

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	Grafos	6
1.1	Matching	6
1.2	Hungarian Algorithm for Bipartite Matching	6
1.3	LCA	7
1.4	LCA	7
1.5	HLD	8
1.6	Heavy-Light Decomposition (hld.cpp)	8
1.7	Binary Lifting	9
1.8	Binary Lifting	9
1.9	Kruskal	11
1.10	Kruskal	11
1.11	Bridge	11
1.12	Graph Center	12
1.13	Graph Center	12
1.14	Dijkstra	13
1.15	Dijkstra	13
1.16	Dijkstra 1:1	13
1.17	Dijkstra 1:N	13
1.18	Dijkstra N:N	13
1.19	Fluxo	14
1.20	Fluxo	14
1.21	Dinic	14
1.22	Edmonds Karp	14
1.23	Min Cost Max Flow	15
1.24	Stoer–Wagner Min-Cut	18

1.25	Stoer-Wagner	18
1.26	2-SAT	19
1.27	2-SAT	19
1.28	Inverse Graph	20
1.29	Inverse Graph	20
1.30	SPFA	21
1.31	Shortest Path Fast Algorithm (SPFA)	21
2	Estruturas de Dados	22
2.1	MergeSort Tree	22
2.2	MergeSort Tree	22
2.3	MergeSort Tree com Update Pontual	22
2.4	Operation Queue	24
2.5	Operation Queue	24
2.6	Operation Stack	25
2.7	Operation Stack	25
2.8	Ordered Set	25
2.9	Ordered Set	25
2.10	Exemplo	26
2.11	LiChao Tree	26
2.12	LiChao Tree	26
2.13	LiChao Tree Sparse	26
2.14	DSU	28
2.15	Disjoint Set Union	28
2.16	DSU Simples	28
2.17	DSU Bipartido	28
2.18	DSU com Rollback	28
2.19	DSU Completo	28
2.20	Kd Fenwick Tree	32
2.21	KD Fenwick Tree	32
2.22	Fenwick Tree	33
2.23	Fenwick Tree	33
2.24	Segment Tree	33
2.25	Segment Tree	33
2.26	Seg Tree	33
2.27	Seg Tree Lazy	34

2.28	Sparse Seg Tree	34
2.29	Persistent Seg Tree	34
2.30	Seg Tree Beats	34
2.31	Seg Tree Beats Max and Sum update	34
2.32	Interval Tree	45
2.33	Interval Tree	45
2.34	Sparse Table	46
2.35	Sparse Table	46
2.36	Disjoint Sparse Table	47
2.37	Disjoint Sparse Table	47
3	String	48
3.1	Aho-Corasick	48
3.2	Aho-Corasick	48
3.3	Hashing	49
3.4	Hashing	49
3.5	Trie	50
3.6	Trie	50
3.7	Prefix Function	50
3.8	Prefix Function	50
3.9	KMP	50
3.10	Autômato de KMP	50
3.11	Prefix Count	50
3.12	Manacher	52
3.13	Algoritmo de Manacher	52
3.14	Lyndon	53
3.15	Lyndon Factorization	53
3.16	Duval	53
3.17	Min Cyclic Shift	53
3.18	Patricia Tree	54
3.19	Patricia Tree ou Patricia Trie	54
3.20	Suffix Array	54
3.21	Suffix Array	54
4	Paradigmas	57
4.1	Busca Ternaria	57
4.2	Busca Ternária	57

4.3	Busca Ternária em Espaço Discreto	57
4.4	Mo	58
4.5	Mo	58
4.6	Mo com Update	58
4.7	All Submasks	60
4.8	All Submask	60
4.9	Exponenciação de Matriz	60
4.10	Exponenciação de Matriz	60
4.11	Uso Comum	60
4.12	Variação que dependa de **constantes** e do **índice**	60
4.13	Variação Multiplicativa	61
4.14	Divide and Conquer	62
4.15	Divide and Conquer	62
4.16	Divide and Conquer com Query on demand	62
4.17	Busca Binária Paralela	64
4.18	Busca Binária Paralela	64
4.19	DP de Permutacao	65
4.20	DP de Permutação	65
4.21	Convex Hull Trick	66
4.22	Convex Hull Trick	66
5	Matemática	67
5.1	Sum of floor($n \div i$)	67
5.2	Soma do floor(n / i)	67
5.3	Primos	67
5.4	Primos	67
5.5	Crivo de Eratóstenes	67
5.6	Miller-Rabin	67
5.7	Teste Ingênuo	67
5.8	NTT	68
5.9	Numeric Theoric Transformation	68
5.10	Totiente de Euler	70
5.11	Totiente de Euler	70
5.12	Totiente de Euler (Phi) para um número	70
5.13	Totiente de Euler (Phi) entre 1 e N	70
5.14	Exponenciação Modular Rápida	70

5.15	Exponenciação modular rápida	70
5.16	Eliminação Gaussiana	71
5.17	Eliminação Gaussiana	71
5.18	Inverso Modular	72
5.19	Modular Inverse	72
5.20	Modular Inverse	72
5.21	Modular Inverse by Extended GDC	73
5.22	Modular Inverse for 1 to MAX	73
5.23	Modular Inverse for all powers	73
5.24	GCD	74
5.25	Máximo divisor comum	74
5.26	Algoritmo de Euclides	74
5.27	Algoritmo de Euclides Estendido	74
5.28	Teorema do Resto Chinês	74
5.29	Teorema do Resto Chinês	74
5.30	Generalizado!!! Retorna -1 se a resposta não existir	74
5.31	Fatoração	75
5.32	Fatoração	75
5.33	Fatoração Simples	75
5.34	Crivo Linear	75
5.35	Fatoração Rápida	75
5.36	Pollard-Rho	75
5.37	FFT	76
5.38	Transformada rápida de Fourier	76

1 Grafos

1.1 Matching

1.2 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 }
```

1.3 LCA

1.4 LCA

Algoritmo de Lowest Common Ancestor usando EulerTour e Sparse Table

Complexidade de tempo:

- $O(N \log(N))$ Preprocessing - $O(1)$ Query LCA

Complexidade de espaço: $O(N \log(N))$

```
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 vector<int> vi;
9 typedef pair<int, int> ii;
10
11 vi tin, tout;
12 vector<vi> adj;
13 vector<ii> prof;
14 vector<vector<ii>> st;
15
16 int n, timer;
17
18 void SparseTable(vector<ii> &v) {
19     int n = v.size();
20     int e = floor(log2(n));
21     st.assign(e + 1, vector<ii>(n));
22     for (int i = 0; i < n; i++) { st[0][i] = v[i]; }
23     for (int i = 1; i <= e; i++) {
24         for (int j = 0; j + (1 << i) <= n; j++) { st[i][j] = min(st[i - 1][j],
25             st[i - 1][j + (1 << (i - 1))]); }
26     }
27
28 void et_dfs(int u, int p, int h) {
29     tin[u] = timer++;
30     prof.emplace_back(h, u);
31     for (int v : adj[u]) {
32         if (v != p) {
33             et_dfs(v, u, h + 1);
34             prof.emplace_back(h, u);
35         }
36     }
37     tout[u] = timer++;
38 }
39
40 void build(int root = 0) {
41     tin.assign(n, 0);
42     tout.assign(n, 0);
43     prof.clear();
44     timer = 0;
45     et_dfs(root, root, 0);
46     SparseTable(prof);
47 }
48
```

```

49 int lca(int u, int v) {
50     int l = tout[u], r = tin[v];
51     if (l > r) { swap(l, r); }
52     int i = floor(log2(r - l + 1));
53     return min(st[i][l], st[i][r - (1 << i) + 1]).se;
54 }
55
56 int main() {
57     cin >> n;
58
59     adj.assign(n, vi(0));
60
61     for (int i = 0; i < n - 1; i++) {
62         int a, b;
63         cin >> a >> b;
64         adj[a].push_back(b);
65         adj[b].push_back(a);
66     }
67
68     build();
69 }

```

1.5 HLD

1.6 Heavy-Light Decomposition (hld.cpp)

Técnica usada para otimizar a execução de operações em árvores.

- Pré-Processamento: $O(N)$ - Range Query/Update: $O(\log(N)) * O(\text{Complexidade de query da estrutura})$ - Point Query/Update: $O(\text{Complexidade de query da estrutura})$ - LCA: $O(\log(N))$ - Subtree Query: $O(\text{Complexidade de query da estrutura})$ - 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 }

```

1.7 Binary Lifting

1.8 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: $O(N \log(N))$

```
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);
18         timer = 0;
19         me = floor(log2(n));
20         st.assign(n, vector<int>(me + 1, 0));
21         et_dfs(root, root);
22     }
23     bool is_ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >=
        tout[v]; }
24     int lca(int u, int v) {
25         if (is_ancestor(u, v)) { return u; }
26         if (is_ancestor(v, u)) { return v; }
27         for (int i = me; i >= 0; i--) {
28             if (!is_ancestor(st[u][i], v)) { u = st[u][i]; }
29         }
30         return st[u][0];
31     }
32     int ancestor(int u, int k) { // k-th ancestor of u
33         for (int i = me; i >= 0; i--) {
34             if ((1 << i) & k) { u = st[u][i]; }
35         }
36         return u;
37     }
38 }

1 namespace st {
2     int n, me;
3     vector<vector<int>> st;
4     void bl_dfs(int u, int p) {
5         st[u][0] = p;
6         for (int i = 1; i <= me; i++) { st[u][i] = st[st[u][i - 1]][i - 1]; }
7         for (int v : adj[u]) {
8             if (v != p) { bl_dfs(v, u); }
9         }
10    }
11    void build(int _n, int root = 0) {
12        n = _n;
13        me = floor(log2(n));
14        st.assign(n, vector<int>(me + 1, 0));
15        bl_dfs(root, root);
16    }
17    int ancestor(int u, int k) { // k-th ancestor of u
18        for (int i = me; i >= 0; i--) {
19            if ((1 << i) & k) { u = st[u][i]; }
20        }
21    }
```

```

21         return u;
22     }
23 }

```

1.9 Kruskal

1.10 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 }

```

1.11 Bridge

Algoritmo que acha pontes utilizando uma dfs

Complexidade de tempo: $O(N + M)$

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

1.12 Graph Center

1.13 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 };

```

1.14 Dijkstra

1.15 Dijkstra

Computa o menor caminho entre nós de um grafo.

1.16 Dijkstra 1:1

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

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

1.17 Dijkstra 1:N

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

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

1.18 Dijkstra N:N

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

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

```

1  const int MAX = 505, INF = 1e9 + 9;
2
3  vector<ii> adj[MAX];
4  int dist[MAX][MAX];
5
6  void dk(int n) {
7      for (int i = 0; i < n; i++) {
8          for (int j = 0; j < n; j++) { dist[i][j] = INF; }
9      }
10     for (int s = 0; s < n; s++) {
11         priority_queue<ii, vector<ii>, greater<ii>> fil;
12         dist[s][s] = 0;
13         fil.emplace(dist[s][s], s);
14         while (!fil.empty()) {
15             auto [d, u] = fil.top();
16             fil.pop();
17             if (d != dist[s][u]) { continue; }
18             for (auto [w, v] : adj[u]) {
19                 if (dist[s][v] > d + w) {

```

```

20         dist[s][v] = d + w;
21         fila.emplace(dist[s][v], v);
22     }
23 }
24 }
25 }
26 }

```

<pre> 1 const int MAX = 1e5 + 5, INF = 1e9 + 9; 2 3 vector<ii> adj[MAX]; 4 int dist[MAX]; 5 6 void dk(int s) { 7 priority_queue<ii, vector<ii>, 8 greater<ii>> fila; 9 fill(begin(dist), end(dist), INF); 10 dist[s] = 0; 11 fila.emplace(dist[s], s); 12 while (!fila.empty()) { </pre>	<pre> 13 14 15 16 17 18 19 20 21 22 </pre>	<pre> 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); } } } } </pre>
---	--	--

<pre> 1 const int MAX = 1e5 + 5, INF = 1e9 + 9; 2 3 vector<ii> adj[MAX]; 4 int dist[MAX]; 5 6 int dk(int s, int t) { 7 priority_queue<ii, vector<ii>, 8 greater<ii>> fila; 9 fill(begin(dist), end(dist), INF); 10 dist[s] = 0; 11 fila.emplace(dist[s], s); 12 while (!fila.empty()) { </pre>	<pre> 13 14 15 16 17 18 19 20 21 22 23 24 </pre>	<pre> 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; } </pre>
---	--	---

1.19 Fluxo

1.20 Fluxo

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

1.21 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 $O(\sqrt{V} * E)$

1.22 Edmonds Karp

Útil para grafos com poucas arestas

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

1.23 Min Cost Max Flow

Computa o fluxo máximo com custo mínimo

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

```
1  const long long INF = 1e18;
2
3  struct FlowEdge {
4      int u, v;
5      long long cap, flow = 0;
6      FlowEdge(int u, int v, long long cap) : u(u), v(v), cap(cap) { }
7  };
8
9  struct EdmondsKarp {
10     int n, s, t, m = 0, vistoken = 0;
11     vector<FlowEdge> edges;
12     vector<vector<int>>> adj;
13     vector<int> visto;
14
15     EdmondsKarp(int n, int s, int t) : n(n), s(s), t(t) {
16         adj.resize(n);
17         visto.resize(n);
18     }
19
20     void add_edge(int u, int v, long long cap) {
21         edges.emplace_back(u, v, cap);
22         edges.emplace_back(v, u, 0);
23         adj[u].push_back(m);
24         adj[v].push_back(m + 1);
25         m += 2;
26     }
27
28     int bfs() {
29         vistoken++;
30         queue<int> fila;
31         fila.push(s);
32         vector<int> pego(n, -1);
33         while (!fila.empty()) {
34             int u = fila.front();
35             if (u == t) { break; }
36             fila.pop();
37             visto[u] = vistoken;
38             for (int id : adj[u]) {
39                 if (edges[id].cap - edges[id].flow < 1) { continue; }
40                 int v = edges[id].v;
41                 if (visto[v] == -1) { continue; }
42                 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,
49             edges[id].cap - edges[id].flow); }
50         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
51             edges[id].flow += f;
52             edges[id ^ 1].flow -= f;
53         }
54         return f;
55     }
56 }
```

```

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             }
41         }
42
43         if (pego[t] == -1) { return 0; }
44         ll f = INF;
45         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
46             f = min(f, edges[id].cap - edges[id].flow);
47             mincost += edges[id].cost;
48         }
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         maxflow += f;

```



```

54         return 1;
55     }
56
57     ll flow() {
58         while (spfa())
59             ;
60         return maxflow;
61     }
62 };

1  typedef long long ll;
2
3  const ll INF = 1e18;
4
5  struct FlowEdge {
6      int u, v;
7      ll cap, flow = 0;
8      FlowEdge(int u, int v, ll cap) : u(u), v(v), cap(cap) { }
9  };

10
11 struct Dinic {
12     vector<FlowEdge> edges;
13     vector<vector<int>>> adj;
14     int n, s, t, m = 0;
15     vector<int> level, ptr;
16     queue<int> q;
17
18     Dinic(int n, int s, int t) : n(n), s(s), t(t) {
19         adj.resize(n);
20         level.resize(n);
21         ptr.resize(n);
22     }
23
24     void add_edge(int u, int v, ll cap) {
25         edges.emplace_back(u, v, cap);
26         edges.emplace_back(v, u, 0);
27         adj[u].push_back(m);
28         adj[v].push_back(m + 1);
29         m += 2;
30     }
31
32     bool bfs() {
33         while (!q.empty()) {
34             int u = q.front();
35             q.pop();
36             for (int id : adj[u]) {
37                 if (edges[id].cap - edges[id].flow < 1) { continue; }
38                 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) {
54             continue; }
55         ll tr = dfs(v, min(f, edges[id].cap - edges[id].flow));
56         if (tr == 0) { continue; }
57         edges[id].flow += tr;
58         edges[id ^ 1].flow -= tr;
59         return tr;
60     }
61     return 0;
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 };

```

1.24 Stoer–Wagner Min-Cut

1.25 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++)
28         adj[prev][i] = adj[i][prev] += adj[sel][i];
29     exist[sel] = false;
30 } else {
31     added[sel] = true;
32     for (int i = 0; i < n; i++) w[i] += adj[sel][i];
33     prev = sel;
34 }
35 }
36 }
37 return bestCost;
38 }

```

1.26 2-SAT

1.27 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 relação 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 };

```

1.28 Inverse Graph

1.29 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 }

```

1.30 SPFA

1.31 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 Estruturas de Dados

2.1 MergeSort Tree

2.2 MergeSort Tree

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

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

2.3 MergeSort Tree com Update Pontual

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 : $O(\log^2(N))$

```
1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3
4 using namespace __gnu_pbds;
5
6 namespace mergesort {
7     typedef tree<ii, null_type, less<ii>, rb_tree_tag,
8         tree_order_statistics_node_update> ordered_set;
9     const int MAX = 1e5 + 5;
10
11     int n;
12     ordered_set mgtree[4 * MAX];
13     vi values;
14
15     int le(int n) { return 2 * n + 1; }
16     int ri(int n) { return 2 * n + 2; }
17
18     ordered_set join(ordered_set set_l, ordered_set set_r) {
19         for (auto v : set_r) { set_l.insert(v); }
20         return set_l;
21     }
22
23     void build(int n, int esq, int dir) {
24         if (esq == dir) {
25             mgtree[n].insert(ii(values[esq], esq));
26         } else {
27             int mid = (esq + dir) / 2;
28             build(le(n), esq, mid);
29             build(ri(n), mid + 1, dir);
30             mgtree[n] = join(mgtree[le(n)], mgtree[ri(n)]);
31         }
32     }
33
34     void build(vi &v) {
35         n = v.size();
36         values = v;
37         build(0, 0, n - 1);
38     }
39
40     int less(int n, int esq, int dir, int l, int r, int k) {
41         if (esq > r || dir < l) { return 0; }
42         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),
71 //         mid+1, dir, l, r));
72 // }
73 // ordered_set debug_query(int l, int r) {return debug_query(0, 0, n-1, l, r);}
74
75 // int greater(int n, int esq, int dir, int l, int r, int k) {
76 //     if (esq > r || dir < l) return 0;
77 //     if (l <= esq && dir <= r) return (r-l+1) - mgtree[n].order_of_key({k,
78 //         1e8});
79 //     int mid = (esq + dir) / 2;
80 //     return greater(le(n), esq, mid, l, r, k) + greater(ri(n), mid+1, dir,
81 //         l, r, k);
82 // }
83 // int greater(int l, int r, int k) {return greater(0, 0, n-1, l, r, k);}
84 };
85
86 namespace mergesort {
87     const int MAX = 1e5 + 5;
88
89     int n;
90     vi mgtree[4 * MAX];
91
92     int le(int n) { return 2 * n + 1; }
93     int ri(int n) { return 2 * n + 2; }
94
95     void build(int n, int esq, int dir, vi &v) {
96         mgtree[n] = vi(dir - esq + 1, 0);
97         if (esq == dir) {
98             mgtree[n][0] = v[esq];
99         } else {
100             int mid = (esq + dir) / 2;
101             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(),
33         mgtree[n].end(), k) - mgtree[n].begin(); }
34     int mid = (esq + dir) / 2;
35     return less(le(n), esq, mid, l, r, k) + less(ri(n), mid + 1, dir, l, r, k);
36 }
37
38 int less(int l, int r, int k) { return less(0, 0, n - 1, l, r, k); }
39
40 // vi debug_query(int n, int esq, int dir, int l, int r) {
41 //     if (esq > r || dir < l) return vi();
42 //     if (l <= esq && dir <= r) return mgtree[n];
43 //     int mid = (esq + dir) / 2;
44 //     auto vl = debug_query(le(n), esq, mid, l, r);
45 //     auto vr = debug_query(ri(n), mid+1, dir, l, r);
46 //     vi ans = vi(vl.size() + vr.size());
47 //     merge(vl.begin(), vl.end(),
48 //         vr.begin(), vr.end(),
49 //         ans.begin());
50 //     return ans;
51 // }
52 // vi debug_query(int l, int r) {return debug_query(0, 0, n-1, l, r);}
53 };

```

2.4 Operation Queue

2.5 Operation Queue

Fila que armazena o resultado do operatório dos itens.

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

```

1  template <typename T> struct op_queue {
2      stack<pair<T, T>> s1, s2;
3      T result;
4      T op(T a, T b) {
5          return a; // TODO: op to compare
6          // min(a, b);
7          // gcd(a, b);
8          // lca(a, b);
9      }
10     T get() {
11         if (s1.empty() || s2.empty()) {
12             return result = s1.empty() ? s2.top().second : s1.top().second;
13         } else {
14             return result = op(s1.top().second, s2.top().second);
15         }
16     }
17 };

```



```

15     }
16 }
17 void add(T element) {
18     result = s1.empty() ? element : op(element, s1.top().second);
19     s1.push({element, result});
20 }
21 void remove() {
22     if (s2.empty()) {
23         while (!s1.empty()) {
24             T elem = s1.top().first;
25             s1.pop();
26             T result = s2.empty() ? elem : op(elem, s2.top().second);
27             s2.push({elem, result});
28         }
29     }
30     T remove_elem = s2.top().first;
31     s2.pop();
32 }
33 };

```

2.6 Operation Stack

2.7 Operation Stack

Pilha que armazena o resultado do operador dos itens.

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

```

1  template <typename T> struct op_stack {
2      stack<pair<T, T>> st;
3      T result;
4      T op(T a, T b) {
5          return a; // TODO: op to compare
6          // min(a, b);
7          // gcd(a, b);
8          // lca(a, b);
9      }
10     T get() { return result = st.top().second; }
11     void add(T element) {
12         result = st.empty() ? element : op(element, st.top().second);
13         st.push({element, result});
14     }
15     void remove() {
16         T removed_element = st.top().first;
17         st.pop();
18     }
19 };

```

2.8 Ordered Set

2.9 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)

2.10 Exemplo

```
1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/trie_policy.hpp>
3
4 using namespace __gnu_pbds;
5 typedef tree<int, null_type, less<int>, rb_tree_tag,
   tree_order_statistics_node_update> ordered_set;
6
7 ordered_set X;
8 X.insert(1);
9 X.insert(2);
10 X.insert(4);
11 X.insert(8);
12 X.insert(16);
13
14 cout<<*X.find_by_order(1)<<endl; // 2
15 cout<<*X.find_by_order(2)<<endl; // 4
16 cout<<*X.find_by_order(4)<<endl; // 16
17 cout<<(end(X)==X.find_by_order(6))<<endl; // true
18
19 cout<<X.order_of_key(-5)<<endl; // 0
20 cout<<X.order_of_key(1)<<endl; // 0
21 cout<<X.order_of_key(3)<<endl; // 2
22 cout<<X.order_of_key(4)<<endl; // 2
23 cout<<X.order_of_key(400)<<endl; // 5
```



```
1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/trie_policy.hpp>
3
4 using namespace __gnu_pbds;
5
6 template <typename T> typedef tree<T, null_type, less<T>, rb_tree_tag,
   tree_order_statistics_node_update> ordered_set;
```

2.11 LiChao Tree

2.12 LiChao Tree

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

Para retornar o mínimo 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: $O(\log(N))$

2.13 LiChao Tree Sparse

O mesmo que a superior, no entanto suporta consultas com $|x| \leq 1e18$.

* Complexidade de consulta : $O(\log(\text{tamanho do intervalo}))$ * Complexidade de update: $O(\log(\text{tamanho do intervalo}))$

```

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 }

```

2.14 DSU

2.15 Disjoint Set Union

2.16 DSU Simples

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.

2.17 DSU Bipartido

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.

2.18 DSU com Rollback

Desfaz as últimas K uniões

- Complexidade de tempo: $O(K)$.

É possível usar um checkpoint, bastando chamar `**rollback()` para ir até o último checkpoint. O rollback não altera a complexidade, uma vez que $K \leq \text{queries}$. `**Só funciona sem compressão de caminho**`

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

2.19 DSU Completo

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

```

1 struct DSU {
2     vector<int> pa, sz;
3     DSU(int n) : pa(n + 1), sz(n + 1, 1) { iota(pa.begin(), pa.end(), 0); }
4     int root(int a) { return pa[a] = (a == pa[a] ? a : root(pa[a])); }
5     bool find(int a, int b) { return root(a) == root(b); }
6     void uni(int a, int b) {
7         int ra = root(a), rb = root(b);
8         if (ra == rb) { return; }
9         if (sz[ra] > sz[rb]) { swap(ra, rb); }
10        pa[ra] = rb;
11        sz[rb] += sz[ra];
12    }
13 };

1 struct rollback_dsu {
2     struct change {
3         int node, old_size;
4     };
5     stack<change> changes;
6     vector<int> parent, size;
7     int number_of_sets;
8
9     rollback_dsu(int n) {
10        size.resize(n + 5, 1);
11        number_of_sets = n;
12        for (int i = 0; i < n + 5; ++i) { parent.push_back(i); }
13    }
14
15    int get(int a) { return (a == parent[a]) ? a : get(parent[a]); }
16    bool same(int a, int b) { return get(a) == get(b); }
17    void checkpoint() { changes.push({-2, 0}); }
18
19    void join(int a, int b) {
20        a = get(a);
21        b = get(b);
22        if (a == b) {
23            changes.push({-1, -1});
24            return;
25        }
26        if (size[a] > size[b]) { swap(a, b); }
27        changes.push({a, size[b]});
28        parent[a] = b;
29        size[b] += size[a];
30        --number_of_sets;
31    }
32
33    void rollback(int qnt = 1 << 31) {
34        for (int i = 0; i < qnt; ++i) {
35            auto ch = changes.top();
36            changes.pop();
37            if (ch.node == -1) { continue; }
38            if (ch.node == -2) {
39                if (qnt == 1 << 31) { break; }
40                --i;
41                continue;
42            }
43            size[parent[ch.node]] = ch.old_size;

```

```

44         parent[ch.node] = ch.node;
45         ++number_of_sets;
46     }
47 }
48 };

1 struct bipartite_dsu {
2     vector<int> parent;
3     vector<int> color;
4     int size;
5     bipartite_dsu(int n) {
6         size = n;
7         color.resize(n + 5, 0);
8         for (int i = 0; i < n + 5; ++i) { parent.push_back(i); }
9     }
10
11     pair<int, bool> get(int a) {
12         if (parent[a] == a) { return {a, 0}; }
13         auto val = get(parent[a]);
14         parent[a] = val.fi;
15         color[a] = (color[a] + val.se) % 2;
16         return {parent[a], color[a]};
17     }
18
19     bool same_color(int a, int b) {
20         get(a);
21         get(b);
22         return color[a] == color[b];
23     }
24     bool same_group(int a, int b) {
25         get(a);
26         get(b);
27         return parent[a] == parent[b];
28     }
29     bool possible_edge(int a, int b) { return !same_color(a, b) || !same_group(a,
30         b); }
31
32     void join(int a, int b) {
33         auto val_a = get(a), val_b = get(b);
34         parent[val_a.fi] = val_b.fi;
35         color[val_a.fi] = (val_a.se + val_b.se + 1) % 2;
36     }
37 };

1 struct full_dsu {
2     struct change {
3         int node, old_size;
4     };
5     struct query {
6         int l, r, u, v, type;
7     };
8     stack<change> changes;
9     map<pair<int, int>, vector<query>> edges;
10    vector<query> queries;
11    vector<int> parent, size;
12    int number_of_sets, time;
13
14    full_dsu(int n) {
15        time = 0;
16        size.resize(n + 5, 1);
17        number_of_sets = n;

```

```

18     loop(i, 0, n + 5) parent.push_back(i);
19 }
20
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 os 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 };

```

2.20 Kd Fenwick Tree

2.21 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 }

```

2.22 Fenwick Tree

2.23 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 };

```

2.24 Segment Tree

2.25 Segment Tree

2.26 Seg Tree

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: $4 * N = O(N)$

2.27 Seg Tree Lazy

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: $2 * 4 * N = O(N)$

2.28 Sparse Seg Tree

Seg Tree Esparsa:

- Complexidade de tempo (Pré-processamento): $O(1)$ - Complexidade de tempo (Consulta em intervalo): $O(\log(N))$ - Complexidade de tempo (Update em ponto): $O(\log(N))$

2.29 Persistent Seg Tree

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**

2.30 Seg Tree Beats

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: $2 * 4 * N = O(N)$

2.31 Seg Tree Beats Max and Sum update

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: $2 * 4 * N = O(N)$

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define ll long long
5 #define INF 1e9
6
7 struct Node {
8     int m1 = INF, m2 = INF, cont = 0, lazy = 0;
9     ll soma = 0;
10
11     void set(int v) {
12         m1 = v;
13         cont = 1;
14         soma = v;
15     }
16 }
```

```

17     void merge(Node a, Node b) {
18         m1 = min(a.m1, b.m1);
19         m2 = INF;
20         if (a.m1 != b.m1) { m2 = min(m2, max(a.m1, b.m1)); }
21         if (a.m2 != m1) { m2 = min(m2, a.m2); }
22         if (b.m2 != m1) { m2 = min(m2, b.m2); }
23         cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
24         soma = a.soma + b.soma;
25     }
26
27     void print() { printf("%d %d %d %lld %d\n", m1, m2, cont, soma, lazy); }
28 };
29
30 int n, q;
31 vector<Node> tree;
32
33 int le(int n) { return 2 * n + 1; }
34 int ri(int n) { return 2 * n + 2; }
35
36 void push(int n, int esq, int dir) {
37     if (tree[n].lazy <= tree[n].m1) { return; }
38     tree[n].soma += (ll)abs(tree[n].m1 - tree[n].lazy) * tree[n].cont;
39     tree[n].m1 = tree[n].lazy;
40     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 < 1) { return 0; }
79     if (1 <= esq && dir <= r) { return tree[n].soma; }
80     int mid = (esq + dir) / 2;
81     return query(le(n), esq, mid, 1, r) + query(ri(n), mid + 1, dir, 1, 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 }

1 const int SEGMAX = 8e6 + 5; // should be Q * log(DIR-ESQ+1)
2 const ll ESQ = 0, DIR = 1e9 + 7;
3
4 struct seg {
5     ll tree[SEGMAX];
6     int R[SEGMAX], L[SEGMAX], ptr = 2; // 0 is NULL; 1 is First Root
7     ll op(ll a, ll b) { return (a + b) % MOD; }
8     int le(int i) {
9         if (L[i] == 0) { L[i] = ptr++; }
10        return L[i];
11    }
12    int ri(int i) {
13        if (R[i] == 0) { R[i] = ptr++; }
14        return R[i];
15    }
16    ll query(ll l, ll r, int n = 1, ll esq = ESQ, ll dir = DIR) {
17        if (r < esq || dir < l) { return 0; }
18        if (l <= esq && dir <= r) { return tree[n]; }
19        ll mid = (esq + dir) / 2;
20        return op(query(l, r, le(n), esq, mid), query(l, r, ri(n), mid + 1, dir));
21    }
22    void update(ll x, ll v, int n = 1, ll esq = ESQ, ll dir = DIR) {
23        if (esq == dir) {
24            tree[n] = (tree[n] + v) % MOD;
25        } else {
26            ll mid = (esq + dir) / 2;
27            if (x <= mid) {
28                update(x, v, le(n), esq, mid);
29            } else {
30                update(x, v, ri(n), mid + 1, dir);
31            }
32            tree[n] = op(tree[le(n)], tree[ri(n)]);
33        }
34    }
35 };

1 const int MAX = 2505;

```

```

2
3 int n, m, mat[MAX][MAX], tree[4 * MAX][4 * MAX];
4
5 int le(int x) { return 2 * x + 1; }
6 int ri(int x) { return 2 * x + 2; }
7
8 void build_y(int nx, int lx, int rx, int ny, int ly, int ry) {
9     if (ly == ry) {
10         if (lx == rx) {
11             tree[nx][ny] = mat[lx][ly];
12         } else {
13             tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
14         }
15     } else {
16         int my = (ly + ry) / 2;
17         build_y(nx, lx, rx, le(ny), ly, my);
18         build_y(nx, lx, rx, ri(ny), my + 1, ry);
19         tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
20     }
21 }
22 void build_x(int nx, int lx, int rx) {
23     if (lx != rx) {
24         int mx = (lx + rx) / 2;
25         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 {
34     if (ly == ry) {
35         if (lx == rx) {
36             tree[nx][ny] = v;
37         } else {
38             tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
39         }
40     } else {
41         int my = (ly + ry) / 2;
42         if (y <= my) {
43             update_y(nx, lx, rx, le(ny), ly, my, x, y, v);
44         } else {
45             update_y(nx, lx, rx, ri(ny), my + 1, ry, x, y, v);
46         }
47         tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
48     }
49 }
50 void update_x(int nx, int lx, int rx, int x, int y, int v) {
51     if (lx != rx) {
52         int mx = (lx + rx) / 2;
53         if (x <= mx) {
54             update_x(le(nx), lx, mx, x, y, v);
55         } else {
56             update_x(ri(nx), mx + 1, rx, x, y, v);
57         }
58     }
59     update_y(nx, lx, rx, 0, 0, m - 1, x, y, v);
60 }
61 void update(int x, int y, int v) { update_x(0, 0, n - 1, x, y, v); }

```

```

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 first 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 < 1) { return 0; }
12         if (1 <= 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,
39         r));
40 }
41 ll query(int l, int r) { return query(0, 0, n - 1, l, r).best; }
42 void update(int n, int esq, int dir, int x, ll v) {
43     if (esq > x || dir < x) { return; }
44     if (esq == dir) {
45         tree[n] = new_node(v);
46     } else {
47         int mid = (esq + dir) / 2;
48         if (x <= mid) {
49             update(le(n), esq, mid, x, v);
50         } else {
51             update(ri(n), mid + 1, dir, x, v);
52         }
53         tree[n] = merge(tree[le(n)], tree[ri(n)]);
54     }
55 }
56 void update(int x, ll v) { update(0, 0, n - 1, x, v); }

```

```

1 namespace seg {
2     const int MAX = 2e5 + 5;
3     const ll NEUTRAL = 0; // merge(a, neutral) = a
4     ll merge(ll a, ll b) { return a + b; }
5     int sz; // size of the array
6     ll tree[4 * MAX], lazy[4 * MAX];
7     int le(int n) { return 2 * n + 1; }
8     int ri(int n) { return 2 * n + 2; }
9     void push(int n, int esq, int dir) {
10         if (lazy[n] == 0) { return; }
11         tree[n] += lazy[n] * (dir - esq + 1);
12         if (esq != dir) {
13             lazy[le(n)] += lazy[n];
14             lazy[ri(n)] += lazy[n];
15         }
16         lazy[n] = 0;
17     }
18     void build(span<const ll> v, int n, int esq, int dir) {
19         if (esq == dir) {
20             tree[n] = v[esq];
21         } else {
22             int mid = (esq + dir) / 2;
23             build(v, le(n), esq, mid);
24             build(v, ri(n), mid + 1, dir);
25             tree[n] = merge(tree[le(n)], tree[ri(n)]);
26         }
27     }
28     void build(span<const ll> v) {
29         sz = v.size();
30         build(v, 0, 0, sz - 1);

```

```

31 }
32 ll query(int l, int r, int n = 0, int esq = 0, int dir = sz - 1) {
33     push(n, esq, dir);
34     if (esq > r || dir < 1) { return NEUTRAL; }
35     if (l <= esq && dir <= r) { return tree[n]; }
36     int mid = (esq + dir) / 2;
37     return merge(query(l, r, le(n), esq, mid), query(l, r, ri(n), mid + 1,
38         dir));
39 }
40 void update(int l, int r, ll v, int n = 0, int esq = 0, int dir = sz - 1) {
41     push(n, esq, dir);
42     if (esq > r || dir < 1) { return; }
43     if (l <= esq && dir <= r) {
44         lazy[n] += v;
45         push(n, esq, dir);
46     } else {
47         int mid = (esq + dir) / 2;
48         update(l, r, v, le(n), esq, mid);
49         update(l, r, v, ri(n), mid + 1, dir);
50         tree[n] = merge(tree[le(n)], tree[ri(n)]);
51     }
52 }

```

```

1 namespace seg {
2     const ll ESQ = 0, DIR = 1e9 + 7;
3     struct node {
4         ll v = 0;
5         node *l = NULL, *r = NULL;
6         node() { }
7         node(ll v) : v(v) { }
8         node(node *l, node *r) : l(l), r(r) { v = l->v + r->v; }
9         void apply() {
10             if (l == NULL) { l = new node(); }
11             if (r == NULL) { r = new node(); }
12         }
13     };
14     vector<node*> roots;
15     void build() { roots.push_back(new node()); }
16     void push(node *n, int esq, int dir) {
17         if (esq != dir) { n->apply(); }
18     }
19     // sum v on x
20     node *update(node *n, int esq, int dir, int x, int v) {
21         push(n, esq, dir);
22         if (esq == dir) { return new node(n->v + v); }
23         int mid = (esq + dir) / 2;
24         if (x <= mid) {
25             return new node(update(n->l, esq, mid, x, v), n->r);
26         } else {
27             return new node(n->l, update(n->r, mid + 1, dir, x, v));
28         }
29     }
30     int update(int root, int pos, int val) {
31         node *novo = update(roots[root], ESQ, DIR, pos, val);
32         roots.push_back(novo);
33         return roots.size() - 1;
34     }
35     // sum in [L, R]
36     ll query(node *n, int esq, int dir, int l, int r) {
37         push(n, esq, dir);

```

```

38         if (esq > r || dir < 1) { return 0; }
39         if (l <= esq && dir <= r) { return n->v; }
40         int mid = (esq + dir) / 2;
41         return query(n->l, esq, mid, 1, r) + query(n->r, mid + 1, dir, 1, r);
42     }
43     ll query(int root, int l, int r) { return query(roots[root], ESQ, DIR, l, r); }
44     // kth min number in [L, R] (l_root can not be 0)
45     int kth(node *L, node *R, int esq, int dir, int k) {
46         push(L, esq, dir);
47         push(R, esq, dir);
48         if (esq == dir) { return esq; }
49         int mid = (esq + dir) / 2;
50         int cont = R->l->v - L->l->v;
51         if (cont >= k) {
52             return kth(L->l, R->l, esq, mid, k);
53         } else {
54             return kth(L->r, R->r, mid + 1, dir, k - cont);
55         }
56     }
57     int kth(int l_root, int r_root, int k) { return kth(roots[l_root - 1],
58         roots[r_root], ESQ, DIR, k); }
59 };

```

2.32 Interval Tree

2.33 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

```

1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/tree_policy.hpp>
3  using namespace __gnu_pbds;
4
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
9              && id < i.id);
10     }
11 };
12 template <class CNI, class NI, class Cmp_Fn, class Allocator> struct
13     intervals_node_update {
14         typedef long long metadata_type;
15         int sz = 0;
16         virtual CNI node_begin() const = 0;
17         virtual CNI node_end() const = 0;
18
19         inline vector<int> overlaps(const long long l, const long long r) {
20             queue<CNI> q;
21             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 :
36         it.get_l_child().get_metadata();
37     const long long r_max = (it.get_r_child() == end_it) ? -INF :
38         it.get_r_child().get_metadata();
39     const_cast<long long &>(it.get_metadata()) = max((*it)->hi, max(l_max,
40         r_max));
41 }
42 };
43
44 typedef tree<interval, null_type, less<interval>, rb_tree_tag,
45     intervals_node_update> interval_tree;

```

2.34 Sparse Table

2.35 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],
9                 st[i - 1][j + (1 << (i - 1))]); }
10        }
11    }
12    // O(log(N)) Query for non overlap friendly operations
13    int logquery(int l, int r) {
14        int res = 2e9;
15        for (int i = e; i >= 0; i--) {
16            if ((1 << i) <= r - l + 1) {
17                res = min(res, st[i][l]);
18                l += 1 << i;
19            }
20        }
21    }
22 }

```

```

20     return res;
21 }
22 // O(1) Query for overlapp 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 };

```

2.36 Disjoint Sparse Table

2.37 Disjoint Sparse Table

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

Pré-processamento em $O(N \log(N))$.

```

1 struct dst {
2     const int neutral = 1;
3 #define comp(a, b) (a | b)
4     vector<vector<int>> t;
5     dst(vector<int> v) {
6         int n, k, sz = v.size();
7         for (n = 1, k = 0; n < sz; n <= 1, k++)
8             ;
9         t.assign(k, vector<int>(n));
10        for (int i = 0; i < n; i++) { t[0][i] = i < sz ? v[i] : neutral; }
11        for (int j = 0, len = 1; j <= k; j++, len <= 1) {
12            for (int s = len; s < n; s += (len < 1)) {
13                t[j][s] = v[s];
14                t[j][s - 1] = v[s - 1];
15                for (int i = 1; i < len; i++) {
16                    t[j][s + i] = comp(t[j][s + i - 1], v[s + i]);
17                    t[j][s - 1 - i] = comp(v[s - 1 - i], t[j][s - i]);
18                }
19            }
20        }
21    }
22    int query(int l, int r) {
23        if (l == r) { return t[0][r]; }
24        int i = 31 - __builtin_clz(l ^ r);
25        return comp(t[i][l], t[i][r]);
26    }
27 };

```

3 String

3.1 Aho-Corasick

3.2 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 occurs + aho[st].cont times
65             h[i] += aho[st].cont;
66         }
67     }
68 }

```

3.3 Hashing

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

3.6 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): $O(|S|)$

```
1 struct trie {
2     map<char, int> trie[100005];
3     int value[100005];
4     int n_nodes = 0;
5     void insert(string &s, int v) {
6         int id = 0;
7         for (char c : s) {
8             if (!trie[id].count(c)) { trie[id][c] = ++n_nodes; }
9             id = trie[id][c];
10        }
11        value[id] = v;
12    }
13    int get_value(string &s) {
14        int id = 0;
15        for (char c : s) {
16            if (!trie[id].count(c)) { return -1; }
17            id = trie[id][c];
18        }
19        return value[id];
20    }
21 };
```

3.7 Prefix Function

3.8 Prefix Function

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

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

3.9 KMP

String matching em $O(n + m)$.

3.10 Autômato de KMP

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

3.11 Prefix Count

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());
3     for (int i = 1, j = 0; i < s.size(); i++) {
```

```

4         while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
5         if (s[i] == s[j]) { j++; }
6         p[i] = j;
7     }
8     return p;
9 }

1 vector<int> pi(string &s) {
2     vector<int> p(s.size());
3     for (int i = 1, j = 0; i < s.size(); i++) {
4         while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
5         if (s[i] == s[j]) { j++; }
6         p[i] = j;
7     }
8     return p;
9 }

10
11 vector<int> kmp(string &s, string t) {
12     t += '$';
13     vector<int> p = pi(t), match;
14     for (int i = 0, j = 0; i < s.size(); i++) {
15         while (j > 0 && s[i] != t[j]) { j = p[j - 1]; }
16         if (s[i] == t[j]) { j++; }
17         if (j == t.size() - 1) { match.push_back(i - j + 1); }
18     }
19     return match;
20 }

1 vector<int> pi(string s) {
2     vector<int> p(s.size());
3     for (int i = 1, j = 0; i < s.size(); i++) {
4         while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
5         if (s[i] == s[j]) { j++; }
6         p[i] = j;
7     }
8     return p;
9 }

10
11 vector<int> prefixCount(string s) {
12     vector<int> p = pi(s + '#');
13     int n = s.size();
14     vector<int> cnt(n + 1, 0);
15     for (int i = 0; i < n; i++) { cnt[p[i]]++; }
16     for (int i = n - 1; i > 0; i--) { cnt[p[i - 1]] += cnt[i]; }
17     for (int i = 0; i <= n; i++) { cnt[i]++; }
18     return cnt;
19 }

1 struct AutKMP {
2     vector<vector<int>> nxt;
3
4     vector<int> pi(string &s) {
5         vector<int> p(s.size());
6         for (int i = 1, j = 0; i < s.size(); i++) {
7             while (j > 0 && s[i] != s[j]) { j = p[j - 1]; }
8             if (s[i] == s[j]) { j++; }
9             p[i] = j;
10        }
11        return p;
12    }
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;

```

3.12 Manacher

3.13 Algoritmo de Manacher

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

$i \dots j$

seja um palindromo.

* Complexidade de tempo: $O(N)$

```

1 struct manacher {
2     long long n, count;
3     vector<int> d1, d2;
4     long long solve(string &s) {
5         n = s.size(), count = 0;
6         solve_odd(s);
7         solve_even(s);
8         return count;
9     }
10 void solve_odd(string &s) {
11     d1.resize(n);
12     for (int i = 0, l = 0, r = -1; i < n; i++) {
13         int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
14         while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) { k++; }
15         count += d1[i] = k--;
16         if (i + k > r) {
17             l = i - k;
18             r = i + k;
19         }
20     }
21 }
22 void solve_even(string &s) {
23     d2.resize(n);
24     for (int i = 0, l = 0, r = -1; i < n; i++) {
25         int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
26         while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) { k++; }

```

```

27         count += d2[i] = k--;
28         if (i + k > r) {
29             l = i - k - 1;
30             r = i + k;
31         }
32     }
33 }
34 } mana;

```

3.14 Lyndon

3.15 Lyndon Factorization

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

3.16 Duval

Gera a Lyndon Factorization de uma string

* Complexidade de tempo: $O(N)$

3.17 Min Cyclic Shift

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

* Complexidade de tempo: $O(N)$

<pre> 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; </pre>	}	<pre> 11 } else { 12 k++; 13 } 14 j++; 15 } 16 while (i <= k) { i += j - k; } 17 return s.substr(ans, n / 2); </pre>
---	---	---

```

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 }

```

3.18 Patricia Tree

3.19 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;

```

3.20 Suffix Array

3.21 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): $O(|S| \cdot \log(|S|))$ * Complexidade de tempo (Contar ocorrencias de S em T): $O(|S| \cdot \log(|T|))$

```

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 ;

4 Paradigmas

4.1 Busca Ternaria

4.2 Busca Ternária

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

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

4.3 Busca Ternária em Espaço Discreto

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: $O(\log(N) * O(\text{eval}))$. Onde N é o tamanho do espaço de busca e $O(\text{eval})$ o custo de avaliação da função.

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

4.4 Mo

4.5 Mo

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

- Complexidade de tempo (Query offline): $O(N * \sqrt{N})$

4.6 Mo com Update

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,
20        (int)queries.size()}); }
21    inline void remove(int idx) {
22        // TODO: remove value at idx from data structure
23    }
24    inline void add(int idx) {
25        // TODO: add value at idx from data structure
26    }
27    inline int get_answer() {
28        // TODO: extract the current answer of the data structure
29        return 0;
30    }
31
32    vector<int> run() {
33        vector<int> answers(queries.size());
34        sort(queries.begin(), queries.end());
35        int L = 0;
36        int R = -1;
37        for (query q : queries) {
38            while (L > q.l) { add(--L); }
39            while (R < q.r) { add(++R); }
40            while (L < q.l) { remove(L++); }
41            while (R > q.r) { remove(R--); }
42            answers[q.idx] = get_answer();
43        }
44    }
45 }
```

```

42     }
43     return answers;
44 }
45
46 };

1 typedef pair<int, int> ii;
2 typedef tuple<int, int, int> iii;
3 int block_sz; // Better if 'const';
4 vector<int> vec;
5 namespace mo {
6     struct query {
7         int l, r, t, idx;
8         bool operator<(query q) const {
9             int _l = l / block_sz;
10            int _r = r / block_sz;
11            int _ql = q.l / block_sz;
12            int _qr = q.r / block_sz;
13            return iii(_l, (_l & 1 ? -_r : _r), (_r & 1 ? t : -t)) <
14                iii(_ql, (_ql & 1 ? -_qr : _qr), (_qr & 1 ? q.t : -q.t));
15        }
16    };
17    vector<query> queries;
18    vector<ii> updates;
19
20    void build(int n) {
21        block_sz = pow(1.4142 * n, 2.0 / 3);
22        // TODO: initialize data structure
23    }
24    inline void add_query(int l, int r) { queries.push_back({l, r,
25        (int)updates.size(), (int)queries.size()}); }
26    inline void add_update(int x, int v) { updates.push_back({x, v}); }
27    inline void remove(int idx) {
28        // TODO: remove value at idx from data structure
29    }
30    inline void add(int idx) {
31        // TODO: add value at idx from data structure
32    }
33    inline void update(int l, int r, int t) {
34        auto &[x, v] = updates[t];
35        if (l <= x && x <= r) { remove(x); }
36        swap(vec[x], v);
37        if (l <= x && x <= r) { add(x); }
38    }
39    inline int get_answer() {
40        // TODO: extract the current answer from the data structure
41        return 0;
42    }
43
44    vector<int> run() {
45        vector<int> answers(queries.size());
46        sort(queries.begin(), queries.end());
47        int L = 0;
48        int R = -1;
49        int T = 0;
50        for (query q : queries) {
51            while (T < q.t) { update(L, R, T++); }
52            while (T > q.t) { update(L, R, --T); }
53            while (L > q.l) { add(--L); }
54            while (R < q.r) { add(++R); }
55            while (L < q.l) { remove(L++); }

```

```

55         while (R > q.r) { remove(R---); }
56         answers[q.idx] = get_answer();
57     }
58     return answers;
59 }
60 };

```

4.7 All Submasks

4.8 All Submask

Percorre todas as submáscaras de uma máscara de tamanho N

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

```

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

```

4.9 Exponenciação de Matriz

4.10 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.

—

4.11 Uso Comum

DP:

$$dp[n] = \sum_{i=1}^n c[i] \cdot dp[n-i]$$

Mapeamento:

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

—

4.12 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:

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

4.13 Variação Multiplicativa

Exemplo de DP:

$$dp[n] = c \times \prod_{i=1}^n dp[n-i]$$

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

$$\log(dp[n]) = \log(c) + \sum_{i=1}^n \log(dp[n-i])$$

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

```

1  ll dp[100];
2  mat T;
3
4  #define MOD 1000000007
5
6  mat mult(mat a, mat b) {
7      mat res(a.size(), vi(b[0].size()));
8      for (int i = 0; i < a.size(); i++) {
9          for (int j = 0; j < b[0].size(); j++) {
10             for (int k = 0; k < b.size(); k++) {
11                 res[i][j] += a[i][k] * b[k][j] % MOD;
12                 res[i][j] %= MOD;
13             }
14         }
15     }
16     return res;
17 }
18
19 mat exp_mod(mat b, ll exp){
20     mat res(b.size(), vi(b.size()));
21     for(int i = 0; i < b.size(); i++) res[i][i] = 1;
22
23     while(exp){
24         if(exp & 1) res = mult(res, b);
25         b = mult(b, b);
26         exp /= 2;
27     }
28     return res;
29 }
30
31 // MUDA MUITO DE ACORDO COM O PROBLEMA
32 // LEIA COMO FAZER O MAPEAMENTO NO README
33 ll solve(ll exp, ll dim){
34     if(exp < dim) return dp[exp];
35
36     T.assign(dim, vi(dim));
37     // TO DO: Preencher a Matriz que vai ser exponenciada
38     // T[0][1] = 1;

```

```

39 // T[1][0] = 1;
40 // T[1][1] = 1;
41
42
43 mat prod = exp_mod(T, exp);
44
45 mat vec; vec.assign(dim, vi(1));
46 for(int i = 0; i < dim; i++) vec[i][0] = dp[i]; // Valores iniciais
47
48 mat ans = mult(prod, vec);
49 return ans[0][0];
50 }

```

4.14 Divide and Conquer

4.15 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: $O(n * k * \log(n) * O(\text{query}))$

4.16 Divide and Conquer com Query on demand

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: $O(n * k * \log(n) * O(\text{update da janela}))$

```

1 namespace DC {
2     vi dp_before, dp_cur;
3     void compute(int l, int r, int optl, int opt) {
4         if (l > r) { return; }
5         int mid = (l + r) >> 1;
6         pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
7         for (int i = optl; i <= min(mid, opt); i++) {
8             best = max(best, {(i ? dp_before[i - 1] : 0) + query(i, mid), i}); //
                min() se quiser minimizar
9         }
10        dp_cur[mid] = best.first;
11        int opt = best.second;
12        compute(l, mid - 1, optl, opt);
13        compute(mid + 1, r, opt, opt);
14    }
15
16    ll solve(int n, int k) {
17        dp_before.assign(n + 5, 0);
18        dp_cur.assign(n + 5, 0);
19        for (int i = 0; i < n; i++) { dp_before[i] = query(0, i); }
20        for (int i = 1; i < k; i++) {
21            compute(0, n - 1, 0, n - 1);
22            dp_before = dp_cur;
23        }
24        return dp_before[n - 1];
25    }
26 };

```

```

1 namespace DC {
2     struct range { // eh preciso definir a forma de calcular o range
3         vi freq;
4         ll sum = 0;
5         int l = 0, r = -1;
6         void back_l(int v) { // Mover o 'l' do range para a esquerda
7             sum += freq[v];
8             freq[v]++;
9             l--;
10        }
11        void advance_r(int v) { // Mover o 'r' do range para a direita
12            sum += freq[v];
13            freq[v]++;
14            r++;
15        }
16        void advance_l(int v) { // Mover o 'l' do range para a direita
17            freq[v]--;
18            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
48                // quiser minimizar
49            removed.push_back(v[s.l]);
50            s.advance_l(v[s.l]);
51        }
52        for (int rem : removed) { s.back_l(v[s.l - 1]); }
53
54        dp_cur[mid] = best.first;
55        int opt = best.second;
56        compute(l, mid - 1, optl, opt);
57        compute(mid + 1, r, opt, optr);
58    }
59
60    ll solve(int n, int k) {
61        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 };

```

4.17 Busca Binaria Paralela

4.18 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.

```

1
2 namespace parallel_binary_search {
3     typedef tuple<int, int, long long, long long> query; //{value, id, l, r}
4     vector<query> queries[1123456]; // pode ser um mapa se
        for muito esparsos
5     long long ans[1123456]; // definir pro tamanho
        das queries
6     long long l, r, mid;
7     int id = 0;
8     void set_lim_search(long long n) {
9         l = 0;
10        r = n;
11        mid = (l + r) / 2;
12    }
13
14    void add_query(long long v) { queries[mid].push_back({v, id++, l, r}); }
15
16    void advance_search(long long v) {
17        // advance search
18    }
19
20    bool satisfies(long long mid, int v, long long l, long long r) {
21        // implement the evaluation
22    }
23
24    bool get_ans() {
25        // implement the get ans
26    }
27
28    void parallel_binary_search(long long l, long long r) {
29
30        bool go = 1;
31        while (go) {
32            go = 0;
33            int i = 0; // outra logica se for usar um mapa

```



```

34         for (auto &vec : queries) {
35             advance_search(i++);
36             for (auto q : vec) {
37                 auto [v, id, l, r] = q;
38                 if (l > r) { continue; }
39                 go = 1;
40                 // return while satisfies
41                 if (satisfies(i, v, l, r)) {
42                     ans[i] = get_ans();
43                     long long mid = (i + l) / 2;
44                     queries[mid] = query(v, id, l, i - 1);
45                 } else {
46                     long long mid = (i + r) / 2;
47                     queries[mid] = query(v, id, i + 1, r);
48                 }
49             }
50             vec.clear();
51         }
52     }
53 }
54
55 } // namespace name

```

4.19 DP de Permutacao

4.20 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).

```

1  const int lim = 17;           // setar para o maximo de itens
2  long double dist[lim][lim];   // eh preciso dar as distancias de n para n
3  long double dp[lim][1 << lim];
4
5  int limMask = (1 << lim) - 1; // 2**(maximo de itens) - 1
6  long double solve(int atual, int mask, int n) {
7      if (dp[atual][mask] != 0) { return dp[atual][mask]; }
8      if (mask == (1 << n) - 1) {
9          return dp[atual][mask] = 0; // o que fazer quando chega no final
10     }
11
12     long double res = 1e13; // pode ser maior se precisar
13     for (int i = 0; i < n; i++) {
14         if (!(mask & (1 << i))) {
15             long double aux = solve(i, mask | (1 << i), n);
16             if (mask) { aux += dist[atual][i]; }
17             res = min(res, aux);
18         }
19     }
20     return dp[atual][mask] = res;
21 }

```

4.21 Convex Hull Trick

4.22 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: $O(1)$

```
1  const ll INF = 1e18 + 18;
2  bool op(ll a, ll b) {
3      return a >= b; // either >= or <=
4  }
5  struct line {
6      ll a, b;
7      ll get(ll x) { return a * x + b; }
8      ll intersect(line l) {
9          return (l.b - b + a - l.a) / (a - l.a); // rounds up for integer only
10     }
11 };
12 deque<pair<line, ll>> fila;
13 void add_line(ll a, ll b) {
14     line nova = {a, b};
15     if (!fila.empty() && fila.back().first.a == a && fila.back().first.b == b) {
16         return; }
17     while (!fila.empty() && op(fila.back().second,
18         nova.intersect(fila.back().first))) { fila.pop_back(); }
19     ll x = fila.empty() ? -INF : nova.intersect(fila.back().first);
20     fila.emplace_back(nova, x);
21 }
22 ll get_binary_search(ll x) {
23     int esq = 0, dir = fila.size() - 1, r = -1;
24     while (esq <= dir) {
25         int mid = (esq + dir) / 2;
26         if (op(x, fila[mid].second)) {
27             esq = mid + 1;
28             r = mid;
29         } else {
30             dir = mid - 1;
31         }
32     }
33     return fila[r].first.get(x);
34 }
35 // O(1), use only when QUERIES are monotonic!
36 ll get(ll x) {
37     while (fila.size() >= 2 && op(x, fila[1].second)) { fila.pop_front(); }
38     return fila.front().first.get(x);
39 }
```

5 Matemática

5.1 Sum of floor($n \div i$)

5.2 Soma do floor(n / i)

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

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

```
1  const int MOD = 1e9 + 7 ;
2
3  long long sumoffloor(long long n){
4      long long answer = 0, i;
5      for (i = 1; i*i <= n; i++) {
6          answer += n/i;
7          answer %= MOD;
8      }
9
10     i--;
11     for (int j = 1; n/(j+1) >= i; j++) {
12         answer += (((n/j - n/(j+1)) %
13                     MOD) * j) % MOD;
14         answer %= MOD;
15     }
16     return answer;
17 }
```

5.3 Primos

5.4 Primos

5.5 Crivo de Eratóstenes

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 $3 * 10^7$.

5.6 Miller-Rabin

Teste de primalidade garantido para números menores do que 2^{64} .

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

5.7 Teste Ingênuo

Computa a primalidade de um número N .

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

```
1  vector<bool> sieve(int n){
2      vector<bool> is_prime(n+5, true);
3      is_prime[0] = false;
4      is_prime[1] = false;
5      long long sq = sqrt(n+5);
6      for(long long i = 2; i <= sq; i++)
7          if(is_prime[i])
8              for (long long j = i * i; j <
9                  n; j+= i)
10                 is_prime[j] = false;
11     return is_prime;
12 }
```

```
1  bool is_prime(int n) {
2      for (long long d = 2; d * d <= n;
3          d++)
4          if (n % d == 0) return false;
5      return true;
6  }
```

```

1 long long power(long long base, long long e, long long mod) {
2     long long result = 1;
3     base %= mod;
4     while (e) {
5         if (e & 1) result = (__int128)result * base % mod;
6         base = (__int128)base * base % mod;
7         e >>= 1;
8     }
9     return result;
10 }
11
12 bool is_composite(long long n, long long a, long long d, int s) {
13     long long x = power(a, d, n);
14     if (x == 1 || x == n - 1) return false;
15     for (int r = 1; r < s; r++) {
16         x = (__int128)x * x % n;
17         if (x == n - 1) return false;
18     }
19     return true;
20 }
21
22 bool miller_rabin(long long n) {
23     if (n < 2) return false;
24     int r = 0;
25     long long d = n - 1;
26     while ((d & 1) == 0) d >>= 1, ++r;
27     for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
28         if (n == a) return true;
29         if (is_composite(n, a, d, r)) return false;
30     }
31     return true;
32 }

```

5.8 NTT

5.9 Numeric Theoric Transformation

Computa multiplicação de polinômio; ****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).

```

1 typedef long long ll;
2 typedef vector<ll> poly;
3
4 ll mod[3] = {998244353LL, 1004535809LL, 1092616193LL};
5 ll root[3] = {102292LL, 12289LL, 23747LL};
6 ll root_1[3] = {116744195LL, 313564925LL, 642907570LL};
7 ll root_pw[3] = {1LL << 23, 1LL << 21, 1LL << 21};
8
9 ll modInv(ll b, ll m) {
10     ll e = m - 2;
11     ll res = 1;
12     while (e) {
13         if (e & 1) res = (res * b) % m;
14         e /= 2;
15         b = (b * b) % m;
16     }

```

```

17     return res;
18 }
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; y = 0;
6         return a;
7     }else{
8         ll g = ext_gcd(b, a%b, y, x);
9         y -= a/b * x;
10        return g;
11    }
12 }
13
14 // convolution mod 1,097,572,091,361,755,137
15 poly big_convolution(poly a, poly b){
16     poly r0, r1, answer;
17     r0 = convolution(a, b, 1);
18     r1 = convolution(a, b, 2);

```

```

19
20     ll s, r, p = mod[1] * mod[2];
21     ext_gcd(mod[1], mod[2], r, s);
22
23     answer.resize(r0.size());
24     for(int i = 0; i < (int) answer.size(); i++){
25         answer[i] = (mod_mul((s*mod[2]+p)%p, r0[i], p) + mod_mul((r*mod[1]+p)%p,
26             r1[i], p) + p) % p;
27     }
28     return answer;
29 }

```

5.10 Totiente de Euler

5.11 Totiente de Euler

5.12 Totiente de Euler (Phi) para um número

Computa o totiente para um único número N.

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

5.13 Totiente de Euler (Phi) entre 1 e N

Computa o totiente entre 1 e N.

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

<pre> 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++) if (phi[i] == i) { </pre>	}	<pre> for (int j = i; j <= n; j += i) phi[j] -= phi[j] / i; return phi; </pre>
<pre> 1 int phi(int n) { 2 int result = n; 3 for (int i = 2; i*i <= n; i++) { 4 if (n % i == 0) { 5 while (n % i == 0) n /= i; 6 result -= result / i; </pre>	}	<pre> } if (n > 1) result -= result / n; return result; </pre>

5.14 Exponenciação Modular Rápida

5.15 Exponenciação modular rápida

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

<pre> 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; </pre>	}	<pre> b = (b * b) % MOD; exp /= 2; } return res; </pre>
---	---	---

5.16 Eliminação Gaussiana

5.17 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]))
13                 sel = i;
14         if (abs (a[sel][col]) < EPS)
15             continue;
16         for (int i=col; i<=m; ++i)
17             swap (a[sel][i], a[row][i]);
18         where[col] = row;
19
20         for (int i=0; i<n; ++i)
21             if (i != row) {
22                 double c = a[i][col] / a[row][col];
23                 for (int j=col; j<=m; ++j)
24                     a[i][j] -= a[row][j] * c;
25             }
26         ++row;
27     }
28
29     ans.assign (m, 0);
30     for (int i=0; i<m; ++i)
31         if (where[i] != -1)
32             ans[i] = a[where[i]][m] / a[where[i]][i];
33     for (int i=0; i<n; ++i) {
34         double sum = 0;
35         for (int j=0; j<m; ++j)
36             sum += ans[j] * a[i][j];
37         if (abs (sum - a[i][m]) > EPS)
38             return 0;
39     }
40
41     for (int i=0; i<m; ++i)
42         if (where[i] == -1)
43             return INF;
44     return 1;
45 }
```

```
1  const int N = 105;
2  const int INF = 2; // tanto faz
3
4  // n -> numero de equacoes, m -> numero de variaveis
```

```

5 // a[i][j] para j em [0, m - 1] -> coeficiente da variavel j na iesima equacao
6 // a[i][j] para j == m -> resultado da equacao da iesima linha
7 // ans -> bitset vazio, que retornara a solucao do sistema (caso exista)
8 int gauss (vector<bitset<N>>a, int n, int m, bitset<N>& ans) {
9     vector<int> where (m, -1);
10
11     for (int col = 0, row = 0; col < m && row < n; col++) {
12         for (int i = row; i < n; i++){
13             if (a[i][col]) {
14                 swap (a[i], a[row]);
15                 break;
16             }
17         }
18         if (!a[row][col]) continue;
19         where[col] = row;
20
21         for (int i = 0; i < n; i++){
22             if (i != row && a[i][col]) a[i] ^= a[row];
23         }
24         row++;
25     }
26
27     for (int i = 0; i < m; i++){
28         if (where[i] != -1){
29             ans[i] = a[where[i]][m] / a[where[i]][i];
30         }
31     }
32     for (int i = 0; i < n; i++){
33         int sum = 0;
34         for (int j = 0; j < m; j++){
35             sum += ans[j] * a[i][j];
36         }
37         if (abs(sum - a[i][m]) > 0) return 0; // Sem solucao
38     }
39
40     for (int i = 0; i < m; i++){
41         if (where[i] == -1) return INF; // Infinitas solucoes
42     }
43     return 1; // Unica solucao (retornada no bitset ans)
44 }

```

5.18 Inverso Modular

5.19 Modular Inverse

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

5.20 Modular Inverse

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: $O(\log(\text{MOD}))$. * Space Complexity: $O(1)$.

5.21 Modular Inverse by Extended GDC

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: $O(\log(\text{MOD}))$. * Space Complexity: $O(1)$.

5.22 Modular Inverse for 1 to MAX

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

expects MOD to be prime.

* Time Complexity: $O(\text{MAX})$. * Space Complexity: $O(\text{MAX})$.

5.23 Modular Inverse for all powers

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 $(\text{MOD}+1)/2$.

expects MOD to be coprime with b .

* Time Complexity: $O(\text{MAX})$. * Space Complexity: $O(\text{MAX})$.

```
1 ll inv[MAX];
2
3 void compute_inv(const ll m=MOD) {
4     inv[1] = 1;
5     for(int i = 2; i < MAX; i++)
6         inv[i] = m - (m/i) * inv[m%i] % m;
7 }
```

```
1 const ll INVB = (MOD + 1) / 2; // Modular inverse of the base, for 2 it is
   (MOD+1)/2
2
3 ll inv[MAX]; // Modular inverse of  $b^i$ 
4
5 void compute_inv() {
6     inv[0] = 1;
7     for(int i = 1; i < MAX; i++)
8         inv[i] = inv[i-1] * INVB % MOD;
9 }
```

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

```
1 int inv(int a) {
2     int x, y;
3     int g = extended_gcd(a, MOD, x, y);
4     if (g == 1) return (x % m + m) % m;
5     return -1;
6 }
```

5.24 GCD

5.25 Máximo divisor comum

5.26 Algoritmo de Euclides

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)`.

5.27 Algoritmo de Euclides Estendido

Algoritmo estendido 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: $O(\log(n))$

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


1 int extended_gcd(int a, int b, int& x, 7 2 int& y) { 3 x = 1, y = 0; 8 4 int x1 = 0, y1 = 1; 5 while (b) { 9 6 int q = a / b; 10 7 tie(x, x1) = make_tuple(x1, x - 11 8 q * x1); 9 } 10 return a; 11 }	tie(y, y1) = make_tuple(y1, y - q * y1); tie(a, b) = make_tuple(b, a - q * b); }
--	--

1 ll extended_gcd(ll a, ll b, ll& x, ll& 2 y) { 3 if(b == 0){ 4 x = 1; y = 0; return a; 5 }else{ 6 ll g = extended_gcd(b, a % b, 7 8 9 10 11 }	y, x); y -= a / b * x; return g; }
--	---

5.28 Teorema do Resto Chinês

5.29 Teorema do Resto Chinês

Resolve em $O(n * \log(n))$ o sistema $x \equiv \text{rem}$

i

$\% \text{ mod}$

i

)** para i entre $*0*$ e $*n*$

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

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

```

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

```

5.31 Fatoração

5.32 Fatoração

5.33 Fatoração Simples

Fatora um número N.

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

5.34 Crivo Linear

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: Fatoração $O(\text{quantidade de fatores de } N)$ - Complexidade de espaço: $O(MAX)$

5.35 Fatoração Rápida

Utiliza Pollar-Rho e Miller-Rabin (ver em Primos) para fatorar um número N.

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

5.36 Pollard-Rho

Descobre um divisor de um número N.

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

<pre> 1 vector<int> factorize(int n) { 2 vector<int> factors; 3 for (long long d = 2; d * d <= n; 4 d++) 5 while (n % d == 0){ </pre>	<div style="border-left: 1px solid black; height: 100px; margin-left: 5px;"></div>	<pre> 6 n /= d; 7 } 8 if (n != 1) factors.push_back(n); 9 return factors; 10 } </pre>
--	--	---

```

1 namespace sieve {
2     const int MAX = 1e4;
3     int lp[MAX+1], factor[MAX+1];
4     vector<int> pr;
5     void build() {
6         for (int i = 2; i <= MAX; ++i) {
7             if (lp[i] == 0) {
8                 lp[i] = i;
9                 pr.push_back(i);
10            }
11            for (int j = 0; i * pr[j] <= MAX; ++j) {
12                lp[i * pr[j]] = pr[j];
13                factor[i * pr[j]] = i;
14                if (pr[j] == lp[i]) break;
15            }
16        }
17    }
18    vector<int> factorize(int x) {
19        if (x < 2) return {};
20        vector<int> v;
21        for (int lpx = lp[x]; x >= lpx; x = factor[x]) v.emplace_back(lp[x]);
22        return v;
23    }
24 }

```

```

1 long long mod_mul(long long a, long long b, long long m){
2     return (__int128)a * b % m;
3 }
4
5 long long pollard_rho(long long n) {
6     auto f = [n](long long x) { return mod_mul(x, x, n) + 1;};
7     long long x = 0, y = 0, t = 30, prd = 2, i = 1, q;
8     while (t++ % 40 || __gcd(prd, n) == 1) {
9         if (x == y) x = ++i, y = f(x);
10        if ((q = mod_mul(prd, max(x,y) - min(x,y), n))) prd = q;
11        x = f(x), y = f(f(y));
12    }
13    return __gcd(prd, n);
14 }

```

<pre> 1 // usa miller_rabin.cpp!! olhar em matematica/primos 2 // usa pollard_rho.cpp!! olhar em matematica/fatoracao 3 4 vector<long long> factorize(long long n) { 5 if (n == 1) return {}; </pre>	<div style="border-left: 1px solid black; height: 100px; margin: 0 5px;"></div>	<pre> 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; </pre>
--	---	---

5.37 FFT

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

```