

BRUTE UDESC

Eduardo Scharwz Moreira, Eliton Machado da Silva, Enzo de Almeida Rodrigues,
Eric Grochowicz, Igor Froehner, João Vitor Frölich,
João Marcos de Oliveira, Rafael Granza de Mello e Vinicius Gasparini

5 de fevereiro de 2024

Índice

1	Estruturas de Dados	6
1.1	Disjoint Set Union	6
1.1.1	DSU	6
1.1.2	DSU Bipartido	6
1.1.3	DSU Rollback	7
1.1.4	DSU Rollback Bipartido	7
1.1.5	Offline DSU	8
1.2	Fenwick Tree	9
1.2.1	Fenwick	9
1.2.2	Kd Fenwick Tree	10
1.3	Interval Tree	10
1.4	LiChao Tree	11
1.5	MergeSort Tree	12
1.6	Operation Queue	14
1.7	Operation Stack	15
1.8	Ordered Set	15
1.9	Segment Tree	16

<i>ÍNDICE</i>	2
1.9.1 Segment Tree	16
1.9.2 Segment Tree 2D	17
1.9.3 Segment Tree Beats Max And Sum Update	18
1.9.4 Segment Tree Beats Max Update	20
1.9.5 Segment Tree Esparsa	21
1.9.6 Segment Tree Kadani	22
1.9.7 Segment Tree Lazy	23
1.9.8 Segment Tree Persistente	24
1.10 Sparse Table	25
1.10.1 Disjoint Sparse Table	25
1.10.2 Sparse Table	25
2 Grafos	27
2.1 2 SAT	27
2.2 Binary Lifting	28
2.3 Bridge	29
2.4 Fluxo	29
2.5 Graph Center	32
2.6 HLD	33
2.7 Inverse Graph	34
2.8 Kruskal	35
2.9 LCA	35
2.10 Matching	36
2.10.1 Hungaro	36
2.11 Shortest Paths	37
2.11.1 Dijkstra	37

<i>ÍNDICE</i>	3
2.11.2 SPFA	38
2.12 Stoer–Wagner Min Cut	39
3 String	40
3.1 Aho Corasick	40
3.2 Hashing	41
3.3 Lyndon	41
3.4 Manacher	42
3.5 Patricia Tree	43
3.6 Prefix Function	43
3.7 Suffix Array	45
3.8 Trie	46
4 Paradigmas	47
4.1 All Submasks	47
4.2 Busca Binaria Paralela	47
4.3 Busca Ternaria	48
4.4 Convex Hull Trick	49
4.5 DP de Permutacao	49
4.6 Divide and Conquer	50
4.7 Exponenciação de Matriz	52
4.8 Mo	53
5 Primitivas	56
5.1 Modular Int	56

6	Matemática	58
6.1	Eliminação Gaussiana	58
6.1.1	Gauss	58
6.1.2	Gauss Mod 2	59
6.2	Exponenciação Modular Rápida	59
6.3	FFT	60
6.4	Fatoração	61
6.5	GCD	62
6.6	Inverso Modular	63
6.7	NTT	64
6.8	Primos	65
6.9	Sum of floor (n div i)	67
6.10	Teorema do Resto Chinês	67
6.11	Totiente de Euler	67
7	Teórico	69
7.1	Alguns Números Primos	70
7.1.1	Primo com Truncamento à Esquerda	70
7.1.2	Números Primos de Mersenne	70
7.2	Constantes em C++	70
7.3	Operadores Lineares	70
7.3.1	Rotação no sentido anti-horário por θ°	70
7.3.2	Reflexão em relação à reta $y = mx$	70
7.3.3	Inversa de uma matriz 2x2 A	71
7.3.4	Cisalhamento horizontal por K	71
7.3.5	Cisalhamento vertical por K	71

<i>ÍNDICE</i>	5
7.3.6 Mudança de base	71
7.3.7 Propriedades das operações de matriz	71

Capítulo 1

Estruturas de Dados

1.1 Disjoint Set Union

1.1.1 DSU

Estrutura que mantém uma coleção de conjuntos e permite as operações de unir dois conjuntos e verificar em qual conjunto um elemento está, ambas em $\mathcal{O}(1)$ amortizado. O método *find* retorna o representante do conjunto que contém o elemento, e o método *unite* une os conjuntos que contém os elementos dados, retornando *true* se eles estavam em conjuntos diferentes e *false* caso contrário.

Código: dsu.cpp

```
1 struct DSU {
2     vector<int> par, sz;
3     int number_of_sets;
4     DSU(int n = 0) : par(n), sz(n, 1), number_of_sets(n) {
5         iota(par.begin(), par.end(), 0);
6     }
7     int find(int a) { return a == par[a] ? a : par[a] = find(par[a]); }
8     bool unite(int a, int b) {
9         a = find(a), b = find(b);
10        if (a == b) {
11            return false;
12        }
13        number_of_sets--;
14        if (sz[a] < sz[b]) {
15            swap(a, b);
```

```
16        }
17        par[b] = a;
18        sz[a] += sz[b];
19        return true;
20    }
21 };
```

1.1.2 DSU Bipartido

DSU que mantém se um conjunto é bipartido (visualize os conjuntos como componentes conexas de um grafo e os elementos como vértices). O método *unite* adiciona uma aresta entre os dois elementos dados, e retorna *true* se os elementos estavam em conjuntos diferentes (componentes conexas diferentes) e *false* caso contrário. O método *bipartite* retorna *true* se o conjunto (componente conexa) que contém o elemento dado é bipartido e *false* caso contrário. Todas as operações são $\mathcal{O}(\log n)$.

Código: bipartite_dsu.cpp

```
1 struct Bipartite_DSU {
2     vector<int> par, sz, c, bip;
3     int number_of_sets, all_bipartite;
4     Bipartite_DSU(int n = 0)
5         : par(n), sz(n, 1), c(n), bip(n, 1), number_of_sets(n), all_bipartite(1) {
6         iota(par.begin(), par.end(), 0);
7     }
```

```

8  int find(int a) { return a == par[a] ? a : find(par[a]); }
9  int color(int a) { return a == par[a] ? c[a] : c[a] ^ color(par[a]); }
10 bool bipartite(int a) { return bip[find(a)]; }
11 bool unite(int a, int b) {
12     bool equal_color = color(a) == color(b);
13     a = find(a), b = find(b);
14     if (a == b) {
15         if (equal_color) {
16             bip[a] = 0;
17             all_bipartite = 0;
18         }
19         return false;
20     }
21     if (sz[a] < sz[b]) {
22         swap(a, b);
23     }
24     number_of_sets--;
25     par[b] = a;
26     sz[a] += sz[b];
27     if (equal_color) {
28         c[b] = 1;
29     }
30     bip[a] &= bip[b];
31     all_bipartite &= bip[a];
32     return true;
33 }
34 };

```

1.1.3 DSU Rollback

DSU que desfaz as últimas operações. O método *checkpoint* salva o estado atual da estrutura, e o método *rollback* desfaz as últimas operações até o último checkpoint. As operações de unir dois conjuntos e verificar em qual conjunto um elemento está são $\mathcal{O}(\log n)$, o rollback é $\mathcal{O}(k)$, onde k é o número de alterações a serem desfeitas e o *checkpoint* é $\mathcal{O}(1)$. Importante notar que o rollback não altera a complexidade de uma solução, uma vez que $\sum k = \mathcal{O}(q)$, onde q é o número de operações realizadas.

Codigo: rollback_dsu.cpp

```

1 struct Rollback_DSU {
2     vector<int> par, sz;

```

```

3     int number_of_sets;
4     stack<stack<pair<int &, int>>> changes;
5     Rollback_DSU(int n = 0) : par(n), sz(n, 1), number_of_sets(n) {
6         iota(par.begin(), par.end(), 0);
7         changes.emplace();
8     }
9     int find(int a) { return a == par[a] ? a : find(par[a]); }
10    void checkpoint() { changes.emplace(); }
11    void save(int &a) { changes.top().emplace(a, a); }
12    bool unite(int a, int b) {
13        a = find(a), b = find(b);
14        if (a == b) {
15            return false;
16        }
17        if (sz[a] < sz[b]) {
18            swap(a, b);
19        }
20        save(number_of_sets);
21        save(par[b]);
22        save(sz[a]);
23        number_of_sets--;
24        par[b] = a;
25        sz[a] += sz[b];
26        return true;
27    }
28    void rollback() {
29        while (changes.top().size()) {
30            auto [a, b] = changes.top().top();
31            a = b;
32            changes.top().pop();
33        }
34        changes.pop();
35    }
36 };

```

1.1.4 DSU Rollback Bipartido

DSU com rollback e bipartido.

Codigo: full_dsu.cpp

```

1 struct Full_DSU {
2     vector<int> par, sz, c, bip;

```



```

3  int number_of_sets, all_bipartite;
4  stack<stack<pair<int &, int>>> changes;
5  Full_DSU(int n = 0)
6      : par(n), sz(n, 1), c(n), bip(n, 1), number_of_sets(n), all_bipartite(1) {
7      iota(par.begin(), par.end(), 0);
8      changes.emplace();
9  }
10 int find(int a) { return a == par[a] ? a : find(par[a]); }
11 int color(int a) { return a == par[a] ? c[a] : c[a] ^ color(par[a]); }
12 bool bipartite(int a) { return bip[find(a)]; }
13 void checkpoint() { changes.emplace(); }
14 void save(int &a) { changes.top().emplace(a, a); }
15 bool unite(int a, int b) {
16     bool equal_color = color(a) == color(b);
17     a = find(a), b = find(b);
18     if (a == b) {
19         if (equal_color) {
20             save(bip[a]);
21             save(all_bipartite);
22             bip[a] = 0;
23             all_bipartite = 0;
24         }
25         return false;
26     }
27     if (sz[a] < sz[b]) {
28         swap(a, b);
29     }
30     save(number_of_sets);
31     save(par[b]);
32     save(sz[a]);
33     save(c[b]);
34     save(bip[a]);
35     save(all_bipartite);
36     number_of_sets--;
37     par[b] = a;
38     sz[a] += sz[b];
39     if (equal_color) {
40         c[b] = 1;
41     }
42     bip[a] &= bip[b];
43     all_bipartite &= bip[a];
44     return true;
45 }
46 void rollback() {
47     while (changes.top().size()) {
48         auto [a, b] = changes.top().top();
49         a = b;

```

```

50         changes.top().pop();
51     }
52     changes.pop();
53 }
54 };

```

1.1.5 Offline DSU

Algoritmo que utiliza o Full DSU (DSU com Rollback e Bipartido) que permite adição e **remoção** de arestas. O algoritmo funciona de maneira offline, recebendo previamente todas as operações de adição e remoção de arestas, bem como todas as perguntas (de qualquer tipo, conectividade, bipartição, etc), e retornando as respostas para cada pergunta no retorno do método *solve*. Complexidade total $\mathcal{O}(q \cdot (\log q + \log n))$, onde q é o número de operações realizadas e n é o número de nodos.

Codigo: offline_dsu.cpp

```

1  struct Offline_DSU : Full_DSU {
2      int time;
3      Offline_DSU(int n = 0) : Full_DSU(n), time(0) { }
4      struct query {
5          int type, a, b;
6      };
7      vector<query> queries;
8      void askConnect(int a, int b) {
9          if (a > b) {
10             swap(a, b);
11         }
12         queries.push_back({0, a, b});
13         time++;
14     }
15     void askBipartite(int a) {
16         queries.push_back({1, a, -1});
17         time++;
18     }
19     void askAllBipartite() {
20         queries.push_back({2, -1, -1});
21         time++;
22     }
23     void addEdge(int a, int b) {
24         if (a > b) {

```

```

25     swap(a, b);
26 }
27 queries.push_back({3, a, b});
28 time++;
29 }
30 void removeEdge(int a, int b) {
31     if (a > b) {
32         swap(a, b);
33     }
34     queries.push_back({4, a, b});
35     time++;
36 }
37 vector<vector<pair<int, int>>> lazy;
38 void update(int l, int r, pair<int, int> edge, int u, int L, int R) {
39     if (R < l || L > r) {
40         return;
41     }
42     if (L >= l && R <= r) {
43         lazy[u].push_back(edge);
44         return;
45     }
46     int mid = (L + R) / 2;
47     update(l, r, edge, 2 * u, L, mid);
48     update(l, r, edge, 2 * u + 1, mid + 1, R);
49 }
50 void dfs(int u, int L, int R, vector<int> &ans) {
51     if (L > R) {
52         return;
53     }
54     checkpoint();
55     for (auto [a, b] : lazy[u]) {
56         unite(a, b);
57     }
58     if (L == R) {
59         auto [type, a, b] = queries[L];
60         if (type == 0) {
61             ans.push_back(find(a) == find(b));
62         } else if (type == 1) {
63             ans.push_back(bipartite(a));
64         } else if (type == 2) {
65             ans.push_back(all_bipartite);
66         }
67     } else {
68         int mid = (L + R) / 2;
69         dfs(2 * u, L, mid, ans);
70         dfs(2 * u + 1, mid + 1, R, ans);
71     }

```

```

72     rollback();
73 }
74 vector<int> solve() {
75     lazy.assign(4 * time, {});
76     map<pair<int, int>, int> edges;
77     for (int i = 0; i < time; i++) {
78         auto [type, a, b] = queries[i];
79         if (type == 3) {
80             edges[{a, b}] = i;
81         } else if (type == 4) {
82             update(edges[{a, b}], i, {a, b}, 1, 0, time - 1);
83             edges.erase({a, b});
84         }
85     }
86     for (auto [k, v] : edges) {
87         update(v, time - 1, k, 1, 0, time - 1);
88     }
89     vector<int> ans;
90     dfs(1, 0, time - 1, ans);
91     return ans;
92 }
93 };

```

1.2 Fenwick Tree

1.2.1 Fenwick

Árvore de Fenwick (ou BIT) é uma estrutura de dados que permite atualizações pontuais e consultas de prefixos em um vetor em $\mathcal{O}(\log n)$. A implementação abaixo é 0-indexada (é mais comum encontrar a implementação 1-indexada). A consulta em ranges arbitrários com o método *query* é possível para qualquer operação inversível, como soma, XOR, multiplicação, etc. A implementação abaixo é para soma, mas é fácil adaptar para outras operações. O método *update* soma d à posição i do vetor, enquanto o método *updateSet* substitui o valor da posição i do vetor por d .

Codigo: fenwick_tree.cpp

```

1 template <typename T> struct FenwickTree {
2     int n;

```

```

3   vector<T> bit, arr;
4   FenwickTree(int n = 0) : n(n), bit(n), arr(n) { }
5   FenwickTree(vector<T> &v) : n(v.size()), bit(n), arr(v) {
6       for (int i = 0; i < n; i++) {
7           bit[i] = arr[i];
8       }
9       for (int i = 0; i < n; i++) {
10          int j = i | (i + 1);
11          if (j < n) {
12              bit[j] = bit[j] + bit[i];
13          }
14      }
15  }
16  T pref(int x) {
17      T res = T();
18      for (int i = x; i >= 0; i = (i & (i + 1)) - 1) {
19          res = res + bit[i];
20      }
21      return res;
22  }
23  T query(int l, int r) {
24      if (l == 0) {
25          return pref(r);
26      }
27      return pref(r) - pref(l - 1);
28  }
29  void update(int x, T d) {
30      for (int i = x; i < n; i = i | (i + 1)) {
31          bit[i] = bit[i] + d;
32      }
33      arr[x] = arr[x] + d;
34  }
35  void updateSet(int i, T d) {
36      // funciona pra fenwick de soma
37      update(i, d - arr[i]);
38      arr[i] = d;
39  }
40 };

```

1.2.2 Kd Fenwick Tree

Fenwick Tree em k dimensões. Faz apenas queries de prefixo e updates pontuais em $\mathcal{O}(\log^k(n))$. Para queries em range, deve-se fazer inclusão-exclusão, porém a complexi-

dade fica exponencial, para k dimensões a query em range é $\mathcal{O}(2^k \log^k(n))$.

Codigo: kd_fenwick_tree.cpp

```

1   const int MAX = 20;
2   long long tree[MAX][MAX][MAX][MAX]; // insira o numero de dimensoes aqui
3
4   long long query(vector<int> s, int pos = 0) { // s eh a coordenada
5       long long sum = 0;
6       while (s[pos] >= 0) {
7           if (pos < (int)s.size() - 1) {
8               sum += query(s, pos + 1);
9           } else {
10              sum += tree[s[0]][s[1]][s[2]][s[3]];
11              // atualizar se mexer no numero de dimensoes
12          }
13          s[pos] = (s[pos] & (s[pos] + 1)) - 1;
14      }
15      return sum;
16  }
17
18  void update(vector<int> s, int v, int pos = 0) {
19      while (s[pos] < MAX) {
20          if (pos < (int)s.size() - 1) {
21              update(s, v, pos + 1);
22          } else {
23              tree[s[0]][s[1]][s[2]][s[3]] += v;
24              // atualizar se mexer no numero de dimensoes
25          }
26          s[pos] |= s[pos] + 1;
27      }
28  }

```

1.3 Interval Tree

Por Rafael Granza de Mello

Estrutura que trata intersecções de intervalos.

Capaz de retornar todos os intervalos que intersectam $[L, R]$. Contém métodos $insert(L, R, ID)$, $erase(L, R, ID)$, $overlaps(L, R)$ e $find(L, R, ID)$. É necessário inserir e apagar indicando tanto os limites quanto o ID do intervalo. Todas as operações são $\mathcal{O}(\log n)$,

exceto *overlaps* que é $\mathcal{O}(k + \log n)$, onde k é o número de intervalos que intersectam $[L, R]$. Também podem ser usadas as operações padrões de um *std::set*.

Código: interval_tree.cpp

```
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 tuple(lo, hi, id) < tuple(i.lo, i.hi, i.id);
9     }
10 };
11
12 const long long INF = 1e18;
13
14 template <class CNI, class NI, class Cmp_Fn, class Allocator>
15 struct intervals_node_update {
16     typedef long long metadata_type;
17     int sz = 0;
18     virtual CNI node_begin() const = 0;
19     virtual CNI node_end() const = 0;
20     inline vector<int> overlaps(const long long l, const long long r) {
21         queue<CNI> q;
22         q.push(node_begin());
23         vector<int> vec;
24         while (!q.empty()) {
25             CNI it = q.front();
26             q.pop();
27             if (it == node_end()) {
28                 continue;
29             }
30             if (r >= (*it)->lo && l <= (*it)->hi) {
31                 vec.push_back((*it)->id);
32             }
33             CNI l_it = it.get_l_child();
34             long long l_max = (l_it == node_end()) ? -INF : l_it.get_metadata();
35             if (l_max >= l) {
36                 q.push(l_it);
37             }
38             if ((*it)->lo <= r) {
39                 q.push(it.get_r_child());
40             }
41         }
42     }
43 }
```

```
42         return vec;
43     }
44     inline void operator()(NI it, CNI end_it) {
45         const long long l_max =
46             (it.get_l_child() == end_it) ? -INF : it.get_l_child().get_metadata();
47         const long long r_max =
48             (it.get_r_child() == end_it) ? -INF : it.get_r_child().get_metadata();
49         const_cast<long long &>(it.get_metadata()) = max((*it)->hi, max(l_max, r_max));
50     }
51 };
52 typedef tree<interval, null_type, less<interval>, rb_tree_tag, intervals_node_update>
53     interval_tree;
```

1.4 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. Ou, alterar a função de comparação da árvore se souber mexer.

Funciona para funções com a seguinte propriedade, sejam duas funções $f(x)$ e $g(x)$, uma vez que $f(x)$ passa a ganhar/perder pra $g(x)$, $f(x)$ nunca mais passa a perder/ganhar pra $g(x)$. Em outras palavras, $f(x)$ e $g(x)$ se intersectam no máximo uma vez.

Essa implementação está pronta para usar função linear do tipo $f(x) = ax + b$.

Sendo L o tamanho do intervalo:

- Complexidade de consulta: $\mathcal{O}(\log(L))$
- Complexidade de update: $\mathcal{O}(\log(L))$

Código: lichao_tree.cpp

```
1 typedef long long ll;
2
3 const ll MAXN = 2e5 + 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]) {
13         L[n] = ++idx;
14     }
15     return L[n];
16 }
17 int ri(int n) {
18     if (!R[n]) {
19         R[n] = ++idx;
20     }
21     return R[n];
22 }
23
24 void insert(Line line, int n = 0, ll l = -MAXR, ll r = MAXR) {
25     ll mid = (l + r) / 2;
26     bool b1 = line(l) < tree[n](l);
27     bool bm = line(mid) < tree[n](mid);
28     if (!bm) {
29         swap(tree[n], line);
30     }
31     if (l == r) {
32         return;
33     }
34     if (b1 != bm) {
35         insert(line, le(n), l, mid);
36     } else {
37         insert(line, ri(n), mid + 1, r);
38     }
39 }
40
41 __int128 query(int x, int n = 0, ll l = -MAXR, ll r = MAXR) {
42     if (l == r) {
43         return tree[n](x);
44     }
45     ll mid = (l + r) / 2;
46     if (x < mid) {
47         return max(tree[n](x), query(x, le(n), l, mid));
48     } else {
49         return max(tree[n](x), query(x, ri(n), mid + 1, r));
50     }
51 }

```

1.5 MergeSort Tree

Árvore que resolve queries que envolvam ordenação em range.

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

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 : $\mathcal{O}(N * \log^2(N))$
- Complexidade de consulta : $\mathcal{O}(\log^2(N))$
- Complexidade de update : $\mathcal{O}(\log^2(N))$

Codigo: mergesort_tree_update.cpp

```

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>
9                 ordered_set;
10     const int MAX = 1e5 + 5;
11
12     int n;
13     ordered_set mgtree[4 * MAX];
14     vi values;
15
16     int le(int n) { return 2 * n + 1; }
17     int ri(int n) { return 2 * n + 2; }
18
19     ordered_set join(ordered_set set_l, ordered_set set_r) {

```

```

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

```

```

66         mgtree[n].erase(ii(values[x], x));
67         mgtree[n].insert(ii(v, x));
68     }
69 }
70 void update(int x, int v) {
71     update(0, 0, n - 1, x, v);
72     values[x] = v;
73 }
74
75 // ordered_set debug_query(int n, int esq, int
76 // dir, int l, int r) {
77 // if (esq > r || dir < l) return
78 // ordered_set(); if (l <= esq && dir <=
79 // r) return mgtree[n]; int mid = (esq +
80 // dir) / 2; return
81 // join(debug_query(le(n), esq, mid, l,
82 // r), debug_query(ri(n), mid+1, dir, l,
83 // r));
84 // }
85 // ordered_set debug_query(int l, int r)
86 // {return debug_query(0, 0, n-1, l, r);}
87
88 // int greater(int n, int esq, int dir, int l,
89 // int r, int k) {
90 // if (esq > r || dir < l) return 0;
91 // if (l <= esq && dir <= r) return
92 // (r-l+1) - mgtree[n].order_of_key({k,
93 // 1e8}); int mid = (esq + dir) / 2;
94 // return greater(le(n), esq, mid, l, r,
95 // k) + greater(ri(n), mid+1, dir, l, r,
96 // k);
97 // }
98 // int greater(int l, int r, int k) {return
99 // greater(0, 0, n-1, l, r, k);}
100 };

```

Codigo: mergesort_tree.cpp

```

1 namespace mergesort {
2     const int MAX = 1e5 + 5;
3
4     int n;
5     vi mgtree[4 * MAX];
6
7     int le(int n) { return 2 * n + 1; }
8     int ri(int n) { return 2 * n + 2; }
9

```

```

10 void build(int n, int esq, int dir, vi &v) {
11     mgtree[n] = vi(dir - esq + 1, 0);
12     if (esq == dir) {
13         mgtree[n][0] = v[esq];
14     } else {
15         int mid = (esq + dir) / 2;
16         build(le(n), esq, mid, v);
17         build(ri(n), mid + 1, dir, v);
18         merge(mgtree[le(n)].begin(),
19             mgtree[le(n)].end(),
20             mgtree[ri(n)].begin(),
21             mgtree[ri(n)].end(),
22             mgtree[n].begin());
23     }
24 }
25 void build(vi &v) {
26     n = v.size();
27     build(0, 0, n - 1, v);
28 }
29
30 int less(int n, int esq, int dir, int l, int r, int k) {
31     if (esq > r || dir < l) {
32         return 0;
33     }
34     if (l <= esq && dir <= r) {
35         return lower_bound(mgtree[n].begin(), mgtree[n].end(), k) -
36             mgtree[n].begin();
37     }
38     int mid = (esq + dir) / 2;
39     return less(le(n), esq, mid, l, r, k) + less(ri(n), mid + 1, dir, l, r, k);
40 }
41 int less(int l, int r, int k) { return less(0, 0, n - 1, l, r, k); }
42
43 // vi debug_query(int n, int esq, int dir, int
44 // l, int r) {
45 // if (esq > r || dir < l) return vi();
46 // if (l <= esq && dir <= r) return
47 // mgtree[n]; int mid = (esq + dir) / 2;
48 // auto vl = debug_query(le(n), esq, mid,
49 // l, r); auto vr = debug_query(ri(n),
50 // mid+1, dir, l, r); vi ans =
51 // vi(vl.size() + vr.size());
52 // merge(vl.begin(), vl.end(),
53 // vr.begin(), vr.end(),
54 // ans.begin());
55 // return ans;
56 // }

```

```

56 // vi debug_query(int l, int r) {return
57 // debug_query(0, 0, n-1, l, r);}
58 };

```

1.6 Operation Queue

Fila que armazena o resultado do operador dos itens (ou seja, dado uma fila, responde qual é o elemento mínimo, por exemplo). É uma extensão da *std::queue*, permitindo todos os métodos já presentes nela, com a diferença de que *push* e *pop* agora são *add* e *remove*, respectivamente, ambos continuam $\mathcal{O}(1)$ amortizado. A fila agora também permite a operação *get* que retorna o resultado do operador dos itens da fila em $\mathcal{O}(1)$ amortizado. Chamar o método *get* em uma fila vazia é indefinido.

Obs: usa a estrutura Operation Stack.

Código: op_queue.cpp

```

1 template <typename T, auto OP> struct op_queue : queue<T> {
2     op_stack<T, OP> st1, st2;
3     T get() {
4         if (st1.empty()) {
5             return st2.get();
6         }
7         if (st2.empty()) {
8             return st1.get();
9         }
10        return OP(st1.get(), st2.get());
11    }
12    void add(T element) {
13        this->push(element);
14        st1.add(element);
15    }
16    void remove() {
17        if (st2.empty()) {
18            while (!st1.empty()) {
19                st2.add(st1.top());
20                st1.remove();
21            }
22        }
23        st2.remove();

```

```

24     this->pop();
25 }
26 };

```

1.7 Operation Stack

Pilha que armazena o resultado do operatório dos itens (ou seja, dado uma pilha, responde qual é o elemento mínimo, por exemplo). É uma extensão da *std::stack*, permitindo todos os métodos já presentes nela, com a diferença de que *push* e *pop* agora são *add* e *remove*, respectivamente, ambos continuam $\mathcal{O}(1)$ amortizado. A pilha agora também permite a operação *get* que retorna o resultado do operatório dos itens da pilha em $\mathcal{O}(1)$ amortizado. Chamar o método *get* em uma pilha vazia é indefinido.

Código: op_stack.cpp

```

1  template <typename T, auto OP> struct op_stack : stack<T> {
2      stack<T> st;
3      T get() { return st.top(); }
4      void add(T element) {
5          this->push(element);
6          st.push(st.empty() ? element : OP(element, st.top()));
7      }
8      void remove() {
9          st.pop();
10         this->pop();
11     }
12 };

```

1.8 Ordered Set

Set com operações de busca por ordem e índice.

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

- *find_by_order(k)*: retorna um iterador para o k-ésimo menor elemento no set (indexado em 0).
- *order_of_key(k)*: retorna o número de elementos menores que k. (ou seja, o índice de k no set)

Ambas as operações são $\mathcal{O}(\log(n))$.

Também é possível criar um *ordered_map*, funciona como um *std::map*, mas com as operações de busca por ordem e índice. *find_by_order(k)* retorna um iterador para a k-ésima menor **key** no mapa (indexado em 0). *order_of_key(k)* retorna o número de **keys** no mapa menores que k. (ou seja, o índice de k no map).

Para simular um *std::multiset*, há várias formas:

- Usar um *std::pair* como elemento do set, com o primeiro elemento sendo o valor e o segundo sendo um identificador único para cada elemento. Para saber o número de elementos menores que *k* no multiset, basta usar *order_of_key(k, -INF)*.
- Usar um *ordered_map* com a key sendo o valor e o value sendo o número de ocorrências do valor no multiset. Para saber o número de elementos menores que *k* no multiset, basta usar *order_of_key(k)*.
- Criar o *ordered_set* substituindo o parâmetro *less < T >* por *less_equal < T >*. Isso faz com que o set aceite elementos repetidos, e *order_of_key(k)* retorna o número de elementos menores ou iguais a *k* no multiset. Porém esse método não é recomendado pois gera algumas inconsistências, como por exemplo: *upper_bound* funciona como *lower_bound* e vice-versa, *find* sempre retorna *end()* e *erase* por valor não funciona, só por iterador. Dá pra usar se souber o que está fazendo.

Exemplo de uso do *ordered_set*:

```

1  ordered_set<int> X;
2  X.insert(1);
3  X.insert(2);
4  X.insert(4);
5  X.insert(8);
6  X.insert(16);
7  cout << *X.find_by_order(1) << endl; // 2
8  cout << *X.find_by_order(2) << endl; // 4

```



```

9 cout << *X.find_by_order(4) << endl; // 16
10 cout << (end(X) == X.find_by_order(5)) << endl; // true
11 cout << X.order_of_key(-5) << endl; // 0
12 cout << X.order_of_key(1) << endl; // 0
13 cout << X.order_of_key(3) << endl; // 2
14 cout << X.order_of_key(4) << endl; // 2
15 cout << X.order_of_key(400) << endl; // 5

```

Exemplo de uso do *ordered_map*:

```

1 ordered_map<int, int> Y;
2 Y[1] = 10;
3 Y[2] = 20;
4 Y[4] = 40;
5 Y[8] = 80;
6 Y[16] = 160;
7 cout << Y.find_by_order(1)->first << endl; // 2
8 cout << Y.find_by_order(1)->second << endl; // 20
9 cout << Y.order_of_key(5) << endl; // 3
10 cout << Y.order_of_key(10) << endl; // 4
11 cout << Y.order_of_key(4) << endl; // 2

```

Código: *ordered_set.cpp*

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3
4 using namespace __gnu_pbds;
5
6 template <typename T>
7 using ordered_set =
8     tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
9
10 template <typename T, typename U>
11 using ordered_map = tree<T, U, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

```

1.9 Segment Tree

1.9.1 Segment Tree

Implementação padrão de Seg Tree

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

Código: *seg_tree.cpp*

```

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) {
24             return 0;
25         }
26         if (l <= esq && dir <= r) {

```

```

27     return tree[n];
28 }
29 int mid = (esq + dir) / 2;
30 return merge(query(le(n), esq, mid, l, r), query(ri(n), mid + 1, dir, l, r));
31 }
32 ll query(int l, int r) { return query(0, 0, n - 1, l, r); }
33 void update(int n, int esq, int dir, int x, ll v) {
34     if (esq > x || dir < x) {
35         return;
36     }
37     if (esq == dir) {
38         tree[n] = v;
39     } else {
40         int mid = (esq + dir) / 2;
41         if (x <= mid) {
42             update(le(n), esq, mid, x, v);
43         } else {
44             update(ri(n), mid + 1, dir, x, v);
45         }
46         tree[n] = merge(tree[le(n)], tree[ri(n)]);
47     }
48 }
49 void update(int x, ll v) { update(0, 0, n - 1, x, v); }
50 }

```

1.9.2 Segment Tree 2D

Segment Tree em 2 dimensões.

- Complexidade de tempo (Pré-processamento): $\mathcal{O}(N \cdot M)$
- Complexidade de tempo (Consulta em intervalo): $\mathcal{O}(\log N \cdot \log M)$
- Complexidade de tempo (Update em ponto): $\mathcal{O}(\log N \cdot \log M)$
- Complexidade de espaço: $4 * N \cdot 4 * M = \mathcal{O}(N \cdot M)$

Codigo: seg_tree_2d.cpp

```

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     if (ly == ry) {
34         if (lx == rx) {
35             tree[nx][ny] = v;
36         } else {
37             tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
38         }
39     } else {
40         int my = (ly + ry) / 2;
41         if (y <= my) {
42             update_y(nx, lx, rx, le(ny), ly, my, x, y, v);
43         } else {
44             update_y(nx, lx, rx, ri(ny), my + 1, ry, x, y, v);
45         }
46         tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
47     }

```

```

48 }
49 void update_x(int nx, int lx, int rx, int x, int y, int v) {
50     if (lx != rx) {
51         int mx = (lx + rx) / 2;
52         if (x <= mx) {
53             update_x(le(nx), lx, mx, x, y, v);
54         } else {
55             update_x(ri(nx), mx + 1, rx, x, y, v);
56         }
57     }
58     update_y(nx, lx, rx, 0, 0, m - 1, x, y, v);
59 }
60 void update(int x, int y, int v) { update_x(0, 0, n - 1, x, y, v); }
61
62 int sum_y(int nx, int ny, int ly, int ry, int qly, int qry) {
63     if (ry < qly || ly > qry) {
64         return 0;
65     }
66     if (qly <= ly && ry <= qry) {
67         return tree[nx][ny];
68     }
69     int my = (ly + ry) / 2;
70     return sum_y(nx, le(ny), ly, my, qly, qry) + sum_y(nx, ri(ny), my + 1, ry, qly,
71     qry);
72 }
73 int sum_x(int nx, int lx, int rx, int qlx, int qrx, int qly, int qry) {
74     if (rx < qlx || lx > qrx) {
75         return 0;
76     }
77     if (qlx <= lx && rx <= qrx) {
78         return sum_y(nx, 0, 0, m - 1, qly, qry);
79     }
80     int mx = (lx + rx) / 2;
81     return sum_x(le(nx), lx, mx, qlx, qrx, qly, qry) +
82     sum_x(ri(nx), mx + 1, rx, qlx, qrx, qly, qry);
83 }
84 int sum(int lx, int rx, int ly, int ry) { return sum_x(0, 0, n - 1, lx, rx, ly, ry); }

```

1.9.3 Segment 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): $\mathcal{O}(N)$
- Complexidade de tempo (Consulta em intervalo): $\mathcal{O}(\log(N))$
- Complexidade de tempo (Update em ponto): $\mathcal{O}(\log(N))$
- Complexidade de tempo (Update em intervalo): $\mathcal{O}(\log(N))$
- Complexidade de espaço: $2 * 4 * N = \mathcal{O}(N)$

Codigo: seg_tree_beats_max_and_sum_update.cpp

```

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) {
26             m2 = min(m2, max(a.m1, b.m1));
27         }
28         if (a.m2 != m1) {
29             m2 = min(m2, a.m2);
30         }
31         if (b.m2 != m1) {
32             m2 = min(m2, b.m2);
33         }

```

```

34     cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
35     soma = a.soma + b.soma;
36 }
37
38 void print() { printf("%d %d %d %lld\n", m1, m2, cont, soma); }
39 };
40
41 int n, q;
42 vector<Node> tree;
43
44 int le(int n) { return 2 * n + 1; }
45 int ri(int n) { return 2 * n + 2; }
46
47 void push(int n, int esq, int dir) {
48     while (!tree[n].lazy.empty()) {
49         ii p = tree[n].lazy.front();
50         tree[n].lazy.pop();
51         int op = p.fi, v = p.se;
52         if (op == 0) {
53             if (v <= tree[n].m1) {
54                 continue;
55             }
56             tree[n].soma += (ll)abs(tree[n].m1 - v) * tree[n].cont;
57             tree[n].m1 = v;
58             if (esq != dir) {
59                 tree[le(n)].lazy.push({0, v});
60                 tree[ri(n)].lazy.push({0, v});
61             }
62         } else if (op == 1) {
63             tree[n].soma += v * (dir - esq + 1);
64             tree[n].m1 += v;
65             tree[n].m2 += v;
66             if (esq != dir) {
67                 tree[le(n)].lazy.push({1, v});
68                 tree[ri(n)].lazy.push({1, v});
69             }
70         }
71     }
72 }
73
74 void build(int n, int esq, int dir, vector<int> &v) {
75     if (esq == dir) {
76         tree[n].set(v[esq]);
77     } else {
78         int mid = (esq + dir) / 2;
79         build(le(n), esq, mid, v);
80         build(ri(n), mid + 1, dir, v);

```

```

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

```

```

128 }
129 void upsoma(int l, int r, int v) { upsoma(0, 0, n - 1, l, r, v); }
130
131 // soma de [l, r]
132 int query(int n, int esq, int dir, int l, int r) {
133     push(n, esq, dir);
134     if (esq > r || dir < l) {
135         return 0;
136     }
137     if (l <= esq && dir <= r) {
138         return tree[n].soma;
139     }
140     int mid = (esq + dir) / 2;
141     return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir, l, r);
142 }
143 int query(int l, int r) { return query(0, 0, n - 1, l, r); }
144
145 int main() {
146     cin >> n;
147     tree.assign(4 * n, Node());
148     build(v);
149 }

```

1.9.4 Segment Tree Beats Max Update

Seg Tree que suporta update de maximo e query de soma

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

Codigo: seg_tree_beats.cpp

```

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) {
21             m2 = min(m2, max(a.m1, b.m1));
22         }
23         if (a.m2 != m1) {
24             m2 = min(m2, a.m2);
25         }
26         if (b.m2 != m1) {
27             m2 = min(m2, b.m2);
28         }
29         cont = (a.m1 == m1 ? a.cont : 0) + (b.m1 == m1 ? b.cont : 0);
30         soma = a.soma + b.soma;
31     }
32
33     void print() { printf("%d %d %d %lld %d\n", m1, m2, cont, soma, lazy); }
34 };
35
36 int n, q;
37 vector<Node> tree;
38
39 int le(int n) { return 2 * n + 1; }
40 int ri(int n) { return 2 * n + 2; }
41
42 void push(int n, int esq, int dir) {
43     if (tree[n].lazy <= tree[n].m1) {
44         return;
45     }
46     tree[n].soma += (ll)abs(tree[n].m1 - tree[n].lazy) * tree[n].cont;
47     tree[n].m1 = tree[n].lazy;

```

```

48     if (esq != dir) {
49         tree[le(n)].lazy = max(tree[le(n)].lazy, tree[n].lazy);
50         tree[ri(n)].lazy = max(tree[ri(n)].lazy, tree[n].lazy);
51     }
52     tree[n].lazy = 0;
53 }
54
55 void build(int n, int esq, int dir, vector<int> &v) {
56     if (esq == dir) {
57         tree[n].set(v[esq]);
58     } else {
59         int mid = (esq + dir) / 2;
60         build(le(n), esq, mid, v);
61         build(ri(n), mid + 1, dir, v);
62         tree[n].merge(tree[le(n)], tree[ri(n)]);
63     }
64 }
65 void build(vector<int> &v) { build(0, 0, n - 1, v); }
66
67 // ai = max(ai, mi) em [l, r]
68 void update(int n, int esq, int dir, int l, int r, int mi) {
69     push(n, esq, dir);
70     if (esq > r || dir < l || mi <= tree[n].m1) {
71         return;
72     }
73     if (l <= esq && dir <= r && mi < tree[n].m2) {
74         tree[n].lazy = mi;
75         push(n, esq, dir);
76     } else {
77         int mid = (esq + dir) / 2;
78         update(le(n), esq, mid, l, r, mi);
79         update(ri(n), mid + 1, dir, l, r, mi);
80         tree[n].merge(tree[le(n)], tree[ri(n)]);
81     }
82 }
83 void update(int l, int r, int mi) { update(0, 0, n - 1, l, r, mi); }
84
85 // soma de [l, r]
86 int query(int n, int esq, int dir, int l, int r) {
87     push(n, esq, dir);
88     if (esq > r || dir < l) {
89         return 0;
90     }
91     if (l <= esq && dir <= r) {
92         return tree[n].soma;
93     }
94     int mid = (esq + dir) / 2;

```

```

95     return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir, l, r);
96 }
97 int query(int l, int r) { return query(0, 0, n - 1, l, r); }
98
99 int main() {
100     cin >> n;
101     tree.assign(4 * n, Node());
102 }

```

1.9.5 Segment Tree Esparsa

Seg Tree Esparsa, ou seja, uma seg tree que não guarda todos os nós, mas apenas os nós que são necessários para responder as queries, permitindo fazer queries em intervalos de tamanho arbitrário.

Seja LEN o tamanho do intervalo em que a Seg Tree foi construída:

- Complexidade de tempo (Pré-processamento): $\mathcal{O}(1)$
- Complexidade de tempo (Consulta em intervalo): $\mathcal{O}(\log(LEN))$
- Complexidade de tempo (Update em ponto): $\mathcal{O}(\log(LEN))$

Código: seg_tree_sparse.cpp

```

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],
7          ptr = 2; // 0 is NULL; 1 is First Root
8      ll op(ll a, ll b) { return (a + b) % MOD; }
9      int le(int i) {
10         if (L[i] == 0) {
11             L[i] = ptr++;
12         }
13         return L[i];
14     }
15     int ri(int i) {

```

```

16     if (R[i] == 0) {
17         R[i] = ptr++;
18     }
19     return R[i];
20 }
21 ll query(ll l, ll r, int n = 1, ll esq = ESQ, ll dir = DIR) {
22     if (r < esq || dir < 1) {
23         return 0;
24     }
25     if (l <= esq && dir <= r) {
26         return tree[n];
27     }
28     ll mid = (esq + dir) / 2;
29     return op(query(l, r, le(n), esq, mid), query(l, r, ri(n), mid + 1, dir));
30 }
31 void update(ll x, ll v, int n = 1, ll esq = ESQ, ll dir = DIR) {
32     if (esq == dir) {
33         tree[n] = (tree[n] + v) % MOD;
34     } else {
35         ll mid = (esq + dir) / 2;
36         if (x <= mid) {
37             update(x, v, le(n), esq, mid);
38         } else {
39             update(x, v, ri(n), mid + 1, dir);
40         }
41         tree[n] = op(tree[le(n)], tree[ri(n)]);
42     }
43 }
44 };

```

1.9.6 Segment Tree Kadani

Implementação de uma Seg Tree que suporta update de soma e query de soma máxima em intervalo.

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

Codigo: seg_tree_kadani.cpp

```

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 < 1) {
36             return NEUTRAL;
37         }
38         if (l <= esq && dir <= r) {
39             return tree[n];
40         }
41         int mid = (esq + dir) / 2;
42         return merge(query(le(n), esq, mid, l, r), query(ri(n), mid + 1, dir, l, r));
43     }
44     ll query(int l, int r) { return query(0, 0, n - 1, l, r).best; }

```

```

45 void update(int n, int esq, int dir, int x, ll v) {
46     if (esq > x || dir < x) {
47         return;
48     }
49     if (esq == dir) {
50         tree[n] = new_node(v);
51     } else {
52         int mid = (esq + dir) / 2;
53         if (x <= mid) {
54             update(le(n), esq, mid, x, v);
55         } else {
56             update(ri(n), mid + 1, dir, x, v);
57         }
58         tree[n] = merge(tree[le(n)], tree[ri(n)]);
59     }
60 }
61 void update(int x, ll v) { update(0, 0, n - 1, x, v); }
62 }

```

1.9.7 Segment Tree Lazy

Implementação padrão de Seg Tree com lazy update

- Complexidade de tempo (Pré-processamento): $\mathcal{O}(N)$
- Complexidade de tempo (Consulta em intervalo): $\mathcal{O}(\log(N))$
- Complexidade de tempo (Update em intervalo): $\mathcal{O}(\log(N))$
- Complexidade de espaço: $2 * 4 * N = \mathcal{O}(N)$

Codigo: seg_tree_lazy.cpp

```

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) {
11             return;
12         }
13         tree[n] += lazy[n] * (dir - esq + 1);
14         if (esq != dir) {
15             lazy[le(n)] += lazy[n];
16             lazy[ri(n)] += lazy[n];
17         }
18         lazy[n] = 0;
19     }
20     void build(span<const ll> v, int n, int esq, int dir) {
21         if (esq == dir) {
22             tree[n] = v[esq];
23         } else {
24             int mid = (esq + dir) / 2;
25             build(v, le(n), esq, mid);
26             build(v, ri(n), mid + 1, dir);
27             tree[n] = merge(tree[le(n)], tree[ri(n)]);
28         }
29     }
30     void build(span<const ll> v) {
31         sz = v.size();
32         build(v, 0, 0, sz - 1);
33     }
34     ll query(int l, int r, int n = 0, int esq = 0, int dir = sz - 1) {
35         push(n, esq, dir);
36         if (esq > r || dir < l) {
37             return NEUTRAL;
38         }
39         if (l <= esq && dir <= r) {
40             return tree[n];
41         }
42         int mid = (esq + dir) / 2;
43         return merge(query(l, r, le(n), esq, mid), query(l, r, ri(n), mid + 1, dir));
44     }
45     void update(int l, int r, ll v, int n = 0, int esq = 0, int dir = sz - 1) {
46         push(n, esq, dir);
47         if (esq > r || dir < l) {
48             return;
49         }
50         if (l <= esq && dir <= r) {
51             lazy[n] += v;
52             push(n, esq, dir);
53         } else {
54             int mid = (esq + dir) / 2;

```



```

55         update(l, r, v, le(n), esq, mid);
56         update(l, r, v, ri(n), mid + 1, dir);
57         tree[n] = merge(tree[le(n)], tree[ri(n)]);
58     }
59 }
60 }

```

1.9.8 Segment Tree Persistente

Seg Tree Esparsa com histórico de Updates:

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

Código: seg_tree_persistent.cpp

```

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) {
11                 l = new node();
12             }
13             if (r == NULL) {
14                 r = new node();
15             }
16         }
17     };

```

```

18     vector<node*> roots;
19     void build() { roots.push_back(new node()); }
20     void push(node *n, int esq, int dir) {
21         if (esq != dir) {
22             n->apply();
23         }
24     }
25     // sum v on x
26     node *update(node *n, int esq, int dir, int x, int v) {
27         push(n, esq, dir);
28         if (esq == dir) {
29             return new node(n->v + v);
30         }
31         int mid = (esq + dir) / 2;
32         if (x <= mid) {
33             return new node(update(n->l, esq, mid, x, v), n->r);
34         } else {
35             return new node(n->l, update(n->r, mid + 1, dir, x, v));
36         }
37     }
38     int update(int root, int pos, int val) {
39         node *novo = update(roots[root], ESQ, DIR, pos, val);
40         roots.push_back(novo);
41         return roots.size() - 1;
42     }
43     // sum in [L, R]
44     ll query(node *n, int esq, int dir, int l, int r) {
45         push(n, esq, dir);
46         if (esq > r || dir < l) {
47             return 0;
48         }
49         if (l <= esq && dir <= r) {
50             return n->v;
51         }
52         int mid = (esq + dir) / 2;
53         return query(n->l, esq, mid, l, r) + query(n->r, mid + 1, dir, l, r);
54     }
55     ll query(int root, int l, int r) { return query(roots[root], ESQ, DIR, l, r); }
56     // kth min number in [L, R] (l_root can not be
57     // 0)
58     int kth(node *L, node *R, int esq, int dir, int k) {
59         push(L, esq, dir);
60         push(R, esq, dir);
61         if (esq == dir) {
62             return esq;
63         }
64         int mid = (esq + dir) / 2;

```

```

65     int cont = R->l->v - L->l->v;
66     if (cont >= k) {
67         return kth(L->l, R->l, esq, mid, k);
68     } else {
69         return kth(L->r, R->r, mid + 1, dir, k - cont);
70     }
71 }
72 int kth(int l_root, int r_root, int k) {
73     return kth(roots[l_root - 1], roots[r_root], ESQ, DIR, k);
74 }
75 };

```

1.10 Sparse Table

1.10.1 Disjoint Sparse Table

Resolve query de range para qualquer operação associativa em $\mathcal{O}(1)$.

Pré-processamento em $\mathcal{O}(n \log n)$.

Código: dst.cpp

```

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++) {
11             t[0][i] = i < sz ? v[i] : neutral;
12         }
13         for (int j = 0, len = 1; j <= k; j++, len <= 1) {
14             for (int s = len; s < n; s += (len < 1)) {
15                 t[j][s] = v[s];
16                 t[j][s - 1] = v[s - 1];
17                 for (int i = 1; i < len; i++) {
18                     t[j][s + i] = comp(t[j][s + i - 1], v[s + i]);
19                     t[j][s - 1 - i] = comp(v[s - 1 - i], t[j][s - i]);

```

```

20         }
21     }
22 }
23 }
24 int query(int l, int r) {
25     if (l == r) {
26         return t[0][r];
27     }
28     int i = 31 - __builtin_clz(1 ^ r);
29     return comp(t[i][l], t[i][r]);
30 }
31 };

```

1.10.2 Sparse Table

Read in [English](README.en.md)

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): $\mathcal{O}(N * \log(N))$
- Complexidade de tempo (Consulta para operações sem sobreposição amigável): $\mathcal{O}(N * \log(N))$
- Complexidade de tempo (Consulta para operações com sobreposição amigável): $\mathcal{O}(1)$
- Complexidade de espaço: $\mathcal{O}(N * \log(N))$

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

Código: sparse_table.cpp

```

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++) {
7         st[0][i] = v[i];
8     }
9     for (int i = 1; i <= e; i++) {
10        for (int j = 0; j + (1 << i) <= n; j++) {
11            st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
12        }
13    }
14 }
15 // O(log(N)) Query for non overlap friendly
16 // operations
17 int logquery(int l, int r) {
18     int res = 2e9;
19     for (int i = e; i >= 0; i--) {
20         if ((1 << i) <= r - l + 1) {
21             res = min(res, st[i][l]);
22             l += 1 << i;
23         }
24     }
25     return res;
26 }
27 // O(1) Query for overlap friendly operations
28 // ex: max(), min(), gcd(), f(x, y) = x
29 int query(int l, int r) {
30     // if (l > r) return 2e9;
31     int i = ilogb(r - l + 1);
32     return min(st[i][l], st[i][r - (1 << i) + 1]);
33 }
34 };
```

Capítulo 2

Grafos

2.1 2 SAT

Resolve problema do 2-SAT.

- Complexidade de tempo (caso médio): $\mathcal{O}(N + M)$

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.

Código: 2_sat.cpp

```
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]) {
24                 dfs1(u);
25             }
26         }
27         order.push_back(v);
28     }
29     void dfs2(int v, int c1) {
30         comp[v] = c1;
31         for (int u : gt[v]) {
32             if (comp[u] == -1) {
33                 dfs2(u, c1);
34             }
35         }
36     }
37     bool solve() {
38         order.clear();
39         used.assign(n, false);
40         for (int i = 0; i < n; ++i) {
41             if (!used[i]) {
42                 dfs1(i);
43             }
44         }
```

```

45
46     comp.assign(n, -1);
47     for (int i = 0, j = 0; i < n; ++i) {
48         int v = order[n - i - 1];
49         if (comp[v] == -1) {
50             dfs2(v, j++);
51         }
52     }
53
54     assignment.assign(n / 2, false);
55     for (int i = 0; i < n; i += 2) {
56         if (comp[i] == comp[i + 1]) {
57             return false;
58         }
59         assignment[i / 2] = comp[i] > comp[i + 1];
60     }
61     return true;
62 }
63 };

```

2.2 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: $\mathcal{O}(N * \log(N))$
- Consulta do k-ésimo ancestral de u: $\mathcal{O}(\log(N))$
- LCA: $\mathcal{O}(\log(N))$

Complexidade de espaço: $\mathcal{O}(N \log(N))$

Codigo: binary_lifting_lca.cpp

```

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++) {
9             st[u][i] = st[st[u][i - 1]][i - 1];
10        }
11        for (int v : adj[u]) {
12            if (v != p) {
13                et_dfs(v, u);
14            }
15        }
16        tout[u] = ++timer;
17    }
18    void build(int _n, int root = 0) {
19        n = _n;
20        tin.assign(n, 0);
21        tout.assign(n, 0);
22        timer = 0;
23        me = floor(log2(n));
24        st.assign(n, vector<int>(me + 1, 0));
25        et_dfs(root, root);
26    }
27    bool is_ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }
28    int lca(int u, int v) {
29        if (is_ancestor(u, v)) {
30            return u;
31        }
32        if (is_ancestor(v, u)) {
33            return v;
34        }
35        for (int i = me; i >= 0; i--) {
36            if (!is_ancestor(st[u][i], v)) {
37                u = st[u][i];
38            }
39        }
40        return st[u][0];
41    }
42    int ancestor(int u,
43                int k) { // k-th ancestor of u
44        for (int i = me; i >= 0; i--) {
45            if ((1 << i) & k) {
46                u = st[u][i];
47            }
48        }
49        return u;

```

```
50 }
51 }
```

Codigo: binary_lifting.cpp

```
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++) {
7             st[u][i] = st[st[u][i - 1]][i - 1];
8         }
9         for (int v : adj[u]) {
10             if (v != p) {
11                 bl_dfs(v, u);
12             }
13         }
14     }
15     void build(int _n, int root = 0) {
16         n = _n;
17         me = floor(log2(n));
18         st.assign(n, vector<int>(me + 1, 0));
19         bl_dfs(root, root);
20     }
21     int ancestor(int u,
22                 int k) { // k-th ancestor of u
23         for (int i = me; i >= 0; i--) {
24             if ((1 << i) & k) {
25                 u = st[u][i];
26             }
27         }
28         return u;
29     }
30 }
```

2.3 Bridge

Algoritmo que acha pontes utilizando uma dfs

Complexidade de tempo: $\mathcal{O}(N + M)$

Codigo: find_bridges.cpp

```
1 int n; // number of nodes
2 vector<vector<int>> adj; // adjacency list of graph
3
4 vector<bool> visited;
5 vector<int> tin, low;
6 int timer;
7
8 void dfs(int u, int p = -1) {
9     visited[u] = true;
10    tin[u] = low[u] = timer++;
11    for (int v : adj[u]) {
12        if (v == p) {
13            continue;
14        }
15        if (visited[v]) {
16            low[u] = min(low[u], tin[v]);
17        } else {
18            dfs(v, u);
19            low[u] = min(low[u], low[v]);
20            if (low[v] > tin[u]) {
21                // edge UV is a bridge
22                // do_something(u, v)
23            }
24        }
25    }
26 }
27
28 void find_bridges() {
29     timer = 0;
30     visited.assign(n, false);
31     tin.assign(n, -1);
32     low.assign(n, -1);
33     for (int i = 0; i < n; ++i) {
34         if (!visited[i]) {
35             dfs(i);
36         }
37     }
38 }
```

2.4 Fluxo

Conjunto de algoritmos para calcular o fluxo máximo em redes de fluxo.

Muito útil para grafos bipartidos e para grafos com muitas arestas

Complexidade de tempo: $\mathcal{O}(V * E)$, mas em grafo bipartido a complexidade é $\mathcal{O}(\sqrt{V} * E)$

Útil para grafos com poucas arestas

Complexidade de tempo: $\mathcal{O}(V * E)$

Computa o fluxo máximo com custo mínimo

Complexidade de tempo: $\mathcal{O}(V * E)$

Codigo: EdmondsKarp.cpp

```

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) {
36                 break;
37             }
38             fila.pop();
39             visto[u] = vistoken;
40             for (int id : adj[u]) {
41                 if (edges[id].cap - edges[id].flow < 1) {
42                     continue;
43                 }
44                 int v = edges[id].v;
45                 if (visto[v] == -1) {
46                     continue;
47                 }
48                 fila.push(v);
49                 pego[v] = id;
50             }
51         }
52         if (pego[t] == -1) {
53             return 0;
54         }
55         long long f = INF;
56         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
57             f = min(f, edges[id].cap - edges[id].flow);
58         }
59         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
60             edges[id].flow += f;
61             edges[id ^ 1].flow -= f;
62         }
63         return f;
64     }
65
66     long long flow() {
67         long long maxflow = 0;
68         while (long long f = bfs()) {
69             maxflow += f;
70         }
71         return maxflow;
72     }

```

73 };

Codigo: MinCostMaxFlow.cpp

```

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) {
31                     continue;
32                 }
33                 int v = edges[id].v;
34                 if (dis[v] > dis[u] + edges[id].cost) {
35                     dis[v] = dis[u] + edges[id].cost;
36                     pego[v] = id;
37                     if (!inq[v]) {
38                         inq[v] = true;
39                         fila.push(v);
40                     }
41                 }
42             }
43         }

```

```

44
45         if (pego[t] == -1) {
46             return 0;
47         }
48         ll f = INF;
49         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
50             f = min(f, edges[id].cap - edges[id].flow);
51             mincost += edges[id].cost;
52         }
53         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
54             edges[id].flow += f;
55             edges[id ^ 1].flow -= f;
56         }
57         maxflow += f;
58         return 1;
59     }
60
61     ll flow() {
62         while (spfa())
63             ;
64         return maxflow;
65     }
66 };

```

Codigo: Dinic.cpp

```

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) {
38                     continue;
39                 }
40                 int v = edges[id].v;
41                 if (level[v] != -1) {
42                     continue;
43                 }
44                 level[v] = level[u] + 1;
45                 q.push(v);
46             }
47         }
48         return level[t] != -1;
49     }
50
51     ll dfs(int u, ll f) {
52         if (f == 0) {
53             return 0;
54         }
55         if (u == t) {
56             return f;
57         }
58         for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
59             int id = adj[u][cid];
60             int v = edges[id].v;
61             if (level[u] + 1 != level[v] || edges[id].cap - edges[id].flow < 1) {
62                 continue;
63             }
64             ll tr = dfs(v, min(f, edges[id].cap - edges[id].flow));
65             if (tr == 0) {
66                 continue;
67             }
68             edges[id].flow += tr;

```

```

69             edges[id ^ 1].flow -= tr;
70             return tr;
71         }
72         return 0;
73     }
74
75     ll flow() {
76         ll maxflow = 0;
77         while (true) {
78             fill(level.begin(), level.end(), -1);
79             level[s] = 0;
80             q.push(s);
81             if (!bfs()) {
82                 break;
83             }
84             fill(ptr.begin(), ptr.end(), 0);
85             while (ll f = dfs(s, INF)) {
86                 maxflow += f;
87             }
88         }
89         return maxflow;
90     }
91 };

```

2.5 Graph Center

Encontra o centro e o diâmetro de um grafo

Complexidade de tempo: $\mathcal{O}(N)$

Codigo: graph_center.cpp

```

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) {
19             maxidist = dist[u], maxinode = u;
20         }
21         for (int v : adj[u]) {
22             if (dist[u] + 1 < dist[v]) {
23                 dist[v] = dist[u] + 1;
24                 pai[v] = u;
25                 q.push(v);
26             }
27         }
28     }
29     diam = max(diam, maxidist);
30     return maxinode;
31 }
32 GraphCenter(int st = 0) : n(adj.size()) {
33     int d1 = bfs(st);
34     int d2 = bfs(d1);
35     vector<int> path;
36     for (int u = d2; u != -1; u = pai[u]) {
37         path.push_back(u);
38     }
39     int len = path.size();
40     if (len % 2 == 1) {
41         centros.push_back(path[len / 2]);
42     } else {
43         centros.push_back(path[len / 2]);
44         centros.push_back(path[len / 2 - 1]);
45     }
46 }
47 };

```

2.6 HLD

Técnica usada para otimizar a execução de operações em árvores.

- Pré-Processamento: $\mathcal{O}(N)$

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

Codigo: HLD.cpp

```

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) {
6         return max(a, b); // how to merge paths
7     }
8     void dfs_sz(int u, int p = -1) {
9         sz[u] = 1;
10        for (int &v : adj[u]) {
11            if (v != p) {
12                dfs_sz(v, u);
13                sz[u] += sz[v];
14                if (sz[v] > sz[adj[u][0]] || adj[u][0] == p) {
15                    swap(v, adj[u][0]);
16                }
17            }
18        }
19    }
20    void dfs_hld(int u, int p = -1) {
21        pos[u] = t++;
22        for (int v : adj[u]) {
23            if (v != p) {
24                pai[v] = u;
25                head[v] = (v == adj[u][0] ? head[u] : v);
26                dfs_hld(v, u);
27            }
28        }
29    }
30    void build(int root) {
31        dfs_sz(root);
32        t = 0;
33        pai[root] = root;

```

```

34     head[root] = root;
35     dfs_hld(root);
36 }
37 void build(int root, vector<ll> &v) {
38     build(root);
39     vector<ll> aux(v.size());
40     for (int i = 0; i < (int)v.size(); i++) {
41         aux[pos[i]] = v[i];
42     }
43     seg::build(aux);
44 }
45 void build(int root,
46             vector<i3> &edges) { // use this if
47                                 // weighted edges
48     build(root);
49     e = 1;
50     vector<ll> aux(edges.size() + 1);
51     for (auto [u, v, w] : edges) {
52         if (pos[u] > pos[v]) {
53             swap(u, v);
54         }
55         aux[pos[v]] = w;
56     }
57     seg::build(aux);
58 }
59 ll query(int u, int v) {
60     if (pos[u] > pos[v]) {
61         swap(u, v);
62     }
63     if (head[u] == head[v]) {
64         return seg::query(pos[u] + e, pos[v]);
65     } else {
66         ll qv = seg::query(pos[head[v]], pos[v]);
67         ll qu = query(u, pai[head[v]]);
68         return merge(qu, qv);
69     }
70 }
71 void update(int u, int v, ll k) {
72     if (pos[u] > pos[v]) {
73         swap(u, v);
74     }
75     if (head[u] == head[v]) {
76         seg::update(pos[u] + e, pos[v], k);
77     } else {
78         seg::update(pos[head[v]], pos[v], k);
79         update(u, pai[head[v]], k);
80 }

```

```

81     }
82     int lca(int u, int v) {
83         if (pos[u] > pos[v]) {
84             swap(u, v);
85         }
86         return (head[u] == head[v] ? u : lca(u, pai[head[v]]));
87     }
88     ll query_subtree(int u) { return seg::query(pos[u], pos[u] + sz[u] - 1); }
89 }

```

2.7 Inverse Graph

Algoritmo que encontra as componentes conexas quando se é dado o grafo complemento.

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

- Complexidade de tempo: $\mathcal{O}(N \log N + N \log M)$

Codigo: inverse_graph.cpp

```

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) {
17                 aux.insert(y);
18             }
19         }

```

```

20     for (int y : aux) {
21         f.push(y);
22         nodes.erase(y);
23     }
24     aux.clear();
25 }
26 }

```

2.8 Kruskal

Algoritmo para encontrar a MST (minimum spanning tree) de um grafo.

Utiliza DSU para construir MST.

- Complexidade de tempo (Construção): $\mathcal{O}(M \log N)$

Código: kruskal.cpp

```

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++) {
14             pa.push_back(i);
15         }
16     }
17     int root(int a) { return pa[a] = (a == pa[a] ? a : root(pa[a])); }
18     bool find(int a, int b) { return root(a) == root(b); }
19     void uni(int a, int b) {
20         int ra = root(a), rb = root(b);
21         if (ra == rb) {
22             return;

```

```

23     }
24     if (sz[ra] > sz[rb]) {
25         swap(ra, rb);
26     }
27     pa[ra] = rb;
28     sz[rb] += sz[ra];
29 }
30 };
31
32 void kruskal(int m, int n) {
33     DSU dsu(n);
34
35     sort(edges.begin(), edges.end());
36
37     for (Edge e : edges) {
38         if (!dsu.find(e.u, e.v)) {
39             cost += e.w;
40             result.push_back(e); // remove if need only cost
41             dsu.uni(e.u, e.v);
42         }
43     }
44 }

```

2.9 LCA

Algoritmo de Lowest Common Ancestor usando EulerTour e Sparse Table

Complexidade de tempo:

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

Complexidade de espaço: $\mathcal{O}(N \log(N))$

Código: lca.cpp

```

1 #include <bits/stdc++.h>
2 using namespace std;
3

```

```

4 #define INF 1e9
5 #define fi first
6 #define se second
7
8 typedef pair<int, int> ii;
9
10 vector<int> tin, tout;
11 vector<vector<int>> adj;
12 vector<ii> prof;
13 vector<vector<ii>> st;
14
15 int n, timer;
16
17 void SparseTable(vector<ii> &v) {
18     int n = v.size();
19     int e = floor(log2(n));
20     st.assign(e + 1, vector<ii>(n));
21     for (int i = 0; i < n; i++) {
22         st[0][i] = v[i];
23     }
24     for (int i = 1; i <= e; i++) {
25         for (int j = 0; j + (1 << i) <= n; j++) {
26             st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
27         }
28     }
29 }
30
31 void et_dfs(int u, int p, int h) {
32     tin[u] = timer++;
33     prof.emplace_back(h, u);
34     for (int v : adj[u]) {
35         if (v != p) {
36             et_dfs(v, u, h + 1);
37             prof.emplace_back(h, u);
38         }
39     }
40     tout[u] = timer++;
41 }
42
43 void build(int root = 0) {
44     tin.assign(n, 0);
45     tout.assign(n, 0);
46     prof.clear();
47     timer = 0;
48     et_dfs(root, root, 0);
49     SparseTable(prof);
50 }

```

```

51
52 int lca(int u, int v) {
53     int l = tout[u], r = tin[v];
54     if (l > r) {
55         swap(l, r);
56     }
57     int i = floor(log2(r - l + 1));
58     return min(st[i][l], st[i][r - (1 << i) + 1]).se;
59 }
60
61 int main() {
62     cin >> n;
63
64     adj.assign(n, vector<int>(0));
65
66     for (int i = 0; i < n - 1; i++) {
67         int a, b;
68         cin >> a >> b;
69         adj[a].push_back(b);
70         adj[b].push_back(a);
71     }
72
73     build();
74 }

```

2.10 Matching

2.10.1 Hungaro

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: $\mathcal{O}(n^2 * m)$

Codigo: hungarian.cpp

```

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]) {
19                         minv[j] = cur, way[j] = j0;
20                     }
21                     if (minv[j] < delta) {
22                         delta = minv[j], j1 = j;
23                     }
24                 }
25             }
26             for (int j = 0; j <= m; j++) {
27                 if (used[j]) {
28                     u[p[j]] += delta, v[j] -= delta;
29                 } else {
30                     minv[j] -= delta;
31                 }
32             }
33             j0 = j1;
34         } while (p[j0] != 0);
35         do {
36             int j1 = way[j0];
37             p[j0] = p[j1];
38             j0 = j1;
39         } while (j0);
40     }
41     for (int i = 1; i <= m; i++) {
42         result.emplace_back(p[i], i);
43     }
44     return -v[0];
45 }

```

2.11 Shortest Paths

2.11.1 Dijkstra

Computa o menor caminho entre nós de um grafo.

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

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

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

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

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

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

Codigo: dijkstra_n_to_n.cpp

```

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++) {
9              dist[i][j] = INF;
10             }
11         }
12         for (int s = 0; s < n; s++) {
13             priority_queue<ii, vector<ii>, greater<ii>> fila;
14             dist[s][s] = 0;
15             fila.emplace(dist[s][s], s);
16             while (!fila.empty()) {
17                 auto [d, u] = fila.top();
18                 fila.pop();
19                 if (d != dist[s][u]) {

```

```

20         continue;
21     }
22     for (auto [w, v] : adj[u]) {
23         if (dist[s][v] > d + w) {
24             dist[s][v] = d + w;
25             fila.emplace(dist[s][v], v);
26         }
27     }
28 }
29 }
30 }

```

Codigo: dijkstra_1_to_n.cpp

```

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>, greater<ii>> fila;
8      fill(begin(dist), end(dist), INF);
9      dist[s] = 0;
10     fila.emplace(dist[s], s);
11     while (!fila.empty()) {
12         auto [d, u] = fila.top();
13         fila.pop();
14         if (d != dist[u]) {
15             continue;
16         }
17         for (auto [w, v] : adj[u]) {
18             if (dist[v] > d + w) {
19                 dist[v] = d + w;
20                 fila.emplace(dist[v], v);
21             }
22         }
23     }
24 }

```

Codigo: dijkstra_1_to_1.cpp

```

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>, greater<ii>> fila;
8      fill(begin(dist), end(dist), INF);
9      dist[s] = 0;
10     fila.emplace(dist[s], s);
11     while (!fila.empty()) {
12         auto [d, u] = fila.top();
13         fila.pop();
14         if (u == t) {
15             return dist[t];
16         }
17         if (d != dist[u]) {
18             continue;
19         }
20         for (auto [w, v] : adj[u]) {
21             if (dist[v] > d + w) {
22                 dist[v] = d + w;
23                 fila.emplace(dist[v], v);
24             }
25         }
26     }
27     return -1;
28 }

```

2.11.2 SPFA

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

Detecta ciclos negativos.

Complexidade de tempo: $\mathcal{O}(|V| * |E|)$

Codigo: spfa.cpp

```

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 }

```

2.12 Stoer–Wagner Min Cut

Algoritmo de Stoer-Wagner para encontrar o corte mínimo de um grafo.

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: $\mathcal{O}(V^3)$

Código: stoer_wagner.cpp

```

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++) {
10         v[i].assign(1, i);
11     }
12     int w[MAXN], sel;
13     bool exist[MAXN], added[MAXN];
14     memset(exist, true, sizeof(exist));
15     for (int phase = 0; phase < n - 1; phase++) {
16         memset(added, false, sizeof(added));
17         memset(w, 0, sizeof(w));
18         for (int j = 0, prev; j < n - phase; j++) {
19             sel = -1;
20             for (int i = 0; i < n; i++) {
21                 if (exist[i] && !added[i] && (sel == -1 || w[i] > w[sel])) {
22                     sel = i;
23                 }
24             }
25             if (j == n - phase - 1) {
26                 if (w[sel] < bestCost) {
27                     bestCost = w[sel];
28                     bestCut = v[sel];
29                 }
30                 v[prev].insert(v[prev].end(), v[sel].begin(), v[sel].end());
31                 for (int i = 0; i < n; i++) {
32                     adj[prev][i] = adj[i][prev] += adj[sel][i];
33                 }
34                 exist[sel] = false;
35             } else {
36                 added[sel] = true;
37                 for (int i = 0; i < n; i++) {
38                     w[i] += adj[sel][i];
39                 }
40                 prev = sel;
41             }
42         }
43     }
44     return bestCost;
45 }

```


Capítulo 3

String

3.1 Aho Corasick

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

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

Código: aho_corasick.cpp

```
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) {
51         return aho[u].exi;
52     }
53     int v = get_link(u);
54     return aho[u].exi = (v == 0 || aho[v].term ? v : exi(v));
55 }
56 void process(const string &s) {
57     int st = 0;
58     for (char c : s) {
59         st = go(st, c);
60         for (int aux = st; aux; aux = exi(aux)) {
61             aho[aux].cont++;
62         }
63     }
64     for (int st = 1; st < aho_sz; st++) {
65         if (!aho[st].term) {
66             continue;
67         }
68         for (int i : aho[st].idxs) {
69             // Do something here
70             // idx i occurs + aho[st].cont times
71             h[i] += aho[st].cont;
72         }
73     }
74 }

```

3.2 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): $\mathcal{O}(N)$

* Complexidade de tempo (Consulta de range): $\mathcal{O}(1)$

Código: hashing.cpp

```

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++) {
10             pw[i] = (pw[i - 1] * p) % m;
11         }
12     }
13     void set_string(string &s) {
14         hsh[0] = s[0];
15         for (int i = 1; i < s.size(); i++) {
16             hsh[i] = (hsh[i - 1] * p + s[i]) % m;
17         }
18     }
19     long long range(int esq, int dir) {
20         long long ans = hsh[dir];
21         if (esq > 0) {
22             ans = (ans - (hsh[esq - 1] * pw[dir - esq + 1] % m) + m) % m;
23         }
24         return ans;
25     }
26 };

```

3.3 Lyndon

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

Duval

Gera a Lyndon Factorization de uma string

* Complexidade de tempo: $\mathcal{O}(N)$

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: $\mathcal{O}(N)$

Codigo: min_cyclic_shift.cpp

```

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

Codigo: duval.cpp

```

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 }
```

3.4 Manacher

Encontra todos os palindromos de uma string.

Dada uma string s de tamanho n , encontra todos os pares (i, j) tal que a substring $s[i...j]$ seja um palindromo.

* Complexidade de tempo: $\mathcal{O}(N)$

Codigo: manacher.cpp

```

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]) {
15                k++;
16            }
17            count += d1[i] = k--;
18            if (i + k > r) {
19                l = i - k;
20                r = i + k;
21            }
22        }
23    }
24    void solve_even(string &s) {
25        d2.resize(n);
26        for (int i = 0, l = 0, r = -1; i < n; i++) {
27            int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
28            while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
```

```

29         k++;
30     }
31     count += d2[i] = k--;
32     if (i + k > r) {
33         l = i - k - 1;
34         r = i + k;
35     }
36 }
37 }
38 } mana;

```

3.5 Patricia Tree

Estrutura de dados que armazena strings e permite consultas por prefixo.

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 $\mathcal{O}(|S|)$

NÃO ACEITA ELEMENTOS REPETIDOS

Código: `patricia_tree.cpp`

```

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,
6         trie_prefix_search_node_update>
7     patricia_tree;

```

3.6 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.

KMP

String matching em $\mathcal{O}(n + m)$.

Autômato de KMP

String matching em $\mathcal{O}(n)$ com $\mathcal{O}(m)$ de pré-processamento.

Prefix Count

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

Código: `prefix_function.cpp`

```

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]) {
5             j = p[j - 1];
6         }
7         if (s[i] == s[j]) {
8             j++;
9         }
10        p[i] = j;
11    }
12    return p;
13 }

```

Codigo: KMP.cpp

```

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]) {
5             j = p[j - 1];
6         }
7         if (s[i] == s[j]) {
8             j++;
9         }
10        p[i] = j;
11    }
12    return p;
13 }
14
15 vector<int> kmp(string &s, string t) {
16     t += '$';
17     vector<int> p = pi(t), match;
18     for (int i = 0, j = 0; i < s.size(); i++) {
19         while (j > 0 && s[i] != t[j]) {
20             j = p[j - 1];
21         }
22         if (s[i] == t[j]) {
23             j++;
24         }
25         if (j == t.size() - 1) {
26             match.push_back(i - j + 1);
27         }
28     }
29     return match;
30 }

```

Codigo: prefix_count.cpp

```

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]) {
5             j = p[j - 1];
6         }
7         if (s[i] == s[j]) {
8             j++;
9         }
10        p[i] = j;
11    }
12    return p;

```

```

13 }
14
15 vector<int> prefixCount(string s) {
16     vector<int> p = pi(s + '#');
17     int n = s.size();
18     vector<int> cnt(n + 1, 0);
19     for (int i = 0; i < n; i++) {
20         cnt[p[i]]++;
21     }
22     for (int i = n - 1; i > 0; i--) {
23         cnt[p[i - 1]] += cnt[i];
24     }
25     for (int i = 0; i <= n; i++) {
26         cnt[i]++;
27     }
28     return cnt;
29 }

```

Codigo: aut_kmp.cpp

```

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]) {
8                 j = p[j - 1];
9             }
10            if (s[i] == s[j]) {
11                j++;
12            }
13            p[i] = j;
14        }
15        return p;
16    }
17
18    void setString(string s) {
19        s += '#';
20        nxt.assign(s.size(), vector<int>(26));
21        vector<int> p = pi(s);
22        for (int c = 0; c < 26; c++) {
23            nxt[0][c] = ('a' + c == s[0]);
24        }
25        for (int i = 1; i < s.size(); i++) {
26            for (int c = 0; c < 26; c++) {
27                nxt[i][c] = ('a' + c == s[i]) ? i + 1 : nxt[p[i - 1]][c];

```

```

28     }
29 }
30 }
31
32 vector<int> kmp(string &s, string &t) {
33     vector<int> match;
34     for (int i = 0, j = 0; i < s.size(); i++) {
35         j = nxt[j][s[i] - 'a'];
36         if (j == t.size()) {
37             match.push_back(i - j + 1);
38         }
39     }
40     return match;
41 }
42 } aut;

```

3.7 Suffix Array

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

Também constrói a tabela LCP (Longest Common Prefix).

* Complexidade de tempo (Pré-Processamento): $\mathcal{O}(|S| \cdot \log(|S|))$

* Complexidade de tempo (Contar ocorrências de S em T): $\mathcal{O}(|S| \cdot \log(|T|))$

Codigo: suffix_array_busca.cpp

```

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]) {
6             L = mid;
7         }
8         if (s[sa[mid] + i] < t[i]) {
9             esq = mid + 1;
10        } else {
11            dir = mid - 1;
12        }
13    }

```

```

14    esq = range.first, dir = range.second;
15    while (esq <= dir) {
16        int mid = (esq + dir) / 2;
17        if (s[sa[mid] + i] == t[i]) {
18            R = mid;
19        }
20        if (s[sa[mid] + i] <= t[i]) {
21            esq = mid + 1;
22        } else {
23            dir = mid - 1;
24        }
25    }
26    return {L, R};
27 }
28 // count ocurences of s on t
29 int busca_string(string &t) {
30     pair<int, int> range = {0, n - 1};
31     for (int i = 0; i < t.size(); i++) {
32         range = busca(t, i, range);
33         if (range.first == -1) {
34             return 0;
35         }
36     }
37     return range.second - range.first + 1;
38 }

```

Codigo: suffix_array.cpp

```

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++) {
9             c[(i + k < n) ? ra[i + k] : 0]++;
10        }
11        for (int i = sum = 0; i < max(256, n); i++) {
12            sum += c[i], c[i] = sum - c[i];
13        }
14        for (int i = 0; i < n; i++) {
15            auxsa[c[sa[i] + k < n ? ra[sa[i] + k] : 0]++] = sa[i];
16        }
17        for (int i = 0; i < n; i++) {
18            sa[i] = auxsa[i];

```

```

19     }
20 }
21 void build_sa() {
22     for (int k = 1; k < n; k <= 1) {
23         counting_sort(k);
24         counting_sort(0);
25         auxra[sa[0]] = r = 0;
26         for (int i = 1; i < n; i++) {
27             auxra[sa[i]] =
28                 (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] == ra[sa[i - 1] + k])
29                 ? r
30                 : ++r;
31         }
32         for (int i = 0; i < n; i++) {
33             ra[i] = auxra[i];
34         }
35         if (ra[sa[n - 1]] == n - 1) {
36             break;
37         }
38     }
39 }
40 void build_lcp() {
41     for (int i = 0, k = 0; i < n - 1; i++) {
42         int j = sa[ra[i] - 1];
43         while (s[i + k] == s[j + k]) {
44             k++;
45         }
46         lcp[ra[i]] = k;
47         if (k) {
48             k--;
49         }
50     }
51 }
52 void set_string(string _s) {
53     s = _s + '$';
54     n = s.size();
55     for (int i = 0; i < n; i++) {
56         ra[i] = s[i], sa[i] = i;
57     }
58     build_sa();
59     build_lcp();
60     // for (int i = 0; i < n; i++)
61     // printf("%2d: %s\n", sa[i], s.c_str() +
62     // sa[i]);
63 }
64 int operator[](int i) { return sa[i]; }
65 } sa;

```

3.8 Trie

Estrutura que guarda informações indexadas por palavra.

Útil encontrar todos os prefixos inseridos anteriormente de uma palavra específica.

* Complexidade de tempo (Update): $\mathcal{O}(|S|)$

* Complexidade de tempo (Consulta de palavra): $\mathcal{O}(|S|)$

Codigo: trie.cpp

```

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)) {
9                 trie[id][c] = ++n_nodes;
10            }
11            id = trie[id][c];
12        }
13        value[id] = v;
14    }
15    int get_value(string &s) {
16        int id = 0;
17        for (char c : s) {
18            if (!trie[id].count(c)) {
19                return -1;
20            }
21            id = trie[id][c];
22        }
23        return value[id];
24    }
25 };

```

Capítulo 4

Paradigmas

4.1 All Submasks

Percorre todas as submáscaras de uma máscara.

* Complexidade de tempo: $\mathcal{O}(3^N)$

Codigo: all_submask.cpp

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

4.2 Busca Binaria Paralela

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

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

Codigo: busca_binaria_paralela.cpp

```
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
5                                     // for muito esperso
6     long long ans[1123456]; // definir pro tamanho
7                                     // das queries
8     long long l, r, mid;
9     int id = 0;
10    void set_lim_search(long long n) {
11        l = 0;
12        r = n;
13        mid = (l + r) / 2;
14    }
15
16    void add_query(long long v) { queries[mid].push_back({v, id++, l, r}); }
17
18    void advance_search(long long v) {
19        // advance search
20    }
21
22    bool satisfies(long long mid, int v, long long l, long long r) {
23        // implement the evaluation
24    }
25
26    bool get_ans() {
27        // implement the get ans
28    }
29
30    void parallel_binary_search(long long l, long long r) {
```



```

31
32     bool go = 1;
33     while (go) {
34         go = 0;
35         int i = 0; // outra logica se for usar
36                     // um mapa
37         for (auto &vec : queries) {
38             advance_search(i++);
39             for (auto q : vec) {
40                 auto [v, id, l, r] = q;
41                 if (l > r) {
42                     continue;
43                 }
44                 go = 1;
45                 // return while satisfies
46                 if (satisfies(i, v, l, r)) {
47                     ans[i] = get_ans();
48                     long long mid = (i + l) / 2;
49                     queries[mid] = query(v, id, l, i - 1);
50                 } else {
47                     long long mid = (i + r) / 2;
49                     queries[mid] = query(v, id, i + 1, r);
50                 }
51             }
52         }
53         vec.clear();
54     }
55 }
56 }
57 }
58 }
59 }
60 } // namespace name

```

4.3 Busca Ternaria

Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (por exemplo, parábolas).

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

Busca Ternária em Espaço Discreto

Encontra um ponto ótimo em uma função que pode ser separada em duas funções estritamente monotônicas (por exemplo, parábolas).

Versão para espaços discretos.

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

Codigo: busca_ternaria.cpp

```

1
2 double eval(double mid) {
3     // implement the evaluation
4 }
5
6 double ternary_search(double l, double r) {
7     int k = 100;
8     while (k--) {
9         double step = (l + r) / 3;
10        double mid_1 = l + step;
11        double mid_2 = r - step;
12
13        // minimizing. To maximize use >= to
14        // compare
15        if (eval(mid_1) <= eval(mid_2)) {
16            r = mid_2;
17        } else {
18            l = mid_1;
19        }
20    }
21    return l;
22 }

```

Codigo: busca_ternaria_discreta.cpp

```

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
13      // compare
14      if (eval(mid) <= eval(mid + 1)) {
15          ans = mid;
16          r = mid - 1;
17      } else {
18          l = mid + 1;
19      }
20  }
21  return ans;
22 }
```

4.4 Convex Hull Trick

Otimização de DP onde se mantêm 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 sejam, usar LiChao Tree para guardar as retas.

Complexidade de tempo:

- Inserir reta: $\mathcal{O}(1)$ amortizado
- Consultar x : $\mathcal{O}(\log(N))$
- Consultar x quando x tem crescimento monotônico: $\mathcal{O}(1)$

Código: Convex Hull Trick.cpp

```

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
10                                     // only
11      }
12 };
13 deque<pair<line, ll>> fila;
14 void add_line(ll a, ll b) {
15     line nova = {a, b};
16     if (!fila.empty() && fila.back().first.a == a && fila.back().first.b == b) {
17         return;
18     }
19     while (!fila.empty() && op(fila.back().second, nova.intersect(fila.back().first)))
20     {
21         fila.pop_back();
22     }
23     ll x = fila.empty() ? -INF : nova.intersect(fila.back().first);
24     fila.emplace_back(nova, x);
25 }
26 ll get_binary_search(ll x) {
27     int esq = 0, dir = fila.size() - 1, r = -1;
28     while (esq <= dir) {
29         int mid = (esq + dir) / 2;
30         if (op(x, fila[mid].second)) {
31             esq = mid + 1;
32             r = mid;
33         } else {
34             dir = mid - 1;
35         }
36     }
37     return fila[r].first.get(x);
38 }
39 // O(1), use only when QUERIES are monotonic!
40 ll get(ll x) {
41     while (fila.size() >= 2 && op(x, fila[1].second)) {
42         fila.pop_front();
43     }
44     return fila.front().first.get(x);
45 }
```

4.5 DP de Permutacao

Otimização do problema do Caixeiro Viajante

* Complexidade de tempo: $\mathcal{O}(n^2 * 2^n)$

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

Codigo: tsp_dp.cpp

```
1 const int lim = 17; // setar para o maximo de itens
2 long double dist[lim][lim]; // eh preciso dar as
3 // distancias de n para n
4 long double dp[lim][1 << lim];
5
6 int limMask = (1 << lim) - 1; // 2**(maximo de itens) - 1
7 long double solve(int atual, int mask, int n) {
8     if (dp[atual][mask] != 0) {
9         return dp[atual][mask];
10    }
11    if (mask == (1 << n) - 1) {
12        return dp[atual][mask] = 0; // o que fazer quando
13        // chega no final
14    }
15
16    long double res = 1e13; // pode ser maior se precisar
17    for (int i = 0; i < n; i++) {
18        if (!(mask & (1 << i))) {
19            long double aux = solve(i, mask | (1 << i), n);
20            if (mask) {
21                aux += dist[atual][i];
22            }
23            res = min(res, aux);
24        }
25    }
26    return dp[atual][mask] = res;
27 }
```

4.6 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: $\mathcal{O}(n \cdot k \cdot \log(n) \cdot \mathcal{O}(\text{query}))$

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

Codigo: dc.cpp

```
1 namespace DC {
2     vi dp_before, dp_cur;
3     void compute(int l, int r, int optl, int opttr) {
4         if (l > r) {
5             return;
6         }
7         int mid = (l + r) >> 1;
8         pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
9         for (int i = optl; i <= min(mid, opttr); i++) {
10             best = max(best,
11                 {(i ? dp_before[i - 1] : 0) + query(i, mid),
12                  i}); // min() se quiser minimizar
13         }
14         dp_cur[mid] = best.first;
15         int opt = best.second;
16         compute(l, mid - 1, optl, opt);
17         compute(mid + 1, r, opt, opttr);
18     }
19
20     ll solve(int n, int k) {
21         dp_before.assign(n + 5, 0);
22         dp_cur.assign(n + 5, 0);
23         for (int i = 0; i < n; i++) {
24             dp_before[i] = query(0, i);
25         }
26         for (int i = 1; i < k; i++) {
27             compute(0, n - 1, 0, n - 1);
28             dp_before = dp_cur;
29         }
30         return dp_before[n - 1];
31     }
32 };
```

Codigo: dc_query_on_demand.cpp

```

1 namespace DC {
2     struct range { // eh preciso definir a forma
3         // de calcular o range
4         vi freq;
5         ll sum = 0;
6         int l = 0, r = -1;
7         void back_l(int v) { // Mover o 'l' do range
8             // para a esquerda
9             sum += freq[v];
10            freq[v]++;
11            l--;
12        }
13        void advance_r(int v) { // Mover o 'r' do range
14            // para a direita
15            sum += freq[v];
16            freq[v]++;
17            r++;
18        }
19        void advance_l(int v) { // Mover o 'l' do range
20            // para a direita
21            freq[v]--;
22            sum -= freq[v];
23            l++;
24        }
25        void back_r(int v) { // Mover o 'r' do range
26            // para a esquerda
27            freq[v]--;
28            sum -= freq[v];
29            r--;
30        }
31        void clear(int n) { // Limpar range
32            l = 0;
33            r = -1;
34            sum = 0;
35            freq.assign(n + 5, 0);
36        }
37    } s;
38
39    vi dp_before, dp_cur;
40    void compute(int l, int r, int optl, int opttr) {
41        if (l > r) {
42            return;
43        }
44        int mid = (l + r) >> 1;
45        pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
46

```

```

47        while (s.l < optl) {
48            s.advance_l(v[s.l]);
49        }
50        while (s.l > optl) {
51            s.back_l(v[s.l - 1]);
52        }
53        while (s.r < mid) {
54            s.advance_r(v[s.r + 1]);
55        }
56        while (s.r > mid) {
57            s.back_r(v[s.r]);
58        }
59
60        vi removed;
61        for (int i = optl; i <= min(mid, opttr); i++) {
62            best =
63                min(best,
64                    {(i ? dp_before[i - 1] : 0) + s.sum, i}); // min() se quiser
65                        minimizar
66            removed.push_back(v[s.l]);
67            s.advance_l(v[s.l]);
68        }
69        for (int rem : removed) {
70            s.back_l(v[s.l - 1]);
71        }
72        dp_cur[mid] = best.first;
73        int opt = best.second;
74        compute(l, mid - 1, optl, opt);
75        compute(mid + 1, r, opt, opttr);
76    }
77
78    ll solve(int n, int k) {
79        dp_before.assign(n, 0);
80        dp_cur.assign(n, 0);
81        s.clear(n);
82        for (int i = 0; i < n; i++) {
83            s.advance_r(v[i]);
84            dp_before[i] = s.sum;
85        }
86        for (int i = 1; i < k; i++) {
87            s.clear(n);
88            compute(0, n - 1, 0, n - 1);
89            dp_before = dp_cur;
90        }
91        return dp_before[n - 1];
92    }

```

93 };

4.7 Exponenciação de Matriz

Otimização para DP de prefixo quando o valor atual está em função dos últimos K valores já calculados.

* Complexidade de tempo: $\mathcal{O}(\log(n) * k^3)$

É preciso mapear a DP para uma exponenciação de matriz.

DP:

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

Mapeamento:

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

• –

Exemplo de DP:

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

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

Mapeamento:

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

Exemplo de DP:

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

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

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

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

Código: matrix_exp.cpp

```
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++) {
22         res[i][i] = 1;
23     }
24
25     while (exp) {
26         if (exp & 1) {
27             res = mult(res, b);
28         }
29         b = mult(b, b);
30         exp /= 2;
31     }
32     return res;
33 }
34
35 // MUDA MUITO DE ACORDO COM O PROBLEMA
36 // LEIA COMO FAZER O MAPEAMENTO NO README
37 ll solve(ll exp, ll dim) {
38     if (exp < dim) {
39         return dp[exp];
40     }
41
42     T.assign(dim, vi(dim));
43     // TO DO: Preencher a Matriz que vai ser
44     // exponenciada T[0][1] = 1; T[1][0] = 1;
45     // T[1][1] = 1;
46
47     mat prod = exp_mod(T, exp);
48
49     mat vec;
50     vec.assign(dim, vi(1));
51     for (int i = 0; i < dim; i++) {
52         vec[i][0] = dp[i]; // Valores iniciais
53     }
54
55     mat ans = mult(prod, vec);
56     return ans[0][0];
57 }

```

4.8 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): $\mathcal{O}(N \cdot \sqrt{N})$

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: $\mathcal{O}(Q \cdot N^{2/3})$

Codigo: mo.cpp

```

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) {

```

```

20     queries.push_back({l, r, (int)queries.size()});
21 }
22 inline void remove(int idx) {
23     // TODO: remove value at idx from data
24     // structure
25 }
26 inline void add(int idx) {
27     // TODO: add value at idx from data
28     // structure
29 }
30 inline int get_answer() {
31     // TODO: extract the current answer of the
32     // data structure
33     return 0;
34 }
35
36 vector<int> run() {
37     vector<int> answers(queries.size());
38     sort(queries.begin(), queries.end());
39     int L = 0;
40     int R = -1;
41     for (query q : queries) {
42         while (L > q.l) {
43             add(--L);
44         }
45         while (R < q.r) {
46             add(++R);
47         }
48         while (L < q.l) {
49             remove(L++);
50         }
51         while (R > q.r) {
52             remove(R--);
53         }
54         answers[q.idx] = get_answer();
55     }
56     return answers;
57 }
58
59 };

```

Codigo: mo_update.cpp

```

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) {
25        queries.push_back({l, r, (int)updates.size(), (int)queries.size()});
26    }
27    inline void add_update(int x, int v) { updates.push_back({x, v}); }
28    inline void remove(int idx) {
29        // TODO: remove value at idx from data
30        // structure
31    }
32    inline void add(int idx) {
33        // TODO: add value at idx from data
34        // structure
35    }
36    inline void update(int l, int r, int t) {
37        auto &[x, v] = updates[t];
38        if (l <= x && x <= r) {
39            remove(x);
40        }
41        swap(vec[x], v);
42        if (l <= x && x <= r) {
43            add(x);
44        }
45    }
46    inline int get_answer() {
47        // TODO: extract the current answer from
48        // the data structure
49        return 0;
50    }
51 }

```

```
52     vector<int> run() {
53         vector<int> answers(queries.size());
54         sort(queries.begin(), queries.end());
55         int L = 0;
56         int R = -1;
57         int T = 0;
58         for (query q : queries) {
59             while (T < q.t) {
60                 update(L, R, T++);
61             }
62             while (T > q.t) {
63                 update(L, R, --T);
64             }
65             while (L > q.l) {
66                 add(--L);
67             }
68             while (R < q.r) {
69                 add(++R);
70             }
71             while (L < q.l) {
72                 remove(L++);
73             }
74             while (R > q.r) {
75                 remove(R--);
76             }
77             answers[q.idx] = get_answer();
78         }
79         return answers;
80     }
81 };
```


Capítulo 5

Primitivas

5.1 Modular Int

O Mint é uma classe que representa um número inteiro módulo um **número primo**. Ela é útil para evitar overflow em operações de multiplicação e exponenciação, e também para facilitar a implementações.

Propriedades básicas de aritmética modular:

- $(a + b) \bmod m \equiv (a \bmod m + b \bmod m) \bmod m$
- $(a - b) \bmod m \equiv (a \bmod m - b \bmod m) \bmod m$
- $(a \cdot b) \bmod m \equiv (a \bmod m \cdot b \bmod m) \bmod m$
- $a^b \bmod m \equiv (a \bmod m)^b \bmod m$

Divisões funcionam um pouco diferente, para realizar a/b deve-se fazer $a \cdot b^{-1}$, onde b^{-1} é o **inverso multiplicativo** de b módulo m , tal que $b \cdot b^{-1} \equiv 1 \bmod m$. No código, basta usar o operador de divisão normal pois a classe já está implementada com o inverso multiplicativo.

Para usar o Mint, basta declarar o tipo e usar como se fosse um inteiro normal. Exemplo:

```
1 const int MOD = 7; // MOD = 7 para fins de exemplo
2 using mint = Mint<MOD>;
3 mint a = 4, b = 3;
4 mint c = a * b; // c == 5
```

```
5 mint d = 1 / a; // d == 2
6 mint e = a * d // e == 1
7 a += 2; // a == 6
8 a += 3; // a == 2
9 a ^= 5; // a == 4
```

Código: mint.cpp

```
1 template <int mod> struct Mint {
2     using m = Mint;
3     int val;
4     Mint() : val(0) { }
5     Mint(ll v) {
6         if (v < -mod || v >= 2 * mod)
7             v %= mod;
8         if (v >= mod)
9             v -= mod;
10        if (v < 0)
11            v += mod;
12        val = v;
13    }
14    m pwr(m b, ll e) {
15        m res = 1;
16        while (e) {
17            if (e & 1)
18                res *= b;
19            b *= b, e >>= 1;
20        }
21        return res;
22    }
```

```
22     }
23     m &operator+=(const m &o) {
24         if ((val += o.val) >= mod)
25             val -= mod;
26         return *this;
27     }
28     m &operator--(const m &o) {
29         if ((val -= o.val) < 0)
30             val += mod;
31         return *this;
32     }
33     m &operator*=(const m &o) {
34         val = (ll)val * o.val % mod;
35         return *this;
36     }
37     m &operator/=(const m &o) { return *this *= pwr(o, mod - 2); }
38     m &operator^(ll e) {
39         assert(e >= 0);
40         return *this = pwr(*this, e);
41     }
42     friend m operator+(m a, const m &b) { return a += b; }
43     friend m operator-(m a, const m &b) { return a -= b; }
44     friend m operator*(m a, const m &b) { return a *= b; }
45     friend m operator/(m a, const m &b) { return a /= b; }
46     friend m operator^(m a, ll e) { return a ^= e; }
47     friend ostream &operator<<(ostream &os, const m &a) { return os << a.val; }
48 };
49
50 const int mod = 998244353;
51 using mint = Mint<mod>;
```

Capítulo 6

Matemática

6.1 Eliminação Gaussiana

6.1.1 Gauss

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes reais.

- Complexidade de tempo: $\mathcal{O}(n^3)$

Código: gauss.cpp

```
1 const double EPS = 1e-9;
2 const int INF = 2; // it doesn't actually have to
3                   // be infinity or a big number
4
5 int gauss(vector<vector<double>> a, vector<double> &ans) {
6     int n = (int)a.size();
7     int m = (int)a[0].size() - 1;
8
9     vector<int> where(m, -1);
10    for (int col = 0, row = 0; col < m && row < n; ++col) {
11        int sel = row;
12        for (int i = row; i < n; ++i) {
13            if (abs(a[i][col]) > abs(a[sel][col])) {
14                sel = i;
15            }
16        }
```

```
16    }
17    if (abs(a[sel][col]) < EPS) {
18        continue;
19    }
20    for (int i = col; i <= m; ++i) {
21        swap(a[sel][i], a[row][i]);
22    }
23    where[col] = row;
24
25    for (int i = 0; i < n; ++i) {
26        if (i != row) {
27            double c = a[i][col] / a[row][col];
28            for (int j = col; j <= m; ++j) {
29                a[i][j] -= a[row][j] * c;
30            }
31        }
32    }
33    ++row;
34 }
35
36 ans.assign(m, 0);
37 for (int i = 0; i < m; ++i) {
38     if (where[i] != -1) {
39         ans[i] = a[where[i]][m] / a[where[i]][i];
40     }
41 }
42 for (int i = 0; i < n; ++i) {
43     double sum = 0;
44     for (int j = 0; j < m; ++j) {
45         sum += ans[j] * a[i][j];
46     }
```

```

47     if (abs(sum - a[i][m]) > EPS) {
48         return 0;
49     }
50 }
51
52 for (int i = 0; i < m; ++i) {
53     if (where[i] == -1) {
54         return INF;
55     }
56 }
57 return 1;
58 }

```

6.1.2 Gauss Mod 2

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes em \mathbb{Z}_2 (inteiros módulo 2).

- Complexidade de tempo: $\mathcal{O}(n^3/32)$

Código: gauss_mod2.cpp

```

1  const int N = 105;
2  const int INF = 2; // tanto faz
3
4  // n -> numero de equacoes, m -> numero de
5  // variaveis a[i][j] para j em [0, m - 1] ->
6  // coeficiente da variavel j na iesima equacao
7  // a[i][j] para j == m -> resultado da equacao da
8  // iesima linha ans -> bitset vazio, que retornara
9  // a solucao do sistema (caso exista)
10
11 int gauss(vector<bitset<N>> a, int n, int m, bitset<N> &ans) {
12     vector<int> where(m, -1);
13
14     for (int col = 0, row = 0; col < m && row < n; col++) {
15         for (int i = row; i < n; i++) {
16             if (a[i][col]) {
17                 swap(a[i], a[row]);
18                 break;

```

```

19         }
20     }
21     if (!a[row][col]) {
22         continue;
23     }
24     where[col] = row;
25
26     for (int i = 0; i < n; i++) {
27         if (i != row && a[i][col]) {
28             a[i] ^= a[row];
29         }
30     }
31     row++;
32 }
33
34 for (int i = 0; i < m; i++) {
35     if (where[i] != -1) {
36         ans[i] = a[where[i]][m] / a[where[i]][i];
37     }
38 }
39 for (int i = 0; i < n; i++) {
40     int sum = 0;
41     for (int j = 0; j < m; j++) {
42         sum += ans[j] * a[i][j];
43     }
44     if (abs(sum - a[i][m]) > 0) {
45         return 0; // Sem solucao
46     }
47 }
48
49 for (int i = 0; i < m; i++) {
50     if (where[i] == -1) {
51         return INF; // Infinitas solucoes
52     }
53 }
54 return 1; // Unica solucao (retornada no
55         // bitset ans)
56 }

```

6.2 Exponenciação Modular Rápida

Computa $(base^{exp}) \% mod$.

- Complexidade de tempo: $\mathcal{O}(\log(\exp))$.
- Complexidade de espaço: $\mathcal{O}(1)$

Codigo: exp_mod.cpp

```

1 ll exp_mod(ll base, ll exp) {
2     ll b = base, res = 1;
3     while (exp) {
4         if (exp & 1) {
5             res = (res * b) % MOD;
6         }
7         b = (b * b) % MOD;
8         exp /= 2;
9     }
10    return res;
11 }
```

6.3 FFT

Algoritmo que computa a transformada rápida de fourier para convolução de polinômios.

Computa convolução (multiplicação) de polinômios.

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

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

Codigo: fft.cpp

```

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) {
8         log_n++;
9     }
10
11     for (int i = 1, j = 0; i < n; ++i) {
12         int bit = n >> 1;
13         for (; j >= bit; bit >>= 1) {
14             j -= bit;
15         }
16         j += bit;
17         if (i < j) {
18             swap(a[i], a[j]);
19         }
20     }
21
22     double angle = 2 * PI / n * (invert ? -1 : 1);
23     poly root(n / 2);
24     for (int i = 0; i < n / 2; ++i) {
25         root[i] = cd(cos(angle * i), sin(angle * i));
26     }
27
28     for (long long len = 2; len <= n; len <= 1) {
29         long long step = n / len;
30         long long aux = len / 2;
31         for (long long i = 0; i < n; i += len) {
32             for (int j = 0; j < aux; ++j) {
33                 cd u = a[i + j], v = a[i + j + aux] * root[step * j];
34                 a[i + j] = u + v;
35                 a[i + j + aux] = u - v;
36             }
37         }
38     }
39     if (invert) {
40         for (int i = 0; i < n; ++i) {
41             a[i] /= n;
42         }
43     }
44 }
45
46 vector<long long> convolution(vector<long long> &a, vector<long long> &b) {
47     int n = 1, len = a.size() + b.size();
48     while (n < len) {
49         n <= 1;
50     }
51     a.resize(n);
```

```

52     b.resize(n);
53     poly fft_a(a.begin(), a.end());
54     fft(fft_a);
55     poly fft_b(b.begin(), b.end());
56     fft(fft_b);
57
58     poly c(n);
59     for (int i = 0; i < n; ++i) {
60         c[i] = fft_a[i] * fft_b[i];
61     }
62     fft(c, 1);
63
64     vector<long long> res(n);
65     for (int i = 0; i < n; ++i) {
66         res[i] = round(c[i].real()); // res = c[i].real();
67                                     // se for vector de
68                                     // double
69     }
70     // while(size(res) > 1 && res.back() == 0)
71     // res.pop_back(); // apenas para quando os
72     // zeros direita nao importarem
73     return res;
74 }

```

6.4 Fatoração

Algoritmos para fatorar um número.

Fatoração Simples

Fatora um número N.

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

Crivo Linear

Pré-computa todos os fatores primos até MAX.

Utilizado para fatorar um número N menor que MAX.

- Complexidade de tempo: Pré-processamento $\mathcal{O}(MAX)$

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

Fatoração Rápida

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

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

Pollard-Rho

Descobre um divisor de um número N.

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

Codigo: naive_factorize.cpp

```

1 vector<int> factorize(int n) {
2     vector<int> factors;
3     for (long long d = 2; d * d <= n; d++) {
4         while (n % d == 0) {
5             factors.push_back(d);
6             n /= d;
7         }
8     }
9     if (n != 1) {
10         factors.push_back(n);
11     }
12     return factors;
13 }

```

Codigo: linear_sieve_factorize.cpp

```

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]) {
15                break;
16            }
17        }
18    }
19 }
20 vector<int> factorize(int x) {
21     if (x < 2) {
22         return {};
23     }
24     vector<int> v;
25     for (int lpx = lp[x]; x >= lpx; x = factor[x]) {
26         v.emplace_back(lp[x]);
27     }
28     return v;
29 }
30 }

```

Codigo: pollard_rho.cpp

```

1 long long mod_mul(long long a, long long b, long long m) { return (__int128)a * b % m;
2 }
3 long long pollard_rho(long long n) {
4     auto f = [n](long long x) {
5         return mod_mul(x, x, n) + 1;
6     };
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) {
10            x = ++i, y = f(x);
11        }
12        if ((q = mod_mul(prd, max(x, y) - min(x, y), n))) {
13            prd = q;
14        }
15        x = f(x), y = f(f(y));
16    }
17    return __gcd(prd, n);
18 }

```

Codigo: fast_factorize.cpp

```

1 // usa miller_rabin.cpp!! olhar em
2 // matematica/primos usa pollard_rho.cpp!! olhar em
3 // matematica/fatoracao
4
5 vector<long long> factorize(long long n) {
6     if (n == 1) {
7         return {};
8     }
9     if (miller_rabin(n)) {
10        return {n};
11    }
12    long long x = pollard_rho(n);
13    auto l = factorize(x), r = factorize(n / x);
14    l.insert(l.end(), all(r));
15    return l;
16 }

```

6.5 GCD

Algoritmo Euclides para computar o Máximo Divisor Comum (MDC em português; GCD em inglês), e variações.

Read in [English](README.en.md)

Algoritmo de Euclides

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

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

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

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: $\mathcal{O}(\log(n))$

Codigo: gcd.cpp

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

Codigo: extended_gcd.cpp

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

Codigo: extended_gcd_recursive.cpp

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

6.6 Inverso Modular

<!-- DESCRIÇÃO -->

Algoritmos para calcular o inverso modular de um número. O inverso modular de um inteiro a é outro inteiro x tal que $a \cdot x \equiv 1 \pmod{MOD}$

<!-- DESCRIÇÃO -->

O inverso modular de um inteiro a é outro inteiro x tal que $a \cdot x$ é congruente a 1 (mod MOD).

Inverso Modular

Calcula o inverso modular de a .

Utiliza o algoritmo Exp Mod, portanto, espera-se que MOD seja um número primo.

* Complexidade de tempo: $\mathcal{O}(\log(\text{MOD}))$.

* Complexidade de espaço: $\mathcal{O}(1)$.

Inverso Modular por MDC Estendido

Calcula o inverso modular de a .

Utiliza o algoritmo Euclides Estendido, portanto, espera-se que MOD seja coprimo com a .

Retorna -1 se essa suposição for quebrada.

* Complexidade de tempo: $\mathcal{O}(\log(\text{MOD}))$.

* Complexidade de espaço: $\mathcal{O}(1)$.

Inverso Modular para 1 até MAX

Calcula o inverso modular para todos os números entre 1 e MAX.

Espera-se que MOD seja primo.

* Complexidade de tempo: $\mathcal{O}(\text{MAX})$.

* Complexidade de espaço: $\mathcal{O}(\text{MAX})$.

Inverso Modular para todas as potências

Seja b um número inteiro qualquer.

Calcula o inverso modular para todas as potências de b entre b^0 e b^{MAX} .

É necessário calcular antecipadamente o inverso modular de b , para 2 é sempre $(\text{MOD}+1)/2$.

Espera-se que MOD seja coprimo com b .

* Complexidade de tempo: $\mathcal{O}(\text{MAX})$.

* Complexidade de espaço: $\mathcal{O}(\text{MAX})$.

Codigo: modular_inverse_linear.cpp

```

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     }
8 }

```

Codigo: modular_inverse_pow.cpp

```

1 const ll INVB = (MOD + 1) / 2; // Modular inverse of the base,
2                               // for 2 it is (MOD+1)/2
3
4 ll inv[MAX]; // Modular inverse of b^i
5
6 void compute_inv() {
7     inv[0] = 1;
8     for (int i = 1; i < MAX; i++) {
9         inv[i] = inv[i - 1] * INVB % MOD;
10    }
11 }

```

Codigo: modular_inverse.cpp

```

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

```

Codigo: modular_inverse_coprime.cpp

```

1 int inv(int a) {
2     int x, y;
3     int g = extended_gcd(a, MOD, x, y);
4     if (g == 1) {
5         return (x % m + m) % m;
6     }
7     return -1;
8 }

```

6.7 NTT

<!-- DESCRIÇÃO -->

Computa a multiplicação de polinômios com coeficientes inteiros módulo um número primo.

<!-- DESCRIÇÃO -->

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

- Complexidade de tempo: $\mathcal{O}(N \cdot \log(N))$

Constantes finais devem ser menores do que 10^9 .

Para constantes entre 10^9 e 10^{18} é necessário codificar também big_convolution

Codigo: ntt.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_inv[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) {
14             res = (res * b) % m;
15         }
16         e /= 2;
17         b = (b * b) % m;
18     }
19     return res;
20 }
21
22 void ntt(poly &a, bool invert, int id) {
23     ll n = (ll)a.size(), m = mod[id];
24     for (ll i = 1, j = 0; i < n; ++i) {
25         ll bit = n >> 1;
26         for (; j >= bit; bit >>= 1) {
27             j -= bit;
28         }
29         j += bit;

```

```

30     if (i < j) {
31         swap(a[i], a[j]);
32     }
33 }
34 for (ll len = 2, wlen; len <= n; len <= 1) {
35     wlen = invert ? root_1[id] : root[id];
36     for (ll i = len; i < root_pw[id]; i <= 1) {
37         wlen = (wlen * wlen) % m;
38     }
39     for (ll i = 0; i < n; i += len) {
40         ll w = 1;
41         for (ll j = 0; j < len / 2; j++) {
42             ll u = a[i + j], v = (a[i + j + len / 2] * w) % m;
43             a[i + j] = (u + v) % m;
44             a[i + j + len / 2] = (u - v + m) % m;
45             w = (w * wlen) % m;
46         }
47     }
48 }
49 if (invert) {
50     ll inv = modInv(n, m);
51     for (ll i = 0; i < n; i++) {
52         a[i] = (a[i] * inv) % m;
53     }
54 }
55 }
56
57 poly convolution(poly a, poly b, int id = 0) {
58     ll n = 1LL, len = (1LL + a.size() + b.size());
59     while (n < len) {
60         n *= 2;
61     }
62     a.resize(n);
63     b.resize(n);
64     ntt(a, 0, id);
65     ntt(b, 0, id);
66     poly answer(n);
67     for (ll i = 0; i < n; i++) {
68         answer[i] = (a[i] * b[i]);
69     }
70     ntt(answer, 1, id);
71     return answer;
72 }

```

Codigo: big_convolution.cpp

1

```

2 ll mod_mul(ll a, ll b, ll m) { return (__int128)a * b % m; }
3 ll ext_gcd(ll a, ll b, ll &x, ll &y) {
4     if (!b) {
5         x = 1;
6         y = 0;
7         return a;
8     } else {
9         ll g = ext_gcd(b, a % b, y, x);
10        y -= a / b * x;
11        return g;
12    }
13 }
14
15 // convolution mod 1,097,572,091,361,755,137
16 poly big_convolution(poly a, poly b) {
17     poly r0, r1, answer;
18     r0 = convolution(a, b, 1);
19     r1 = convolution(a, b, 2);
20
21     ll s, r, p = mod[1] * mod[2];
22     ext_gcd(mod[1], mod[2], r, s);
23
24     answer.resize(r0.size());
25     for (int i = 0; i < (int)answer.size(); i++) {
26         answer[i] = (mod_mul((s * mod[2] + p) % p, r0[i], p) +
27             mod_mul((r * mod[1] + p) % p, r1[i], p) + p) %
28             p;
29     }
30     return answer;
31 }

```

6.8 Primos

Algoritmos relacionados a números primos.

Crivo de Eratóstenes

Computa a primalidade de todos os números até N , quase tão rápido quanto o crivo linear.

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

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

Miller-Rabin

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

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

Teste Ingênuo

Computa a primalidade de um número N.

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

Codigo: sieve.cpp

```
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 < n; j += i) {
9                 is_prime[j] = false;
10            }
11        }
12    }
13    return is_prime;
14 }
```

Codigo: naive_is_prime.cpp

```
1 bool is_prime(int n) {
2     for (long long d = 2; d * d <= n; d++) {
3         if (n % d == 0) {
4             return false;
5         }
6     }
7     return true;
8 }
```

Codigo: miller_rabin.cpp

```
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) {
6             result = (__int128)result * base % mod;
7         }
8         base = (__int128)base * base % mod;
9         e >>= 1;
10    }
11    return result;
12 }
13
14 bool is_composite(long long n, long long a, long long d, int s) {
15     long long x = power(a, d, n);
16     if (x == 1 || x == n - 1) {
17         return false;
18     }
19     for (int r = 1; r < s; r++) {
20         x = (__int128)x * x % n;
21         if (x == n - 1) {
22             return false;
23         }
24     }
25     return true;
26 }
27
28 bool miller_rabin(long long n) {
29     if (n < 2) {
30         return false;
31     }
32     int r = 0;
33     long long d = n - 1;
34     while ((d & 1) == 0) {
35         d >>= 1, ++r;
36     }
37     for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
38         if (n == a) {
39             return true;
40         }
41         if (is_composite(n, a, d, r)) {
42             return false;
43         }
44     }
45     return true;
46 }
```

6.9 Sum of floor (n div i)

Computa

$$\sum_{i=1}^n \left\lfloor \frac{n}{i} \right\rfloor$$

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

Código: sum_of_floor.cpp

```
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     i--;
10    for (int j = 1; n / (j + 1) >= i; j++) {
11        answer += ((n / j - n / (j + 1)) % MOD) * j % MOD;
12        answer %= MOD;
13    }
14    return answer;
15 }
```

6.10 Teorema do Resto Chinês

Algoritmo que resolve o sistema $x \equiv a_i \pmod{m_i}$, onde m_i são primos entre si.

Retorna -1 se a resposta não existir.

Código: crt.cpp

```
1 ll extended_gcd(ll a, ll b, ll &x, ll &y) {
2     if (b == 0) {
3         x = 1;
4         y = 0;
5         return a;
6     } else {
7         ll g = extended_gcd(b, a % b, y, x);
8         y -= a / b * x;
9         return g;
10    }
11 }
12
13 ll crt(vector<ll> rem, vector<ll> mod) {
14     int n = rem.size();
15     if (n == 0) {
16         return 0;
17     }
18     __int128 ans = rem[0], m = mod[0];
19     for (int i = 1; i < n; i++) {
20         ll x, y;
21         ll g = extended_gcd(mod[i], m, x, y);
22         if ((ans - rem[i]) % g != 0) {
23             return -1;
24         }
25         ans = ans + (__int128)1 * (rem[i] - ans) * (m / g) * y;
26         m = (__int128)(mod[i] / g) * (m / g) * g;
27         ans = (ans % m + m) % m;
28     }
29     return ans;
30 }
```

6.11 Totiente de Euler

Código para computar o totiente de Euler.

Totiente de Euler (Phi) para um número

Computa o totiente para um único número N.

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

Totiente de Euler (Phi) entre 1 e N

Computa o totiente entre 1 e N.

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

Codigo: phi_1_to_n.cpp

```
1 vector<int> phi_1_to_n(int n) {
2     vector<int> phi(n + 1);
3     for (int i = 0; i <= n; i++) {
4         phi[i] = i;
5     }
6     for (int i = 2; i <= n; i++) {
7         if (phi[i] == i) {
8             for (int j = i; j <= n; j += i) {
9                 phi[j] -= phi[j] / i;
10            }
11        }
12    }
13    return phi;
14 }
```

Codigo: phi.cpp

```
1 int phi(int n) {
2     int result = n;
3     for (int i = 2; i * i <= n; i++) {
4         if (n % i == 0) {
5             while (n % i == 0) {
6                 n /= i;
7             }
8             result -= result / i;
9         }
10    }
11    if (n > 1) {
12        result -= result / n;
13    }
14    return result;
15 }
```

Capítulo 7

Teórico

7.1 Alguns Números Primos

7.1.1 Primo com Truncamento à Esquerda

Número primo tal que qualquer sufixo dele é um número primo
357,686,312,646,216,567,629,137

7.1.2 Números Primos de Mersenne

Números primos da forma $2^m - 1$

Expoente (m)	Representação Decimal
2	3
3	7
5	31
7	127
13	8,191
17	131,071
19	524,287
31	2,147,483,647
61	$2,3 * 10^{18}$
89	$6,1 * 10^{26}$
107	$1,6 * 10^{32}$
127	$1,7 * 10^{38}$

7.2 Constantes em C++

Constante	Nome em C++	Valor
π	M_PI	3.141592...
$\pi/2$	M_PI_2	1.570796...
$\pi/4$	M_PI_4	0.785398...
$1/\pi$	M_1_PI	0.318309...
$2/\pi$	M_2_PI	0.636619...
$2/\sqrt{\pi}$	M_2_SQRTPI	1.128379...
$\sqrt{2}$	M_SQRT2	1.414213...
$1/\sqrt{2}$	M_SQRT1_2	0.707106...
e	M_E	2.718281...
$\log_2 e$	M_LOG2E	1.442695...
$\log_{10} e$	M_LOG10E	0.434294...
$\ln 2$	M_LN2	0.693147...
$\ln 10$	M_LN10	2.302585...

7.3 Operadores Lineares

7.3.1 Rotação no sentido anti-horário por θ°

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

7.3.2 Reflexão em relação à reta $y = mx$

$$\frac{1}{m^2 + 1} \begin{bmatrix} 1 - m^2 & 2m \\ 2m & m^2 - 1 \end{bmatrix}$$

7.3.3 Inversa de uma matriz 2x2 A

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

7.3.4 Cisalhamento horizontal por K

$$\begin{bmatrix} 1 & K \\ 0 & 1 \end{bmatrix}$$

7.3.5 Cisalhamento vertical por K

$$\begin{bmatrix} 1 & 0 \\ K & 1 \end{bmatrix}$$

7.3.6 Mudança de base

\vec{a}_β são as coordenadas do vetor \vec{a} na base β .

\vec{a} são as coordenadas do vetor \vec{a} na base canônica.

\vec{b}_1 e \vec{b}_2 são os vetores de base para β .

C é uma matriz que muda da base β para a base canônica.

$$C\vec{a}_\beta = \vec{a}$$

$$C^{-1}\vec{a} = \vec{a}_\beta$$

$$C = \begin{bmatrix} b_{1x} & b_{2x} \\ b_{1y} & b_{2y} \end{bmatrix}$$

7.3.7 Propriedades das operações de matriz

$$(AB)^{-1} = A^{-1}B^{-1}$$

$$(AB)^T = B^T A^T$$

$$(A^{-1})^T = (A^T)^{-1}$$

$$(A + B)^T = A^T + B^T$$

$$\det(A) = \det(A^T)$$

$$\det(AB) = \det(A)\det(B)$$

Seja A uma matriz $N \times N$:

$$\det(kA) = k^N \det(A)$$