checkpoint. O rollback não altera a complexidade, uma vez que K <= queries. **Só funciona sem compressão de caminho**

- Complexidade de tempo: O(log(N))

[DSU Completo](full_dsu.cpp) DSU com capacidade de adicionar e remover vértices. **EXTREMAMENTE PODEROSO!** Funciona de maneira off-line, recebendo as operações e dando as respostas das consultas no retorno da função **solve()**

- Complexidade de tempo: O(Q * log(Q) * log(N)); Onde Q é o número de consultas e N o número de nodos

Roda em 0,6ms para $3 * 10^5$ queries e nodos com printf e scanf. Possivelmente aguenta 10^6 em 3s

```
struct DSU {
    vector<int> pa, sz;
    DSU(int n) : pa(n + 1), sz(n + 1, 1) { iota(pa.begin(), pa.end(), 0); }
    int root(int a) { return pa[a] = (a == pa[a] ? a : root(pa[a])); }
    bool find(int a, int b) { return root(a) == root(b); }
    void uni(int a, int b) {
        int ra = root(a), rb = root(b);
        if (ra == rb) { return; }
        if (sz[ra] > sz[rb]) { swap(ra, rb); }
        pa[ra] = rb;
        sz[rb] += sz[ra];
    }
};
```

```
struct rollback_dsu {
    struct change {
        int node, old_size;
    };
    stack<change> changes;
    vector<int> parent, size;
    int number_of_sets;

    rollback_dsu(int n) {
        size.resize(n + 5, 1);
        number_of_sets = n;
        for (int i = 0; i < n + 5; ++i) { parent.push_back(i); }
    }

    int get(int a) { return (a == parent[a]) ? a : get(parent[a]); }
    bool same(int a, int b) { return get(a) == get(b); }
    void checkpoint() { changes.push({-2, 0}); }

    void join(int a, int b) {
        a = get(a);
        b = get(b);
        if (a == b) {
            changes.push({-1, -1});
            return;
        }
        if (size[a] > size[b]) { swap(a, b); }
        changes.push({a, size[b]});
        parent[a] = b;
        size[b] += size[a];
        --number_of_sets;
    }

    void rollback(int qnt = 1 << 31) {
        for (int i = 0; i < qnt; ++i) {
            auto ch = changes.top();
            changes.pop();
            if (ch.node == -1) { continue; }
            if (ch.node == -2) {
                if (qnt == 1 << 31) { break; }
                --i;
                continue;
            }
            size[parent[ch.node]] = ch.old_size;
            parent[ch.node] = ch.node;
            ++number_of_sets;
        }
```

```
    }
};
```

```
struct bipartite_dsu {
    vector<int> parent;
    vector<int> color;
    int size;
    bipartite_dsu(int n) {
        size = n;
        color.resize(n + 5, 0);
        for (int i = 0; i < n + 5; ++i) { parent.push_back(i); }
    }

    pair<int, bool> get(int a) {
        if (parent[a] == a) { return {a, 0}; }
        auto val = get(parent[a]);
        parent[a] = val.fi;
        color[a] = (color[a] + val.se) % 2;
        return {parent[a], color[a]};
    }

    bool same_color(int a, int b) {
        get(a);
        get(b);
        return color[a] == color[b];
    }
    bool same_group(int a, int b) {
        get(a);
        get(b);
        return parent[a] == parent[b];
    }
    bool possible_edge(int a, int b) { return !same_color(a, b) || !same_group(a,
        b); }

    void join(int a, int b) {
        auto val_a = get(a), val_b = get(b);
        parent[val_a.fi] = val_b.fi;
        color[val_a.fi] = (val_a.se + val_b.se + 1) % 2;
    }
};
```

```
struct full_dsu {
    struct change {
        int node, old_size;
    };
    struct query {
        int l, r, u, v, type;
    };
    stack<change> changes;
    map<pair<int, int>, vector<query>> edges;
    vector<query> queries;
    vector<int> parent, size;
    int number_of_sets, time;

    full_dsu(int n) {
        time = 0;
        size.resize(n + 5, 1);
        number_of_sets = n;
        loop(i, 0, n + 5) parent.push_back(i);
    }

```

```
21      int get(int a) { return (parent[a] == a ? a : get(parent[a])); }
22      bool same(int a, int b) { return get(a) == get(b); }
23      void checkpoint() { changes.push({-2, 0}); }
24
25      void join(int a, int b) {
26          a = get(a);
27          b = get(b);
28          if (a == b) { return; }
29          if (size[a] > size[b]) { swap(a, b); }
30          changes.push({a, size[b]});
31          parent[a] = b;
32          size[b] += size[a];
33          --number_of_sets;
34      }
35
36      void rollback() {
37          while (!changes.empty()) {
38              auto ch = changes.top();
39              changes.pop();
40              if (ch.node == -2) { break; }
41              size[parent[ch.node]] = ch.old_size;
42              parent[ch.node] = ch.node;
43              ++number_of_sets;
44          }
45      }
46
47      void ord(int &a, int &b) {
48          if (a > b) { swap(a, b); }
49      }
50
51      void add(int u, int v) {
52          ord(u, v);
53          edges[{u, v}].push_back({time++, (int)1e9, u, v, 0});
54      }
55
56      void remove(int u, int v) {
57          ord(u, v);
58          edges[{u, v}].back().r = time++;
59      }
60
61      // consulta se dois vertices estao no mesmo grupo
62      void question(int u, int v) {
63          ord(u, v);
64          queries.push_back({time, time, u, v, 1});
65          ++time;
66      }
67
68      // consulta a quantidade de grupos distintos
69      void question() {
70          queries.push_back({time, time, 0, 0, 1});
71          ++time;
72      }
73
74      vector<int> solve() {
75          for (auto [p, v] : edges) { queries.insert(queries.end(), all(v)); }
76          vector<int> vec(time, -1), ans;
77          run(queries, 0, time, vec);
78          for (int i : vec) {
79              if (i != -1) { ans.push_back(i); }
80          }
81          return ans;
```

```
82      }
83
84      void run(const vector<query> &qrs, int l, int r, vector<int> &ans) {
85          if (l > r) { return; }
86          checkpoint();
87          vector<query> qrs_aux;
88          for (auto &q : qrs) {
89              if (!q.type && q.l <= l && r <= q.r) {
90                  join(q.u, q.v);
91              } else if (r < q.l || l > q.r) {
92                  continue;
93              } else {
94                  qrs_aux.push_back(q);
95              }
96          }
97          if (l == r) {
98              for (auto &q : qrs) {
99                  if (q.type && q.l == l) {
100                     ans[l] = number_of_sets; // numero de grupos nesse tempo
101                     // ans[l] = same(q.u, q.v); // se u e v estao no mesmo grupo
102                 }
103             }
104             rollback();
105             return;
106         }
107         int m = (l + r) / 2;
108         run(qrs_aux, l, m, ans);
109         run(qrs_aux, m + 1, r, ans);
110         rollback();
111     }
112 };
```

1.8 LiChao Tree

Uma árvore de Funções. Retorna o F(x) máximo em um ponto X.

Para retornar o minimo deve-se inserir o negativo da função e pegar o negativo do resultado.

Está pronta para usar função linear do tipo F(x) = mx + b.

Funciona para funções com a seguinte propriedade, sejam duas funções f(x) e g(x), uma vez que f(x) ganha/perde de g(x), f(x) vai continuar ganhando/perdendo de g(x), ou seja f(x) e g(x) se intersectam apenas uma vez.

* Complexidade de consulta : O(log(N)) * Complexidade de update: O(log(N))

[LiChao Tree Sparse](lichao_tree_sparse.cpp)

O mesmo que a superior, no entanto suporta consultas com |x| <= 1e18.

* Complexidade de consulta : O(log(tamanho do intervalo)) * Complexidade de update: O(log(tamanho do intervalo))

```
1  typedef long long ll;
2
3  const ll MAXN = 1e5 + 5, INF = 1e18 + 9;
4
5  struct Line {
6      ll a, b = -INF;
7      ll operator()(ll x) { return a * x + b; }
8  } tree[4 * MAXN];
```

```
9
10 int le(int n) { return 2 * n + 1; }
11 int ri(int n) { return 2 * n + 2; }
12
13 void insert(Line line, int n = 0, int l = 0, int r = MAXN) {
14     int mid = (l + r) / 2;
15     bool bl = line(l) < tree[n](l);
16     bool bm = line(mid) < tree[n](mid);
17     if (!bm) { swap(tree[n], line); }
18     if (l == r) { return; }
19     if (bl != bm) {
20         insert(line, le(n), l, mid);
21     } else {
22         insert(line, ri(n), mid + 1, r);
23     }
24 }
25
26 ll query(int x, int n = 0, int l = 0, int r = MAXN) {
27     if (l == r) { return tree[n](x); }
28     int mid = (l + r) / 2;
29     if (x < mid) {
30         return max(tree[n](x), query(x, le(n), l, mid));
31     } else {
32         return max(tree[n](x), query(x, ri(n), mid + 1, r));
33     }
34 }
```

```
1  typedef long long ll;
2
3  const ll MAXN = 1e5 + 5, INF = 1e18 + 9, MAXR = 1e18;
4
5  struct Line {
6      ll a, b = -INF;
7      __int128 operator()(ll x) { return (__int128)a * x + b; }
8  } tree[4 * MAXN];
9  int idx = 0, L[4 * MAXN], R[4 * MAXN];
10
11 int le(int n) {
12     if (!L[n]) { L[n] = ++idx; }
13     return L[n];
14 }
15 int ri(int n) {
16     if (!R[n]) { R[n] = ++idx; }
17     return R[n];
18 }
19
20 void insert(Line line, int n = 0, ll l = -MAXR, ll r = MAXR) {
21     ll mid = (l + r) / 2;
22     bool bl = line(l) < tree[n](l);
23     bool bm = line(mid) < tree[n](mid);
24     if (!bm) { swap(tree[n], line); }
25     if (l == r) { return; }
26     if (bl != bm) {
27         insert(line, le(n), l, mid);
28     } else {
29         insert(line, ri(n), mid + 1, r);
30     }
31 }
32
33 __int128 query(int x, int n = 0, ll l = -MAXR, ll r = MAXR) {
34     if (l == r) { return tree[n](x); }
```

```
35      ll mid = (l + r) / 2;
36      if (x < mid) {
37          return max(tree[n](x), query(x, le(n), l, mid));
38      } else {
39          return max(tree[n](x), query(x, ri(n), mid + 1, r));
40      }
41  }
```

1.9 KD Fenwick Tree

Fenwick Tree em K dimensoes.

* Complexidade de update: $O(log^k(N))$. * Complexidade de query: $O(log^k(N))$.

```
1  const int MAX = 10;
2  ll tree[MAX][MAX][MAX][MAX][MAX][MAX][MAX][MAX]; // insira a quantidade necessaria
       de dimensoes
3
4  int lsONE(int x) { return x & (-x); }
5
6  ll query(vector<int> s, int pos) {
7      ll sum = 0;
8      while (s[pos] > 0) {
9          if (pos < s.size() - 1) {
10             sum += query(s, pos + 1);
11         } else {
12             sum += tree[s[0]][s[1]][s[2]][s[3]][s[4]][s[5]][s[6]][s[7]];
13         }
14         s[pos] -= lsONE(s[pos]);
15     }
16     return sum;
17 }
18
19 void update(vector<int> s, int pos, int v) {
20     while (s[pos] < MAX + 1) {
21         if (pos < s.size() - 1) {
22             update(s, pos + 1, v);
23         } else {
24             tree[s[0]][s[1]][s[2]][s[3]][s[4]][s[5]][s[6]][s[7]] += v;
25         }
26
27         s[pos] += lsONE(s[pos]);
28     }
29 }
```

1.10 Ordered Set

Pode ser usado como um set normal, a principal diferença são duas novas operações possíveis:

- $find\_by\_order(x)$: retorna o item na posição x. - $order\_of\_key(k)$: retorna o número de elementos menores que k. (o índice de k)

Exemplo

```
1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/trie_policy.hpp>
3
4  using namespace __gnu_pbds;
```

```
typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;

ordered_set X;
X.insert(1);
X.insert(2);
X.insert(4);
X.insert(8);
X.insert(16);

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

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
```

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>

using namespace __gnu_pbds;

template <typename T> typedef tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;
```

1.11 MergeSort Tree

Resolve Queries que envolvam ordenação em Range. (**SEM UPDATE**)

- Complexidade de construção : $O(N * log(N))$ - Complexidade de consulta : $O(log^2(N))$

[MergeSort Tree com Update Pontual](mergesort_tree_update.cpp)

Resolve Queries que envolvam ordenação em Range. (**COM UPDATE**) **1 segundo para vetores de tamanho $3 * 10^5$**

- Complexidade de construção : $O(N * log^2(N))$ - Complexidade de consulta : $O(log^2(N))$ - Complexidade de update : $O(log^2(N))$

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

namespace mergesort {
    typedef tree<ii, null_type, less<ii>, rb_tree_tag,
        tree_order_statistics_node_update> ordered_set;
    const int MAX = 1e5 + 5;

    int n;
    ordered_set mgtree[4 * MAX];
    vi values;

    int le(int n) { return 2 * n + 1; }
    int ri(int n) { return 2 * n + 2; }

```

```
17      ordered_set join(ordered_set set_l, ordered_set set_r) {
18          for (auto v : set_r) { set_l.insert(v); }
19          return set_l;
20      }
21
22      void build(int n, int esq, int dir) {
23          if (esq == dir) {
24              mgtree[n].insert(ii(values[esq], esq));
25          } else {
26              int mid = (esq + dir) / 2;
27              build(le(n), esq, mid);
28              build(ri(n), mid + 1, dir);
29              mgtree[n] = join(mgtree[le(n)], mgtree[ri(n)]);
30          }
31      }
32      void build(vi &v) {
33          n = v.size();
34          values = v;
35          build(0, 0, n - 1);
36      }
37
38      int less(int n, int esq, int dir, int l, int r, int k) {
39          if (esq > r || dir < l) { return 0; }
40          if (l <= esq && dir <= r) { return mgtree[n].order_of_key({k, -1}); }
41          int mid = (esq + dir) / 2;
42          return less(le(n), esq, mid, l, r, k) + less(ri(n), mid + 1, dir, l, r, k);
43      }
44      int less(int l, int r, int k) { return less(0, 0, n - 1, l, r, k); }
45
46      void update(int n, int esq, int dir, int x, int v) {
47          if (esq > x || dir < x) { return; }
48          if (esq == dir) {
49              mgtree[n].clear(), mgtree[n].insert(ii(v, x));
50          } else {
51              int mid = (esq + dir) / 2;
52              if (x <= mid) {
53                  update(le(n), esq, mid, x, v);
54              } else {
55                  update(ri(n), mid + 1, dir, x, v);
56              }
57              mgtree[n].erase(ii(values[x], x));
58              mgtree[n].insert(ii(v, x));
59          }
60      }
61      void update(int x, int v) {
62          update(0, 0, n - 1, x, v);
63          values[x] = v;
64      }
65
66      // ordered_set debug_query(int n, int esq, int dir, int l, int r) {
67      //     if (esq > r || dir < l) return ordered_set();
68      //     if (l <= esq && dir <= r) return mgtree[n];
69      //     int mid = (esq + dir) / 2;
70      //     return join(debug_query(le(n), esq, mid, l, r), debug_query(ri(n),
            mid+1, dir, l, r));
71      // }
72      // ordered_set debug_query(int l, int r) {return debug_query(0, 0, n-1, l, r);}
73
74      // int greater(int n, int esq, int dir, int l, int r, int k) {
75      //     if (esq > r || dir < l) return 0;
76      //     if (l <= esq && dir <= r) return (r-l+1) - mgtree[n].order_of_key({k,
```

```
        1e8});
77      //      int mid = (esq + dir) / 2;
78      //      return greater(le(n), esq, mid, l, r, k) + greater(ri(n), mid+1, dir,
            l, r, k);
79      // }
80      // int greater(int l, int r, int k) {return greater(0, 0, n-1, l, r, k);}
81  };
```

```
 1  namespace mergesort {
 2      const int MAX = 1e5 + 5;
 3
 4      int n;
 5      vi mgtree[4 * MAX];
 6
 7      int le(int n) { return 2 * n + 1; }
 8      int ri(int n) { return 2 * n + 2; }
 9
10      void build(int n, int esq, int dir, vi &v) {
11          mgtree[n] = vi(dir - esq + 1, 0);
12          if (esq == dir) {
13              mgtree[n][0] = v[esq];
14          } else {
15              int mid = (esq + dir) / 2;
16              build(le(n), esq, mid, v);
17              build(ri(n), mid + 1, dir, v);
18              merge(mgtree[le(n)].begin(),
19                    mgtree[le(n)].end(),
20                    mgtree[ri(n)].begin(),
21                    mgtree[ri(n)].end(),
22                    mgtree[n].begin());
23          }
24      }
25      void build(vi &v) {
26          n = v.size();
27          build(0, 0, n - 1, v);
28      }
29
30      int less(int n, int esq, int dir, int l, int r, int k) {
31          if (esq > r || dir < l) { return 0; }
32          if (l <= esq && dir <= r) { return lower_bound(mgtree[n].begin(),
                mgtree[n].end(), k) - mgtree[n].begin(); }
33          int mid = (esq + dir) / 2;
34          return less(le(n), esq, mid, l, r, k) + less(ri(n), mid + 1, dir, l, r, k);
35      }
36      int less(int l, int r, int k) { return less(0, 0, n - 1, l, r, k); }
37
38      // vi debug_query(int n, int esq, int dir, int l, int r) {
39      //      if (esq > r || dir < l) return vi();
40      //      if (l <= esq && dir <= r) return mgtree[n];
41      //      int mid = (esq + dir) / 2;
42      //      auto vl = debug_query(le(n), esq, mid, l, r);
43      //      auto vr = debug_query(ri(n), mid+1, dir, l, r);
44      //      vi ans = vi(vl.size() + vr.size());
45      //      merge(vl.begin(), vl.end(),
46      //          vr.begin(), vr.end(),
47      //          ans.begin());
48      //      return ans;
49      // }
50      // vi debug_query(int l, int r) {return debug_query(0, 0, n-1, l, r);}
51  };
```

1.12 Sparse Table

Responde consultas de maneira eficiente em um conjunto de dados estáticos. Realiza um pré-processamento para diminuir o tempo de cada consulta.

- Complexidade de tempo (Pré-processamento): O(N * log(N)) - Complexidade de tempo (Consulta para operações sem sobreposição amigável): O(N * log(N)) - Complexidade de tempo (Consulta para operações com sobreposição amigável): O(1) - Complexidade de espaço: O(N * log(N))

Exemplo de operações com sobreposição amigável: max(), min(), gcd(), f(x, y) = x

```
1  struct SparseTable {
2      int n, e;
3      vector<vector<int>> st;
4      SparseTable(vector<int> &v) : n(v.size()), e(floor(log2(n))) {
5          st.assign(e + 1, vector<int>(n));
6          for (int i = 0; i < n; i++) { st[0][i] = v[i]; }
7          for (int i = 1; i <= e; i++) {
8              for (int j = 0; j + (1 << i) <= n; j++) { st[i][j] = min(st[i - 1][j],
                   st[i - 1][j + (1 << (i - 1))]); }
9          }
10     }
11     // O(log(N)) Query for non overlap friendly operations
12     int logquery(int l, int r) {
13         int res = 2e9;
14         for (int i = e; i >= 0; i--) {
15             if ((1 << i) <= r - l + 1) {
16                 res = min(res, st[i][l]);
17                 l += 1 << i;
18             }
19         }
20         return res;
21     }
22     // O(1) Query for overlab friendly operations
23     // ex: max(), min(), gcd(), f(x, y) = x
24     int query(int l, int r) {
25         // if (l > r) return 2e9;
26         int i = ilogb(r - l + 1);
27         return min(st[i][l], st[i][r - (1 << i) + 1]);
28     }
29 };
```

<div style=ẗpage-break-after: always;ẗ>

+ **[DSU (Disjoint Set Union)](DSU/)** Consultas e atualizações em grupos de objetos. + **[DST (Disjoint Sparse Table)](Disjoint%20Sparse%20Table)** Consultas diversas em intervalo em **O(1)**. **SEM ATUALIZAÇÂO** + **[Fenwick Tree](Fenwick%20Tree)** Consultas e atualizações de soma em intervalo. **COM ATUALIZAÇÂO** + **[Interval Tree](Interval%20Tree) (Autoral)** Responde quantos intervalos intersectam com outro. **COM ATUALIZAÇÂO** + **[KD Fenwick Tree](KD%20Fenwick%20Tree) Fenwick Tree em K dimensões. **COM ATUALIZAÇÂO** + **[LiChao Tree](LiChao%20Tree)** Retorna valor máximo ou mínimo de um conjunto de retas em um intervalo. **COM ATUALIZAÇÂO** + **[MergeSort Tree](MergeSort%20Tree)** Retorna quantidade de valores em um range que são menores que K. **COM E SEM ATUALIZAÇÂO** + **[Odered Set](Ordered%20Set)** Set com possibilidade de retornar a ordem da chave, e o item em uma posição específica. + **[Operation Stack](Operation%20Stack)/[Queue] Pilha/Fila que armazena o resultado do operatório dos itens. + **[Segment Tree](Segment%20Tree)** Consultas e atualizações diversos em intervalo. + **[Sparse Table](Sparse%20Table/)** Consultas diversas em intervalo. **SEM ATUALIZAÇÂO**

2 Grafos

2.1 Hungarian Algorithm for Bipartite Matching

Resolve o problema de Matching para uma matriz A[n][m], onde $n \leq m$.

A implementação minimiza os custos, para maximizar basta multiplicar os pesos por -1.

A matriz de entrada precisa ser indexada em 1 !!!

O vetor result guarda os pares do matching.

Complexidade de tempo: $O(n^2 * m)$

```
1  const ll INF = 1e18 + 18;
2
3  vector<pair<int, int>> result;
4
5  ll hungarian(int n, int m, vector<vector<int>> &A) {
6      vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
7      for (int i = 1; i <= n; i++) {
8          p[0] = i;
9          int j0 = 0;
10         vector<int> minv(m + 1, INF);
11         vector<char> used(m + 1, false);
12         do {
13             used[j0] = true;
14             ll i0 = p[j0], delta = INF, j1;
15             for (int j = 1; j <= m; j++) {
16                 if (!used[j]) {
17                     int cur = A[i0][j] - u[i0] - v[j];
18                     if (cur < minv[j]) { minv[j] = cur, way[j] = j0; }
19                     if (minv[j] < delta) { delta = minv[j], j1 = j; }
20                 }
21             }
22             for (int j = 0; j <= m; j++) {
23                 if (used[j]) {
24                     u[p[j]] += delta, v[j] -= delta;
25                 } else {
26                     minv[j] -= delta;
27                 }
28             }
29             j0 = j1;
30         } while (p[j0] != 0);
31         do {
32             int j1 = way[j0];
33             p[j0] = p[j1];
34             j0 = j1;
35         } while (j0);
36     }
37     for (int i = 1; i <= m; i++) { result.emplace_back(p[i], i); }
38     return -v[0];
39 }
```

2.2 Stoer-Wagner

O algoritmo de Stoer-Wagner é um algoritmo para resolver o problema de corte mínimo em grafos não direcionados com pesos não negativos. A ideia essencial deste algoritmo é encolher o grafo mesclando os vértices mais intensos até que o grafo contenha apenas dois conjuntos de vértices combinados

Complexidade de tempo: $O(V^3)$

```
1  const int MAXN = 555, INF = 1e9 + 7;
2
3  int n, e, adj[MAXN][MAXN];
4  vector<int> bestCut;
5
6  int mincut() {
7      int bestCost = INF;
8      vector<int> v[MAXN];
9      for (int i = 0; i < n; i++) { v[i].assign(1, i); }
10     int w[MAXN], sel;
11     bool exist[MAXN], added[MAXN];
12     memset(exist, true, sizeof(exist));
13     for (int phase = 0; phase < n - 1; phase++) {
14         memset(added, false, sizeof(added));
15         memset(w, 0, sizeof(w));
16         for (int j = 0, prev; j < n - phase; j++) {
17             sel = -1;
18             for (int i = 0; i < n; i++) {
19                 if (exist[i] && !added[i] && (sel == -1 || w[i] > w[sel])) { sel =
                       i; }
20             }
21             if (j == n - phase - 1) {
22                 if (w[sel] < bestCost) {
23                     bestCost = w[sel];
24                     bestCut = v[sel];
25                 }
26                 v[prev].insert(v[prev].end(), v[sel].begin(), v[sel].end());
27                 for (int i = 0; i < n; i++) { adj[prev][i] = adj[i][prev] +=
                       adj[sel][i]; }
28                 exist[sel] = false;
29             } else {
30                 added[sel] = true;
31                 for (int i = 0; i < n; i++) { w[i] += adj[sel][i]; }
32                 prev = sel;
33             }
34         }
35     }
36     return bestCost;
37 }
```

2.3 LCA

Algoritmo de Lowest Common Ancestor usando EulerTour e Sparse Table

Complexidade de tempo:

- O(Nlog(N)) Preprocessing - O(1) Query LCA

Complexidade de espaço: O(Nlog(N))

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define INF 1e9
5  #define fi first
6  #define se second
7
8  typedef pair<int, int> ii;
9
```

```
10 vector<int> tin, tout;
11 vector<vector<int>> adj;
12 vector<ii> prof;
13 vector<vector<ii>> st;
14
15 int n, timer;
16
17 void SparseTable(vector<ii> &v) {
18     int n = v.size();
19     int e = floor(log2(n));
20     st.assign(e + 1, vector<ii>(n));
21     for (int i = 0; i < n; i++) { st[0][i] = v[i]; }
22     for (int i = 1; i <= e; i++) {
23         for (int j = 0; j + (1 << i) <= n; j++) { st[i][j] = min(st[i - 1][j],
               st[i - 1][j + (1 << (i - 1))]); }
24     }
25 }
26
27 void et_dfs(int u, int p, int h) {
28     tin[u] = timer++;
29     prof.emplace_back(h, u);
30     for (int v : adj[u]) {
31         if (v != p) {
32             et_dfs(v, u, h + 1);
33             prof.emplace_back(h, u);
34         }
35     }
36     tout[u] = timer++;
37 }
38
39 void build(int root = 0) {
40     tin.assign(n, 0);
41     tout.assign(n, 0);
42     prof.clear();
43     timer = 0;
44     et_dfs(root, root, 0);
45     SparseTable(prof);
46 }
47
48 int lca(int u, int v) {
49     int l = tout[u], r = tin[v];
50     if (l > r) { swap(l, r); }
51     int i = floor(log2(r - l + 1));
52     return min(st[i][l], st[i][r - (1 << i) + 1]).se;
53 }
54
55 int main() {
56     cin >> n;
57
58     adj.assign(n, vector<int>(0));
59
60     for (int i = 0; i < n - 1; i++) {
61         int a, b;
62         cin >> a >> b;
63         adj[a].push_back(b);
64         adj[b].push_back(a);
65     }
66
67     build();
68 }
```

2.4 Heavy-Light Decomposition (hld.cpp)

Técnica usada para otimizar a execução de operações em árvores.

- Pré-Processamento: O(N) - Range Query/Update: O(Log(N)) * O(Complexidade de query da estrutura) - Point Query/Update: O(Complexidade de query da estrutura) - LCA: O(Log(N)) - Subtree Query: O(Complexidade de query da estrutura) - Complexidade de espaço: O(N)

```
1  namespace hld {
2      const int MAX = 2e5 + 5;
3      int t, sz[MAX], pos[MAX], pai[MAX], head[MAX];
4      bool e = 0;
5      ll merge(ll a, ll b) { return max(a, b); } // how to merge paths
6      void dfs_sz(int u, int p = -1) {
7          sz[u] = 1;
8          for (int &v : adj[u]) {
9              if (v != p) {
10                 dfs_sz(v, u);
11                 sz[u] += sz[v];
12                 if (sz[v] > sz[adj[u][0]] || adj[u][0] == p) { swap(v, adj[u][0]);
                       }
13             }
14         }
15     }
16     void dfs_hld(int u, int p = -1) {
17         pos[u] = t++;
18         for (int v : adj[u]) {
19             if (v != p) {
20                 pai[v] = u;
21                 head[v] = (v == adj[u][0] ? head[u] : v);
22                 dfs_hld(v, u);
23             }
24         }
25     }
26     void build(int root) {
27         dfs_sz(root);
28         t = 0;
29         pai[root] = root;
30         head[root] = root;
31         dfs_hld(root);
32     }
33     void build(int root, vector<ll> &v) {
34         build(root);
35         vector<ll> aux(v.size());
36         for (int i = 0; i < (int)v.size(); i++) { aux[pos[i]] = v[i]; }
37         seg::build(aux);
38     }
39     void build(int root, vector<i3> &edges) { // use this if weighted edges
40         build(root);
41         e = 1;
42         vector<ll> aux(edges.size() + 1);
43         for (auto [u, v, w] : edges) {
44             if (pos[u] > pos[v]) { swap(u, v); }
45             aux[pos[v]] = w;
46         }
47         seg::build(aux);
48     }
49     ll query(int u, int v) {
50         if (pos[u] > pos[v]) { swap(u, v); }
51         if (head[u] == head[v]) {
52             return seg::query(pos[u] + e, pos[v]);
```

```
53          } else {
54              ll qv = seg::query(pos[head[v]], pos[v]);
55              ll qu = query(u, pai[head[v]]);
56              return merge(qu, qv);
57          }
58      }
59      void update(int u, int v, ll k) {
60          if (pos[u] > pos[v]) { swap(u, v); }
61          if (head[u] == head[v]) {
62              seg::update(pos[u] + e, pos[v], k);
63          } else {
64              seg::update(pos[head[v]], pos[v], k);
65              update(u, pai[head[v]], k);
66          }
67      }
68      int lca(int u, int v) {
69          if (pos[u] > pos[v]) { swap(u, v); }
70          return (head[u] == head[v] ? u : lca(u, pai[head[v]]));
71      }
72      ll query_subtree(int u) { return seg::query(pos[u], pos[u] + sz[u] - 1); }
73  }
```

2.5 Kruskal

Utiliza [DSU](../../Estruturas%20de%20Dados/DSU/dsu.cpp) - (disjoint set union) - para construir MST - (minimum spanning tree)

- Complexidade de tempo (Construção): O(M log N)

```
1   struct Edge {
2       int u, v, w;
3       bool operator<(Edge const &other) { return w < other.w; }
4   };
5
6   vector<Edge> edges, result;
7   int cost;
8
9   struct DSU {
10      vector<int> pa, sz;
11      DSU(int n) {
12          sz.assign(n + 5, 1);
13          for (int i = 0; i < n + 5; i++) { pa.push_back(i); }
14      }
15      int root(int a) { return pa[a] = (a == pa[a] ? a : root(pa[a])); }
16      bool find(int a, int b) { return root(a) == root(b); }
17      void uni(int a, int b) {
18          int ra = root(a), rb = root(b);
19          if (ra == rb) { return; }
20          if (sz[ra] > sz[rb]) { swap(ra, rb); }
21          pa[ra] = rb;
22          sz[rb] += sz[ra];
23      }
24  };
25
26  void kruskal(int m, int n) {
27      DSU dsu(n);
28
29      sort(edges.begin(), edges.end());
30
31      for (Edge e : edges) {
```

```
32          if (!dsu.find(e.u, e.v)) {
33              cost += e.w;
34              result.push_back(e); // remove if need only cost
35              dsu.uni(e.u, e.v);
36          }
37      }
38  }
```

Algoritmo que acha pontes utilizando uma dfs

Complexidade de tempo: O(N + M)

```
1  int n;                          // number of
       nodes
2  vector<vector<int>> adj; // adjacency
       list of graph
3
4  vector<bool> visited;
5  vector<int> tin, low;
6  int timer;
7
8  void dfs(int u, int p = -1) {
9      visited[u] = true;
10     tin[u] = low[u] = timer++;
11     for (int v : adj[u]) {
12         if (v == p) { continue; }
13         if (visited[v]) {
14             low[u] = min(low[u],
                   tin[v]);
15         } else {
16             dfs(v, u);
17             low[u] = min(low[u],
                   low[v]);
18             if (low[v] > tin[u]) {
19                 // edge UV is a bridge
20                 // do_something(u, v)
21             }
22         }
23     }
24 }
25
26 void find_bridges() {
27     timer = 0;
28     visited.assign(n, false);
29     tin.assign(n, -1);
30     low.assign(n, -1);
31     for (int i = 0; i < n; ++i) {
32         if (!visited[i]) { dfs(i); }
33     }
34 }
```

2.6 Binary Lifting

Usa uma sparse table para calcular o k-ésimo ancestral de u. Pode ser usada com o algoritmo de EulerTour para calcular o LCA.

Complexidade de tempo:

- Pré-processamento: O(N * log(N)) - Consulta do k-ésimo ancestral de u: O(log(N)) - LCA: O(log(N))

Complexidade de espaço: O(Nlog(N))

```
1  namespace st {
2      int n, me, timer;
3      vector<int> tin, tout;
4      vector<vector<int>> st;
5      void et_dfs(int u, int p) {
6          tin[u] = ++timer;
7          st[u][0] = p;
8          for (int i = 1; i <= me; i++) { st[u][i] = st[st[u][i - 1]][i - 1]; }
9          for (int v : adj[u]) {
10             if (v != p) { et_dfs(v, u); }
11         }
12         tout[u] = ++timer;
13     }
14     void build(int _n, int root = 0) {
15         n = _n;
16         tin.assign(n, 0);
17         tout.assign(n, 0);
```

```
        timer = 0;
        me = floor(log2(n));
        st.assign(n, vector<int>(me + 1, 0));
        et_dfs(root, root);
    }
    bool is_ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >=
        tout[v]; }
    int lca(int u, int v) {
        if (is_ancestor(u, v)) { return u; }
        if (is_ancestor(v, u)) { return v; }
        for (int i = me; i >= 0; i--) {
            if (!is_ancestor(st[u][i], v)) { u = st[u][i]; }
        }
        return st[u][0];
    }
    int ancestor(int u, int k) { // k-th ancestor of u
        for (int i = me; i >= 0; i--) {
            if ((1 << i) & k) { u = st[u][i]; }
        }
        return u;
    }
}
```

```
namespace st {
    int n, me;
    vector<vector<int>> st;
    void bl_dfs(int u, int p) {
        st[u][0] = p;
        for (int i = 1; i <= me; i++) { st[u][i] = st[st[u][i - 1]][i - 1]; }
        for (int v : adj[u]) {
            if (v != p) { bl_dfs(v, u); }
        }
    }
    void build(int _n, int root = 0) {
        n = _n;
        me = floor(log2(n));
        st.assign(n, vector<int>(me + 1, 0));
        bl_dfs(root, root);
    }
    int ancestor(int u, int k) { // k-th ancestor of u
        for (int i = me; i >= 0; i--) {
            if ((1 << i) & k) { u = st[u][i]; }
        }
        return u;
    }
}
```

2.7 Dijkstra

Computa o menor caminho entre nós de um grafo.

Dijkstra 1:1

Dado dois nós u e v, computa o menor caminho de u para v.

Complexidade de tempo: O((E + V) * log(E))

Dijkstra 1:N

Dado um nó u, computa o menor caminho de u para todos os nós.

Complexidade de tempo: O((E + V) * log(E))

Dijkstra N:N

Computa o menor caminho de todos os nós para todos os nós

Complexidade de tempo: O(V * ((E + V) * log(E)))

```cpp
const int MAX = 505, INF = 1e9 + 9;

vector<ii> adj[MAX];
int dist[MAX][MAX];

void dk(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) { dist[i][j] = INF; }
    }
    for (int s = 0; s < n; s++) {
        priority_queue<ii, vector<ii>, greater<ii>> fila;
        dist[s][s] = 0;
        fila.emplace(dist[s][s], s);
        while (!fila.empty()) {
            auto [d, u] = fila.top();
            fila.pop();
            if (d != dist[s][u]) { continue; }
            for (auto [w, v] : adj[u]) {
                if (dist[s][v] > d + w) {
                    dist[s][v] = d + w;
                    fila.emplace(dist[s][v], v);
                }
            }
        }
    }
}
```

```cpp
const int MAX = 1e5 + 5, INF = 1e9 + 9;

vector<ii> adj[MAX];
int dist[MAX];

void dk(int s) {
    priority_queue<ii, vector<ii>, greater<ii>> fila;
    fill(begin(dist), end(dist), INF);
    dist[s] = 0;
    fila.emplace(dist[s], s);
    while (!fila.empty()) {
        auto [d, u] = fila.top();
        fila.pop();
        if (d != dist[u]) { continue; }
        for (auto [w, v] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                fila.emplace(dist[v], v);
            }
        }
    }
}
```

```cpp
const int MAX = 1e5 + 5, INF = 1e9 + 9;

vector<ii> adj[MAX];
int dist[MAX];

int dk(int s, int t) {
    priority_queue<ii, vector<ii>, greater<ii>> fila;
    fill(begin(dist), end(dist), INF);
    dist[s] = 0;
    fila.emplace(dist[s], s);
    while (!fila.empty()) {
        auto [d, u] = fila.top();
        fila.pop();
        if (u == t) { return dist[t]; }
        if (d != dist[u]) { continue; }
        for (auto [w, v] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                fila.emplace(dist[v], v);
            }
        }
    }
    return -1;
}
```

2.8 Fluxo

Conjunto de algoritmos para calcular o fluxo máximo em problemas relacionados de fluxo

Dinic

Muito útil para grafos bipartidos e para grafos com muitas arestas

Complexidade de tempo: O(V^2 * E), mas em grafo bipartido a complexidade é O(sqrt(V) * E)

Edmonds Karp

Útil para grafos com poucas arestas

Complexidade de tempo: O(V * E^2)

Min Cost Max Flow

Computa o fluxo máximo com custo mínimo

Complexidade de tempo: O(V^2 * E^2)

```cpp
const long long INF = 1e18;

struct FlowEdge {
    int u, v;
    long long cap, flow = 0;
    FlowEdge(int u, int v, long long cap) : u(u), v(v), cap(cap) { }
};

struct EdmondsKarp {
    int n, s, t, m = 0, vistoken = 0;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    vector<int> visto;

    EdmondsKarp(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        visto.resize(n);
    }

    void add_edge(int u, int v, long long cap) {
        edges.emplace_back(u, v, cap);
        edges.emplace_back(v, u, 0);
        adj[u].push_back(m);
        adj[v].push_back(m + 1);
        m += 2;
    }

    int bfs() {
        vistoken++;
        queue<int> fila;
        fila.push(s);
        vector<int> pego(n, -1);
        while (!fila.empty()) {
            int u = fila.front();
            if (u == t) { break; }
            fila.pop();
            visto[u] = vistoken;
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) { continue; }
                int v = edges[id].v;
                if (visto[v] == -1) { continue; }
                fila.push(v);
```

```
43                  pego[v] = id;
44              }
45          }
46          if (pego[t] == -1) { return 0; }
47          long long f = INF;
48          for (int id = pego[t]; id != -1; id = pego[edges[id].u]) { f = min(f,
                edges[id].cap - edges[id].flow); }
49          for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
50              edges[id].flow += f;
51              edges[id ^ 1].flow -= f;
52          }
53          return f;
54      }
55
56      long long flow() {
57          long long maxflow = 0;
58          while (long long f = bfs()) { maxflow += f; }
59          return maxflow;
60      }
61  };
```

```
 1  struct MinCostMaxFlow {
 2      int n, s, t, m = 0;
 3      ll maxflow = 0, mincost = 0;
 4      vector<FlowEdge> edges;
 5      vector<vector<int>> adj;
 6
 7      MinCostMaxFlow(int n, int s, int t) : n(n), s(s), t(t) { adj.resize(n); }
 8
 9      void add_edge(int u, int v, ll cap, ll cost) {
10          edges.emplace_back(u, v, cap, cost);
11          edges.emplace_back(v, u, 0, -cost);
12          adj[u].push_back(m);
13          adj[v].push_back(m + 1);
14          m += 2;
15      }
16
17      bool spfa() {
18          vector<int> pego(n, -1);
19          vector<ll> dis(n, INF);
20          vector<bool> inq(n, false);
21          queue<int> fila;
22          fila.push(s);
23          dis[s] = 0;
24          inq[s] = 1;
25          while (!fila.empty()) {
26              int u = fila.front();
27              fila.pop();
28              inq[u] = false;
29              for (int id : adj[u]) {
30                  if (edges[id].cap - edges[id].flow < 1) { continue; }
31                  int v = edges[id].v;
32                  if (dis[v] > dis[u] + edges[id].cost) {
33                      dis[v] = dis[u] + edges[id].cost;
34                      pego[v] = id;
35                      if (!inq[v]) {
36                          inq[v] = true;
37                          fila.push(v);
38                      }
39                  }
40              }
```

```
        }

        if (pego[t] == -1) { return 0; }
        ll f = INF;
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            f = min(f, edges[id].cap - edges[id].flow);
            mincost += edges[id].cost;
        }
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            edges[id].flow += f;
            edges[id ^ 1].flow -= f;
        }
        maxflow += f;
        return 1;
    }

    ll flow() {
        while (spfa())
            ;
        return maxflow;
    }
};
```

```
typedef long long ll;

const ll INF = 1e18;

struct FlowEdge {
    int u, v;
    ll cap, flow = 0;
    FlowEdge(int u, int v, ll cap) : u(u), v(v), cap(cap) { }
};

struct Dinic {
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, s, t, m = 0;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int u, int v, ll cap) {
        edges.emplace_back(u, v, cap);
        edges.emplace_back(v, u, 0);
        adj[u].push_back(m);
        adj[v].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) { continue; }
                int v = edges[id].v;
```

```
39                  if (level[v] != -1) { continue; }
40                  level[v] = level[u] + 1;
41                  q.push(v);
42              }
43          }
44          return level[t] != -1;
45      }
46
47      ll dfs(int u, ll f) {
48          if (f == 0) { return 0; }
49          if (u == t) { return f; }
50          for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
51              int id = adj[u][cid];
52              int v = edges[id].v;
53              if (level[u] + 1 != level[v] || edges[id].cap - edges[id].flow < 1) {
                    continue; }
54              ll tr = dfs(v, min(f, edges[id].cap - edges[id].flow));
55              if (tr == 0) { continue; }
56              edges[id].flow += tr;
57              edges[id ^ 1].flow -= tr;
58              return tr;
59          }
60          return 0;
61      }
62
63      ll flow() {
64          ll maxflow = 0;
65          while (true) {
66              fill(level.begin(), level.end(), -1);
67              level[s] = 0;
68              q.push(s);
69              if (!bfs()) { break; }
70              fill(ptr.begin(), ptr.end(), 0);
71              while (ll f = dfs(s, INF)) { maxflow += f; }
72          }
73          return maxflow;
74      }
75  };
```

2.9 Inverse Graph

Resolve problemas em que se deseja encontrar as componentes conexas quando são dadas as arestas que não pertencem ao grafo

- Complexidade de tempo: O(N log N + N log M)

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  set<int> nodes;
5  vector<set<int>> adj;
6
7  void bfs(int s) {
8      queue<int> f;
9      f.push(s);
10     nodes.erase(s);
11     set<int> aux;
12     while (!f.empty()) {
13         int x = f.front();
14         f.pop();
```

```
15          for (int y : nodes) {
16              if (adj[x].count(y) == 0) { aux.insert(y); }
17          }
18          for (int y : aux) {
19              f.push(y);
20              nodes.erase(y);
21          }
22          aux.clear();
23      }
24  }
```

2.10 2-SAT

Resolve problema do 2-SAT.

- Complexidade de tempo (caso médio): O(N + M)

N é o número de variáveis e M é o número de cláusulas. A configuração da solução fica guardada no vetor *assignment*.

Em relaçõa ao sinal, tanto faz se 0 liga ou desliga, apenas siga o mesmo padrão.

```
1   struct sat2 {
2       int n;
3       vector<vector<int>> g, gt;
4       vector<bool> used;
5       vector<int> order, comp;
6       vector<bool> assignment;
7
8       // number of variables
9       sat2(int _n) {
10          n = 2 * (_n + 5);
11          g.assign(n, vector<int>());
12          gt.assign(n, vector<int>());
13      }
14      void add_edge(int v, int u, bool v_sign, bool u_sign) {
15          g[2 * v + v_sign].push_back(2 * u + !u_sign);
16          g[2 * u + u_sign].push_back(2 * v + !v_sign);
17          gt[2 * u + !u_sign].push_back(2 * v + v_sign);
18          gt[2 * v + !v_sign].push_back(2 * u + u_sign);
19      }
20      void dfs1(int v) {
21          used[v] = true;
22          for (int u : g[v]) {
23              if (!used[u]) { dfs1(u); }
24          }
25          order.push_back(v);
26      }
27      void dfs2(int v, int cl) {
28          comp[v] = cl;
29          for (int u : gt[v]) {
30              if (comp[u] == -1) { dfs2(u, cl); }
31          }
32      }
33      bool solve() {
34          order.clear();
35          used.assign(n, false);
36          for (int i = 0; i < n; ++i) {
37              if (!used[i]) { dfs1(i); }
38          }
```

```
39
40          comp.assign(n, -1);
41          for (int i = 0, j = 0; i < n; ++i) {
42              int v = order[n - i - 1];
43              if (comp[v] == -1) { dfs2(v, j++); }
44          }
45
46          assignment.assign(n / 2, false);
47          for (int i = 0; i < n; i += 2) {
48              if (comp[i] == comp[i + 1]) { return false; }
49              assignment[i / 2] = comp[i] > comp[i + 1];
50          }
51          return true;
52      }
53  };
```

2.11 Graph Center

Encontra o centro e o diâmetro de um grafo

Complexidade de tempo: O(N)

```
1   const int INF = 1e9 + 9;
2
3   vector<vector<int>> adj;
4
5   struct GraphCenter {
6       int n, diam = 0;
7       vector<int> centros, dist, pai;
8       int bfs(int s) {
9           queue<int> q;
10          q.push(s);
11          dist.assign(n + 5, INF);
12          pai.assign(n + 5, -1);
13          dist[s] = 0;
14          int maxidist = 0, maxinode = 0;
15          while (!q.empty()) {
16              int u = q.front();
17              q.pop();
18              if (dist[u] >= maxidist) { maxidist = dist[u], maxinode = u; }
19              for (int v : adj[u]) {
20                  if (dist[u] + 1 < dist[v]) {
21                      dist[v] = dist[u] + 1;
22                      pai[v] = u;
23                      q.push(v);
24                  }
25              }
26          }
27          diam = max(diam, maxidist);
28          return maxinode;
29      }
30      GraphCenter(int st = 0) : n(adj.size()) {
31          int d1 = bfs(st);
32          int d2 = bfs(d1);
33          vector<int> path;
34          for (int u = d2; u != -1; u = pai[u]) { path.push_back(u); }
35          int len = path.size();
36          if (len % 2 == 1) {
37              centros.push_back(path[len / 2]);
38          } else {
```

```
39                 centros.push_back(path[len / 2]);
40                 centros.push_back(path[len / 2 - 1]);
41             }
42         }
43 };
```

2.12 Shortest Path Fast Algorithm (SPFA)

Encontra o caminho mais curto entre um vértice e todos os outros vértices de um grafo.

Detecta ciclos negativos.

Complexidade de tempo: O(|V| * |E|)

```
1  const int MAX = 1e4 + 4;
2  const ll INF = 1e18 + 18;
3
4  vector<ii> adj[MAX];
5  ll dist[MAX];
6
7  void spfa(int s, int n) {
8      fill(dist, dist + n, INF);
9      vector<int> cnt(n, 0);
10     vector<bool> inq(n, false);
11     queue<int> fila;
12     fila.push(s);
13     inq[s] = true;
14     dist[s] = 0;
15     while (!fila.empty()) {
16         int u = fila.front();
17         fila.pop();
18         inq[u] = false;
19         for (auto [w, v] : adj[u]) {
20             ll newd = (dist[u] == -INF ? -INF : max(w + dist[u], -INF));
21             if (newd < dist[v]) {
22                 dist[v] = newd;
23                 if (!inq[v]) {
24                     fila.push(v);
25                     inq[v] = true;
26                     cnt[v]++;
27                     if (cnt[v] > n) { // negative cycle
28                         dist[v] = -INF;
29                     }
30                 }
31             }
32         }
33     }
34 }
```

- [**2-SAT**](2-SAT) Resolve o problema do 2-SAT. - [**Bridge**](Bridge) Algoritimo que acha pontes. - [**Binary Lifting**](Binary%20Lifting) Busca por antepassados em uma árvore. - [**Dijkstra**](Dijkstra) Caminho mínimo de 1 para todos. - [**SPFA**](SPFA) Caminho mínimo de 1 para todos. Detecta ciclos negativos. - [**Fluxo**](Fluxo) Problemas sobre fluxo - [**Graph Center**](Graph%20Center) Encontra o centro do grafo. - [**HLD (Heavy Light Decomposition)**](HLD) Quebra uma árvore em cadeias e facilita consultas. - [**LCA (Lowest Common Ancestor)**](LCA) Encontra o menor ancetral comum de dois vétices em uma árvore. - [**Matching**](Matching) Encontra o conjunto de pares com o min/max custo em um grafo bipartido. - [**Kruskal**](Kruskal) Monta a Minimum Spanning Tree. - [**Stoer-Wagner minimum cut**](Stoer–Wagner%20minimum%20cut) Resolver o problema de corte mínimo em gráficos não direcionados com pesos não negativos

3 String

3.1 Aho-Corasick

Constrói uma estrutura de dados semelhante a um trie com links adicionais e, em seguida, constrói uma máquina de estados finitos (autômato). Útil para pattern matching de um set de strings em um texto.

Complexidade de tempo: O(|S|+|T|), onde |S| é o somatório do tamanho das strings e |T| é o tamanho do texto

```
1  const int K = 26;
2
3  struct Vertex {
4      int next[K], p = -1, link = -1, exi = -1, go[K], cont = 0;
5      bool term = false;
6      vector<int> idxs;
7      char pch;
8      Vertex(int p = -1, char ch = '$') : p(p), pch(ch) {
9          fill(begin(next), end(next), -1);
10         fill(begin(go), end(go), -1);
11     }
12 };
13 vector<Vertex> aho(1);
14 void add_string(const string &s, int idx) {
15     int v = 0;
16     for (char ch : s) {
17         int c = ch - 'a';
18         if (aho[v].next[c] == -1) {
19             aho[v].next[c] = aho.size();
20             aho.emplace_back(v, ch);
21         }
22         v = aho[v].next[c];
23     }
24     aho[v].term = true;
25     aho[v].idxs.push_back(idx);
26 }
27 int go(int u, char ch);
28 int get_link(int u) {
29     if (aho[u].link == -1) {
30         if (u == 0 || aho[u].p == 0) {
31             aho[u].link = 0;
32         } else {
33             aho[u].link = go(get_link(aho[u].p), aho[u].pch);
34         }
35     }
36     return aho[u].link;
37 }
38 int go(int u, char ch) {
39     int c = ch - 'a';
40     if (aho[u].go[c] == -1) {
41         if (aho[u].next[c] != -1) {
42             aho[u].go[c] = aho[u].next[c];
43         } else {
44             aho[u].go[c] = u == 0 ? 0 : go(get_link(u), ch);
45         }
46     }
47     return aho[u].go[c];
48 }
49 int exi(int u) {
50     if (aho[u].exi != -1) { return aho[u].exi; }
```

```
51      int v = get_link(u);
52      return aho[u].exi = (v == 0 || aho[v].term ? v : exi(v));
53  }
54  void process(const string &s) {
55      int st = 0;
56      for (char c : s) {
57          st = go(st, c);
58          for (int aux = st; aux; aux = exi(aux)) { aho[aux].cont++; }
59      }
60      for (int st = 1; st < aho_sz; st++) {
61          if (!aho[st].term) { continue; }
62          for (int i : aho[st].idxs) {
63              // Do something here
64              // idx i ocurs + aho[st].cont times
65              h[i] += aho[st].cont;
66          }
67      }
68  }
```

3.2 Patricia Tree ou Patricia Trie

Implementação PB-DS, extremamente curta e confusa:

- Criar: patricia_tree pat; - Inserir: pat.insert("sei la"); - Remover: pat.erase("sei la"); - Verificar existência: pat.find("sei la") != pat.end(); - Pegar palavras que começam com um prefixo: **auto** match = pat.prefix_range("sei"); - Percorrer *match* : **for**(**auto** it = match.first; it != match.second; ++it); - Pegar menor elemento lexicográfico *maior ou igual* ao prefixo: *pat.lower_bound("sei"); - Pegar menor elemento lexicográfico *maior* ao prefixo: *pat.upper_bound("sei");

TODAS AS OPERAÇÕES EM O(|S|) **NÃO ACEITA ELEMENTOS REPETIDOS**

```
1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/trie_policy.hpp>
3
4  using namespace __gnu_pbds;
5  typedef trie<string, null_type, trie_string_access_traits<>, pat_trie_tag,
       trie_prefix_search_node_update>
6      patricia_tree;
```

3.3 Prefix Function

Para cada prefixo k de uma dada string s, calcula o maior prefixo que tambem é sufixo de k.

Seja n o tamanho do texto e m o tamanho do padrão.

[KMP](KMP.cpp)

String matching em O(n + m).

[Autômato de KMP](aut_kmp.cpp)

String matching em O(n) com O(m) de pré-processamento.

[Prefix Count](prefix_count.cpp)

Dada uma string s, calcula quantas vezes cada prefixo de s aparece em s com complexidade de tempo de O(n).

```
1  vector<int> pi(string &s) {
2      vector<int> p(s.size());
```

```
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
        if (s[i] == s[j]) { j++; }
        p[i] = j;
    }
    return p;
}
```

```
vector<int> pi(string &s) {
    vector<int> p(s.size());
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
        if (s[i] == s[j]) { j++; }
        p[i] = j;
    }
    return p;
}

vector<int> kmp(string &s, string t) {
    t += '$';
    vector<int> p = pi(t), match;
    for (int i = 0, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != t[j]) { j = p[j - 1]; }
        if (s[i] == t[j]) { j++; }
        if (j == t.size() - 1) { match.push_back(i - j + 1); }
    }
    return match;
}
```

```
vector<int> pi(string s) {
    vector<int> p(s.size());
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
        if (s[i] == s[j]) { j++; }
        p[i] = j;
    }
    return p;
}

vector<int> prefixCount(string s) {
    vector<int> p = pi(s + '#');
    int n = s.size();
    vector<int> cnt(n + 1, 0);
    for (int i = 0; i < n; i++) { cnt[p[i]]++; }
    for (int i = n - 1; i > 0; i--) { cnt[p[i - 1]] += cnt[i]; }
    for (int i = 0; i <= n; i++) { cnt[i]++; }
    return cnt;
}
```

```
struct AutKMP {
    vector<vector<int>> nxt;

    vector<int> pi(string &s) {
        vector<int> p(s.size());
        for (int i = 1, j = 0; i < s.size(); i++) {
            while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
            if (s[i] == s[j]) { j++; }
            p[i] = j;
        }
        return p;
    }
```

```
13
14      void setString(string s) {
15          s += '#';
16          nxt.assign(s.size(), vector<int>(26));
17          vector<int> p = pi(s);
18          for (int c = 0; c < 26; c++) { nxt[0][c] = ('a' + c == s[0]); }
19          for (int i = 1; i < s.size(); i++) {
20              for (int c = 0; c < 26; c++) { nxt[i][c] = ('a' + c == s[i]) ? i + 1 :
                    nxt[p[i - 1]][c]; }
21          }
22      }
23
24      vector<int> kmp(string &s, string &t) {
25          vector<int> match;
26          for (int i = 0, j = 0; i < s.size(); i++) {
27              j = nxt[j][s[i] - 'a'];
28              if (j == t.size()) { match.push_back(i - j + 1); }
29          }
30          return match;
31      }
32  } aut;
```

3.4 Hashing

Hashing para testar igualdade de duas strings A função ***range(i, j)*** retorna o hash da substring nesse range. Pode ser necessário usar pares de hash para evitar colisões.

* Complexidade de tempo (Construção): O(N) * Complexidade de tempo (Consulta de range): O(1)

```
1   struct hashing {
2       const long long LIM = 1000006;
3       long long p, m;
4       vector<long long> pw, hsh;
5       hashing(long long _p, long long _m) : p(_p), m(_m) {
6           pw.resize(LIM);
7           hsh.resize(LIM);
8           pw[0] = 1;
9           for (int i = 1; i < LIM; i++) { pw[i] = (pw[i - 1] * p) % m; }
10      }
11      void set_string(string &s) {
12          hsh[0] = s[0];
13          for (int i = 1; i < s.size(); i++) { hsh[i] = (hsh[i - 1] * p + s[i]) % m;
                }
14      }
15      long long range(int esq, int dir) {
16          long long ans = hsh[dir];
17          if (esq > 0) { ans = (ans - (hsh[esq - 1] * pw[dir - esq + 1] % m) + m) %
                m; }
18          return ans;
19      }
20  };
```

3.5 Trie

Estrutura que guarda informações indexadas por palavra. Útil encontrar todos os prefixos inseridos anteriormente de uma palavra específica.

* Complexidade de tempo (Update): O(|S|) * Complexidade de tempo (Consulta de palavra): O(|S|)

```
struct trie {
    map<char, int> trie[100005];
    int value[100005];
    int n_nodes = 0;
    void insert(string &s, int v) {
        int id = 0;
        for (char c : s) {
            if (!trie[id].count(c)) { trie[id][c] = ++n_nodes; }
            id = trie[id][c];
        }
        value[id] = v;
    }
    int get_value(string &s) {
        int id = 0;
        for (char c : s) {
            if (!trie[id].count(c)) { return -1; }
            id = trie[id][c];
        }
        return value[id];
    }
};
```

3.6 Algoritmo de Manacher

Dada uma string s de tamanho n, encontra todos os pares (i,j) tal que a substring s

$$i...j$$

seja um palindromo.

* Complexidade de tempo: O(N)

```
struct manacher {
    long long n, count;
    vector<int> d1, d2;
    long long solve(string &s) {
        n = s.size(), count = 0;
        solve_odd(s);
        solve_even(s);
        return count;
    }
    void solve_odd(string &s) {
        d1.resize(n);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
            while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) { k++; }
            count += d1[i] = k--;
            if (i + k > r) {
                l = i - k;
                r = i + k;
            }
        }
    }
    void solve_even(string &s) {
        d2.resize(n);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
            while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) { k++;
                }
            count += d2[i] = k--;
```

```
28                 if (i + k > r) {
29                     l = i - k - 1;
30                     r = i + k;
31                 }
32             }
33         }
34 } mana;
```

3.7 Lyndon Factorization

Strings em decomposição única em subcadeias que são ordenadas lexicograficamente e não podem ser mais reduzidas.

[Duval](duval.cpp)

Gera a Lyndon Factorization de uma string

* Complexidade de tempo: O(N)

[Min Cyclic Shift](min_cyclic_shift.cpp)

Gera a menor rotação circular da string original que pode ser obtida por meio de deslocamentos cíclicos dos caracteres.

* Complexidade de tempo: O(N)

```
1  string min_cyclic_shift(string s) {
2      s += s;
3      int n = s.size();
4      int i = 0, ans = 0;
5      while (i < n / 2) {
6          ans = i;
7          int j = i + 1, k = i;
8          while (j < n && s[k] <= s[j]) {
9              if (s[k] < s[j]) {
10                 k = i;
11             } else {
12                 k++;
13             }
14             j++;
15         }
16         while (i <= k) { i += j - k; }
17     }
18     return s.substr(ans, n / 2);
19 }
```

```
1  vector<string> duval(string const &s) {
2      int n = s.size();
3      int i = 0;
4      vector<string> factorization;
5      while (i < n) {
6          int j = i + 1, k = i;
7          while (j < n && s[k] <= s[j]) {
8              if (s[k] < s[j]) {
9                  k = i;
10             } else {
11                 k++;
12             }
13             j++;
14         }
15         while (i <= k) {
16             factorization.push_back(s.substr(i, j - k));
17             i += j - k;
18         }
19     }
20     return factorization;
21 }
```

3.8 Suffix Array

Estrutura que conterá inteiros que representam os índices iniciais de todos os sufixos ordenados de uma determinada string.

Tambem Constroi a tabela LCP(Longest common prefix).

* Complexidade de tempo (Pré-Processamento): O(|S|*log(|S|)) * Complexidade de tempo (Contar ocorrencias de S em T): O(|S|*log(|T|))

```
1  pair<int, int> busca(string &t, int i, pair<int, int> &range) {
2      int esq = range.first, dir = range.second, L = -1, R = -1;
3      while (esq <= dir) {
4          int mid = (esq + dir) / 2;
5          if (s[sa[mid] + i] == t[i]) { L = mid; }
6          if (s[sa[mid] + i] < t[i]) {
7              esq = mid + 1;
8          } else {
9              dir = mid - 1;
10         }
11     }
12     esq = range.first, dir = range.second;
13     while (esq <= dir) {
14         int mid = (esq + dir) / 2;
15         if (s[sa[mid] + i] == t[i]) { R = mid; }
16         if (s[sa[mid] + i] <= t[i]) {
17             esq = mid + 1;
18         } else {
19             dir = mid - 1;
20         }
21     }
22     return {L, R};
23 }
24 // count ocurences of s on t
25 int busca_string(string &t) {
26     pair<int, int> range = {0, n - 1};
27     for (int i = 0; i < t.size(); i++) {
28         range = busca(t, i, range);
29         if (range.first == -1) { return 0; }
30     }
31     return range.second - range.first + 1;
32 }
```

```
1  const int MAX_N = 5e5 + 5;
2
3  struct suffix_array {
4      string s;
5      int n, sum, r, ra[MAX_N], sa[MAX_N], auxra[MAX_N], auxsa[MAX_N], c[MAX_N],
           lcp[MAX_N];
6      void counting_sort(int k) {
7          memset(c, 0, sizeof(c));
8          for (int i = 0; i < n; i++) { c[(i + k < n) ? ra[i + k] : 0]++; }
9          for (int i = sum = 0; i < max(256, n); i++) { sum += c[i], c[i] = sum -
               c[i]; }
10         for (int i = 0; i < n; i++) { auxsa[c[sa[i] + k < n ? ra[sa[i] + k] :
               0]++] = sa[i]; }
11         for (int i = 0; i < n; i++) { sa[i] = auxsa[i]; }
12     }
13     void build_sa() {
14         for (int k = 1; k < n; k <<= 1) {
15             counting_sort(k);
```

```
16              counting_sort(0);
17              auxra[sa[0]] = r = 0;
18              for (int i = 1; i < n; i++) {
19                  auxra[sa[i]] = (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] ==
                        ra[sa[i - 1] + k]) ? r : ++r;
20              }
21              for (int i = 0; i < n; i++) { ra[i] = auxra[i]; }
22              if (ra[sa[n - 1]] == n - 1) { break; }
23          }
24      }
25      void build_lcp() {
26          for (int i = 0, k = 0; i < n - 1; i++) {
27              int j = sa[ra[i] - 1];
28              while (s[i + k] == s[j + k]) { k++; }
29              lcp[ra[i]] = k;
30              if (k) { k--; }
31          }
32      }
33      void set_string(string _s) {
34          s = _s + '$';
35          n = s.size();
36          for (int i = 0; i < n; i++) { ra[i] = s[i], sa[i] = i; }
37          build_sa();
38          build_lcp();
39          // for (int i = 0; i < n; i++) printf("%2d: %s\n", sa[i], s.c_str() +
                sa[i]);
40      }
41      int operator[](int i) { return sa[i]; }
42  } sa;
```

- **Aho-Corasick** Constrói uma estrutura de dados semelhante a um trie com links adicionais e, em seguida, constrói uma máquina de estados finitos (autômato). Útil para pattern matching de um set de strings em um texto.

- **Manacher** Dado string s com tamanho n. Encontre todos os pares (i, j) tal que a substring s [i...j] seja um palíndromo.

- **Hashing** Hasha Strings e faz comparações em O(1).

- **[Patricia Tree](Patricia%20Tree)** Trie Rápida. (Não guarda valores para cada string)

- **[Prefix Function](Prefix%20Function)** Algoritmos usando a Prefix Function, como KMP e PrefixCount.

- **[Suffix Array](Suffix%20Array)** Trabalha com todos os sufixos sem precisar criá-los.

- **Trie** Estrutura para trabalhar com Strings e encontrar prefixos. Pode servir para converter uma String em outro valor em O(|S|)

- **[Lyndon Factorization](Lyndon%20Factorization)** Strings em decomposição única em subcadeias que são ordenadas lexicograficamente e não podem ser mais reduzidas.

4 Paradigmas

4.1 Mo

Resolve Queries Complicadas Offline de forma rápida. É preciso manter uma estrutura que adicione e remova elementos nas extremeidades de um range (tipo janela).

- Complexidade de tempo (Query offline): O(N * sqrt(N))

[Mo com Update](mo_update.cpp)

Resolve Queries Complicadas Offline de forma rápida. Permite que existam **UPDATES PONTUAIS!** É preciso manter uma estrutura que adicione e remova elementos nas extremidades de um range (tipo janela).

- Complexidade de tempo: $O(Q * N^(2/3))$

```
1  typedef pair<int, int> ii;
2  int block_sz; // Better if 'const';
3
4  namespace mo {
5      struct query {
6          int l, r, idx;
7          bool operator<(query q) const {
8              int _l = l / block_sz;
9              int _ql = q.l / block_sz;
10             return ii(_l, (_l & 1 ? -r : r)) < ii(_ql, (_ql & 1 ? -q.r : q.r));
11         }
12     };
13     vector<query> queries;
14
15     void build(int n) {
16         block_sz = (int)sqrt(n);
17         // TODO: initialize data structure
18     }
19     inline void add_query(int l, int r) { queries.push_back({l, r,
           (int)queries.size()}); }
20     inline void remove(int idx) {
21         // TODO: remove value at idx from data structure
22     }
23     inline void add(int idx) {
24         // TODO: add value at idx from data structure
25     }
26     inline int get_answer() {
27         // TODO: extract the current answer of the data structure
28         return 0;
29     }
30
31     vector<int> run() {
32         vector<int> answers(queries.size());
33         sort(queries.begin(), queries.end());
34         int L = 0;
35         int R = -1;
36         for (query q : queries) {
37             while (L > q.l) { add(--L); }
38             while (R < q.r) { add(++R); }
39             while (L < q.l) { remove(L++); }
40             while (R > q.r) { remove(R--); }
41             answers[q.idx] = get_answer();
42         }
43         return answers;
```

```
    }

};
```

```
typedef pair<int, int> ii;
typedef tuple<int, int, int> iii;
int block_sz; // Better if 'const';
vector<int> vec;
namespace mo {
    struct query {
        int l, r, t, idx;
        bool operator<(query q) const {
            int _l = l / block_sz;
            int _r = r / block_sz;
            int _ql = q.l / block_sz;
            int _qr = q.r / block_sz;
            return iii(_l, (_l & 1 ? -_r : _r), (_r & 1 ? t : -t)) <
                   iii(_ql, (_ql & 1 ? -_qr : _qr), (_qr & 1 ? q.t : -q.t));
        }
    };
    vector<query> queries;
    vector<ii> updates;

    void build(int n) {
        block_sz = pow(1.4142 * n, 2.0 / 3);
        // TODO: initialize data structure
    }
    inline void add_query(int l, int r) { queries.push_back({l, r,
       (int)updates.size(), (int)queries.size()}); }
    inline void add_update(int x, int v) { updates.push_back({x, v}); }
    inline void remove(int idx) {
        // TODO: remove value at idx from data structure
    }
    inline void add(int idx) {
        // TODO: add value at idx from data structure
    }
    inline void update(int l, int r, int t) {
        auto &[x, v] = updates[t];
        if (l <= x && x <= r) { remove(x); }
        swap(vec[x], v);
        if (l <= x && x <= r) { add(x); }
    }
    inline int get_answer() {
        // TODO: extract the current answer from the data structure
        return 0;
    }

    vector<int> run() {
        vector<int> answers(queries.size());
        sort(queries.begin(), queries.end());
        int L = 0;
        int R = -1;
        int T = 0;
        for (query q : queries) {
            while (T < q.t) { update(L, R, T++); }
            while (T > q.t) { update(L, R, --T); }
            while (L > q.l) { add(--L); }
            while (R < q.r) { add(++R); }
            while (L < q.l) { remove(L++); }
            while (R > q.r) { remove(R--); }
            answers[q.idx] = get_answer();
```

```
57            }
58            return answers;
59        }
60    };
```

4.2 Exponenciação de Matriz

Otimização para DP de prefixo quando o valor atual está em função dos últimos K valores já calculados.
* Complexidade de tempo: $O(log(n) * k^3)$

É preciso mapear a DP para uma exponenciação de matriz.

—

Uso Comum

DP:

$$dp[n] = \sum\_i = 1^k c[i] \cdot dp[n-i]$$

Mapeamento:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & ... & 0 \\ 0 & 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 0 & 1 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ c[k] & c[k-1] & c[k-2] & ... & c[1] & 0 \end{pmatrix}^n \times \begin{pmatrix} dp[0] \\ dp[1] \\ dp[2] \\ ... \\ dp[k-1] \end{pmatrix}$$

— ### Variação que dependa de **constantes** e do **índice**

Exemplo de DP:

$$dp[i] = dp[i-1] + 2 \cdot i^2 + 3 \cdot i + 5$$

Nesses casos é preciso fazer uma linha para manter cada constante e potência do índice.

Mapeamento:

$$\begin{pmatrix} 1 & 5 & 3 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}^n \times \begin{pmatrix} dp[0] \\ 1 \\ 1 \\ 1 \end{pmatrix} \begin{matrix} mantm\ dp[i] \\ mantm\ 1 \\ mantm\ i \\ mantm\ i \end{matrix}$$

Variação Multiplicativa

Exemplo de DP:

$$dp[n] = c \times \prod\_i = 1^k dp[n-i]$$

Nesses casos é preciso trabalhar com o logaritmo e temos o caso padrão:

$$\log(dp[n]) = log(c) + \sum\_i = 1^k log(dp[n-i])$$

Se a resposta precisar ser inteira, deve-se fatorar a constante e os valores inicias e então fazer uma exponenciação para cada fator primo. Depois é só juntar a resposta no final.

```cpp
ll dp[100];
mat T;

#define MOD 1000000007

mat mult(mat a, mat b) {
    mat res(a.size(), vi(b[0].size()));
    for (int i = 0; i < a.size(); i++) {
        for (int j = 0; j < b[0].size(); j++) {
            for (int k = 0; k < b.size(); k++) {
                res[i][j] += a[i][k] * b[k][j] % MOD;
                res[i][j] %= MOD;
            }
        }
    }
    return res;
}

mat exp_mod(mat b, ll exp) {
    mat res(b.size(), vi(b.size()));
    for (int i = 0; i < b.size(); i++) { res[i][i] = 1; }

    while (exp) {
        if (exp & 1) { res = mult(res, b); }
        b = mult(b, b);
        exp /= 2;
    }
    return res;
}

// MUDA MUITO DE ACORDO COM O PROBLEMA
// LEIA COMO FAZER O MAPEAMENTO NO README
ll solve(ll exp, ll dim) {
    if (exp < dim) { return dp[exp]; }

    T.assign(dim, vi(dim));
    // TO DO: Preencher a Matriz que vai ser exponenciada
    // T[0][1] = 1;
    // T[1][0] = 1;
    // T[1][1] = 1;

    mat prod = exp_mod(T, exp);

    mat vec;
    vec.assign(dim, vi(1));
    for (int i = 0; i < dim; i++) {
        vec[i][0] = dp[i]; // Valores iniciais
    }

    mat ans = mult(prod, vec);
    return ans[0][0];
}
```

4.3 Busca Binária Paralela

Faz a busca binária para múltiplas consultas quando a busca binária é muito pesada.

- Complexidade de tempo: O((N+Q)log(N) * O(F)), onde N é o tamanho do espaço de busca, Q é o número de consultas e O(F), o custo de avaliação da função.

```cpp

namespace parallel_binary_search {
    typedef tuple<int, int, long long, long long> query; //{value, id, l, r}
    vector<query> queries[1123456];                      // pode ser um mapa se for muito esparso
    long long ans[1123456];                              // definir pro tamanho das queries
    long long l, r, mid;
    int id = 0;
    void set_lim_search(long long n) {
        l = 0;
        r = n;
        mid = (l + r) / 2;
    }

    void add_query(long long v) { queries[mid].push_back({v, id++, l, r}); }

    void advance_search(long long v) {
        // advance search
    }

    bool satisfies(long long mid, int v, long long l, long long r) {
        // implement the evaluation
    }

    bool get_ans() {
        // implement the get ans
    }

    void parallel_binary_search(long long l, long long r) {

        bool go = 1;
        while (go) {
            go = 0;
            int i = 0; // outra logica se for usar um mapa
            for (auto &vec : queries) {
                advance_search(i++);
                for (auto q : vec) {
                    auto [v, id, l, r] = q;
                    if (l > r) { continue; }
                    go = 1;
                    // return while satisfies
                    if (satisfies(i, v, l, r)) {
                        ans[i] = get_ans();
                        long long mid = (i + l) / 2;
                        queries[mid] = query(v, id, l, i - 1);
                    } else {
                        long long mid = (i + r) / 2;
                        queries[mid] = query(v, id, i + 1, r);
                    }
                }
                vec.clear();
            }
        }
    }

} // namespace name
```

4.4 Divide and Conquer

Otimização para DP de prefixo quando se pretende separar o vetor em K subgrupos. É preciso fazer a função query(i, j) que computa o custo do subgrupo

$$i, j$$

. * Complexidade de tempo: O(n * k * log(n) * O(query))

[Divide and Conquer com Query on demand](dc_query_ondemand.cpp)

Usado para evitar queries pesadas ou o custo de pré-processamento. É preciso fazer as funções da estrutura **janela**, eles adicionam e removem itens um a um como uma janela flutuante.

* Complexidade de tempo: O(n * k * log(n) * O(update da janela))

```cpp
namespace DC {
    vi dp_before, dp_cur;
    void compute(int l, int r, int optl, int optr) {
        if (l > r) { return; }
        int mid = (l + r) >> 1;
        pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
        for (int i = optl; i <= min(mid, optr); i++) {
            best = max(best, {(i ? dp_before[i - 1] : 0) + query(i, mid), i}); // min() se quiser minimizar
        }
        dp_cur[mid] = best.first;
        int opt = best.second;
        compute(l, mid - 1, optl, opt);
        compute(mid + 1, r, opt, optr);
    }

    ll solve(int n, int k) {
        dp_before.assign(n + 5, 0);
        dp_cur.assign(n + 5, 0);
        for (int i = 0; i < n; i++) { dp_before[i] = query(0, i); }
        for (int i = 1; i < k; i++) {
            compute(0, n - 1, 0, n - 1);
            dp_before = dp_cur;
        }
        return dp_before[n - 1];
    }
};
```

```cpp
namespace DC {
    struct range { // eh preciso definir a forma de calcular o range
        vi freq;
        ll sum = 0;
        int l = 0, r = -1;
        void back_l(int v) { // Mover o 'l' do range para a esquerda
            sum += freq[v];
            freq[v]++;
            l--;
        }
        void advance_r(int v) { // Mover o 'r' do range para a direita
            sum += freq[v];
            freq[v]++;
            r++;
        }
        void advance_l(int v) { // Mover o 'l' do range para a direita
            freq[v]--;
            sum -= freq[v];
```

```
19              l++;
20          }
21          void back_r(int v) { // Mover o 'r' do range para a esquerda
22              freq[v]--;
23              sum -= freq[v];
24              r--;
25          }
26          void clear(int n) { // Limpar range
27              l = 0;
28              r = -1;
29              sum = 0;
30              freq.assign(n + 5, 0);
31          }
32      } s;
33
34      vi dp_before, dp_cur;
35      void compute(int l, int r, int optl, int optr) {
36          if (l > r) { return; }
37          int mid = (l + r) >> 1;
38          pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
39
40          while (s.l < optl) { s.advance_l(v[s.l]); }
41          while (s.l > optl) { s.back_l(v[s.l - 1]); }
42          while (s.r < mid) { s.advance_r(v[s.r + 1]); }
43          while (s.r > mid) { s.back_r(v[s.r]); }
44
45          vi removed;
46          for (int i = optl; i <= min(mid, optr); i++) {
47              best = min(best, {(i ? dp_before[i - 1] : 0) + s.sum, i}); // min() se
                    quiser minimizar
48              removed.push_back(v[s.l]);
49              s.advance_l(v[s.l]);
50          }
51          for (int rem : removed) { s.back_l(v[s.l - 1]); }
52
53          dp_cur[mid] = best.first;
54          int opt = best.second;
55          compute(l, mid - 1, optl, opt);
56          compute(mid + 1, r, opt, optr);
57      }
58
59      ll solve(int n, int k) {
60          dp_before.assign(n, 0);
61          dp_cur.assign(n, 0);
62          s.clear(n);
63          for (int i = 0; i < n; i++) {
64              s.advance_r(v[i]);
65              dp_before[i] = s.sum;
66          }
67          for (int i = 1; i < k; i++) {
68              s.clear(n);
69              compute(0, n - 1, 0, n - 1);
70              dp_before = dp_cur;
71          }
72          return dp_before[n - 1];
73      }
74  };
```

4.5 Busca Ternária

Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (e.g. parábolas).

- Complexidade de tempo: O(log(N) * O(eval)). Onde N é o tamanho do espaço de busca e O(eval) o custo de avaliação da função.

[Busca Ternária em Espaço Discreto](busca_ternaria_discreta.cpp)

Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (e.g. parábolas). Versão para espaços discretos.

- Complexidade de tempo: O(log(N) * O(eval)). Onde N é o tamanho do espaço de busca e O(eval) o custo de avaliação da função.

```
1
2  double eval(double mid) {
3      // implement the evaluation
4  }
5
6  double ternary_search(double l, double
       r) {
7      int k = 100;
8      while (k--) {
9          double step = (l + r) / 3;
10         double mid_1 = l + step;
11         double mid_2 = r - step;
12
13         // minimizing. To maximize use
               >= to compare
14         if (eval(mid_1) <= eval(mid_2))
               {
15             r = mid_2;
16         } else {
17             l = mid_1;
18         }
19     }
20     return l;
21 }
```

```
1
2  long long eval(long long mid) {
3      // implement the evaluation
4  }
5
6  long long discrete_ternary_search(long long l, long long r) {
7      long long ans = -1;
8      r--; // to not space r
9      while (l <= r) {
10         long long mid = (l + r) / 2;
11
12         // minimizing. To maximize use >= to compare
13         if (eval(mid) <= eval(mid + 1)) {
14             ans = mid;
15             r = mid - 1;
16         } else {
17             l = mid + 1;
18         }
19     }
20     return ans;
21 }
```

4.6 DP de Permutação

Otimização do problema do Caixeiro Viajante

* Complexidade de tempo: $O(n^2 * 2^n)$

Para rodar a função basta setar a matriz de adjacência 'dist' e chamar solve(0,0,n).

```cpp
const int lim = 17;          // setar para o maximo de itens
long double dist[lim][lim]; // eh preciso dar as distancias de n para n
long double dp[lim][1 << lim];

int limMask = (1 << lim) - 1; // 2**(maximo de itens) - 1
long double solve(int atual, int mask, int n) {
    if (dp[atual][mask] != 0) { return dp[atual][mask]; }
    if (mask == (1 << n) - 1) {
        return dp[atual][mask] = 0; // o que fazer quando chega no final
    }

    long double res = 1e13; // pode ser maior se precisar
    for (int i = 0; i < n; i++) {
        if (!(mask & (1 << i))) {
            long double aux = solve(i, mask | (1 << i), n);
            if (mask) { aux += dist[atual][i]; }
            res = min(res, aux);
        }
    }
    return dp[atual][mask] = res;
}
```

4.7 Convex Hull Trick

Otimização de DP onde se mantém as retas que formam um Convex Hull em uma estrutura que permite consultar qual o melhor valor para um determinado x.

Só funciona quando as retas são monotônicas. Caso não forem, usar LiChao Tree para guardar as retas

Complexidade de tempo:

- Inserir reta: O(1) amortizado - Consultar x: O(log(N)) - Consultar x quando x tem crescimento monotônico: O(1)

```cpp
const ll INF = 1e18 + 18;
bool op(ll a, ll b) {
    return a >= b; // either >= or <=
}
struct line {
    ll a, b;
    ll get(ll x) { return a * x + b; }
    ll intersect(line l) {
        return (l.b - b + a - l.a) / (a - l.a); // rounds up for integer only
    }
};
deque<pair<line, ll>> fila;
void add_line(ll a, ll b) {
    line nova = {a, b};
    if (!fila.empty() && fila.back().first.a == a && fila.back().first.b == b) {
        return; }
    while (!fila.empty() && op(fila.back().second,
        nova.intersect(fila.back().first))) { fila.pop_back(); }
    ll x = fila.empty() ? -INF : nova.intersect(fila.back().first);
    fila.emplace_back(nova, x);
}
ll get_binary_search(ll x) {
    int esq = 0, dir = fila.size() - 1, r = -1;
    while (esq <= dir) {
        int mid = (esq + dir) / 2;
```

```
24            if (op(x, fila[mid].second)) {
25                esq = mid + 1;
26                r = mid;
27            } else {
28                dir = mid - 1;
29            }
30        }
31        return fila[r].first.get(x);
32    }
33    // O(1), use only when QUERIES are monotonic!
34    ll get(ll x) {
35        while (fila.size() >= 2 && op(x, fila[1].second)) { fila.pop_front(); }
36        return fila.front().first.get(x);
37    }
```

4.8 All Submask

Percorre todas as submáscaras de uma máscara de tamanho N

* Complexidade de tempo: $O(3^N)$

```
1  int mask;                                          |     1) & mask) { }
2  for (int sub = mask; sub; sub = (sub -             |
```

+ **[All Submasks](All%20Submasks)** Percorre todas as submáscaras de uma máscara de bits. + **[Busca Binária Paralela](Busca%20Binaria%20Paralela)** Faz a busca binária para múltiplas consultas quando a busca binaria é muito pesada. + **[Busca Ternária](Busca%20Ternaria)** Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (e.g. parábolas). + **[Convex Hull Trick](Convex%20Hull%20Trick)** Otimização para DP utilizando retas monotônicas que formam um Convex Hull. + **[Divide and Conquer](Divide%20and%20Conquer)** Otimização para DP de prefixo quando se pretende separar o vetor em K subgrupos. + **[DP de Permutação](DP%20de%20Permutacao)** Simula todas as permutações de um grupo. Resolve o caxeiro viajante 'rápido'. + **[Exponeciação de Matriz](Exponencia%C3%A7%C3%A3o%20de%20Matriz)** Otimização para DP de prefixo quando o valor atual está em função dos últimos K valores já calculados. + **[Mo](Mo/)** Resolve operações em range Offline. **COM E SEM ATUALIZAÇÂO**

5 Matemática

5.1 Soma do floor(n / i)

Computa o somatório de n dividido de 1 a n (divisão arredondado pra baixo).

- Complexidade de tempo: O(sqrt(n)).

```
const int MOD = 1e9 + 7;

long long sumoffloor(long long n) {
    long long answer = 0, i;
    for (i = 1; i * i <= n; i++) {
        answer += n / i;
        answer %= MOD;
    }
    i--;
    for (int j = 1; n / (j + 1) >= i; j++) {
        answer += (((n / j - n / (j + 1)) % MOD) * j) % MOD;
        answer %= MOD;
    }
    return answer;
}
```

5.2 Primos

[Crivo de Eratóstenes](sieve.cpp) Computa a primalidade de todos os números até N, quase tão rápido quanto o crivo linear.

- Complexidade de tempo: O(N * log(log(N)))

Demora 1 segundo para LIM igual a $3 * 10^7$.

[Miller-Rabin](miller_rabin.cpp) Teste de primalidade garantido para números menores do que 2^64.

- Complexidade de tempo: O(log(N))

[Teste Ingênuo](naive_is_prime.cpp) Computa a primalidade de um número N.

- Complexidade de tempo: $O(N^(1/2))$

```
vector<bool> sieve(int n) {
    vector<bool> is_prime(n + 5, true);
    is_prime[0] = false;
    is_prime[1] = false;
    long long sq = sqrt(n + 5);
    for (long long i = 2; i <= sq; i++) {
        if (is_prime[i]) {
            for (long long j = i * i; j < n; j += i) { is_prime[j] = false; }
        }
    }
    return is_prime;
}
```

```
bool is_prime(int n) {
    for (long long d = 2; d * d <= n; d++) {
        if (n % d == 0) { return false; }
        }
    }
    return true;
}
```

```cpp
long long power(long long base, long long e, long long mod) {
    long long result = 1;
    base %= mod;
    while (e) {
        if (e & 1) { result = (__int128)result * base % mod; }
        base = (__int128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool is_composite(long long n, long long a, long long d, int s) {
    long long x = power(a, d, n);
    if (x == 1 || x == n - 1) { return false; }
    for (int r = 1; r < s; r++) {
        x = (__int128)x * x % n;
        if (x == n - 1) { return false; }
    }
    return true;
}

bool miller_rabin(long long n) {
    if (n < 2) { return false; }
    int r = 0;
    long long d = n - 1;
    while ((d & 1) == 0) { d >>= 1, ++r; }
    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a) { return true; }
        if (is_composite(n, a, d, r)) { return false; }
    }
    return true;
}
```

5.3 Numeric Theoric Transformation

Computa multiplicação de polinômino; **Somente para inteiros**.

- Complexidade de tempo: O(N * log(N))

Constantes finais devem ser menor do que 10^9. Para constantes entre 10^9 e 10^{18} é necessário codar também [big_convolution](big_convolution.cpp).

```cpp
typedef long long ll;
typedef vector<ll> poly;

ll mod[3] = {998244353LL, 1004535809LL, 1092616193LL};
ll root[3] = {102292LL, 12289LL, 23747LL};
ll root_1[3] = {116744195LL, 313564925LL, 642907570LL};
ll root_pw[3] = {1LL << 23, 1LL << 21, 1LL << 21};

ll modInv(ll b, ll m) {
    ll e = m - 2;
    ll res = 1;
    while (e) {
        if (e & 1) { res = (res * b) % m; }
        e /= 2;
        b = (b * b) % m;
    }
    return res;
}
```

```
19
20 void ntt(poly &a, bool invert, int id) {
21     ll n = (ll)a.size(), m = mod[id];
22     for (ll i = 1, j = 0; i < n; ++i) {
23         ll bit = n >> 1;
24         for (; j >= bit; bit >>= 1) { j -= bit; }
25         j += bit;
26         if (i < j) { swap(a[i], a[j]); }
27     }
28     for (ll len = 2, wlen; len <= n; len <<= 1) {
29         wlen = invert ? root_1[id] : root[id];
30         for (ll i = len; i < root_pw[id]; i <<= 1) { wlen = (wlen * wlen) % m; }
31         for (ll i = 0; i < n; i += len) {
32             ll w = 1;
33             for (ll j = 0; j < len / 2; j++) {
34                 ll u = a[i + j], v = (a[i + j + len / 2] * w) % m;
35                 a[i + j] = (u + v) % m;
36                 a[i + j + len / 2] = (u - v + m) % m;
37                 w = (w * wlen) % m;
38             }
39         }
40     }
41     if (invert) {
42         ll inv = modInv(n, m);
43         for (ll i = 0; i < n; i++) { a[i] = (a[i] * inv) % m; }
44     }
45 }
46
47 poly convolution(poly a, poly b, int id = 0) {
48     ll n = 1LL, len = (1LL + a.size() + b.size());
49     while (n < len) { n *= 2; }
50     a.resize(n);
51     b.resize(n);
52     ntt(a, 0, id);
53     ntt(b, 0, id);
54     poly answer(n);
55     for (ll i = 0; i < n; i++) { answer[i] = (a[i] * b[i]); }
56     ntt(answer, 1, id);
57     return answer;
58 }
```

```
1
2 ll mod_mul(ll a, ll b, ll m) { return (__int128)a * b % m; }
3 ll ext_gcd(ll a, ll b, ll &x, ll &y) {
4     if (!b) {
5         x = 1;
6         y = 0;
7         return a;
8     } else {
9         ll g = ext_gcd(b, a % b, y, x);
10        y -= a / b * x;
11        return g;
12    }
13 }
14
15 // convolution mod 1,097,572,091,361,755,137
16 poly big_convolution(poly a, poly b) {
17     poly r0, r1, answer;
18     r0 = convolution(a, b, 1);
19     r1 = convolution(a, b, 2);
20
```

```
21      ll s, r, p = mod[1] * mod[2];
22      ext_gcd(mod[1], mod[2], r, s);
23
24      answer.resize(r0.size());
25      for (int i = 0; i < (int)answer.size(); i++) {
26          answer[i] = (mod_mul((s * mod[2] + p) % p, r0[i], p) + mod_mul((r * mod[1]
                + p) % p, r1[i], p) + p) % p;
27      }
28      return answer;
29  }
```

5.4 Eliminação Gaussiana

Método de eliminação gaussiana para resolução de sistemas lineares.

- Complexidade de tempo: $O(n^3)$.

Dica: Se os valores forem apenas 0 e 1 o algoritmo [gauss_mod2](gauss_mod2.cpp) é muito mais rápido.

```
1   const double EPS = 1e-9;
2   const int INF = 2; // it doesn't actually have to be infinity or a big number
3
4   int gauss(vector<vector<double>> a, vector<double> &ans) {
5       int n = (int)a.size();
6       int m = (int)a[0].size() - 1;
7
8       vector<int> where(m, -1);
9       for (int col = 0, row = 0; col < m && row < n; ++col) {
10          int sel = row;
11          for (int i = row; i < n; ++i) {
12              if (abs(a[i][col]) > abs(a[sel][col])) { sel = i; }
13          }
14          if (abs(a[sel][col]) < EPS) { continue; }
15          for (int i = col; i <= m; ++i) { swap(a[sel][i], a[row][i]); }
16          where[col] = row;
17
18          for (int i = 0; i < n; ++i) {
19              if (i != row) {
20                  double c = a[i][col] / a[row][col];
21                  for (int j = col; j <= m; ++j) { a[i][j] -= a[row][j] * c; }
22              }
23          }
24          ++row;
25      }
26
27      ans.assign(m, 0);
28      for (int i = 0; i < m; ++i) {
29          if (where[i] != -1) { ans[i] = a[where[i]][m] / a[where[i]][i]; }
30      }
31      for (int i = 0; i < n; ++i) {
32          double sum = 0;
33          for (int j = 0; j < m; ++j) { sum += ans[j] * a[i][j]; }
34          if (abs(sum - a[i][m]) > EPS) { return 0; }
35      }
36
37      for (int i = 0; i < m; ++i) {
38          if (where[i] == -1) { return INF; }
39      }
40      return 1;
```

```
}
```

```
const int N = 105;
const int INF = 2; // tanto faz

// n -> numero de equacoes, m -> numero de variaveis
// a[i][j] para j em [0, m - 1] -> coeficiente da variavel j na iesima equacao
// a[i][j] para j == m -> resultado da equacao da iesima linha
// ans -> bitset vazio, que retornara a solucao do sistema (caso exista)
int gauss(vector<bitset<N>> a, int n, int m, bitset<N> &ans) {
    vector<int> where(m, -1);

    for (int col = 0, row = 0; col < m && row < n; col++) {
        for (int i = row; i < n; i++) {
            if (a[i][col]) {
                swap(a[i], a[row]);
                break;
            }
        }
        if (!a[row][col]) { continue; }
        where[col] = row;

        for (int i = 0; i < n; i++) {
            if (i != row && a[i][col]) { a[i] ^= a[row]; }
        }
        row++;
    }

    for (int i = 0; i < m; i++) {
        if (where[i] != -1) { ans[i] = a[where[i]][m] / a[where[i]][i]; }
    }
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = 0; j < m; j++) { sum += ans[j] * a[i][j]; }
        if (abs(sum - a[i][m]) > 0) {
            return 0; // Sem solucao
        }
    }

    for (int i = 0; i < m; i++) {
        if (where[i] == -1) {
            return INF; // Infinitas solucoes
        }
    }
    return 1; // Unica solucao (retornada no bitset ans)
}
```

5.5 Máximo divisor comum

[Algoritmo de Euclides](gcd.cpp)

Computa o Máximo Divisor Comum (MDC em português; GCD em inglês).

- Complexidade de tempo: O(log(n))

Mais demorado que usar a função do compilador C++ __gcd(a,b).

[Algoritmo de Euclides Estendido](extended_gcd.cpp)

Algoritmo extendido de euclides que computa o Máximo Divisor Comum e os valores x e y tal que a * x + b * y = gcd(a, b).

- Complexidade de tempo: O(log(n))

```
long long gcd(long long a, long long b) { return (b == 0) ? a : gcd(b, a % b); }
```

```
int extended_gcd(int a, int b, int &x, int &y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1;
    while (b) {
        int q = a / b;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a, b) = make_tuple(b, a - q * b);
    }
    return a;
}
```

```
ll extended_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        ll g = extended_gcd(b, a % b, y, x);
        y -= a / b * x;
        return g;
    }
}
```

5.6 Fatoração

[Fatoração Simples](naive_factorize.cpp) Fatora um número N.

- Complexidade de tempo: $O(\sqrt{n})$

[Crivo Linear](linear_sieve_factorize.cpp) Pré-computa todos os fatores primos até MAX. Utilizado para fatorar um número N menor que MAX.

- Complexidade de tempo: Pré-processamento O(MAX) - Complexidade de tempo: Fatoraração O(quantidade de fatores de N) - Complexidade de espaço: O(MAX)

[Fatoração Rápida](fast_factorize.cpp) Utiliza Pollar-Rho e Miller-Rabin (ver em Primos) para fatorar um número N.

- Complexidade de tempo: $O(N^{1/4} \cdot log(N))$

[Pollard-Rho](pollard-rho.cpp) Descobre um divisor de um número N.

- Complexidade de tempo: $O(N^{1/4} \cdot log(N))$ - Complexidade de espaço: $O(N^{1/2})$

```
vector<int> factorize(int n) {
    vector<int> factors;
    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factors.push_back(d);
            n /= d;
        }
    }
    if (n != 1) { factors.push_back(n); }
    return factors;
}
```

```
namespace sieve {
    const int MAX = 1e4;
    int lp[MAX + 1], factor[MAX + 1];
    vector<int> pr;
    void build() {
        for (int i = 2; i <= MAX; ++i) {
            if (lp[i] == 0) {
                lp[i] = i;
                pr.push_back(i);
            }
```

```
            for (int j = 0; i * pr[j] <= MAX; ++j) {
                lp[i * pr[j]] = pr[j];
                factor[i * pr[j]] = i;
                if (pr[j] == lp[i]) { break; }
            }
        }
    }
    vector<int> factorize(int x) {
        if (x < 2) { return {}; }
        vector<int> v;
        for (int lpx = lp[x]; x >= lpx; x = factor[x]) { v.emplace_back(lp[x]); }
        return v;
    }
}
```

```
long long mod_mul(long long a, long long b, long long m) { return (__int128)a * b % m; }

long long pollard_rho(long long n) {
    auto f = [n](long long x) {
        return mod_mul(x, x, n) + 1;
    };
    long long x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 || __gcd(prd, n) == 1) {
        if (x == y) { x = ++i, y = f(x); }
        if ((q = mod_mul(prd, max(x, y) - min(x, y), n))) { prd = q; }
        x = f(x), y = f(f(y));
    }
    return __gcd(prd, n);
}
```

```
// usa miller_rabin.cpp!! olhar em matematica/primos
// usa pollar_rho.cpp!! olhar em matematica/fatoracao

vector<long long> factorize(long long n) {
    if (n == 1) { return {}; }
    if (miller_rabin(n)) { return {n}; }
    long long x = pollard_rho(n);
    auto l = factorize(x), r = factorize(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

5.7 Teorema do Resto Chinês

Resolve em O(n * log(n)) o sistema **(x = rem $_i$ % mod $_i$)** para i entre *0* e *n*

Generalizado!!! Retorna -1 se a resposta não existir

```
ll extended_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    } else {
        ll g = extended_gcd(b, a % b, y, x);
```

```
8           y -= a / b * x;
9           return g;
10      }
11  }
12
13  ll crt(vector<ll> rem, vector<ll> mod) {
14      int n = rem.size();
15      if (n == 0) { return 0; }
16      __int128 ans = rem[0], m = mod[0];
17      for (int i = 1; i < n; i++) {
18          ll x, y;
19          ll g = extended_gcd(mod[i], m, x, y);
20          if ((ans - rem[i]) % g != 0) { return -1; }
21          ans = ans + (__int128)1 * (rem[i] - ans) * (m / g) * y;
22          m = (__int128)(mod[i] / g) * (m / g) * g;
23          ans = (ans % m + m) % m;
24      }
25      return ans;
26  }
```

5.8 Transformada rápida de Fourier

Computa multiplicação de polinômio.

- Complexidade de tempo (caso médio): O(N * log(N)) - Complexidade de tempo (considerando alto overhead): $O(n * log^2(n) * log(log(n)))$

Garante que não haja erro de precisão para polinômios com grau até $3 * 10^5$ e constantes até 10^6.

```
1   typedef complex<double> cd;
2   typedef vector<cd> poly;
3   const double PI = acos(-1);
4
5   void fft(poly &a, bool invert = 0) {
6       int n = a.size(), log_n = 0;
7       while ((1 << log_n) < n) { log_n++; }
8
9       for (int i = 1, j = 0; i < n; ++i) {
10          int bit = n >> 1;
11          for (; j >= bit; bit >>= 1) { j -= bit; }
12          j += bit;
13          if (i < j) { swap(a[i], a[j]); }
14      }
15
16      double angle = 2 * PI / n * (invert ? -1 : 1);
17      poly root(n / 2);
18      for (int i = 0; i < n / 2; ++i) { root[i] = cd(cos(angle * i), sin(angle *
            i)); }
19
20      for (long long len = 2; len <= n; len <<= 1) {
21          long long step = n / len;
22          long long aux = len / 2;
23          for (long long i = 0; i < n; i += len) {
24              for (int j = 0; j < aux; ++j) {
25                  cd u = a[i + j], v = a[i + j + aux] * root[step * j];
26                  a[i + j] = u + v;
27                  a[i + j + aux] = u - v;
28              }
29          }
30      }
```

```
31      if (invert) {
32          for (int i = 0; i < n; ++i) { a[i] /= n; }
33      }
34  }
35
36  vector<long long> convolution(vector<long long> &a, vector<long long> &b) {
37      int n = 1, len = a.size() + b.size();
38      while (n < len) { n <<= 1; }
39      a.resize(n);
40      b.resize(n);
41      poly fft_a(a.begin(), a.end());
42      fft(fft_a);
43      poly fft_b(b.begin(), b.end());
44      fft(fft_b);
45
46      poly c(n);
47      for (int i = 0; i < n; ++i) { c[i] = fft_a[i] * fft_b[i]; }
48      fft(c, 1);
49
50      vector<long long> res(n);
51      for (int i = 0; i < n; ++i) {
52          res[i] = round(c[i].real()); //  res = c[i].real(); se for vector de double
53      }
54      // while(size(res) > 1 && res.back() == 0) res.pop_back(); // apenas para
           quando os zeros direita nao importarem
55      return res;
56  }
```

5.9 Exponenciação modular rápida

Computa $(base^{e}xp)\%mod$. - Complexidade de tempo: $O(log(exp))$. - Complexidade de espaço: $O(1)$

```
1  ll exp_mod(ll base, ll exp) {
2      ll b = base, res = 1;
3      while (exp) {
4          if (exp & 1) { res = (res * b)
               % MOD; }
5          b = (b * b) % MOD;
6          exp /= 2;
7      }
8      return res;
9  }
```

5.10 Totiente de Euler

[Totiente de Euler (Phi) para um número](phi.cpp) Computa o totiente para um único número N.

- Complexidade de tempo: $O(N^(1/2))$

[Totiente de Euler (Phi) entre 1 e N](phi_1_to_n.cpp) Computa o totiente entre 1 e N.

- Complexidade de tempo: O(N * log(log(N)))

```
1  vector<int> phi_1_to_n(int n) {
2      vector<int> phi(n + 1);
3      for (int i = 0; i <= n; i++) { phi[i] = i; }
4      for (int i = 2; i <= n; i++) {
5          if (phi[i] == i) {
6              for (int j = i; j <= n; j += i) { phi[j] -= phi[j] / i; }
7          }
8      }
9      return phi;
10 }
```

```cpp
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;
}
```

5.11 Modular Inverse

The modular inverse of an integer a is another integer x such that a * x is congruent to 1 (mod MOD).

[Modular Inverse](modular_inverse.cpp)

Calculates the modular inverse of a.

Uses the [exp_mod](/Matemática/Exponenciação%20Modular%20Rápida/exp_mod.cpp) algorithm, thus expects MOD to be prime.

* Time Complexity: O(log(MOD)). * Space Complexity: O(1).

[Modular Inverse by Extended GDC](modular_inverse_coprime.cpp)

Calculates the modular inverse of a.

Uses the [extended_gcd](/Matemática/GCD/extended_gcd.cpp) algorithm, thus expects MOD to be coprime with a.

Returns −1 if this assumption is broken.

* Time Complexity: O(log(MOD)). * Space Complexity: O(1).

[Modular Inverse for 1 to MAX](modular_inverse_linear.cpp)

Calculates the modular inverse for all numbers between 1 and MAX.

expects MOD to be prime.

* Time Complexity: O(MAX). * Space Complexity: O(MAX).

[Modular Inverse for all powers](modular_inverse_pow.cpp)

Let b be any integer.

Calculates the modular inverse for all powers of b between b^0 and b^MAX.

Needs you calculate beforehand the modular inverse of b, for 2 it is always (MOD+1)/2.

expects MOD to be coprime with b.

* Time Complexity: O(MAX). * Space Complexity: O(MAX).

```cpp
ll inv[MAX];

void compute_inv(const ll m = MOD) {
    inv[1] = 1;
    for (int i = 2; i < MAX; i++) { inv[i] = m - (m / i) * inv[m % i] % m; }
}
```

```cpp
const ll INVB = (MOD + 1) / 2; // Modular inverse of the base, for 2 it is (MOD+1)/2

ll inv[MAX]; // Modular inverse of b^i

void compute_inv() {
    inv[0] = 1;
```

```
    for (int i = 1; i < MAX; i++) { inv[i] = inv[i - 1] * INVB % MOD; }
}
```

```
ll inv(ll a) { return exp_mod(a, MOD -     2); }
```

```
int inv(int a) {
    int x, y;
    int g = extended_gcd(a, MOD, x, y);
    if (g == 1) { return (x % m + m) %     m; }
    return -1;
}
```

- [**Eliminação Gaussiana**](Elimina%C3%A7%C3%A3o%20Gaussiana) Resolve sistema linear de equações. - [**Exponenciação Modular Rápida**](Exponencia%C3%A7%C3%A3o%20Modular%20R%C3%A1pida Computa potenciação rápido. - [**FFT (Fast Fourier Transform)**](FFT) Multiplica dois polinômios. **double** - [**Fatoração**](Fatora%C3%A7%C3%A3o) Fatora inteiros. - [**GCD (Greatest Common Divisor)**](GCD) Encontra o maior divisor comum. - [**Inverso Modular**](Inverso%20Modular) Calcula o inverso modular. - [**NTT (Numeric Theoric Transform)**](NTT) Multiplica polinômios. **long long** - [**Primos**](Primos) Testes de Primalidade. - [**Sum of floor(n / i)**](Sum%20of%20floor(n%20div%20i) Encontra a soma de n dividido de 1 a n - [**Totiente de Euler**](Totiente%20de%20Euler) Computa o totiente. - [**Teorema do resto Chines**](Teorema%20do%20resto%20Chines) Computa congruencias lineares usando o teorema do resto chines.