Almanaque de Códigos pra Maratona de Programação

BRUTE UDESC

25 de agosto de 2025

Índice

Capítulo 1

C++

1.1 Compilador

Para compilar um arquivo `.cpp` com o compilador `g++`, usar o comando:

```
g++ -std=c++20 -O2 arquivo.cpp
```

Obs: a flag `-std=c++20` é para usar a versão 20 do C++, os códigos desse Almanaque são testados com essa versão.

Algumas flags úteis para o `g++` são:

- `-O2`: Otimizações de compilação
- `-Wall`: Mostra todos os warnings
- `-Wextra`: Mostra mais warnings
- `-Wconversion`: Mostra warnings para conversões implícitas
- `-fsanitize=address`: Habilita o AddressSanitizer
- `-fsanitize=undefined`: Habilita o UndefinedBehaviorSanitizer

Todas essas flags já estão presente no script 'run' da seção Extra.

1.2 STL (Standard Template Library)

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.

1.2.1 Vector

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

- `vector<int> v(n, x)`: Cria um vetor de inteiros com `n` elementos, todos inicializados com x - $\mathcal{O}(n)$
- `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` - $\mathcal{O}(n)$

1.2.2 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 - $\mathcal{O}(1)$

1.2.3 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 - $\mathcal{O}(1)$

1.2.4 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))`.

1.2.5 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 - $\mathcal{O}(1)$

1.2.6 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 - $\mathcal{O}(1)$

1.2.7 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`.

1.2.8 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 - $\mathcal{O}(1)$

1.2.9 Bitset

Um conjunto de bits, serve para representar máscaras quando um inteiro não é suficiente. Possui operações bitwise otimizadas pelo processador.

- `bitset<N> a`: Cria um bitset de tamanho `N` (`N` deve ser constante) - $\mathcal{O}(1)$
- `a[i]`: Retorna o valor do bit na posição `i` - $\mathcal{O}(1)$
- `a.count()`: Retorna o número de bits 1 no bitset - $\mathcal{O}(N/w)$
- `a.size()`: Retorna o tamanho do bitset - $\mathcal{O}(1)$
- `a.set(i)`: Seta o bit na posição `i` para 1 - $\mathcal{O}(1)$
- `a.set()`: Seta todos os bits para 1 - $\mathcal{O}(N/w)$
- `a.reset(i)`: Seta o bit na posição `i` para 0 - $\mathcal{O}(1)$

- `a.reset()`: Seta todos os bits para 0 - $\mathcal{O}(N/w)$
- `a.flip(i)`: Inverte o bit na posição `i` - $\mathcal{O}(1)$
- `a.flip()`: Inverte todos os bits - $\mathcal{O}(N/w)$
- `a.to_ullong()`: Retorna o valor do bitset como um inteiro - $\mathcal{O}(N/w)$

O bitset também suporta operações como `&`, `|`, `^`, `~`, `<<`, `>>`, entre outros.

OBS: w é o tamanho da palavra do processador, em geral 32 ou 64 bits.

1.2.10 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` - $\mathcal{O}(1)$

1.2.11 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:

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

Lembrete: `v.end()` é um iterador para o elemento seguinte ao último do vetor, então não é um iterador válido.

As funções de vetor em geral são da forma `[L, R)`, ou seja, `L` é incluso e `R` é excluso.

- `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 - $\mathcal{O}(n)$ I
- `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 - $\mathcal{O}(n)$

1.3 Pragmas

Os pragmas são diretivas para o compilador, que podem ser usadas para otimizar o código.

Temos os pragmas de otimização, como por exemplo:

- `#pragma GCC optimize("O2")`: Otimizações de nível 2 (padrão de competições)
- `#pragma GCC optimize("O3")`: Otimizações de nível 3 (seguro para usar)
- `#pragma GCC optimize("Ofast")`: Otimizações agressivas (perigoso!)
- `#pragma GCC optimize("unroll-loops")`: Otimiza os loops mas pode levar a cache misses

E também os pragmas de target, que são usados para otimizar o código para um certo processador:

- `#pragma GCC target("avx2")`: Otimiza instruções para processadores com suporte a AVX2
- `#pragma GCC target("sse4")`: Parecido com o de cima, mas mais antigo
- `#pragma GCC target("popcnt")`: Otimiza o popcount em processadores que suportam
- `#pragma GCC target("lzcnt")`: Otimiza o leading zero count em processadores que suportam
- `#pragma GCC target("bmi")`: Otimiza instruções de bit manipulation em processadores que suportam
- `#pragma GCC target("bmi2")`: Mesmo que o de cima, mas mais recente

Em geral, esses pragmas são usados para otimizar o código em competições, mas é importante usálos com certa sabedoria, em alguns casos eles podem piorar o desempenho do código.

Uma opção relativamente segura de se usar é a seguinte:

```
1  #pragma GCC optimize("O3,unroll-loops")
2  #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```

1.4 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...

Capítulo 2

Teórico

2.1 Definições

Algumas definições e termos importantes:

2.1.1 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 $f(x, x) = x$.

2.1.2 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)*.

2.2 Números primos

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

2.2.1 Primos com truncamento à esquerda

Números primos tais que qualquer sufixo deles é um número primo:

$$33{,}333{,}31$$
$$357{,}686{,}312{,}646{,}216{,}567{,}629{,}137$$

2.2.2 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

2.2.3 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	$1,7 * 10^{38}$

2.3 Operadores lineares

2.3.1 Rotação no sentido anti-horário por θ°

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

2.3.2 Reflexão em relação à reta $y = mx$

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

2.3.3 Inversa de uma matriz 2x2 A

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

2.3.4 Cisalhamento horizontal por K

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

2.3.5 Cisalhamento vertical por K

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

2.3.6 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.

$$C\vec{a}_\beta = \vec{a}$$
$$C^{-1}\vec{a} = \vec{a}_\beta$$
$$C = \begin{bmatrix} b1_x & b2_x \\ b1_y & b2_y \end{bmatrix}$$

2.3.7 Propriedades das operações de matriz

$$(AB)^{-1} = A^{-1}B^{-1}$$
$$(AB)^T = B^T A^T$$
$$(A^{-1})^T = (A^T)^{-1}$$
$$(A+B)^T = A^T + B^T$$
$$\det(A) = \det(A^T)$$
$$\det(AB) = \det(A)\det(B)$$

Seja A uma matriz NxN:

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

2.4 Sequências numéricas

2.4.1 Sequência de Fibonacci

Primeiros termos: **1, 1, 2, 3, 5, 8, 13, 21, 34, ...**

Definição:

$$F_0 = F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Matriz de recorrência:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F_{n-2} \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}$$

2.4.2 Sequência de Catalan

Primeiros termos: **1, 1, 2, 5, 14, 42, 132, 429, 1430, ...**

Definição:

$$C_0 = C_1 = 1$$

$$C_n = \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i}$$

Definição analítica:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Propriedades úteis:

- C_n é o número de árvores binárias com $n+1$ folhas.
- C_n é o número de sequências de parênteses bem formadas com n pares de parênteses.

2.5 Análise combinatória

2.5.1 Fatorial

O fatorial de um número n é o produto de todos os inteiros positivos menores ou iguais a n.

O fatorial conta o número de permutações de n elementos.

$$n! = n \cdot (n-1)!$$

Em particular, $0! = 1$.

2.5.2 Combinação

Conta o número de maneiras de escolher k elementos de um conjunto de n elementos.

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

2.5.3 Arranjo

Conta o número de maneiras de escolher k elementos de um conjunto de n elementos, onde a ordem importa.

$$P(n,k) = \frac{n!}{(n-k)!}$$

2.5.4 Estrelas e barras

Conta o número de maneiras de distribuir n elementos idênticos em k recipientes distintos.

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

2.5.5 Princípio da inclusão-exclusão

O princípio da inclusão-exclusão é uma técnica para contar o número de elementos em uma união de conjuntos.

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1 \le i_1 < i_2 < \ldots < i_k \le n} |A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}| \right)$$

2.5.6 Princípio da casa dos pombos

Se n pombos são colocados em m casas, então pelo menos uma casa terá $\lceil \frac{n}{m} \rceil$ pombos ou mais.

2.6 Teoria dos números

2.6.1 Pequeno teorema de Fermat

Se p é um número primo e a é um inteiro não divisível por p, então $a^{p-1} \equiv 1 \pmod{p}$.

2.6.2 Teorema de Euler

Se m e a são inteiros positivos coprimos, então $a^{\phi(m)} \equiv 1 \pmod{m}$, onde $\phi(m)$ é a função totiente de Euler.

2.6.3 Aritmética modular

Quando estamos trabalhando com aritmética módulo um número p, todos os valores existentes estão entre $[0, p-1]$.

Algumas propriedades e equivalências úteis para usar aritmética modular em código são:

- $(a + b) \mod p \equiv ((a \mod p) + (b \mod p)) \mod p$
- $(a - b) \mod p \equiv ((a \mod p) - (b \mod p)) \mod p$
  - Note que o resultado pode ser negativo, nesse é necessário adicionar p ao resultado. De forma geral, geralmente fazemos $(a - b + p) \mod p$ (assumindo que a e b já estão no intervalo $[0, p - 1]$).
- $(a \cdot b) \mod p \equiv ((a \mod p) \cdot (b \mod p)) \mod p$
- $a^b \mod p \equiv ((a \mod p)^b) \mod p$
- $a^b \mod p \equiv ((a \mod p)^{(b \mod \phi(p))}) \mod p$ se a e p são coprimos

Capítulo 3

Extra

3.1 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();
}
```

3.2 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
template <typename T, typename U>
ostream &operator<<(ostream &os, const pair<T, U> &p) { // opcional
    os << "(" << p.first << ", " << p.second << ")";
    return os;
}
template <typename T>
ostream &operator<<(ostream &os, const vector<T> &v) { // opcional
    os << "{";
    int n = (int)v.size();
    for (int i = 0; i < n; i++) {
        os << v[i];
        if (i < n - 1) os << ", ";
    }
    os << "}";
    return os;
}

void _print() { }
template <typename T, typename... U>
void _print(T a, U... b) {
    if (sizeof...(b)) {
        cerr << a << ", ";
        _print(b...);
    } else cerr << a;
}
#ifdef BRUTE
#define debug(x...) cerr << "[" << #x << "] = [", _print(x), cerr << "]" << endl
#else
#define debug(...)
```

```
#endif
```

3.3 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

```
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); }
```

3.4 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

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

3.5 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

```
#!/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
```

3.6 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);
    }
};
```

3.7 Vim

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

Codigo: vimrc

```
set nu ai si cindent et ts=4 sw=4 so=10 nosm undofile

inoremap {} {}<left><return><up><end><return>
" remap de chaves

au BufReadPost * if line("'\"") > 0 && line("'\"") <= line("$") | exe "normal! g'\"" |
      endif
" volta pro lugar onde estava quando saiu do arquivo
```

Capítulo 4

Matemática

4.1 Convolução

4.1.1 AND Convolution

Calcula o vetor C a partir de A e B onde $C[i] = \sum_{(j \wedge k) = i} A[j] \cdot B[k]$ em $\mathcal{O}(N \cdot \log N)$.

Obs: $\wedge$ representa o bitwise and.

Codigo: and_convolution.cpp

```
vector<mint> and_convolution(vector<mint> A, vector<mint> B) {
    int n = (int)max(A.size(), B.size());
    int N = 0;
    while ((1 << N) < n) N++;
    A.resize(1 << N);
    B.resize(1 << N);
    vector<mint> C(1 << N);
    for (int j = 0; j < N; j++) {
        for (int i = (1 << N) - 1; i >= 0; i--) {
            if (~i >> j & 1) {
                A[i] += A[i | (1 << j)];
                B[i] += B[i | (1 << j)];
            }
        }
    }
    for (int i = 0; i < 1 << N; i++) C[i] = A[i] * B[i];
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < 1 << N; i++)
            if (~i >> j & 1) C[i] -= C[i | (1 << j)];
    }
    return C;
}
```

4.1.2 GCD Convolution

Calcula o vetor C a partir de A e B onde $C[i] = \sum_{gcd(j,k) = i} A[j] \cdot B[k]$ em $\mathcal{O}(N \cdot \log N)$.

Codigo: gcd_convolution.cpp

```
vector<mint> gcd_convolution(vector<mint> A, vector<mint> B) {
    int N = (int)max(A.size(), B.size());
    A.resize(N + 1);
    B.resize(N + 1);
    vector<mint> C(N + 1);
    for (int i = 1; i <= N; i++) {
        mint a = 0;
        mint b = 0;
        for (int j = i; j <= N; j += i) {
            a += A[j];
            b += B[j];
        }
        C[i] = a * b;
    }
    for (int i = N; i >= 1; i--)
```

```
        for (int j = 2 * i; j <= N; j += i) C[i] -= C[j];
    return C;
}
```

4.1.3 LCM Convolution

Calcula o vetor C a partir de A e B $C[i] = \sum_{lcm(j,k)=i} A[j] \cdot B[k]$ em $\mathcal{O}(N \cdot \log N)$.

Codigo: lcm_convolution.cpp

```
vector<mint> lcm_convolution(vector<mint> A, vector<mint> B) {
    int N = (int)max(A.size(), B.size());
    A.resize(N + 1);
    B.resize(N + 1);
    vector<mint> C(N + 1), a(N + 1), b(N + 1);
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j += i) {
            a[j] += A[i];
            b[j] += B[i];
        }
        C[i] = a[i] * b[i];
    }
    for (int i = 1; i <= N; i++)
        for (int j = 2 * i; j <= N; j += i) C[j] -= C[i];
    return C;
}
```

4.1.4 OR Convolution

Calcula o vetor C a partir de A e B tal que $C[i] = \sum_{(j \| k)=i} A[j] \cdot B[k]$ em $\mathcal{O}(N \cdot \log N)$

Codigo: or_convolution.cpp

```
vector<mint> or_convolution(vector<mint> A, vector<mint> B) {
    int n = (int)max(A.size(), B.size());
    int N = 0;
    while ((1 << N) < n) N++;
    A.resize(1 << N);
    B.resize(1 << N);
    vector<mint> C(1 << N);
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < 1 << N; i++) {
            if (i >> j & 1) {
                A[i] += A[i ^ (1 << j)];
                B[i] += B[i ^ (1 << j)];
            }
        }
    }
    for (int i = 0; i < 1 << N; i++) C[i] = A[i] * B[i];
    for (int j = N - 1; j >= 0; j--) {
        for (int i = (1 << N) - 1; i >= 0; i--)
            if (i >> j & 1) C[i] -= C[i ^ (1 << j)];
    }
    return C;
}
```

4.1.5 Subset Convolution

Calcula o vetor C a partir de A e B tal que $C[i] = \sum_{(j \| k)=i, (j \wedge k)=0} A[j] \cdot B[k]$ em $\mathcal{O}(N \cdot \log^2 N)$

Obs: $\wedge$ representa o bitwise and e $\|$ representa o bitwise or.

Codigo: subset_convolution.cpp

```
vector<mint> subset_convolution(vector<mint> A, vector<mint> B) {
    int n = int(max(A.size(), B.size()));
    int N = 0;
    while ((1 << N) < n) N++;
    A.resize(1 << N), B.resize(1 << N);
    vector a(1 << N, vector<mint>(N + 1)), b(1 << N, vector<mint>(N + 1));
    for (int i = 0; i < 1 << N; i++) {
        int popcnt = __builtin_popcount(i);
        a[i][popcnt] = A[i];
        b[i][popcnt] = B[i];
    }
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < 1 << N; i++) {
            if (~i >> j & 1) continue;
            for (int popcnt = 0; popcnt <= N; popcnt++) {
```

```
                a[i][popcnt] += a[i ^ (1 << j)][popcnt];
                b[i][popcnt] += b[i ^ (1 << j)][popcnt];
            }
        }
    }
    vector c(1 << N, vector<mint>(N + 1));
    for (int i = 0; i < 1 << N; i++) {
        for (int j = 0; j <= N; j++)
            for (int k = 0; k + j <= N; k++) c[i][j + k] += a[i][j] * b[i][k];
    }
    for (int j = N - 1; j >= 0; j--) {
        for (int i = (1 << N) - 1; i >= 0; i--) {
            if (~i >> j & 1) continue;
            for (int popcnt = 0; popcnt <= N; popcnt++)
                c[i][popcnt] -= c[i ^ (1 << j)][popcnt];
        }
    }
    vector<mint> ans(1 << N);
    for (int i = 0; i < 1 << N; i++) {
        int popcnt = __builtin_popcount(i);
        ans[i] = c[i][popcnt];
    }
    return ans;
}
```

4.1.6 XOR Convolution

Calcula o vetor C a partir de A e B tal que $C[i] = \sum_{(j \oplus k) = i} A[j] \cdot B[k]$ em $\mathcal{O}(N \cdot \log N)$

Codigo: xor_convolution.cpp

```
vector<mint> xor_convolution(vector<mint> A, vector<mint> B) {
    int n = int(A.size());
    for (int rep = 0; rep < 2; rep++) {
        for (int len = n >> 1; len; len >>= 1) {
            for (int i = 0; i < n; i += len << 1) {
                for (int j = 0; j < len; j++) {
                    int id = i + j;
                    mint x = A[id];
                    mint y = A[id + len];
                    A[id] = x + y;
                    A[id + len] = x - y;
                }
            }
        }
        swap(A, B);
    }
    vector<mint> ans(n);
    for (int i = 0; i < n; i++) ans[i] = A[i] * B[i];
    for (int len = 1; len < n; len <<= 1) {
        for (int i = 0; i < n; i += len << 1) {
            for (int j = 0; j < len; j++) {
                int id = i + j;
                mint x = ans[id];
                mint y = ans[id + len];
                ans[id] = x + y;
                ans[id + len] = x - y;
            }
        }
    }
    return ans;
}

vector<mint> xor_multiply(vector<mint> A, vector<mint> B) {
    int N = 1;
    int n = int(max(A.size(), B.size()));
    while (N < n) N <<= 1;
    A.resize(N);
    B.resize(N);
    auto ans = xor_convolution(A, B);
    for (int i = 0; i < N; i++) ans[i] /= N;
    return ans;
}
```

4.2 Eliminação Gaussiana

4.2.1 Gauss

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes reais, a complexidade é $\mathcal{O}(n^3)$.

A função `gauss` recebe como parâmetros:

- `vector<vector<double>> a`: uma matriz $N \times (M+1)$, onde N é o número de equações e M é o número de variáveis, a última coluna de `a` deve conter o resultado das equações.

- `vector<double> &ans`: um vetor de tamanho M, que será preenchido com a solução do sistema, caso exista.

A função retorna:

- `0`: se o sistema não tem solução.
- `1`: se o sistema tem uma única solução.
- `INF`: se o sistema tem infinitas soluções. Nesse caso, as variáveis em que `where[i] == -1` são as variáveis livres.

Codigo: gauss.cpp

```cpp
const double EPS = 1e-9;
const int INF = 2; // nao tem que ser infinito ou um numero grande
                   // so serve para indicar que tem infinitas solucoes

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;
}
```

4.2.2 Gauss Mod 2

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes em $\mathbb{Z}_2$ (inteiros módulo 2), a complexidade é $\mathcal{O}(n^3/\sqsupseteq)$, onde $\sqsupseteq$ é a palavra do processador (geralmente 32 ou 64 bits, dependendo da arquitetura).

No código, a constante M deve ser definida como o `(número de variáveis + 1)`.

A função `gauss` recebe como parâmetros:

- `vector<bitset<M>> a`: um vector de bitsets, representando as equações do sistema. Cada bitset tem tamanho M, onde o bit j do bitset i representa o coeficiente da variável j na equação i. A última posição do bitset i representa o resultado da equação i.

- `n` e `m`: inteiros representando o número de equações e variáveis, respectivamente.

- `bitset<M> &ans`: um bitset de tamanho M, que será preenchido com a solução do sistema, caso exista.

A função retorna:

- `0`: se o sistema não tem solução.
- `1`: se o sistema tem uma única solução.
- `INF`: se o sistema tem infinitas soluções. Nesse caso, as variáveis em que `where[i] == -1` são as variáveis livres. Note que, pela natureza de $\mathbb{Z}_2$, o sistema não terá de fato infinitas soluções, mas sim 2^L soluções, onde L é o número de variáveis livres.

Codigo: gauss_mod2.cpp

```cpp
const int M = 105;
const int INF = 2; // nao tem que ser infinito ou um numero grande
                   // so serve para indicar que tem infinitas solucoes

int gauss(vector<bitset<M>> a, int n, int m, bitset<M> &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++;
    }

    ans.reset();
    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
    // Unica solucao (retornada no bitset ans)
    return 1;
}
```

4.3 Exponenciação Modular Rápida

Computa $(\text{base}^{\text{exp}}) \mod MOD$ em $\mathcal{O}(\log(\text{exp}))$.

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;
}
```

4.4 FFT

Algoritmo que computa a Transformada Rápida de Fourier para convolução de polinômios.

Computa convolução (multiplicação) de polinômios em $\mathcal{O}(N \cdot \log N)$, sendo N a soma dos graus dos polinômios.

Testado e sem erros de precisão com polinômios de grau até $3 * 10^5$ e constantes até 10^6. Para convolução de inteiros sem erro de precisão, consultar a seção de NTT.

Codigo: fft.cpp

```cpp
struct base {
    double a, b;
    base(double _a = 0, double _b = 0) : a(_a), b(_b) { }
    const base operator+(const base &c) const { return base(a + c.a, b + c.b); }
    const base operator-(const base &c) const { return base(a - c.a, b - c.b); }
    const base operator*(const base &c) const {
        return base(a * c.a - b * c.b, a * c.b + b * c.a);
    }
};

using poly = vector<base>;
const double PI = acos(-1);

void fft(poly &a, bool inv = 0) {
    int n = (int)a.size();

    for (int i = 0; i < n; i++) {
        int bit = n >> 1, j = 0, k = i;
```

```
        while (bit > 0) {
            if (k & 1) j += bit;
            k >>= 1, bit >>= 1;
        }
        if (i < j) swap(a[i], a[j]);
    }

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

    for (int len = 2; len <= n; len <<= 1) {
        int aux = len / 2;
        int step = n / len;
        for (int i = 0; i < n; i += len) {
            for (int j = 0; j < aux; j++) {
                base v = a[i + j + aux] * wn[step * j];
                a[i + j + aux] = a[i + j] - v;
                a[i + j] = a[i + j] + v;
            }
        }
    }

    for (int i = 0; inv && i < n; i++) a[i].a /= n, a[i].b /= n;
}

vector<ll> multiply(vector<ll> &ta, vector<ll> &tb) {
    int n = (int)ta.size(), m = (int)tb.size();
    int t = n + m - 1, sz = 1;
    while (sz < t) sz <<= 1;

    poly a(sz), b(sz), c(sz);

    for (int i = 0; i < sz; i++) {
        a[i] = i < n ? base((double)ta[i]) : base(0);
        b[i] = i < m ? base((double)tb[i]) : base(0);
    }

    fft(a, 0), fft(b, 0);
    for (int i = 0; i < sz; i++) c[i] = a[i] * b[i];
    fft(c, 1);

    vector<ll> res(sz);
    for (int i = 0; i < sz; i++) res[i] = ll(round(c[i].a));

    while ((int)res.size() > t && res.back() == 0) res.pop_back();

    return res;
}
```

4.5 Fatoração e Primos

4.5.1 Crivo

Crivo

Crivo de Eratóstenes para encontrar os primos até um limite P. O `vector<bool> is_prime` é um vetor que diz se um número é primo ou não. A complexidade é $\mathcal{O}(P \log(\log P))$.

Obs: Para aplicações mais complexas ou pra fatorar um número, consulte o Crivo Linear.

Obs: Não esquecer de chamar `Sieve::build()` antes de usar.

Codigo: sieve.cpp

```
namespace Sieve {
    const int P = 5e6 + 1;
    vector<bool> is_prime(P, true);
    void build() {
        is_prime[0] = is_prime[1] = 0;
        for (int i = 2; i < P; i++) {
            if (is_prime[i])
                for (int j = i + i; j < P; j += i) is_prime[j] = 0;
        }
    }
}
```

Crivo Linear

Crivo de Eratóstenes para encontrar os primos até um limite P, mas com complexidade $\mathcal{O}(P)$.

- `vector<bool> is_prime` é um vetor que diz se um número é primo ou não.

- `int cnt` é o número de primos encontrados.
- `int primes[P]` é um vetor com `cnt` os primos encontrados.
- `int lpf[P]` é o menor fator primo de cada número (usado para fatoração).

A função `Sieve::factorize()` fatora um número N em tempo $\mathcal{O}(\log N)$.

Obs: Não esquecer de chamar `Sieve::build()` antes de usar.

Codigo: linear_sieve.cpp

```cpp
namespace Sieve {
    const int P = 5e6 + 1;
    vector<bool> is_prime(P, true);
    int lpf[P], primes[P], cnt = 0;
    void build() {
        is_prime[0] = is_prime[1] = 0;
        for (int i = 2; i < P; i++) {
            if (is_prime[i]) {
                lpf[i] = i;
                primes[cnt++] = i;
            }
            for (int j = 0; j < cnt && i * primes[j] < P; j++) {
                is_prime[i * primes[j]] = 0;
                lpf[i * primes[j]] = primes[j];
                if (i % primes[j] == 0) break;
            }
        }
    }
    vector<int> factorize(int n) {
#warning lembra de chamar o build() antes de fatorar!
        vector<int> f;
        while (n > 1) {
            f.push_back(lpf[n]);
            n /= lpf[n];
        }
        return f;
    }
}
```

4.5.2 Divisores

Divisores Naive

Algoritmo que obtém todos os divisores de um número X em $\mathcal{O}(\sqrt{X})$. Muito similar ao algoritmo naive de fatoração.

Codigo: get_divs_naive.cpp

```cpp
vector<int> get_divs(int n) {
    vector<int> divs;
    for (int d = 1; d * d <= n; d++) {
        if (n % d == 0) {
            divs.push_back(d);
            if (d * d != n) divs.push_back(n / d);
        }
    }
    sort(divs.begin(), divs.end());
    return divs;
}
```

Divisores Rápido

Algoritmo que obtém todos os divisores de um número em $\mathcal{O}(\mathrm{d(X)})$, onde $d(X)$ é a quantidade de divisores do número. Geralmente, para um número X, dizemos que a quantidade de divisores é $\mathcal{O}(\sqrt[3]{X})$.

De fato, para números até 10^{88}, é verdade que $d(n) < 3.6 \cdot \sqrt[3]{n}$.

Obs: Usar algum código de fatoração presente nesse almanaque para obter os fatores do número.

- `Crivo/Crivo-Linear/linear_sieve.cpp` tem uma função de fatoração em $\mathcal{O}(\log X)$.
- `Fatores/Fatoração-Rápida/fast_factorize.cpp` tem uma função de fatoração em tempo médio $\mathcal{O}(\log X)$ que aceita até inteiros de 64 bits.

Codigo: get_divs.cpp

```cpp
vector<ll> get_divs(ll n) {
    vector<ll> divs;
    auto f = factorize(n); // qualquer código que fatore n
    sort(f.begin(), f.end());
    vector<pair<ll, int>> v;
    for (auto x : f)
        if (v.empty() || v.back().first != x) v.emplace_back(x, 1);
        else v.back().second += 1;
    function<void(int, ll)> dfs = [&](int i, ll cur) {
        if (i == (int)v.size()) {
            divs.push_back(cur);
            return;
        }
        ll p = 1;
        for (int j = 0; j <= v[i].second; j++) {
            dfs(i + 1, cur * p);
            p *= v[i].first;
        }
    };
    dfs(0, 1);
    sort(divs.begin(), divs.end());
    return divs;
}
```

4.5.3 Fatores

Fatoração Naive

Fatoração de um número. A função `factorize(X)` retorna os fatores primos de X em ordem crescente. A complexidade do algoritmo é $\mathcal{O}(\sqrt{X})$.

Codigo: naive_factorize.cpp

```cpp
vector<int> factorize(int n) {
    vector<int> factors;
    for (int 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;
}
```

Fatoração Rápida

Algoritmo que combina o Crivo de Eratóstenes Linear, Miller-Rabin e Pollard Rho para fatorar um número X em tempo médio $\mathcal{O}(\log X)$, no pior caso pode ser $\mathcal{O}(\sqrt[4]{X} \cdot \log X)$, mas na prática é seguro considerar $\mathcal{O}(\log X)$.

Esse código deve ser usado quando se deseja fatorar números $> 10^7$, por exemplo. Caso os números não sejam tão grandes assim, usar apenas o Crivo de Eratóstenes Linear sozinho é mais prático.

Obs: Usa três outros códigos desse Almanaque da seção Matemática:

- `Crivo/Crivo-Linear/linear_sieve.cpp`
- `Teste-Primalidade/Miller-Rabin/miller_rabin.cpp-`
- `Pollard-Rho/pollard_rho.cpp`.

Codigo: fast_factorize.cpp

```cpp
vector<ll> factorize(ll y) {
    vector<ll> f;
    if (y == 1) return f;
    function<void(ll)> dfs = [&](ll x) {
        if (x == 1) return;
        if (x < Sieve::P) {
            auto fs = Sieve::factorize(x);
            f.insert(f.end(), fs.begin(), fs.end());
        } else if (MillerRabin::prime(x)) {
            f.push_back(x);
        } else {
            ll d = PollardRho::rho(x);
            dfs(d);
            dfs(x / d);
        }
    };
    dfs(y);
    sort(f.begin(), f.end());
    return f;
}
```

4.5.4 Pollard Rho

Algoritmo de Pollard Rho. A função `PollardRho::rho(X)` retorna um fator não trivial de X. Um fator não trivial é um fator que não é 1 nem X. A complexidade esperada do algoritmo no pior caso é $\mathcal{O}(\sqrt[4]{X})$ (geralmente é mais rápido que isso).

Obs: cuidado para não passar um número primo ou o número 1 para a função `rho`, o comportamente é indefinido (provavelmente entra em loop e não retorna nunca).

Codigo: pollard_rho.cpp

```cpp
namespace PollardRho {
    mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
    const ll P = 1e6 + 1;
    ll seq[P];
    inline ll add_mod(ll x, ll y, ll m) { return (x += y) < m ? x : x - m; }
    inline ll mul_mod(ll a, ll b, ll m) { return (ll)((__int128)a * b % m); }
    ll rho(ll n) {
        if (n % 2 == 0) return 2;
        if (n % 3 == 0) return 3;
        ll x0 = rng() % n, c = rng() % n;
        while (1) {
            ll x = x0++, y = x, u = 1, v, t = 0;
            ll *px = seq, *py = seq;
            while (1) {
                *py++ = y = add_mod(mul_mod(y, y, n), c, n);
                *py++ = y = add_mod(mul_mod(y, y, n), c, n);
                if ((x = *px++) == y) break;
                v = u;
                u = mul_mod(u, abs(y - x), n);
                if (!u) return gcd(v, n);
                if (++t == 32) {
                    t = 0;
                    if ((u = gcd(u, n)) > 1 && u < n) return u;
                }
            }
            if (t && (u = gcd(u, n)) > 1 && u < n) return u;
        }
    }
}
```

4.5.5 Teste Primalidade

Miller Rabin

Teste de primalidade Miller-Rabin. A função `MillerRabin::prime(X)` retorna verdadeiro se X é primo e falso caso contrário. O teste é determinístico para para números até 2^{64}. A complexixade do algoritmo é $\mathcal{O}(\log X)$, considerando multiplicação e exponenciação constantes.

Para números até 2^{32}, é suficiente usar `primes[] = 2, 3, 5, 7`.

Codigo: miller_rabin.cpp

```cpp
namespace MillerRabin {
    inline ll mul_mod(ll a, ll b, ll m) { return (ll)((__int128)a * b % m); }
    inline ll power(ll b, ll e, ll m) {
        ll r = 1;
        b = b % m;
        while (e > 0) {
            if (e & 1) r = mul_mod(r, b, m);
            b = mul_mod(b, b, m), e >>= 1;
        }
        return r;
    }
    inline bool composite(ll n, ll a, ll d, ll s) {
        ll x = power(a, d, n);
        if (x == 1 || x == n - 1 || a % n == 0) return false;
        for (int r = 1; r < s; r++) {
            x = mul_mod(x, x, n);
            if (x == n - 1) return false;
        }
        return true;
    }

    // com esses "primos", o teste funciona garantido para n <= 2^64
    int primes[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};

    // funciona para n <= 3*10^24 com os primos ate 41, mas tem que cuidar com overflow
    // int primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41};

    bool prime(ll n) {
        if (n <= 2 || (n % 2 == 0)) return n == 2;
        ll d = n - 1, r = 0;
        while (d % 2 == 0) d /= 2, r++;
        for (int a : primes)
            if (composite(n, a, d, r)) return false;
```

```
        return true;
    }
}
```

Teste Primalidade Naive

Teste de primalidade "ingênuo". A função `is_prime(X)` retorna verdadeiro se X é primo e falso caso contrário. A complexidade do algoritmo é $\mathcal{O}(\sqrt{X})$.

Codigo: naive_is_prime.cpp

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

4.6 Floor Values

Código para encontrar todos os $\mathcal{O}(\sqrt{n})$ valores distintos de $\lfloor \frac{n}{i} \rfloor$ para $i = 1, 2, \ldots, n$.

Útil para computar, dentre outras coisas, os seguintes somatórios:

- Somatório de $\lfloor \frac{n}{i} \rfloor$ para $i = 1, 2, \ldots, n$.
- Somatório de $\sigma(i)$ para $i = 1, 2, \ldots, n$, onde $\sigma(i)$ é a soma dos divisores de i.

- Usa o fato de que um número i é divisor de exatamente $\lfloor \frac{n}{i} \rfloor$ números entre 1 e n.

Codigo: floor_values.cpp

```
void floor_values(ll n) {
    ll j = 1;
    while (j <= n) {
        ll floor_now = n / j;
        ll last_j = n / floor_now;
        // j -> primeiro inteiro que tem floor_now como floor
        // last_j -> ultimo inteiro que tem floor_now como floor

        // faz algo

        j = last_j + 1;
    }
}
```

4.7 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: 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;
    }
}
```

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;
}
```

4.8 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

```
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

```
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_coprime.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;
}
```

Codigo: modular_inverse.cpp

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

4.9 NTT

4.9.1 NTT

Computa a multiplicação de polinômios com coeficientes inteiros módulo um número primo em $\mathcal{O}(N \cdot \log N)$. Exatamente o mesmo algoritmo da FFT, mas com inteiros.

Não é qualquer módulo que funciona, aqui tem alguns que funcionam:

1. 998244353 ($\approx 9 \times 10^8$): Para polinômios de tamanho até 2^{23}.
2. 1004535809 ($\approx 10^9$): Para polinômios de tamanho até 2^{21}.
3. 1092616193 ($\approx 10^9$): Para polinômios de tamanho até 2^{21}.
4. 9223372036737335297 ($\approx 9 \times 10^{18}$): Para polinômios de tamanho até 2^{24}, se usar esse módulo, cuidado com os `ints`, use `long long`.

Obs: Essa implementação usa a primitiva `Mint` desse Almanaque. Se você não quiser usar o `Mint`, basta substituir todas as ocorrências de `Mint` por `int` ou `long long` e tratar adequadamente as operações com aritmética modular.

Obs 2: Nem tente usar 10^9+7 ou 10^9+9 como módulo, eles não funcionam. Para isso, pode-se tentar usar a NTT Big Modulo.

Codigo: ntt.cpp

```
template <auto MOD, typename T = Mint<MOD>>
void ntt(vector<T> &a, bool inv = 0) {
    int n = (int)a.size();
    auto b = a;
    T g = 1;
    while ((g ^ (MOD / 2)) == 1) g += 1;
    if (inv) g = T(1) / g;
    for (int step = n / 2; step; step /= 2) {
        T w = g ^ (MOD / (n / step)), wn = 1;
        for (int i = 0; i < n / 2; i += step) {
            for (int j = 0; j < step; j++) {
                auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
                b[i + j] = u + v;
                b[i + n / 2 + j] = u - v;
            }
            wn = wn * w;
        }
        swap(a, b);
    }
    if (inv) {
        T invn = T(1) / n;
        for (int i = 0; i < n; i++) a[i] *= invn;
    }
}

template <auto MOD, typename T = Mint<MOD>>
vector<T> multiply(vector<T> a, vector<T> b) {
    int n = (int)a.size(), m = (int)b.size();
    int t = n + m - 1, sz = 1;
    while (sz < t) sz <<= 1;
    a.resize(sz), b.resize(sz);
    ntt<MOD>(a, 0), ntt<MOD>(b, 0);
    for (int i = 0; i < sz; i++) a[i] *= b[i];
    ntt<MOD>(a, 1);
    while ((int)a.size() > t) a.pop_back();
    return a;
}
```

4.9.2 NTT Big Modulo

NTT usada para computar a multiplicação de polinômios com coeficientes inteiros módulo um número primo grande. A ideia na maioria dos casos é computar a multiplicação como se não houvesse módulo, por isso usamos um módulo grande.

Uma forma de fazer essa NTT com módulo grande é usar o módulo grande que está na seção NTT.

A forma usada nesse código é usar dois módulos na ordem de 10^9 e fazer a multiplicação com eles. E depois usar o Teorema do Resto Chinês para achar o resultado módulo o produto dos módulos.

Codigo: big_ntt.cpp

```cpp
template <auto MOD, typename T = Mint<MOD>>
void ntt(vector<T> &a, bool inv = 0) {
    int n = (int)a.size();
    auto b = a;
    T g = 1;
    while ((g ^ (MOD / 2)) == 1) g += 1;
    if (inv) g = T(1) / g;
    for (int step = n / 2; step; step /= 2) {
        T w = g ^ (MOD / (n / step)), wn = 1;
        for (int i = 0; i < n / 2; i += step) {
            for (int j = 0; j < step; j++) {
                auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
                b[i + j] = u + v;
                b[i + n / 2 + j] = u - v;
            }
            wn = wn * w;
        }
        swap(a, b);
    }
    if (inv) {
        T invn = T(1) / n;
        for (int i = 0; i < n; i++) a[i] *= invn;
    }
}

template <auto MOD, typename T = Mint<MOD>>
vector<T> multiply(vector<T> a, vector<T> b) {
    int n = (int)a.size(), m = (int)b.size();
    int t = n + m - 1, sz = 1;
    while (sz < t) sz <<= 1;
    a.resize(sz), b.resize(sz);
    ntt<MOD>(a, 0), ntt<MOD>(b, 0);
    for (int i = 0; i < sz; i++) a[i] *= b[i];
    ntt<MOD>(a, 1);
    while ((int)a.size() > t) a.pop_back();
    return a;
}

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

ll crt(array<int, 2> rem, array<int, 2> mod) {
    __int128 ans = rem[0], m = mod[0];
    ll x, y;
    ll g = extended_gcd(mod[1], (ll)m, x, y);
    if ((ans - rem[1]) % g != 0) return -1;
    ans = ans + (__int128)1 * (rem[1] - ans) * (m / g) * y;
    m = (__int128)(mod[1] / g) * (m / g) * g;
    ans = (ans % m + m) % m;
    return (ll)ans;
}

template <auto MOD1, auto MOD2, typename T = Mint<MOD1>, typename U = Mint<MOD2>>
vector<ll> big_multiply(vector<ll> ta, vector<ll> tb) {
    vector<T> a1(ta.size()), b1(tb.size());
    vector<U> a2(ta.size()), b2(tb.size());
    for (int i = 0; i < (int)ta.size(); i++) a1[i] = ta[i];
    for (int i = 0; i < (int)tb.size(); i++) b1[i] = tb[i];
    for (int i = 0; i < (int)ta.size(); i++) a2[i] = ta[i];
    for (int i = 0; i < (int)tb.size(); i++) b2[i] = tb[i];
    auto c1 = multiply<MOD1>(a1, b1);
    vector<ll> res(c1.size());
    for (int i = 0; i < (int)res.size(); i++)
        res[i] = crt({c1[i].v, c2[i].v}, {MOD1, MOD2});
    return res;
}

const int MOD1 = 1004535809;
const int MOD2 = 1092616193;
```

4.9.3 Taylor Shift

Usa NTT para computar o polinômio $p(x+k)$, dados p e k. A complexidade é $O(n \log n)$.

Codigo: taylor_shift.cpp

```cpp
template <auto MOD, typename T = Mint<MOD>>
vector<T> shift(vector<T> a, int k) {
    int n = (int)a.size();
    vector<T> fat(n, 1), ifat(n), shifting(n);
    for (int i = 1; i < n; i++) fat[i] = fat[i - 1] * i;
    ifat[n - 1] = T(1) / fat[n - 1];
    for (int i = n - 1; i > 0; i--) ifat[i - 1] = ifat[i] * i;
    for (int i = 0; i < n; i++) a[i] *= fat[i];
    T pk = 1;
    for (int i = 0; i < n; i++) {
        shifting[n - i - 1] = pk * ifat[i];
        pk *= k;
    }
    auto ans = multiply<MOD>(a, shifting);
    ans.erase(ans.begin(), ans.begin() + n - 1);
    for (int i = 0; i < n; i++) ans[i] *= ifat[i];
    return ans;
}
```

4.10 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

```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;
}
```

4.11 Totiente de Euler

Código para computar a função Totiente de Euler, que conta quantos números inteiros positivos menores que N são coprimos com N. A função é denotada por $\phi(N)$.

É possível computar o totiente de Euler para um único número em $\mathcal{O}(\sqrt{N})$ e para todos os números entre 1 e N em $\mathcal{O}(N \cdot \log(\log N))$.

Codigo: phi.cpp

```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;
}
```

Codigo: phi_1_to_n.cpp

```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;
}
```

4.12 XOR Gauss

Mantém uma base num espaço vetorial de L dimensões sobre $\mathbb{Z}_2$. Permite adicionar um vetor v à base em $\mathcal{O}(L)$ e verificar se um vetor v é representável pela base em $\mathcal{O}(L)$.

Em termos mais simples, dados n inteiros, podemos adicionar cada um deles à base e isso nos dará uma base que consegue representar todos os XORs possíveis entre esses inteiros.

Também acha o k-ésimo menor vetor representável pela base em $\mathcal{O}(L)$, ou o k-ésimo maior vetor representável pela base em $\mathcal{O}(L)$.

Informações relevantes:

- rank de uma base é o número de vetores que ela contém. No código é a variável `R`.
- Uma base consegue criar 2^{rank} vetores diferentes, ou seja, se criarmos uma base com base em um vetor de tamanho n, dentre todos os 2^n subsets possíveis, existem exatamente 2^{rank} XORs diferentes.
- Se uma base for criada a partir de um vetor de tamanho n, cada XOR possível feito por um subset desse vetor pode ser criado de exatamente $2^{n-\text{rank}}$ formas diferentes.

Os métodos são:

- `reduce`: recebe um número x (será tratado como um vetor no espaço vetorial) e subtrai os vetores já existentes na base que estão presentes em x. Sendo assim, se ao final do `reduce`, x for diferente de zero, ele não é representável por uma combinação linear dos vetores da base, se for zero, ele é representável.
- `insert`: insere um vetor na base, se ele não for representável. No caso, o vetor inserido ao tentar inserir um valor x na base, será o `reduce` de x.
- `kth_greatest`: retorna o k-ésimo maior vetor representável pela base.
- `kth_smallest`: retorna o k-ésimo menor vetor não representável pela base.

Todos os métodos são $\mathcal{O}(L)$.

Codigo: xor_gauss.cpp

```cpp
const int L = 60;
struct Basis {
    ll B[L], R, last_bit;
    Basis() { memset(B, 0, sizeof B), R = 0; }
    ll reduce(ll x) {
        for (int i = L - 1; i >= 0; i--) {
            if ((x >> i) & 1) {
                if (B[i] != 0) {
                    x ^= B[i];
                } else {
                    last_bit = i;
                    return x;
                }
            }
        }
        // assert(x == 0);
        return 0;
    }
    bool insert(ll x) {
        x = reduce(x);
        if (x > 0) {
            R++;
            B[last_bit] = x;
            return true;
        }
        return false;
    }
    ll kth_smallest(ll k) {
        ll ans = 0;
        ll half = 1LL << (R - 1);
        for (int i = L - 1; i >= 0; i--) {
            if (B[i] != 0) {
```

```
33                if ((ans >> i) & 1) {
34                    if (k > half) k -= half;
35                    else ans ^= B[i];
36                } else if (k > half) {
37                    ans ^= B[i];
38                    k -= half;
39                }
40                half >>= 1;
41            }
42        }
43        return ans;
44    }
45    ll kth_greatest(ll k) { return kth_smallest((1LL << R) - k + 1); }
46 };
```

Capítulo 5

Grafos

5.1 2 SAT

Algoritmo que resolve problema do 2-SAT. No 2-SAT, temos um conjunto de variáveis booleanas e cláusulas lógicas, onde cada cláusula é composta por duas variáveis. O problema é determinar se existe uma configuração das variáveis que satisfaça todas as cláusulas. O problema se transforma em um problema de encontrar as componentes fortemente conexas de um grafo direcionado, que resolvemos em $\mathcal{O}(N + M)$ com o algoritmo de Kosaraju. Onde N é o número de variáveis e M é o número de cláusulas.

A configuração da solução fica guardada no vetor `assignment`.

Exemplos de uso:

- `sat.add_or(x, y)` $\Leftrightarrow (x \vee y)$
- `sat.add_or( x, y)` $\Leftrightarrow (\neg x \vee y)$
- `sat.add_impl(x, y)` $\Leftrightarrow (x \rightarrow y)$
- `sat.add_and(x,  y)` $\Leftrightarrow (x \wedge \neg y)$
- `sat.add_xor(x, y)` $\Leftrightarrow (x \vee y) \wedge \neg(x \wedge y)$
- `sat.add_equals(x, y)` $\Leftrightarrow (x \wedge y) \vee (\neg x \wedge \neg y)$

Codigo: 2_sat.cpp

```cpp
struct sat2 {
    int n;
    vector<vector<int>> g, rg;
    vector<bool> vis, assignment;
    vector<int> topo, comp;

    void build(int _n) {
        n = 2 * _n;
        g.assign(n, vector<int>());
        rg.assign(n, vector<int>());
    }

    int get(int u) {
        if (u < 0) return 2 * (~u) + 1;
        else return 2 * u;
    }

    void add_impl(int u, int v) {
        u = get(u), v = get(v);
        g[u].push_back(v);
        rg[v].push_back(u);
        g[v ^ 1].push_back(u ^ 1);
        rg[u ^ 1].push_back(v ^ 1);
    }

    void add_or(int u, int v) { add_impl(~u, v); }

    void add_and(int u, int v) {
        add_or(u, u);
        add_or(v, v);
    }
```

```
32
33      void add_xor(int u, int v) {
34          add_impl(u, ~v);
35          add_impl(~u, v);
36      }
37
38      void add_equals(int u, int v) {
39          add_impl(u, v);
40          add_impl(~u, ~v);
41      }
42
43      void toposort(int u) {
44          vis[u] = true;
45          for (int v : g[u])
46              if (!vis[v]) toposort(v);
47          topo.push_back(u);
48      }
49
50      void dfs(int u, int cc) {
51          comp[u] = cc;
52          for (int v : rg[u])
53              if (comp[v] == -1) dfs(v, cc);
54      }
55
56      pair<bool, vector<bool>> solve() {
57          topo.clear();
58          vis.assign(n, false);
59
60          for (int i = 0; i < n; i++)
61              if (!vis[i]) toposort(i);
62          reverse(topo.begin(), topo.end());
63
64          comp.assign(n, -1);
65          int cc = 0;
66          for (auto u : topo)
67              if (comp[u] == -1) dfs(u, cc++);
68
69          assignment.assign(n / 2, false);
70          for (int i = 0; i < n; i += 2) {
71              if (comp[i] == comp[i + 1]) return {false, {}};
72              assignment[i / 2] = comp[i] > comp[i + 1];
73          }
74
75          return {true, assignment};
76      }
77  };
```

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

```
1  const int N = 3e5 + 5, LG = 20;
2  vector<int> adj[N];
3
4  namespace bl {
5      int t, up[N][LG], tin[N], tout[N];
6
7      void dfs(int u, int p = -1) {
8          tin[u] = t++;
9          for (int i = 0; i < LG - 1; i++) up[u][i + 1] = up[up[u][i]][i];
10         for (int v : adj[u])
11             if (v != p) {
12                 up[v][0] = u;
13                 dfs(v, u);
14             }
15         tout[u] = t++;
16     }
17
18     void build(int root) {
19         t = 1;
20         up[root][0] = root;
21         dfs(root);
22     }
23
24     bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }
25
26     int lca(int u, int v) {
27         if (ancestor(u, v)) return u;
28         if (ancestor(v, u)) return v;
29         for (int i = LG - 1; i >= 0; i--)
30             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;
    }

}
```

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 $f(par[u], up[u][i])$, em que `up[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

```
const int N = 3e5 + 5, LG = 20;
vector<int> adj[N];

namespace bl {
    int t, up[N][LG], st[N][LG], tin[N], tout[N], val[N];

    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) {
        t = 1;
        up[root][0] = root;
        st[root][0] = neutral;
        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;
    }
```

```
63
64 }
```

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

```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

```
const int N = 3e5 + 5, LG = 20;
vector<pair<int, int>> adj[N];

namespace bl {
    int t, up[N][LG], st[N][LG], tin[N], tout[N], val[N];

    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) {
        t = 1;
        up[root][0] = root;
        st[root][0] = neutral;
        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));
    }

    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;
    }
}
```

5.3 Block Cut Tree

Algoritmo que separa o grafo em componentes biconexas em $\mathcal{O}(V + E)$.

- `id[u]` é o index do nodo `u` na Block Cut Tree.
- `is_articulation_point(u)` diz se o nodo `u` é ou não é um ponto de articulação.
- `number_of_splits(u)` diz a quantidade de componentes conexas que o grafo terá se o nodo `u` for removido.

Codigo: block_cut_tree.cpp

```cpp
struct Bct {
    int T;
    vector<int> tin, low, stk, art, id, splits;
    vector<vector<int>> adj, g, comp, up;
    int n, sz, m;
    void build(int _n, int _m) {
        n = _n, m = _m;
        adj.resize(n);
    }
    void add_edge(int u, int v) {
        adj[u].emplace_back(v);
        adj[v].emplace_back(u);
    }
    void dfs(int u, int p) {
        low[u] = tin[u] = ++T;
        stk.emplace_back(u);
        for (auto v : adj[u]) {
            if (tin[v] == -1) {
                dfs(v, u);
                low[u] = min(low[u], low[v]);
                if (low[v] >= tin[u]) {
                    int x;
                    sz++;
                    do {
                        assert(stk.size());
                        x = stk.back();
                        stk.pop_back();
                        comp[x].emplace_back(sz);
                    } while (x != v);
                    comp[u].emplace_back(sz);
                }
            } else if (v != p) {
                low[u] = min(low[u], tin[v]);
            }
        }
    }
    inline bool is_articulation_point(int u) { return art[id[u]]; }
    inline int number_of_splits(int u) { return splits[id[u]]; }
    void work() {
        T = sz = 0;
        stk.clear();
        tin.resize(n, -1);
        comp.resize(n);
        low.resize(n);
        for (int i = 0; i < n; i++)
            if (tin[i] == -1) dfs(i, 0);
        art.resize(sz + n + 1);
        splits.resize(n + sz + 1, 1);
        id.resize(n);
        g.resize(sz + n + 1);
        for (int i = 0; i < n; i++) {
            if ((int)comp[i].size() > 1) {
                id[i] = ++sz;
                art[id[i]] = 1;
                splits[id[i]] = (int)comp[i].size();
                for (auto u : comp[i]) {
                    g[id[i]].emplace_back(u);
                    g[u].emplace_back(id[i]);
                }
            } else if (comp[i].size()) {
                id[i] = comp[i][0];
            }
        }
    }
};
```

5.4 Caminho Euleriano

5.4.1 Caminho Euleriano Direcionado

Algoritmo para encontrar um caminho euleriano em um grafo direcionado em $\mathcal{O}(V+E)$. O algoritmo também encontrará um ciclo euleriano se o mesmo existir. Se nem um ciclo nem um caminho euleriano existir, o algoritmo retornará um vetor vazio.

Definição: Um caminho euleriano é um caminho que passa por todas as arestas de um grafo exatamente uma vez. Um ciclo euleriano é um caminho euleriano que começa e termina no mesmo vértice. A condição de existência de um ciclo euleriano (em um grafo direcionado) é que todos os vértices do grafo possuam grau de entrada e saída iguais. A condição de existência de um caminho euleriano (em um grafo direcionado) é que o grafo possua exatamente dois vértices com grau de entrada e saída diferentes, sendo um deles o vértice de início (`deg_in[u] == deg_out[u] - 1`) e o outro o vértice de término (`deg_out[v] == deg_in[v] - 1`).

Codigo: directed_eulerian_path.cpp

```cpp
const int MAXN = 1e6 + 6;

vector<int> adj[MAXN];

struct EulerianTrail {
    int n;
    int it[MAXN], deg_in[MAXN], deg_out[MAXN];
    void build(int _n) {
        n = _n;
        for (int i = 0; i < n; i++) it[i] = deg_in[i] = deg_out[i] = 0;
    }
    vector<int> find() {
        vector<int> cur;
        int m = 0;
        for (int i = 0; i < n; i++) {
            for (int j : adj[i]) {
                m++;
                deg_out[i]++, deg_in[j]++;
            }
        }
        int start = -1, end = -1;
        for (int i = 0; i < n; i++) {
            if (deg_in[i] != deg_out[i]) {
                if (deg_in[i] == deg_out[i] - 1)
                    if (start == -1) start = i;
                    else return {};
                else if (deg_in[i] - 1 == deg_out[i])
                    if (end == -1) end = i;
                    else return {};
                else return {};
            }
        }
        if (start == -1 && end == -1) {
            // pode comecar em qualquer vertice com alguma aresta (mas tem que terminar
            // nele tambem), nesse caso eh ciclo euleriano
            for (int i = 0; i < n; i++) {
                if (deg_out[i] > 0) {
                    start = i;
                    end = i;
                    break;
                }
            }
        } else if (start == -1 || end == -1) {
            return {};
        }
        function<void(int)> dfs_et = [&](int u) {
            while (it[u] < (int)adj[u].size()) {
                int v = adj[u][it[u]++];
                dfs_et(v);
            }
            cur.push_back(u);
        };
        dfs_et(start);
        if ((int)cur.size() != m + 1) return {};
        reverse(cur.begin(), cur.end());
        return cur;
    }
} et_finder;
```

5.4.2 Caminho Euleriano Nao Direcionado

Algoritmo para encontrar um caminho euleriano em um grafo não direcionado em $\mathcal{O}(V+E)$. O algoritmo também encontrará um ciclo euleriano se o mesmo existir. Se nem um ciclo nem um caminho euleriano existir, o algoritmo retornará um vetor vazio.

Definição: Um caminho euleriano é um caminho que passa por todas as arestas de um grafo exatamente uma vez. Um ciclo euleriano é um caminho euleriano que começa e termina no mesmo vértice. A condição de existência de um ciclo euleriano (em um grafo

não direcionado) é que todos os vértices do grafo possuam grau par. A condição de existência de um caminho euleriano (em um grafo não direcionado) é que o grafo possua exatamente dois vértices com grau ímpar, um deles será o vértice de início e o outro o vértice de término.

Codigo: undirected_eulerian_path.cpp

```cpp
const int MAXN = 1e6 + 6, MAXM = 2e6 + 6;

vector<pair<int, int>> adj[MAXN]; // {nodo, id da aresta}

struct EulerianTrail {
    int n, m;
    int it[MAXN], deg[MAXN], vis_edge[MAXM];
    void build(int _n, int _m) {
        n = _n;
        m = _m;
        for (int i = 0; i < n; i++) it[i] = deg[i] = 0;
        for (int i = 0; i < m; i++) vis_edge[i] = 0;
    }
    vector<int> find() {
        vector<int> cur;
        for (int i = 0; i < n; i++) deg[i] = (int)adj[i].size();
        int start = -1, end = -1;
        for (int i = 0; i < n; i++) {
            if (deg[i] & 1) {
                if (start == -1) start = i;
                else if (end == -1) end = i;
                else return {};
            }
        }
        if (start == -1 && end == -1) {
            // pode comecar em qualquer vertice com alguma aresta (mas tem que terminar
            // nele tambem), nesse caso eh ciclo euleriano
            for (int i = 0; i < n; i++) {
                if (deg[i] > 0) {
                    start = i;
                    end = i;
                    break;
                }
            }
        } else if (start == -1 || end == -1) {
            return {};
        }
        function<void(int)> dfs_et = [&](int u) {
            while (it[u] < (int)adj[u].size()) {
                auto [v, id] = adj[u][it[u]++];
                if (vis_edge[id]) continue;
                vis_edge[id] = 1;
                dfs_et(v);
            }
            cur.push_back(u);
        };
        dfs_et(start);
        if ((int)cur.size() != m + 1) return {};
        reverse(cur.begin(), cur.end());
        return cur;
    }
} et_finder;
```

5.5 Centro e Diametro

Algoritmo que encontra o centro e o diâmetro de um grafo em $\mathcal{O}(N + M)$ com duas BFS.

Definição: O centro de um grafo é igual ao subconjunto de nodos com excentricidade mínima. A excentricidade de um nodo é a maior distância dele para qualquer outro nodo. Em outras palavras, pra um nodo ser centro do grafo, ele deve minimizar a maior distância para qualquer outro nodo.

O diâmetro de um grafo é a maior distância entre dois nodos quaisquer.

Codigo: graph_center.cpp

```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 Centroids

5.6.1 Centroid

Algoritmo que encontra os dois centroides de uma árvore em $\mathcal{O}(N)$.

Definição: O centroide de uma árvore é o nodo tal que, ao ser removido, divide a árvore em subárvores com no máximo metade dos nodos da árvore original. Em outras palavras, se a árvore tem tamanho N, todas as subárvores geradas pela remoção do centroide têm tamanho no máximo $\frac{N}{2}$. Uma árvore pode ter até dois centróides.

Codigo: find_centroid.cpp

```
const int N = 3e5 + 5;

int sz[N];
vector<int> adj[N];

void dfs_sz(int u, int p) {
    sz[u] = 1;
    for (int v : adj[u]) {
        if (v != p) {
            dfs_sz(v, u);
            sz[u] += sz[v];
        }
    }
}

int centroid(int u, int p, int n) {
    for (int v : adj[u])
        if (v != p && sz[v] > n / 2) return centroid(v, u, n);
    return u;
}

pair<int, int> centroids(int u) {
    dfs_sz(u, u);
    int c = centroid(u, u, sz[u]);
    int c2 = -1;
    for (int v : adj[c])
        if (sz[u] == sz[v] * 2) c2 = v;
    return {c, c2};
}
```

5.6.2 Centroid Decomposition

Algoritmo que constrói a decomposição por centroides de uma árvore em $\mathcal{O}(N \log N)$.

Basicamente, a decomposição consiste em, repetidamente:

- Encontrar o centroide da árvore atual.
- Remover o centroide e decompor as subárvores restantes.

A decomposição vai gerar uma nova árvore (chamada comumente de "Centroid Tree") onde cada nodo é um centroide da árvore original e as arestas representam a relação de pai-filho entre os centroides. A árvore tem altura $\log N$.

No código, `dis[u][j]` é a distância entre o nodo u e seu j-ésimo ancestral na Centroid Tree.

Codigo: centroid_decomposition.cpp

```cpp
const int N = 3e5 + 5;

int sz[N], par[N];
bool rem[N];
vector<int> dis[N];
vector<int> adj[N];

int dfs_sz(int u, int p) {
    sz[u] = 1;
    for (int v : adj[u])
        if (v != p && !rem[v]) sz[u] += dfs_sz(v, u);
    return sz[u];
}

int centroid(int u, int p, int szn) {
    for (int v : adj[u])
        if (v != p && !rem[v] && sz[v] > szn / 2) return centroid(v, u, szn);
    return u;
}

void dfs_dis(int u, int p, int d = 0) {
    dis[u].push_back(d);
    for (int v : adj[u])
        if (v != p && !rem[v]) dfs_dis(v, u, d + 1);
}

void decomp(int u, int p) {
    int c = centroid(u, u, dfs_sz(u, u));

    rem[c] = true;
    par[c] = p;

    dfs_dis(c, c);

    // Faz algo na subárvore de c

    for (int v : adj[c])
        if (!rem[v]) decomp(v, c);
}

void build(int n) {
    for (int i = 0; i < n; i++) {
        rem[i] = false;
        dis[i].clear();
    }
    decomp(0, -1);
    for (int i = 0; i < n; i++) reverse(dis[i].begin(), dis[i].end());
}
```

5.7 Ciclos

5.7.1 Find Cycle

Encontra um ciclo no grafo em $\mathcal{O}(|V| + |E|)$, retorna um vetor vazio caso nenhum ciclo seja encontrado. O método `build` possui uma flag que indica se o algoritmo deve aceitar ciclos de tamanho 1 ou ciclos de tamanho 2.

Codigo: find_cycle.cpp

```cpp
const int MAXN = 1e6 + 6;

vector<int> adj[MAXN];

struct CycleFinder {
    int n;
    bool trivial;
    vector<int> vis, par;
    int start = -1, end = -1;
    void build(int _n, bool _trivial = 1) {
        n = _n;
        trivial = _trivial;
        // trivial eh um flag que indica se o algoritmo deve aceitar ou nao
        // ciclos triviais, um ciclo trivial eh um ciclo de tamanho 1 ou 2
    }
    bool dfs(int u) {
        vis[u] = 1;
        for (int v : adj[u]) {
            if (vis[v] == 0) {
```

```
                par[v] = u;
                if (dfs(v)) return true;
            } else if (vis[v] == 1) {
                if (trivial || (par[u] != v && u != v)) {
                    end = u;
                    start = v;
                    return true;
                }
            }
        }
        vis[u] = 2;
        return false;
    }
    vector<int> get_cycle() {
        vis.assign(n, 0);
        par.assign(n, -1);
        for (int v = 0; v < n; v++)
            if (vis[v] == 0 && dfs(v)) break;
        vector<int> cycle;
        if (start != -1) {
            cycle.emplace_back(start);
            for (int v = end; v != start; v = par[v]) cycle.emplace_back(v);
            cycle.emplace_back(start);
            reverse(cycle.begin(), cycle.end());
        }
        return cycle;
    }
} finder;
```

5.7.2 Find Negative Cycle

Encontra um ciclo com soma negativa no grafo em $\mathcal{O}(|V| * |E|)$ usando o algoritmo Bellman Ford, retorna um vetor vazio caso nenhum ciclo seja encontrado.

Codigo: find_negative_cycle.cpp

```
struct NegativeCycleFinder {
    const ll INF = 1e18;
    int n;
    vector<tuple<int, int, ll>> edges;
    void build(int _n) {
        n = _n;
        edges.clear();
    }
    void add_edge(int u, int v, ll w) { edges.emplace_back(u, v, w); }
    vector<int> get_cycle() {
        vector<ll> d(n);
        vector<int> p(n, -1);
        int x;
        for (int i = 0; i < n; ++i) {
            x = -1;
            for (auto [u, v, w] : edges) {
                if (d[u] < INF) {
                    if (d[u] + w < d[v]) {
                        d[v] = max(-INF, d[u] + w);
                        p[v] = u;
                        x = v;
                    }
                }
            }
        }
        vector<int> cycle;
        if (x != -1) {
            for (int i = 0; i < n; ++i) x = p[x];
            for (int v = x;; v = p[v]) {
                cycle.push_back(v);
                if (v == x && cycle.size() > 1) break;
            }
            reverse(cycle.begin(), cycle.end());
        }
        return cycle;
    }
} finder;
```

5.8 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 $\mathcal{O}(sqrt(V)*E)$

Guarda o grafo internamente, as arestas devem ser adicionadas pela função `add_edge`.

A função `max_flow` modifica o grafo adicionando a maior quantidade de fluxo possivel e retona a quantidade de fluxo adicionado.

O corte minimo de um grafo é equivalente ao fluxo máximo.

A Função `min_cut` acha as arestas pertencentes ao corte minimo do grafo, deve ser chamado após a função `max_flow`

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

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

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

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

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

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

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

```
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;
}
void min_cut() {
    vector<bool> vis(n);
    function<void(int)> dfs = [&](int u) {
        vis[u] = 1;
        for (int id : adj[u]) {
            int v = edges[id].v;
            if (!vis[v] && edges[id].cap - edges[id].flow > 0) dfs(v);
        }
    };
    dfs(s);
    for (int id = 0; id < (int)edges.size(); id++) {
        auto [u, v, cap, flow] = edges[id];
        if (vis[u] ^ vis[v] && cap > 0) {
            // this edge is in the min cut
            // do something here
        }
    }
}
};
```

5.9 HLD

5.9.1 HLD Aresta

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. Ao mudar a estrutura, pode ser necessário adaptar os métodos `query` e `update` da HLD.

A HLD pode ser feita com os valores estando tanto nos vértices quanto nas arestas, essa implementação é feita com os valores nas **arestas**, para ter os valores nos vértices, consulte a implementação de HLD em vértices.

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_edge.cpp

```
const int N = 3e5 + 5;

vector<pair<int, ll>> adj[N];

namespace HLD {
    int t, sz[N], pos[N], par[N], head[N];
    SegTree seg; // por padrao, a HLD esta codada para usar a SegTree lazy,
                 // mas pode usar qualquer estrutura de dados aqui
```

```
    void dfs_sz(int u, int p = -1) {
        sz[u] = 1;
        for (int i = 0; i < (int)adj[u].size(); i++) {
            auto &[v, w] = adj[u][i];
            if (v != p) {
                dfs_sz(v, u);
                sz[u] += sz[v];
                if (sz[v] > sz[adj[u][0].first] || adj[u][0].first == p)
                    swap(adj[u][0], adj[u][i]);
            }
        }
    }
    void dfs_hld(int u, int p = -1) {
        pos[u] = t++;
        for (auto [v, w] : adj[u]) {
            if (v != p) {
                par[v] = u;
                head[v] = (v == adj[u][0].first ? head[u] : v);
                dfs_hld(v, u);
            }
        }
    }
    void build_hld(int u) {
        dfs_sz(u);
        t = 0, par[u] = u, head[u] = u;
        dfs_hld(u);
    }
    void build(int n, int root) {
        build_hld(root);
        vector<ll> aux(n, seg.neutral);
        for (int u = 0; u < n; u++) {
            for (auto [v, w] : adj[u])
                if (u == par[v]) aux[pos[v]] = w;
        }
        seg.build(aux);
    }
    ll query(int u, int v) {
        if (u == v) return seg.neutral;
        if (pos[u] > pos[v]) swap(u, v);
        if (head[u] == head[v]) {
            return seg.query(pos[u] + 1, 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 (sz[u] == 1) return seg.neutral;
        return seg.query(pos[u] + 1, pos[u] + sz[u] - 1);
    }
    // a flag repl diz se o update é de soma ou de replace
    void update(int u, int v, ll k, bool repl) {
        if (u == v) return;
        if (pos[u] > pos[v]) swap(u, v);
        if (head[u] == head[v]) {
            seg.update(pos[u] + 1, pos[v], k, repl);
        } else {
            seg.update(pos[head[v]], pos[v], k, repl);
            update(u, par[head[v]], k, repl);
        }
    }
    void update_subtree(int u, ll k, bool repl) {
        if (sz[u] == 1) return;
        seg.update(pos[u] + 1, pos[u] + sz[u] - 1, k, repl);
    }
    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]]);
    }
}
```

5.9.2 HLD Vértice

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. Ao mudar a estrutura, pode ser necessário adaptar os métodos `query` e `update` da HLD.

A HLD pode ser feita com os valores estando tanto nos vértices quanto nas arestas, essa implementação é feita com os valores nos **vértices**, para ter os valores nas arestas, consulte a implementação de HLD em arestas.

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

```cpp
const int N = 3e5 + 5;

vector<int> adj[N];

namespace HLD {
    int t, sz[N], pos[N], par[N], head[N];
    SegTree seg; // por padrao, a HLD esta codada para usar a SegTree lazy,
                 // mas 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) {
        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) {
        dfs_sz(u);
        t = 0, par[u] = u, head[u] = u;
        dfs_hld(u);
    }
    void build(vector<ll> v, int root) { // pra buildar com valores nos nodos
        build_hld(root);
        vector<ll> aux(v.size());
        for (int i = 0; i < (int)v.size(); i++) aux[pos[i]] = v[i];
        seg.build(aux);
    }
    void build(int n, int root) { // pra buildar com neutro nos nodos
        build(vector<ll>(n, seg.neutral), root);
    }
    void build(ll *bg, ll *en, int root) { // pra buildar com array de C
        build(vector<ll>(bg, en), root);
    }
    ll query(int u, int v) {
        if (pos[u] > pos[v]) swap(u, v);
        if (head[u] == head[v]) {
            return seg.query(pos[u], 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) { return seg.query(pos[u], pos[u] + sz[u] - 1); }
    // a flag repl diz se o update é de soma ou de replace
    void update(int u, int v, ll k, bool repl) {
        if (pos[u] > pos[v]) swap(u, v);
        if (head[u] == head[v]) {
            seg.update(pos[u], pos[v], k, repl);
        } else {
            seg.update(pos[head[v]], pos[v], k, repl);
            update(u, par[head[v]], k, repl);
        }
    }
    void update_subtree(int u, ll k, bool repl) {
        seg.update(pos[u], pos[u] + sz[u] - 1, k, repl);
    }
    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]]);
    }
}
```

5.10 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, em $\mathcal{O}(N \cdot \log N + N \cdot \log M)$.

Codigo: inverse_graph.cpp

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

```
3
4  void bfs(int s) {
5      queue<int> f;
6      f.push(s);
7      nodes.erase(s);
8      set<int> aux;
9      while (!f.empty()) {
10         int x = f.front();
11         f.pop();
12         for (int y : nodes)
13             if (adj[x].count(y) == 0) aux.insert(y);
14         for (int y : aux) {
15             f.push(y);
16             nodes.erase(y);
17         }
18         aux.clear();
19     }
20 }
```

5.11 Kosaraju

Algoritmo que encontra as componentes fortemente conexas (SCCs) de um grafo direcionado.

O algoritmo de Kosaraju resolve isso em $\mathcal{O}(N + M)$, onde N é o número de vértices e M é o número de arestas do grafo.

O componente fortemente conexo de cada vértice é armazenado no vetor `root`.

A grafo condensado é armazenado no vetor `gc`.

Codigo: kosaraju.cpp

```
1  namespace kosaraju {
2      const int N = 1e5 + 5;
3      int n, vis[N], root[N];
4      vector<int> adj[N], inv[N], gc[N], topo;
5      void add_edge(int u, int v) {
6          adj[u].emplace_back(v);
7          inv[v].emplace_back(u);
8      }
9      void toposort(int u) {
10         vis[u] = 1;
11         for (auto v : adj[u]) {
12             if (vis[v]) continue;
13             toposort(v);
14         }
15         topo.emplace_back(u);
16     }
17     void dfs(int u) {
18         vis[u] = 1;
19         for (auto v : inv[u]) {
20             if (vis[v]) continue;
21             root[v] = root[u];
22             dfs(v);
23         }
24     }
25     void solve(int n) {
26         fill(vis, vis + n, 0);
27         topo.clear();
28         for (int i = 0; i < n; i++)
29             if (!vis[i]) toposort(i);
30         fill(vis, vis + n, 0);
31         iota(root, root + n, 0);
32         for (int i = n - 1; i >= 0; i--)
33             if (!vis[topo[i]]) dfs(topo[i]);
34         set<pair<int, int>> st;
35         for (int u = 0; u < n; u++) {
36             int ru = root[u];
37             for (int v : adj[u]) {
38                 int rv = root[v];
39                 if (ru == rv) continue;
40                 if (!st.count(ii(ru, rv))) {
41                     gc[ru].emplace_back(rv);
42                     st.insert(ii(ru, rv));
43                 }
44             }
45         }
46     }
47 }
```

5.12 Kruskal

Algoritimo que utiliza DSU (Disjoint Set Union, descrita na seção de Estrutura de Dados) para encontrar a MST (Minimum Spanning Tree) de um grafo em $\mathcal{O}(E \log E)$.

A Minimum Spanning Tree é a árvore geradora mínima de um grafo, ou seja, um conjunto de arestas que conecta todos os nodos do grafo com o menor custo possível.

Propriedades importantes da MST:

- É uma árvore! :O
- Entre quaisquer dois nodos u e v do grafo, a MST minimiza a maior aresta no caminho de u a v.

Ideia do Kruskal: ordenar as arestas do grafo por peso e, para cada aresta, adicionar ela à MST se ela não forma um ciclo com as arestas já adicionadas.

Codigo: kruskal.cpp

```cpp
vector<tuple<int, int, int>> edges; // {u, v, w}

void kruskal(int n) {
    DSU dsu(n); // DSU da seção Estruturas de Dados

    sort(edges.begin(), edges.end(), [](auto a, auto b) {
        return get<2>(a) < get<2>(b);
    });

    for (auto [u, v, w] : edges) {
        if (dsu.unite(u, v)) {
            // edge u-v is in the MST
        }
    }
}
```

5.13 LCA

Algoritmo para computar Lowest Common Ancestor usando Euler Tour e Sparse Table (descrita na seção Estruturas de Dados), com pré-processamento em $\mathcal{O}(N \log N)$ e consulta em $\mathcal{O}(1)$.

Codigo: lca.cpp

```cpp
const int N = 5e5 + 5;
int timer, tin[N];
vector<int> adj[N];
vector<pair<int, int>> prof;

struct SparseTable {
    int n, LG;
    using T = pair<int, int>;
    vector<vector<T>> st;
    T merge(T a, T b) { return min(a, b); }
    const T neutral = {INT_MAX, -1};
    void build(const vector<T> &v) {
        n = (int)v.size();
        LG = 32 - __builtin_clz(n);
        st = vector<vector<T>>(LG, vector<T>(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)]);
    }
    T 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]);
    }
} st_lca;

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);
        }
    }
    timer++;
}

int lca(int u, int v) {
    int l = tin[u], r = tin[v];
    if (l > r) swap(l, r);
    return st_lca.query(l, r).second;
}
```

```
45
46 void build() {
47     timer = 0;
48     prof.clear();
49     et_dfs(0, -1, 0);
50     st_lca.build(prof);
51 }
```

5.14 Matching

5.14.1 Hungaro

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

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

A matriz de entrada precisa ser indexada em 1

O vetor `result` guarda os pares do matching.

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

Codigo: hungarian.cpp

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

5.15 Pontes

5.15.1 Componentes Aresta Biconexas

Código que acha componentes aresta-biconexas, que são componentes que para se desconectar é necessário remover pelo menos duas arestas. Para obter essas componentes, basta achar as pontes e contrair o resto do grafo, o resultado é uma árvore em que as arestas são as pontes do grafo original.

Esse algoritmo acha as pontes e constrói o grafo comprimido em $\mathcal{O}(V + E)$. Pontes são arestas cuja remoção aumenta o número de componentes conexas do grafo.

No código, `ebcc[u]` é o índice da componente aresta-biconexa a qual o vértice `u` pertence.

Codigo: ebcc_components.cpp

```
1 const int N = 3e5 + 5;
2 int n, m, timer, ncc;
3 vector<int> adjbcc[N];
4 vector<int> adj[N];
5 int tin[N], low[N], ebcc[N];
```

```
6
7  void dfs_bridge(int u, int p = -1) {
8      low[u] = tin[u] = ++timer;
9      for (int v : adj[u]) {
10         if (tin[v] != 0 && v != p) {
11             low[u] = min(low[u], tin[v]);
12         } else if (v != p) {
13             dfs_bridge(v, u);
14             low[u] = min(low[u], low[v]);
15         }
16     }
17 }
18
19 void dfs_ebcc(int u, int p, int cc) {
20     if (p != -1 && low[u] == tin[u]) {
21         // edge (u, p) eh uma ponte
22         cc = ++ncc;
23     }
24     ebcc[u] = cc;
25     for (int v : adj[u])
26         if (ebcc[v] == -1) dfs_ebcc(v, u, cc);
27 }
28
29 void build_ebcc_graph() {
30     ncc = timer = 0;
31     for (int i = 0; i < n; i++) {
32         tin[i] = low[i] = 0;
33         ebcc[i] = -1;
34         adjbcc[i].clear();
35     }
36     for (int i = 0; i < n; i++)
37         if (tin[i] == 0) dfs_bridge(i);
38     for (int i = 0; i < n; i++)
39         if (ebcc[i] == -1) dfs_ebcc(i, -1, ncc), ++ncc;
40     // Opcao 1 - constroi o grafo comprimido passando por todas as edges
41     for (int u = 0; u < n; u++) {
42         for (auto v : adj[u]) {
43             if (ebcc[u] != ebcc[v]) {
44                 adjbcc[ebcc[u]].emplace_back(ebcc[v]);
45             } else {
46                 // faz algo
47             }
48         }
49     }
50     // Opcao 2 - constroi o grafo comprimido passando so pelas pontes
51     // for (auto [u, v] : bridges) {
52     // adjbcc[ebcc[u]].emplace_back(ebcc[v]);
53     // adjbcc[ebcc[v]].emplace_back(ebcc[u]);
54     // }
55 }
```

5.15.2 Pontes

Algoritmo que acha pontes em um grafo utilizando DFS. $\mathcal{O}(V + E)$. Pontes são arestas cuja remoção aumenta o número de componentes conexas do grafo.

Codigo: find_bridges.cpp

```
1  const int N = 3e5 + 5;
2  int n, m, timer;
3  vector<int> adj[N];
4  int tin[N], low[N];
5
6  void dfs_bridge(int u, int p = -1) {
7      low[u] = tin[u] = ++timer;
8      for (int v : adj[u]) {
9          if (tin[v] != 0 && v != p) {
10             low[u] = min(low[u], tin[v]);
11         } else if (v != p) {
12             dfs_bridge(v, u);
13             low[u] = min(low[u], low[v]);
14         }
15     }
16     if (p != -1 && low[u] == tin[u]) {
17         // edge (p, u) eh ponte
18     }
19 }
20
21 void find_bridges() {
22     timer = 0;
23     for (int i = 0; i < n; i++) tin[i] = low[i] = 0;
24     for (int i = 0; i < n; i++)
25         if (tin[i] == 0) dfs_bridge(i);
26 }
```

5.16 Pontos de Articulacao

Algoritmo que acha pontos de articulação em um grafo utilizando DFS. $\mathcal{O}(V+E)$. Pontos de articulação são nodos cuja remoção aumenta o número de componentes conexas do grafo.

Codigo: articulation_points.cpp

```cpp
const int N = 3e5 + 5;
int n, m, timer;
vector<int> adj[N];
int tin[N], low[N];

void dfs(int u, int p = -1) {
    low[u] = tin[u] = ++timer;
    int child = 0;
    for (int v : adj[u]) {
        if (tin[v] != 0 && v != p) {
            low[u] = min(low[u], tin[v]);
        } else if (v != p) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (p != -1 && low[v] >= tin[u]) {
                // vertice u eh um ponto de articulacao
            }
            child++;
        }
    }
    if (p == -1 && child > 1) {
        // vertice u eh um ponto de articulacao
    }
}

void find_articulation_points() {
    timer = 0;
    for (int i = 0; i < n; i++) tin[i] = low[i] = 0;
    for (int i = 0; i < n; i++)
        if (tin[i] == 0) dfs(i);
}
```

5.17 Shortest Paths

5.17.1 01 BFS

Computa o menor caminho entre nodos de um grafo com arestas de peso 0 ou 1.

Dado um nodo s, computa o menor caminho de s para todos os outros nodos em $\mathcal{O}(V+E)$.

Muito semelhante a uma BFS, mas usa uma `deque` (fila dupla) ao invés de uma fila comum.

Importante: As arestas só podem ter peso 0 ou 1.

Codigo: bfs01.cpp

```cpp
const int N = 3e5 + 5;
const int INF = 1e9;

int n;
vector<pair<int, int>> adj[N];

vector<int> bfs01(int s) {
    vector<int> dist(n, INF);
    deque<int> q;
    dist[s] = 0;
    q.emplace_back(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop_front();
        for (auto [w, v] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (w == 0) q.push_front(v);
                else q.push_back(v);
            }
        }
    }
    return dist;
}
```

5.17.2 BFS

Computa o menor caminho entre nodos de um grafo com arestas de peso 1.

Dado um nodo s, computa o menor caminho de s para todos os outros nodos em $\mathcal{O}(V + E)$.

Importante: Todas arestas do grafo devem ter peso 1.

Codigo: bfs.cpp

```cpp
const int N = 3e5 + 5;

int n;
vector<int> adj[N];

vector<int> bfs(int s) {
    vector<int> dist(n, -1);
    queue<int> q;
    dist[s] = 0;
    q.emplace(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : adj[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.emplace(v);
            }
        }
    }
    return dist;
}
```

5.17.3 Bellman Ford

Encontra o caminho mais curto entre um nodo e todos os outros nodos de um grafo em $\mathcal{O}(|V| * |E|)$.

Importante: Detecta ciclos negativos.

Codigo: bellman_ford.cpp

```cpp
const ll INF = 1e18;

int n;
vector<tuple<int, int, int>> edges;

vector<ll> bellman_ford(int s) {
    vector<ll> dist(n, INF);
    dist[s] = 0;
    for (int i = 0; i < n; i++) {
        for (auto [u, v, w] : edges)
            if (dist[u] < INF) dist[v] = min(dist[v], dist[u] + w);
    }
    for (int i = 0; i < n; i++) {
        for (auto [u, v, w] : edges)
            if (dist[u] < INF && dist[u] + w < dist[v]) dist[v] = -INF;
    }
    // dist[u] = -INF se tem um ciclo negativo que chega em u
    return dist;
}
```

5.17.4 Dijkstra

Computa o menor caminho entre nodos de um grafo com pesos quaisquer nas arestas.

Dado um nodo s, computa o menor caminho de s para todos os outros nodos em $\mathcal{O}((V + E) \cdot \log E)$.

Muito semelhante a uma BFS, mas usa uma fila de prioridade ao invés de uma fila comum.

Importante: O grafo não pode conter arestas de peso negativo.

Codigo: dijkstra.cpp

```cpp
const int N = 3e5 + 5;
const ll INF = 1e18;

int n;
vector<pair<int, int>> adj[N];

vector<ll> dijkstra(int s) {
    vector<ll> dist(n, INF);
    using T = pair<ll, int>;
    priority_queue<T, vector<T>, greater<>> pq;
```

```
    dist[s] = 0;
    pq.emplace(dist[s], s);
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (d != dist[u]) continue;
        for (auto [w, v] : adj[u]) {
            if (dist[v] > d + w) {
                dist[v] = d + w;
                pq.emplace(dist[v], v);
            }
        }
    }
    return dist;
}
```

5.17.5 Floyd Warshall

Algoritmo que encontra o menor caminho entre todos os pares de nodos de um grafo com pesos em $\mathcal{O}(N^3)$.

A ideia do algoritmo é: para cada nodo k, passamos por todos os pares de nodos (i, j) e verificamos se é mais curto passar por k para ir de i a j do que o caminho atual de i a j. Se for, atualizamos o caminho.

Codigo: floyd_warshall.cpp

```
const int N = 3e3 + 5;
const ll INF = 1e18;
int n;

ll adj[N][N]; // adj[u][v] = peso da aresta u-v, INF se não existe
ll dist[N][N];

void floydwarshall() {
    for (int u = 0; u < n; u++)
        for (int v = 0; v < n; v++) dist[u][v] = adj[u][v];
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
    }
}
```

5.17.6 SPFA

Encontra o caminho mais curto entre um nodo e todos os outros nodos de um grafo em $\mathcal{O}(|V| * |E|)$. Na prática, é bem mais rápido que o Bellman-Ford.

Detecta ciclos negativos.

Codigo: spfa.cpp

```
const int N = 1e4 + 5;
const ll INF = 1e18;

int n;
vector<pair<int, int>> adj[N];

vector<ll> spfa(int s) {
    vector<ll> dist(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(s);
    inq[s] = true;
    dist[s] = 0;
    while (!q.empty()) {
        int u = q.front();
        q.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]) {
                    q.push(v);
                    inq[v] = true;
                    cnt[v]++;
                    if (cnt[v] > n) dist[v] = -INF; // negative cycle
                }
            }
        }
    }
    return dist;
}
```

5.18 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 nodos mais intensos até que o grafo contenha apenas dois conjuntos de nodos 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;
}
```

5.19 Virtual Tree

Dado um conjunto de nodos S, cria uma árvore com todos os nodos do conjunto e os `LCA` de todos os pares de nodos

desse conjunto em $\mathcal{O}(|S| \cdot \log |S|)$.

Obs: Precisa do código de LCA encontrado em `Grafos/Binary-Lifting-LCA`.

Codigo: virtual_tree.cpp

```cpp
const int N = 3e5 + 5;
#warning nao esquece de copiar o codigo de LCA
vector<int> vir_tree[N];
vector<int> vir_nodes;

void build_virtual_tree(vector<int> S) {
    sort(S.begin(), S.end(), [&](int i, int j) { return bl::tin[i] < bl::tin[j]; });
    for (int i = 1; i < (int)S.size(); i++) S.emplace_back(bl::lca(S[i - 1], S[i]));
    sort(S.begin(), S.end(), [&](int i, int j) { return bl::tin[i] < bl::tin[j]; });
    S.erase(unique(S.begin(), S.end()), S.end());
    vir_nodes = S;
    for (auto u : S) vir_tree[u].clear();
    vector<int> stk;
    for (auto u : S) {
        while (stk.size() && !bl::ancestor(stk.back(), u)) stk.pop_back();
        if (stk.size()) vir_tree[stk.back()].emplace_back(u);
        stk.emplace_back(u);
    }
}
```

Capítulo 6

Primitivas

6.1 Modular Int

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

Ao usar o Mint, você deve passar os valores pra ele **já modulados**, ou seja, valores entre $-\text{MOD}$ e $\text{MOD} - 1$, o próprio Mint normaliza depois para ficar entre 0 e $\text{MOD} - 1$.

Para lembrar as propriedades de aritmética modular, consulte a seção Teórico desse Almanaque.

Para usar o Mint, basta criar um tipo com o valor de MOD desejado. O valor de MOD deve ser um número inteiro positivo, podendo ser tanto do tipo `int` quanto `long long`.

```
using mint = Mint<7>;
// using mint = Mint<(ll)1e18 + 9> para long long
mint a = 4, b = 3;
mint c = a * b; // c.v == 5
mint d = 1 / a; // d.v == 2, MOD deve ser primo para usar o operador de divisão
mint e = a * d // e.v == 1
a = a + 2; // a.v == 6
a = a + 3; // a.v == 2
a = a ^ 5; // a.v == 4
a = a - 6; // a.v == 5
```

Obs: para operador de divisão, o Mint usa o inverso multiplicativo de a baseado no Teorema de Euler (consulte o Teórico para mais detalhes), que é $a^{\text{MOD}-2}$, ou seja, para isso o MOD deve ser primo.

Codigo: mint.cpp

```
template <auto MOD, typename T = decltype(MOD)>
struct Mint {
    using U = long long;
    // se o modulo for long long, usar U = __int128
    using m = Mint<MOD, T>;
    T v;
    Mint(T val = 0) : v(val) {
        assert(sizeof(T) * 2 <= sizeof(U));
        if (v < -MOD || v >= 2 * MOD) v %= MOD;
        if (v < 0) v += MOD;
        if (v >= MOD) v -= MOD;
    }
    Mint(U val) : v(T(val % MOD)) {
        assert(sizeof(T) * 2 <= sizeof(U));
        if (v < 0) v += MOD;
    }
    bool operator==(m o) const { return v == o.v; }
    bool operator<(m o) const { return v < o.v; }
    bool operator!=(m o) const { return v != o.v; }
    m pwr(m b, U e) {
        m res = 1;
        while (e > 0) {
            if (e & 1) res *= b;
            b *= b, e /= 2;
        }
        return res;
    }
    m &operator+=(m o) {
```

```
        v -= MOD - o.v;
        if (v < 0) v += MOD;
        return *this;
    }
    m &operator-=(m o) {
        v -= o.v;
        if (v < 0) v += MOD;
        return *this;
    }
    m &operator*=(m o) {
        v = (T)((U)v * o.v % MOD);
        return *this;
    }
    m &operator/=(m o) { return *this *= o.pwr(o, MOD - 2); }
    m &operator^=(U e) { return *this = pwr(*this, e); }
    friend m operator-(m a, m b) { return a -= b; }
    friend m operator+(m a, m b) { return a += b; }
    friend m operator*(m a, m b) { return a *= b; }
    friend m operator/(m a, m b) { return a /= b; }
    friend m operator^(m a, U e) { return a.pwr(a, e); }
};
```

6.2 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

```
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 7

Estruturas de Dados

7.1 Disjoint Set Union

7.1.1 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

```cpp
struct DSU {
    vector<int> par, sz;
    void build(int n) {
        par.assign(n, 0);
        iota(par.begin(), par.end(), 0);
        sz.assign(n, 1);
    }
    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;
        if (sz[a] < sz[b]) swap(a, b);
        par[b] = a;
        sz[a] += sz[b];
        return true;
    }
};
```

7.1.2 DSU Bipartido

DSU que mantém se um conjunto é bipartido (visualize os conjuntos como componentes conexas de um grafo e os elementos como nodos). 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

```cpp
struct Bipartite_DSU {
    vector<int> par, sz, c, bip;
    bool all_bipartite;
    void build(int n) {
        par.assign(n, 0);
        iota(par.begin(), par.end(), 0);
        sz.assign(n, 1);
        c.assign(n, 0);
        bip.assign(n, 1);
        all_bipartite = 1;
    }
```

```
    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);
        par[b] = a;
        sz[a] += sz[b];
        if (equal_color) c[b] = 1;
        bip[a] &= bip[b];
        all_bipartite &= bip[a];
        return true;
    }
};
```

7.1.3 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.

Para alterar uma variável da DSU durante o unite, deve-se usar o método `change`, pois ele coloca as alterações numa stack para depois poder revertê-las.

Codigo: rollback_dsu.cpp

```
struct Rollback_DSU {
    vector<int> par, sz;
    stack<stack<pair<int &, int>>> changes;
    void build(int n) {
        par.assign(n, 0);
        sz.assign(n, 1);
        iota(par.begin(), par.end(), 0);
        while (changes.size()) changes.pop();
        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(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();
    }
};
```

7.1.4 DSU Rollback Bipartido

DSU com rollback e bipartido.

Codigo: bipartite_rollback_dsu.cpp

```
struct BipartiteRollback_DSU {
    vector<int> par, sz, c, bip;
    int all_bipartite;
    stack<stack<pair<int &, int>>> changes;
    void build(int n) {
        par.assign(n, 0);
        iota(par.begin(), par.end(), 0);
        sz.assign(n, 1);
        c.assign(n, 0);
```

```
        bip.assign(n, 1);
        all_bipartite = true;
        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(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();
    }
};
```

7.1.5 Offline DSU

Algoritmo que utiliza o 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 : BipartiteRollback_DSU {
    int time;
    void build(int n) {
        BipartiteRollback_DSU::build(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);
        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;
    }
};
```

7.2 Fenwick Tree

7.2.1 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

```
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;
```

```
    }
};
```

7.2.2 Kd Fenwick Tree

Fenwick Tree em k dimensões. Faz apenas queries de prefixo e updates pontuais em $\mathcal{O}(k \cdot \log^k n)$. Para queries em range, deve-se fazer inclusão-exclusão, porém a complexidade fica exponencial, para k dimensões a query em range é $\mathcal{O}(2^k \cdot k \cdot \log^k n)$.

Codigo: kd_fenwick_tree.cpp

```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;
    }
}
```

7.3 Implicit Treap

Simula um array com as seguintes operações em $\mathcal{O}(\log N)$:

- Inserir um elemento X na posição i (todos os elementos em posições maiores que i serão "empurrados"para a direita).
- Remover o elemento na posição i (todos os elementos em posições maiores que i serão "puxados"para a esquerda).
- Query em intervalo $[L, R]$ de alguma operação. Pode ser soma, máximo, mínimo, `gcd`, etc.
- Adição em intervalo $[L, R]$ (sua operação deve suportar propagação lazy).
- Reverter um intervalo $[L, R]$, ou seja, $a[L], a[L+1], \cdots, a[R] \to a[R], a[R-1], \cdots, a[L]$.

Obs: Inserir em uma posição < 0 vai inserir na posição 0, assim como inserir em uma posição $>$ Tamanho da Treap vai inserir no final dela.

Codigo: implicit_treap.cpp

```cpp
mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
namespace imp_treap {
    using T = ll; // mudar pra int se nao precisar pra melhorar a performance
    T merge(T a, T b) { return a + b; }
    T neutral = 0;
    struct node_info {
        node_info *l, *r;
        int y, size;
        T val, acc, add;
        bool rev;
        node_info() { }
        node_info(T _val)
            : l(0), r(0), y(rng()), size(0), val(_val), acc(0), add(0), rev(false) { }
    };
    using node = node_info *;
    node root = 0;
    inline int size(node t) { return t ? t->size : 0; }
    inline T acc(node t) { return t ? t->acc : 0; }
    inline bool rev(node t) { return t ? t->rev : false; }
    inline void push(node t) {
```

```
        if (!t) return;
        if (rev(t)) {
            t->rev = false;
            swap(t->l, t->r);
            if (t->l) t->l->rev ^= 1;
            if (t->r) t->r->rev ^= 1;
        }
        t->acc += t->add * size(t);
        // t->acc += t->add se for RMQ
        t->val += t->add;
        if (t->l) t->l->add += t->add;
        if (t->r) t->r->add += t->add;
        t->add = 0;
    }
    inline void pull(node t) {
        if (t) {
            push(t->l), push(t->r);
            t->size = size(t->l) + size(t->r) + 1;
            t->acc = merge(t->val, merge(acc(t->l), acc(t->r)));
        }
    }
    void merge(node &t, node L, node R) {
        push(L), push(R);
        if (!L || !R) {
            t = L ? L : R;
        } else if (L->y > R->y) {
            merge(L->r, L->r, R);
            t = L;
        } else {
            merge(R->l, L, R->l);
            t = R;
        }
        pull(t);
    }
    void split(node t, int pos, node &L, node &R, int add = 0) {
        if (!t) {
            L = R = nullptr;
        } else {
            push(t);
            int imp_key = add + size(t->l);
            if (pos <= imp_key) {
                split(t->l, pos, L, t->l, add);
                R = t;
            } else {
                split(t->r, pos, t->r, R, imp_key + 1);
                L = t;
            }
        }
        pull(t);
    }
    inline void insert(node to, int pos) {
        node L, R;
        split(root, pos, L, R);
        merge(L, L, to);
        merge(root, L, R);
    }
    bool remove(node &t, int pos, int add = 0) {
        if (!t) return false;
        push(t);
        int imp_key = add + size(t->l);
        if (pos == imp_key) {
            node me = t;
            merge(t, t->l, t->r);
            delete me;
            return true;
        }
        bool ok;
        if (pos < imp_key) ok = remove(t->l, pos, add);
        else ok = remove(t->r, pos, imp_key + 1);
        pull(t);
        return ok;
    }
    inline T query(int l, int r) {
        if (l > r) return neutral;
        node L1, L2, R1, R2;
        split(root, r + 1, L1, R1);
        split(L1, l, L2, R2);
        T ans = acc(R2);
        merge(L1, L2, R2);
        merge(root, L1, R1);
        return ans;
    }
    inline void update_sum(int l, int r, T val) {
        if (l > r) return;
        node L1, L2, R1, R2;
        split(root, r + 1, L1, R1);
        split(L1, l, L2, R2);
        assert(R2);
        R2->add += val;
        merge(L1, L2, R2);
        merge(root, L1, R1);
    }
    inline void reverse(int l, int r) {
        if (l > r) return;
```

```
        node L1, L2, R1, R2;
        split(root, r + 1, L1, R1);
        split(L1, l, L2, R2);
        R2->rev ^= 1;
        merge(L1, L2, R2);
        merge(root, L1, R1);
    }
    inline void insert(int pos, int val) { insert(new node_info(val), pos); }
    inline bool remove(int pos) { return remove(root, pos); }
}
```

7.4 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;
```

7.5 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 $\mathcal{O}(\log L)$.

Dica: No construtor da LiChao Tree, fazer `tree.reserve(MAX); L.reserve(MAX); R.reserve(MAX);` pode ajudar bastante no runtime, pois aloca espaço para os vetores e evita muitas realocações durante a execução. Nesse caso, `MAX` é geralmente $\mathcal{O}(Q \cdot \log L)$, onde Q é o número de queries e L é o tamanho do intervalo.

Codigo: lichao_tree.cpp

```cpp
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; }
};

template <ll MINL = ll(-1e9 - 5), ll MAXR = ll(1e9 + 5)>
struct LichaoTree {
    vector<Line> tree;
    vector<int> L, R;

    int newnode() {
        tree.push_back(Line());
        L.push_back(0);
        R.push_back(0);
        return int(tree.size()) - 1;
    }

    LichaoTree() {
        newnode();
        newnode();
    }

    int lc(int p, bool create = false) {
        if (create && L[p] == 0) L[p] = newnode();
        return L[p];
    }

    int rc(int p, bool create = false) {
        if (create && R[p] == 0) R[p] = newnode();
        return R[p];
    }

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

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

7.6 Merge Sort Tree

7.6.1 Merge Sort Tree

Árvore muito semelhante a uma Segment Tree, mas ao invés de armazenar um valor em cada nodo, 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 $A \leq X \leq B$.

Obs: o método `kth` presente nessa implementação encontra o k-ésimo elemento no intervalo $[L, R]$ em $\mathcal{O}(\log^3 N)$. É possível otimizar esse método para $\mathcal{O}(\log^2 N)$, basta se criar um vetor que possui pares da forma `[A[i], i]` e ordená-lo de acordo com o valor de `A[i]`, agora, construa a Merge Sort Tree com esse vetor e no merge faça a união mantendo os valores de `i` ordenados. Dessa forma, sua Merge Sort Tree guardará em um nodo que representa o intervalo $[L, R]$ os índices ordenados de todos os elementos que estão entre o $(L+1)$-ésimo e o $(R+1)$-ésimo menor elemento do vetor original. Assim, para encontrar o k-ésimo elemento no intervalo $[L, R]$ basta fazer uma busca binária semelhante a busca binária de encontrar k-ésimo menor elemento em uma Segment Tree.

Codigo: mergesort_tree.cpp

```cpp
template <typename T = int>
struct MergeSortTree {
    vector<vector<T>> tree;
    int n;
    int lc(int u) { return u << 1; }
    int rc(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(lc(u), l, mid, a);
        build(rc(u), mid + 1, r, a);
        merge(
            tree[lc(u)].begin(),
            tree[lc(u)].end(),
            tree[rc(u)].begin(),
            tree[rc(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) {
        if (l > R || r < L || a > b) return 0;
        if (l >= L && r <= R) {
            auto ub = upper_bound(tree[u].begin(), tree[u].end(), b);
            auto lb = upper_bound(tree[u].begin(), tree[u].end(), a - 1);
            return (int)(ub - lb);
        }
        int mid = (l + r) >> 1;
        return count(lc(u), l, mid, L, R, a, b) + count(rc(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];
    }
};
```

7.6.2 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.

Obs: usa a estrutura `ordered_set`, descrita nesse Almanaque também.

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 lc(int u) { return u << 1; }
    int rc(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(lc(u), l, mid, a);
        build(rc(u), mid + 1, r, a);
        for (auto x : tree[lc(u)]) tree[u].insert(x);
        for (auto x : tree[rc(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(lc(u), l, mid, L, R, a, b) + count(rc(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(lc(u), l, mid, i, x);
            else update(rc(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); }
};
```

7.7 Operation Deque

Deque que armazena o resultado do operatório dos itens (ou seja, dado um deque, responde qual é o elemento mínimo, por exemplo). O deque possui a operação `get` que retorna o resultado do operatório dos itens do deque em $\mathcal{O}(1)$ amortizado. Chamar o método `get` em um deque vazia é indefinido.

Obs: usa a estrutura Operation Stack (também descrita nesse Almanaque).

Codigo: op_deque.cpp

```
template <typename T, auto OP>
struct op_deque {
    op_stack<T, OP> in, out;
    void push_back(T x) { in.push(x); }
    void push_front(T x) { out.push(x); }
    void work() {
        op_stack<T, OP> to;
        bool sw = false;
        if (in.empty()) sw = true, swap(in, out);
        int m = in.size();
        for (int i = 0; i < m / 2; i++) to.push(in.top()), in.pop();
        while (in.size()) out.push(in.top()), in.pop();
        while (to.size()) in.push(to.top()), to.pop();
        if (sw) swap(in, out);
    }
    T pop_front() {
        if (out.empty()) work();
        T ret = out.top();
        out.pop();
        return ret;
    }
    T pop_back() {
        if (in.empty()) work();
        T ret = in.top();
        in.pop();
        return ret;
    }
    T get() {
        if (in.empty()) return out.get();
        if (out.empty()) return in.get();
        return OP(in.get(), out.get());
    }
};
```

7.8 Operation Queue

Fila que armazena o resultado do operatório dos itens (ou seja, dado uma fila, responde qual é o elemento mínimo, por exemplo). A fila possui 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 (também descrita nesse Almanaque).

Codigo: op_queue.cpp

```cpp
template <typename T, auto OP>
struct op_queue {
    op_stack<T, OP> in, out;
    void push(T x) { in.push(x); }
    void pop() {
        if (out.empty()) {
            while (!in.empty()) {
                out.push(in.top());
                in.pop();
            }
        }
        out.pop();
    }
    T get() {
        if (out.empty()) return in.get();
        if (in.empty()) return out.get();
        return OP(in.get(), out.get());
    }
    T front() {
        if (out.empty()) return in.bottom();
        return out.top();
    }
    T back() {
        if (in.empty()) return out.bottom();
        return in.top();
    }
};
```

7.9 Operation Stack

Pilha que armazena o resultado do operatório dos itens (ou seja, dado uma pilha, responde qual é o elemento mínimo, por exemplo). A pilha possui 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.

A pilha é um template e recebe como argumentos o tipo dos itens e a função operatória. A função operatória deve receber dois argumentos do tipo dos itens e retornar um valor do mesmo tipo.

Exemplo de como passar a função operatória para a pilha:

```cpp
int f(int a, int b) { return a + b; }

void test() {
    auto g = [](int a, int b) { return a ^ b; };

    op_stack<int, f> st;
    op_stack<int, g> st2;

    st.push(1);
    st.push(1);
    st2.push(1);
    st2.push(1);
    cout << st.get() << endl; // 2
    cout << st2.get() << endl; // 0
}
```

Pode ser tanto função normal quanto lambda.

Codigo: op_stack.cpp

```cpp
template <typename T, auto OP>
struct op_stack {
    vector<pair<T, T>> st;
    T get() { return st.back().second; }
    T top() { return st.back().first; }
    T bottom() { return st.front().first; }
    void push(T x) {
        auto snd = st.empty() ? x : OP(st.back().second, x);
        st.push_back({x, snd});
    }
    void pop() { st.pop_back(); }
    bool empty() { return st.empty(); }
    int size() { return (int)st.size(); }
};
```

7.10 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`:

```
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`:

```
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

```
#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>;
```

7.11 Segment Tree

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

Dica: A Seg Tree usa $4 \cdot n$ de memória pois cada nodo p tem seus filhos $2 \cdot p$ (filho esquerdo) e $2 \cdot p + 1$ (filho direito). Há uma forma de indexar os nodos que usa $2 \cdot n$ de memória. Dado um nodo p que representa o intervalo $[l, r]$, seu filho esquerdo é $p + 1$ (e representa o intervalo $[l, mid]$) e seu filho direito é $p + 2 \cdot (mid - l + 1)$ (e representa o intervalo $[mid + 1, r]$), onde $mid = (l + r)/2$.

Codigo: seg_tree.cpp

```
struct SegTree {
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;
    inline int lc(int p) { return p * 2; }
    inline int rc(int p) { return p * 2 + 1; }
    int n;
    vector<ll> t;
    void build(int p, int l, int r, const vector<ll> &v) {
        if (l == r) {
            t[p] = v[l];
        } else {
            int mid = (l + r) / 2;
            build(lc(p), l, mid, v);
            build(rc(p), mid + 1, r, v);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n * 4, neutral);
    }
    void build(const vector<ll> &v) { // pra construir com vector
        n = (int)v.size();
        t.assign(n * 4, 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 p, int l, int r, int L, int R) {
        if (l > R || r < L) return neutral;
        if (l >= L && r <= R) return t[p];
        int mid = (l + r) / 2;
        auto ql = query(lc(p), l, mid, L, R);
        auto qr = query(rc(p), 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 p, int l, int r, int i, ll x, bool repl = 0) {
        if (l == r) {
            if (repl) t[p] = x; // substitui
            else t[p] += x; // soma
        } else {
            int mid = (l + r) / 2;
            if (i <= mid) update(lc(p), l, mid, i, x, repl);
            else update(rc(p), mid + 1, r, i, x, repl);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void update(int i, ll x, bool repl) { update(1, 0, n - 1, i, x, repl); }
    void sumUpdate(int i, ll x) { update(i, x, 0); }
    void setUpdate(int i, ll x) { update(i, x, 1); }
} seg;
```

7.11.2 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

```
struct SegTree2D {
    ll merge(ll a, ll b) { return a + b; }
    ll neutral = 0;
    int n, m;
    vector<vector<ll>> t;
    void build(int _n, int _m) {
        n = _n, m = _m;
```

```
        t.assign(2 * n, vector<ll>(2 * m, neutral));
        for (int i = 2 * n - 1; i >= n; i--)
            for (int j = m - 1; j > 0; j--)
                t[i][j] = merge(t[i][j << 1], t[i][j << 1 | 1]);
        for (int i = n - 1; i > 0; i--)
            for (int j = 2 * m - 1; j > 0; j--)
                t[i][j] = merge(t[i << 1][j], t[i << 1 | 1][j]);
    }
    ll inner_query(int idx, int l, int r) {
        ll res = neutral;
        for (l += m, r += m + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) res = merge(res, t[idx][l++]);
            if (r & 1) res = merge(res, t[idx][--r]);
        }
        return res;
    }
    // query do ponto (a, b) ate o ponto (c, d), retorna neutro se a > c ou b > d
    ll query(int a, int b, int c, int d) {
        ll res = neutral;
        for (a += n, c += n + 1; a < c; a >>= 1, c >>= 1) {
            if (a & 1) res = merge(res, inner_query(a++, b, d));
            if (c & 1) res = merge(res, inner_query(--c, b, d));
        }
        return res;
    }
    void inner_update(int idx, int i, ll x) {
        auto &c = t[idx];
        i += m;
        c[i] = x;
        for (i >>= 1; i > 0; i >>= 1) c[i] = merge(c[i << 1], c[i << 1 | 1]);
    }
    void update(int i, int j, ll x) {
        i += n;
        inner_update(i, j, x);
        for (i >>= 1; i > 0; i >>= 1) {
            ll val = merge(t[i << 1][j + m], t[i << 1 | 1][j + m]);
            inner_update(i, j, val);
        }
    }
} seg;
```

7.11.3 Segment Tree Beats

Segment Tree que suporta update de máximo em range, update de mínimo em range, update de soma em range, e query de soma em range. A construção é $\mathcal{O}(n)$ e as operações de consulta e update são $\mathcal{O}(\log^2 n)$.

Update de máximo em um range $[L, R]$ passando um valor X, significa para cada i tal que $L \leq i \leq R$, fazer a operação $a[i] = max(a[i], X)$. Update de mínimo é análogo.

Obs: Se não usar o update de soma, a complexidade é das operações é $\mathcal{O}(\log n)$

Codigo: seg_tree_beats.cpp

```
const ll INF = 1e18;
struct node {
    ll mi, smi, mx, smx, sum, lazy;
    int fmi, fmx;
    node() {
        mi = smi = INF;
        mx = smx = -INF;
        fmi = 0, fmx = 0, sum = 0, lazy = 0;
    }
    node(ll val) {
        mi = mx = sum = val;
        smi = INF, smx = -INF;
        fmx = fmi = 1;
        lazy = 0;
    }
};

node operator+(node a, node b) {
    node ret;
    ret.sum = a.sum + b.sum;
    if (a.mi == b.mi) {
        ret.mi = a.mi;
        ret.fmi = a.fmi + b.fmi;
        ret.smi = min(a.smi, b.smi);
    } else if (a.mi < b.mi) {
        ret.mi = a.mi;
        ret.fmi = a.fmi;
        ret.smi = min(a.smi, b.mi);
    } else {
        ret.mi = b.mi;
        ret.fmi = b.fmi;
        ret.smi = min(b.smi, a.mi);
```

```
    }
    if (a.mx == b.mx) {
        ret.mx = a.mx;
        ret.fmx = a.fmx + b.fmx;
        ret.smx = max(a.smx, b.smx);
    } else if (a.mx > b.mx) {
        ret.mx = a.mx;
        ret.fmx = a.fmx;
        ret.smx = max(b.mx, a.smx);
    } else {
        ret.fmx = b.fmx;
        ret.mx = b.mx;
        ret.smx = max(a.mx, b.smx);
    }
    return ret;
}

struct SegBeats {
    vector<node> t;
    int n;
    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n * 4, node());
    }
    void build(const vector<ll> &v) { // pra construir com vector
        n = (int)v.size();
        t.assign(n * 4, node());
        build(1, 0, n - 1, v);
    }
    void build(ll *bg, ll *en) { // pra construir com array de C
        build(vector<ll>(bg, en));
    }
    inline int lc(int p) { return 2 * p; }
    inline int rc(int p) { return 2 * p + 1; }
    node build(int p, int l, int r, const vector<ll> &a) {
        if (l == r) return t[p] = node(a[l]);
        int mid = (l + r) >> 1;
        return t[p] = build(lc(p), l, mid, a) + build(rc(p), mid + 1, r, a);
    }
    void pushsum(int p, int l, int r, ll x) {
        t[p].sum += (r - l + 1) * x;
        t[p].mi += x;
        t[p].mx += x;
        t[p].lazy += x;
        if (t[p].smi != INF) t[p].smi += x;
        if (t[p].smx != -INF) t[p].smx += x;
    }
    void pushmax(int p, ll x) {
        if (x <= t[p].mi) return;
        t[p].sum += t[p].fmi * (x - t[p].mi);
        if (t[p].mx == t[p].mi) t[p].mx = x;
        if (t[p].smx == t[p].mi) t[p].smx = x;
        t[p].mi = x;
    }
    void pushmin(int p, ll x) {
        if (x >= t[p].mx) return;
        t[p].sum += t[p].fmx * (x - t[p].mx);
        if (t[p].mi == t[p].mx) t[p].mi = x;
        if (t[p].smi == t[p].mx) t[p].smi = x;
        t[p].mx = x;
    }
    void pushdown(int p, int l, int r) {
        if (l == r) return;
        int mid = (l + r) >> 1;
        pushsum(lc(p), l, mid, t[p].lazy);
        pushsum(rc(p), mid + 1, r, t[p].lazy);
        t[p].lazy = 0;

        pushmax(lc(p), t[p].mi);
        pushmax(rc(p), t[p].mi);

        pushmin(lc(p), t[p].mx);
        pushmin(rc(p), t[p].mx);
    }
    node updatemin(int p, int l, int r, int L, int R, ll x) {
        if (l > R || r < L || x >= t[p].mx) return t[p];
        if (l >= L && r <= R && x > t[p].smx) {
            pushmin(p, x);
            return t[p];
        }
        pushdown(p, l, r);
        int mid = (l + r) >> 1;
        t[p] = updatemin(lc(p), l, mid, L, R, x) + updatemin(rc(p), mid + 1, r, L, R,
            x);
        return t[p];
    }
    node updatemax(int p, int l, int r, int L, int R, ll x) {
        if (l > R || r < L || x <= t[p].mi) return t[p];
        if (l >= L && r <= R && x < t[p].smi) {
            pushmax(p, x);
            return t[p];
        }
        pushdown(p, l, r);
        int mid = (l + r) >> 1;
```

```
126         t[p] = updatemax(lc(p), l, mid, L, R, x) + updatemax(rc(p), mid + 1, r, L, R,
                x);
127         return t[p];
128     }
129     node updatesum(int p, int l, int r, int L, int R, ll x) {
130         if (l > R || r < L) return t[p];
131         if (l >= L && r <= R) {
132             pushsum(p, l, r, x);
133             return t[p];
134         }
135         pushdown(p, l, r);
136         int mid = (l + r) >> 1;
137         return t[p] = updatesum(lc(p), l, mid, L, R, x) +
138                       updatesum(rc(p), mid + 1, r, L, R, x);
139     }
140     node query(int p, int l, int r, int L, int R) {
141         if (l > R || r < L) return node();
142         if (l >= L && r <= R) return t[p];
143         pushdown(p, l, r);
144         int mid = (l + r) >> 1;
145         return query(lc(p), l, mid, L, R) + query(rc(p), mid + 1, r, L, R);
146     }
147     ll query(int l, int r) { return query(1, 0, n - 1, l, r).sum; }
148     void updatemax(int l, int r, ll x) { updatemax(1, 0, n - 1, l, r, x); }
149     void updatemin(int l, int r, ll x) { updatemin(1, 0, n - 1, l, r, x); }
150     void updatesum(int l, int r, ll x) { updatesum(1, 0, n - 1, l, r, x); }
151 } seg;
```

7.11.4 Segment Tree Esparsa

Segment Tree Esparsa, ou seja, não armazena todos os nodos 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.

Dica: No construtor da Seg Tree, fazer `t.reserve(MAX); Lc.reserve(MAX); Rc.reserve(MAX);` pode ajudar bastante no runtime, pois aloca espaço para os vetores e evita muitas realocações durante a execução. Nesse caso, `MAX` é geralmente $\mathcal{O}(Q \cdot \log L)$, onde Q é o número de queries e L é o tamanho do intervalo.

Codigo: seg_tree_sparse.cpp

```
1  const ll MINL = (ll)-1e9 - 5, MAXR = (ll)1e9 + 5;
2  struct SegTree {
3      ll merge(ll a, ll b) { return a + b; }
4      const ll neutral = 0;
5      vector<ll> t;
6      vector<int> Lc, Rc;
7      inline int newnode() {
8          t.push_back(neutral);
9          Lc.push_back(0);
10         Rc.push_back(0);
11         return (int)t.size() - 1;
12     }
13     inline int lc(int p, bool create = false) {
14         if (create && Lc[p] == 0) Lc[p] = newnode();
15         return Lc[p];
16     }
17     inline int rc(int p, bool create = false) {
18         if (create && Rc[p] == 0) Rc[p] = newnode();
19         return Rc[p];
20     }
21     SegTree() {
22         newnode();
23         newnode();
24     }
25     ll query(int p, ll l, ll r, ll L, ll R) {
26         if (p == 0 || l > R || r < L) return neutral;
27         if (l >= L && r <= R) return t[p];
28         ll mid = l + (r - l) / 2;
29         auto ql = query(lc(p), l, mid, L, R);
30         auto qr = query(rc(p), mid + 1, r, L, R);
31         return merge(ql, qr);
32     }
33     ll query(ll l, ll r) { return query(1, MINL, MAXR, l, r); }
34     void update(int p, ll l, ll r, ll i, ll x, bool repl) {
35         if (p == 0) return;
36         if (l == r) {
37             if (repl) t[p] = x; // substitui
38             else t[p] += x; // soma
39             return;
40         }
41         ll mid = l + (r - l) / 2;
42         if (i <= mid) update(lc(p, true), l, mid, i, x, repl);
43         else update(rc(p, true), mid + 1, r, i, x, repl);
44         t[p] = merge(t[lc(p)], t[rc(p)]);
```

```
    }
    void update(ll i, ll x, bool repl) { update(1, MINL, MAXR, i, x, repl); }
    void sumUpdate(ll i, ll x) { update(i, x, 0); }
    void setUpdate(ll i, ll x) { update(i, x, 1); }
} seg;
```

7.11.5 Segment Tree Iterativa

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

Essa implementação é iterativa, o que a torna mais eficiente que a recursiva, além de ser mais fácil de implementar.

Codigo: itseg_tree.cpp

```
struct SegTree {
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;
    inline int lc(int p) { return p * 2; }
    inline int rc(int p) { return p * 2 + 1; }
    int n;
    vector<ll> t;
    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);
        for (int i = 0; i < n; i++) t[i + n] = v[i];
        for (int i = n - 1; i > 0; i--) t[i] = merge(t[lc(i)], t[rc(i)]);
    }
    void build(ll *bg, ll *en) { // pra construir com array de C
        build(vector<ll>(bg, en));
    }
    ll query(int l, int r) {
        ll ansl = neutral, ansr = neutral;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) ansl = merge(ansl, t[l++]);
            if (r & 1) ansr = merge(t[--r], ansr);
        }
        return merge(ansl, ansr);
    }
    void update(int i, ll x, bool replace) {
        i += n;
        t[i] = replace ? x : merge(t[i], x);
        for (i >>= 1; i > 0; i >>= 1) t[i] = merge(t[lc(i)], t[rc(i)]);
    }
    void sumUpdate(int i, ll x) { update(i, x, 0); }
    void setUpdate(int i, ll x) { update(i, x, 1); }
} seg;
```

7.11.6 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.

Dica: A Seg Tree usa $4 \cdot n$ de memória pois cada nodo p tem seus filhos $2 \cdot p$ (filho esquerdo) e $2 \cdot p + 1$ (filho direito). Há uma forma de indexar os nodos que usa $2 \cdot n$ de memória. Dado um nodo p que representa o intervalo $[l, r]$, seu filho esquerdo é $p+1$ (e representa o intervalo $[l, mid]$) e seu filho direito é $p + 2 \cdot (mid - l + 1)$ (e representa o intervalo $[mid + 1, r]$), onde $mid = (l + r)/2$.

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})
        };
    }
    inline int lc(int p) { return p * 2; }
    inline int rc(int p) { return p * 2 + 1; }
    int n;
    vector<node> t;
    void build(int p, int l, int r, const vector<ll> &v) {
        if (l == r) {
            t[p] = {v[l], v[l], v[l], v[l]};
        } else {
            int mid = (l + r) / 2;
            build(lc(p), l, mid, v);
            build(rc(p), mid + 1, r, v);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n * 4, neutral);
    }
    void build(const vector<ll> &v) { // pra construir com vector
        n = int(v.size());
        t.assign(n * 4, 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 p, int l, int r, int L, int R) {
        if (l > R || r < L) return neutral;
        if (l >= L && r <= R) return t[p];
        int mid = (l + r) / 2;
        auto ql = query(lc(p), l, mid, L, R);
        auto qr = query(rc(p), 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 p, int l, int r, int i, ll x) {
        if (l == r) {
            t[p] = {x, x, x, x};
        } else {
            int mid = (l + r) / 2;
            if (i <= mid) update(lc(p), l, mid, i, x);
            else update(rc(p), mid + 1, r, i, x);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void update(int i, ll x) { update(1, 0, n - 1, i, x); }
} seg;
```

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

Dica: A Seg Tree usa $4 \cdot n$ de memória pois cada nodo p tem seus filhos $2 \cdot p$ (filho esquerdo) e $2 \cdot p + 1$ (filho direito). Há uma forma de indexar os nodos que usa $2 \cdot n$ de memória. Dado um nodo p que representa o intervalo $[l, r]$, seu filho esquerdo é $p+1$ (e representa o intervalo $[l, mid]$) e seu filho direito é $p + 2 \cdot (mid - l + 1)$ (e representa o intervalo $[mid + 1, r]$), onde $mid = (l + r)/2$.

Codigo: seg_tree_lazy.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;
    inline int lc(int p) { return p * 2; }
    inline int rc(int p) { return p * 2 + 1; }
    void push(int p, int l, int r) {
        if (replace[p]) {
            t[p] = lazy[p] * (r - l + 1);
            if (l != r) {
                lazy[lc(p)] = lazy[p];
                lazy[rc(p)] = lazy[p];
                replace[lc(p)] = true;
                replace[rc(p)] = true;
            }
        } else if (lazy[p] != 0) {
            t[p] += lazy[p] * (r - l + 1);
            if (l != r) {
```

```
            lazy[lc(p)] += lazy[p];
            lazy[rc(p)] += lazy[p];
        }
    }
    replace[p] = false;
    lazy[p] = 0;
}
void build(int p, int l, int r, const vector<ll> &v) {
    if (l == r) {
        t[p] = v[l];
    } else {
        int mid = (l + r) / 2;
        build(lc(p), l, mid, v);
        build(rc(p), mid + 1, r, v);
        t[p] = merge(t[lc(p)], t[rc(p)]);
    }
}
void build(int _n) { // pra construir com tamanho, mas vazia
    n = _n;
    t.assign(n * 4, neutral);
    lazy.assign(n * 4, 0);
    replace.assign(n * 4, false);
}
void build(const vector<ll> &v) { // pra construir com vector
    n = (int)v.size();
    t.assign(n * 4, neutral);
    lazy.assign(n * 4, 0);
    replace.assign(n * 4, 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 p, int l, int r, int L, int R) {
    push(p, l, r);
    if (l > R || r < L) return neutral;
    if (l >= L && r <= R) return t[p];
    int mid = (l + r) / 2;
    auto ql = query(lc(p), l, mid, L, R);
    auto qr = query(rc(p), 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 p, int l, int r, int L, int R, ll val, bool repl = 0) {
    push(p, l, r);
    if (l > R || r < L) return;
    if (l >= L && r <= R) {
        lazy[p] = val;
        replace[p] = repl;
        push(p, l, r);
    } else {
        int mid = (l + r) / 2;
        update(lc(p), l, mid, L, R, val, repl);
        update(rc(p), mid + 1, r, L, R, val, repl);
        t[p] = merge(t[lc(p)], t[rc(p)]);
    }
}
void update(int l, int r, ll val, bool repl) { update(1, 0, n - 1, l, r, val,
    repl); }
void sumUpdate(int l, int r, ll val) { update(l, r, val, 0); }
void setUpdate(int l, int r, ll val) { update(l, r, val, 1); }
} seg;
```

7.11.8 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.

Dica: No construtor da Seg Tree, fazer `t.reserve(MAX); lazy.reserve(MAX); replace.res Lc.reserve(MAX); Rc.reserve(MAX);` pode ajudar bastante no runtime, pois aloca espaço para os vetores e evita muitas realocações durante a execução. Nesse caso, `MAX` é geralmente $\mathcal{O}(Q \cdot \log L)$, onde Q é o número de queries e L é o tamanho do intervalo.

Codigo: seg_tree_sparse_lazy.cpp

```
const ll MINL = (ll)-1e9 - 5, 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);
        lazy.push_back(0);
        replace.push_back(false);
        return (int)t.size() - 1;
```

```
    }
    inline int lc(int p) {
        if (Lc[p] == -1) Lc[p] = newnode();
        return Lc[p];
    }
    inline int rc(int p) {
        if (Rc[p] == -1) Rc[p] = newnode();
        return Rc[p];
    }
    SegTree() { newnode(); }
    void push(int p, ll l, ll r) {
        if (replace[p]) {
            t[p] = lazy[p] * (r - l + 1);
            if (l != r) {
                lazy[lc(p)] = lazy[p];
                lazy[rc(p)] = lazy[p];
                replace[lc(p)] = true;
                replace[rc(p)] = true;
            }
        } else if (lazy[p] != 0) {
            t[p] += lazy[p] * (r - l + 1);
            if (l != r) {
                lazy[lc(p)] += lazy[p];
                lazy[rc(p)] += lazy[p];
            }
        }
        replace[p] = false;
        lazy[p] = 0;
    }
    ll query(int p, ll l, ll r, ll L, ll R) {
        push(p, l, r);
        if (l > R || r < L) return neutral;
        if (l >= L && r <= R) return t[p];
        ll mid = l + (r - l) / 2;
        auto ql = query(lc(p), l, mid, L, R);
        auto qr = query(rc(p), 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 p, ll l, ll r, ll L, ll R, ll val, bool repl) {
        push(p, l, r);
        if (l > R || r < L) return;
        if (l >= L && r <= R) {
            lazy[p] = val;
            replace[p] = repl;
            push(p, l, r);
        } else {
            ll mid = l + (r - l) / 2;
            update(lc(p), l, mid, L, R, val, repl);
            update(rc(p), mid + 1, r, L, R, val, repl);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void update(ll l, ll r, ll val, bool repl) { update(0, MINL, MAXR, l, r, val,
        repl); }
    void sumUpdate(ll l, ll r, ll val) { update(l, r, val, 0); }
    void setUpdate(ll l, ll r, ll val) { update(l, r, val, 1); }
} seg;
```

7.11.9 Segment Tree PA

Implementação de Segment Tree para soma de Progressão Aritimética, suporta operações de consulta em intervalo e update em range. Está implementada para soma, mas pode ser 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_pa.cpp

```
struct SegTree {
    using ii = pair<ll, ll>;
    ll merge(ll a, ll b) { return a + b; }
    const ll neutral = 0;
    int n;
    vector<ll> t;
    vector<ii> lazy;
    inline int lc(int p) { return p * 2; }
    inline int rc(int p) { return p * 2 + 1; }
    void push(int p, int l, int r) {
        if (lazy[p].second) {
            auto [a, d] = lazy[p];
            t[p] += a * (r - l + 1) + d * (r - l) * (r - l + 1) / 2;
            if (l != r) {
                int mid = (l + r) / 2;
                lazy[lc(p)].first += a;
                lazy[lc(p)].second += d;
                lazy[rc(p)].first += a + (mid + 1 - l) * d;
                lazy[rc(p)].second += d;
            }
            lazy[p] = ii(0, 0);
```

```
        }
    }
    void build(int p, int l, int r, const vector<ll> &v) {
        if (l == r) {
            t[p] = v[l];
        } else {
            int mid = (l + r) / 2;
            build(lc(p), l, mid, v);
            build(rc(p), mid + 1, r, v);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void build(int _n) { // pra construir com tamanho, mas vazia
        n = _n;
        t.assign(n * 4, neutral);
    }
    void build(const vector<ll> &v) { // pra construir com vector
        n = (int)v.size();
        t.assign(n * 4, 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 p, int l, int r, int L, int R) {
        push(p, l, r);
        if (l > R || r < L) return neutral;
        if (l >= L && r <= R) return t[p];
        int mid = (l + r) / 2;
        auto ql = query(lc(p), l, mid, L, R);
        auto qr = query(rc(p), 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 p, int l, int r, int L, int R, ii pa) {
        push(p, l, r);
        if (l > R || r < L) return;
        if (l >= L && r <= R) {
            auto [a, d] = pa;
            lazy[p] = ii(a + (l - L) * d, d);
            push(p, l, r);
        } else {
            int mid = (l + r) / 2;
            update(lc(p), l, mid, L, R, pa);
            update(rc(p), mid + 1, r, L, R, pa);
            t[p] = merge(t[lc(p)], t[rc(p)]);
        }
    }
    void update(int l, int r, ll a0, ll d) { update(1, 0, n - 1, l, r, ii(a0, d)); }
} seg;
```

7.11.10 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 a root (versão da árvore) que se deja modificar. Todos os métodos continuam $\mathcal{O}(\log n)$.

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

Dica: No construtor da Seg Tree, fazer `t.reserve(MAX); Lc.reserve(MAX); Rc.reserve(MA roots.reserve(Q);` pode ajudar bastante no runtime, pois aloca espaço para os vetores e evita muitas realocações durante a execução. Nesse caso, `MAX` é geralmente $\mathcal{O}(Q \cdot \log L)$, onde Q é o número de queries e L é o tamanho do intervalo.

Codigo: seg_tree_persistent.cpp

```
const ll MINL = (ll)-1e9 - 5, 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, roots;
    inline int newnode() {
        t.push_back(neutral);
        Lc.push_back(0);
        Rc.push_back(0);
        return (int)t.size() - 1;
    }
    inline int lc(int p, bool create = false) {
        if (create && Lc[p] == 0) Lc[p] = newnode();
        return Lc[p];
    }
    inline int rc(int p, bool create = false) {
        if (create && Rc[p] == 0) Rc[p] = newnode();
        return Rc[p];
    }
    SegTree() {
        newnode();
```

```
        roots.push_back(newnode());
    }
    ll query(int p, ll l, ll r, ll L, ll R) {
        if (p == 0 || l > R || r < L) return neutral;
        if (l >= L && r <= R) return t[p];
        ll mid = l + (r - l) / 2;
        auto ql = query(lc(p), l, mid, L, R);
        auto qr = query(rc(p), mid + 1, r, L, R);
        return merge(ql, qr);
    }
    ll query(ll l, ll r, int root = -1) {
        if (root == -1) root = roots.back();
        else root = roots[root];
        return query(root, MINL, MAXR, l, r);
    }
    void update(int p, int old, ll l, ll r, ll i, ll x, bool repl) {
        t[p] = t[old];
        if (l == r) {
            if (repl) t[p] = x; // substitui
            else t[p] += x; // soma
            return;
        }
        ll mid = l + (r - l) / 2;
        if (i <= mid) {
            Rc[p] = rc(old);
            update(lc(p, true), lc(old), l, mid, i, x, repl);
        } else {
            Lc[p] = lc(old);
            update(rc(p, true), rc(old), mid + 1, r, i, x, repl);
        }
        t[p] = merge(t[lc(p)], t[rc(p)]);
    }
    int update(ll i, ll x, bool repl, int root = -1) {
        // root é qual versão da segtree vai ser atualizada,
        // -1 atualiza a ultima root
        int new_root = newnode();
        if (root == -1) root = roots.back();
        else root = roots[root];
        update(new_root, root, MINL, MAXR, i, x, repl);
        roots.push_back(new_root);
        return roots.back();
    }
    int sumUpdate(ll i, ll x, int root = -1) { return update(i, x, 0, root); }
    int setUpdate(ll i, ll x, int root = -1) { return update(i, x, 1, root); }
} seg;
```

7.12 Sparse Table

7.12.1 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

```
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;
```

7.12.2 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

```
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]);
    }
};
```

7.13 Treap

Uma árvore de busca binária balanceada. Se não quiser ter elementos repetidos, basta fazer `treap::setify = true`.

- `insert(X)`: insere um elemento X na árvore em $\mathcal{O}(\log N)$
- `remove(X)`: remove uma ocorrência de X na árvore, e retorna `false` caso não tenha nenhuma ocorrência de X na árvore em $\mathcal{O}(\log N)$.
- `find(X)`: retorna `true` se X aparece pelo menos uma vez na árvore em $\mathcal{O}(\log N)$.

Codigo: treap.cpp

```
mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
namespace treap {
    struct node_info {
        node_info *l, *r;
        int x, y, size;
        node_info() { }
        node_info(int _x) : l(0), r(0), x(_x), y(rng()), size(0) { }
    };
    using node = node_info *;
    node root = 0;
    bool setify = false;
    inline int size(node t) { return t ? t->size : 0; }
    inline void upd_size(node t) {
        if (t) t->size = size(t->l) + size(t->r) + 1;
    }
    void merge(node &t, node L, node R) {
        if (!L || !R) {
            t = L ? L : R;
        } else if (L->y > R->y) {
            merge(L->r, L->r, R);
            t = L;
        } else {
            merge(R->l, L, R->l);
            t = R;
        }
        upd_size(t);
    }
    void split(node t, int x, node &L, node &R) {
```

```
        if (!t) {
            L = R = 0;
        } else if (t->x <= x) {
            split(t->r, x, t->r, R);
            L = t;
        } else {
            split(t->l, x, L, t->l);
            R = t;
        }
        upd_size(t);
    }
    void insert(node &t, node to) {
        if (!t) {
            t = to;
        } else if (to->y > t->y) {
            split(t, to->x, to->l, to->r);
            t = to;
        } else {
            insert(to->x < t->x ? t->l : t->r, to);
        }
        upd_size(t);
    }
    bool remove(node &t, int x) {
        if (!t) return false;
        if (x == t->x) {
            node rem = t;
            merge(t, t->l, t->r);
            upd_size(t);
            delete rem;
            return true;
        }
        bool ok = remove(x < t->x ? t->l : t->r, x);
        upd_size(t);
        return ok;
    }
    bool find(node &t, int x) {
        return t ? (t->x == x || find(x < t->x ? t->l : t->r, x)) : false;
    }
    bool find(int x) { return find(root, x); }
    inline void insert(int x) {
        if (setify) {
            if (find(x)) return;
        }
        insert(root, new node_info(x));
    }
    inline void remove(int x) { remove(root, x); }
}
```

7.14 XOR Trie

Uma Trie que armazena os números em binario (do bit mais significativo para o menos). Permite realizar inserção de um número X em $\mathcal{O}(\log X)$. O inteiro `bits` no template da estrutura é a quantidade bits dos números você deseja considerar.

O método `max_xor(X)` retorna o resultado do maior XOR de X com algum número contido na Trie e `min_xor(X)` resultado do menor XOR de X com algum número contido na Trie. Note que o valor X não precisa estar na Trie. Ambos os métodos são $\mathcal{O}(\log X)$.

Codigo: xor_trie.cpp

```
struct XorTrie {
    const int bits = 30;
    vector<vector<int>> go;
    int root = 0, cnt = 1;
    void build(int n) { go.assign((n + 1) * bits, vector<int>(2, -1)); }
    void insert(int x) {
        int v = root;
        for (int i = bits - 1; i >= 0; i--) {
            int b = x >> i & 1;
            if (go[v][b] == -1) go[v][b] = cnt++;
            v = go[v][b];
        }
    }
    int max_xor(int x) {
        int v = root;
        int ans = 0;
        if (cnt <= 1) return -1;
        for (int i = bits - 1; i >= 0; i--) {
            int b = x >> i & 1;
            int good = go[v][!b];
            int bad = go[v][b];
            if (good != -1) {
                v = good;
                ans |= 1 << i;
            } else v = bad;
        }
        return ans;
    }
    int min_xor(int x) {
        int flipped = x ^ ((1 << bits) - 1);
        int query = max_xor(flipped);
        if (query == -1) return -1;
```

```
33          return x ^ flipped ^ query;
34      }
35  } trie;
```

Capítulo 8

Paradigmas

8.1 All Submasks

Percorre todas as submáscaras de uma máscara em $\mathcal{O}(3^n)$.

Codigo: all_submask.cpp

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

8.2 Busca Binaria Paralela

Faz a busca binária para múltiplas consultas quando a busca binária é muito pesada. A complexidade é $\mathcal{O}((N+Q)\log(N)\cdot\mathcal{O}(F))$, 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) {

        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
```

8.3 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

```

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

```

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
    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;
```

```
}
```

8.4 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);
}
```

8.5 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;
}
```

8.6 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_query_on_demand.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];
    }
};
```

Codigo: dc.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++) {
            // min() se quiser minimizar
            best = max(best, {(i ? dp_before[i - 1] : 0) + query(i, mid), i});
        }
        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];
    }
};
```

8.7 Exponenciação de Matriz

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

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

É preciso mapear a DP para uma exponenciação de matriz.

DP:

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

Mapeamento:

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

- –

Exemplo de DP:

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

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

Mapeamento:

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

Exemplo de DP:

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

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
using mat = vector<vector<ll>>;
ll dp[100];
mat T;

#define MOD 1000000007

mat operator*(mat a, mat b) {
    mat res(a.size(), vector<ll>(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 operator^(mat a, ll k) {
    mat res(a.size(), vector<ll>(a.size()));
    for (int i = 0; i < a.size(); i++) res[i][i] = 1;
    while (k) {
        if (k & 1) res = res * a;
        a = a * a;
        k >>= 1;
    }
    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 = 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 = prod * vec;
    return ans[0][0];
}
```

8.8 Mo

8.8.1 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).

A complexidade do `run` é $\mathcal{O}(Q * B + N^2/B)$, onde B é o tamanho do bloco.

Para $B = \sqrt{N}$, a complexidade é $\mathcal{O}((N + Q) * \sqrt{N})$.

Para $B = N/\sqrt{Q}$, a complexidade é $\mathcal{O}(N * \sqrt{Q})$.

Codigo: mo.cpp

```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;
    }

};
```

8.8.2 Mo 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). A complexidade é $\mathcal{O}(Q \cdot \sqrt[3]{N^2})$

Codigo: mo_update.cpp

```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;
        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 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

```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) { // 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

String

10.1 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).

No código, `cur` é o próximo nodo a ser criado. A root é o nodo 1.

Codigo: aho_corasick.cpp

```cpp
namespace aho {
    const int M = 3e5 + 1;
    const int K = 26;

    const char norm = 'a';
    inline int get(int c) { return c - norm; }

    int next[M][K], link[M], out_link[M], par[M], cur = 2;
    char pch[M];
    bool out[M];
    vector<int> output[M];

    int node(int p, char c) {
        link[cur] = out_link[cur] = 0;
        par[cur] = p;
        pch[cur] = c;
        return cur++;
    }

    int T = 0;

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

    int go(int u, char c);

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

```
    }

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

    int exit(int u) {
        if (out_link[u] == 0) {
            int v = get_link(u);
            out_link[u] = (out[v] || v == 1) ? v : exit(v);
        }
        return out_link[u];
    }

    bool matched(int u) { return out[u] || exit(u) > 1; }

}
```

10.2 EertreE

Constrói a Palindromic Tree de uma string S em $\mathcal{O}(|S|)$. Todo nodo da árvore representa exatamente uma substring palindrômica de S.

- `len[u]` representa o tamanho do palíndromo representado pelo nodo `u`.
- `lnk[u]` é o nodo que representa o maior sufixo palindrômico do nodo `u`.
- `cnt[u]` é a frequência da substring representada pelo nodo `u`.
- `first[u]` representa a primeira ocorrência da substring representada pelo nodo `u`, note que `first[u]` guarda o índice do último caractere dessa substring.
- `number_of_palindromes()` retorna a quantidade de substrings palindrômicas de S, lembre-se de chamar a função `build_cnt()` antes dessa.
- `number_of_distinct_palindromes()` retorna a quantidade de substrings palindrômicas distintas.

Codigo: eertree.cpp

```
const int N = 2e5 + 15;
const int ALF = 26;

struct eertree {
    int str[N], len[N], lnk[N], cnt[N], first[N], node_cnt, it, last;
    ll palindrome_substring_sum;
    const char norm = 'a';
    array<int, ALF> to[N];

    inline int get(char c) { return c - norm; }

    void set_string(const string &s) {
        int n = (int)s.size();
        memset(str, 0, sizeof(int) * (it + 1));
        memset(len, 0, sizeof(int) * (it + 1));
        memset(lnk, 0, sizeof(int) * (it + 1));
        memset(cnt, 0, sizeof(int) * (it + 1));
        for (int i = 0; i <= it; i++)
            for (int j = 0; j < ALF; j++) to[i][j] = 0;
        node_cnt = 2, it = 1, last = 0, str[0] = -1;
        len[0] = 0, len[1] = -1, lnk[0] = 1, lnk[1] = 1;
        for (int i = 0; i < n; i++) insert(s[i]);
        build_cnt();
    }

    void insert(char ch) {
        int c = get(ch);
        str[it] = c;
        while (str[it - 1 - len[last]] != c) last = lnk[last];
        if (!to[last][c]) {
            int prev = lnk[last];
            while (str[it - 1 - len[prev]] != c) prev = lnk[prev];
            lnk[node_cnt] = to[prev][c];
            len[node_cnt] = len[last] + 2;
            to[last][c] = node_cnt++;
        }
        last = to[last][c];
        first[last] = it;
        cnt[last]++;
        it++;
    }

    void build_cnt() {
        ll ans = 0;
```

```
        for (int i = it; i > 1; i--) {
            ans += cnt[i];
            cnt[lnk[i]] += cnt[i];
        }
        palindrome_substring_sum = ans;
    }

    inline ll number_of_palindromes() { return palindrome_substring_sum; }
    inline int number_of_distinct_palindromes() { return node_cnt - 2; }
} et;
```

10.3 Hashing

10.3.1 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 h(s);
cout << (h(0, 1) == h(2, 3)) << endl; // 0
cout << (h(0, 1) == h(4, 5)) << endl; // 1
cout << (h(0, 2) == h(4, 6)) << endl; // 1
cout << (h(0, 3) == h(4, 7)) << endl; // 0
cout << (h(0, 3) + h(4, n - 1) * pot[4] == h(0, n - 1)) << endl; // 1, da pra shiftar
    o hash
string t = "abacabab";
Hashing h2(t);
cout << (h() == h2()) << endl; // 1, pode comparar os hashes diretamente
```

Codigo: hashing.cpp

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

struct Hash {
    mint1 h1;
    mint2 h2;
    Hash(mint1 _h1 = 0, mint2 _h2 = 0) : 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 = 33333331; // 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() { }
    Hashing(string s) : N((int)s.size()), hsh(N + 1) {
        for (int i = 0; i < N; i++) {
            int c = (int)s[i];
            hsh[i + 1] = hsh[i] + (pot[i + 1] * Hash(c, c));
        }
    }
    Hash operator()(int l = 0, int r = -1) const {
#warning Chamou o precalc()?
        // se ja chamou o precalc() pode apagar essa linha de cima
        if (r == -1) r = N - 1;
        return (hsh[r + 1] - hsh[l]) * invpot[l];
```

```
    }
};
```

10.3.2 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:

```
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

```
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];
            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;
        hsh.updateSet(i, pot[i + 1] * Hash(c, c));
    }
};
```

10.4 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;
}
```

10.5 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;
```

10.6 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;
```

10.7 Prefix Function KMP

10.7.1 Automato KMP

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

$$nxt[i][c] = \max\{k \mid s[0,k) = s(i-k, i-1]c\}$$

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

```
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;
```

10.7.2 KMP

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

$$p[i] = \max\{k \mid s[0,k) = s(i-k,i]\}$$

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;
}
```

10.8 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).

- `sa[i]` = Índice inicial do i-ésimo menor sufixo.
- `ra[i]` = Rank do sufixo que começa em `i`.
- `LCP[i]` = Comprimento do maior prefixo comum entre os sufixos `sa[i]` e `sa[i-1]`.

* 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.cpp

```cpp
const int MAX = 5e5 + 5;
struct suffix_array {
    string s;
    int n, sum, r, ra[MAX], sa[MAX], auxra[MAX], auxsa[MAX], c[MAX], lcp[MAX];
    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++)
                if (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] == ra[sa[i - 1] + k])
                    auxra[sa[i]] = r;
                else auxra[sa[i]] = ++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;
```

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;
}
```

10.9 Suffix Automaton

Constrói o autômato de sufixos de uma string S em $\mathcal{O}(|S|)$ de forma online.

- `len[u]` é o tamanho da maior string na classe de equivalência de `u`.
- `lnk[u]` é o nodo que representa o maior sufixo de `u` que não pertence a classe de equivalência de `u`.
- `to[u]` é um array que representa as possivéis transições de um nodo `u`.
- `cnt[u]` é um array que conta para cada classe de equivalência quantas ocorrências essas substrings tem.
- `where[i]` diz em qual nodo do autômato a substring $s[0..i]$ está.

Por definição, `len[lca(u, v)]` diz o tamanho da maior substring que é sufixo de `u` e `v` ao mesmo tempo, ou seja, é o longest common suffix.

Algumas aplicações estão no código, é importante notar que essas aplicações funcionam de maneira **offline**, ou seja, uma vez settada a string no autômato, não se deve fazer inserts adicionais de caracteres.

Para outras possíveis aplicações, consulte a Suffix Tree.

Codigo: suffix_automaton.cpp

```
const int N = 5e5 + 5;
const int S = 2 * N;

struct SuffixAutomaton {
    array<int, 26> to[S];
    int lnk[S], len[S], cnt[S], idx[S], where[S];
    int last = 1, id = 2;
    ll distinct_substrings = 0;

    const char norm = 'a';
    inline int get(char c) { return c - norm; }

    void insert(int i, char ch) {
        int cur = id++;
        int c = get(ch);
        len[cur] = len[last] + 1;
        where[idx[cur] = i] = cur;
```

```
        cnt[cur] = 1;
        int p = last;
        while (p > 0 && !to[p][c]) {
            to[p][c] = cur;
            p = lnk[p];
        }
        if (p == 0) {
            lnk[cur] = 1;
        } else {
            int sp = to[p][c];
            if (len[p] + 1 == len[sp]) {
                lnk[cur] = sp;
            } else {
                int clone = id++;
                len[clone] = len[p] + 1;
                lnk[clone] = lnk[sp];
                idx[clone] = idx[sp];
                to[clone] = to[sp];
                while (p > 0 && to[p][c] == sp) {
                    to[p][c] = clone;
                    p = lnk[p];
                }
                lnk[sp] = lnk[cur] = clone;
            }
        }
        last = cur;
    }

    vector<int> adj[S];

    void dfs(int u) {
        for (int v : adj[u]) {
            dfs(v);
            cnt[u] += cnt[v];
        }
        distinct_substrings += len[u] - len[lnk[u]];
    }

    void set_string(const string &s) {
        int n = (int)s.size();
        for (int i = 0; i < id; i++) {
            to[i].fill(0);
            len[i] = lnk[i] = cnt[i] = 0;
            adj[i].clear();
        }
        last = 1, id = 2, distinct_substrings = 0;
        for (int i = 0; i < n; i++) insert(i, s[i]);
        for (int i = 2; i < id; i++) adj[lnk[i]].push_back(i);
        dfs(1);
    }

    int count_occurrences(const string &t) {
        int cur = 1;
        for (char ch : t) {
            int c = get(ch);
            if (!to[cur][c]) return 0;
            cur = to[cur][c];
        }
        return cnt[cur];
    }

    int longest_common_substring(const string &t) {
        int cur = 1, clen = 0, ans = 0;
        for (char ch : t) {
            int c = get(ch);
            while (cur > 0 && !to[cur][c]) {
                cur = lnk[cur];
                clen = len[cur];
            }
            if (to[cur][c]) {
                cur = to[cur][c];
                clen++;
            }
            ans = max(ans, clen);
            cur = max(cur, 1);
        }
        return ans;
    }

    int lcs(int i, int j) {
        // retorna o maior sufixo comum entre os prefixos s[0..i] e s[0..j]
        // tem que codar o lca aqui no automato
        return len[lca(where[i], where[j])];
    }
} sam;
```

10.10 Suffix Tree

Constrói a árvore de sufixos de uma string S em $\mathcal{O}(|S|)$. A árvore não é construída da forma clássica com o algoritmo de Ukkonen, mas sim utilizando do Suffix Automaton.

Teorema: a árvore de links do Suffix Automaton sobre uma string S, é a Suffix Tree de $\overline{S}$, onde $\overline{S}$ é a string S reversa. Aqui não cabe provar esse teorema, basta crer.

Praticamente tudo do suffix automaton ainda vale aqui. Uma diferença é que `where[i]` agora diz em qual nodo da árvore o sufixo $s[i..|s|-1]$ está. E `lnk[i]` agora aponta para o maior **prefixo** de `i` que não está na mesma classe de equivalência que i.

Além disso, temos um vetor adicional `suff[i]` que diz se o nodo `i` é um sufixo da string original.

Por definição, `len[lca(u, v)]` diz o tamanho da maior substring que é prefixo de `u` e `v` ao mesmo tempo, ou seja, é o longest common prefix (LCP).

Ou seja, o que temos nesse código é o Suffix Automaton, mas que ao passar uma string pra ele, ele constrói o autômato da string reversa. As aplicações no código vão se basear no fato de que a árvore que temos é a Suffix Tree. Se usar com carinho, podemos usar tanto o autômato quanto a Suffix Tree ao mesmo tempo (só tem que lembrar que a string original está invertida no autômato, caso queira fazer processamento de alguma string ou algo assim).

Nesse código, a lista de adjacência da árvore está ordenada lexicograficamente, ou seja, se passarmos pela árvore em ordem de DFS, os nodos que estão marcados com `suff[u] = 1`, são os sufixos da string original ordenados lexicograficamente, ou seja, é o Suffix Array.

Obs: apesar do algoritmo de inserção de caractere funcionar de maneira online, todas aplicações no código requerem que a string seja passada de uma vez, e que não sejam feitos inserts adicionais.

Codigo: suffix_tree.cpp

```cpp
const int N = 5e5 + 5;
const int S = 2 * N;

struct SuffixTree {
    array<int, 26> to[S];
    int lnk[S], len[S], cnt[S], idx[S], where[S], suff[S];
    int last = 1, id = 2;
    ll distinct_substrings = 0;
    string s;

    const char norm = 'a';
    inline int get(char c) { return c - norm; }

    void insert(int i, char ch) {
        int cur = id++;
        int c = get(ch);
        len[cur] = len[last] + 1;
        where[idx[cur] = i] = cur;
        cnt[cur] = suff[cur] = 1;
        int p = last;
        while (p > 0 && !to[p][c]) {
            to[p][c] = cur;
            p = lnk[p];
        }
        if (p == 0) {
            lnk[cur] = 1;
        } else {
            int sp = to[p][c];
            if (len[p] + 1 == len[sp]) {
                lnk[cur] = sp;
            } else {
                int clone = id++;
                len[clone] = len[p] + 1;
                lnk[clone] = lnk[sp];
                idx[clone] = idx[sp];
                to[clone] = to[sp];
                while (p > 0 && to[p][c] == sp) {
                    to[p][c] = clone;
                    p = lnk[p];
                }
                lnk[sp] = lnk[cur] = clone;
            }
        }
        last = cur;
    }

    vector<int> adj[S];

    void dfs(int u) {
        sort(adj[u].begin(), adj[u].end(), [&](int v1, int v2) {
            // sorta os filhos de u por ordem lexicografica,
            // isso faz com que a ordem de dfs seja a ordem
            // lexicografica dos sufixos (e de qualquer substring na verdade)
            return s[idx[v1] + len[u]] < s[idx[v2] + len[u]];
        });
        for (int v : adj[u]) {
            dfs(v);
            cnt[u] += cnt[v];
        }
        distinct_substrings += len[u] - len[lnk[u]];
```

```
    }

    void set_string(const string &s2) {
        s = s2;
        int n = (int)s.size();
        for (int i = 0; i < id; i++) {
            to[i].fill(0);
            len[i] = lnk[i] = cnt[i] = 0;
        }
        id = 2;
        for (int i = n - 1; i >= 0; i--) insert(i, s[i]);
        for (int i = 2; i < id; i++) adj[lnk[i]].push_back(i);
        dfs(1);
    }

    int count_occurrences(const string &t) {
        int cur = 1, m = (int)t.size();
        for (int i = m - 1; i >= 0; i--) {
            int c = get(t[i]);
            if (!to[cur][c]) return 0;
            cur = to[cur][c];
        }
        return cnt[cur];
    }

    int longest_common_substring(const string &t) {
        int cur = 1, clen = 0, ans = 0, m = (int)t.size();
        for (int i = m - 1; i >= 0; i--) {
            int c = get(t[i]);
            while (cur > 0 && !to[cur][c]) {
                cur = lnk[cur];
                clen = len[cur];
            }
            if (to[cur][c]) {
                cur = to[cur][c];
                clen++;
            }
            ans = max(ans, clen);
            cur = max(cur, 1);
        }
        return ans;
    }

    string kth_substring(ll k) {
        // esse metodo retorna a k-esima menor substring lexicograficamente,
        // dentre todas as substrings distintas
        string res = "";
        function<bool(int)> dfs_kth = [&](int u) {
            int min_len = len[lnk[u]] + 1, max_len = len[u];
            int qnt = (max_len - min_len + 1);
            if (qnt < k) {
                k -= qnt;
            } else {
                res = s.substr(idx[u], min_len + k - 1);
                return true;
            }
            for (int v : adj[u])
                if (dfs_kth(v)) return true;
            return false;
        };
        dfs_kth(1);
        return res;
    }

    string kth_substring2(ll k) {
        // esse metodo retorna a k-esima menor substring lexicograficamente,
        // dentre todas as substrings nao necessariamente distintas
        string res = "";
        function<bool(int)> dfs_kth = [&](int u) {
            int min_len = len[lnk[u]] + 1, max_len = len[u];
            ll qnt = 1LL * (max_len - min_len + 1) * cnt[u];
            if (qnt < k) {
                k -= qnt;
            } else {
                res = s.substr(idx[u], min_len + (k - 1) / cnt[u]);
                return true;
            }
            for (int v : adj[u])
                if (dfs_kth(v)) return true;
            return false;
        };
        dfs_kth(1);
        return res;
    }

    int lcp(int i, int j) {
        // retorna o maior prefixo comum entre os sufixos s[i..n) e s[j..n)
        // tem que codar o lca aqui no automato
        return len[lca(where[i], where[j])];
    }
} st;
```

10.11 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

```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];
    }
};
```

10.12 Z function

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

$$z[i] = \max\{k \mid s[0,k) = s[i,i+k)\}$$

Em outras palavras, $z[i]$ é o tamanho do maior prefixo de s é prefixo de $s[i,|s|-1]$.

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

Codigo: z.cpp

```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;
}
```