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

12 de março de 2024

Índice

Capítulo 1

STL (Standard Template Library - C++)

Os templates da STL são estruturas de dados e algoritmos já implementadas em `C++` que facilitam as implementações, além de serem muito eficients. Em geral, todas estão incluídas no cabeçalho `<bits/stdc++.h>`. As estruturas são templates genéricos, podem ser usadas com qualquer tipo, todos os exemplos a seguir são com `int` apenas por motivos de simplicidade.

1.1 Vector

Um vetor dinâmico (que pode crescer e diminuir de tamanho).

- `vector<int> v(n, 0)`: Cria um vetor de inteiros com `n` elementos, todos inicializados com 0 - $\mathcal{O}(n)$
- `v.push_back(x)`: Adiciona o elemento `x` no final do vetor - $\mathcal{O}(1)$
- `v.pop_back()`: Remove o último elemento do vetor - $\mathcal{O}(1)$
- `v.size()`: Retorna o tamanho do vetor - $\mathcal{O}(1)$
- `v.empty()`: Retorna `true` se o vetor estiver vazio - $\mathcal{O}(1)$
- `v.clear()`: Remove todos os elementos do vetor - $\mathcal{O}(n)$
- `v.front()`: Retorna o primeiro elemento do vetor - $\mathcal{O}(1)$
- `v.back()`: Retorna o último elemento do vetor - $\mathcal{O}(1)$
- `v.begin()`: Retorna um iterador para o primeiro elemento do vetor - $\mathcal{O}(1)$
- `v.end()`: Retorna um iterador para o elemento seguinte ao último do vetor - $\mathcal{O}(1)$
- `v.insert(it, x)`: Insere o elemento `x` na posição apontada pelo iterador `it` - $\mathcal{O}(n)$
- `v.erase(it)`: Remove o elemento apontado pelo iterador `it` - $\mathcal{O}(n)$
- `v.erase(it1, it2)`: Remove os elementos no intervalo `[it1, it2)` - $\mathcal{O}(n)$
- `v.resize(n)`: Redimensiona o vetor para `n` elementos - $\mathcal{O}(n)$
- `v.resize(n, x)`: Redimensiona o vetor para `n` elementos, todos inicializados com `x` - $\mathcal{O}(n)$

1.2 Pair

Um par de elementos (de tipos possivelmente diferentes).

- `pair<int, int> p`: Cria um par de inteiros - $\mathcal{O}(1)$
- `p.first`: Retorna o primeiro elemento do par - $\mathcal{O}(1)$
- `p.second`: Retorna o segundo elemento do par - $\mathcal{O}(1)$

1.3 Set

Um conjunto de elementos únicos. Por baixo, é uma árvore de busca binária balanceada.

- `set<int> s`: Cria um conjunto de inteiros - $\mathcal{O}(1)$
- `s.insert(x)`: Insere o elemento `x` no conjunto - $\mathcal{O}(\log n)$
- `s.erase(x)`: Remove o elemento `x` do conjunto - $\mathcal{O}(\log n)$
- `s.find(x)`: Retorna um iterador para o elemento `x` no conjunto, ou `s.end()` se não existir - $\mathcal{O}(\log n)$
- `s.size()`: Retorna o tamanho do conjunto - $\mathcal{O}(1)$
- `s.empty()`: Retorna `true` se o conjunto estiver vazio - $\mathcal{O}(1)$
- `s.clear()`: Remove todos os elementos do conjunto - $\mathcal{O}(n)$
- `s.begin()`: Retorna um iterador para o primeiro elemento do conjunto - $\mathcal{O}(1)$
- `s.end()`: Retorna um iterador para o elemento seguinte ao último do conjunto - $\mathcal{O}(1)$

1.4 Multiset

Basicamente um `set`, mas permite elementos repetidos. Possui todos os métodos de um `set`.

Declaração: `multiset<int> ms`.

Um detalhe é que, ao usar o método `erase`, ele remove todas as ocorrências do elemento. Para remover apenas uma ocorrência, usar `ms.erase(ms.find(x))`.

1.5 Map

Um conjunto de pares chave-valor, onde as chaves são únicas. Por baixo, é uma árvore de busca binária balanceada.

- `map<int, int> m`: Cria um mapa de inteiros para inteiros - $\mathcal{O}(1)$
- `m[key]`: Retorna o valor associado à chave `key` - $\mathcal{O}(\log n)$
- `m[key] = value`: Associa o valor `value` à chave `key` - $\mathcal{O}(\log n)$
- `m.erase(key)`: Remove a chave `key` do mapa - $\mathcal{O}(\log n)$
- `m.find(key)`: Retorna um iterador para o par chave-valor com chave `key`, ou `m.end()` se não existir - $\mathcal{O}(\log n)$
- `m.size()`: Retorna o tamanho do mapa - $\mathcal{O}(1)$
- `m.empty()`: Retorna `true` se o mapa estiver vazio - $\mathcal{O}(1)$
- `m.clear()`: Remove todos os pares chave-valor do mapa - $\mathcal{O}(n)$
- `m.begin()`: Retorna um iterador para o primeiro par chave-valor do mapa - $\mathcal{O}(1)$
- `m.end()`: Retorna um iterador para o par chave-valor seguinte ao último do mapa - $\mathcal{O}(1)$

1.6 Queue

Uma fila (primeiro a entrar, primeiro a sair).

- `queue<int> q`: Cria uma fila de inteiros - $\mathcal{O}(1)$
- `q.push(x)`: Adiciona o elemento `x` no final da fila - $\mathcal{O}(1)$
- `q.pop()`: Remove o primeiro elemento da fila - $\mathcal{O}(1)$
- `q.front()`: Retorna o primeiro elemento da fila - $\mathcal{O}(1)$
- `q.size()`: Retorna o tamanho da fila - $\mathcal{O}(1)$
- `q.empty()`: Retorna `true` se a fila estiver vazia - $\mathcal{O}(1)$

1.7 Priority Queue

Uma fila de prioridade (o maior elemento é o primeiro a sair).

- `priority_queue<int> pq`: Cria uma fila de prioridade de inteiros - $\mathcal{O}(1)$
- `pq.push(x)`: Adiciona o elemento `x` na fila de prioridade - $\mathcal{O}(\log n)$
- `pq.pop()`: Remove o maior elemento da fila de prioridade - $\mathcal{O}(\log n)$
- `pq.top()`: Retorna o maior elemento da fila de prioridade - $\mathcal{O}(1)$
- `pq.size()`: Retorna o tamanho da fila de prioridade - $\mathcal{O}(1)$
- `pq.empty()`: Retorna `true` se a fila de prioridade estiver vazia - $\mathcal{O}(1)$

Para fazer uma fila de prioridade que o menor elemento é o primeiro a sair, usar `priority_queue<int, vector<int>, greater<>> pq`.

1.8 Stack

Uma pilha (último a entrar, primeiro a sair).

- `stack<int> s`: Cria uma pilha de inteiros - $\mathcal{O}(1)$
- `s.push(x)`: Adiciona o elemento `x` no topo da pilha - $\mathcal{O}(1)$
- `s.pop()`: Remove o elemento do topo da pilha - $\mathcal{O}(1)$
- `s.top()`: Retorna o elemento do topo da pilha - $\mathcal{O}(1)$
- `s.size()`: Retorna o tamanho da pilha - $\mathcal{O}(1)$
- `s.empty()`: Retorna `true` se a pilha estiver vazia - $\mathcal{O}(1)$

1.9 Funções úteis

- `min(a, b)`: Retorna o menor entre `a` e `b` - $\mathcal{O}(1)$
- `max(a, b)`: Retorna o maior entre `a` e `b` - $\mathcal{O}(1)$
- `abs(a)`: Retorna o valor absoluto de `a` - $\mathcal{O}(1)$
- `swap(a, b)`: Troca os valores de `a` e `b` - $\mathcal{O}(1)$
- `sqrt(a)`: Retorna a raiz quadrada de `a` - $\mathcal{O}(\log a)$
- `ceil(a)`: Retorna o menor inteiro maior ou igual a `a` - $\mathcal{O}(1)$
- `floor(a)`: Retorna o maior inteiro menor ou igual a `a` - $\mathcal{O}(1)$
- `round(a)`: Retorna o inteiro mais próximo de `a` - $\mathcal{O}(1)$

1.10 Funções úteis para vetores

Para usar em `std::vector`, sempre passar `v.begin()` e `v.end()` como argumentos pra essas funções.

Se for um vetor estilo `C`, usar `v` e `v + n`. Exemplo:

```cpp
int v[10];
sort(v, v + 10);
```

- `fill(v.begin(), v.end(), x)`: Preenche o vetor `v` com o valor `x` - $\mathcal{O}(n)$
- `sort(v.begin(), v.end())`: Ordena o vetor `v` - $\mathcal{O}(n \log n)$
- `reverse(v.begin(), v.end())`: Inverte o vetor `v` - $\mathcal{O}(n)$
- `accumulate(v.begin(), v.end(), 0)`: Soma todos os elementos do vetor `v` - $\mathcal{O}(n)$
- `max_element(v.begin(), v.end())`: Retorna um iterador para o maior elemento do vetor `v` - $\mathcal{O}(n)$

- `min_element(v.begin(), v.end())`: Retorna um iterador para o menor elemento do vetor `v` - $\mathcal{O}(n)$
- `count(v.begin(), v.end(), x)`: Retorna o número de ocorrências do elemento `x` no vetor `v` - $\mathcal{O}(n)$
- `find(v.begin(), v.end(), x)`: Retorna um iterador para a primeira ocorrência do elemento `x` no vetor `v`, ou `v.end()` se não existir - $\mathcal{O}(n)$
- `lower_bound(v.begin(), v.end(), x)`: Retorna um iterador para o primeiro elemento maior ou igual a `x` no vetor `v` (o vetor deve estar ordenado) - $\mathcal{O}(\log n)$
- `upper_bound(v.begin(), v.end(), x)`: Retorna um iterador para o primeiro elemento estritamente maior que `x` no vetor `v` (o vetor deve estar ordenado) - $\mathcal{O}(\log n)$
- `next_permutation(a.begin(), a.end())`: Rearranja os elementos do vetor `a` para a próxima permutação lexicograficamente maior - $\mathcal{O}(n)$

Capítulo 2

Extra

2.1 CPP

Template de C++ para usar na Maratona.

Codigo: template.cpp

```cpp
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;
using ll = long long;

void solve() { }

signed main() {
    cin.tie(0)->sync_with_stdio(0);
    solve();
}
```

2.2 Debug

Template para debugar variáveis em `C++`. Até a linha 17 é opcional, é pra permitir que seja possível debugar `std::pair` e `std::vector`.

Para usar, basta compilar com a flag `-DBRUTE` (o template `run` já tem essa flag). E no código usar `debug(x, y, z)` para debugar as variáveis `x`, `y` e `z`.

Codigo: debug.cpp

```cpp
1  template <typename T, typename U>
2  ostream &operator<<(ostream &os, const pair<T, U> &p) { // opcional
3      os << "(" << p.first << ", " << p.second << ")";
4      return os;
5  }
6  template <typename T> ostream &operator<<(ostream &os, const vector<T> &v) { // opcional
7      os << "{";
8      int n = (int)v.size();
9      for (int i = 0; i < n; i++) {
10         os << v[i];
11         if (i < n - 1)
12             os << ", ";
13     }
14     os << "}";
15     return os;
16 }
17
18 void _print() { }
19 template <typename T, typename... U> void _print(T a, U... b) {
20     if (sizeof...(b)) {
21         cerr << a << ", ";
22         _print(b...);
23     } else
24         cerr << a;
25 }
26 #ifdef BRUTE
27 #define debug(x...) cerr << "[" << #x << "] = [", _print(x), cerr << "]" << endl
28 #else
```

```
#define debug(...)
#endif
```

2.3 Random

É possível usar a função `rand()` para gerar números aleatórios em `C++`.

Útil para gerar casos aleatórios em stress test, porém não é recomendado para usar em soluções.

`rand()` gera números entre `0` e `RAND_MAX` (que é pelo menos `32767`), mas costuma ser `2147483647` (depende do sistema/arquitetura).

Para usar o `rand()`, recomenda-se no mínimo chamar a função `srand(time(0))` no início da `main()` para inicializar a seed do gerador de números aleatórios.

Para usar números aleatórios em soluções, recomenda-se o uso do `mt19937` que está no código abaixo.

A função `rng()` gera números entre `0` e `UINT_MAX` (que é `4294967295`).

Para gerar números aleatórios de 64 bits, usar `mt19937_64` como tipo do `rng`.

Recomenda-se o uso da função `uniform(l, r)` para gerar números aleatórios no intervalo fechado $[l, r]$ usando o `mt19937`.

Codigo: rand.cpp

```cpp
mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());

int uniform(int l, int r) { return uniform_int_distribution<int>(l, r)(rng); }
```

2.4 Run

Arquivo útil para compilar e rodar um programa em `C++` com flags que ajudam a debugar.

Basta criar um arquivo chamado `run`, adicionar o código abaixo e dar permissão de execução com `chmod +x run`.

Para executar um arquivo `a.cpp`, basta rodar `./run a.cpp`.

Codigo: run

```bash
#!/bin/bash
g++ -std=c++20 -DBRUTE -O2 -Wall -Wextra -Wconversion -Wfatal-errors -fsanitize=address,undefined $1 && ./a.out
```

2.5 Stress Test

Script muito útil para achar casos em que sua solução gera uma resposta incorreta.

Deve-se criar uma solução bruteforce (que garantidamente está correta, ainda que seja lenta) e um gerador de casos aleatórios para seu problema.

Codigo: stress.sh

```bash
#!/bin/bash
set -e

g++ -O2 gen.cpp -o gen # pode fazer o gerador em python se preferir
g++ -O2 brute.cpp -o brute
g++ -O2 code.cpp -o code

for((i = 1; ; ++i)); do
    ./gen $i > in
    ./code < in > out
    ./brute < in > ok
    diff -w out ok || break
    echo "Passed test: " $i
done

echo "WA no seguinte teste:"
cat in
echo "Sua resposta eh:"
cat out
echo "A resposta correta eh:"
cat ok
```

2.6 Unordered Custom Hash

As funções de hash padrão do `unordered_map` e `unordered_set` são muito propícias a colisões (principalmente se o setter da questão criar casos de teste pensando nisso).

Para evitar isso, é possível criar uma função de hash customizada.

Entretanto, é bem raro ser necessário usar isso. Geralmente o fator $\mathcal{O}(\log n)$ de um `map` é suficiente.

Exemplo de uso: `unordered_map<int, int, custom_hash> mp;`

Codigo: custom_hash.cpp

```cpp
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
            chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

2.7 Vim

Template de arquivo `.vimrc` para configuração do Vim.

Codigo: vimrc

```vim
set nu ai si cindent et ts=4 sw=4 so=10 nosm nohls
inoremap {} {}<left><return><up><end><return> " remap pra chaves
au BufReadPost * if line("'\"") > 0 && line("'\"") <= line("$") | exe "normal! g'\"" | endif " volta pro lugar onde estava quando saiu do arquivo
```

Capítulo 3

Teórico

3.1 Definições

Algumas definições e termos importantes:

3.1.1 Funções

- **Comutativa**: Uma função $f(x, y)$ é comutativa se $f(x, y) = f(y, x)$.
- **Associativa**: Uma função $f(x, y)$ é associativa se $f(x, f(y, z)) = f(f(x, y), z)$.
- **Idempotente**: Uma função $f(x, y)$ é idempotente se $f(x, x) = x$.

3.1.2 Grafos

- **Grafo**: Um grafo é um conjunto de vértices e um conjunto de arestas que conectam os vértices.
- **Grafo Conexo**: Um grafo é conexo se existe um caminho entre todos os pares de vértices.
- **Grafo Bipartido**: Um grafo é bipartido se é possível dividir os vértices em dois conjuntos disjuntos de forma que todas as arestas conectem um vértice de um conjunto com um vértice do outro conjunto, ou seja, não existem arestas que conectem vértices do mesmo conjunto.
- **Árvore**: Um grafo é uma árvore se ele é conexo e não possui ciclos.
- **Árvore Geradora Mínima (AGM)**: Uma árvore geradora mínima é uma árvore que conecta todos os vértices de um grafo e possui o menor custo possível, também conhecido como *Minimum Spanning Tree (MST)*.

3.2 Números primos

Números primos são muito úteis para funções de hashing (dentre outras coisas). Aqui vão vários números primos interessantes:

3.2.1 Primo com truncamento à esquerda

Número primo tal que qualquer sufixo dele é um número primo

$$357{,}686{,}312{,}646{,}216{,}567{,}629{,}137$$

3.2.2 Primos gêmeos (Twin Primes)

Pares de primos da forma $(p, p+2)$ (aqui tem só alguns pares aleatórios, existem muitos outros).

Primo	**Primo + 2**	**Ordem**
5	7	10^0
17	19	10^1
461	463	10^2
3461	3463	10^3
34499	34501	10^4
487829	487831	10^5
5111999	5112001	10^6
30684887	30684889	10^7
361290539	361290541	10^8
1000000007	1000000009	10^9
1005599459	1005599461	10^9

3.2.3 Números primos de Mersenne

São os números primos da forma $2^m - 1$, onde m é um número inteiro positivo.

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	$1,7 * 10^{38}$

3.3 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...

3.4 Operadores lineares

3.4.1 Rotação no sentido anti-horário por $\theta°$

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

3.4.2 Reflexão em relação à reta $y = mx$

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

3.4.3 Inversa de uma matriz 2x2 A

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

3.4.4 Cisalhamento horizontal por K

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

3.4.5 Cisalhamento vertical por K

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

3.4.6 Mudança de base

$\vec{a}_\beta$ são as coordenadas do vetor $\vec{a}$ na base β.
$\vec{a}$ são as coordenadas do vetor $\vec{a}$ na base canônica.
$\vec{b1}$ e $\vec{b2}$ são os vetores de base para β.
C é uma matriz que muda da base β para a base canônica.

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

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

$$C = \begin{bmatrix} b1_x & b2_x \\ b1_y & b2_y \end{bmatrix}$$

3.4.7 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 NxN:

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

Capítulo 4

Estruturas de Dados

4.1 Disjoint Set Union

4.1.1 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.

Codigo: dsu.cpp

```
struct DSU {
    vector<int> par, sz;
    int number_of_sets;
    DSU(int n = 0) : par(n), sz(n, 1), number_of_sets(n) {
        iota(par.begin(), par.end(), 0);
    }
    int find(int a) { return a == par[a] ? a : par[a] = find(par[a]); }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) {
            return false;
        }
        number_of_sets--;
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        par[b] = a;
        sz[a] += sz[b];
        return true;
    }
};
```

4.1.2 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 $\mathcal{O}(\log n)$.

Codigo: bipartite_dsu.cpp

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

```cpp
    int find(int a) { return a == par[a] ? a : find(par[a]); }
    int color(int a) { return a == par[a] ? c[a] : c[a] ^ color(par[a]); }
    bool bipartite(int a) { return bip[find(a)]; }
    bool unite(int a, int b) {
        bool equal_color = color(a) == color(b);
        a = find(a), b = find(b);
        if (a == b) {
            if (equal_color) {
                bip[a] = 0;
                all_bipartite = 0;
            }
            return false;
        }
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        number_of_sets--;
        par[b] = a;
        sz[a] += sz[b];
        if (equal_color) {
            c[b] = 1;
        }
        bip[a] &= bip[b];
        all_bipartite &= bip[a];
        return true;
    }
};
```

4.1.3 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

```cpp
struct Rollback_DSU {
    vector<int> par, sz;
    int number_of_sets;
    stack<stack<pair<int &, int>>> changes;
    Rollback_DSU(int n = 0) : par(n), sz(n, 1), number_of_sets(n) {
        iota(par.begin(), par.end(), 0);
        changes.emplace();
    }
    int find(int a) { return a == par[a] ? a : find(par[a]); }
    void checkpoint() { changes.emplace(); }
    void change(int &a, int b) {
        changes.top().emplace(a, a);
        a = b;
    }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) {
            return false;
        }
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        change(number_of_sets, number_of_sets - 1);
        change(par[b], a);
        change(sz[a], sz[a] + sz[b]);
        return true;
    }
    void rollback() {
        while (changes.top().size()) {
            auto [a, b] = changes.top().top();
            a = b;
            changes.top().pop();
        }
        changes.pop();
    }
};
```

4.1.4 DSU Rollback Bipartido

DSU com rollback e bipartido.

Codigo: full_dsu.cpp

```cpp
struct Full_DSU {
    vector<int> par, sz, c, bip;
```

```
    int number_of_sets, all_bipartite;
    stack<stack<pair<int &, int>>> changes;
    Full_DSU(int n = 0)
        : par(n), sz(n, 1), c(n), bip(n, 1), number_of_sets(n), all_bipartite(1) {
        iota(par.begin(), par.end(), 0);
        changes.emplace();
    }
    int find(int a) { return a == par[a] ? a : find(par[a]); }
    int color(int a) { return a == par[a] ? c[a] : c[a] ^ color(par[a]); }
    bool bipartite(int a) { return bip[find(a)]; }
    void checkpoint() { changes.emplace(); }
    void change(int &a, int b) {
        changes.top().emplace(a, a);
        a = b;
    }
    bool unite(int a, int b) {
        bool equal_color = color(a) == color(b);
        a = find(a), b = find(b);
        if (a == b) {
            if (equal_color) {
                change(bip[a], 0);
                change(all_bipartite, 0);
            }
            return false;
        }
        if (sz[a] < sz[b]) {
            swap(a, b);
        }
        change(number_of_sets, number_of_sets - 1);
        change(par[b], a);
        change(sz[a], sz[a] + sz[b]);
        change(bip[a], bip[a] && bip[b]);
        change(all_bipartite, all_bipartite && bip[a]);
        if (equal_color) {
            change(c[b], 1);
        }
        return true;
    }
    void rollback() {
        while (changes.top().size()) {
            auto [a, b] = changes.top().top();
            a = b;
            changes.top().pop();
        }
        changes.pop();
    }
};
```

4.1.5 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

```
struct Offline_DSU : Full_DSU {
    int time;
    Offline_DSU(int n = 0) : Full_DSU(n), time(0) { }
    struct query {
        int type, a, b;
    };
    vector<query> queries;
    void askConnect(int a, int b) {
        if (a > b) {
            swap(a, b);
        }
        queries.push_back({0, a, b});
        time++;
    }
    void askBipartite(int a) {
        queries.push_back({1, a, -1});
        time++;
    }
    void askAllBipartite() {
        queries.push_back({2, -1, -1});
        time++;
    }
    void addEdge(int a, int b) {
        if (a > b) {
            swap(a, b);
        }
        queries.push_back({3, a, b});
        time++;
    }
    void removeEdge(int a, int b) {
        if (a > b) {
            swap(a, b);
        }
```

```cpp
        queries.push_back({4, a, b});
        time++;
    }
    vector<vector<pair<int, int>>> lazy;
    void update(int l, int r, pair<int, int> edge, int u, int L, int R) {
        if (R < l || L > r) {
            return;
        }
        if (L >= l && R <= r) {
            lazy[u].push_back(edge);
            return;
        }
        int mid = (L + R) / 2;
        update(l, r, edge, 2 * u, L, mid);
        update(l, r, edge, 2 * u + 1, mid + 1, R);
    }
    void dfs(int u, int L, int R, vector<int> &ans) {
        if (L > R) {
            return;
        }
        checkpoint();
        for (auto [a, b] : lazy[u]) {
            unite(a, b);
        }
        if (L == R) {
            auto [type, a, b] = queries[L];
            if (type == 0) {
                ans.push_back(find(a) == find(b));
            } else if (type == 1) {
                ans.push_back(bipartite(a));
            } else if (type == 2) {
                ans.push_back(all_bipartite);
            }
        } else {
            int mid = (L + R) / 2;
            dfs(2 * u, L, mid, ans);
            dfs(2 * u + 1, mid + 1, R, ans);
        }
        rollback();
    }
    vector<int> solve() {
        lazy.assign(4 * time, {});
        map<pair<int, int>, int> edges;
        for (int i = 0; i < time; i++) {
            auto [type, a, b] = queries[i];
            if (type == 3) {
                edges[{a, b}] = i;
            } else if (type == 4) {
                update(edges[{a, b}], i, {a, b}, 1, 0, time - 1);
                edges.erase({a, b});
            }
        }
        for (auto [k, v] : edges) {
            update(v, time - 1, k, 1, 0, time - 1);
        }
        vector<int> ans;
        dfs(1, 0, time - 1, ans);
        return ans;
    }
};
```

4.2 Fenwick Tree

4.2.1 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` substitue o valor da posição i do vetor por d.

Codigo: fenwick_tree.cpp

```cpp
template <typename T> struct FenwickTree {
    int n;
    vector<T> bit, arr;
    FenwickTree(int _n = 0) : n(_n), bit(n), arr(n) { }
    FenwickTree(vector<T> &v) : n(int(v.size())), bit(n), arr(v) {
        for (int i = 0; i < n; i++) {
            bit[i] = arr[i];
        }
        for (int i = 0; i < n; i++) {
            int j = i | (i + 1);
            if (j < n) {
```

```
                bit[j] = bit[j] + bit[i];
            }
        }
    }
    T pref(int x) {
        T res = T();
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1) {
            res = res + bit[i];
        }
        return res;
    }
    T query(int l, int r) {
        if (l == 0) {
            return pref(r);
        }
        return pref(r) - pref(l - 1);
    }
    void update(int x, T d) {
        for (int i = x; i < n; i = i | (i + 1)) {
            bit[i] = bit[i] + d;
        }
        arr[x] = arr[x] + d;
    }
    void updateSet(int i, T d) {
        // funciona pra fenwick de soma
        update(i, d - arr[i]);
        arr[i] = d;
    }
};
```

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

Codigo: kd_fenwick_tree.cpp

```
const int MAX = 20;
long long tree[MAX][MAX][MAX][MAX]; // insira o numero de dimensoes aqui

long long query(vector<int> s, int pos = 0) { // s eh a coordenada
    long long sum = 0;
    while (s[pos] >= 0) {
        if (pos < (int)s.size() - 1) {
            sum += query(s, pos + 1);
        } else {
            sum += tree[s[0]][s[1]][s[2]][s[3]];
            // atualizar se mexer no numero de dimensoes
        }
        s[pos] = (s[pos] & (s[pos] + 1)) - 1;
    }
    return sum;
}

void update(vector<int> s, int v, int pos = 0) {
    while (s[pos] < MAX) {
        if (pos < (int)s.size() - 1) {
            update(s, v, pos + 1);
        } else {
            tree[s[0]][s[1]][s[2]][s[3]] += v;
            // atualizar se mexer no numero de dimensoes
        }
        s[pos] |= s[pos] + 1;
    }
}
```

4.3 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`

Codigo: interval_tree.cpp

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

```
using namespace __gnu_pbds;

struct interval {
    long long lo, hi, id;
    bool operator<(const interval &i) const {
        return tuple(lo, hi, id) < tuple(i.lo, i.hi, i.id);
    }
};

const long long INF = 1e18;

template <class CNI, class NI, class Cmp_Fn, class Allocator>
struct intervals_node_update {
    typedef long long metadata_type;
    int sz = 0;
    virtual CNI node_begin() const = 0;
    virtual CNI node_end() const = 0;
    inline vector<int> overlaps(const long long l, const long long r) {
        queue<CNI> q;
        q.push(node_begin());
        vector<int> vec;
        while (!q.empty()) {
            CNI it = q.front();
            q.pop();
            if (it == node_end()) {
                continue;
            }
            if (r >= (*it)->lo && l <= (*it)->hi) {
                vec.push_back((*it)->id);
            }
            CNI l_it = it.get_l_child();
            long long l_max = (l_it == node_end()) ? -INF : l_it.get_metadata();
            if (l_max >= l) {
                q.push(l_it);
            }
            if ((*it)->lo <= r) {
                q.push(it.get_r_child());
            }
        }
        return vec;
    }
    inline void operator()(NI it, CNI end_it) {
        const long long l_max =
            (it.get_l_child() == end_it) ? -INF : it.get_l_child().get_metadata();
        const long long r_max =
            (it.get_r_child() == end_it) ? -INF : it.get_r_child().get_metadata();
        const_cast<long long &>(it.get_metadata()) = max((*it)->hi, max(l_max, r_max));
    }
};
typedef tree<interval, null_type, less<interval>, rb_tree_tag, intervals_node_update>
    interval_tree;
```

4.4 LiChao Tree

Uma árvore de funções. Retorna o $f(x)$ máximo em um ponto x.

Para retornar o minimo deve-se inserir o negativo da função ($g(x) = -ax - b$) 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, a complexidade de consulta e inserção de funções é $\mathcal{O}(log(L))$.

Codigo: lichao_tree.cpp

```
template <ll MINL = ll(-1e9 - 5), ll MAXR = ll(1e9 + 5)> struct LichaoTree {
    const ll INF = ll(2e18) + 10;
    struct Line {
        ll a, b;
        Line(ll a_ = 0, ll b_ = -INF) : a(a_), b(b_) { }
        ll operator()(ll x) { return a * x + b; }
    };
    vector<Line> tree;
    vector<int> L, R;

    int newnode() {
        tree.push_back(Line());
        L.push_back(-1);
        R.push_back(-1);
        return int(tree.size()) - 1;
    }

    LichaoTree() { newnode(); }

    int le(int u) {
```

```
        if (L[u] == -1) {
            L[u] = newnode();
        }
        return L[u];
    }

    int ri(int u) {
        if (R[u] == -1) {
            R[u] = newnode();
        }
        return R[u];
    }

    void insert(Line line, int n = 0, ll l = MINL, ll r = MAXR) {
        ll mid = (l + r) / 2;
        bool bl = line(l) > tree[n](l);
        bool bm = line(mid) > tree[n](mid);
        bool br = line(r) > tree[n](r);
        if (bm) {
            swap(tree[n], line);
        }
        if (line.b == -INF) {
            return;
        }
        if (bl != bm) {
            insert(line, le(n), l, mid - 1);
        } else if (br != bm) {
            insert(line, ri(n), mid + 1, r);
        }
    }

    ll query(int x, int n = 0, ll l = MINL, ll r = MAXR) {
        if (tree[n](x) == -INF || (l > r))
            return -INF;
        if (l == r) {
            return tree[n](x);
        }
        ll mid = (l + r) / 2;
        if (x < mid) {
            return max(tree[n](x), query(x, le(n), l, mid - 1));
        } else {
            return max(tree[n](x), query(x, ri(n), mid + 1, r));
        }
    }
};
```

4.5 Merge Sort Tree

4.5.1 Merge Sort Tree

Árvore muito semelhante a uma Segment Tree, mas ao invés de armazenar um valor em cada nó, armazena um vetor ordenado. Permite realizar consultas do tipo: `count(L, R, A, B)` que retorna quantos elementos no intervalo $[L, R]$ estão no intervalo $[A, B]$ em $\mathcal{O}(\log^2 n)$. Em outras palavras, count(L, R, A, B) retorna quantos elementos x existem no intervalo $[L, R]$ tal que $A \leq x \leq B$.

Codigo: mergesort_tree.cpp

```
template <typename T = int> struct MergeSortTree {
    vector<vector<T>> tree;
    int n;
    int le(int u) { return u << 1; }
    int ri(int u) { return u << 1 | 1; }
    void build(int u, int l, int r, const vector<T> &a) {
        tree[u] = vector<T>(r - l + 1);
        if (l == r) {
            tree[u][0] = a[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(le(u), l, mid, a);
        build(ri(u), mid + 1, r, a);
        merge(tree[le(u)].begin(),
              tree[le(u)].end(),
              tree[ri(u)].begin(),
              tree[ri(u)].end(),
              tree[u].begin());
    }

    void build(const vector<T> &a) { // para construir com vector
        n = (int)a.size();
        tree.assign(4 * n, vector<T>());
        build(1, 0, n - 1, a);
    }
    void build(T *bg, T *en) { // para construir com array de C
        build(vector<T>(bg, en));
    }

    int count(int u, int l, int r, int L, int R, int a, int b) {
```

```cpp
        if (l > R || r < L || a > b)
            return 0;
        if (l >= L && r <= R) {
            int ub =
                (int)(upper_bound(tree[u].begin(), tree[u].end(), b) - begin(tree[u]));
            int lb = (int)(upper_bound(tree[u].begin(), tree[u].end(), a - 1) -
                           begin(tree[u]));
            return (ub - lb);
        }
        int mid = (l + r) >> 1;
        return count(le(u), l, mid, L, R, a, b) + count(ri(u), mid + 1, r, L, R, a, b);
    }
    int count(int l, int r, int a, int b) { return count(1, 0, n - 1, l, r, a, b); }
    int less(int l, int r, int k) { return count(l, r, tree[1][0], k - 1); }
    int kth(int l, int r, int k) {
        int L = 0, R = n - 1;
        int ans = -1;
        while (L <= R) {
            int mid = (L + R) >> 1;
            if (count(l, r, tree[1][0], tree[1][mid]) > k) {
                ans = mid;
                R = mid - 1;
            } else {
                L = mid + 1;
            }
        }
        return tree[1][ans];
    }
};
```

4.5.2 Merge Sort Tree Update

Merge Sort Tree com updates pontuais. O update é $\mathcal{O}(\log^2 n)$ e a query é $\mathcal{O}(\log^2 n)$, ambos com constante alta.

Codigo: mergesort_tree_update.cpp

```cpp
template <typename T = int> struct MergeSortTree {
    vector<ordered_set<pair<T, int>>> tree;
    vector<T> v;
    int n;
    int le(int u) { return u << 1; }
    int ri(int u) { return u << 1 | 1; }
    void build(int u, int l, int r, const vector<T> &a) {
        if (l == r) {
            tree[u].insert({a[l], l});
            return;
        }
        int mid = (l + r) >> 1;
        build(le(u), l, mid, a);
        build(ri(u), mid + 1, r, a);
        for (auto x : tree[le(u)]) {
            tree[u].insert(x);
        }
        for (auto x : tree[ri(u)]) {
            tree[u].insert(x);
        }
    }

    void build(const vector<T> &a) { // para construir com vector
        n = (int)a.size();
        v = a;
        tree.assign(4 * n, ordered_set<pair<T, int>>());
        build(1, 0, n - 1, a);
    }
    void build(T *bg, T *en) { // para construir com array de C
        build(vector<T>(bg, en));
    }

    int count(int u, int l, int r, int L, int R, int a, int b) {
        if (l > R || r < L || a > b)
            return 0;
        if (l >= L && r <= R) {
            int ub = (int)tree[u].order_of_key({b + 1, INT_MIN});
            int lb = (int)tree[u].order_of_key({a, INT_MIN});
            return (ub - lb);
        }
        int mid = (l + r) >> 1;
        return count(le(u), l, mid, L, R, a, b) + count(ri(u), mid + 1, r, L, R, a, b);
    }
    int count(int l, int r, int a, int b) { return count(1, 0, n - 1, l, r, a, b); }
    int less(int l, int r, int k) { return count(l, r, tree[1].begin()->first, k - 1); }
    void update(int u, int l, int r, int i, T x) {
        tree[u].erase({v[i], i});
        if (l == r) {
            v[i] = x;
        } else {
            int mid = (l + r) >> 1;
            if (i <= mid) {
```

```
                update(le(u), l, mid, i, x);
            } else {
                update(ri(u), mid + 1, r, i, x);
            }
        }
        tree[u].insert({v[i], i});
    }
    void update(int i, T x) { update(1, 0, n - 1, i, x); }
};
```

4.6 Operation Queue

Fila que armazena o resultado do operatório 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 operatório 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.

Codigo: op_queue.cpp

```
template <typename T, auto OP> struct op_queue : queue<T> {
    op_stack<T, OP> st1, st2;
    T get() {
        if (st1.empty()) {
            return st2.get();
        }
        if (st2.empty()) {
            return st1.get();
        }
        return OP(st1.get(), st2.get());
    }
    void add(T element) {
        this->push(element);
        st1.add(element);
    }
    void remove() {
        if (st2.empty()) {
            while (!st1.empty()) {
                st2.add(st1.top());
                st1.remove();
            }
        }
        st2.remove();
        this->pop();
    }
};
```

4.7 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.

Codigo: op_stack.cpp

```
template <typename T, auto OP> struct op_stack : stack<T> {
    stack<T> st;
    T get() { return st.top(); }
    void add(T element) {
        this->push(element);
        st.push(st.empty() ? element : OP(element, st.top()));
    }
    void remove() {
        st.pop();
        this->pop();
    }
};
```

4.8 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 set trocando 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`:

```cpp
ordered_set<int> X;
X.insert(1);
X.insert(2);
X.insert(4);
X.insert(8);
X.insert(16);
cout << *X.find_by_order(1) << endl; // 2
cout << *X.find_by_order(2) << endl; // 4
cout << *X.find_by_order(4) << endl; // 16
cout << (end(X) == X.find_by_order(5)) << endl; // true
cout << X.order_of_key(-5) << endl; // 0
cout << X.order_of_key(1) << endl; // 0
cout << X.order_of_key(3) << endl; // 2
cout << X.order_of_key(4) << endl; // 2
cout << X.order_of_key(400) << endl; // 5
```

Exemplo de uso do `ordered_map`:

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

Codigo: ordered_set.cpp

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

using namespace __gnu_pbds;

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

template <typename T, typename U>
using ordered_map = tree<T, U, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
```

4.9 Segment Tree

4.9.1 Segment Tree

Implementação padrão de Segment Tree, suporta operações de consulta em intervalo e update pontual. Está implementada para soma, mas pode ser facilmente modificada para outras operações. A construção é $\mathcal{O}(n)$ e as operações de consulta e update são $\mathcal{O}(log(n))$.

Codigo: seg_tree.cpp

```cpp
struct SegTree {
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;

    int n;
    vector<ll> t;

    void build(int u, int l, int r, const vector<ll> &v) {
        if (l == r) {
            t[u] = v[l];
        } else {
            int mid = (l + r) >> 1;
            build(u << 1, l, mid, v);
            build(u << 1 | 1, mid + 1, r, v);
            t[u] = merge(t[u << 1], t[u << 1 | 1]);
        }
    }

    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n << 2, neutral);
    }

    void build(const vector<ll> &v) { // pra construir com vector
        n = int(v.size());
        t.assign(n << 2, neutral);
        build(1, 0, n - 1, v);
    }
    void build(ll *bg, ll *en) { // pra construir com array de C
        build(vector<ll>(bg, en));
    }

    ll query(int u, int l, int r, int L, int R) {
        if (l > R || r < L) {
            return neutral;
        }
        if (l >= L && r <= R) {
            return t[u];
        }
        int mid = (l + r) >> 1;
        ll ql = query(u << 1, l, mid, L, R);
        ll qr = query(u << 1 | 1, mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(int l, int r) { return query(1, 0, n - 1, l, r); }

    void update(int u, int l, int r, int i, ll x) {
        if (l == r) {
            t[u] += x; // soma
            // t[u] = x; // substitui
        } else {
            int mid = (l + r) >> 1;
            if (i <= mid) {
                update(u << 1, l, mid, i, x);
            } else {
                update(u << 1 | 1, mid + 1, r, i, x);
            }
            t[u] = merge(t[u << 1], t[u << 1 | 1]);
        }
    }
    void update(int i, ll x) { update(1, 0, n - 1, i, x); }
};
```

4.9.2 Segment Tree 2D

Segment Tree em 2 dimensões, suporta operações de update pontual e consulta em intervalo. A construção é $\mathcal{O}(n \cdot m)$ e as operações de consulta e update são $\mathcal{O}(log(n) \cdot log(m))$.

Codigo: seg_tree_2d.cpp

```cpp
const int MAX = 2505;

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

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

```cpp
int ri(int x) { return 2 * x + 2; }

void build_y(int nx, int lx, int rx, int ny, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx) {
            tree[nx][ny] = mat[lx][ly];
        } else {
            tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
        }
    } else {
        int my = (ly + ry) / 2;
        build_y(nx, lx, rx, le(ny), ly, my);
        build_y(nx, lx, rx, ri(ny), my + 1, ry);
        tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
    }
}
void build_x(int nx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x(le(nx), lx, mx);
        build_x(ri(nx), mx + 1, rx);
    }
    build_y(nx, lx, rx, 0, 0, m - 1);
}
void build() { build_x(0, 0, n - 1); }

void update_y(int nx, int lx, int rx, int ny, int ly, int ry, int x, int y, int v) {
    if (ly == ry) {
        if (lx == rx) {
            tree[nx][ny] = v;
        } else {
            tree[nx][ny] = tree[le(nx)][ny] + tree[ri(nx)][ny];
        }
    } else {
        int my = (ly + ry) / 2;
        if (y <= my) {
            update_y(nx, lx, rx, le(ny), ly, my, x, y, v);
        } else {
            update_y(nx, lx, rx, ri(ny), my + 1, ry, x, y, v);
        }
        tree[nx][ny] = tree[nx][le(ny)] + tree[nx][ri(ny)];
    }
}
void update_x(int nx, int lx, int rx, int x, int y, int v) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        if (x <= mx) {
            update_x(le(nx), lx, mx, x, y, v);
        } else {
            update_x(ri(nx), mx + 1, rx, x, y, v);
        }
    }
    update_y(nx, lx, rx, 0, 0, m - 1, x, y, v);
}
void update(int x, int y, int v) { update_x(0, 0, n - 1, x, y, v); }

int sum_y(int nx, int ny, int ly, int ry, int qly, int qry) {
    if (ry < qly || ly > qry) {
        return 0;
    }
    if (qly <= ly && ry <= qry) {
        return tree[nx][ny];
    }
    int my = (ly + ry) / 2;
    return sum_y(nx, le(ny), ly, my, qly, qry) + sum_y(nx, ri(ny), my + 1, ry, qly, qry);
}
int sum_x(int nx, int lx, int rx, int qlx, int qrx, int qly, int qry) {
    if (rx < qlx || lx > qrx) {
        return 0;
    }
    if (qlx <= lx && rx <= qrx) {
        return sum_y(nx, 0, 0, m - 1, qly, qry);
    }
    int mx = (lx + rx) / 2;
    return sum_x(le(nx), lx, mx, qlx, qrx, qly, qry) +
           sum_x(ri(nx), mx + 1, rx, qlx, qrx, qly, qry);
}
int sum(int lx, int rx, int ly, int ry) { return sum_x(0, 0, n - 1, lx, rx, ly, ry); }
```

4.9.3 Segment Tree Beats Max And Sum Update

Segment Tree que suporta update de maximo, update de soma e query de soma. Utiliza uma fila de lazy para diferenciar os updates. A construção é $\mathcal{O}(n)$ e as operações de consulta e update são $\mathcal{O}(log(n))$.

Codigo: seg_tree_beats_max_and_sum_update.cpp

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

#define ll long long
```

```
#define INF 1e9
#define fi first
#define se second

typedef pair<int, int> ii;

struct Node {
    int m1 = INF, m2 = INF, cont = 0;
    ll soma = 0;
    queue<ii> lazy;

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

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

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

int n, q;
vector<Node> tree;

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

void push(int n, int esq, int dir) {
    while (!tree[n].lazy.empty()) {
        ii p = tree[n].lazy.front();
        tree[n].lazy.pop();
        int op = p.fi, v = p.se;
        if (op == 0) {
            if (v <= tree[n].m1) {
                continue;
            }
            tree[n].soma += (ll)abs(tree[n].m1 - v) * tree[n].cont;
            tree[n].m1 = v;
            if (esq != dir) {
                tree[le(n)].lazy.push({0, v});
                tree[ri(n)].lazy.push({0, v});
            }
        } else if (op == 1) {
            tree[n].soma += v * (dir - esq + 1);
            tree[n].m1 += v;
            tree[n].m2 += v;
            if (esq != dir) {
                tree[le(n)].lazy.push({1, v});
                tree[ri(n)].lazy.push({1, v});
            }
        }
    }
}

void build(int n, int esq, int dir, vector<int> &v) {
    if (esq == dir) {
        tree[n].set(v[esq]);
    } else {
        int mid = (esq + dir) / 2;
        build(le(n), esq, mid, v);
        build(ri(n), mid + 1, dir, v);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void build(vector<int> &v) { build(0, 0, n - 1, v); }

// ai = max(ai, mi) em [l, r]
void update(int n, int esq, int dir, int l, int r, int mi) {
    push(n, esq, dir);
    if (esq > r || dir < l || mi <= tree[n].m1) {
        return;
    }
    if (l <= esq && dir <= r && mi < tree[n].m2) {
        tree[n].soma += (ll)abs(tree[n].m1 - mi) * tree[n].cont;
        tree[n].m1 = mi;
        if (esq != dir) {
            tree[le(n)].lazy.push({0, mi});
            tree[ri(n)].lazy.push({0, mi});
        }
```

```
    } else {
        int mid = (esq + dir) / 2;
        update(le(n), esq, mid, l, r, mi);
        update(ri(n), mid + 1, dir, l, r, mi);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void update(int l, int r, int mi) { update(0, 0, n - 1, l, r, mi); }

// soma v em [l, r]
void upsoma(int n, int esq, int dir, int l, int r, int v) {
    push(n, esq, dir);
    if (esq > r || dir < l) {
        return;
    }
    if (l <= esq && dir <= r) {
        tree[n].soma += v * (dir - esq + 1);
        tree[n].m1 += v;
        tree[n].m2 += v;
        if (esq != dir) {
            tree[le(n)].lazy.push({1, v});
            tree[ri(n)].lazy.push({1, v});
        }
    } else {
        int mid = (esq + dir) / 2;
        upsoma(le(n), esq, mid, l, r, v);
        upsoma(ri(n), mid + 1, dir, l, r, v);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void upsoma(int l, int r, int v) { upsoma(0, 0, n - 1, l, r, v); }

// soma de [l, r]
int query(int n, int esq, int dir, int l, int r) {
    push(n, esq, dir);
    if (esq > r || dir < l) {
        return 0;
    }
    if (l <= esq && dir <= r) {
        return tree[n].soma;
    }
    int mid = (esq + dir) / 2;
    return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir, l, r);
}
int query(int l, int r) { return query(0, 0, n - 1, l, r); }

int main() {
    cin >> n;
    tree.assign(4 * n, Node());
    build(v);
}
```

4.9.4 Segment Tree Beats Max Update

Segment Tree que suporta update de maximo e query de soma. A construção é $\mathcal{O}(n)$ e as operações de consulta e update são $\mathcal{O}(log(n))$.

Codigo: seg_tree_beats.cpp

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

#define ll long long
#define INF 1e9

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

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

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

```cpp
    }

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

int n, q;
vector<Node> tree;

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

void push(int n, int esq, int dir) {
    if (tree[n].lazy <= tree[n].m1) {
        return;
    }
    tree[n].soma += (ll)abs(tree[n].m1 - tree[n].lazy) * tree[n].cont;
    tree[n].m1 = tree[n].lazy;
    if (esq != dir) {
        tree[le(n)].lazy = max(tree[le(n)].lazy, tree[n].lazy);
        tree[ri(n)].lazy = max(tree[ri(n)].lazy, tree[n].lazy);
    }
    tree[n].lazy = 0;
}

void build(int n, int esq, int dir, vector<int> &v) {
    if (esq == dir) {
        tree[n].set(v[esq]);
    } else {
        int mid = (esq + dir) / 2;
        build(le(n), esq, mid, v);
        build(ri(n), mid + 1, dir, v);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void build(vector<int> &v) { build(0, 0, n - 1, v); }

// ai = max(ai, mi) em [l, r]
void update(int n, int esq, int dir, int l, int r, int mi) {
    push(n, esq, dir);
    if (esq > r || dir < l || mi <= tree[n].m1) {
        return;
    }
    if (l <= esq && dir <= r && mi < tree[n].m2) {
        tree[n].lazy = mi;
        push(n, esq, dir);
    } else {
        int mid = (esq + dir) / 2;
        update(le(n), esq, mid, l, r, mi);
        update(ri(n), mid + 1, dir, l, r, mi);
        tree[n].merge(tree[le(n)], tree[ri(n)]);
    }
}
void update(int l, int r, int mi) { update(0, 0, n - 1, l, r, mi); }

// soma de [l, r]
int query(int n, int esq, int dir, int l, int r) {
    push(n, esq, dir);
    if (esq > r || dir < l) {
        return 0;
    }
    if (l <= esq && dir <= r) {
        return tree[n].soma;
    }
    int mid = (esq + dir) / 2;
    return query(le(n), esq, mid, l, r) + query(ri(n), mid + 1, dir, l, r);
}
int query(int l, int r) { return query(0, 0, n - 1, l, r); }

int main() {
    cin >> n;
    tree.assign(4 * n, Node());
}
```

4.9.5 Segment Tree Esparsa

Segment Tree Esparsa, ou seja, não armazena todos os nós da árvore, apenas os necessários, dessa forma ela suporta operações em intervalos arbitrários. A construção é $\mathcal{O}(1)$ e as operações de consulta e update são $\mathcal{O}(log(L))$, onde L é o tamanho do intervalo. A implementação suporta operações de consulta em intervalo e update pontual. Está implementada para soma, mas pode ser facilmente modificada para outras operações.

Para usar, declarar `SegTree<L, R> st` para suportar updates e queries em posições de `L` a `R`. `L` e `R` podem inclusive ser negativos.

Codigo: seg_tree_sparse.cpp

```cpp
template <ll MINL = (ll)-1e9 - 5, ll MAXR = (ll)1e9 + 5> struct SegTree {
    ll merge(ll a, ll b) { return a + b; }
```

```
    const ll neutral = 0;

    vector<ll> t;
    vector<int> Lc, Rc;

    inline int newnode() {
        t.push_back(neutral);
        Lc.push_back(-1);
        Rc.push_back(-1);
        return (int)t.size() - 1;
    }

    inline int le(int u) {
        if (Lc[u] == -1) {
            Lc[u] = newnode();
        }
        return Lc[u];
    }

    inline int ri(int u) {
        if (Rc[u] == -1) {
            Rc[u] = newnode();
        }
        return Rc[u];
    }

    SegTree() { newnode(); }

    ll query(int u, ll l, ll r, ll L, ll R) {
        if (l > R || r < L) {
            return neutral;
        }
        if (l >= L && r <= R) {
            return t[u];
        }
        ll mid = l + (r - l) / 2;
        ll ql = query(le(u), l, mid, L, R);
        ll qr = query(ri(u), mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(ll l, ll r) { return query(0, MINL, MAXR, l, r); }

    void update(int u, ll l, ll r, ll i, ll x) {
        if (l == r) {
            t[u] += x; // soma
            // t[u] = x; // substitui
            return;
        }
        ll mid = l + (r - l) / 2;
        if (i <= mid) {
            update(le(u), l, mid, i, x);
        } else {
            update(ri(u), mid + 1, r, i, x);
        }
        t[u] = merge(t[le(u)], t[ri(u)]);
    }
    void update(ll i, ll x) { update(0, MINL, MAXR, i, x); }
};
```

4.9.6 Segment Tree Kadane

Implementação de uma Segment Tree que suporta update pontual e query de soma máxima de um subarray em um intervalo. A construção é $\mathcal{O}(n)$ e as operações de consulta e update são $\mathcal{O}(log(n))$.

É uma Seg Tree normal, a magia está na função `merge` que é a função que computa a resposta do nodo atual. A ideia do `merge` da Seg Tree de Kadane de combinar respostas e informações já computadas dos filhos é muito útil e pode ser aplicada em muitos problemas.

Obs: não considera o subarray vazio como resposta.

Codigo: seg_tree_kadane.cpp

```
struct SegTree {
    struct node {
        ll sum, pref, suf, ans;
    };
    const node neutral = {0, 0, 0, 0};
    node merge(const node &a, const node &b) {
        return {a.sum + b.sum,
                max(a.pref, a.sum + b.pref),
                max(b.suf, b.sum + a.suf),
                max({a.ans, b.ans, a.suf + b.pref})};
    }

    int n;
    vector<node> t;

    void build(int u, int l, int r, const vector<ll> &v) {
```

```cpp
        if (l == r) {
            t[u] = {v[l], v[l], v[l], v[l]};
        } else {
            int mid = (l + r) >> 1;
            build(u << 1, l, mid, v);
            build(u << 1 | 1, mid + 1, r, v);
            t[u] = merge(t[u << 1], t[u << 1 | 1]);
        }
    }

    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n << 2, neutral);
    }

    void build(const vector<ll> &v) { // pra construir com vector
        n = int(v.size());
        t.assign(n << 2, neutral);
        build(1, 0, n - 1, v);
    }
    void build(ll *bg, ll *en) { // pra construir com array de C
        build(vector<ll>(bg, en));
    }

    node query(int u, int l, int r, int L, int R) {
        if (l > R || r < L) {
            return neutral;
        }
        if (l >= L && r <= R) {
            return t[u];
        }
        int mid = (l + r) >> 1;
        node ql = query(u << 1, l, mid, L, R);
        node qr = query(u << 1 | 1, mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(int l, int r) { return query(1, 0, n - 1, l, r).ans; }

    void update(int u, int l, int r, int i, ll x) {
        if (l == r) {
            t[u] = {x, x, x, x};
        } else {
            int mid = (l + r) >> 1;
            if (i <= mid) {
                update(u << 1, l, mid, i, x);
            } else {
                update(u << 1 | 1, mid + 1, r, i, x);
            }
            t[u] = merge(t[u << 1], t[u << 1 | 1]);
        }
    }
    void update(int i, ll x) { update(1, 0, n - 1, i, x); }
};
```

4.9.7 Segment Tree Lazy

Lazy Propagation é uma técnica para updatar a Segment Tree que te permite fazer updates em intervalos, não necessariamente pontuais. Esta implementação responde consultas de soma em intervalo e updates de soma ou atribuição em intervalo, veja o método `update`.

A construção é $\mathcal{O}(n)$ e as operações de consulta e update são $\mathcal{O}(log(n))$.

Codigo: seg_tree_lazy.cpp

```cpp
struct SegTree {
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;

    int n;
    vector<ll> t, lazy;
    vector<bool> replace;

    void push(int u, int l, int r) {
        if (replace[u]) {
            t[u] = lazy[u] * (r - l + 1);
            if (l != r) {
                lazy[u << 1] = lazy[u];
                lazy[u << 1 | 1] = lazy[u];
                replace[u << 1] = replace[u];
                replace[u << 1 | 1] = replace[u];
            }
        } else if (lazy[u] != 0) {
            t[u] += lazy[u] * (r - l + 1);
            if (l != r) {
                lazy[u << 1] += lazy[u];
                lazy[u << 1 | 1] += lazy[u];
            }
        }
```

```
        replace[u] = false;
        lazy[u] = 0;
    }

    void build(int u, int l, int r, const vector<ll> &v) {
        if (l == r) {
            t[u] = v[l];
        } else {
            int mid = (l + r) / 2;
            build(u << 1, l, mid, v);
            build(u << 1 | 1, mid + 1, r, v);
            t[u] = merge(t[u << 1], t[u << 1 | 1]);
        }
    }

    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n << 2, neutral);
        lazy.assign(n << 2, 0);
        replace.assign(n << 2, false);
    }

    void build(const vector<ll> &v) { // pra construir com vector
        n = (int)v.size();
        t.assign(n << 2, neutral);
        lazy.assign(n << 2, 0);
        replace.assign(n << 2, false);
        build(1, 0, n - 1, v);
    }
    void build(ll *bg, ll *en) { // pra construir com array de C
        build(vector<ll>(bg, en));
    }

    ll query(int u, int l, int r, int L, int R) {
        push(u, l, r);
        if (l > R || r < L) {
            return neutral;
        }
        if (l >= L && r <= R) {
            return t[u];
        }
        int mid = (l + r) >> 1;
        ll ql = query(u << 1, l, mid, L, R);
        ll qr = query(u << 1 | 1, mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(int l, int r) { return query(1, 0, n - 1, l, r); }

    void update(int u, int l, int r, int L, int R, ll val, bool repl) {
        push(u, l, r);
        if (l > R || r < L) {
            return;
        }
        if (l >= L && r <= R) {
            lazy[u] = val;
            replace[u] = repl;
            push(u, l, r);
        } else {
            int mid = (l + r) >> 1;
            update(u << 1, l, mid, L, R, val, repl);
            update(u << 1 | 1, mid + 1, r, L, R, val, repl);
            t[u] = merge(t[u << 1], t[u << 1 | 1]);
        }
    }
    void update(int l, int r, ll val, bool repl = false) {
        update(1, 0, n - 1, l, r, val, repl);
    }
};
```

4.9.8 Segment Tree Lazy Esparsa

Segment Tree com Lazy Propagation e Esparsa. Está implementada com update de soma em range e atribuição em range, e query de soma em range. Construção em $\mathcal{O}(1)$ e operações de update e query em $\mathcal{O}(\log(L))$, onde L é o tamanho do intervalo.

Codigo: seg_tree_sparse_lazy.cpp

```
template <ll MINL = (ll)-1e9 - 5, ll MAXR = (ll)1e9 + 5> struct SegTree {
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;

    vector<ll> t, lazy;
    vector<int> Lc, Rc;
    vector<bool> replace;

    inline int newnode() {
        t.push_back(neutral);
        Lc.push_back(-1);
        Rc.push_back(-1);
```

```cpp
        lazy.push_back(0);
        replace.push_back(false);
        return (int)t.size() - 1;
    }

    inline int le(int u) {
        if (Lc[u] == -1) {
            Lc[u] = newnode();
        }
        return Lc[u];
    }

    inline int ri(int u) {
        if (Rc[u] == -1) {
            Rc[u] = newnode();
        }
        return Rc[u];
    }

    SegTree() { newnode(); }

    void push(int u, ll l, ll r) {
        if (replace[u]) {
            t[u] = lazy[u] * (r - l + 1);
            if (l != r) {
                lazy[le(u)] = lazy[u];
                lazy[ri(u)] = lazy[u];
                replace[le(u)] = replace[u];
                replace[ri(u)] = replace[u];
            }
        } else if (lazy[u] != 0) {
            t[u] += lazy[u] * (r - l + 1);
            if (l != r) {
                lazy[le(u)] += lazy[u];
                lazy[ri(u)] += lazy[u];
            }
        }
        replace[u] = false;
        lazy[u] = 0;
    }

    ll query(int u, ll l, ll r, ll L, ll R) {
        push(u, l, r);
        debug(u, l, r, L, R, t[u]);
        if (l > R || r < L) {
            return neutral;
        }
        if (l >= L && r <= R) {
            return t[u];
        }
        ll mid = l + (r - l) / 2;
        ll ql = query(le(u), l, mid, L, R);
        ll qr = query(ri(u), mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(ll l, ll r) { return query(0, MINL, MAXR, l, r); }

    void update(int u, ll l, ll r, ll L, ll R, ll val, bool repl) {
        push(u, l, r);
        if (l > R || r < L) {
            return;
        }
        if (l >= L && r <= R) {
            lazy[u] = val;
            replace[u] = repl;
            push(u, l, r);
        } else {
            ll mid = l + (r - l) / 2;
            update(le(u), l, mid, L, R, val, repl);
            update(ri(u), mid + 1, r, L, R, val, repl);
            t[u] = merge(t[le(u)], t[ri(u)]);
        }
    }
    void update(ll l, ll r, ll val, bool repl = false) {
        update(0, MINL, MAXR, l, r, val, repl);
    }
};
```

4.9.9 Segment Tree Persisente

Uma Seg Tree Esparsa, só que com persistência, ou seja, pode voltar para qualquer estado anterior da árvore, antes de qualquer modificação.

Os métodos `query` e `update` agora recebem um parâmetro a mais, que é o índice da root (versão da árvore) que se deja modificar.

O vetor `roots` guarda na posição `i` a root da árvore após o `i`-ésimo update.

Codigo: seg_tree_persistent.cpp

```cpp
template <ll MINL = (ll)-1e9 - 5, ll MAXR = (ll)1e9 + 5> struct SegTree {
```

```cpp
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;

    vector<ll> t;
    vector<int> Lc, Rc, roots;

    inline int newnode() {
        t.push_back(neutral);
        Lc.push_back(-1);
        Rc.push_back(-1);
        return (int)t.size() - 1;
    }

    inline int le(int u) {
        if (Lc[u] == -1) {
            Lc[u] = newnode();
        }
        return Lc[u];
    }

    inline int ri(int u) {
        if (Rc[u] == -1) {
            Rc[u] = newnode();
        }
        return Rc[u];
    }

    SegTree() { roots.push_back(newnode()); }

    ll query(int u, ll l, ll r, ll L, ll R) {
        if (l > R || r < L) {
            return neutral;
        }
        if (l >= L && r <= R) {
            return t[u];
        }
        ll mid = l + (r - l) / 2;
        ll ql = query(le(u), l, mid, L, R);
        ll qr = query(ri(u), mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(ll l, ll r, int root = -1) {
        if (root == -1)
            root = roots.back();
        debug(root, MINL, MAXR, l, r);
        return query(root, MINL, MAXR, l, r);
    }

    void update(int u, int old, ll l, ll r, ll i, ll x) {
        if (l == r) {
            t[u] = x; // substitui
            // t[u] += x; // soma
            return;
        }
        ll mid = l + (r - l) / 2;
        if (i <= mid) {
            Rc[u] = ri(old);
            update(le(u), le(old), l, mid, i, x);
        } else {
            Lc[u] = le(old);
            update(ri(u), ri(old), mid + 1, r, i, x);
        }
        t[u] = merge(t[le(u)], t[ri(u)]);
    }
    int update(ll i, ll x, int root = -1) {
        int new_root = newnode();
        if (root == -1)
            root = roots.back();
        update(new_root, root, MINL, MAXR, i, x);
        roots.push_back(new_root);
        return roots.back();
    }
    int copy_root(int root) {
        int new_root = newnode();
        Lc[new_root] = le(root);
        Rc[new_root] = ri(root);
        roots.push_back(new_root);
        return roots.back();
    }
};
```

4.10 Sparse Table

4.10.1 Disjoint Sparse Table

Uma Sparse Table melhorada, construção ainda em $\mathcal{O}(n \log n)$, mas agora suporta queries de **qualquer** operação associativa em $\mathcal{O}(1)$, não precisando mais ser idempotente.

Codigo: dst.cpp

```cpp
struct DisjointSparseTable {
    int n, LG;
    vector<vector<ll>> st;
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;
    void build(const vector<ll> &v) {
        int sz = (int)v.size();
        n = 1, LG = 1;
        while (n < sz) {
            n <<= 1, LG++;
        }
        st = vector<vector<ll>>(LG, vector<ll>(n));
        for (int i = 0; i < n; i++) {
            st[0][i] = i < sz ? v[i] : neutral;
        }
        for (int i = 1; i < LG - 1; i++) {
            for (int j = (1 << i); j < n; j += (1 << (i + 1))) {
                st[i][j] = st[0][j];
                st[i][j - 1] = st[0][j - 1];
                for (int k = 1; k < (1 << i); k++) {
                    st[i][j + k] = merge(st[i][j + k - 1], st[0][j + k]);
                    st[i][j - 1 - k] = merge(st[0][j - k - 1], st[i][j - k]);
                }
            }
        }
    }
    void build(ll *bg, ll *en) { build(vector<ll>(bg, en)); }
    ll query(int l, int r) {
        if (l == r) {
            return st[0][l];
        }
        int i = 31 - __builtin_clz(l ^ r);
        return merge(st[i][l], st[i][r]);
    }
} dst;
```

4.10.2 Sparse Table

Precomputa em $\mathcal{O}(n \log n)$ uma tabela que permite responder consultas de mínimo/máximo em intervalos em $\mathcal{O}(1)$.

A implementação atual é para mínimo, mas pode ser facilmente modificada para máximo ou outras operações.

A restrição é de que a operação deve ser associativa e idempotente (ou seja, $f(x, x) = x$).

Exemplos de operações idempotentes: `min`, `max`, `gcd`, `lcm`.

Exemplos de operações não idempotentes: `soma`, `xor`, `produto`.

Obs: não suporta updates.

Codigo: sparse_table.cpp

```cpp
struct SparseTable {
    int n, LG;
    vector<vector<ll>> st;
    ll merge(ll a, ll b) { return min(a, b); }
    const ll neutral = 1e18;
    void build(const vector<ll> &v) {
        n = (int)v.size();
        LG = 32 - __builtin_clz(n);
        st = vector<vector<ll>>(LG, vector<ll>(n));
        for (int i = 0; i < n; i++) {
            st[0][i] = v[i];
        }
        for (int i = 0; i < LG - 1; i++) {
            for (int j = 0; j + (1 << i) < n; j++) {
                st[i + 1][j] = merge(st[i][j], st[i][j + (1 << i)]);
            }
        }
    }
    void build(ll *bg, ll *en) { build(vector<ll>(bg, en)); }
    ll query(int l, int r) {
        if (l > r)
            return neutral;
        int i = 31 - __builtin_clz(r - l + 1);
        return merge(st[i][l], st[i][r - (1 << i) + 1]);
    }
};
```

Capítulo 5

Grafos

5.1 2 SAT

Resolve problema do 2-SAT.

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

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.

Codigo: 2_sat.cpp

```cpp
struct sat2 {
    int n;
    vector<vector<int>> g, gt;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;

    // number of variables
    sat2(int _n) {
        n = 2 * (_n + 5);
        g.assign(n, vector<int>());
        gt.assign(n, vector<int>());
    }
    void add_edge(int v, int u, bool v_sign, bool u_sign) {
        g[2 * v + v_sign].push_back(2 * u + !u_sign);
        g[2 * u + u_sign].push_back(2 * v + !v_sign);
        gt[2 * u + !u_sign].push_back(2 * v + v_sign);
        gt[2 * v + !v_sign].push_back(2 * u + u_sign);
    }
    void dfs1(int v) {
        used[v] = true;
        for (int u : g[v]) {
            if (!used[u]) {
                dfs1(u);
            }
        }
        order.push_back(v);
    }
    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : gt[v]) {
            if (comp[u] == -1) {
                dfs2(u, cl);
            }
        }
    }
    bool solve() {
        order.clear();
        used.assign(n, false);
        for (int i = 0; i < n; ++i) {
            if (!used[i]) {
                dfs1(i);
            }
        }
```

```cpp

        comp.assign(n, -1);
        for (int i = 0, j = 0; i < n; ++i) {
            int v = order[n - i - 1];
            if (comp[v] == -1) {
                dfs2(v, j++);
            }
        }

        assignment.assign(n / 2, false);
        for (int i = 0; i < n; i += 2) {
            if (comp[i] == comp[i + 1]) {
                return false;
            }
            assignment[i / 2] = comp[i] > comp[i + 1];
        }
        return true;
    }
};
```

5.2 Binary Lifting

5.2.1 Binary Lifting LCA

Usa uma matriz para precomputar os ancestrais de um nodo, em que `up[u][i]` é o 2^i-ésimo ancestral de `u`. A construção é $\mathcal{O}(n \log n)$, e é possível consultar pelo k-ésimo ancestral de um nodo e pelo **LCA** de dois nodos em $\mathcal{O}(\log n)$.

LCA: Lowest Common Ancestor, o LCA de dois nodos u e v é o nodo mais profundo que é ancestral de ambos.

Codigo: binary_lifting_lca.cpp

```cpp
struct BinaryLifting {
    vector<vector<int>> adj, up;
    vector<int> tin, tout;
    int N, LG, t;

    void dfs(int u, int p = -1) {
        tin[u] = t++;
        for (int i = 0; i < LG - 1; i++) {
            up[u][i + 1] = up[up[u][i]][i];
        }
        for (int v : adj[u])
            if (v != p) {
                up[v][0] = u;
                dfs(v, u);
            }
        tout[u] = t++;
    }

    void build(int root, vector<vector<int>> adj2) {
        t = 1;
        N = (int)adj2.size();
        LG = 32 - __builtin_clz(N);
        adj = adj2;
        tin = tout = vector<int>(N);
        up = vector(N, vector<int>(LG));
        up[root][0] = root;
        dfs(root);
    }

    bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }

    int lca(int u, int v) {
        if (ancestor(u, v)) {
            return u;
        }
        if (ancestor(v, u)) {
            return v;
        }
        for (int i = LG - 1; i >= 0; i--) {
            if (!ancestor(up[u][i], v)) {
                u = up[u][i];
            }
        }
        return up[u][0];
    }

    int kth(int u, int k) {
        for (int i = 0; i < LG; i++) {
            if (k & (1 << i)) {
                u = up[u][i];
            }
        }
        return u;
    }
```

```
} bl;
```

5.2.2 Binary Lifting Query

Binary Lifting em que, além de queries de ancestrais, podemos fazer queries em caminhos. Seja $f(u,v)$ uma função que retorna algo sobre o caminho entre u e v, como a soma dos valores dos nodos ou máximo valor do caminho, `st[u][i]` é o valor de $f(par[u], st[u][i])$, em que `st[u][i]` é o 2^i-ésimo ancestral de `u` e `par[u]` é o pai de `u`. A função f deve ser associativa e comutativa.

A construção é $\mathcal{O}(n \log n)$, e é possível consultar em $\mathcal{O}(\log n)$ pelo valor de $f(u,v)$, em que u e v são nodos do grafo, através do método `query`. Também computa LCA e k-ésimo ancestral em $\mathcal{O}(\log n)$.

Obs: os valores precisam estar nos **nodos** e não nas arestas, para valores nas arestas verificar o `Binary Lifting Query Aresta`.

Codigo: binary_lifting_query_nodo.cpp

```cpp
struct BinaryLifting {
    vector<vector<int>> adj, up, st;
    vector<int> val, tin, tout;
    int N, LG, t;

    const int neutral = 0;
    int merge(int l, int r) { return l + r; }

    void dfs(int u, int p = -1) {
        tin[u] = t++;
        for (int i = 0; i < LG - 1; i++) {
            up[u][i + 1] = up[up[u][i]][i];
            st[u][i + 1] = merge(st[u][i], st[up[u][i]][i]);
        }
        for (int v : adj[u])
            if (v != p) {
                up[v][0] = u, st[v][0] = val[u];
                dfs(v, u);
            }
        tout[u] = t++;
    }

    void build(int root, vector<vector<int>> adj2, vector<int> v) {
        t = 1;
        N = (int)adj2.size();
        LG = 32 - __builtin_clz(N);
        adj = adj2;
        val = v;
        tin = tout = vector<int>(N);
        up = st = vector(N, vector<int>(LG, neutral));
        up[root][0] = root;
        st[root][0] = val[root];
        dfs(root);
    }

    bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }

    int query2(int u, int v, bool include_lca) {
        if (ancestor(u, v)) {
            return include_lca ? val[u] : neutral;
        }
        int ans = val[u];
        for (int i = LG - 1; i >= 0; i--) {
            if (!ancestor(up[u][i], v)) {
                ans = merge(ans, st[u][i]);
                u = up[u][i];
            }
        }
        return include_lca ? merge(ans, st[u][0]) : ans;
    }

    int query(int u, int v) {
        if (u == v) {
            return val[u];
        }
        return merge(query2(u, v, 1), query2(v, u, 0));
    }

    int lca(int u, int v) {
        if (ancestor(u, v)) {
            return u;
        }
        if (ancestor(v, u)) {
            return v;
        }
        for (int i = LG - 1; i >= 0; i--) {
            if (!ancestor(up[u][i], v)) {
                u = up[u][i];
            }
```

```
        return up[u][0];
    }

    int kth(int u, int k) {
        for (int i = 0; i < LG; i++) {
            if (k & (1 << i)) {
                u = up[u][i];
            }
        }
        return u;
    }

} bl;
```

5.2.3 Binary Lifting Query 2

Basicamente o mesmo que o anterior, mas esse resolve queries em que o `merge` não é necessariamente **comutativo**. Para fins de exemplo, o código está implementado para resolver queries de Kadane (máximo subarray sum) em caminhos.

Foi usado para passar esse problema:

https://codeforces.com/contest/1843/problem/F2

Codigo: binary_lifting_query_nodo2.cpp

```
struct node {
    int pref, suff, sum, best;
    node() : pref(0), suff(0), sum(0), best(0) { }
    node(int x) : pref(x), suff(x), sum(x), best(x) { }
    node(int a, int b, int c, int d) : pref(a), suff(b), sum(c), best(d) { }
};

node merge(node l, node r) {
    int pref = max(l.pref, l.sum + r.pref);
    int suff = max(r.suff, r.sum + l.suff);
    int sum = l.sum + r.sum;
    int best = max(l.suff + r.pref, max(l.best, r.best));
    return node(pref, suff, sum, best);
}

struct BinaryLifting {
    vector<vector<int>> adj, up;
    vector<int> val, tin, tout;
    vector<vector<node>> st, st2;
    int N, LG, t;

    void build(int u, int p = -1) {
        tin[u] = t++;
        for (int i = 0; i < LG - 1; i++) {
            up[u][i + 1] = up[up[u][i]][i];
            st[u][i + 1] = merge(st[u][i], st[up[u][i]][i]);
            st2[u][i + 1] = merge(st2[up[u][i]][i], st2[u][i]);
        }
        for (int v : adj[u])
            if (v != p) {
                up[v][0] = u;
                st[v][0] = node(val[u]);
                st2[v][0] = node(val[u]);
                build(v, u);
            }
        tout[u] = t++;
    }

    void build(int root, vector<vector<int>> adj2, vector<int> v) {
        t = 1;
        N = (int)adj2.size();
        LG = 32 - __builtin_clz(N);
        adj = adj2;
        val = v;
        tin = tout = vector<int>(N);
        up = vector(N, vector<int>(LG));
        st = st2 = vector(N, vector<node>(LG));
        up[root][0] = root;
        st[root][0] = node(val[root]);
        st2[root][0] = node(val[root]);
        build(root);
    }

    bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }

    node query2(int u, int v, bool include_lca, bool invert) {
        if (ancestor(u, v)) {
            return include_lca ? node(val[u]) : node();
        }
        node ans = node(val[u]);
        for (int i = LG - 1; i >= 0; i--) {
            if (!ancestor(up[u][i], v)) {
                if (invert) {
                    ans = merge(st2[u][i], ans);
                } else {
```

```
                    ans = merge(ans, st[u][i]);
                }
                u = up[u][i];
            }
        }
        return include_lca ? merge(ans, st[u][0]) : ans;
    }

    node query(int u, int v) {
        if (u == v) {
            return node(val[u]);
        }
        node l = query2(u, v, 1, 0);
        node r = query2(v, u, 0, 1);
        return merge(l, r);
    }

    int lca(int u, int v) {
        if (ancestor(u, v)) {
            return u;
        }
        if (ancestor(v, u)) {
            return v;
        }
        for (int i = LG - 1; i >= 0; i--) {
            if (!ancestor(up[u][i], v)) {
                u = up[u][i];
            }
        }
        return up[u][0];
    }

} bl, bl2;
```

5.2.4 Binary Lifting Query Aresta

O mesmo Binary Lifting de query em nodos, porém agora com os valores nas arestas. As complexidades são as mesmas.

Codigo: binary_lifting_query_aresta.cpp

```
struct BinaryLifting {
    vector<vector<pair<int, int>>> adj;
    vector<vector<int>> up, st;
    vector<int> tin, tout;
    int N, LG, t;

    const int neutral = 0;
    int merge(int l, int r) { return l + r; }

    void dfs(int u, int p = -1) {
        tin[u] = t++;
        for (int i = 0; i < LG - 1; i++) {
            up[u][i + 1] = up[up[u][i]][i];
            st[u][i + 1] = merge(st[u][i], st[up[u][i]][i]);
        }
        for (auto [w, v] : adj[u])
            if (v != p) {
                up[v][0] = u, st[v][0] = w;
                dfs(v, u);
            }
        tout[u] = t++;
    }

    void build(int root, vector<vector<pair<int, int>>> adj2) {
        t = 1;
        N = (int)adj2.size();
        LG = 32 - __builtin_clz(N);
        adj = adj2;
        tin = tout = vector<int>(N);
        up = st = vector(N, vector<int>(LG, neutral));
        up[root][0] = root;
        dfs(root);
    }

    bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }

    int query2(int u, int v) {
        if (ancestor(u, v)) {
            return neutral;
        }
        int ans = neutral;
        for (int i = LG - 1; i >= 0; i--) {
            if (!ancestor(up[u][i], v)) {
                ans = merge(ans, st[u][i]);
                u = up[u][i];
            }
        }
        return merge(ans, st[u][0]);
```

```
    }

    int query(int u, int v) {
        if (u == v) {
            return neutral;
#warning TRATAR ESSE CASO ACIMA
        }
        return merge(query2(u, v), query2(v, u));
    }
} bl;
```

5.3 Bridge

Algoritmo que acha pontes utilizando uma dfs

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

Codigo: find_bridges.cpp

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int u, int p = -1) {
    visited[u] = true;
    tin[u] = low[u] = timer++;
    for (int v : adj[u]) {
        if (v == p) {
            continue;
        }
        if (visited[v]) {
            low[u] = min(low[u], tin[v]);
        } else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u]) {
                // edge UV is a bridge
                // do_something(u, v)
            }
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
}
```

5.4 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

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

```cpp

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

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

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

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

    int bfs() {
        vistoken++;
        queue<int> fila;
        fila.push(s);
        vector<int> pego(n, -1);
        while (!fila.empty()) {
            int u = fila.front();
            if (u == t) {
                break;
            }
            fila.pop();
            visto[u] = vistoken;
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) {
                    continue;
                }
                int v = edges[id].v;
                if (visto[v] == -1) {
                    continue;
                }
                fila.push(v);
                pego[v] = id;
            }
        }
        if (pego[t] == -1) {
            return 0;
        }
        long long f = INF;
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            f = min(f, edges[id].cap - edges[id].flow);
        }
        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
            edges[id].flow += f;
            edges[id ^ 1].flow -= f;
        }
        return f;
    }

    long long flow() {
        long long maxflow = 0;
        while (long long f = bfs()) {
            maxflow += f;
        }
        return maxflow;
    }
};
```

Codigo: MinCostMaxFlow.cpp

```cpp
struct MinCostMaxFlow {
    int n, s, t, m = 0;
    ll maxflow = 0, mincost = 0;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;

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

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

    bool spfa() {
        vector<int> pego(n, -1);
        vector<ll> dis(n, INF);
```

```cpp
        vector<bool> inq(n, false);
        queue<int> fila;
        fila.push(s);
        dis[s] = 0;
        inq[s] = 1;
        while (!fila.empty()) {
            int u = fila.front();
            fila.pop();
            inq[u] = false;
            for (int id : adj[u]) {
                if (edges[id].cap - edges[id].flow < 1) {
                    continue;
                }
                int v = edges[id].v;
                if (dis[v] > dis[u] + edges[id].cost) {
                    dis[v] = dis[u] + edges[id].cost;
                    pego[v] = id;
                    if (!inq[v]) {
                        inq[v] = true;
                        fila.push(v);
                    }
                }
            }
        }

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

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

Codigo: Dinic.cpp

```cpp
typedef long long ll;

const ll INF = 1e18;

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

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

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

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

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

```
        }
        return level[t] != -1;
    }

    ll dfs(int u, ll f) {
        if (f == 0) {
            return 0;
        }
        if (u == t) {
            return f;
        }
        for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
            int id = adj[u][cid];
            int v = edges[id].v;
            if (level[u] + 1 != level[v] || edges[id].cap - edges[id].flow < 1) {
                continue;
            }
            ll tr = dfs(v, min(f, edges[id].cap - edges[id].flow));
            if (tr == 0) {
                continue;
            }
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    ll flow() {
        ll maxflow = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs()) {
                break;
            }
            fill(ptr.begin(), ptr.end(), 0);
            while (ll f = dfs(s, INF)) {
                maxflow += f;
            }
        }
        return maxflow;
    }
};
```

5.5 Graph Center

Encontra o centro e o diâmetro de um grafo

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

Codigo: graph_center.cpp

```
const int INF = 1e9 + 9;

vector<vector<int>> adj;

struct GraphCenter {
    int n, diam = 0;
    vector<int> centros, dist, pai;
    int bfs(int s) {
        queue<int> q;
        q.push(s);
        dist.assign(n + 5, INF);
        pai.assign(n + 5, -1);
        dist[s] = 0;
        int maxidist = 0, maxinode = 0;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            if (dist[u] >= maxidist) {
                maxidist = dist[u], maxinode = u;
            }
            for (int v : adj[u]) {
                if (dist[u] + 1 < dist[v]) {
                    dist[v] = dist[u] + 1;
                    pai[v] = u;
                    q.push(v);
                }
            }
        }
        diam = max(diam, maxidist);
        return maxinode;
    }
    GraphCenter(int st = 0) : n(adj.size()) {
        int d1 = bfs(st);
        int d2 = bfs(d1);
        vector<int> path;
        for (int u = d2; u != -1; u = pai[u]) {
            path.push_back(u);
```

```
        }
        int len = path.size();
        if (len % 2 == 1) {
            centros.push_back(path[len / 2]);
        } else {
            centros.push_back(path[len / 2]);
            centros.push_back(path[len / 2 - 1]);
        }
    }
};
```

5.6 HLD

Técnica utilizada para decompor uma árvore em cadeias, e assim realizar operações de caminho e subárvore em $\mathcal{O}(\log n \cdot g(n))$, onde $g(n)$ é a complexidade da operação. Esta implementação suporta queries de soma e update de soma/atribuição, pois usa a estrutura de dados `Segment Tree Lazy` desse almanaque, fazendo assim com que updates e consultas sejam $\mathcal{O}(\log^2 n)$. A estrutura (bem como a operação feita nela) pode ser facilmente trocada, basta alterar o código da `Segment Tree Lazy`, ou ainda, utilizar outra estrutura de dados, como uma `Sparse Table`, caso você tenha queries de mínimo/máximo sem updates, por exemplo.

A decomposição pode ser feita com os valores estando tanto nos vértices quanto nas arestas, consulte os métodos `build` do código para mais detalhes.

A construção da HLD é feita em $\mathcal{O}(n + b(n))$, onde $b(n)$ é a complexidade de construir a estrutura de dados utilizada.

Codigo: hld.cpp

```
struct HLD {
    int n, t;
    vector<vector<int>> adj;
    vector<int> sz, pos, par, head, who;
    bool e = 0; // flag pra dizer se eh de aresta ou nao
    SegTree seg; // pode usar qualquer estrutura de dados aqui

    void dfs_sz(int u, int p = -1) {
        sz[u] = 1;
        for (int &v : adj[u]) {
            if (v != p) {
                dfs_sz(v, u);
                sz[u] += sz[v];
                if (sz[v] > sz[adj[u][0]] || adj[u][0] == p) {
                    swap(v, adj[u][0]);
                }
            }
        }
    }
    void dfs_hld(int u, int p = -1) {
        who[t] = u;
        pos[u] = t++;
        for (int v : adj[u]) {
            if (v != p) {
                par[v] = u;
                head[v] = (v == adj[u][0] ? head[u] : v);
                dfs_hld(v, u);
            }
        }
    }
    void build_hld(int u) {
        sz = pos = par = head = who = vector<int>(n);
        dfs_sz(u);
        t = 0;
        par[u] = u;
        head[u] = u;
        dfs_hld(u);
    }

    void build(int root, vector<ll> v, vector<vector<int>> adj2) {
        // usar esse build pra iniciar com valores nos nodos
        n = (int)adj2.size();
        adj = adj2;
        build_hld(root);
        vector<ll> aux(n);
        for (int i = 0; i < (int)v.size(); i++) {
            aux[pos[i]] = v[i];
        }
        seg.build(aux);
    }
    void build(int root, vector<vector<int>> adj2) {
        // esse build eh para iniciar vazia
        build(root, vector<ll>(adj2.size(), seg.neutral), adj2);
    }
    void build(int root, vector<tuple<int, int, ll>> edges) {
        // usar esse build se os pesos estiverem nas arestas
        n = (int)edges.size() + 1;
        adj = vector<vector<int>>(n);
```

```
        for (auto [u, v, w] : edges) {
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
        build_hld(root);
        e = 1;
        vector<ll> aux(n);
        for (auto [u, v, w] : edges) {
            if (pos[u] > pos[v]) {
                swap(u, v);
            }
            aux[pos[v]] = w;
        }
        seg.build(aux);
    }

    ll query(int u, int v) {
        if (e && u == v) {
#warning "Tratar esse caso"
            return seg.neutral;
        }
        if (pos[u] > pos[v]) {
            swap(u, v);
        }
        if (head[u] == head[v]) {
            return seg.query(pos[u] + e, pos[v]);
        } else {
            ll qv = seg.query(pos[head[v]], pos[v]);
            ll qu = query(u, par[head[v]]);
            return seg.merge(qu, qv);
        }
    }
    ll query_subtree(int u) {
        if (e && sz[u] == 1) {
            return seg.neutral;
        }
        return seg.query(pos[u] + e, pos[u] + sz[u] - 1);
    }

    void update(int u, int v, ll k, bool replace = false) {
        if (e && u == v) {
            return;
        }
        if (pos[u] > pos[v]) {
            swap(u, v);
        }
        if (head[u] == head[v]) {
            seg.update(pos[u] + e, pos[v], k, replace);
        } else {
            seg.update(pos[head[v]], pos[v], k, replace);
            update(u, par[head[v]], k, replace);
        }
    }
    void update_subtree(int u, ll k, bool replace = false) {
        if (e && sz[u] == 1) {
            return;
        }
        seg.update(pos[u] + e, pos[u] + sz[u] - 1, k, replace);
    }

    int lca(int u, int v) {
        if (pos[u] > pos[v]) {
            swap(u, v);
        }
        return (head[u] == head[v] ? u : lca(u, par[head[v]]));
    }
} hld;
```

5.7 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: $\mathcal{O}(NlogN + NlogM)$

Codigo: inverse_graph.cpp

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

set<int> nodes;
vector<set<int>> adj;

void bfs(int s) {
    queue<int> f;
```

```
    f.push(s);
    nodes.erase(s);
    set<int> aux;
    while (!f.empty()) {
        int x = f.front();
        f.pop();
        for (int y : nodes) {
            if (adj[x].count(y) == 0) {
                aux.insert(y);
            }
        }
        for (int y : aux) {
            f.push(y);
            nodes.erase(y);
        }
        aux.clear();
    }
}
```

5.8 Kruskal

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

Utiliza DSU para construir MST.

- Complexidade de tempo (Construção): $\mathcal{O}(MlogN)$

Codigo: kruskal.cpp

```
struct Edge {
    int u, v, w;
    bool operator<(Edge const &other) { return w < other.w; }
};

vector<Edge> edges, result;
int cost;

struct DSU {
    vector<int> pa, sz;
    DSU(int n) {
        sz.assign(n + 5, 1);
        for (int i = 0; i < n + 5; i++) {
            pa.push_back(i);
        }
    }
    int root(int a) { return pa[a] = (a == pa[a] ? a : root(pa[a])); }
    bool find(int a, int b) { return root(a) == root(b); }
    void uni(int a, int b) {
        int ra = root(a), rb = root(b);
        if (ra == rb) {
            return;
        }
        if (sz[ra] > sz[rb]) {
            swap(ra, rb);
        }
        pa[ra] = rb;
        sz[rb] += sz[ra];
    }
};

void kruskal(int m, int n) {
    DSU dsu(n);

    sort(edges.begin(), edges.end());

    for (Edge e : edges) {
        if (!dsu.find(e.u, e.v)) {
            cost += e.w;
            result.push_back(e); // remove if need only cost
            dsu.uni(e.u, e.v);
        }
    }
}
```

5.9 LCA

Algoritmo de Lowest Common Ancestor usando EulerTour e Sparse Table

Complexidade de tempo:

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

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

Codigo: lca.cpp

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

#define INF 1e9
#define fi first
#define se second

typedef pair<int, int> ii;

vector<int> tin, tout;
vector<vector<int>> adj;
vector<ii> prof;
vector<vector<ii>> st;

int n, timer;

void SparseTable(vector<ii> &v) {
    int n = v.size();
    int e = floor(log2(n));
    st.assign(e + 1, vector<ii>(n));
    for (int i = 0; i < n; i++) {
        st[0][i] = v[i];
    }
    for (int i = 1; i <= e; i++) {
        for (int j = 0; j + (1 << i) <= n; j++) {
            st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
        }
    }
}

void et_dfs(int u, int p, int h) {
    tin[u] = timer++;
    prof.emplace_back(h, u);
    for (int v : adj[u]) {
        if (v != p) {
            et_dfs(v, u, h + 1);
            prof.emplace_back(h, u);
        }
    }
    tout[u] = timer++;
}

void build(int root = 0) {
    tin.assign(n, 0);
    tout.assign(n, 0);
    prof.clear();
    timer = 0;
    et_dfs(root, root, 0);
    SparseTable(prof);
}

int lca(int u, int v) {
    int l = tout[u], r = tin[v];
    if (l > r) {
        swap(l, r);
    }
    int i = floor(log2(r - l + 1));
    return min(st[i][l], st[i][r - (1 << i) + 1]).se;
}

int main() {
    cin >> n;

    adj.assign(n, vector<int>(0));

    for (int i = 0; i < n - 1; i++) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    build();
}
```

5.10 Matching

5.10.1 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

```
const ll INF = 1e18 + 18;

vector<pair<int, int>> result;

ll hungarian(int n, int m, vector<vector<int>> &A) {
    vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
    for (int i = 1; i <= n; i++) {
        p[0] = i;
        int j0 = 0;
        vector<int> minv(m + 1, INF);
        vector<char> used(m + 1, false);
        do {
            used[j0] = true;
            ll i0 = p[j0], delta = INF, j1;
            for (int j = 1; j <= m; j++) {
                if (!used[j]) {
                    int cur = A[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) {
                        minv[j] = cur, way[j] = j0;
                    }
                    if (minv[j] < delta) {
                        delta = minv[j], j1 = j;
                    }
                }
            }
            for (int j = 0; j <= m; j++) {
                if (used[j]) {
                    u[p[j]] += delta, v[j] -= delta;
                } else {
                    minv[j] -= delta;
                }
            }
            j0 = j1;
        } while (p[j0] != 0);
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    for (int i = 1; i <= m; i++) {
        result.emplace_back(p[i], i);
    }
    return -v[0];
}
```

5.11 Shortest Paths

5.11.1 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

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

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

void dk(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dist[i][j] = INF;
        }
```

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

Codigo: dijkstra_1_to_n.cpp

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

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

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

Codigo: dijkstra_1_to_1.cpp

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

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

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

5.11.2 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

```cpp
const int MAX = 1e4 + 4;
const ll INF = 1e18 + 18;
```

```cpp

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

void spfa(int s, int n) {
    fill(dist, dist + n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inq(n, false);
    queue<int> fila;
    fila.push(s);
    inq[s] = true;
    dist[s] = 0;
    while (!fila.empty()) {
        int u = fila.front();
        fila.pop();
        inq[u] = false;
        for (auto [w, v] : adj[u]) {
            ll newd = (dist[u] == -INF ? -INF : max(w + dist[u], -INF));
            if (newd < dist[v]) {
                dist[v] = newd;
                if (!inq[v]) {
                    fila.push(v);
                    inq[v] = true;
                    cnt[v]++;
                    if (cnt[v] > n) { // negative cycle
                        dist[v] = -INF;
                    }
                }
            }
        }
    }
}
```

5.12 Stoer–Wagner Min Cut

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

Codigo: stoer_wagner.cpp

```cpp
const int MAXN = 555, INF = 1e9 + 7;

int n, e, adj[MAXN][MAXN];
vector<int> bestCut;

int mincut() {
    int bestCost = INF;
    vector<int> v[MAXN];
    for (int i = 0; i < n; i++) {
        v[i].assign(1, i);
    }
    int w[MAXN], sel;
    bool exist[MAXN], added[MAXN];
    memset(exist, true, sizeof(exist));
    for (int phase = 0; phase < n - 1; phase++) {
        memset(added, false, sizeof(added));
        memset(w, 0, sizeof(w));
        for (int j = 0, prev; j < n - phase; j++) {
            sel = -1;
            for (int i = 0; i < n; i++) {
                if (exist[i] && !added[i] && (sel == -1 || w[i] > w[sel])) {
                    sel = i;
                }
            }
            if (j == n - phase - 1) {
                if (w[sel] < bestCost) {
                    bestCost = w[sel];
                    bestCut = v[sel];
                }
                v[prev].insert(v[prev].end(), v[sel].begin(), v[sel].end());
                for (int i = 0; i < n; i++) {
                    adj[prev][i] = adj[i][prev] += adj[sel][i];
                }
                exist[sel] = false;
            } else {
                added[sel] = true;
                for (int i = 0; i < n; i++) {
                    w[i] += adj[sel][i];
                }
                prev = sel;
            }
        }
    }
    return bestCost;
```

```
}
```

Capítulo 6

String

6.1 Aho Corasick

Muito parecido com uma Trie, porém muito mais poderoso. O autômato de Aho-Corasick é um autômato finito determinístico que pode ser construído a partir de um conjunto de padrões. Nesse autômato, para qualquer nodo u do autômato e qualquer caractere c do alfabeto, é possível transicionar de u usando o caractere c.

A transição é feita por uma aresta direta de u pra v, se a aresta de u pra v estiver marcada com o caractere c. Se não, a transição de u com o caractere c é a transição de $link(u)$ com o caractere c.

Definição: $link(u)$ é um nodo v, tal que o prefixo do autômato ate v é sufixo de u, e esse prefixo é o maior possível. Ou seja, $link(u)$ é o maior prefixo do autômato que é sufixo de u. Com apenas um padrão inserido, o autômato de Aho-Corasick é a Prefix Function (KMP).

Codigo: aho_corasick.cpp

```cpp
struct AC {
    const int K = 26;
    const char norm = 'a';
    inline int get(int c) { return c - norm; }

    vector<int> link, out_link, par;
    vector<char> pch;
    vector<vector<int>> next;
    vector<bool> out;
    vector<vector<int>> output;

    AC() { node(); }

    int node(int p = -1, char c = -1) {
        link.emplace_back(-1);
        out_link.emplace_back(-1);
        par.emplace_back(p);
        pch.emplace_back(c);
        next.emplace_back(K, -1);
        out.emplace_back(0);
        output.emplace_back();
        return (int)link.size() - 1;
    }

    int T = 0;

    int insert(const string &s) {
        int u = 0;
        for (int i = 0; i < (int)s.size(); i++) {
            auto v = next[u][get(s[i])];
            if (v == -1) {
                next[u][get(s[i])] = v = node(u, s[i]);
            }
            u = v;
        }
        out[u] = true;
        output[u].emplace_back(T);
        return T++;
```

```
    }

    int get_link(int u) {
        if (link[u] == -1) {
            link[u] = par[u] > 0 ? go(get_link(par[u]), pch[u]) : 0;
        }
        return link[u];
    }

    int go(int u, char c) {
        if (next[u][get(c)] == -1) {
            next[u][get(c)] = u ? go(get_link(u), c) : 0;
        }
        return next[u][get(c)];
    }

    int exit(int u) {
        if (out_link[u] == -1) {
            int v = get_link(u);
            out_link[u] = (out[v] || !v) ? v : exit(v);
        }
        return out_link[u];
    }

    bool matched(int u) { return out[u] || exit(u) != 0; }

} aho;
```

6.2 Hashing

6.2.1 Hashing

Hashing polinomial para testar igualdade de strings (ou de vetores). Requer precomputar as potências de um primo, como indicado na função `precalc`. A implementação está com dois MODS e usa a primitiva `Mint`, a escolha de usar apenas um MOD ou não usar o `Mint` vai da sua preferência ou necessidade, se não usar o `Mint`, trate adequadamente as operações com aritmética modular. A construção é $\mathcal{O}(n)$ e a consulta é $\mathcal{O}(1)$.

Obs: lembrar de chamar a função `precalc`!

Exemplo de uso:

```
string s = "abacabab";
Hashing a(s);
cout << (a(0, 1) == a(2, 3)) << endl; // 0
cout << (a(0, 1) == a(4, 5)) << endl; // 1
cout << (a(0, 2) == a(4, 6)) << endl; // 1
cout << (a(0, 3) == a(4, 7)) << endl; // 0
```

Codigo: hashing.cpp

```
const int MOD1 = 998244353;
const int MOD2 = 1e9 + 7;
using mint1 = Mint<MOD1>;
using mint2 = Mint<MOD2>;

struct Hash {
    mint1 h1;
    mint2 h2;
    Hash() { }
    Hash(mint1 _h1, mint2 _h2) : h1(_h1), h2(_h2) { }
    bool operator==(Hash o) const { return h1 == o.h1 && h2 == o.h2; }
    bool operator!=(Hash o) const { return h1 != o.h1 || h2 != o.h2; }
    bool operator<(Hash o) const { return h1 == o.h1 ? h2 < o.h2 : h1 < o.h1; }
    Hash operator+(Hash o) const { return {h1 + o.h1, h2 + o.h2}; }
    Hash operator-(Hash o) const { return {h1 - o.h1, h2 - o.h2}; }
    Hash operator*(Hash o) const { return {h1 * o.h1, h2 * o.h2}; }
    Hash operator/(Hash o) const { return {h1 / o.h1, h2 / o.h2}; }
};

const int PRIME = 1001003; // qualquer primo na ordem do alfabeto
const int MAXN = 1e6 + 5;
Hash PR = {PRIME, PRIME};
Hash invPR = {mint1(1) / PRIME, mint2(1) / PRIME};
Hash pot[MAXN], invpot[MAXN];
void precalc() {
    pot[0] = invpot[0] = Hash(1, 1);
    for (int i = 1; i < MAXN; i++) {
        pot[i] = pot[i - 1] * PR;
        invpot[i] = invpot[i - 1] * invPR;
    }
}

struct Hashing {
    int N;
    vector<Hash> hsh;
    Hashing() { }
```

```cpp
    Hashing(string s) : N(int(s.size())), hsh(N + 1) {
        for (int i = 0; i < N; i++) {
            int c = int(s[i] - 'a');
            hsh[i + 1] = hsh[i] + (pot[i + 1] * Hash(c, c));
        }
    }
    Hash operator()(int l, int r) const { return (hsh[r + 1] - hsh[l]) * invpot[l]; }
};
```

6.2.2 Hashing Dinâmico

Hashing polinomial para testar igualdade de strings (ou de vetores). Requer precomputar as potências de um primo, como indicado na função `precalc`. A implementação está com dois MODS e usa a primitiva `Mint`, a escolha de usar apenas um MOD ou não usar o `Mint` vai da sua preferência ou necessidade, se não usar o `Mint`, trate adequadamente as operações com aritmética modular.

Essa implementação suporta updates pontuais, utilizando-se de uma `Fenwick Tree` para isso. A construção é $\mathcal{O}(n)$, consultas e updates são $\mathcal{O}(\log n)$.

Obs: lembrar de chamar a função `precalc`!

Exemplo de uso:

```cpp
string s = "abacabab";
DynamicHashing a(s);
cout << (a(0, 1) == a(2, 3)) << endl; // 0
cout << (a(0, 1) == a(4, 5)) << endl; // 1
a.update(0, 'c');
cout << (a(0, 1) == a(4, 5)) << endl; // 0
```

Codigo: dynamic_hashing.cpp

```cpp
const int MOD1 = 998244353;
const int MOD2 = 1e9 + 7;
using mint1 = Mint<MOD1>;
using mint2 = Mint<MOD2>;

struct Hash {
    mint1 h1;
    mint2 h2;
    Hash() { }
    Hash(mint1 _h1, mint2 _h2) : h1(_h1), h2(_h2) { }
    bool operator==(Hash o) const { return h1 == o.h1 && h2 == o.h2; }
    bool operator!=(Hash o) const { return h1 != o.h1 || h2 != o.h2; }
    bool operator<(Hash o) const { return h1 == o.h1 ? h2 < o.h2 : h1 < o.h1; }
    Hash operator+(Hash o) const { return {h1 + o.h1, h2 + o.h2}; }
    Hash operator-(Hash o) const { return {h1 - o.h1, h2 - o.h2}; }
    Hash operator*(Hash o) const { return {h1 * o.h1, h2 * o.h2}; }
    Hash operator/(Hash o) const { return {h1 / o.h1, h2 / o.h2}; }
};

const int PRIME = 1001003; // qualquer primo na ordem do alfabeto
const int MAXN = 1e6 + 5;
Hash PR = {PRIME, PRIME};
Hash invPR = {mint1(1) / PRIME, mint2(1) / PRIME};
Hash pot[MAXN], invpot[MAXN];
void precalc() {
    pot[0] = invpot[0] = Hash(1, 1);
    for (int i = 1; i < MAXN; i++) {
        pot[i] = pot[i - 1] * PR;
        invpot[i] = invpot[i - 1] * invPR;
    }
}

struct DynamicHashing {
    int N;
    FenwickTree<Hash> hsh;
    DynamicHashing() { }
    DynamicHashing(string s) : N(int(s.size())) {
        vector<Hash> v(N);
        for (int i = 0; i < N; i++) {
            int c = int(s[i] - 'a');
            v[i] = pot[i + 1] * Hash(c, c);
        }
        hsh = FenwickTree<Hash>(v);
    }
    Hash operator()(int l, int r) { return hsh.query(l, r) * invpot[l]; }
    void update(int i, char ch) {
        int c = int(ch - 'a');
        hsh.updateSet(i, pot[i + 1] * Hash(c, c));
    }
};
```

6.3 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

```cpp
string min_cyclic_shift(string s) {
    s += s;
    int n = s.size();
    int i = 0, ans = 0;
    while (i < n / 2) {
        ans = i;
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k++;
            }
            j++;
        }
        while (i <= k) {
            i += j - k;
        }
    }
    return s.substr(ans, n / 2);
}
```

Codigo: duval.cpp

```cpp
vector<string> duval(string const &s) {
    int n = s.size();
    int i = 0;
    vector<string> factorization;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k++;
            }
            j++;
        }
        while (i <= k) {
            factorization.push_back(s.substr(i, j - k));
            i += j - k;
        }
    }
    return factorization;
}
```

6.4 Manacher

O algoritmo de manacher encontra todos os palíndromos de uma string em $\mathcal{O}(n)$. Para cada centro, ele conta quantos palíndromos de tamanho ímpar e par existem (nos vetores `d1` e `d2` respectivamente). O método `solve` computa os palíndromos e retorna o número de substrings palíndromas. O método `query` retorna se a substring `s[i...j]` é palíndroma em $\mathcal{O}(1)$.

Codigo: manacher.cpp

```cpp
struct Manacher {
    int n;
    ll count;
    vector<int> d1, d2, man;
    ll solve(const string &s) {
        n = int(s.size()), count = 0;
        solve_odd(s);
        solve_even(s);
        man.assign(2 * n - 1, 0);
        for (int i = 0; i < n; i++) {
            man[2 * i] = 2 * d1[i] - 1;
        }
        for (int i = 0; i < n - 1; i++) {
```

```
            man[2 * i + 1] = 2 * d2[i + 1];
        }
        return count;
    }
    void solve_odd(const string &s) {
        d1.assign(n, 0);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
            while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
                k++;
            }
            count += d1[i] = k--;
            if (i + k > r) {
                l = i - k, r = i + k;
            }
        }
    }
    void solve_even(const string &s) {
        d2.assign(n, 0);
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
            while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
                k++;
            }
            count += d2[i] = k--;
            if (i + k > r) {
                l = i - k - 1, r = i + k;
            }
        }
    }
    bool query(int i, int j) {
        assert(man.size());
        return man[i + j] >= j - i + 1;
    }
} mana;
```

6.5 Patricia Tree

Estrutura de dados que armazena strings e permite consultas por prefixo, muito similar a uma Trie. Todas as operações são $\mathcal{O}(|s|)$.

Obs: Não aceita elementos repetidos.

Implementação PB-DS, extremamente curta e confusa:

Exemplo de uso:

```
patricia_tree pat;
pat.insert("exemplo");
pat.erase("exemplo");
pat.find("exemplo") != pat.end(); // verifica existência
auto match = pat.prefix_range("ex"); // pega palavras que começam com "ex"
for (auto it = match.first; it != match.second; ++it); // percorre match
pat.lower_bound("ex"); // menor elemento lexicográfico maior ou igual a "ex"
pat.upper_bound("ex"); // menor elemento lexicográfico maior que "ex"
```

Codigo: patricia_tree.cpp

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

using namespace __gnu_pbds;
typedef trie<string, null_type, trie_string_access_traits<>, pat_trie_tag,
            trie_prefix_search_node_update>
    patricia_tree;
```

6.6 Prefix Function KMP

6.6.1 Automato KMP

O autômato de KMP computa em $\mathcal{O}(|s| \cdot \Sigma)$ a função de transição de uma string, que é definida por:

$$nxt[i][c] = \max\{k : k \leq i \land s[0:k] = s[i-k:i-1]c\} + 1$$

Em outras palavras, $nxt[i][c]$ é o tamanho do maior prefixo de s que é sufixo de $s[0:i-1]c$.

O autômato de KMP é útil para mútiplos pattern matchings, ou seja, dado um padrão t, encontrar todas as ocorrências de t em várias strings $s_1, s_2, \ldots, s_k$, em $\mathcal{O}(|t| + \sum |s_i|)$. O método `matching` faz isso.

Obs: utiliza o código do KMP.

Codigo: aut_kmp.cpp

```cpp
struct AutKMP {
    vector<vector<int>> nxt;
    void setString(string s) {
        s += '#';
        nxt.assign(s.size(), vector<int>(26));
        vector<int> p = pi(s);
        for (int c = 0; c < 26; c++) {
            nxt[0][c] = ('a' + c == s[0]);
        }
        for (int i = 1; i < int(s.size()); i++) {
            for (int c = 0; c < 26; c++) {
                nxt[i][c] = ('a' + c == s[i]) ? i + 1 : nxt[p[i - 1]][c];
            }
        }
    }
    vector<int> matching(string &s, string &t) {
        vector<int> match;
        for (int i = 0, j = 0; i < int(s.size()); i++) {
            j = nxt[j][s[i] - 'a'];
            if (j == int(t.size())) {
                match.push_back(i - j + 1);
            }
        }
        return match;
    }
} aut;
```

6.6.2 KMP

O algoritmo de Knuth-Morris-Pratt (KMP) computa em $\mathcal{O}(|s|)$ a Prefix Function de uma string, cuja definição é dada por:

$$p[i] = \max\{k : k < i \wedge s[0 : k] = s[i - k : i]\} + 1$$

Em outras palavras, $p[i]$ é o tamanho do maior prefixo de s que é sufixo próprio de $s[0 : i]$.

O KMP é útil para pattern matching, ou seja, encontrar todas as ocorrências de uma string t em uma string s, como faz a função `matching` em $O(|s| + |t|)$.

Codigo: kmp.cpp

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

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

6.7 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

```
pair<int, int> busca(string &t, int i, pair<int, int> &range) {
    int esq = range.first, dir = range.second, L = -1, R = -1;
    while (esq <= dir) {
        int mid = (esq + dir) / 2;
        if (s[sa[mid] + i] == t[i]) {
            L = mid;
        }
        if (s[sa[mid] + i] < t[i]) {
            esq = mid + 1;
        } else {
            dir = mid - 1;
        }
    }
    esq = range.first, dir = range.second;
    while (esq <= dir) {
        int mid = (esq + dir) / 2;
        if (s[sa[mid] + i] == t[i]) {
            R = mid;
        }
        if (s[sa[mid] + i] <= t[i]) {
            esq = mid + 1;
        } else {
            dir = mid - 1;
        }
    }
    return {L, R};
}
// count ocurences of s on t
int busca_string(string &t) {
    pair<int, int> range = {0, n - 1};
    for (int i = 0; i < t.size(); i++) {
        range = busca(t, i, range);
        if (range.first == -1) {
            return 0;
        }
    }
    return range.second - range.first + 1;
}
```

Codigo: suffix_array.cpp

```
const int MAX_N = 5e5 + 5;

struct suffix_array {
    string s;
    int n, sum, r, ra[MAX_N], sa[MAX_N], auxra[MAX_N], auxsa[MAX_N], c[MAX_N],
        lcp[MAX_N];
    void counting_sort(int k) {
        memset(c, 0, sizeof(c));
        for (int i = 0; i < n; i++) {
            c[(i + k < n) ? ra[i + k] : 0]++;
        }
        for (int i = sum = 0; i < max(256, n); i++) {
            sum += c[i], c[i] = sum - c[i];
        }
        for (int i = 0; i < n; i++) {
            auxsa[c[sa[i] + k < n ? ra[sa[i] + k] : 0]++] = sa[i];
        }
        for (int i = 0; i < n; i++) {
            sa[i] = auxsa[i];
        }
    }
    void build_sa() {
        for (int k = 1; k < n; k <<= 1) {
            counting_sort(k);
            counting_sort(0);
            auxra[sa[0]] = r = 0;
            for (int i = 1; i < n; i++) {
                auxra[sa[i]] =
                    (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] == ra[sa[i - 1] + k])
                        ? r
                        : ++r;
            }
            for (int i = 0; i < n; i++) {
                ra[i] = auxra[i];
            }
            if (ra[sa[n - 1]] == n - 1) {
                break;
            }
        }
    }
    void build_lcp() {
        for (int i = 0, k = 0; i < n - 1; i++) {
            int j = sa[ra[i] - 1];
            while (s[i + k] == s[j + k]) {
                k++;
            }
            lcp[ra[i]] = k;
            if (k) {
                k--;
            }
        }
```

```
    }
    void set_string(string _s) {
        s = _s + '$';
        n = s.size();
        for (int i = 0; i < n; i++) {
            ra[i] = s[i], sa[i] = i;
        }
        build_sa();
        build_lcp();
        // for (int i = 0; i < n; i++)
        // printf("%2d: %s\n", sa[i], s.c_str() +
        // sa[i]);
    }
    int operator[](int i) { return sa[i]; }
} sa;
```

6.8 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

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

6.9 Z function

O algoritmo abaixo computa o vetor Z de uma string, definido por:

$$Z[i] = \max\{k \geq 0 : s[0, k) = s[i, i + k)\} + 1$$

É muito semelhante ao KMP em termos de aplicações. Usado principalmente para pattern matching.

Codigo: z.cpp

```
vector<int> get_z(string &s) {
    int n = (int)s.size();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; i++) {
        if (i <= r) {
            z[i] = min(r - i + 1, z[i - l]);
        }
        while (i + z[i] < n && s[i + z[i]] == s[z[i]]) {
            z[i]++;
        }
        if (i + z[i] - 1 > r) {
            r = i + z[i] - 1;
            l = i;
        }
    }
    return z;
}

vector<int> matching(string &s, string &t) { // s = texto, t = padrao
```

```
    string k = t + "$" + s;
    vector<int> z = get_z(k), match;
    int n = (int)t.size();
    for (int i = n + 1; i < (int)z.size(); i++) {
        if (z[i] == n) {
            match.push_back(i - n - 1);
        }
    }
    return match;
}
```

Capítulo 7

Paradigmas

7.1 All Submasks

Percorre todas as submáscaras de uma máscara.

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

Codigo: all_submask.cpp

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

7.2 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

```cpp

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

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

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

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

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

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

```cpp

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

} // namespace name
```

7.3 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

```cpp

double eval(double mid) {
    // implement the evaluation
}

double ternary_search(double l, double r) {
    int k = 100;
    while (k--) {
        double step = (l + r) / 3;
        double mid_1 = l + step;
        double mid_2 = r - step;

        // minimizing. To maximize use >= to
        // compare
        if (eval(mid_1) <= eval(mid_2)) {
            r = mid_2;
        } else {
            l = mid_1;
        }
    }
    return l;
}
```

Codigo: busca_ternaria_discreta.cpp

```cpp

long long eval(long long mid) {
    // implement the evaluation
}

long long discrete_ternary_search(long long l, long long r) {
    long long ans = -1;
    r--; // to not space r
```

```cpp
    while (l <= r) {
        long long mid = (l + r) / 2;

        // minimizing. To maximize use >= to
        // compare
        if (eval(mid) <= eval(mid + 1)) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}
```

7.4 Convex Hull Trick

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

Só funciona quando as retas são monotônicas. Caso não 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: $\mathcal{O}(1)$

Codigo: Convex Hull Trick.cpp

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

7.5 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

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

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

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

7.6 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

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

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

Codigo: dc_query_on_demand.cpp

```cpp
namespace DC {
    struct range { // eh preciso definir a forma
                   // de calcular o range
        vi freq;
        ll sum = 0;
        int l = 0, r = -1;
        void back_l(int v) { // Mover o 'l' do range
                             // para a esquerda
            sum += freq[v];
            freq[v]++;
            l--;
        }
        void advance_r(int v) { // Mover o 'r' do range
                                // para a direita
            sum += freq[v];
            freq[v]++;
            r++;
        }
        void advance_l(int v) { // Mover o 'l' do range
                                // para a direita
            freq[v]--;
            sum -= freq[v];
            l++;
        }
        void back_r(int v) { // Mover o 'r' do range
                             // para a esquerda
            freq[v]--;
            sum -= freq[v];
            r--;
        }
        void clear(int n) { // Limpar range
            l = 0;
            r = -1;
            sum = 0;
            freq.assign(n + 5, 0);
        }
    } s;

    vi dp_before, dp_cur;
    void compute(int l, int r, int optl, int optr) {
        if (l > r) {
            return;
        }
        int mid = (l + r) >> 1;
        pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar

        while (s.l < optl) {
            s.advance_l(v[s.l]);
        }
        while (s.l > optl) {
            s.back_l(v[s.l - 1]);
        }
        while (s.r < mid) {
            s.advance_r(v[s.r + 1]);
        }
        while (s.r > mid) {
            s.back_r(v[s.r]);
        }

        vi removed;
        for (int i = optl; i <= min(mid, optr); i++) {
            best =
                min(best,
                    {(i ? dp_before[i - 1] : 0) + s.sum, i}); // min() se quiser minimizar
            removed.push_back(v[s.l]);
            s.advance_l(v[s.l]);
        }
        for (int rem : removed) {
            s.back_l(v[s.l - 1]);
        }

        dp_cur[mid] = best.first;
        int opt = best.second;
        compute(l, mid - 1, optl, opt);
        compute(mid + 1, r, opt, optr);
    }

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

7.7 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:

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

- –

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:

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

Exemplo de DP:

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

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

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

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

Codigo: matrix_exp.cpp

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

#define MOD 1000000007

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

```

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

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

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

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

    mat prod = exp_mod(T, exp);

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

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

7.8 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

```
typedef pair<int, int> ii;
int block_sz; // Better if 'const';

namespace mo {
    struct query {
        int l, r, idx;
        bool operator<(query q) const {
            int _l = l / block_sz;
            int _ql = q.l / block_sz;
            return ii(_l, (_l & 1 ? -r : r)) < ii(_ql, (_ql & 1 ? -q.r : q.r));
        }
    };
    vector<query> queries;

    void build(int n) {
        block_sz = (int)sqrt(n);
        // TODO: initialize data structure
    }
    inline void add_query(int l, int r) {
        queries.push_back({l, r, (int)queries.size()});
    }
    inline void remove(int idx) {
        // TODO: remove value at idx from data
        // structure
```

```
    }
    inline void add(int idx) {
        // TODO: add value at idx from data
        // structure
    }
    inline int get_answer() {
        // TODO: extract the current answer of the
        // data structure
        return 0;
    }

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

};
```

Codigo: mo_update.cpp

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

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

    vector<int> run() {
        vector<int> answers(queries.size());
        sort(queries.begin(), queries.end());
        int L = 0;
        int R = -1;
```

```cpp
        int T = 0;
        for (query q : queries) {
            while (T < q.t) {
                update(L, R, T++);
            }
            while (T > q.t) {
                update(L, R, --T);
            }
            while (L > q.l) {
                add(--L);
            }
            while (R < q.r) {
                add(++R);
            }
            while (L < q.l) {
                remove(L++);
            }
            while (R > q.r) {
                remove(R--);
            }
            answers[q.idx] = get_answer();
        }
        return answers;
    }
};
```

Capítulo 8

Primitivas

8.1 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:

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

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 $b \cdot b^{-1} \equiv 1 \mod m$. 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:

```cpp
const int MOD = 7; // MOD = 7 para fins de exemplo
using mint = Mint<MOD>;
mint a = 4, b = 3;
mint c = a * b; // c == 5
mint d = 1 / a; // d == 2
mint e = a * d // e == 1
a += 2; // a == 6
a += 3; // a == 2
a ^= 5; // a == 4
```

Codigo: mint.cpp

```cpp
template <int MOD> struct Mint {
    using m = Mint;
    int v;
    Mint() : v(0) { }
    Mint(ll val) {
        if (val < -MOD or val >= 2 * MOD) {
            val %= MOD;
        }
        if (val >= MOD) {
            val -= MOD;
        }
        if (val < 0) {
            val += MOD;
        }
        v = (int)val;
    }
    bool operator==(const m &o) const { return v == o.v; }
    bool operator!=(const m &o) const { return v != o.v; }
    bool operator<(const m &o) const { return v < o.v; }
    m pwr(m b, ll e) {
        m res;
```

```cpp
        for (res = 1; e > 0; e >>= 1, b *= b) {
            if (e & 1) {
                res *= b;
            }
        }
        return res;
    }
    m &operator+=(const m &o) {
        v += o.v;
        if (v >= MOD) {
            v -= MOD;
        }
        return *this;
    }
    m &operator-=(const m &o) {
        v -= o.v;
        if (v < 0) {
            v += MOD;
        }
        return *this;
    }
    m &operator*=(const m &o) { return *this = m((ll)v * o.v % MOD); }
    m &operator/=(const m &o) { return *this *= pwr(o, MOD - 2); }
    m &operator^=(ll e) {
        assert(e >= 0);
        return *this = pwr(*this, e);
    }
    friend m operator+(m a, const m &b) { return a += b; }
    friend m operator-(m a, const m &b) { return a -= b; }
    friend m operator*(m a, const m &b) { return a *= b; }
    friend m operator/(m a, const m &b) { return a /= b; }
    friend m operator^(m a, ll e) { return a ^= e; }
    friend ostream &operator<<(ostream &os, const m &a) { return os << a.v; }
    friend istream &operator>>(istream &is, m &a) {
        ll x;
        is >> x, a = m(x);
        return is;
    }
};

const int MOD = 998244353; // o MOD tem que ser primo
using mint = Mint<MOD>;
```

8.2 Ponto 2D

Estrutura que representa um ponto no plano cartesiano em duas dimensões. Suporta operações de soma, subtração, multiplicação por escalar, produto escalar, produto vetorial e distância euclidiana. Pode ser usado também para representar um vetor.

Codigo: point2d.cpp

```cpp
template <typename T> struct point {
    T x, y;
    point(T _x = 0, T _y = 0) : x(_x), y(_y) { }

    using p = point;

    p operator*(const T o) { return p(o * x, o * y); }
    p operator-(const p o) { return p(x - o.x, y - o.y); }
    p operator+(const p o) { return p(x + o.x, y + o.y); }
    T operator*(const p o) { return x * o.x + y * o.y; }
    T operator^(const p o) { return x * o.y - y * o.x; }
    bool operator<(const p o) const { return (x == o.x) ? y < o.y : x < o.x; }
    bool operator==(const p o) const { return (x == o.x) and (y == o.y); }
    bool operator!=(const p o) const { return (x != o.x) or (y != o.y); }

    T dist2(const p o) {
        T dx = x - o.x, dy = y - o.y;
        return dx * dx + dy * dy;
    }

    friend ostream &operator<<(ostream &out, const p &a) {
        return out << "(" << a.x << "," << a.y << ")";
    }
    friend istream &operator>>(istream &in, p &a) { return in >> a.x >> a.y; }
};

using pt = point<ll>;
```

Capítulo 9

Geometria

9.1 Convex Hull

Algoritmo Graham's Scan para encontrar o fecho convexo de um conjunto de pontos em $\mathcal{O}(n \log n)$. Retorna os pontos do fecho convexo em sentido horário.

Definição: o fecho convexo de um conjunto de pontos é o menor polígono convexo que contém todos os pontos do conjunto.

Obs: utiliza a primitiva `Ponto 2D`.

Codigo: convex_hull.cpp

```
bool ccw(pt &p, pt &a, pt &b, bool include_collinear = 0) {
    pt p1 = a - p;
    pt p2 = b - p;
    return include_collinear ? (p2 ^ p1) <= 0 : (p2 ^ p1) < 0;
}

void sort_by_angle(vector<pt> &v) { // sorta o vetor por angulo em relacao ao pivo
    pt p0 = *min_element(begin(v), end(v));
    sort(begin(v), end(v), [&](pt &l, pt &r) { // sorta clockwise
        pt p1 = l - p0;
        pt p2 = r - p0;
        ll c1 = p1 ^ p2;
        return c1 < 0 || ((c1 == 0) && p0.dist2(l) < p0.dist2(r));
    });
}

vector<pt> convex_hull(vector<pt> v, bool include_collinear = 0) {
    int n = size(v);

    sort_by_angle(v);

    if (include_collinear) {
        for (int i = n - 2; i >= 0; i--) { // reverte o ultimo lado do poligono
            if (ccw(v[0], v[n - 1], v[i])) {
                reverse(begin(v) + i + 1, end(v));
                break;
            }
        }
    }

    vector<pt> ch{v[0], v[1]};
    for (int i = 2; i < n; i++) {
        while (ch.size() > 2 &&
                (ccw(ch.end()[-2], ch.end()[-1], v[i], !include_collinear)))
            ch.pop_back();
        ch.emplace_back(v[i]);
    }

    return ch;
}
```

Capítulo 10

Matemática

10.1 Eliminação Gaussiana

10.1.1 Gauss

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

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

Codigo: gauss.cpp

```cpp
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to
                   // be infinity or a big number

int gauss(vector<vector<double>> a, vector<double> &ans) {
    int n = (int)a.size();
    int m = (int)a[0].size() - 1;

    vector<int> where(m, -1);
    for (int col = 0, row = 0; col < m && row < n; ++col) {
        int sel = row;
        for (int i = row; i < n; ++i) {
            if (abs(a[i][col]) > abs(a[sel][col])) {
                sel = i;
            }
        }
        if (abs(a[sel][col]) < EPS) {
            continue;
        }
        for (int i = col; i <= m; ++i) {
            swap(a[sel][i], a[row][i]);
        }
        where[col] = row;

        for (int i = 0; i < n; ++i) {
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j = col; j <= m; ++j) {
                    a[i][j] -= a[row][j] * c;
                }
            }
        }
        ++row;
    }

    ans.assign(m, 0);
    for (int i = 0; i < m; ++i) {
        if (where[i] != -1) {
            ans[i] = a[where[i]][m] / a[where[i]][i];
        }
    }
    for (int i = 0; i < n; ++i) {
        double sum = 0;
        for (int j = 0; j < m; ++j) {
            sum += ans[j] * a[i][j];
        }
```

```
        if (abs(sum - a[i][m]) > EPS) {
            return 0;
        }
    }

    for (int i = 0; i < m; ++i) {
        if (where[i] == -1) {
            return INF;
        }
    }
    return 1;
}
```

10.1.2 Gauss Mod 2

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

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

Codigo: gauss_mod2.cpp

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

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

int gauss(vector<bitset<N>> a, int n, int m, bitset<N> &ans) {
    vector<int> where(m, -1);

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

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

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

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

10.2 Exponenciação Modular Rápida

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

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

Codigo: exp_mod.cpp

```cpp
ll exp_mod(ll base, ll exp) {
    ll b = base, res = 1;
    while (exp) {
        if (exp & 1) {
            res = (res * b) % MOD;
        }
        b = (b * b) % MOD;
        exp /= 2;
    }
    return res;
}
```

10.3 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

```cpp
typedef complex<double> cd;
typedef vector<cd> poly;
const double PI = acos(-1);

void fft(poly &a, bool invert = 0) {
    int n = a.size(), log_n = 0;
    while ((1 << log_n) < n) {
        log_n++;
    }

    for (int i = 1, j = 0; i < n; ++i) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) {
            j -= bit;
        }
        j += bit;
        if (i < j) {
            swap(a[i], a[j]);
        }
    }

    double angle = 2 * PI / n * (invert ? -1 : 1);
    poly root(n / 2);
    for (int i = 0; i < n / 2; ++i) {
        root[i] = cd(cos(angle * i), sin(angle * i));
    }

    for (long long len = 2; len <= n; len <<= 1) {
        long long step = n / len;
        long long aux = len / 2;
        for (long long i = 0; i < n; i += len) {
            for (int j = 0; j < aux; ++j) {
                cd u = a[i + j], v = a[i + j + aux] * root[step * j];
                a[i + j] = u + v;
                a[i + j + aux] = u - v;
            }
        }
    }
    if (invert) {
        for (int i = 0; i < n; ++i) {
            a[i] /= n;
        }
    }
}

vector<long long> convolution(vector<long long> &a, vector<long long> &b) {
    int n = 1, len = a.size() + b.size();
    while (n < len) {
        n <<= 1;
    }
    a.resize(n);
```

```
    b.resize(n);
    poly fft_a(a.begin(), a.end());
    fft(fft_a);
    poly fft_b(b.begin(), b.end());
    fft(fft_b);

    poly c(n);
    for (int i = 0; i < n; ++i) {
        c[i] = fft_a[i] * fft_b[i];
    }
    fft(c, 1);

    vector<long long> res(n);
    for (int i = 0; i < n; ++i) {
        res[i] = round(c[i].real()); // res = c[i].real();
                                     // se for vector de
                                     // double
    }
    // while(size(res) > 1 && res.back() == 0)
    // res.pop_back(); // apenas para quando os
    // zeros direita nao importarem
    return res;
}
```

10.4 Fatoração

Algortimos 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: Fatoraração $\mathcal{O}$(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

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

Codigo: linear_sieve_factorize.cpp

```
namespace sieve {
    const int MAX = 1e4;
    int lp[MAX + 1], factor[MAX + 1];
    vector<int> pr;
    void build() {
```

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

Codigo: pollard_rho.cpp

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

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

Codigo: fast_factorize.cpp

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

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

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

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

Codigo: gcd.cpp

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

Codigo: extended_gcd.cpp

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

Codigo: extended_gcd_recursive.cpp

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

10.6 Inverso Modular

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

O inverso modular de um inteiro a é outro inteiro x tal que $a * x$ 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 Extendido, 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 $(MOD+1)/2$.

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

```cpp
ll inv[MAX];

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

Codigo: modular_inverse_pow.cpp

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

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

void compute_inv() {
    inv[0] = 1;
    for (int i = 1; i < MAX; i++) {
        inv[i] = inv[i - 1] * INVB % MOD;
    }
}
```

Codigo: modular_inverse.cpp

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

Codigo: modular_inverse_coprime.cpp

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

10.7 NTT

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

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](big_convoluti

Codigo: ntt.cpp

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

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

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

void ntt(poly &a, bool invert, int id) {
    ll n = (ll)a.size(), m = mod[id];
    for (ll i = 1, j = 0; i < n; ++i) {
        ll bit = n >> 1;
        for (; j >= bit; bit >>= 1) {
            j -= bit;
        }
        j += bit;
        if (i < j) {
            swap(a[i], a[j]);
        }
    }
    for (ll len = 2, wlen; len <= n; len <<= 1) {
```

```
            wlen = invert ? root_1[id] : root[id];
            for (ll i = len; i < root_pw[id]; i <<= 1) {
                wlen = (wlen * wlen) % m;
            }
            for (ll i = 0; i < n; i += len) {
                ll w = 1;
                for (ll j = 0; j < len / 2; j++) {
                    ll u = a[i + j], v = (a[i + j + len / 2] * w) % m;
                    a[i + j] = (u + v) % m;
                    a[i + j + len / 2] = (u - v + m) % m;
                    w = (w * wlen) % m;
                }
            }
        }
        if (invert) {
            ll inv = modInv(n, m);
            for (ll i = 0; i < n; i++) {
                a[i] = (a[i] * inv) % m;
            }
        }
}

poly convolution(poly a, poly b, int id = 0) {
        ll n = 1LL, len = (1LL + a.size() + b.size());
        while (n < len) {
            n *= 2;
        }
        a.resize(n);
        b.resize(n);
        ntt(a, 0, id);
        ntt(b, 0, id);
        poly answer(n);
        for (ll i = 0; i < n; i++) {
            answer[i] = (a[i] * b[i]);
        }
        ntt(answer, 1, id);
        return answer;
}
```

Codigo: big_convolution.cpp

```

ll mod_mul(ll a, ll b, ll m) { return (__int128)a * b % m; }
ll ext_gcd(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1;
        y = 0;
        return a;
    } else {
        ll g = ext_gcd(b, a % b, y, x);
        y -= a / b * x;
        return g;
    }
}

// convolution mod 1,097,572,091,361,755,137
poly big_convolution(poly a, poly b) {
    poly r0, r1, answer;
    r0 = convolution(a, b, 1);
    r1 = convolution(a, b, 2);

    ll s, r, p = mod[1] * mod[2];
    ext_gcd(mod[1], mod[2], r, s);

    answer.resize(r0.size());
    for (int i = 0; i < (int)answer.size(); i++) {
        answer[i] = (mod_mul((s * mod[2] + p) % p, r0[i], p) +
                     mod_mul((r * mod[1] + p) % p, r1[i], p) + p) %
                    p;
    }
    return answer;
}
```

10.8 Primos

Algortimos 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 até $10^2$4.

- 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

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

Codigo: naive_is_prime.cpp

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

Codigo: miller_rabin.cpp

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

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

bool miller_rabin(ll n) {
    if (n < 2) {
        return false;
    }
    int r = 0;
    ll d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1, ++r;
    }
    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41}) {
        if (n == a) {
            return true;
        }
        if (is_composite(n, a, d, r)) {
            return false;
        }
    }
    return true;
}
```

10.9 Sum of floor (n div i)

Esse código computa, em $\mathcal{O}(\sqrt{n})$, o seguinte somatório:

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

Codigo: sum_of_floor.cpp

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

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

10.10 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.

Codigo: crt.cpp

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

ll crt(vector<ll> rem, vector<ll> mod) {
    int n = rem.size();
    if (n == 0) {
        return 0;
    }
    __int128 ans = rem[0], m = mod[0];
    for (int i = 1; i < n; i++) {
        ll x, y;
        ll g = extended_gcd(mod[i], m, x, y);
        if ((ans - rem[i]) % g != 0) {
            return -1;
        }
        ans = ans + (__int128)1 * (rem[i] - ans) * (m / g) * y;
        m = (__int128)(mod[i] / g) * (m / g) * g;
        ans = (ans % m + m) % m;
    }
    return ans;
}
```

10.11 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

```
vector<int> phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++) {
        phi[i] = i;
    }
    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i) {
                phi[j] -= phi[j] / i;
            }
        }
    }
    return phi;
}
```

Codigo: phi.cpp

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0) {
                n /= i;
            }
            result -= result / i;
        }
    }
    if (n > 1) {
        result -= result / n;
    }
    return result;
}
```