

Almanaque de Códigos pra
Maratona de Programação

BRUTE UDESC

8 de outubro de 2025

Índice

1 C++	10
1.1 Compilador	10
1.2 STL (Standard Template Library)	10
1.2.1 Vector	10
1.2.2 Pair	11
1.2.3 Set	11
1.2.4 Multiset	11
1.2.5 Map	11
1.2.6 Queue	12
1.2.7 Priority Queue	12
1.2.8 Stack	12
1.2.9 Bitset	12
1.2.10 Funções úteis	13
1.2.11 Funções úteis para vetores	13
1.3 Pragmas	13
1.4 Constantes em C++	14

2 Teórico	15
2.1 Definições	15
2.1.1 Funções	15
2.1.2 Grafos	15
2.2 Números primos	15
2.2.1 Primos com truncamento à esquerda	15
2.2.2 Primos gêmeos (Twin Primes)	16
2.2.3 Números primos de Mersenne	16
2.3 Operadores lineares	16
2.3.1 Rotação no sentido anti-horário por θ°	16
2.3.2 Reflexão em relação à reta $y = mx$	16
2.3.3 Inversa de uma matriz 2×2 A	16
2.3.4 Cisalhamento horizontal por K	16
2.3.5 Cisalhamento vertical por K	16
2.3.6 Mudança de base	16
2.3.7 Propriedades das operações de matriz	17
2.4 Sequências numéricas	17
2.4.1 Sequência de Fibonacci	17
2.4.2 Sequência de Catalan	17
2.5 Análise combinatória	18
2.5.1 Fatorial	18
2.5.2 Combinação	18
2.5.3 Arranjo	18
2.5.4 Estrelas e barras	18
2.5.5 Princípio da inclusão-exclusão	18

2.5.6	Princípio da casa dos pombos	18
2.6	Teoria dos números	18
2.6.1	Pequeno teorema de Fermat	18
2.6.2	Teorema de Euler	18
2.6.3	Aritmética modular	18
2.7	Markov Chains	19
2.7.1	Distribuições Estacionárias	19
2.7.2	Markov Chains Absorventes	19
3	Extra	20
3.1	CPP	20
3.2	Debug	20
3.3	Random	21
3.4	Run	21
3.5	Stress Test	21
3.6	Unordered Custom Hash	22
3.7	Vim	22
4	Matemática	23
4.1	Continued Fractions	23
4.2	Convolução	24
4.2.1	AND Convolution	24
4.2.2	Dirichlet Convolution	24
4.2.3	GCD Convolution	26
4.2.4	LCM Convolution	27
4.2.5	OR Convolution	27

4.2.6	Subset Convolution	27
4.2.7	XOR Convolution	28
4.3	Discrete Root	28
4.4	Eliminação Gaussiana	29
4.4.1	Gauss	29
4.4.2	Gauss Mod 2	30
4.5	Exponenciação Modular Rápida	31
4.6	FFT	31
4.7	Fatoração e Primos	32
4.7.1	Crivo	32
4.7.2	Divisores	33
4.7.3	Fatores	34
4.7.4	Pollard Rho	35
4.7.5	Teste Primalidade	35
4.8	Floor Values	36
4.9	Floor and Mod Sum of Arithmetic Progressions	36
4.10	GCD	36
4.11	Inverso Modular	37
4.12	NTT	38
4.12.1	NTT	38
4.12.2	NTT Big Modulo	39
4.12.3	Taylor Shift	40
4.13	Polinomios	40
4.14	Teorema do Resto Chinês	44
4.15	Totiente de Euler	45

4.16 XOR Gauss	45
5 Paradigmas	47
5.1 All Submasks	47
5.2 Busca Binaria Paralela	47
5.3 Busca Ternaria	48
5.4 Convex Hull Trick	49
5.5 DP de Permutacao	49
5.6 Divide and Conquer	50
5.7 Exponenciação de Matriz	51
5.8 Mo	53
5.8.1 Mo	53
5.8.2 Mo Update	53
6 Primitivas	55
6.1 Modular Int	55
6.2 Ponto 2D	56
7 String	57
7.1 Aho Corasick	57
7.2 EertreE	58
7.3 Hashing	59
7.3.1 Hashing	59
7.3.2 Hashing Dinâmico	60
7.4 Lyndon	60
7.5 Manacher	61

7.6	Patricia Tree	62
7.7	Prefix Function KMP	62
7.7.1	Automato KMP	62
7.7.2	KMP	63
7.8	Suffix Array	63
7.9	Suffix Automaton	64
7.10	Suffix Tree	65
7.11	Trie	68
7.12	Z function	68
8	Geometria	69
8.1	Convex Hull	69
9	Grafos	70
9.1	2 SAT	70
9.2	Binary Lifting	71
9.2.1	Binary Lifting LCA	71
9.2.2	Binary Lifting Query	72
9.2.3	Binary Lifting Query 2	73
9.2.4	Binary Lifting Query Aresta	74
9.3	Block Cut Tree	75
9.4	Caminho Euleriano	76
9.4.1	Caminho Euleriano Direcionado	76
9.4.2	Caminho Euleriano Nao Direcionado	76
9.5	Centro e Diametro	77
9.6	Centroids	78

9.6.1	Centroid	78
9.6.2	Centroid Decomposition	78
9.7	Ciclos	79
9.7.1	Find Cycle	79
9.7.2	Find Negative Cycle	80
9.8	Fluxo	80
9.9	HLD	83
9.9.1	HLD Aresta	83
9.9.2	HLD Vértice	84
9.10	Inverse Graph	85
9.11	Kosaraju	86
9.12	Kruskal	87
9.13	LCA	87
9.14	Matching	88
9.14.1	Hungaro	88
9.15	Pontes	88
9.15.1	Componentes Aresta Biconexas	88
9.15.2	Pontes	89
9.16	Pontos de Articulacao	90
9.17	Shortest Paths	90
9.17.1	01 BFS	90
9.17.2	BFS	91
9.17.3	Bellman Ford	91
9.17.4	Dial	91
9.17.5	Dijkstra	92

9.17.6 Floyd Warshall	92
9.17.7 SPFA	93
9.18 Stoer–Wagner Min Cut	93
9.19 Virtual Tree	94
10 Estruturas de Dados	95
10.1 Disjoint Set Union	95
10.1.1 DSU	95
10.1.2 DSU Bipartido	95
10.1.3 DSU Rollback	96
10.1.4 DSU Rollback Bipartido	96
10.1.5 Offline DSU	97
10.2 Fenwick Tree	98
10.2.1 Fenwick	98
10.2.2 Kd Fenwick Tree	99
10.3 Implicit Treap	99
10.4 Interval Tree	101
10.5 LiChao Tree	101
10.6 Merge Sort Tree	102
10.6.1 Merge Sort Tree	102
10.6.2 Merge Sort Tree Update	103
10.7 Operation Deque	104
10.8 Operation Queue	104
10.9 Operation Stack	105
10.10 Ordered Set	105
10.11 Segment Tree	107

10.11.1 Segment Tree	107
10.11.2 Segment Tree 2D	107
10.11.3 Segment Tree Beats	108
10.11.4 Segment Tree Esparsa	110
10.11.5 Segment Tree Iterativa	111
10.11.6 Segment Tree Kadane	111
10.11.7 Segment Tree Lazy	112
10.11.8 Segment Tree Lazy Esparsa	113
10.11.9 Segment Tree PA	114
10.11.10 Segment Tree Persistente	115
10.12 Sparse Table	116
10.12.1 Disjoint Sparse Table	116
10.12.2 Sparse Table	117
10.13 Treap	117
10.14 XOR Trie	118

Capítulo 1

C++

1.1 Compilador

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

```
1 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 eficientes. 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<int>> 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 dobitset como um inteiro - $\mathcal{O}(N/w)$

Obitset 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 inclusivo e R é exclusivo.

- `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())`: Rearrange 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 usá-los 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 (entre 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}_β 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 factorial de um número n é o produto de todos os inteiros positivos menores ou iguais a n .

O factorial 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.

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

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:

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

2.7 Markov Chains

Markov Chains (Cadeias de Markov) são grafos onde nodos são estados e arestas representam a probabilidade de transicionar de um estado para outro em um passeio aleatório.

A partir das arestas do grafo, podemos construir uma matriz de transição P , onde $P_{i,j}$ representa a probabilidade de ir do nodo i ao nodo j em um passo. A x -ésima potência de P representa a probabilidade de ir do nodo i ao nodo j em x passos.

Um estado é dito transitente se existe probabilidade de nunca voltar para ele uma vez que se saiu dele. Se não existe, ele é recorrente. Se a probabilidade de sair dele é zero, ele é absorvente.

Um estado tem período $k \geq 1$ se só é possível retornar a ele com passeios de tamanho múltiplo de k . Se $k = 1$, ele é dito aperiódico, se não, ele é periódico. Se todos os estados são aperiódicos, a Markov Chain é aperiódica.

Uma Markov Chain é dita irreduzível se, entre cada par de estados, existe um passeio com probabilidade positiva de ocorrer.

Uma Markov Chain é dita ergódica se todos os seus estados são recorrentes e ela é aperiódica.

2.7.1 Distribuições Estacionárias

Uma distribuição estacionária é um vetor de probabilidades π tal que $\pi P = \pi$, ou seja, $(P^T - I)\pi = 0$. Isso significa que é uma distribuição de probabilidades que diz, para cada estado, a probabilidade de estar nele, e essa distribuição permanece igual ao aplicarmos a matriz de transição. O somatório dos valores em π resulta em 1.

Uma Markov Chain irreduzível tem uma distribuição estacionária se, e somente se, for aperiódica e ergódica. Se for ergódica, a distribuição estacionária é única.

Se uma Markov Chain for ergódica, o número esperado de passos para voltar ao estado i depois de começar nele é dado por $E(i) = 1/\pi_i$.

2.7.2 Markov Chains Absorventes

Uma Markov Chain é dita absorvente se todo estado transitente pode alcançar algum estado absorvente com probabilidade positiva.

Considere uma Markov chain com T estados transitentes e S estados absorventes. A matriz de transição P pode ser escrita dessa forma:

$$P = \begin{bmatrix} Q & R \\ 0 & I \end{bmatrix}$$

Onde Q é a submatriz $T \times T$ que corresponde à probabilidade de estados transitentes transacionarem entre si, R é a submatriz $T \times S$ que corresponde à probabilidade de estados transitentes alcançarem estados absorventes, 0 é a matriz zero $S \times T$ e I é a matriz identidade $S \times S$.

A matriz fundamental N , onde a célula $N_{i,j}$ representa a quantidade esperada de vezes que o estado transitente j é alcançado quando começando em i , é dada por:

$$N = (I - Q)^{-1}$$

A matriz M , onde a célula $M_{i,j}$ representa a probabilidade de ser absorvido pelo estado j quando começando em i , é dada por:

$$M = NR$$

Capítulo 3

Extra

3.1 CPP

Template de C++ para usar na Maratona.

Código: template.cpp

```
1 #include <bits/stdc++.h>
2 #define endl '\n'
3 using namespace std;
4 using ll = long long;
5
6 void solve() { }
7
8 signed main() {
9     cin.tie(0)->sync_with_stdio(0);
10    solve();
11 }
```

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`.

Código: debug.cpp

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

```
30 #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`.

Código: rand.cpp

```
1 mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
2
3 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`.

Código: run

```
1#!/bin/bash
2g++ -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.

Código: stress.sh

```
1#!/bin/bash
2set -e
3
4g++ -O2 gen.cpp -o gen # pode fazer o gerador em python se preferir
5g++ -O2 brute.cpp -o brute
6g++ -O2 code.cpp -o code
7
8for((i = 1; ; ++i)); do
9    ./gen $i > in
10   ./code < in > out
11   ./brute < in > ok
12   diff -w out ok || break
13   echo "Passed test: " $i
14done
15
16echo "WA no seguinte teste:"
17cat in
18echo "Sua resposta eh:"
19cat out
20echo "A resposta correta eh:"
21cat 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;`

Código: `custom_hash.cpp`

```

1 struct custom_hash {
2     static uint64_t splitmix64(uint64_t x) {
3         x += 0x9e3779b97f4a7c15;
4         x = (x ^ (x >> 30)) * 0xbff58476d1ce4e5b9;
5         x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
6         return x ^ (x >> 31);
7     }
8
9     size_t operator()(uint64_t x) const {
10        static const uint64_t FIXED_RANDOM =
11            chrono::steady_clock::now().time_since_epoch().count();
12        return splitmix64(x + FIXED_RANDOM);
13    }
14 };

```

```

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

```

3.7 Vim

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

Código: `vimrc`

```

1 set nu ai si cindent et ts=4 sw=4 so=10 nosm undofile
2
3 inoremap {} {}<left><return><up><end><return>
4 " remap de chaves
5

```

Capítulo 4

Matemática

4.1 Continued Fractions

Inspirado pelo artigo no CP Algorithms: <https://cp-algorithms.com/algebra/continued-fractions.html>

Computa fração contínua a partir de fração racional e vice-versa.

`lca(u, v)` computa o lca das frações u e v na Stern-Brocott Tree

Tamanho da fração contínua e complexidade para computá-la a partir da fração racional A/B : $\mathcal{O}(\log(A + B))$.

Portanto, lca e outros métodos têm mesma complexidade.

Testado apenas em: https://atcoder.jp/contests/abc408/tasks/abc408_g

IMPORTANTE: Cuidado com "inf" quando usando "plus_minus_eps"- chamar "convergents" para uma fração contínua

com "inf" no final resulta em overflow.

Código: continued_fractions.cpp

```
1 template <typename T>
2 struct fraction {
3     T p, q;
4 };
5 using F = fraction<ll>;
6
```

```
7 const ll inf = LLONG_MAX;
8
9 vector<ll> continued_fraction(F f) {
10    vector<ll> a;
11    while (f.q) {
12        a.emplace_back(f.p / f.q);
13        tie(f.p, f.q) = make_pair(f.q, f.p % f.q);
14    }
15    return a;
16}
17
18 #warning "'convergents(a)' assumes 'a.back() != inf'"
19 // CAREFUL: assumes A.BACK()
20 pair<vector<ll>, vector<ll>> convergents(vector<ll> a) {
21    vector<ll> p = {0, 1};
22    vector<ll> q = {1, 0};
23
24    int n = a.size();
25    for (int i = 0; i < n; i++) {
26        p.emplace_back(p.end()[-1] * a[i] + p.end()[-2]);
27        q.emplace_back(q.end()[-1] * a[i] + q.end()[-2]);
28    }
29
30    return make_pair(p, q);
31}
32
33 vector<ll> lca_continued(vector<ll> a, vector<ll> b) {
34    int n_f = a.size();
35    int n_g = b.size();
36
37    vector<ll> ans;
```

```

38     for (int i = 0; i < min(n_f, n_g); i++) {
39         ans.emplace_back(min(a[i], b[i]));
40         if (a[i] != b[i]) break;
41     }
42     ans.back()++;
43
44     return ans;
45 }
46
47 F lca(F f, F g) {
48     vector<ll> cont_f = continued_fraction(f);
49     vector<ll> cont_g = continued_fraction(g);
50
51     vector<ll> cont = lca_continued(cont_f, cont_g);
52
53     auto [P, Q] = convergents(cont);
54     return F(P.back(), Q.back());
55 }
56
57 // assumes a != b and a.back() == b.back() == inf
58 bool less_cont(vector<ll> a, vector<ll> b) {
59     for (int i = 1; i < a.size(); i += 2) a[i] *= -1;
60     for (int i = 1; i < b.size(); i += 2) b[i] *= -1;
61     return a < b;
62 }
63
64 vector<ll> expand_continued_fraction(vector<ll> a) {
65     // empty a = inf
66     if (a.size()) {
67         a.back()--;
68         a.emplace_back(1);
69         return a;
70     } else return a;
71 }
72
73 // returns {a-eps, a+eps}
74 pair<vector<ll>, vector<ll>> plus_minus_eps(vector<ll> a) {
75     vector<ll> b = expand_continued_fraction(a);
76     a.emplace_back(inf);
77     b.emplace_back(inf);
78     return less_cont(a, b) ? make_pair(a, b) : make_pair(b, a);
79 }
```

4.2 Convolução

4.2.1 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.

Código: and_convolution.cpp

```

1  vector<mint> and_convolution(vector<mint> A, vector<mint> B) {
2      int n = (int)max(A.size(), B.size());
3      int N = 0;
4      while ((1 << N) < n) N++;
5      A.resize(1 << N);
6      B.resize(1 << N);
7      vector<mint> C(1 << N);
8      for (int j = 0; j < N; j++) {
9          for (int i = (1 << N) - 1; i >= 0; i--) {
10              if (~i >> j & 1) {
11                  A[i] += A[i | (1 << j)];
12                  B[i] += B[i | (1 << j)];
13              }
14          }
15      }
16      for (int i = 0; i < 1 << N; i++) C[i] = A[i] * B[i];
17      for (int j = 0; j < N; j++) {
18          for (int i = 0; i < 1 << N; i++)
19              if (~i >> j & 1) C[i] -= C[i | (1 << j)];
20      }
21      return C;
22 }
```

4.2.2 Dirichlet Convolution

Dirichlet Convolution

Calcula o vetor C a partir de A e B onde $C[n] = \sum_{d|n} A[d] \cdot B\left[\frac{n}{d}\right]$, ou seja, a convolução de Dirichlet, em $\mathcal{O}(N)$.

Código: dirichlet_convolution.cpp

```

1 template <typename T>
2 vector<T> dirichlet(const vector<int> &f, const vector<int> &g) {
3     int n = int(max(f.size(), g.size()));
4     vector<int> primes, spf(n, 1), cnt(n);
5     vector<T> ans(n);
6     ans[1] = (T)f[1] * g[1];
7     for (int i = 2; i < n; i++) {
8         if (spf[i] == 1) {
9             spf[i] = i;
10            primes.emplace_back(i);
11            cnt[i] = i;
12        }
13
14        for (auto p : primes) {
15            if (i * p >= n) break;
16            spf[i * p] = p;
17            if (spf[i] == p) {
18                cnt[i * p] = cnt[i] * p;
19                break;
20            } else cnt[i * p] = p;
21        }
22
23        if (i == cnt[i])
24            for (int j = 1; j <= cnt[i]; j *= spf[i]) ans[i] += (T)f[j] * g[cnt[i] / j];
25        else ans[i] = (T)ans[i / cnt[i]] * ans[cnt[i]];
26    }
27    return ans;
28 }
```

Dirichlet Convolution Prefix

Dadas duas funções aritméticas f e g com seus valores computados para $1 \leq i \leq N^{\frac{2}{3}}$; e seus prefixos de soma F, G com valores computados para todo $\lfloor \frac{N}{i} \rfloor$ com $1 \leq i \leq N^{\frac{1}{3}}$, obtém o vetor H tal que $H_i = \sum_{j=1}^{\lfloor \frac{n}{i} \rfloor} h(j)$

para todo $1 \leq i \leq N^{\frac{1}{3}}$ em $O(N^{\frac{2}{3}})$. Sendo $h(n) = \sum_{d|n} f(d) * g(\lfloor \frac{n}{d} \rfloor)$, ou seja, a convolução de Dirichlet de f com g . Para atingir essa complexidade, deve-se inicializar a estrutura com $T = N^{\frac{2}{3}}$.

Para obter os demais valores de H (para $1 \leq i \leq N^{\frac{2}{3}}$) basta utilizar a [convolução de Dirichlet linear](..../Dirichlet-Convolution/dirichlet_convolution.cpp).

O código usa a primitiva Mint para realizar operações modulares de forma eficiente.

* $f(x)$: função que retorna o valor de $f(x)$ em $O(1)$.
* $g(x)$: função que retorna o valor de $g(x)$ em $O(1)$.
* $F(x)$: função que retorna o valor de $\sum_{i=1}^{\lfloor \frac{n}{x} \rfloor} f(i)$ em $O(1)$.
* $G(x)$: função que retorna o valor de $\sum_{i=1}^{\lfloor \frac{n}{x} \rfloor} g(i)$ em $O(1)$.

Código: dirichlet_convolution_prefix.cpp

```

1 struct DirichletConvolutionPrefix {
2     ll N;
3     int T;
4     vector<mint> ans;
5
6     DirichletConvolutionPrefix(ll n, int t) : N(n), T(t) { ans.assign(n / T + 1, 0); }
7
8     vector<mint> solve(auto &&f, auto &&F, auto &&g, auto &&G) {
9         if (N == 1) return vector<mint>(2, 1);
10        ans.assign(N / T + 1, 0);
11        for (ll i = 1; i <= N / T; i++) {
12            ll now = N / i;
13            mint f_sum = 0, g_sum = 0;
14            for (int j = 1; (ll)j * j <= now; j++) {
15                f_sum += f(j);
16                g_sum += g(j);
17
18                ans[i] += G(i * j) * f(j);
19                ans[i] += F(i * j) * g(j);
20            }
21            ans[i] -= f_sum * g_sum;
22        }
23        return ans;
24    };
25 }
```

Dirichlet Inverse Prefix

Dadas 3 funções aritméticas f , F e f^{-1} tal que $F = \sum_{j=1}^i f(j)$ e f^{-1} é a inversa de Dirichlet de f com seus valores computados para $1 \leq i \leq N^{2/3}$,

$$\text{obtem-se } F^{-1}[\lfloor n/i \rfloor] = \sum_{j=1}^{\lfloor n/i \rfloor} f^{-1}(j)$$

Para todo $1 \leq i \leq N^{1/3}$ em $\mathcal{O}(N^{2/3})$. Para obter os demais $N^{2/3}$ valores, basta calcular

$$F^{-1}(i) = \sum_{j=1}^i f^{-1}(j) \text{ usando soma de prefixo.}$$

Para atingir essa complexidade, deve-se inicializar a estrutura com $T = N^{2/3}$ e realizar as pré-computações necessárias de modo que todas as funções auxiliares possam ser obtidas em $\mathcal{O}(1)$.

O código utiliza a primitiva Mint para realizar operações modulares de forma eficiente.

- $f(x)$: retorna o valor de $f[x]$.
- $F(x)$: retorna o valor de $\sum_{j=1}^x f[j]$.
- $g(x)$: retorna o valor de $f^{-1}[x]$.

Código: dirichlet_inverse_prefix.cpp

```

1 struct DirichletInversePrefix {
2     ll N;
3     int T;
4     vector<mint> init_pref, ans;
5     vector<bool> vis;
6     DirichletInversePrefix(ll n, int t, vector<mint> _init_pref)
7         : N(n), T(t), init_pref(_init_pref) {
8             vis.assign(n / T + 1, 0);
9             ans.assign(n / T + 1, 0);
10        }
11
12    mint solve(ll n, ll floor_N, auto &&f, auto &&F, auto &&g) {
13        if (n < int(init_pref.size())) {
14            if (floor_N <= N / T) {
15                vis[floor_N] = true;

```

```

16                ans[floor_N] = init_pref[n];
17            }
18            return init_pref[n];
19        }
20
21        if (vis[floor_N]) return ans[floor_N];
22
23        vis[floor_N] = true;
24        ans[floor_N] = 1;
25        int j = 1;
26        for (; (ll)j * j <= n; j++) {
27            if (j > 1) ans[floor_N] -= solve(n / j, floor_N * j, f, F, g) * f(j);
28            { ans[floor_N] -= g(j) * F(n / j); }
29        }
30        --j;
31        ans[floor_N] += init_pref[j] * F(j);
32        return ans[floor_N];
33    }
34
35    vector<mint> solve(auto &&f, auto &&F, auto &&g) {
36        if (N == 1) return vector<mint>(2, 1);
37        vis.assign(N / T + 1, 0);
38        ans.assign(N / T + 1, 0);
39        for (int i = 1; i <= N / T; i++)
40            if (!vis[i]) solve(N / i, i, f, F, g);
41        return ans;
42    }
43};

```

4.2.3 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 $\mathcal{O}(N \cdot \log N)$.

Código: gcd_convolution.cpp

```

1 vector<mint> gcd_convolution(vector<mint> A, vector<mint> B) {
2     int N = (int)max(A.size(), B.size());
3     A.resize(N + 1);
4     B.resize(N + 1);
5     vector<mint> C(N + 1);
6     for (int i = 1; i <= N; i++) {
7         mint a = 0;
8         mint b = 0;

```

```

9     for (int j = i; j <= N; j += i) {
10        a += A[j];
11        b += B[j];
12    }
13    C[i] = a * b;
14 }
15 for (int i = N; i >= 1; i--)
16    for (int j = 2 * i; j <= N; j += i) C[i] -= C[j];
17 return C;
18 }
```

4.2.4 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 $\mathcal{O}(N \cdot \log N)$.

Código: lcm_convolution.cpp

```

1 vector<mint> lcm_convolution(vector<mint> A, vector<mint> B) {
2     int N = (int)max(A.size(), B.size());
3     A.resize(N + 1);
4     B.resize(N + 1);
5     vector<mint> C(N + 1), a(N + 1), b(N + 1);
6     for (int i = 1; i <= N; i++) {
7         for (int j = i; j <= N; j += i) {
8             a[j] += A[i];
9             b[j] += B[i];
10        }
11        C[i] = a[i] * b[i];
12    }
13    for (int i = 1; i <= N; i++)
14        for (int j = 2 * i; j <= N; j += i) C[j] -= C[i];
15    return C;
16 }
```

4.2.5 OR Convolution

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

Código: or_convolution.cpp

```

1 vector<mint> or_convolution(vector<mint> A, vector<mint> B) {
2     int n = (int)max(A.size(), B.size());
3     int N = 0;
4     while ((1 << N) < n) N++;
5     A.resize(1 << N);
6     B.resize(1 << N);
7     vector<mint> C(1 << N);
8     for (int j = 0; j < N; j++) {
9         for (int i = 0; i < 1 << N; i++) {
10            if (i >> j & 1) {
11                A[i] += A[i ^ (1 << j)];
12                B[i] += B[i ^ (1 << j)];
13            }
14        }
15    }
16    for (int i = 0; i < 1 << N; i++) C[i] = A[i] * B[i];
17    for (int j = N - 1; j >= 0; j--) {
18        for (int i = (1 << N) - 1; i >= 0; i--) {
19            if (i >> j & 1) C[i] -= C[i ^ (1 << j)];
20        }
21    }
22 }
```

4.2.6 Subset Convolution

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

Obs: \wedge representa o bitwise and e \parallel representa o bitwise or.

Código: subset_convolution.cpp

```

1 vector<mint> subset_convolution(vector<mint> A, vector<mint> B) {
2     int n = int(max(A.size(), B.size()));
3     int N = 0;
4     while ((1 << N) < n) N++;
5     A.resize(1 << N), B.resize(1 << N);
6     vector<mint> a(1 << N, vector<mint>(N + 1)), b(1 << N, vector<mint>(N + 1));
7     for (int i = 0; i < 1 << N; i++) {
8         int popcnt = __builtin_popcount(i);
9         a[i][popcnt] = A[i];
```

```

10     b[i][popcnt] = B[i];
11 }
12 for (int j = 0; j < N; j++) {
13     for (int i = 0; i < 1 << N; i++) {
14         if (~i >> j & 1) continue;
15         for (int popcnt = 0; popcnt <= N; popcnt++) {
16             a[i][popcnt] += a[i ^ (1 << j)][popcnt];
17             b[i][popcnt] += b[i ^ (1 << j)][popcnt];
18         }
19     }
20 }
21 vector<mint> c(1 << N, vector<mint>(N + 1));
22 for (int i = 0; i < 1 << N; i++) {
23     for (int j = 0; j <= N; j++)
24         for (int k = 0; k + j <= N; k++) c[i][j + k] += a[i][j] * b[i][k];
25 }
26 for (int j = N - 1; j >= 0; j--) {
27     for (int i = (1 << N) - 1; i >= 0; i--) {
28         if (~i >> j & 1) continue;
29         for (int popcnt = 0; popcnt <= N; popcnt++)
30             c[i][popcnt] -= c[i ^ (1 << j)][popcnt];
31     }
32 }
33 vector<mint> ans(1 << N);
34 for (int i = 0; i < 1 << N; i++) {
35     int popcnt = __builtin_popcount(i);
36     ans[i] = c[i][popcnt];
37 }
38 return ans;
39 }
```

4.2.7 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 $\mathcal{O}(N \cdot \log N)$

Código: xor_convolution.cpp

```

1 vector<mint> xor_convolution(vector<mint> A, vector<mint> B) {
2     int n = int(A.size());
3     for (int rep = 0; rep < 2; rep++) {
4         for (int len = n >> 1; len; len >>= 1) {
5             for (int i = 0; i < n; i += len << 1) {
6                 for (int j = 0; j < len; j++) {
```

```

7                     int id = i + j;
8                     mint x = A[id];
9                     mint y = A[id + len];
10                    A[id] = x + y;
11                    A[id + len] = x - y;
12                }
13            }
14        }
15        swap(A, B);
16    }
17    vector<mint> ans(n);
18    for (int i = 0; i < n; i++) ans[i] = A[i] * B[i];
19    for (int len = 1; len < n; len <<= 1) {
20        for (int i = 0; i < n; i += len << 1) {
21            for (int j = 0; j < len; j++) {
22                int id = i + j;
23                mint x = ans[id];
24                mint y = ans[id + len];
25                ans[id] = x + y;
26                ans[id + len] = x - y;
27            }
28        }
29    }
30    return ans;
31 }
32
33 vector<mint> xor_multiply(vector<mint> A, vector<mint> B) {
34     int N = 1;
35     int n = int(max(A.size(), B.size()));
36     while (N < n) N <<= 1;
37     A.resize(N);
38     B.resize(N);
39     auto ans = xor_convolution(A, B);
40     for (int i = 0; i < N; i++) ans[i] /= N;
41 }
42 }
```

4.3 Discrete Root

Fonte: <https://github.com/ShahjalalShohag/code-library/blob/main/Number%20Theory/DiscreteRoot.cpp>

Algoritmo que computa a raiz discreta, isto é, para uma equação $x^k \equiv a \pmod{n}$, sendo n primo; computa algum (ou todos) os valores de x

que a satisfazem.

Complexidade $\approx \mathcal{O}(\sqrt{n} \cdot \log n + \log^6 n)$.

Código: discrete_root.cpp

```

1 int power(int a, int b, int m) {
2     int res = 1;
3     while (b > 0) {
4         if (b & 1) res = 1LL * res * a % m;
5         a = 1LL * a * a % m;
6         b >>= 1;
7     }
8     return res;
9 }
10 // p is prime
11 int primitive_root(int p) {
12     vector<int> fact;
13     int phi = p - 1, n = phi;
14     for (int i = 2; i * i <= n; ++i) {
15         if (n % i == 0) {
16             fact.push_back(i);
17             while (n % i == 0) n /= i;
18         }
19     }
20     if (n > 1) fact.push_back(n);
21     for (int res = 2; res <= p; ++res) { // this loop will run at most (log p ^ 6) times
22         // i.e. until a root is found
23         bool ok = true;
24         // check if this is a primitive root modulo p
25         for (size_t i = 0; i < fact.size() && ok; ++i)
26             ok &= power(res, phi / fact[i], p) != 1;
27         if (ok) return res;
28     }
29     return -1;
30 }
31 // returns any or all numbers x such that x ^ k = a (mod m)
32 // existence: a = 0 is trivial, and if a > 0: a ^ (phi(m) / gcd(k, phi(m))) == 1 mod m
33 // if solution exists, then number of solutions = gcd(k, phi(m)).
34 // here m is prime, but it will work for any integer which has a primitive root
35 int discrete_root(int k, int a, int m) {
36     if (a == 0) return 1;
37     int g = primitive_root(m);
38     int phi = m - 1; // m is prime
39     // run baby step-giant step
40     int sq = (int)sqrt(m + .0) + 1;

```

```

41     vector<pair<int, int>> dec(sq);
42     for (int i = 1; i <= sq; ++i)
43         dec[i - 1] = make_pair(power(g, 1LL * i * sq % phi * k % phi, m), i);
44     sort(dec.begin(), dec.end());
45     int any_ans = -1;
46     for (int i = 0; i < sq; ++i) {
47         int my = power(g, 1LL * i * k % phi, m) * 1LL * a % m;
48         auto it = lower_bound(dec.begin(), dec.end(), make_pair(my, 0));
49         if (it != dec.end() && it->first == my) {
50             any_ans = it->second * sq - i;
51             break;
52         }
53     }
54     if (any_ans == -1) return -1; // no solution
55     // for any answer
56     int delta = (m - 1) / __gcd(k, m - 1);
57     return power(g, any_ans % delta, m);
58
59     // // for all possible answers
60     // int delta = (m - 1) / __gcd(k, m - 1);
61     // vector<int> ans;
62     // for (int cur = any_ans % delta; cur < m-1; cur += delta) ans.push_back(power(g,
63     // cur, m)); sort(ans.begin(), ans.end());
64     // // assert(ans.size() == __gcd(k, m - 1))
65     // return ans;
66 }

```

4.4 Eliminação Gaussiana

4.4.1 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.

Código: gauss.cpp

```

1 const double EPS = 1e-9;
2 const int INF = 2; // nao tem que ser infinito ou um numero grande
3 // so serve para indicar que tem infinitas solucoes
4
5 int gauss(vector<vector<double>> a, vector<double> &ans) {
6     int n = (int)a.size();
7     int m = (int)a[0].size() - 1;
8
9     vector<int> where(m, -1);
10    for (int col = 0, row = 0; col < m && row < n; col++) {
11        int sel = row;
12        for (int i = row; i < n; i++)
13            if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
14        if (abs(a[sel][col]) < EPS) continue;
15        for (int i = col; i <= m; i++) swap(a[sel][i], a[row][i]);
16        where[col] = row;
17
18        for (int i = 0; i < n; i++) {
19            if (i != row) {
20                double c = a[i][col] / a[row][col];
21                for (int j = col; j <= m; j++) a[i][j] -= a[row][j] * c;
22            }
23        }
24        row++;
25    }
26
27    ans.assign(m, 0);
28    for (int i = 0; i < m; i++)
29        if (where[i] != -1) ans[i] = a[where[i]][m] / a[where[i]][i];
30    for (int i = 0; i < n; i++) {
31        double sum = 0;
32        for (int j = 0; j < m; j++) sum += ans[j] * a[i][j];
33        if (abs(sum - a[i][m]) > EPS) return 0;

```

```

34    }
35
36    for (int i = 0; i < m; i++)
37        if (where[i] == -1) return INF;
38    return 1;
39 }
```

4.4.2 Gauss Mod 2

Método de eliminação gaussiana para resolução de sistemas lineares com coeficientes em \mathbb{Z}_2 (inteiros módulo 2), a complexidade é $\mathcal{O}(n^3/\square)$, onde \square 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 vetor 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.

Código: gauss_mod2.cpp

```

1 const int M = 105;
2 const int INF = 2; // nao tem que ser infinito ou um numero grande
3 // so serve para indicar que tem infinitas solucoes
4
5 int gauss(vector<bitset<M>> a, int n, int m, bitset<M> &ans) {
6     vector<int> where(m, -1);
7
8     for (int col = 0, row = 0; col < m && row < n; col++) {
9         for (int i = row; i < n; i++) {
10            if (a[i][col]) {
11                swap(a[i], a[row]);
12                break;
13            }
14        }
15        if (!a[row][col]) continue;
16        where[col] = row;
17
18        for (int i = 0; i < n; i++)
19            if (i != row && a[i][col]) a[i] ^= a[row];
20        row++;
21    }
22
23    ans.reset();
24    for (int i = 0; i < m; i++)
25        if (where[i] != -1) ans[i] = a[where[i]][m] / a[where[i]][i];
26    for (int i = 0; i < n; i++) {
27        int sum = 0;
28        for (int j = 0; j < m; j++) sum += ans[j] * a[i][j];
29        if (abs(sum - a[i][m]) > 0) return 0; // Sem solucao
30    }
31
32    for (int i = 0; i < m; i++)
33        if (where[i] == -1) return INF; // Infinitas solucoes
34    // Unica solucao (retornada no bitset ans)
35    return 1;
36}

```

4.5 Exponenciação Modular Rápida

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

Código: exp_mod.cpp

```

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

```

4.6 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 $3 \cdot 10^5$ e constantes até 10^6 . Para convolução de inteiros sem erro de precisão, consultar a seção de NTT.

Código: fft.cpp

```

1 struct base {
2     double a, b;
3     base(double _a = 0, double _b = 0) : a(_a), b(_b) { }
4     const base operator+(const base &c) const { return base(a + c.a, b + c.b); }
5     const base operator-(const base &c) const { return base(a - c.a, b - c.b); }
6     const base operator*(const base &c) const {
7         return base(a * c.a - b * c.b, a * c.b + b * c.a);
8     }
9 };
10
11 const double PI = acos(-1);
12
13 void fft(vector<base> &a, bool inv = 0) {
14     int n = (int)a.size();
15
16     for (int i = 0; i < n; i++) {
17         int bit = n >> 1, j = 0, k = i;
18         while (bit > 0) {

```

```

19         if (k & 1) j += bit;
20         k >>= 1, bit >>= 1;
21     }
22     if (i < j) swap(a[i], a[j]);
23 }
24
25 double angle = 2 * PI / n * (inv ? -1 : 1);
26 vector<base> wn(n / 2);
27 for (int i = 0; i < n / 2; i++) wn[i] = {cos(angle * i), sin(angle * i)};
28
29 for (int len = 2; len <= n; len <= 1) {
30     int aux = len / 2;
31     int step = n / len;
32     for (int i = 0; i < n; i += len) {
33         for (int j = 0; j < aux; j++) {
34             base v = a[i + j + aux] * wn[step * j];
35             a[i + j + aux] = a[i + j] - v;
36             a[i + j] = a[i + j] + v;
37         }
38     }
39 }
40
41 for (int i = 0; inv && i < n; i++) a[i].a /= n, a[i].b /= n;
42 }
43
44 vector<ll> multiply(vector<ll> &ta, vector<ll> &tb) {
45     int n = (int)ta.size(), m = (int)tb.size();
46     int t = n + m - 1, sz = 1;
47     while (sz < t) sz <= 1;
48
49     vector<base> a(sz), b(sz), c(sz);
50
51     for (int i = 0; i < sz; i++) {
52         a[i] = i < n ? base((double)ta[i]) : base(0);
53         b[i] = i < m ? base((double)tb[i]) : base(0);
54     }
55
56     fft(a, 0), fft(b, 0);
57     for (int i = 0; i < sz; i++) c[i] = a[i] * b[i];
58     fft(c, 1);
59
60     vector<ll> res(sz);
61     for (int i = 0; i < sz; i++) res[i] = ll(round(c[i].a));
62
63     while ((int)res.size() > t && res.back() == 0) res.pop_back();
64
65     return res;
66 }
```

4.7 Fatoração e Primos

4.7.1 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.

Código: `sieve.cpp`

```

1 namespace Sieve {
2     const int P = 5e6 + 1;
3     vector<bool> is_prime(P, true);
4     void build() {
5         is_prime[0] = is_prime[1] = 0;
6         for (int i = 2; i < P; i++) {
7             if (is_prime[i])
8                 for (int j = i + i; j < P; j += i) is_prime[j] = 0;
9         }
10    }
11 }
```

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.

Código: `linear_sieve.cpp`

```

1  namespace Sieve {
2      const int P = 5e6 + 1;
3      vector<bool> is_prime(P, true);
4      int lpf[P], primes[P], cnt = 0;
5      void build() {
6          is_prime[0] = is_prime[1] = 0;
7          for (int i = 2; i < P; i++) {
8              if (is_prime[i]) {
9                  lpf[i] = i;
10                 primes[cnt++] = i;
11             }
12             for (int j = 0; j < cnt && i * primes[j] < P; j++) {
13                 is_prime[i * primes[j]] = 0;
14                 lpf[i * primes[j]] = primes[j];
15                 if (i % primes[j] == 0) break;
16             }
17         }
18     }
19     vector<int> factorize(int n) {
20 #warning lembra de chamar o build() antes de fatorar!
21         vector<int> f;
22         while (n > 1) {
23             f.push_back(lpf[n]);
24             n /= lpf[n];
25         }
26         return f;
27     }
28 }
```

4.7.2 Divisores

Divisores Naive

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

Código: `get_divs_naive.cpp`

```

1  vector<int> get_divs(int n) {
2      vector<int> divs;
3      for (int d = 1; d * d <= n; d++) {
4          if (n % d == 0) {
5              divs.push_back(d);
6              if (d * d != n) divs.push_back(n / d);
7          }
8      }
9      sort(divs.begin(), divs.end());
10     return divs;
11 }
```

Divisores Rápido

Algoritmo que obtém todos os divisores de um número em $\mathcal{O}(d(X))$, 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.

Código: `get_divs.cpp`

```

1 vector<ll> get_divs(ll n) {
2     vector<ll> divs;
3     auto f = factorize(n); // qualquer código que fatore n
4     sort(f.begin(), f.end());
5     vector<pair<ll, int>> v;
6     for (auto x : f)
7         if (v.empty() || v.back().first != x) v.emplace_back(x, 1);
8         else v.back().second += 1;
9     function<void(int, ll)> dfs = [&](int i, ll cur) {
10         if (i == (int)v.size()) {
11             divs.push_back(cur);
12             return;
13         }
14         ll p = 1;
15         for (int j = 0; j <= v[i].second; j++) {
16             dfs(i + 1, cur * p);
17             p *= v[i].first;
18         }
19     };
20     dfs(0, 1);
21     sort(divs.begin(), divs.end());
22     return divs;
23 }
```

4.7.3 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 $\mathcal{O}(\sqrt{X})$.

Código: `naive_factorize.cpp`

```

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

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.

Código: `fast_factorize.cpp`

```

1 vector<ll> factorize(ll y) {
2     vector<ll> f;
3     if (y == 1) return f;
4     function<void(ll)> dfs = [&](ll x) {
5         if (x == 1) return;
6         if (x < Sieve::P) {
7             auto fs = Sieve::factorize(x);
8             f.insert(f.end(), fs.begin(), fs.end());
9         } else if (MillerRabin::prime(x)) {
10             f.push_back(x);
11         } else {
12             ll d = PollardRho::rho(x);
13             dfs(d);
14             dfs(x / d);
15         }
16     };
17     dfs(y);
18     sort(f.begin(), f.end());
19     return f;
20 }
```

4.7.4 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 comportamento é indefinido (provavelmente entra em loop e não retorna nunca).

Código: `pollard_rho.cpp`

```

1  namespace PollardRho {
2      mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
3      const ll P = 1e6 + 1;
4      ll seq[P];
5      inline ll add_mod(ll x, ll y, ll m) { return (x += y) < m ? x : x - m; }
6      inline ll mul_mod(ll a, ll b, ll m) { return (ll)((__int128)a * b % m); }
7      ll rho(ll n) {
8          if (n % 2 == 0) return 2;
9          if (n % 3 == 0) return 3;
10         ll x0 = rng() % n, c = rng() % n;
11         while (1) {
12             ll x = x0++, y = x, u = 1, v, t = 0;
13             ll *px = seq, *py = seq;
14             while (1) {
15                 *py++ = y = add_mod(mul_mod(y, y, n), c, n);
16                 *py++ = y = add_mod(mul_mod(y, y, n), c, n);
17                 if ((x = *px++) == y) break;
18                 v = u;
19                 u = mul_mod(u, abs(y - x), n);
20                 if (!u) return gcd(v, n);
21                 if (++t == 32) {
22                     t = 0;
23                     if ((u = gcd(u, n)) > 1 && u < n) return u;
24                 }
25             }
26             if (t && (u = gcd(u, n)) > 1 && u < n) return u;
27         }
28     }
29 }
```

4.7.5 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 números até 2^{64} . A complexidade do algoritmo é $\mathcal{O}(\log X)$, considerando multiplicação e expoenciação constantes.

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

Código: `miller_rabin.cpp`

```

1  namespace MillerRabin {
2      inline ll mul_mod(ll a, ll b, ll m) { return (ll)((__int128)a * b % m); }
3      inline ll power(ll b, ll e, ll m) {
4          ll r = 1;
5          b = b % m;
6          while (e > 0) {
7              if (e & 1) r = mul_mod(r, b, m);
8              b = mul_mod(b, b, m), e >>= 1;
9          }
10         return r;
11     }
12     inline bool composite(ll n, ll a, ll d, ll s) {
13         ll x = power(a, d, n);
14         if (x == 1 || x == n - 1 || a % n == 0) return false;
15         for (int r = 1; r < s; r++) {
16             x = mul_mod(x, x, n);
17             if (x == n - 1) return false;
18         }
19         return true;
20     }
21
22     // com esses "primos", o teste funciona garantido para n <= 2^64
23     int primes[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
24
25     // funciona para n <= 3*10^24 com os primos ate 41, mas tem que cuidar com overflow
26     // int primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41};
27
28     bool prime(ll n) {
29         if (n <= 2 || (n % 2 == 0)) return n == 2;
30         ll d = n - 1, r = 0;
31         while (d % 2 == 0) d /= 2, r++;
32         for (int a : primes)
33             if (composite(n, a, d, r)) return false;
34     }
35 }
```

```

34     return true;
35 }
36 }
```

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 $\mathcal{O}(\sqrt{X})$.

Código: `naive_is_prime.cpp`

```

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

4.8 Floor Values

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

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

- Somatório de $\lfloor \frac{n}{i} \rfloor$ para $i = 1, 2, \dots, n$.
 - Somatório de $\sigma(i)$ para $i = 1, 2, \dots, n$, 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 .

Código: `floor_values.cpp`

```

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

```

8         // faz algo
9
10        j = last_j + 1;
11    }
12 }
13 }
```

4.9 Floor and Mod Sum of Arithmetic Progressions

Fonte: <https://github.com/ShahjalalShohag/code-library/blob/main/Number%20Theory/FloorModSum.cpp>

Computa soma de $\lfloor \frac{a+d \cdot i}{m} \rfloor$ para $i \in [0, n - 1]$ em $\mathcal{O}(\log m)$.

Código: `floor_and_mod_sum_of_arithmetic_progressions.cpp`

```

1 ll sumsq(ll n) { return n / 2 * ((n - 1) | 1); }
2 // \sum_{i=0}^{n-1} \{(a + d * i) / m\}, O(log m)
3 ll floor_sum(ll a, ll d, ll m, ll n) {
4     ll res = d / m * sumsq(n) + a / m * n;
5     d %= m;
6     a %= m;
7     if (!d) return res;
8     ll to = (n * d + a) / m;
9     return res + (n - 1) * to - floor_sum(m - 1 - a, m, d, to);
10 }
11 // \sum_{i=0}^{n-1} \{(a + d * i) \% m\}
12 ll mod_sum(ll a, ll d, ll m, ll n) {
13     a = ((a \% m) + m) \% m;
14     d = ((d \% m) + m) \% m;
15     return n * a + d * sumsq(n) - m * floor_sum(a, d, m, n);
16 }
```

4.10 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 $a * x + b * y = \text{gcd}(a, b)$.

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

Código: `extended_gcd.cpp`

```

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

Código: `extended_gcd_recursive.cpp`

```

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

```

10    }
11 }
```

Código: `gcd.cpp`

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

4.11 Inverso Modular

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

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: $\mathcal{O}(\log(MOD))$.

* 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: $\mathcal{O}(\log(MOD))$.

* 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 $b^M AX$.

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: $\mathcal{O}(\text{MAX})$.

Código: modular_inverse_coprime.cpp

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

Código: modular_inverse.cpp

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

Código: modular_inverse_linear.cpp

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

Código: modular_inverse_pow.cpp

```
1 const ll INV_B = (MOD + 1) / 2; // Modular inverse of the base,
2 // for 2 it is (MOD+1)/2
3
```

```
4 ll inv[MAX]; // Modular inverse of b^i
5
6 void compute_inv() {
7     inv[0] = 1;
8     for (int i = 1; i < MAX; i++) inv[i] = inv[i - 1] * INV_B % MOD;
9 }
```

4.12 NTT

4.12.1 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.

Código: ntt.cpp

```
1 const int MOD = mod;
2 void ntt(vector<mint> &a, bool inv = 0) {
3     int n = (int)a.size();
4     auto b = a;
5     mint g = 1;
6     while ((g ^ (MOD / 2)) == 1) g += 1;
7     if (inv) g = mint(1) / g;
8     for (int step = n / 2; step; step /= 2) {
```

```

9     mint w = g ^ (MOD / (n / step)), wn = 1;
10    for (int i = 0; i < n / 2; i += step) {
11        for (int j = 0; j < step; j++) {
12            auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
13            b[i + j] = u + v;
14            b[i + n / 2 + j] = u - v;
15        }
16        wn = wn * w;
17    }
18    swap(a, b);
19 }
20 if (inv) {
21     mint invn = mint(1) / n;
22     for (int i = 0; i < n; i++) a[i] *= invn;
23 }
24 }
25
26 vector<mint> multiply(vector<mint> a, vector<mint> b) {
27     int n = (int)a.size(), m = (int)b.size();
28     int t = n + m - 1, sz = 1;
29     while (sz < t) sz <<= 1;
30     a.resize(sz), b.resize(sz);
31     ntt(a, 0), ntt(b, 0);
32     for (int i = 0; i < sz; i++) a[i] *= b[i];
33     ntt(a, 1);
34     while ((int)a.size() > t) a.pop_back();
35     return a;
36 }
```

4.12.2 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.

Código: big_ntt.cpp

```

1 template <auto MOD, typename T = Mint<MOD>>
2 void ntt(vector<T> &a, bool inv = 0) {
3     int n = (int)a.size();
4     auto b = a;
5     T g = 1;
6     while ((g ^ (MOD / 2)) == 1) g *= 1;
7     if (inv) g = T(1) / g;
8     for (int step = n / 2; step; step /= 2) {
9         T w = g ^ (MOD / (n / step)), wn = 1;
10        for (int i = 0; i < n / 2; i += step) {
11            for (int j = 0; j < step; j++) {
12                auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
13                b[i + j] = u + v;
14                b[i + n / 2 + j] = u - v;
15            }
16            wn = wn * w;
17        }
18        swap(a, b);
19    }
20    if (inv) {
21        T invn = T(1) / n;
22        for (int i = 0; i < n; i++) a[i] *= invn;
23    }
24 }
25
26 template <auto MOD, typename T = Mint<MOD>>
27 vector<T> multiply(vector<T> a, vector<T> b) {
28     int n = (int)a.size(), m = (int)b.size();
29     int t = n + m - 1, sz = 1;
30     while (sz < t) sz <<= 1;
31     a.resize(sz), b.resize(sz);
32     ntt<MOD>(a, 0), ntt<MOD>(b, 0);
33     for (int i = 0; i < sz; i++) a[i] *= b[i];
34     ntt<MOD>(a, 1);
35     while ((int)a.size() > t) a.pop_back();
36 }
37
38 ll extended_gcd(ll a, ll b, ll &x, ll &y) {
39     if (b == 0) {
40         x = 1;
41         y = 0;
42         return a;
43     } else {
44         ll g = extended_gcd(b, a % b, y, x);
45         y -= a / b * x;
46         return g;
47 }
```

```

48     }
49 }
50
51 ll crt(array<int, 2> rem, array<int, 2> mod) {
52     __int128 ans = rem[0], m = mod[0];
53     ll x, y;
54     ll g = extended_gcd(mod[1], (ll)m, x, y);
55     if ((ans - rem[1]) % g != 0) return -1;
56     ans = ans + (__int128)1 * (rem[1] - ans) * (m / g) * y;
57     m = (__int128)(mod[1] / g) * (m / g) * g;
58     ans = (ans % m + m) % m;
59     return (ll)ans;
60 }
61
62 template <auto MOD1, auto MOD2, typename T = Mint<MOD1>, typename U = Mint<MOD2>>
63 vector<ll> big_multiply(vector<ll> ta, vector<ll> tb) {
64     vector<T> a1(ta.size()), b1(tb.size());
65     vector<U> a2(ta.size()), b2(tb.size());
66     for (int i = 0; i < (int)ta.size(); i++) a1[i] = ta[i];
67     for (int i = 0; i < (int)tb.size(); i++) b1[i] = tb[i];
68     for (int i = 0; i < (int)ta.size(); i++) a2[i] = ta[i];
69     for (int i = 0; i < (int)tb.size(); i++) b2[i] = tb[i];
70     auto c1 = multiply<MOD1>(a1, b1);
71     vector<ll> res(c1.size());
72     for (int i = 0; i < (int)res.size(); i++)
73         res[i] = crt({c1[i].v, c2[i].v}, {MOD1, MOD2});
74     return res;
75 }
76
77 const int MOD1 = 1004535809;
78 const int MOD2 = 1092616193;

```

4.12.3 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

```

1 template <auto MOD, typename T = Mint<MOD>>
2 vector<T> shift(vector<T> a, int k) {
3     int n = (int)a.size();
4     vector<T> fat(n, 1), ifat(n), shifting(n);
5     for (int i = 1; i < n; i++) fat[i] = fat[i - 1] * i;
6     ifat[n - 1] = T(1) / fat[n - 1];

```

```

7     for (int i = n - 1; i > 0; i--) ifat[i - 1] = ifat[i] * i;
8     for (int i = 0; i < n; i++) a[i] *= fat[i];
9     T pk = 1;
10    for (int i = 0; i < n; i++) {
11        shifting[n - i - 1] = pk * ifat[i];
12        pk *= k;
13    }
14    auto ans = multiply<MOD>(a, shifting);
15    ans.erase(ans.begin(), ans.begin() + n - 1);
16    for (int i = 0; i < n; i++) ans[i] *= ifat[i];
17    return ans;
18 }

```

4.13 Polinomios

Fonte: <https://github.com/ShahjalalShohag/code-library/blob/main/Math/Polynomial.cpp>

Estrutura de polinômio, possui diversas operações; entre elas: divisão, módulo, multiplicação, log, etc.

Além de algoritmos como Multipoint Evaluation (eval).

Depende de ntt.cpp e mint.cpp.

Testado apenas em <https://codeforces.com/gym/105949/problem/G>

Codigo: polinomio.cpp

```

1 struct poly {
2     vector<mint> a;
3     inline void normalize() {
4         while ((int)a.size() && a.back() == 0) a.pop_back();
5     }
6     template <class... Args>
7     poly(Args... args) : a(args...) { }
8     poly(const initializer_list<mint> &x) : a(x.begin(), x.end()) { }
9     int size() const { return (int)a.size(); }
10    inline mint coef(const int i) const {
11        return (i < a.size() && i >= 0) ? a[i] : mint(0);
12    }
13
14    // Beware!! p[i] = k won't change the value of p.a[i]

```

```

15     mint operator[](const int i) const {
16         return (i < a.size() && i >= 0) ? a[i] : mint(0);
17     }
18
19     bool is_zero() const {
20         for (int i = 0; i < size(); i++)
21             if (a[i] != 0) return 0;
22         return 1;
23     }
24     poly operator+(const poly &x) const {
25         int n = max(size(), x.size());
26         vector<mint> ans(n);
27         for (int i = 0; i < n; i++) ans[i] = coef(i) + x.coef(i);
28         while ((int)ans.size() && ans.back() == 0) ans.pop_back();
29         return ans;
30     }
31     poly operator-(const poly &x) const {
32         int n = max(size(), x.size());
33         vector<mint> ans(n);
34         for (int i = 0; i < n; i++) ans[i] = coef(i) - x.coef(i);
35         while ((int)ans.size() && ans.back() == 0) ans.pop_back();
36         return ans;
37     }
38     poly operator*(const poly &b) const {
39         if (is_zero() || b.is_zero()) return {};
40         vector<mint> ans = multiply(a, b.a);
41         while ((int)ans.size() && ans.back() == 0) ans.pop_back();
42         return ans;
43     }
44     poly operator*(const mint &x) const {
45         int n = size();
46         vector<mint> ans(n);
47         for (int i = 0; i < n; i++) ans[i] = a[i] * x;
48         return ans;
49     }
50     poly operator/(const mint &x) const { return (*this) * x.inv(); }
51     poly &operator+=(const poly &x) { return *this = (*this) + x; }
52     poly &operator-=(const poly &x) { return *this = (*this) - x; }
53     poly &operator*=(const poly &x) { return *this = (*this) * x; }
54     poly &operator/=(const mint &x) { return *this = (*this) / x; }
55     poly &operator/=(const mint &x) { return *this = (*this) / x; }
56     poly mod_xk(int k) const {
57         // modulo by x^k
58         return {a.begin(), a.begin() + min(k, size())};
59     }
60     poly mul_xk(int k) const {
61         // multiply by x^k
62         poly ans(*this);
63         ans.a.insert(ans.a.begin(), k, 0);
64         return ans;
65     }
66     poly div_xk(int k) const {
67         // divide by x^k
68         return vector<mint>(a.begin() + min(k, (int)a.size()), a.end());
69     }
70     poly substr(int l, int r) const {
71         // return mod_xk(r).div_xk(l)
72         l = min(l, size());
73         r = min(r, size());
74         return vector<mint>(a.begin() + l, a.begin() + r);
75     }
76     poly reverse_it(int n, bool rev = 0) const {
77         // reverses and leaves only n terms
78         poly ans(*this);
79         if (rev) // if rev = 1 then tail goes to head
80             ans.a.resize(max(n, (int)ans.a.size()));
81         reverse(ans.a.begin(), ans.a.end());
82         return ans.mod_xk(n);
83     }
84     poly differentiate() const {
85         int n = size();
86         vector<mint> ans(n);
87         for (int i = 1; i < size(); i++) ans[i - 1] = coef(i) * i;
88         return ans;
89     }
90     poly integrate() const {
91         int n = size();
92         vector<mint> ans(n + 1);
93         for (int i = 0; i < size(); i++) ans[i + 1] = coef(i) / (i + 1);
94         return ans;
95     }
96     poly inverse(int n) const {
97         // 1 / p(x) % x^n, O(nlogn)
98         assert(!is_zero());
99         assert(a[0] != 0);
100        poly ans{mint(1) / a[0]};
101        for (int i = 1; i < n; i *= 2)
102            ans = (ans * mint(2) - ans * ans * mod_xk(2 * i)).mod_xk(2 * i);
103        return ans.mod_xk(n);
104    }
105    pair<poly, poly> divmod_slow(const poly &b) const {
106        // when divisor or quotient is small
107        vector<mint> A(a);
108        vector<mint> ans;

```

```

109     while (A.size() >= b.a.size()) {
110         ans.push_back(A.back() / b.a.back());
111         if (ans.back() != mint(0))
112             for (size_t i = 0; i < b.a.size(); i++)
113                 A[A.size() - i - 1] -= ans.back() * b.a[b.a.size() - i - 1];
114         A.pop_back();
115     }
116     reverse(ans.begin(), ans.end());
117     return {ans, A};
118 }
119 pair<poly, poly> divmod(const poly &b) const {
120     // returns quotient and remainder of a mod b
121     if (size() < b.size()) return {poly{0}, *this};
122     int d = size() - b.size();
123     if (min(d, b.size()) < 250) return divmod_slow(b);
124     poly D = (reverse_it(d + 1) * b.reverse_it(d + 1).inverse(d + 1))
125         .mod_xk(d + 1)
126         .reverse_it(d + 1, 1);
127     return {D, *this - (D * b)};
128 }
129 poly operator/(const poly &t) const { return divmod(t).first; }
130 poly operator%(const poly &t) const { return divmod(t).second; }
131 poly &operator/=(const poly &t) { return *this = divmod(t).first; }
132 poly &operator%=(const poly &t) { return *this = divmod(t).second; }
133 poly log(int n) const {
134     // ln p(x) mod x^n
135     assert(a[0] == 1);
136     return (differentiate().mod_xk(n) * inverse(n)).integrate().mod_xk(n);
137 }
138 poly exp(int n) const {
139     // e ^ p(x) mod x^n
140     if (is_zero()) return {1};
141     assert(a[0] == 0);
142     poly ans({1});
143     int i = 1;
144     while (i < n) {
145         poly C = ans.log(2 * i).div_xk(i) - substr(i, 2 * i);
146         ans -= (ans * C).mod_xk(i).mul_xk(i);
147         i *= 2;
148     }
149     return ans.mod_xk(n);
150 }
151 // better for small k, k < 100000
152 poly pow(int k, int n) const {
153     // p(x)^k mod x^n
154     if (is_zero()) return *this;
155     poly ans({1}), b = mod_xk(n);
156     while (k) {
157         if (k & 1) ans = (ans * b).mod_xk(n);
158         b = (b * b).mod_xk(n);
159         k >>= 1;
160     }
161     return ans;
162 }
163 int leading_xk() const {
164     // minimum i such that C[i] > 0
165     if (is_zero()) return 1000000000;
166     int res = 0;
167     while (a[res] == 0) res++;
168     return res;
169 }
170 // better for k > 100000
171 poly pow2(int k, int n) const {
172     // p(x)^k mod x^n
173     if (is_zero()) return *this;
174     int i = leading_xk();
175     mint j = a[i];
176     poly t = div_xk(i) / j;
177     poly ans = (t.log(n) * mint(k)).exp(n);
178     if (1LL * i * k > n) ans = {0};
179     else ans = ans.mul_xk(i * k).mod_xk(n);
180     ans *= (j.pwr(j, k));
181     return ans;
182 }
183 // if the poly is not zero but the result is zero, then no solution
184 // poly sqrt(int n) const {
185 //     if ((*this)[0] == mint(0)) {
186 //         for (int i = 1; i < size(); i++) {
187 //             if ((*this)[i] != mint(0)) {
188 //                 if (i & 1) return {};
189 //                 if (n - i / 2 <= 0) break;
190 //                 return div_xk(i).sqrt(n - i / 2).mul_xk(i / 2);
191 //             }
192 //         }
193 //         return {};
194 //     }
195 //     mint s = (*this)[0].sqrt();
196 //     if (s == 0) return {};
197 //     poly y = *this / (*this)[0];
198 //     poly ret({1});
199 //     mint inv2 = mint(1) / 2;
200 //     for (int i = 1; i < n; i <= 1) {
201 //         ret = (ret + y.mod_xk(i << 1) * ret.inverse(i << 1)) * inv2;
202 //     }

```

```

203 // return ret.mod_xk(n) * s;
204 //}
205 poly root(int n, int k = 2) const {
206     // kth root of p(x) mod x^n
207     if (is_zero()) return *this;
208     if (k == 1) return mod_xk(n);
209     assert(a[0] == 1);
210     poly q({1});
211     for (int i = 1; i < n; i <= 1) {
212         if (k == 2) q += mod_xk(2 * i) * q.inverse(2 * i);
213         else q = q * mint(k - 1) + mod_xk(2 * i) * q.inverse(2 * i).pow(k - 1, 2 *
214             i);
215         q = q.mod_xk(2 * i) / mint(k);
216     }
217     return q.mod_xk(n);
218 }
219 poly mulx(mint x) {
220     // component-wise multiplication with x^k
221     mint cur = 1;
222     poly ans(*this);
223     for (int i = 0; i < size(); i++) ans.a[i] *= cur, cur *= x;
224     return ans;
225 }
226 poly mulx_sq(mint x) {
227     // component-wise multiplication with x^{k^2}
228     mint cur = x, total = 1, xx = x * x;
229     poly ans(*this);
230     for (int i = 0; i < size(); i++) ans.a[i] *= total, total *= cur, cur *= xx;
231     return ans;
232 }
233 vector<mint> chirpz_even(mint z, int n) {
234     // P(1), P(z^2), P(z^4), ..., P(z^{2(n-1)})
235     int m = size() - 1;
236     if (is_zero()) return vector<mint>(n, 0);
237     vector<mint> vv(m + n);
238     mint iz = z.inv(), zz = iz * iz, cur = iz, total = 1;
239     for (int i = 0; i <= max(n - 1, m); i++) {
240         if (i <= m) vv[m - i] = total;
241         if (i < n) vv[m + i] = total;
242         total *= cur;
243         cur *= zz;
244     }
245     poly w = (mulx_sq(z) * vv).substr(m, m + n).mulx_sq(z);
246     vector<mint> ans(n);
247     for (int i = 0; i < n; i++) ans[i] = w[i];
248     return ans;
}

```

```

249     // O(nlogn)
250     vector<mint> chirpz(mint z, int n) {
251         // P(1), P(z), P(z^2), ..., P(z^{n-1})
252         auto even = chirpz_even(z, (n + 1) / 2);
253         auto odd = mulx(z).chirpz_even(z, n / 2);
254         vector<mint> ans(n);
255         for (int i = 0; i < n / 2; i++) {
256             ans[2 * i] = even[i];
257             ans[2 * i + 1] = odd[i];
258         }
259         if (n % 2 == 1) ans[n - 1] = even.back();
260         return ans;
261     }
262     poly shift_it(int m, vector<poly> &pw) {
263         if (size() <= 1) return *this;
264         while (m >= size()) m /= 2;
265         poly q(a.begin() + m, a.end());
266         return q.shift_it(m, pw) * pw[m] + mod_xk(m).shift_it(m, pw);
267     }
268     // n log(n)
269     poly shift(mint a) {
270         // p(x + a)
271         int n = size();
272         if (n == 1) return *this;
273         vector<poly> pw(n);
274         pw[0] = poly({1});
275         pw[1] = poly({a, 1});
276         int i = 2;
277         for (; i < n; i *= 2) pw[i] = pw[i / 2] * pw[i / 2];
278         return shift_it(i, pw);
279     }
280     mint eval(mint x) {
281         // evaluates in single point x
282         mint ans(0);
283         for (int i = size() - 1; i >= 0; i--) {
284             ans *= x;
285             ans += a[i];
286         }
287         return ans;
288     }
289     // p(g(x))
290     // O(n^2 logn)
291     poly composition_brute(poly g, int deg) {
292         int n = size();
293         poly c(deg, 0), pw({1});
294         for (int i = 0; i < min(deg, n); i++) {
295             int d = min(deg, (int)pw.size());

```

```

296     for (int j = 0; j < d; j++) c.a[j] += coef(i) * pw[j];
297     pw *= g;
298     if (pw.size() > deg) pw.a.resize(deg);
299   }
300   return c;
301 }
302 // p(g(x))
303 // O(nlogn * sqrt(nlogn))
304 poly composition(poly g, int deg) {
305   int n = size();
306   int k = 1;
307   while (k * k <= n) k++;
308   k--;
309   int d = n / k;
310   if (k * d < n) d++;
311   vector<poly> pw(k + 3, poly({1}));
312   for (int i = 1; i <= k + 2; i++) {
313     pw[i] = pw[i - 1] * g;
314     if (pw[i].size() > deg) pw[i].a.resize(deg);
315   }
316   vector<poly> fi(k, poly(deg, 0));
317   for (int i = 0; i < k; i++) {
318     for (int j = 0; j < d; j++) {
319       int idx = i * d + j;
320       if (idx >= n) break;
321       int sz = min(fi[i].size(), pw[j].size());
322       for (int t = 0; t < sz; t++) fi[i].a[t] += pw[j][t] * coef(idx);
323     }
324   }
325   poly ret(deg, 0), gd({1});
326   for (int i = 0; i < k; i++) {
327     fi[i] = fi[i] * gd;
328     int sz = min((int)ret.size(), (int)fi[i].size());
329     for (int j = 0; j < sz; j++) ret.a[j] += fi[i][j];
330     gd = gd * pw[d];
331     if (gd.size() > deg) gd.a.resize(deg);
332   }
333   return ret;
334 }
335 poly build(vector<poly> &ans, int v, int l, int r, vector<mint> &vec) {
336   // builds evaluation tree for (x-a1)(x-a2)...(x-an)
337   if (l == r) return ans[v] = poly({-vec[1], 1});
338   int mid = l + r >> 1;
339   return ans[v] = build(ans, 2 * v, l, mid, vec) *
340     build(ans, 2 * v + 1, mid + 1, r, vec);
341 }
342 vector<mint> eval(vector<poly> &tree, int v, int l, int r, vector<mint> &vec) {

```

```

343   // auxiliary evaluation function
344   if (l == r) return {eval(vec[1])};
345   if (size() < 400) {
346     vector<mint> ans(r - l + 1, 0);
347     for (int i = l; i <= r; i++) ans[i - l] = eval(vec[i]);
348     return ans;
349   }
350   int mid = l + r >> 1;
351   auto A = (*this % tree[2 * v]).eval(tree, 2 * v, l, mid, vec);
352   auto B = (*this % tree[2 * v + 1]).eval(tree, 2 * v + 1, mid + 1, r, vec);
353   A.insert(A.end(), B.begin(), B.end());
354   return A;
355 }
356 // O(nlog^2n)
357 vector<mint> eval(vector<mint> x) {
358   // evaluate polynomial in (x_0, ..., x_n-1)
359   int n = x.size();
360   if (is_zero()) return vector<mint>(n, mint(0));
361   vector<poly> tree(4 * n);
362   build(tree, 1, 0, n - 1, x);
363   return eval(tree, 1, 0, n - 1, x);
364 }
365 poly interpolate(
366   vector<poly> &tree, int v, int l, int r, int ly, int ry, vector<mint> &yy
367 ) {
368   // auxiliary interpolation function
369   if (l == r) return {y[ly] / a[0]};
370   int mid = l + r >> 1;
371   int midy = ly + ry >> 1;
372   auto A = (*this % tree[2 * v]).interpolate(tree, 2 * v, l, mid, ly, midy, y);
373   auto B = (*this % tree[2 * v + 1])
374     .interpolate(tree, 2 * v + 1, mid + 1, r, midy + 1, ry, y);
375   return A * tree[2 * v + 1] + B * tree[2 * v];
376 }
377 };

```

4.14 Teorema do Resto Chinês

Algoritmo que resolve o sistema $x \equiv a_i \pmod{m_i}$, onde m_i são primos entre si. Retorna -1 se a resposta não existir.

Código: crt.cpp

```

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

4.15 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 $\mathcal{O}(N \cdot \log(\log N))$.

Código: phi.cpp

```

1  int phi(int n) {
2      int result = n;
3      for (int i = 2; i * i <= n; i++) {
```

```

4          if (n % i == 0) {
5              while (n % i == 0) n /= i;
6              result -= result / i;
7          }
8      }
9      if (n > 1) result -= result / n;
10     return result;
11 }
```

Código: phi_1_to_n.cpp

```

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

4.16 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 $\mathcal{O}(L)$.

Código: xor_gauss.cpp

```

1 const int L = 60;
2 struct Basis {
3     ll B[L], R, last_bit;
4     Basis() { memset(B, 0, sizeof B), R = 0; }
5     ll reduce(ll x) {
6         for (int i = L - 1; i >= 0; i--) {
7             if ((x >> i) & 1) {
8                 if (B[i] != 0) {
9                     x ^= B[i];
10                } else {
11                    last_bit = i;
12                    return x;
13                }
14            }
15        }
16        // assert(x == 0);
17        return 0;
18    }
19    bool insert(ll x) {
20        x = reduce(x);
21        if (x > 0) {

```

```

22            R++;
23            B[last_bit] = x;
24            return true;
25        }
26        return false;
27    }
28    ll kth_smallest(ll k) {
29        ll ans = 0;
30        ll half = 1LL << (R - 1);
31        for (int i = L - 1; i >= 0; i--) {
32            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

Paradigmas

5.1 All Submasks

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

Código: all_submask.cpp

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

5.2 Busca Binária 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.

Código: busca_binaria_paralela.cpp

```
1
2 namespace parallel_binary_search {
3     typedef tuple<int, int, long long, long long> query; // {value, id, l, r}
4     vector<query> queries[1123456]; // pode ser um mapa se
5                                     // for muito esparsos
```

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

```

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

```

5.3 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.

Código: busca_ternaria_discreta.cpp

```

1
2 long long eval(long long mid) {
3     // implement the evaluation
4 }
5
6 long long discrete_ternary_search(long long l, long long r) {
7     long long ans = -1;
8     r--; // to not space r
9     while (l <= r) {
10         long long mid = (l + r) / 2;
11
12         // minimizing. To maximize use >= to
13         // compare
14         if (eval(mid) <= eval(mid + 1)) {
15             ans = mid;
16             r = mid - 1;
17         } else {
18             l = mid + 1;
19         }
20     }
21     return ans;
22 }

```

Código: busca_ternaria.cpp

```

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

```

19 }

5.4 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: $\mathcal{O}(1)$

Código: Convex Hull Trick.cpp

```

1 const ll INF = 1e18 + 18;
2 bool op(ll a, ll b) {
3     return a >= b; // either >= or <=
4 }
5 struct line {
6     ll a, b;
7     ll get(ll x) { return a * x + b; }
8     ll intersect(line l) {
9         return (l.b - b + a - l.a) / (a - l.a); // rounds up for integer
10        // only
11    }
12 };
13 deque<pair<line, ll>> fila;
14 void add_line(ll a, ll b) {
15     line nova = {a, b};
16     if (!fila.empty() && fila.back().first.a == a && fila.back().first.b == b) return;
17     while (!fila.empty() && op(fila.back().second, nova.intersect(fila.back().first)))
18         fila.pop_back();
19     ll x = fila.empty() ? -INF : nova.intersect(fila.back().first);

```

```

20     fila.emplace_back(nova, x);
21 }
22 ll get_binary_search(ll x) {
23     int esq = 0, dir = fila.size() - 1, r = -1;
24     while (esq <= dir) {
25         int mid = (esq + dir) / 2;
26         if (op(x, fila[mid].second)) {
27             esq = mid + 1;
28             r = mid;
29         } else {
30             dir = mid - 1;
31         }
32     }
33     return fila[r].first.get(x);
34 }
35 // O(1), use only when QUERIES are monotonic!
36 ll get(ll x) {
37     while (fila.size() >= 2 && op(x, fila[1].second)) fila.pop_front();
38     return fila.front().first.get(x);
39 }
```

5.5 DP de Permutação

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)`.

Código: tsp_dp.cpp

```

1 const int lim = 17; // setar para o maximo de itens
2 long double dist[lim][lim]; // eh preciso dar as
3 // distancias de n para n
4 long double dp[lim][1 << lim];
5
6 int limMask = (1 << lim) - 1; // 2**maximo de itens) - 1
7 long double solve(int atual, int mask, int n) {
8     if (dp[atual][mask] != 0) return dp[atual][mask];
9     if (mask == (1 << n) - 1) {
10        return dp[atual][mask] = 0; // o que fazer quando
11        // chega no final
```

```

12     }
13
14     long double res = 1e13; // pode ser maior se precisar
15     for (int i = 0; i < n; i++) {
16         if (!(mask & (1 << i))) {
17             long double aux = solve(i, mask | (1 << i), n);
18             if (mask) aux += dist[atual][i];
19             res = min(res, aux);
20         }
21     }
22     return dp[atual][mask] = res;
23 }
```

5.6 Divide and Conquer

Otimização para DP de prefixo quando se pretende separar o vetor em K subgrupos.

É preciso fazer a função $\text{query}(i, j)$ 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: $\mathcal{O}(n \cdot k \cdot \log(n) \cdot \mathcal{O}(\text{update da janela}))$

Código: dc_query_on_demand.cpp

```

1 namespace DC {
2     struct range { // eh preciso definir a forma
3         // de calcular o range
4         vi freq;
5         ll sum = 0;
6         int l = 0, r = -1;
7         void back_l(int v) { // Mover o 'l' do range
```

```

8             sum += freq[v]; // para a esquerda
9             freq[v]++;
10            l--;
11        }
12    }
13    void advance_r(int v) { // Mover o 'r' do range
14        // para a direita
15        sum += freq[v];
16        freq[v]++;
17        r++;
18    }
19    void advance_l(int v) { // Mover o 'l' do range
20        // para a direita
21        freq[v]--;
22        sum -= freq[v];
23        l++;
24    }
25    void back_r(int v) { // Mover o 'r' do range
26        // para a esquerda
27        freq[v]--;
28        sum -= freq[v];
29        r--;
30    }
31    void clear(int n) { // Limpar range
32        l = 0;
33        r = -1;
34        sum = 0;
35        freq.assign(n + 5, 0);
36    }
37 } s;
38
39 vi dp_before, dp_cur;
40 void compute(int l, int r, int optl, int optr) {
41     if (l > r) return;
42     int mid = (l + r) >> 1;
43     pair<ll, int> best = {0, -1}; // {INF, -1} se quiser minimizar
44
45     while (s.l < optl) s.advance_l(v[s.l]);
46     while (s.l > optl) s.back_l(v[s.l - 1]);
47     while (s.r < mid) s.advance_r(v[s.r + 1]);
48     while (s.r > mid) s.back_r(v[s.r]);
49
50     vi removed;
51     for (int i = optl; i <= min(mid, optr); i++) {
52         best =
53             min(best,
54                 {(i ? dp_before[i - 1] : 0) + s.sum, i}); // min() se quiser
```

```

        minimizar
55     removed.push_back(v[s.l]);
56     s.advance_l(v[s.l]);
57 }
58 for (int rem : removed) s.back_l(v[s.l - 1]);
59
60 dp_cur[mid] = best.first;
61 int opt = best.second;
62 compute(l, mid - 1, optl, opt);
63 compute(mid + 1, r, opt, optr);
64 }
65
66 ll solve(int n, int k) {
67     dp_before.assign(n, 0);
68     dp_cur.assign(n, 0);
69     s.clear(n);
70     for (int i = 0; i < n; i++) {
71         s.advance_r(v[i]);
72         dp_before[i] = s.sum;
73     }
74     for (int i = 1; i < k; i++) {
75         s.clear(n);
76         compute(0, n - 1, 0, n - 1);
77         dp_before = dp_cur;
78     }
79     return dp_before[n - 1];
80 }
81 };

```

Código: dc.cpp

```

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

```

```

17     ll solve(int n, int k) {
18         dp_before.assign(n + 5, 0);
19         dp_cur.assign(n + 5, 0);
20         for (int i = 0; i < n; i++) dp_before[i] = query(0, i);
21         for (int i = 1; i < k; i++) {
22             compute(0, n - 1, 0, n - 1);
23             dp_before = dp_cur;
24         }
25         return dp_before[n - 1];
26     }
27 };

```

5.7 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:

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

mantém 1
mantém i
mantém i^2

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 iniciais e então fazer uma exponenciação para cada fator primo. Depois é só juntar a resposta no final.

Código: matrix_exp.cpp

```

1 using mat = vector<vector<ll>>;
2 ll dp[100];
3 mat T;
4
5 #define MOD 1000000007
6
```

```

7 mat operator*(mat a, mat b) {
8     mat res(a.size(), vector<ll>(b[0].size()));
9     for (int i = 0; i < a.size(); i++) {
10         for (int j = 0; j < b[0].size(); j++) {
11             for (int k = 0; k < b.size(); k++) {
12                 res[i][j] += a[i][k] * b[k][j] % MOD;
13                 res[i][j] %= MOD;
14             }
15         }
16     }
17     return res;
18 }

19
20 mat operator^(mat a, ll k) {
21     mat res(a.size(), vector<ll>(a.size()));
22     for (int i = 0; i < a.size(); i++) res[i][i] = 1;
23     while (k) {
24         if (k & 1) res = res * a;
25         a = a * a;
26         k >>= 1;
27     }
28     return res;
29 }

30
31 // MUDA MUITO DE ACORDO COM O PROBLEMA
32 // LEIA COMO FAZER O MAPEAMENTO NO README
33 ll solve(ll exp, ll dim) {
34     if (exp < dim) return dp[exp];
35
36     T.assign(dim, vi(dim));
37     // TO DO: Preencher a Matriz que vai ser
38     // exponenciada T[0][1] = 1; T[1][0] = 1;
39     // T[1][1] = 1;
40
41     mat prod = T ^ exp;
42
43     mat vec;
44     vec.assign(dim, vi(1));
45     for (int i = 0; i < dim; i++) vec[i][0] = dp[i]; // Valores iniciais
46
47     mat ans = prod * vec;
48     return ans[0][0];
49 }
```

5.8 Mo

5.8.1 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 $\mathcal{O}(N * \sqrt{Q})$.

Código: mo.cpp

```

1 typedef pair<int, int> ii;
2 int block_sz; // Better if 'const';
3
4 namespace mo {
5     struct query {
6         int l, r, idx;
7         bool operator<(query q) const {
8             int _l = l / block_sz;
9             int _ql = q.l / block_sz;
10            return ii(_l, _l & 1 ? -r : r) < ii(_ql, _ql & 1 ? -q.r : q.r);
11        }
12    };
13    vector<query> queries;
14
15    void build(int n) {
16        block_sz = (int)sqrt(n);
17        // TODO: initialize data structure
18    }
19    inline void add_query(int l, int r) {
20        queries.push_back({l, r, (int)queries.size()});
21    }
22    inline void remove(int idx) {
23        // TODO: remove value at idx from data
24        // structure
25    }
26    inline void add(int idx) {
27        // TODO: add value at idx from data

```

```

28        // structure
29    }
30    inline int get_answer() {
31        // TODO: extract the current answer of the
32        // data structure
33        return 0;
34    }
35
36    vector<int> run() {
37        vector<int> answers(queries.size());
38        sort(queries.begin(), queries.end());
39        int L = 0;
40        int R = -1;
41        for (query q : queries) {
42            while (L > q.l) add(--L);
43            while (R < q.r) add(++R);
44            while (L < q.l) remove(L++);
45            while (R > q.r) remove(R--);
46            answers[q.idx] = get_answer();
47        }
48        return answers;
49    }
50
51 };

```

5.8.2 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 $\mathcal{O}(Q \cdot \sqrt[3]{N^2})$

Código: mo_update.cpp

```

1 typedef pair<int, int> ii;
2 typedef tuple<int, int, int> iii;
3 int block_sz; // Better if 'const';
4 vector<int> vec;
5 namespace mo {
6     struct query {

```

```

7     int l, r, t, idx;
8     bool operator<(query q) const {
9         int _l = l / block_sz;
10        int _r = r / block_sz;
11        int _ql = q.l / block_sz;
12        int _qr = q.r / block_sz;
13        return iii(_l, _l & 1 ? -_r : _r, _r & 1 ? t : -t) <
14            iii(_ql, _ql & 1 ? -_qr : _qr, _qr & 1 ? q.t : -q.t);
15    }
16 };
17 vector<query> queries;
18 vector<ii> updates;
19
20 void build(int n) {
21     block_sz = pow(1.4142 * n, 2.0 / 3);
22     // TODO: initialize data structure
23 }
24 inline void add_query(int l, int r) {
25     queries.push_back({l, r, (int)updates.size(), (int)queries.size()});
26 }
27 inline void add_update(int x, int v) { updates.push_back({x, v}); }
28 inline void remove(int idx) {
29     // TODO: remove value at idx from data
30     // structure
31 }
32 inline void add(int idx) {
33     // TODO: add value at idx from data
34     // structure
35 }
36 inline void update(int l, int r, int t) {
37     auto &[x, v] = updates[t];
38     if (l <= x && x <= r) remove(x);
39     swap(vec[x], v);
40     if (l <= x && x <= r) add(x);
41 }
42 inline int get_answer() {
43     // TODO: extract the current answer from
44     // the data structure
45     return 0;
46 }
47
48 vector<int> run() {
49     vector<int> answers(queries.size());
50     sort(queries.begin(), queries.end());
51     int L = 0;
52     int R = -1;
53     int T = 0;

```

```

54     for (query q : queries) {
55         while (T < q.t) update(L, R, T++);
56         while (T > q.t) update(L, R, --T);
57         while (L > q.l) add(--L);
58         while (R < q.r) add(++R);
59         while (L < q.l) remove(L++);
60         while (R > q.r) remove(R--);
61         answers[q.idx] = get_answer();
62     }
63     return answers;
64 }
65 };

```

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 $-MOD$ e $MOD - 1$, o próprio Mint normaliza depois para ficar entre 0 e $MOD - 1$.

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`.

```
1 using mint = Mint<7>;
2 // using mint = Mint<(11)1e18 + 9> para long long
3 mint a = 4, b = 3;
4 mint c = a * b; // c.v == 5
5 mint d = 1 / a; // d.v == 2, MOD deve ser primo para usar o operador de divisão
6 mint e = a * d // e.v == 1
7 a = a + 2; // a.v == 6
8 a = a + 3; // a.v == 2
9 a = a ^ 5; // a.v == 4
10 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^{MOD-2} , ou seja, para isso o MOD deve ser primo.

Código: mint.cpp

```
1 template <auto MOD, typename T = decltype(MOD)>
2 struct Mint {
3     using U = long long;
4     // se o modulo for long long, usar U = __int128
5     using m = Mint<MOD, T>;
6     T v;
7     Mint(T val = 0) : v(val) {
8         assert(sizeof(T) * 2 <= sizeof(U));
9         if (v < -MOD || v >= 2 * MOD) v %= MOD;
10        if (v < 0) v += MOD;
11        if (v >= MOD) v -= MOD;
12    }
13    Mint(U val) : v(T(val % MOD)) {
14        assert(sizeof(T) * 2 <= sizeof(U));
15        if (v < 0) v += MOD;
16    }
17    bool operator==(const Mint& o) const { return v == o.v; }
18    bool operator<(const Mint& o) const { return v < o.v; }
19    bool operator!=(const Mint& o) const { return v != o.v; }
20    Mint pwr(m b, U e) const {
21        m res = 1;
22        while (e > 0) {
23            if (e & 1) res *= b;
24            b *= b, e /= 2;
25        }
26        return res;
27    }
28    Mint& operator+=(const Mint& o) {
29        v = (v + o.v) % MOD;
30        if (v < 0) v += MOD;
31        if (v >= MOD) v -= MOD;
32        return *this;
33    }
34    Mint& operator-=(const Mint& o) {
35        v = (v - o.v) % MOD;
36        if (v < 0) v += MOD;
37        if (v >= MOD) v -= MOD;
38        return *this;
39    }
40    Mint& operator*=(const Mint& o) {
41        v = (v * o.v) % MOD;
42        if (v < 0) v += MOD;
43        if (v >= MOD) v -= MOD;
44        return *this;
45    }
46    Mint& operator/=(const Mint& o) {
47        v = pwr(o, MOD-2).v * v % MOD;
48        if (v < 0) v += MOD;
49        if (v >= MOD) v -= MOD;
50        return *this;
51    }
52    Mint& operator%=(const Mint& o) {
53        v = v % o.v;
54        if (v < 0) v += o.v;
55        if (v >= o.v) v -= o.v;
56        return *this;
57    }
58    Mint& operator<<=(int e) {
59        v = pwr(2, e).v * v % MOD;
60        if (v < 0) v += MOD;
61        if (v >= MOD) v -= MOD;
62        return *this;
63    }
64    Mint& operator>>=(int e) {
65        v = pwr(2, -e).v * v % MOD;
66        if (v < 0) v += MOD;
67        if (v >= MOD) v -= MOD;
68        return *this;
69    }
70    Mint& operator^=(const Mint& o) {
71        v = (v ^ o.v) % MOD;
72        if (v < 0) v += MOD;
73        if (v >= MOD) v -= MOD;
74        return *this;
75    }
76    Mint& operator~() {
77        v = (MOD - v) % MOD;
78        if (v < 0) v += MOD;
79        if (v >= MOD) v -= MOD;
80        return *this;
81    }
82    Mint& operator=(const Mint& o) {
83        v = o.v;
84        if (v < 0) v += MOD;
85        if (v >= MOD) v -= MOD;
86        return *this;
87    }
88    Mint& operator=(U v) {
89        this->v = v;
90        if (v < 0) v += MOD;
91        if (v >= MOD) v -= MOD;
92        return *this;
93    }
94    friend std::ostream& operator<<(std::ostream& os, const Mint& m) {
95        os << m.v;
96        return os;
97    }
98    friend std::istream& operator>>(std::istream& is, Mint& m) {
99        is >> m.v;
100       if (m.v < 0) m.v += MOD;
101       if (m.v >= MOD) m.v -= MOD;
102       return is;
103   }
104 }
```

```

29     v -= MOD - o.v;
30     if (v < 0) v += MOD;
31     return *this;
32 }
33 m &operator-=(m o) {
34     v -= o.v;
35     if (v < 0) v += MOD;
36     return *this;
37 }
38 m &operator*=(m o) {
39     v = (T)((U)v * o.v % MOD);
40     return *this;
41 }
42 m &operator/=(m o) { return *this *= o.pwr(o, MOD - 2); }
43 m &operator^=(U e) { return *this = pwr(*this, e); }
44 friend m operator-(m a, m b) { return a -= b; }
45 friend m operator+(m a, m b) { return a += b; }
46 friend m operator*(m a, m b) { return a *= b; }
47 friend m operator/(m a, m b) { return a /= b; }
48 friend m operator^(m a, U e) { return a.pwr(a, e); }
49
50 m operator-() { return m(this->v ? MOD - this->v : 0); }
51 m inv() const { return pwr(*this, MOD - 2); } // MOD must be prime
52 };

```

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.

Código: point2d.cpp

```

1 template <typename T>
2 struct point {
3     T x, y;
4     point(T _x = 0, T _y = 0) : x(_x), y(_y) { }
5
6     using p = point;
7
8     p operator*(const T o) { return p(o * x, o * y); }

```

```

9     p operator-(const p o) { return p(x - o.x, y - o.y); }
10    p operator+(const p o) { return p(x + o.x, y + o.y); }
11    T operator*(const p o) { return x * o.x + y * o.y; }
12    T operator^(const p o) { return x * o.y - y * o.x; }
13    bool operator<(const p o) const { return (x == o.x) ? y < o.y : x < o.x; }
14    bool operator==(const p o) const { return (x == o.x) and (y == o.y); }
15    bool operator!=(const p o) const { return (x != o.x) or (y != o.y); }
16
17    T dist2(const p o) {
18        T dx = x - o.x, dy = y - o.y;
19        return dx * dx + dy * dy;
20    }
21
22    friend ostream &operator<<(ostream &out, const p &a) {
23        return out << "(" << a.x << "," << a.y << ")";
24    }
25    friend istream &operator>>(istream &in, p &a) { return in >> a.x >> a.y; }
26 };
27
28 using pt = point<ll>;

```

Capítulo 7

String

7.1 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.

Código: aho_corasick.cpp

```
1 namespace aho {
2     const int M = 3e5 + 1;
3     const int K = 26;
4
5     const char norm = 'a';
6     inline int get(int c) { return c - norm; }
7
8     int next[M][K], link[M], out_link[M], par[M], cur = 2;
```

```
9     char pch[M];
10    bool out[M];
11    vector<int> output[M];
12
13    int node(int p, char c) {
14        link[cur] = out_link[cur] = 0;
15        par[cur] = p;
16        pch[cur] = c;
17        return cur++;
18    }
19
20    int T = 0;
21
22    int insert(const string &s) {
23        int u = 1;
24        for (int i = 0; i < (int)s.size(); i++) {
25            auto v = next[u][get(s[i])];
26            if (v == 0) next[u][get(s[i])] = v = node(u, s[i]);
27            u = v;
28        }
29        out[u] = true;
30        output[u].emplace_back(T);
31        return T++;
32    }
33
34    int go(int u, char c);
35
36    int get_link(int u) {
37        if (link[u] == 0) link[u] = par[u] > 1 ? go(get_link(par[u]), pch[u]) : 1;
38        return link[u];
```

```

39     }
40
41     int go(int u, char c) {
42         if (next[u][get(c)] == 0) next[u][get(c)] = u > 1 ? go(get_link(u), c) : 1;
43         return next[u][get(c)];
44     }
45
46     int exit(int u) {
47         if (out_link[u] == 0) {
48             int v = get_link(u);
49             out_link[u] = (out[v] || v == 1) ? v : exit(v);
50         }
51         return out_link[u];
52     }
53
54     bool matched(int u) { return out[u] || exit(u) > 1; }
55
56 }

```

7.2 EertreE

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

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

Código: eertree.cpp

```

1 const int N = 2e5 + 15;
2 const int ALF = 26;
3
4 struct eertree {
5     int str[N], len[N], lnk[N], cnt[N], first[N], node_cnt, it, last;
6     ll palindrome_substring_sum;
7     const char norm = 'a';
8     array<int, ALF> to[N];
9
10    inline int get(char c) { return c - norm; }
11
12    void set_string(const string &s) {
13        int n = (int)s.size();
14        memset(str, 0, sizeof(int) * (it + 1));
15        memset(len, 0, sizeof(int) * (it + 1));
16        memset(lnk, 0, sizeof(int) * (it + 1));
17        memset(cnt, 0, sizeof(int) * (it + 1));
18        for (int i = 0; i <= it; i++)
19            for (int j = 0; j < ALF; j++) to[i][j] = 0;
20        node_cnt = 2, it = 1, last = 0, str[0] = -1;
21        len[0] = 0, len[1] = -1, lnk[0] = 1, lnk[1] = 1;
22        for (int i = 0; i < n; i++) insert(s[i]);
23        build_cnt();
24    }
25
26    void insert(char ch) {
27        int c = get(ch);
28        str[it] = c;
29        while (str[it - 1 - len[last]] != c) last = lnk[last];
30        if (!to[last][c]) {
31            int prev = lnk[last];
32            while (str[it - 1 - len[prev]] != c) prev = lnk[prev];
33            lnk[node_cnt] = to[prev][c];
34            len[node_cnt] = len[last] + 2;
35            to[last][c] = node_cnt++;
36        }
37        last = to[last][c];
38        first[last] = it;
39        cnt[last]++;
40        it++;
41    }
42
43    void build_cnt() {
44        ll ans = 0;

```

```

45     for (int i = it; i > 1; i--) {
46         ans += cnt[i];
47         cnt[lnk[i]] += cnt[i];
48     }
49     palindrome_substring_sum = ans;
50 }
51
52 inline ll number_of_palindromes() { return palindrome_substring_sum; }
53 inline int number_of_distinct_palindromes() { return node_cnt - 2; }
54 } et;

```

7.3 Hashing

7.3.1 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:

```

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

```

Código: hashing.cpp

```

1 const int MOD1 = 998244353;
2 const int MOD2 = (int)(1e9) + 7;
3 using mint1 = Mint<MOD1>;
4 using mint2 = Mint<MOD2>;
5
6 struct Hash {
7     mint1 h1;
8     mint2 h2;
9     Hash(mint1 _h1 = 0, mint2 _h2 = 0) : h1(_h1), h2(_h2) { }
10    bool operator==(Hash o) const { return h1 == o.h1 && h2 == o.h2; }
11    bool operator!=(Hash o) const { return h1 != o.h1 || h2 != o.h2; }
12    bool operator<(Hash o) const { return h1 == o.h1 ? h2 < o.h2 : h1 < o.h1; }
13    Hash operator+(Hash o) const { return {h1 + o.h1, h2 + o.h2}; }
14    Hash operator-(Hash o) const { return {h1 - o.h1, h2 - o.h2}; }
15    Hash operator*(Hash o) const { return {h1 * o.h1, h2 * o.h2}; }
16    Hash operator/(Hash o) const { return {h1 / o.h1, h2 / o.h2}; }
17 };
18
19 const int PRIME = 33333331; // qualquer primo na ordem do alfabeto
20 const int MAXN = 1e6 + 5;
21 Hash PR = {PRIME, PRIME};
22 Hash invPR = {mint1(1) / PRIME, mint2(1) / PRIME};
23 Hash pot[MAXN], invpot[MAXN];
24
25 void precalc() {
26     pot[0] = invpot[0] = Hash(1, 1);
27     for (int i = 1; i < MAXN; i++) {
28         pot[i] = pot[i - 1] * PR;
29         invpot[i] = invpot[i - 1] * invPR;
30     }
31 }
32
33 struct Hashing {
34     int N;
35     vector<Hash> hsh;
36     Hashing() { }
37     Hashing(string s) : N((int)s.size()), hsh(N + 1) {
38         for (int i = 0; i < N; i++) {
39             int c = (int)s[i];
40             hsh[i + 1] = hsh[i] + (pot[i + 1] * Hash(c, c));
41         }
42     }
43     Hash operator()(int l = 0, int r = -1) const {
44 #warning Chamou o precalc()?
45         // se ja chamou o precalc() pode apagar essa linha de cima
46         if (r == -1) r = N - 1;
47         return (hsh[r + 1] - hsh[l]) * invpot[l];

```

```

48 }
49 };

```

7.3.2 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:

```

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

```

Código: `dynamic_hashing.cpp`

```

1 const int MOD1 = 998244353;
2 const int MOD2 = 1e9 + 7;
3 using mint1 = Mint<MOD1>;
4 using mint2 = Mint<MOD2>;
5
6 struct Hash {
7     mint1 h1;
8     mint2 h2;
9     Hash() {}
10    Hash(mint1 _h1, mint2 _h2) : h1(_h1), h2(_h2) {}
11    bool operator==(Hash o) const { return h1 == o.h1 && h2 == o.h2; }
12    bool operator!=(Hash o) const { return h1 != o.h1 || h2 != o.h2; }
13    bool operator<(Hash o) const { return h1 == o.h1 ? h2 < o.h2 : h1 < o.h1; }
14    Hash operator+(Hash o) const { return {h1 + o.h1, h2 + o.h2}; }

```

```

15    Hash operator-(Hash o) const { return {h1 - o.h1, h2 - o.h2}; }
16    Hash operator*(Hash o) const { return {h1 * o.h1, h2 * o.h2}; }
17    Hash operator/(Hash o) const { return {h1 / o.h1, h2 / o.h2}; }
18 };
19
20 const int PRIME = 1001003; // qualquer primo na ordem do alfabeto
21 const int MAXN = 1e6 + 5;
22 Hash PR = {PRIME, PRIME};
23 Hash invPR = {mint1(1) / PRIME, mint2(1) / PRIME};
24 Hash pot[MAXN], invpot[MAXN];
25 void precalc() {
26     pot[0] = invpot[0] = Hash(1, 1);
27     for (int i = 1; i < MAXN; i++) {
28         pot[i] = pot[i - 1] * PR;
29         invpot[i] = invpot[i - 1] * invPR;
30     }
31 }
32
33 struct DynamicHashing {
34     int N;
35     FenwickTree<Hash> hsh;
36     DynamicHashing() {}
37     DynamicHashing(string s) : N(int(s.size())) {
38         vector<Hash> v(N);
39         for (int i = 0; i < N; i++) {
40             int c = (int)s[i];
41             v[i] = pot[i + 1] * Hash(c, c);
42         }
43         hsh = FenwickTree<Hash>(v);
44     }
45     Hash operator()(int l, int r) { return hsh.query(l, r) * invpot[l]; }
46     void update(int i, char ch) {
47         int c = (int)ch;
48         hsh.updateSet(i, pot[i + 1] * Hash(c, c));
49     }
50 };

```

7.4 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 cílicos dos caracteres.

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

Código: min_cyclic_shift.cpp

```

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

Código: duval.cpp

```

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

```

16     }
17     return factorization;
18 }
```

7.5 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 d_1 e d_2 respectivamente). O método `solve` computa os palíndromos e retorna o número de substrings palíndromas. O método `query` retorna se a substring $s[i \dots j]$ é palíndroma em $\mathcal{O}(1)$.

Código: manacher.cpp

```

1 struct Manacher {
2     int n;
3     ll count;
4     vector<int> d1, d2, man;
5     ll solve(const string &s) {
6         n = int(s.size()), count = 0;
7         solve_odd(s);
8         solve_even(s);
9         man.assign(2 * n - 1, 0);
10        for (int i = 0; i < n; i++) man[2 * i] = 2 * d1[i] - 1;
11        for (int i = 0; i < n - 1; i++) man[2 * i + 1] = 2 * d2[i + 1];
12        return count;
13    }
14    void solve_odd(const string &s) {
15        d1.assign(n, 0);
16        for (int i = 0, l = 0, r = -1; i < n; i++) {
17            int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
18            while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) k++;
19            count += d1[i] = k--;
20            if (i + k > r) l = i - k, r = i + k;
21        }
22    }
23    void solve_even(const string &s) {
24        d2.assign(n, 0);
25        for (int i = 0, l = 0, r = -1; i < n; i++) {
26            int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
27            while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) k++;
28            count += d2[i] = k--;
29            if (i + k > r) l = i - k - 1, r = i + k;
30        }
31    }
32 }
```

```

30     }
31 }
32 bool query(int i, int j) {
33     assert(man.size());
34     return man[i + j] >= j - i + 1;
35 }
36 } mana;

```

7.6 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:

```

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

```

Codigo: patricia_tree.cpp

```

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

```

7.7 Prefix Function KMP

7.7.1 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 múltiplos pattern matchings, ou seja, dado um padrão t , encontrar todas as ocorrências de t em várias strings s_1, s_2, \dots, s_k , em $\mathcal{O}(|t| + \sum |s_i|)$. O método matching faz isso.

Obs: utiliza o código do KMP.

Codigo: aut_kmp.cpp

```

1 struct AutKMP {
2     vector<vector<int>> nxt;
3     void setString(string s) {
4         s += '#';
5         nxt.assign(s.size(), vector<int>(26));
6         vector<int> p = pi(s);
7         for (int c = 0; c < 26; c++) nxt[0][c] = ('a' + c == s[0]);
8         for (int i = 1; i < int(s.size()); i++)
9             for (int c = 0; c < 26; c++)
10                 nxt[i][c] = ('a' + c == s[i]) ? i + 1 : nxt[p[i - 1]][c];
11     }
12     vector<int> matching(string &s, string &t) {
13         vector<int> match;
14         for (int i = 0, j = 0; i < int(s.size()); i++) {
15             j = nxt[j][s[i] - 'a'];
16             if (j == int(t.size())) match.push_back(i - j + 1);
17         }
18         return match;
19     }
20 } aut;

```

7.7.2 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 $\mathcal{O}(|s| + |t|)$.

Código: `kmp.cpp`

```

1 vector<int> pi(string &s) {
2     vector<int> p(s.size());
3     for (int i = 1, j = 0; i < int(s.size()); i++) {
4         while (j > 0 && s[i] != s[j]) j = p[j - 1];
5         if (s[i] == s[j]) j++;
6         p[i] = j;
7     }
8     return p;
9 }
10
11 vector<int> matching(string &s, string &t) { // s = texto, t = padrão
12     vector<int> p = pi(t), match;
13     for (int i = 0, j = 0; i < (int)s.size(); i++) {
14         while (j > 0 && s[i] != t[j]) j = p[j - 1];
15         if (s[i] == t[j]) j++;
16         if (j == (int)t.size()) {
17             match.push_back(i - j + 1);
18             j = p[j - 1];
19         }
20     }
21     return match;
22 }
```

7.8 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).

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

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

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

Código: `suffix_array_busca.cpp`

```

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

Código: `suffix_array.cpp`

```

1 const int MAX = 5e5 + 5;
2 struct suffix_array {
3     string s;
4     int n, sum, r, ra[MAX], sa[MAX], auxra[MAX], auxsa[MAX], c[MAX], lcp[MAX];
5     void counting_sort(int k) {
6         memset(c, 0, sizeof(c));
7         for (int i = 0; i < n; i++) c[(i + k < n) ? ra[i + k] : 0]++;
8         for (int i = sum = 0; i < max(256, n); i++) sum += c[i], c[i] = sum - c[i];
9         for (int i = 0; i < n; i++) auxsa[c[sa[i]] + k < n ? ra[sa[i] + k] : 0]++ =
10            sa[i];
11        for (int i = 0; i < n; i++) sa[i] = auxsa[i];
12    }
13    void build_sa() {
14        for (int k = 1; k < n; k <= 1) {
15            counting_sort(k);
16            counting_sort(0);
17            auxra[sa[0]] = r = 0;
18            for (int i = 1; i < n; i++)
19                if (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] == ra[sa[i - 1] + k])
20                    auxra[sa[i]] = r;
21                else auxra[sa[i]] = ++r;
22            for (int i = 0; i < n; i++) ra[i] = auxra[i];
23            if (ra[sa[n - 1]] == n - 1) break;
24        }
25    }
26    void build_lcp() {
27        for (int i = 0, k = 0; i < n - 1; i++) {
28            int j = sa[ra[i] - 1];
29            while (s[i + k] == s[j + k]) k++;
30            lcp[ra[i]] = k;
31            if (k) k--;
32        }
33    }
34    void set_string(string _s) {
35        s = _s + '$';
36        n = s.size();
37        for (int i = 0; i < n; i++) ra[i] = s[i], sa[i] = i;
38        build_sa();
39        build_lcp();
40        // for (int i = 0; i < n; i++)
41        // printf("%d: %s\n", sa[i], s.c_str() +
42        // sa[i]);
43    }
44    int operator[](int i) { return sa[i]; }
} sa;

```

7.9 Suffix Automaton

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

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

Por definição, $\text{len}[\text{lca}(u, v)]$ 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.

Código: suffix_automaton.cpp

```

1 const int N = 5e5 + 5;
2 const int S = 2 * N;
3
4 struct SuffixAutomaton {
5     array<int, 26> to[S];
6     int lnk[S], len[S], cnt[S], idx[S], where[S];
7     int last = 1, id = 2;
8     ll distinct_substrings = 0;
9
10    const char norm = 'a';
11    inline int get(char c) { return c - norm; }
12
13    void insert(int i, char ch) {
14        int cur = id++;
15        int c = get(ch);
16        len[cur] = len[last] + 1;
17        where[idx[cur]] = i = cur;

```

```

18     cnt[cur] = 1;
19     int p = last;
20     while (p > 0 && !to[p][c]) {
21         to[p][c] = cur;
22         p = lnk[p];
23     }
24     if (p == 0) {
25         lnk[cur] = 1;
26     } else {
27         int sp = to[p][c];
28         if (len[p] + 1 == len[sp]) {
29             lnk[cur] = sp;
30         } else {
31             int clone = id++;
32             len[clone] = len[p] + 1;
33             lnk[clone] = lnk[sp];
34             idx[clone] = idx[sp];
35             to[clone] = to[sp];
36             while (p > 0 && to[p][c] == sp) {
37                 to[p][c] = clone;
38                 p = lnk[p];
39             }
40             lnk[sp] = lnk[cur] = clone;
41         }
42     }
43     last = cur;
44 }
45
46 vector<int> adj[S];
47
48 void dfs(int u) {
49     for (int v : adj[u]) {
50         dfs(v);
51         cnt[u] += cnt[v];
52     }
53     distinct_substrings += len[u] - len[lnk[u]];
54 }
55
56 void set_string(const string &s) {
57     int n = (int)s.size();
58     for (int i = 0; i < id; i++) {
59         to[i].fill(0);
60         len[i] = lnk[i] = cnt[i] = 0;
61         adj[i].clear();
62     }
63     last = 1, id = 2, distinct_substrings = 0;
64     for (int i = 0; i < n; i++) insert(i, s[i]);

```

```

65         for (int i = 2; i < id; i++) adj[lnk[i]].push_back(i);
66         dfs(i);
67     }
68
69     int count_occurrences(const string &t) {
70         int cur = 1;
71         for (char ch : t) {
72             int c = get(ch);
73             if (!to[cur][c]) return 0;
74             cur = to[cur][c];
75         }
76         return cnt[cur];
77     }
78
79     int longest_common_substring(const string &t) {
80         int cur = 1, clen = 0, ans = 0;
81         for (char ch : t) {
82             int c = get(ch);
83             while (cur > 0 && !to[cur][c]) {
84                 cur = lnk[cur];
85                 clen = len[cur];
86             }
87             if (to[cur][c]) {
88                 cur = to[cur][c];
89                 clen++;
90             }
91             ans = max(ans, clen);
92             cur = max(cur, 1);
93         }
94         return ans;
95     }
96
97     int lcs(int i, int j) {
98         // retorna o maior sufixo comum entre os prefixos s[0..i] e s[0..j]
99         // tem que codar o lca aqui no automato
100         return len[lca(where[i], where[j])];
101     }
102 } sam;

```

7.10 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 \bar{S} , onde \bar{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.

Código: `suffix_tree.cpp`

```

1 const int N = 5e5 + 5;
2 const int S = 2 * N;
3
4 struct SuffixTree {
5     array<int, 26> to[S];
6     int lnk[S], len[S], cnt[S], idx[S], where[S], suff[S];
7     int last = 1, id = 2;
8     ll distinct_substrings = 0;
9     string s;
10
11    const char norm = 'a';
12    inline int get(char c) { return c - norm; }
13

```

```

14    void insert(int i, char ch) {
15        int cur = id++;
16        int c = get(ch);
17        len[cur] = len[last] + 1;
18        where[idx[cur] = i] = cur;
19        cnt[cur] = suff[cur] = 1;
20        int p = last;
21        while (p > 0 && !to[p][c]) {
22            to[p][c] = cur;
23            p = lnk[p];
24        }
25        if (p == 0) {
26            lnk[cur] = 1;
27        } else {
28            int sp = to[p][c];
29            if (len[p] + 1 == len[sp]) {
30                lnk[cur] = sp;
31            } else {
32                int clone = id++;
33                len[clone] = len[p] + 1;
34                lnk[clone] = lnk[sp];
35                idx[clone] = idx[sp];
36                to[clone] = to[sp];
37                while (p > 0 && to[p][c] == sp) {
38                    to[p][c] = clone;
39                    p = lnk[p];
40                }
41                lnk[sp] = lnk[cur] = clone;
42            }
43        }
44        last = cur;
45    }
46
47    vector<int> adj[S];
48
49    void dfs(int u) {
50        sort(adj[u].begin(), adj[u].end(), [&](int v1, int v2) {
51            // sorteia os filhos de u por ordem lexicografica,
52            // isso faz com que a ordem de dfs seja a ordem
53            // lexicografica dos sufixos (e de qualquer substring na verdade)
54            return s[idx[v1] + len[u] < s[idx[v2] + len[u]];
55        });
56        for (int v : adj[u]) {
57            dfs(v);
58            cnt[u] += cnt[v];
59        }
60        distinct_substrings += len[u] - len[lnk[u]];
61    }
62
63    int lca(int u, int v) {
64        if (u == v) return u;
65        if (lnk[u] == lnk[v]) return lnk[u];
66        if (idx[u] < idx[v]) swap(u, v);
67        return lca(lnk[u], v);
68    }
69
70    int lcp(int u, int v) {
71        if (u == v) return len[u];
72        if (lnk[u] == lnk[v]) return len[lnk[u]];
73        if (idx[u] < idx[v]) swap(u, v);
74        int lca_u_v = lca(lnk[u], v);
75        return len[u] - len[lca_u_v];
76    }
77
78    int suffix_array() {
79        sort(idx.begin(), idx.end());
80        for (int i = 0; i < S; i++) {
81            if (suff[i] == 0) {
82                suff[i] = id++;
83            }
84        }
85        return distinct_substrings;
86    }
87
88    int count_substrings() {
89        return distinct_substrings;
90    }
91
92    int count_distinct_substrings() {
93        return distinct_substrings;
94    }
95
96    int count_leaves() {
97        return count_substrings();
98    }
99
100   int count_leaves() {
101       return count_substrings();
102   }
103
104   int count_leaves() {
105       return count_substrings();
106   }
107
108   int count_leaves() {
109       return count_substrings();
110   }
111
112   int count_leaves() {
113       return count_substrings();
114   }
115
116   int count_leaves() {
117       return count_substrings();
118   }
119
120   int count_leaves() {
121       return count_substrings();
122   }
123
124   int count_leaves() {
125       return count_substrings();
126   }
127
128   int count_leaves() {
129       return count_substrings();
130   }
131
132   int count_leaves() {
133       return count_substrings();
134   }
135
136   int count_leaves() {
137       return count_substrings();
138   }
139
140   int count_leaves() {
141       return count_substrings();
142   }
143
144   int count_leaves() {
145       return count_substrings();
146   }
147
148   int count_leaves() {
149       return count_substrings();
150   }
151
152   int count_leaves() {
153       return count_substrings();
154   }
155
156   int count_leaves() {
157       return count_substrings();
158   }
159
160   int count_leaves() {
161       return count_substrings();
162   }
163
164   int count_leaves() {
165       return count_substrings();
166   }
167
168   int count_leaves() {
169       return count_substrings();
170   }
171
172   int count_leaves() {
173       return count_substrings();
174   }
175
176   int count_leaves() {
177       return count_substrings();
178   }
179
180   int count_leaves() {
181       return count_substrings();
182   }
183
184   int count_leaves() {
185       return count_substrings();
186   }
187
188   int count_leaves() {
189       return count_substrings();
190   }
191
192   int count_leaves() {
193       return count_substrings();
194   }
195
196   int count_leaves() {
197       return count_substrings();
198   }
199
200   int count_leaves() {
201       return count_substrings();
202   }
203
204   int count_leaves() {
205       return count_substrings();
206   }
207
208   int count_leaves() {
209       return count_substrings();
210   }
211
212   int count_leaves() {
213       return count_substrings();
214   }
215
216   int count_leaves() {
217       return count_substrings();
218   }
219
220   int count_leaves() {
221       return count_substrings();
222   }
223
224   int count_leaves() {
225       return count_substrings();
226   }
227
228   int count_leaves() {
229       return count_substrings();
230   }
231
232   int count_leaves() {
233       return count_substrings();
234   }
235
236   int count_leaves() {
237       return count_substrings();
238   }
239
240   int count_leaves() {
241       return count_substrings();
242   }
243
244   int count_leaves() {
245       return count_substrings();
246   }
247
248   int count_leaves() {
249       return count_substrings();
250   }
251
252   int count_leaves() {
253       return count_substrings();
254   }
255
256   int count_leaves() {
257       return count_substrings();
258   }
259
260   int count_leaves() {
261       return count_substrings();
262   }
263
264   int count_leaves() {
265       return count_substrings();
266   }
267
268   int count_leaves() {
269       return count_substrings();
270   }
271
272   int count_leaves() {
273       return count_substrings();
274   }
275
276   int count_leaves() {
277       return count_substrings();
278   }
279
280   int count_leaves() {
281       return count_substrings();
282   }
283
284   int count_leaves() {
285       return count_substrings();
286   }
287
288   int count_leaves() {
289       return count_substrings();
290   }
291
292   int count_leaves() {
293       return count_substrings();
294   }
295
296   int count_leaves() {
297       return count_substrings();
298   }
299
300   int count_leaves() {
301       return count_substrings();
302   }
303
304   int count_leaves() {
305       return count_substrings();
306   }
307
308   int count_leaves() {
309       return count_substrings();
310   }
311
312   int count_leaves() {
313       return count_substrings();
314   }
315
316   int count_leaves() {
317       return count_substrings();
318   }
319
320   int count_leaves() {
321       return count_substrings();
322   }
323
324   int count_leaves() {
325       return count_substrings();
326   }
327
328   int count_leaves() {
329       return count_substrings();
330   }
331
332   int count_leaves() {
333       return count_substrings();
334   }
335
336   int count_leaves() {
337       return count_substrings();
338   }
339
340   int count_leaves() {
341       return count_substrings();
342   }
343
344   int count_leaves() {
345       return count_substrings();
346   }
347
348   int count_leaves() {
349       return count_substrings();
350   }
351
352   int count_leaves() {
353       return count_substrings();
354   }
355
356   int count_leaves() {
357       return count_substrings();
358   }
359
360   int count_leaves() {
361       return count_substrings();
362   }
363
364   int count_leaves() {
365       return count_substrings();
366   }
367
368   int count_leaves() {
369       return count_substrings();
370   }
371
372   int count_leaves() {
373       return count_substrings();
374   }
375
376   int count_leaves() {
377       return count_substrings();
378   }
379
380   int count_leaves() {
381       return count_substrings();
382   }
383
384   int count_leaves() {
385       return count_substrings();
386   }
387
388   int count_leaves() {
389       return count_substrings();
390   }
391
392   int count_leaves() {
393       return count_substrings();
394   }
395
396   int count_leaves() {
397       return count_substrings();
398   }
399
400   int count_leaves() {
401       return count_substrings();
402   }
403
404   int count_leaves() {
405       return count_substrings();
406   }
407
408   int count_leaves() {
409       return count_substrings();
410   }
411
412   int count_leaves() {
413       return count_substrings();
414   }
415
416   int count_leaves() {
417       return count_substrings();
418   }
419
420   int count_leaves() {
421       return count_substrings();
422   }
423
424   int count_leaves() {
425       return count_substrings();
426   }
427
428   int count_leaves() {
429       return count_substrings();
430   }
431
432   int count_leaves() {
433       return count_substrings();
434   }
435
436   int count_leaves() {
437       return count_substrings();
438   }
439
440   int count_leaves() {
441       return count_substrings();
442   }
443
444   int count_leaves() {
445       return count_substrings();
446   }
447
448   int count_leaves() {
449       return count_substrings();
450   }
451
452   int count_leaves() {
453       return count_substrings();
454   }
455
456   int count_leaves() {
457       return count_substrings();
458   }
459
460   int count_leaves() {
461       return count_substrings();
462   }
463
464   int count_leaves() {
465       return count_substrings();
466   }
467
468   int count_leaves() {
469       return count_substrings();
470   }
471
472   int count_leaves() {
473       return count_substrings();
474   }
475
476   int count_leaves() {
477       return count_substrings();
478   }
479
480   int count_leaves() {
481       return count_substrings();
482   }
483
484   int count_leaves() {
485       return count_substrings();
486   }
487
488   int count_leaves() {
489       return count_substrings();
490   }
491
492   int count_leaves() {
493       return count_substrings();
494   }
495
496   int count_leaves() {
497       return count_substrings();
498   }
499
500   int count_leaves() {
501       return count_substrings();
502   }
503
504   int count_leaves() {
505       return count_substrings();
506   }
507
508   int count_leaves() {
509       return count_substrings();
510   }
511
512   int count_leaves() {
513       return count_substrings();
514   }
515
516   int count_leaves() {
517       return count_substrings();
518   }
519
520   int count_leaves() {
521       return count_substrings();
522   }
523
524   int count_leaves() {
525       return count_substrings();
526   }
527
528   int count_leaves() {
529       return count_substrings();
530   }
531
532   int count_leaves() {
533       return count_substrings();
534   }
535
536   int count_leaves() {
537       return count_substrings();
538   }
539
540   int count_leaves() {
541       return count_substrings();
542   }
543
544   int count_leaves() {
545       return count_substrings();
546   }
547
548   int count_leaves() {
549       return count_substrings();
550   }
551
552   int count_leaves() {
553       return count_substrings();
554   }
555
556   int count_leaves() {
557       return count_substrings();
558   }
559
560   int count_leaves() {
561       return count_substrings();
562   }
563
564   int count_leaves() {
565       return count_substrings();
566   }
567
568   int count_leaves() {
569       return count_substrings();
570   }
571
572   int count_leaves() {
573       return count_substrings();
574   }
575
576   int count_leaves() {
577       return count_substrings();
578   }
579
580   int count_leaves() {
581       return count_substrings();
582   }
583
584   int count_leaves() {
585       return count_substrings();
586   }
587
588   int count_leaves() {
589       return count_substrings();
590   }
591
592   int count_leaves() {
593       return count_substrings();
594   }
595
596   int count_leaves() {
597       return count_substrings();
598   }
599
600   int count_leaves() {
601       return count_substrings();
602   }
603
604   int count_leaves() {
605       return count_substrings();
606   }
607
608   int count_leaves() {
609       return count_substrings();
610   }
611
612   int count_leaves() {
613       return count_substrings();
614   }
615
616   int count_leaves() {
617       return count_substrings();
618   }
619
620   int count_leaves() {
621       return count_substrings();
622   }
623
624   int count_leaves() {
625       return count_substrings();
626   }
627
628   int count_leaves() {
629       return count_substrings();
630   }
631
632   int count_leaves() {
633       return count_substrings();
634   }
635
636   int count_leaves() {
637       return count_substrings();
638   }
639
640   int count_leaves() {
641       return count_substrings();
642   }
643
644   int count_leaves() {
645       return count_substrings();
646   }
647
648   int count_leaves() {
649       return count_substrings();
650   }
651
652   int count_leaves() {
653       return count_substrings();
654   }
655
656   int count_leaves() {
657       return count_substrings();
658   }
659
660   int count_leaves() {
661       return count_substrings();
662   }
663
664   int count_leaves() {
665       return count_substrings();
666   }
667
668   int count_leaves() {
669       return count_substrings();
670   }
671
672   int count_leaves() {
673       return count_substrings();
674   }
675
676   int count_leaves() {
677       return count_substrings();
678   }
679
680   int count_leaves() {
681       return count_substrings();
682   }
683
684   int count_leaves() {
685       return count_substrings();
686   }
687
688   int count_leaves() {
689       return count_substrings();
690   }
691
692   int count_leaves() {
693       return count_substrings();
694   }
695
696   int count_leaves() {
697       return count_substrings();
698   }
699
700   int count_leaves() {
701       return count_substrings();
702   }
703
704   int count_leaves() {
705       return count_substrings();
706   }
707
708   int count_leaves() {
709       return count_substrings();
710   }
711
712   int count_leaves() {
713       return count_substrings();
714   }
715
716   int count_leaves() {
717       return count_substrings();
718   }
719
720   int count_leaves() {
721       return count_substrings();
722   }
723
724   int count_leaves() {
725       return count_substrings();
726   }
727
728   int count_leaves() {
729       return count_substrings();
730   }
731
732   int count_leaves() {
733       return count_substrings();
734   }
735
736   int count_leaves() {
737       return count_substrings();
738   }
739
740   int count_leaves() {
741       return count_substrings();
742   }
743
744   int count_leaves() {
745       return count_substrings();
746   }
747
748   int count_leaves() {
749       return count_substrings();
750   }
751
752   int count_leaves() {
753       return count_substrings();
754   }
755
756   int count_leaves() {
757       return count_substrings();
758   }
759
760   int count_leaves() {
761       return count_substrings();
762   }
763
764   int count_leaves() {
765       return count_substrings();
766   }
767
768   int count_leaves() {
769       return count_substrings();
770   }
771
772   int count_leaves() {
773       return count_substrings();
774   }
775
776   int count_leaves() {
777       return count_substrings();
778   }
779
780   int count_leaves() {
781       return count_substrings();
782   }
783
784   int count_leaves() {
785       return count_substrings();
786   }
787
788   int count_leaves() {
789       return count_substrings();
790   }
791
792   int count_leaves() {
793       return count_substrings();
794   }
795
796   int count_leaves() {
797       return count_substrings();
798   }
799
800   int count_leaves() {
801       return count_substrings();
802   }
803
804   int count_leaves() {
805       return count_substrings();
806   }
807
808   int count_leaves() {
809       return count_substrings();
810   }
811
812   int count_leaves() {
813       return count_substrings();
814   }
815
816   int count_leaves() {
817       return count_substrings();
818   }
819
820   int count_leaves() {
821       return count_substrings();
822   }
823
824   int count_leaves() {
825       return count_substrings();
826   }
827
828   int count_leaves() {
829       return count_substrings();
830   }
831
832   int count_leaves() {
833       return count_substrings();
834   }
835
836   int count_leaves() {
837       return count_substrings();
838   }
839
840   int count_leaves() {
841       return count_substrings();
842   }
843
844   int count_leaves() {
845       return count_substrings();
846   }
847
848   int count_leaves() {
849       return count_substrings();
850   }
851
852   int count_leaves() {
853       return count_substrings();
854   }
855
856   int count_leaves() {
857       return count_substrings();
858   }
859
860   int count_leaves() {
861       return count_substrings();
862   }
863
864   int count_leaves() {
865       return count_substrings();
866   }
867
868   int count_leaves() {
869       return count_substrings();
870   }
871
872   int count_leaves() {
873       return count_substrings();
874   }
875
876   int count_leaves() {
877       return count_substrings();
878   }
879
880   int count_leaves() {
881       return count_substrings();
882   }
883
884   int count_leaves() {
885       return count_substrings();
886   }
887
888   int count_leaves() {
889       return count_substrings();
890   }
891
892   int count_leaves() {
893       return count_substrings();
894   }
895
896   int count_leaves() {
897       return count_substrings();
898   }
899
900   int count_leaves() {
901       return count_substrings();
902   }
903
904   int count_leaves() {
905       return count_substrings();
906   }
907
908   int count_leaves() {
909       return count_substrings();
910   }
911
912   int count_leaves() {
913       return count_substrings();
914   }
915
916   int count_leaves() {
917       return count_substrings();
918   }
919
920   int count_leaves() {
921       return count_substrings();
922   }
923
924   int count_leaves() {
925       return count_substrings();
926   }
927
928   int count_leaves() {
929       return count_substrings();
930   }
931
932   int count_leaves() {
933       return count_substrings();
934   }
935
936   int count_leaves() {
937       return count_substrings();
938   }
939
940   int count_leaves() {
941       return count_substrings();
942   }
943
944   int count_leaves() {
945       return count_substrings();
946   }
947
948   int count_leaves() {
949       return count_substrings();
950   }
951
952   int count_leaves() {
953       return count_substrings();
954   }
955
956   int count_leaves() {
957       return count_substrings();
958   }
959
960   int count_leaves() {
961       return count_substrings();
962   }
963
964   int count_leaves() {
965       return count_substrings();
966   }
967
968   int count_leaves() {
969       return count_substrings();
970   }
971
972   int count_leaves() {
973       return count_substrings();
974   }
975
976   int count_leaves() {
977       return count_substrings();
978   }
979
980   int count_leaves() {
981       return count_substrings();
982   }
983
984   int count_leaves() {
985       return count_substrings();
986   }
987
988   int count_leaves() {
989       return count_substrings();
990   }
991
992   int count_leaves() {
993       return count_substrings();
994   }
995
996   int count_leaves() {
997       return count_substrings();
998   }
999
1000  int count_leaves() {
1001      return count_substrings();
1002  }
1003
1004  int count_leaves() {
1005      return count_substrings();
1006  }
1007
1008  int count_leaves() {
1009      return count_substrings();
1010  }
1011
1012  int count_leaves() {
1013      return count_substrings();
1014  }
1015
1016  int count_leaves() {
1017      return count_substrings();
1018  }
1019
1020  int count_leaves() {
1021      return count_substrings();
1022  }
1023
1024  int count_leaves() {
1025      return count_substrings();
1026  }
1027
1028  int count_leaves() {
1029      return count_substrings();
1030  }
1031
1032  int count_leaves() {
1033      return count_substrings();
1034  }
1035
1036  int count_leaves() {
1037      return count_substrings();
1038  }
1039
1040  int count_leaves() {
1041      return count_substrings();
1042  }
1043
1044  int count_leaves() {
1045      return count_substrings();
1046  }
1047
1048  int count_leaves() {
1049      return count_substrings();
1050  }
1051
1052  int count_leaves() {
1053      return count_substrings();
1054  }
1055
1056  int count_leaves() {
1057      return count_substrings();
1058  }
1059
1060  int count_leaves() {
1061      return count_substrings();
1062  }
1063
1064  int count_leaves() {
1065      return count_substrings();
1066  }
1067
1068  int count_leaves() {
1069      return count_substrings();
1070  }
1071
1072  int count_leaves() {
1073      return count_substrings();
1074  }
1075
1076  int count_leaves() {
1077      return count_substrings();
1078  }
1079
1080  int count_leaves() {
1081      return count_substrings();
1082  }
1083
1084  int count_leaves() {
1085      return count_substrings();
1086  }
1087
1088  int count_leaves() {
1089      return count_substrings();
1090  }
1091
1092  int count_leaves() {
1093      return count_substrings();
1094  }
1095
1096  int count_leaves() {
1097      return count_substrings();
1098  }
1099
1100  int count_leaves() {
1101      return count_substrings();
1102  }
1103
1104  int count_leaves() {
1105      return count_substrings();
1106  }
1107
1108  int count_leaves() {
1109      return count_substrings();
1110  }
1111
1112  int count_leaves() {
1113      return count_substrings();
1114  }
1115
1116  int count_leaves() {
1117      return count_substrings();
1118  }
1119
1120  int count_leaves() {
1121      return count_substrings();
1122  }
1123
1124  int count_leaves() {
1125      return count_substrings();
1126  }
1127
1128  int count_leaves() {
1129      return count_substrings();
1130  }
1131
1132  int count_leaves() {
1133      return count_substrings();
1134  }
1135
1136  int count_leaves() {
1137      return count_substrings();
1138  }
1139
1140  int count_leaves() {
1141      return count_substrings();
1142  }
1143
1144  int count_leaves() {
1145      return count_substrings();
1146  }
1147
1148  int count_leaves() {
1149      return count_substrings();
1150  }
1151
1152  int count_leaves() {
1153      return count_substrings();
1154  }
1155
1156  int count_leaves() {
1157      return count_substrings();
1158  }
1159
1160  int count_leaves() {
1161      return count_substrings();
1162  }
1163
1164  int count_leaves() {
1165      return count_substrings();
1166  }
1167
1168  int count_leaves() {
1169      return count_substrings();
1170  }
1171
1172  int count_leaves() {
1173      return count_substrings();
1174  }
1175
1176  int count_leaves() {
1177      return count_substrings();
1178  }
1179
1180  int count_leaves() {
1181      return count_substrings();
1182  }
1183
1184  int count_leaves() {
1185      return count_substrings();
1186  }
1187
1188  int count_leaves() {
1189      return count_substrings();
1190  }
1191
1192  int count_leaves() {
1193      return count_substrings();
1194  }
1195
1196  int count_leaves() {
1197      return count_substrings();
1198  }
1199
1200  int count_leaves() {
1201      return count_substrings();
1202  }
1203
1204  int count_leaves() {
1205      return count_substrings();
1206  }
1207
1208  int count_leaves() {
1209      return count_substrings();
1210  }
1211
1212  int count_leaves() {
1213      return count_substrings();
1214  }
1215
1216  int count_leaves() {
1217      return count_substrings();
1218  }
1219
1220  int count_leaves() {
1221      return count_substrings();
1222  }
1223
1224  int count_leaves() {
1225      return count_substrings();
1226  }
1227
1228  int count_leaves() {
1229      return count_substrings();
1230  }
1231
1232  int count_leaves() {
1233      return count_substrings();
1234  }
1235
1236  int count_leaves() {
1237      return count_substrings();
1238  }
1239
1240  int count_leaves() {
1241      return count_substrings();
1242  }
1243
1244  int count_leaves() {
1245      return count_substrings();
1246  }
1247
1248  int count_leaves() {
1249      return count_substrings();
1250  }
1251
1252  int count_leaves() {
1253      return count_substrings();
1254  }
1255
1256  int count_leaves() {
1257      return count_substrings();
1258  }
1259
1260  int count_leaves() {
1261      return count_substrings();
1262  }
1263
1264  int count_leaves() {
1265      return count_substrings();
1266  }
1267
1268  int count_leaves() {
1269      return count_substrings();
1270  }
1271
1272  int count_leaves() {
1273      return count_substrings();
1274  }
1275
1276  int count_leaves() {
1277      return count_substrings();
1278  }
1279
1280  int count_leaves() {
1281      return count_substrings();
1282  }
1283
1284  int count_leaves() {
1285      return count_substrings();
1286  }
1287
1288  int count_leaves() {
1289      return count_substrings();
1290  }
1291
1292  int count_leaves() {
1293      return count_substrings();
1294  }
1295
1296  int count_leaves() {
1297      return count_substrings();
1298  }
1299
1300  int count_leaves() {
1301      return count_substrings();
1302  }
1303
1304  int count_leaves() {
1305      return count_substrings();
1306  }
1307
1308  int count_leaves() {
1309      return count_substrings();
1310  }
1311
1312  int count_leaves() {
1313      return count_substrings();
1314  }
1315
1316  int count_leaves() {
1317      return count_substrings();
1318  }
1319
1320  int count_leaves() {
1321      return count_substrings();
1322  }
1323
1324  int count_leaves() {
1325      return count_substrings();
1326  }
1327
1328  int count_leaves() {
1329      return count_substrings();
1330  }
1331
1332  int count_leaves() {
1333      return count_substrings();
1334  }
1335
1336  int count_leaves() {
1337      return count_substrings();
1338  }
1339
1340  int count_leaves() {
1341      return count_substrings();
1342  }
1343
1344  int count_leaves() {
1345      return count_substrings();
1346  }
1347
1348  int count_leaves() {
1349      return count_substrings();
1350  }
1351
1352  int count_leaves() {
1353      return count_substrings();
1354  }
1355
1356  int count_leaves() {
1357      return count_substrings();
1358  }
1359
1360  int count_leaves() {
1361      return count_substrings();
1362  }
1363
1364  int count_leaves() {
1365      return count_substrings();
1366  }
136
```

```

61     }
62
63 void set_string(const string &s2) {
64     s = s2;
65     int n = (int)s.size();
66     for (int i = 0; i < id; i++) {
67         to[i].fill(0);
68         len[i] = lnk[i] = cnt[i] = 0;
69     }
70     id = 2;
71     for (int i = n - 1; i >= 0; i--) insert(i, s[i]);
72     for (int i = 2; i < id; i++) adj[lnk[i]].push_back(i);
73     dfs(1);
74 }
75
76 int count_occurrences(const string &t) {
77     int cur = 1, m = (int)t.size();
78     for (int i = m - 1; i >= 0; i--) {
79         int c = get(t[i]);
80         if (!to[cur][c]) return 0;
81         cur = to[cur][c];
82     }
83     return cnt[cur];
84 }
85
86 int longest_common_substring(const string &t) {
87     int cur = 1, clen = 0, ans = 0, m = (int)t.size();
88     for (int i = m - 1; i >= 0; i--) {
89         int c = get(t[i]);
90         while (cur > 0 && !to[cur][c]) {
91             cur = lnk[cur];
92             clen = len[cur];
93         }
94         if (to[cur][c]) {
95             cur = to[cur][c];
96             clen++;
97         }
98         ans = max(ans, clen);
99         cur = max(cur, 1);
100    }
101    return ans;
102 }
103
104 string kth_substring(ll k) {
105     // esse metodo retorna a k-esima menor substring lexicograficamente,
106     // dentre todas as substrings distintas
107     string res = "";

```

```

108     function<bool(int)> dfs_kth = [&](int u) {
109         int min_len = len[lnk[u]] + 1, max_len = len[u];
110         int qnt = (max_len - min_len + 1);
111         if (qnt < k) {
112             k -= qnt;
113         } else {
114             res = s.substr(idx[u], min_len + k - 1);
115             return true;
116         }
117         for (int v : adj[u])
118             if (dfs_kth(v)) return true;
119         return false;
120     };
121     dfs_kth(1);
122     return res;
123 }
124
125 string kth_substring2(ll k) {
126     // esse metodo retorna a k-esima menor substring lexicograficamente,
127     // dentre todas as substrings nao necessariamente distintas
128     string res = "";
129     function<bool(int)> dfs_kth = [&](int u) {
130         int min_len = len[lnk[u]] + 1, max_len = len[u];
131         ll qnt = 1LL * (max_len - min_len + 1) * cnt[u];
132         if (qnt < k) {
133             k -= qnt;
134         } else {
135             res = s.substr(idx[u], min_len + (k - 1) / cnt[u]);
136             return true;
137         }
138         for (int v : adj[u])
139             if (dfs_kth(v)) return true;
140         return false;
141     };
142     dfs_kth(1);
143     return res;
144 }
145
146 int lcp(int i, int j) {
147     // retorna o maior prefixo comum entre os sufixos s[i..n] e s[j..n]
148     // tem que codar o lca aqui no automato
149     return len[lca(where[i], where[j])];
150 }
151 } st;

```

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

Código: trie.cpp

```

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

```

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.

Código: z.cpp

```

1 vector<int> get_z(string &s) {
2     int n = (int)s.size();
3     vector<int> z(n);
4     for (int i = 1, l = 0, r = 0; i < n; i++) {
5         if (i <= r) z[i] = min(r - i + 1, z[i - 1]);
6         while (i + z[i] < n && s[i + z[i]] == s[z[i]]) z[i]++;
7         if (i + z[i] - 1 > r) {
8             r = i + z[i] - 1;
9             l = i;
10        }
11    }
12    return z;
13 }
14
15 vector<int> matching(string &s, string &t) { // s = texto, t = padrao
16     string k = t + "$" + s;
17     vector<int> z = get_z(k), match;
18     int n = (int)t.size();
19     for (int i = n + 1; i < (int)z.size(); i++)
20         if (z[i] == n) match.push_back(i - n - 1);
21     return match;
22 }

```

7.12 Z function

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

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

Capítulo 8

Geometria

8.1 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.

Código: convex_hull.cpp

```
1 bool ccw(pt &p, pt &a, pt &b, bool include_collinear = 0) {
2     pt p1 = a - p;
3     pt p2 = b - p;
4     return include_collinear ? (p2 ^ p1) <= 0 : (p2 ^ p1) < 0;
5 }
6
7 void sort_by_angle(vector<pt> &v) { // sorteia o vetor por ângulo em relação ao pivô
8     pt p0 = *min_element(begin(v), end(v));
9     sort(begin(v), end(v), [&](pt &l, pt &r) { // clockwise
10         pt p1 = l - p0;
11         pt p2 = r - p0;
12         ll c1 = p1 ^ p2;
13         return c1 < 0 || ((c1 == 0) && p0.dist2(l) < p0.dist2(r));
14     });
15 }
16
```

```
17 vector<pt> convex_hull(vector<pt> v, bool include_collinear = 0) {
18     int n = size(v);
19
20     sort_by_angle(v);
21
22     if (include_collinear) {
23         for (int i = n - 2; i >= 0; i--) { // reverte o último lado do polígono
24             if (ccw(v[0], v[n - 1], v[i])) {
25                 reverse(begin(v) + i + 1, end(v));
26                 break;
27             }
28         }
29     }
30
31     vector<pt> ch{v[0], v[1]};
32     for (int i = 2; i < n; i++) {
33         while (ch.size() > 2 &&
34               (ccw(ch.end()[-2], ch.end()[-1], v[i], !include_collinear)))
35             ch.pop_back();
36         ch.emplace_back(v[i]);
37     }
38
39     return ch;
40 }
```

Capítulo 9

Grafos

9.1 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) $\Leftrightarrow (x \wedge y) \vee (\neg x \wedge \neg y)$`

Código: `2_sat.cpp`

```
1 struct sat2 {
2     int n;
3     vector<vector<int>> g, rg;
4     vector<bool> vis, assignment;
5     vector<int> topo, comp;
6
7     void build(int _n) {
8         n = 2 * _n;
9         g.assign(n, vector<int>());
10        rg.assign(n, vector<int>());
11    }
12
13    int get(int u) {
14        if (u < 0) return 2 * (~u) + 1;
15        else return 2 * u;
16    }
17
18    void add_impl(int u, int v) {
19        u = get(u), v = get(v);
20        g[u].push_back(v);
21        rg[v].push_back(u);
22        g[v ^ 1].push_back(u ^ 1);
23        rg[u ^ 1].push_back(v ^ 1);
24    }
25
26    void add_or(int u, int v) { add_impl(~u, v); }
27
28    void add_and(int u, int v) {
29        add_or(u, u);
30        add_or(v, v);
31    }
}
```

```

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

9.2 Binary Lifting

9.2.1 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.

Código: 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];
31    }
32 } 
```

```

31     return up[u][0];
32 }
33
34 int kth(int u, int k) {
35     for (int i = 0; i < LG; i++)
36         if (k & (1 << i)) u = up[u][i];
37     return u;
38 }
39
40 }

```

9.2.2 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**.

Código: `binary_lifting_query_nodo.cpp`

```

1 const int N = 3e5 + 5, LG = 20;
2 vector<int> adj[N];
3
4 namespace bl {
5     int t, up[N][LG], st[N][LG], tin[N], tout[N], val[N];
6
7     const int neutral = 0;
8     int merge(int l, int r) { return l + r; }
9
10    void dfs(int u, int p = -1) {
11        tin[u] = t++;
12        for (int i = 0; i < LG - 1; i++) {
13            up[u][i + 1] = up[up[u][i]][i];
14            st[u][i + 1] = merge(st[u][i], st[up[u][i]][i]);
15        }
16    }
17
18    int kth(int u, int k) {
19        for (int i = 0; i < LG; i++)
20            if (k & (1 << i)) u = up[u][i];
21        return u;
22    }
23
24    void build(int root) {
25        t = 1;
26        up[root][0] = root;
27        st[root][0] = neutral;
28        dfs(root);
29    }
30
31    bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }
32
33    int query2(int u, int v, bool include_lca) {
34        if (ancestor(u, v)) return include_lca ? val[u] : neutral;
35        int ans = val[u];
36        for (int i = LG - 1; i >= 0; i--) {
37            if (!ancestor(up[u][i], v)) {
38                ans = merge(ans, st[u][i]);
39                u = up[u][i];
40            }
41        }
42        return include_lca ? merge(ans, st[u][0]) : ans;
43    }
44
45    int query(int u, int v) {
46        if (u == v) return val[u];
47        return merge(query2(u, v, 1), query2(v, u, 0));
48    }
49
50    int lca(int u, int v) {
51        if (ancestor(u, v)) return u;
52        if (ancestor(v, u)) return v;
53        for (int i = LG - 1; i >= 0; i--)
54            if (!ancestor(up[u][i], v)) u = up[u][i];
55        return up[u][0];
56    }
57
58    int kth(int u, int k) {
59        for (int i = 0; i < LG; i++)
60            if (k & (1 << i)) u = up[u][i];
61        return u;
62    }
63
64    void print() {
65        cout << "Tin: ";
66        for (int i = 0; i < N; i++) cout << tin[i] << " ";
67        cout << endl;
68        cout << "Out: ";
69        for (int i = 0; i < N; i++) cout << tout[i] << " ";
70        cout << endl;
71        cout << "Up: ";
72        for (int i = 0; i < N; i++) {
73            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
74        }
75        cout << endl;
76        cout << "St: ";
77        for (int i = 0; i < N; i++) {
78            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
79        }
80        cout << endl;
81    }
82
83    void print_val() {
84        for (int i = 0; i < N; i++) cout << val[i] << " ";
85        cout << endl;
86    }
87
88    void print_tin() {
89        for (int i = 0; i < N; i++) cout << tin[i] << " ";
90        cout << endl;
91    }
92
93    void print_tout() {
94        for (int i = 0; i < N; i++) cout << tout[i] << " ";
95        cout << endl;
96    }
97
98    void print_up() {
99        for (int i = 0; i < N; i++) {
100            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
101        }
102        cout << endl;
103    }
104
105    void print_st() {
106        for (int i = 0; i < N; i++) {
107            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
108        }
109        cout << endl;
110    }
111
112    void print_val() {
113        for (int i = 0; i < N; i++) cout << val[i] << " ";
114        cout << endl;
115    }
116
117    void print_tin() {
118        for (int i = 0; i < N; i++) cout << tin[i] << " ";
119        cout << endl;
120    }
121
122    void print_tout() {
123        for (int i = 0; i < N; i++) cout << tout[i] << " ";
124        cout << endl;
125    }
126
127    void print_up() {
128        for (int i = 0; i < N; i++) {
129            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
130        }
131        cout << endl;
132    }
133
134    void print_st() {
135        for (int i = 0; i < N; i++) {
136            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
137        }
138        cout << endl;
139    }
140
141    void print_val() {
142        for (int i = 0; i < N; i++) cout << val[i] << " ";
143        cout << endl;
144    }
145
146    void print_tin() {
147        for (int i = 0; i < N; i++) cout << tin[i] << " ";
148        cout << endl;
149    }
150
151    void print_tout() {
152        for (int i = 0; i < N; i++) cout << tout[i] << " ";
153        cout << endl;
154    }
155
156    void print_up() {
157        for (int i = 0; i < N; i++) {
158            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
159        }
160        cout << endl;
161    }
162
163    void print_st() {
164        for (int i = 0; i < N; i++) {
165            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
166        }
167        cout << endl;
168    }
169
170    void print_val() {
171        for (int i = 0; i < N; i++) cout << val[i] << " ";
172        cout << endl;
173    }
174
175    void print_tin() {
176        for (int i = 0; i < N; i++) cout << tin[i] << " ";
177        cout << endl;
178    }
179
180    void print_tout() {
181        for (int i = 0; i < N; i++) cout << tout[i] << " ";
182        cout << endl;
183    }
184
185    void print_up() {
186        for (int i = 0; i < N; i++) {
187            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
188        }
189        cout << endl;
190    }
191
192    void print_st() {
193        for (int i = 0; i < N; i++) {
194            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
195        }
196        cout << endl;
197    }
198
199    void print_val() {
200        for (int i = 0; i < N; i++) cout << val[i] << " ";
201        cout << endl;
202    }
203
204    void print_tin() {
205        for (int i = 0; i < N; i++) cout << tin[i] << " ";
206        cout << endl;
207    }
208
209    void print_tout() {
210        for (int i = 0; i < N; i++) cout << tout[i] << " ";
211        cout << endl;
212    }
213
214    void print_up() {
215        for (int i = 0; i < N; i++) {
216            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
217        }
218        cout << endl;
219    }
220
221    void print_st() {
222        for (int i = 0; i < N; i++) {
223            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
224        }
225        cout << endl;
226    }
227
228    void print_val() {
229        for (int i = 0; i < N; i++) cout << val[i] << " ";
230        cout << endl;
231    }
232
233    void print_tin() {
234        for (int i = 0; i < N; i++) cout << tin[i] << " ";
235        cout << endl;
236    }
237
238    void print_tout() {
239        for (int i = 0; i < N; i++) cout << tout[i] << " ";
240        cout << endl;
241    }
242
243    void print_up() {
244        for (int i = 0; i < N; i++) {
245            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
246        }
247        cout << endl;
248    }
249
250    void print_st() {
251        for (int i = 0; i < N; i++) {
252            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
253        }
254        cout << endl;
255    }
256
257    void print_val() {
258        for (int i = 0; i < N; i++) cout << val[i] << " ";
259        cout << endl;
260    }
261
262    void print_tin() {
263        for (int i = 0; i < N; i++) cout << tin[i] << " ";
264        cout << endl;
265    }
266
267    void print_tout() {
268        for (int i = 0; i < N; i++) cout << tout[i] << " ";
269        cout << endl;
270    }
271
272    void print_up() {
273        for (int i = 0; i < N; i++) {
274            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
275        }
276        cout << endl;
277    }
278
279    void print_st() {
280        for (int i = 0; i < N; i++) {
281            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
282        }
283        cout << endl;
284    }
285
286    void print_val() {
287        for (int i = 0; i < N; i++) cout << val[i] << " ";
288        cout << endl;
289    }
290
291    void print_tin() {
292        for (int i = 0; i < N; i++) cout << tin[i] << " ";
293        cout << endl;
294    }
295
296    void print_tout() {
297        for (int i = 0; i < N; i++) cout << tout[i] << " ";
298        cout << endl;
299    }
300
301    void print_up() {
302        for (int i = 0; i < N; i++) {
303            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
304        }
305        cout << endl;
306    }
307
308    void print_st() {
309        for (int i = 0; i < N; i++) {
310            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
311        }
312        cout << endl;
313    }
314
315    void print_val() {
316        for (int i = 0; i < N; i++) cout << val[i] << " ";
317        cout << endl;
318    }
319
320    void print_tin() {
321        for (int i = 0; i < N; i++) cout << tin[i] << " ";
322        cout << endl;
323    }
324
325    void print_tout() {
326        for (int i = 0; i < N; i++) cout << tout[i] << " ";
327        cout << endl;
328    }
329
330    void print_up() {
331        for (int i = 0; i < N; i++) {
332            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
333        }
334        cout << endl;
335    }
336
337    void print_st() {
338        for (int i = 0; i < N; i++) {
339            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
340        }
341        cout << endl;
342    }
343
344    void print_val() {
345        for (int i = 0; i < N; i++) cout << val[i] << " ";
346        cout << endl;
347    }
348
349    void print_tin() {
350        for (int i = 0; i < N; i++) cout << tin[i] << " ";
351        cout << endl;
352    }
353
354    void print_tout() {
355        for (int i = 0; i < N; i++) cout << tout[i] << " ";
356        cout << endl;
357    }
358
359    void print_up() {
360        for (int i = 0; i < N; i++) {
361            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
362        }
363        cout << endl;
364    }
365
366    void print_st() {
367        for (int i = 0; i < N; i++) {
368            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
369        }
370        cout << endl;
371    }
372
373    void print_val() {
374        for (int i = 0; i < N; i++) cout << val[i] << " ";
375        cout << endl;
376    }
377
378    void print_tin() {
379        for (int i = 0; i < N; i++) cout << tin[i] << " ";
380        cout << endl;
381    }
382
383    void print_tout() {
384        for (int i = 0; i < N; i++) cout << tout[i] << " ";
385        cout << endl;
386    }
387
388    void print_up() {
389        for (int i = 0; i < N; i++) {
390            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
391        }
392        cout << endl;
393    }
394
395    void print_st() {
396        for (int i = 0; i < N; i++) {
397            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
398        }
399        cout << endl;
400    }
401
402    void print_val() {
403        for (int i = 0; i < N; i++) cout << val[i] << " ";
404        cout << endl;
405    }
406
407    void print_tin() {
408        for (int i = 0; i < N; i++) cout << tin[i] << " ";
409        cout << endl;
410    }
411
412    void print_tout() {
413        for (int i = 0; i < N; i++) cout << tout[i] << " ";
414        cout << endl;
415    }
416
417    void print_up() {
418        for (int i = 0; i < N; i++) {
419            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
420        }
421        cout << endl;
422    }
423
424    void print_st() {
425        for (int i = 0; i < N; i++) {
426            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
427        }
428        cout << endl;
429    }
430
431    void print_val() {
432        for (int i = 0; i < N; i++) cout << val[i] << " ";
433        cout << endl;
434    }
435
436    void print_tin() {
437        for (int i = 0; i < N; i++) cout << tin[i] << " ";
438        cout << endl;
439    }
440
441    void print_tout() {
442        for (int i = 0; i < N; i++) cout << tout[i] << " ";
443        cout << endl;
444    }
445
446    void print_up() {
447        for (int i = 0; i < N; i++) {
448            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
449        }
450        cout << endl;
451    }
452
453    void print_st() {
454        for (int i = 0; i < N; i++) {
455            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
456        }
457        cout << endl;
458    }
459
460    void print_val() {
461        for (int i = 0; i < N; i++) cout << val[i] << " ";
462        cout << endl;
463    }
464
465    void print_tin() {
466        for (int i = 0; i < N; i++) cout << tin[i] << " ";
467        cout << endl;
468    }
469
470    void print_tout() {
471        for (int i = 0; i < N; i++) cout << tout[i] << " ";
472        cout << endl;
473    }
474
475    void print_up() {
476        for (int i = 0; i < N; i++) {
477            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
478        }
479        cout << endl;
480    }
481
482    void print_st() {
483        for (int i = 0; i < N; i++) {
484            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
485        }
486        cout << endl;
487    }
488
489    void print_val() {
490        for (int i = 0; i < N; i++) cout << val[i] << " ";
491        cout << endl;
492    }
493
494    void print_tin() {
495        for (int i = 0; i < N; i++) cout << tin[i] << " ";
496        cout << endl;
497    }
498
499    void print_tout() {
500        for (int i = 0; i < N; i++) cout << tout[i] << " ";
501        cout << endl;
502    }
503
504    void print_up() {
505        for (int i = 0; i < N; i++) {
506            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
507        }
508        cout << endl;
509    }
510
511    void print_st() {
512        for (int i = 0; i < N; i++) {
513            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
514        }
515        cout << endl;
516    }
517
518    void print_val() {
519        for (int i = 0; i < N; i++) cout << val[i] << " ";
520        cout << endl;
521    }
522
523    void print_tin() {
524        for (int i = 0; i < N; i++) cout << tin[i] << " ";
525        cout << endl;
526    }
527
528    void print_tout() {
529        for (int i = 0; i < N; i++) cout << tout[i] << " ";
530        cout << endl;
531    }
532
533    void print_up() {
534        for (int i = 0; i < N; i++) {
535            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
536        }
537        cout << endl;
538    }
539
540    void print_st() {
541        for (int i = 0; i < N; i++) {
542            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
543        }
544        cout << endl;
545    }
546
547    void print_val() {
548        for (int i = 0; i < N; i++) cout << val[i] << " ";
549        cout << endl;
550    }
551
552    void print_tin() {
553        for (int i = 0; i < N; i++) cout << tin[i] << " ";
554        cout << endl;
555    }
556
557    void print_tout() {
558        for (int i = 0; i < N; i++) cout << tout[i] << " ";
559        cout << endl;
560    }
561
562    void print_up() {
563        for (int i = 0; i < N; i++) {
564            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
565        }
566        cout << endl;
567    }
568
569    void print_st() {
570        for (int i = 0; i < N; i++) {
571            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
572        }
573        cout << endl;
574    }
575
576    void print_val() {
577        for (int i = 0; i < N; i++) cout << val[i] << " ";
578        cout << endl;
579    }
580
581    void print_tin() {
582        for (int i = 0; i < N; i++) cout << tin[i] << " ";
583        cout << endl;
584    }
585
586    void print_tout() {
587        for (int i = 0; i < N; i++) cout << tout[i] << " ";
588        cout << endl;
589    }
590
591    void print_up() {
592        for (int i = 0; i < N; i++) {
593            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
594        }
595        cout << endl;
596    }
597
598    void print_st() {
599        for (int i = 0; i < N; i++) {
600            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
601        }
602        cout << endl;
603    }
604
605    void print_val() {
606        for (int i = 0; i < N; i++) cout << val[i] << " ";
607        cout << endl;
608    }
609
610    void print_tin() {
611        for (int i = 0; i < N; i++) cout << tin[i] << " ";
612        cout << endl;
613    }
614
615    void print_tout() {
616        for (int i = 0; i < N; i++) cout << tout[i] << " ";
617        cout << endl;
618    }
619
620    void print_up() {
621        for (int i = 0; i < N; i++) {
622            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
623        }
624        cout << endl;
625    }
626
627    void print_st() {
628        for (int i = 0; i < N; i++) {
629            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
630        }
631        cout << endl;
632    }
633
634    void print_val() {
635        for (int i = 0; i < N; i++) cout << val[i] << " ";
636        cout << endl;
637    }
638
639    void print_tin() {
640        for (int i = 0; i < N; i++) cout << tin[i] << " ";
641        cout << endl;
642    }
643
644    void print_tout() {
645        for (int i = 0; i < N; i++) cout << tout[i] << " ";
646        cout << endl;
647    }
648
649    void print_up() {
650        for (int i = 0; i < N; i++) {
651            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
652        }
653        cout << endl;
654    }
655
656    void print_st() {
657        for (int i = 0; i < N; i++) {
658            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
659        }
660        cout << endl;
661    }
662
663    void print_val() {
664        for (int i = 0; i < N; i++) cout << val[i] << " ";
665        cout << endl;
666    }
667
668    void print_tin() {
669        for (int i = 0; i < N; i++) cout << tin[i] << " ";
670        cout << endl;
671    }
672
673    void print_tout() {
674        for (int i = 0; i < N; i++) cout << tout[i] << " ";
675        cout << endl;
676    }
677
678    void print_up() {
679        for (int i = 0; i < N; i++) {
680            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
681        }
682        cout << endl;
683    }
684
685    void print_st() {
686        for (int i = 0; i < N; i++) {
687            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
688        }
689        cout << endl;
690    }
691
692    void print_val() {
693        for (int i = 0; i < N; i++) cout << val[i] << " ";
694        cout << endl;
695    }
696
697    void print_tin() {
698        for (int i = 0; i < N; i++) cout << tin[i] << " ";
699        cout << endl;
700    }
701
702    void print_tout() {
703        for (int i = 0; i < N; i++) cout << tout[i] << " ";
704        cout << endl;
705    }
706
707    void print_up() {
708        for (int i = 0; i < N; i++) {
709            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
710        }
711        cout << endl;
712    }
713
714    void print_st() {
715        for (int i = 0; i < N; i++) {
716            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
717        }
718        cout << endl;
719    }
720
721    void print_val() {
722        for (int i = 0; i < N; i++) cout << val[i] << " ";
723        cout << endl;
724    }
725
726    void print_tin() {
727        for (int i = 0; i < N; i++) cout << tin[i] << " ";
728        cout << endl;
729    }
730
731    void print_tout() {
732        for (int i = 0; i < N; i++) cout << tout[i] << " ";
733        cout << endl;
734    }
735
736    void print_up() {
737        for (int i = 0; i < N; i++) {
738            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
739        }
740        cout << endl;
741    }
742
743    void print_st() {
744        for (int i = 0; i < N; i++) {
745            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
746        }
747        cout << endl;
748    }
749
750    void print_val() {
751        for (int i = 0; i < N; i++) cout << val[i] << " ";
752        cout << endl;
753    }
754
755    void print_tin() {
756        for (int i = 0; i < N; i++) cout << tin[i] << " ";
757        cout << endl;
758    }
759
760    void print_tout() {
761        for (int i = 0; i < N; i++) cout << tout[i] << " ";
762        cout << endl;
763    }
764
765    void print_up() {
766        for (int i = 0; i < N; i++) {
767            for (int j = 0; j < LG; j++) cout << up[i][j] << " ";
768        }
769        cout << endl;
770    }
771
772    void print_st() {
773        for (int i = 0; i < N; i++) {
774            for (int j = 0; j < LG; j++) cout << st[i][j] << " ";
775        }
776        cout << endl;
777    }
778
779    void print_val() {
780        for (int i = 0; i < N; i++) cout << val[i] << " ";
781        cout << endl;
782    }
783
784    void print_tin() {
785        for (int i = 0; i < N; i++) cout << tin[i] << " ";
786        cout << endl;
787
```

```
63
64 }
```

9.2.3 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>

Código: `binary_lifting_query_nodo2.cpp`

```
1 struct node {
2     int pref, suff, sum, best;
3     node() : pref(0), suff(0), sum(0), best(0) { }
4     node(int x) : pref(x), suff(x), sum(x), best(x) { }
5     node(int a, int b, int c, int d) : pref(a), suff(b), sum(c), best(d) { }
6 };
7
8 node merge(node l, node r) {
9     int pref = max(l.pref, l.sum + r.pref);
10    int suff = max(r.suff, r.sum + l.suff);
11    int sum = l.sum + r.sum;
12    int best = max(l.suff + r.pref, max(l.best, r.best));
13    return node(pref, suff, sum, best);
14 }
15
16 struct BinaryLifting {
17     vector<vector<int>> adj, up;
18     vector<int> val, tin, tout;
19     vector<vector<node>> st, st2;
20     int N, LG, t;
21
22     void build(int u, int p = -1) {
23         tin[u] = t++;
24         for (int i = 0; i < LG - 1; i++) {
25             up[u][i + 1] = up[up[u][i]][i];
26             st[u][i + 1] = merge(st[u][i], st[up[u][i]][i]);
27             st2[u][i + 1] = merge(st2[up[u][i]][i], st2[u][i]);
28         }
29         for (int v : adj[u])
```

```
30             if (v != p) {
31                 up[v][0] = u;
32                 st[v][0] = node(val[u]);
33                 st2[v][0] = node(val[u]);
34                 build(v, u);
35             }
36             tout[u] = t++;
37         }
38
39         void build(int root, vector<vector<int>> adj2, vector<int> v) {
40             t = 1;
41             N = (int)adj2.size();
42             LG = 32 - __builtin_clz(N);
43             adj = adj2;
44             val = v;
45             tin = tout = vector<int>(N);
46             up = vector(N, vector<int>(LG));
47             st = st2 = vector(N, vector<node>(LG));
48             up[root][0] = root;
49             st[root][0] = node(val[root]);
50             st2[root][0] = node(val[root]);
51             build(root);
52         }
53
54         bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }
55
56         node query2(int u, int v, bool include_lca, bool invert) {
57             if (ancestor(u, v)) return include_lca ? node(val[u]) : node();
58             node ans = node(val[u]);
59             for (int i = LG - 1; i >= 0; i--) {
60                 if (!ancestor(up[u][i], v)) {
61                     if (invert) ans = merge(st2[u][i], ans);
62                     else ans = merge(ans, st[u][i]);
63                     u = up[u][i];
64                 }
65             }
66             return include_lca ? merge(ans, st[u][0]) : ans;
67         }
68
69         node query(int u, int v) {
70             if (u == v) return node(val[u]);
71             node l = query2(u, v, 1, 0);
72             node r = query2(v, u, 0, 1);
73             return merge(l, r);
74         }
75
76         int lca(int u, int v) {
```

```

77     if (ancestor(u, v)) return u;
78     if (ancestor(v, u)) return v;
79     for (int i = LG - 1; i >= 0; i--)
80         if (!ancestor(up[u][i], v)) u = up[u][i];
81     return up[u][0];
82 }
83 } bl, bl2;

```

9.2.4 Binary Lifting Query Aresta

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

Código: binary_lifting_query_aresta.cpp

```

1 const int N = 3e5 + 5, LG = 20;
2 vector<pair<int, int>> adj[N];
3
4 namespace bl {
5     int t, up[N][LG], st[N][LG], tin[N], tout[N], val[N];
6
7     const int neutral = 0;
8     int merge(int l, int r) { return l + r; }
9
10    void dfs(int u, int p = -1) {
11        tin[u] = t++;
12        for (int i = 0; i < LG - 1; i++) {
13            up[u][i + 1] = up[up[u][i]][i];
14            st[u][i + 1] = merge(st[u][i], st[up[u][i]][i]);
15        }
16        for (auto [w, v] : adj[u])
17            if (v != p) {
18                up[v][0] = u, st[v][0] = w;
19                dfs(v, u);
20            }
21        tout[u] = t++;
22    }
23
24    void build(int root) {
25        t = 1;
26        up[root][0] = root;

```

```

27        st[root][0] = neutral;
28        dfs(root);
29    }
30
31    bool ancestor(int u, int v) { return tin[u] <= tin[v] && tout[u] >= tout[v]; }
32
33    int query2(int u, int v) {
34        if (ancestor(u, v)) return neutral;
35        int ans = neutral;
36        for (int i = LG - 1; i >= 0; i--) {
37            if (!ancestor(up[u][i], v)) {
38                ans = merge(ans, st[u][i]);
39                u = up[u][i];
40            }
41        }
42        return merge(ans, st[u][0]);
43    }
44
45    int query(int u, int v) {
46        if (u == v) {
47            return neutral;
48 #warning TRATAR ESSE CASO ACIMA
49        }
50        return merge(query2(u, v), query2(v, u));
51    }
52
53    int lca(int u, int v) {
54        if (ancestor(u, v)) return u;
55        if (ancestor(v, u)) return v;
56        for (int i = LG - 1; i >= 0; i--) {
57            if (!ancestor(up[u][i], v)) u = up[u][i];
58        }
59        return up[u][0];
60    }
61    int kth(int u, int k) {
62        for (int i = 0; i < LG; i++) {
63            if (k & (1 << i)) u = up[u][i];
64        }
65    }
66 }

```

9.3 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`

```

1 struct Bct {
2     int T;
3     vector<int> tin, low, stk, art, id, splits;
4     vector<vector<int>> adj, g, comp, up;
5     int n, sz, m;
6     void build(int _n, int _m) {
7         n = _n, m = _m;
8         adj.resize(n);
9     }
10    void add_edge(int u, int v) {
11        adj[u].emplace_back(v);
12        adj[v].emplace_back(u);
13    }
14    void dfs(int u, int p) {
15        low[u] = tin[u] = ++T;
16        stk.emplace_back(u);
17        for (auto v : adj[u]) {
18            if (tin[v] == -1) {
19                dfs(v, u);
20                low[u] = min(low[u], low[v]);
21                if (low[v] >= tin[u]) {
22                    int x;
23                    sz++;
24                    do {
25                        assert(stk.size());
26                        x = stk.back();
27                        stk.pop_back();
28                        comp[x].emplace_back(sz);
29                    } while (x != v);
30                }
31            } else if (v != p) {
32                low[u] = min(low[u], tin[v]);
33            }
34        }
35    }
36}
37 inline bool is_articulation_point(int u) { return art[id[u]]; }
38 inline int number_of_splits(int u) { return splits[id[u]]; }
39 void work() {
40     T = sz = 0;
41     stk.clear();
42     tin.resize(n, -1);
43     comp.resize(n);
44     low.resize(n);
45     for (int i = 0; i < n; i++)
46         if (tin[i] == -1) dfs(i, 0);
47     art.resize(sz + n + 1);
48     splits.resize(n + sz + 1, 1);
49     id.resize(n);
50     g.resize(sz + n + 1);
51     for (int i = 0; i < n; i++) {
52         if ((int)comp[i].size() > 1) {
53             id[i] = ++sz;
54             art[id[i]] = 1;
55             splits[id[i]] = (int)comp[i].size();
56             for (auto u : comp[i]) {
57                 g[id[i]].emplace_back(u);
58                 g[u].emplace_back(id[i]);
59             }
60         } else if (comp[i].size()) {
61             id[i] = comp[i][0];
62         }
63     }
64 }
65 };

```

9.4 Caminho Euleriano

9.4.1 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 ($\deg_{out}[v] == \deg_{in}[v] - 1$).

Código: directed_eulerian_path.cpp

```

1 const int MAXN = 1e6 + 6;
2
3 vector<int> adj[MAXN];
4
5 struct EulerianTrail {
6     int n;
7     int it[MAXN], deg_in[MAXN], deg_out[MAXN];
8     void build(int _n) {
9         n = _n;
10        for (int i = 0; i < n; i++) it[i] = deg_in[i] = deg_out[i] = 0;
11    }
12    vector<int> find() {
13        vector<int> cur;
14        int m = 0;
15        for (int i = 0; i < n; i++) {
16            for (int j : adj[i]) {
17                m++;
18                deg_out[i]++, deg_in[j]++;
19            }
20        }
21        int start = -1, end = -1;
22        for (int i = 0; i < n; i++) {
23            if (deg_in[i] != deg_out[i]) {
24                if (deg_in[i] == deg_out[i] - 1)
25                    if (start == -1) start = i;

```

```

26                else return {};
27            else if (deg_in[i] - 1 == deg_out[i])
28                if (end == -1) end = i;
29                else return {};
30            else return {};
31        }
32    }
33    if (start == -1 && end == -1) {
34        // pode começar em qualquer vértice com alguma aresta (mas tem que terminar
35        // nele também), nesse caso eh ciclo euleriano
36        for (int i = 0; i < n; i++) {
37            if (deg_out[i] > 0) {
38                start = i;
39                end = i;
40                break;
41            }
42        }
43    } else if (start == -1 || end == -1) {
44        return {};
45    }
46    function<void(int)> dfs_et = [&](int u) {
47        while (it[u] < (int)adj[u].size()) {
48            int v = adj[u][it[u]++];
49            dfs_et(v);
50        }
51        cur.push_back(u);
52    };
53    dfs_et(start);
54    if ((int)cur.size() != m + 1) return {};
55    reverse(cur.begin(), cur.end());
56    return cur;
57}
58} et_finder;

```

9.4.2 Caminho Euleriano Não 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

```

1 const int MAXN = 1e6 + 6, MAXM = 2e6 + 6;
2
3 vector<pair<int, int>> adj[MAXN]; // {nodo, id da aresta}
4
5 struct EulerianTrail {
6     int n, m;
7     int it[MAXN], deg[MAXN], vis_edge[MAXM];
8     void build(int _n, int _m) {
9         n = _n;
10        m = _m;
11        for (int i = 0; i < n; i++) it[i] = deg[i] = 0;
12        for (int i = 0; i < m; i++) vis_edge[i] = 0;
13    }
14    vector<int> find() {
15        vector<int> cur;
16        for (int i = 0; i < n; i++) deg[i] = (int)adj[i].size();
17        int start = -1, end = -1;
18        for (int i = 0; i < n; i++) {
19            if (deg[i] & 1) {
20                if (start == -1) start = i;
21                else if (end == -1) end = i;
22                else return {};
23            }
24        }
25        if (start == -1 && end == -1) {
26            // pode comecar em qualquer vertice com alguma aresta (mas tem que terminar
27            // nele tambem), nesse caso eh ciclo euleriano
28            for (int i = 0; i < n; i++) {
29                if (deg[i] > 0) {
30                    start = i;
31                    end = i;
32                    break;
33                }
34            }
35        } else if (start == -1 || end == -1) {
36            return {};
37        }
38        function<void(int)> dfs_et = [&](int u) {

```

```

39            while (it[u] < (int)adj[u].size()) {
40                auto [v, id] = adj[u][it[u]++;
41                if (vis_edge[id]) continue;
42                vis_edge[id] = 1;
43                dfs_et(v);
44            }
45            cur.push_back(u);
46        };
47        dfs_et(start);
48        if ((int)cur.size() != m + 1) return {};
49        reverse(cur.begin(), cur.end());
50        return cur;
51    }
52} et_finder;

```

9.5 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

```

1 const int INF = 1e9 + 9;
2
3 vector<vector<int>> adj;
4
5 struct GraphCenter {
6     int n, diam = 0;
7     vector<int> centros, dist, pai;
8     int bfs(int s) {
9         queue<int> q;
10        q.push(s);
11        dist.assign(n + 5, INF);
12        pai.assign(n + 5, -1);
13        dist[s] = 0;
14        int maxidist = 0, maxinode = 0;

```

```

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

```

9.6 Centroids

9.6.1 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 subárvore com no máximo metade dos nodos da árvore original. Em outras palavras, se a árvore tem tamanho N , todas as subárvore geradas pela remoção do centroide têm tamanho no máximo $\frac{N}{2}$. Uma árvore pode ter até dois centróides.

Código: find_centroid.cpp

```

1 const int N = 3e5 + 5;
2
3 int sz[N];
4 vector<int> adj[N];
5
6 void dfs_sz(int u, int p) {
7     sz[u] = 1;
8     for (int v : adj[u]) {
9         if (v != p) {
10             dfs_sz(v, u);
11             sz[u] += sz[v];
12         }
13     }
14 }
15
16 int centroid(int u, int p, int n) {
17     for (int v : adj[u])
18         if (v != p && sz[v] > n / 2) return centroid(v, u, n);
19     return u;
20 }
21
22 pair<int, int> centroids(int u) {
23     dfs_sz(u, u);
24     int c = centroid(u, u, sz[u]);
25     int c2 = -1;
26     for (int v : adj[c])
27         if (sz[u] == sz[v] * 2) c2 = v;
28     return {c, c2};
29 }

```

9.6.2 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 subárvore 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.

Código: centroid_decomposition.cpp

```

1 const int N = 3e5 + 5;
2
3 int sz[N], par[N];
4 bool rem[N];
5 vector<int> dis[N];
6 vector<int> adj[N];
7
8 int dfs_sz(int u, int p) {
9     sz[u] = 1;
10    for (int v : adj[u])
11        if (v != p && !rem[v]) sz[u] += dfs_sz(v, u);
12    return sz[u];
13 }
14
15 int centroid(int u, int p, int szn) {
16     for (int v : adj[u])
17         if (v != p && !rem[v] && sz[v] > szn / 2) return centroid(v, u, szn);
18     return u;
19 }
20
21 void dfs_dis(int u, int p, int d = 0) {
22     dis[u].push_back(d);
23     for (int v : adj[u])
24         if (v != p && !rem[v]) dfs_dis(v, u, d + 1);
25 }
26
27 void decomp(int u, int p) {
28     int c = centroid(u, u, dfs_sz(u, u));
29
30     rem[c] = true;
31     par[c] = p;
32
33     dfs_dis(c, c);
34
35     // Faz algo na subárvore de c
36 }
```

```

37     for (int v : adj[c])
38         if (!rem[v]) decomp(v, c);
39 }
40
41 void build(int n) {
42     for (int i = 0; i < n; i++) {
43         rem[i] = false;
44         dis[i].clear();
45     }
46     decomp(0, -1);
47     for (int i = 0; i < n; i++) reverse(dis[i].begin(), dis[i].end());
48 }
```

9.7 Ciclos

9.7.1 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.

Código: find_cycle.cpp

```

1 const int MAXN = 1e6 + 6;
2
3 vector<int> adj[MAXN];
4
5 struct CycleFinder {
6     int n;
7     bool trivial;
8     vector<int> vis, par;
9     int start = -1, end = -1;
10    void build(int _n, bool _trivial = 1) {
11        n = _n;
12        trivial = _trivial;
13        // trivial eh um flag que indica se o algoritmo deve aceitar ou nao
14        // ciclos triviais, um ciclo trivial eh um ciclo de tamanho 1 ou 2
15    }
16    bool dfs(int u) {
17        vis[u] = 1;
18        for (int v : adj[u]) {
19            if (vis[v] == 0) {
```

```

20         par[v] = u;
21         if (dfs(v)) return true;
22     } else if (vis[v] == 1) {
23         if (trivial || (par[u] != v && u != v)) {
24             end = u;
25             start = v;
26             return true;
27         }
28     }
29     vis[u] = 2;
30     return false;
31 }
32 vector<int> get_cycle() {
33     vis.assign(n, 0);
34     par.assign(n, -1);
35     for (int v = 0; v < n; v++)
36         if (vis[v] == 0 && dfs(v)) break;
37     vector<int> cycle;
38     if (start != -1) {
39         cycle.emplace_back(start);
40         for (int v = end; v != start; v = par[v]) cycle.emplace_back(v);
41         cycle.emplace_back(start);
42         reverse(cycle.begin(), cycle.end());
43     }
44     return cycle;
45 }
46 } finder;
47 }
```

9.7.2 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.

Código: find_negative_cycle.cpp

```

1 struct NegativeCycleFinder {
2     const ll INF = 1e18;
3     int n;
4     vector<tuple<int, int, ll>> edges;
5     void build(int _n) {
6         n = _n;
```

```

7         edges.clear();
8     }
9     void add_edge(int u, int v, ll w) { edges.emplace_back(u, v, w); }
10    vector<int> get_cycle() {
11        vector<ll> d(n);
12        vector<int> p(n, -1);
13        int x;
14        for (int i = 0; i < n; ++i) {
15            x = -1;
16            for (auto [u, v, w] : edges) {
17                if (d[u] < INF) {
18                    if (d[u] + w < d[v]) {
19                        d[v] = max(-INF, d[u] + w);
20                        p[v] = u;
21                        x = v;
22                    }
23                }
24            }
25        }
26        vector<int> cycle;
27        if (x != -1) {
28            for (int i = 0; i < n; ++i) x = p[x];
29            for (int v = x;; v = p[v]) {
30                cycle.push_back(v);
31                if (v == x && cycle.size() > 1) break;
32            }
33            reverse(cycle.begin(), cycle.end());
34        }
35        return cycle;
36    }
37 } finder;
```

9.8 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 possível e retorna a quantidade de fluxo adicionado.

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

A Função `min_cut` acha as arestas pertencentes ao corte mínimo 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: $\mathcal{O}(V * E)$

Código: EdmondsKarp.cpp

```

1 const long long INF = 1e18;
2
3 struct FlowEdge {
4     int u, v;
5     long long cap, flow = 0;
6     FlowEdge(int u, int v, long long cap) : u(u), v(v), cap(cap) {}
7 };
8
9 struct EdmondsKarp {
10     int n, s, t, m = 0, vistoken = 0;
11     vector<FlowEdge> edges;
12     vector<vector<int>> adj;
13     vector<int> visto;
14
15     EdmondsKarp(int n, int s, int t) : n(n), s(s), t(t) {
16         adj.resize(n);
17         visto.resize(n);
18     }
19
20     void add_edge(int u, int v, long long cap) {
21         edges.emplace_back(u, v, cap);
22         edges.emplace_back(v, u, 0);

```

```

23         adj[u].push_back(m);
24         adj[v].push_back(m + 1);
25         m += 2;
26     }
27
28     int bfs() {
29         vistoken++;
30         queue<int> fila;
31         fila.push(s);
32         vector<int> pego(n, -1);
33         while (!fila.empty()) {
34             int u = fila.front();
35             if (u == t) break;
36             fila.pop();
37             visto[u] = vistoken;
38             for (int id : adj[u]) {
39                 if (edges[id].cap - edges[id].flow < 1) continue;
40                 int v = edges[id].v;
41                 if (visto[v] == -1) continue;
42                 fila.push(v);
43                 pego[v] = id;
44             }
45         }
46         if (pego[t] == -1) return 0;
47         long long f = INF;
48         for (int id = pego[t]; id != -1; id = pego[edges[id].u])
49             f = min(f, edges[id].cap - edges[id].flow);
50         for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
51             edges[id].flow += f;
52             edges[id ^ 1].flow -= f;
53         }
54         return f;
55     }
56
57     long long flow() {
58         long long maxflow = 0;
59         while (long long f = bfs()) maxflow += f;
60         return maxflow;
61     }
62 };

```

Código: Dinic.cpp

```

1 typedef long long ll;
2
3 const ll INF = 1e18;
4

```

```

5 struct FlowEdge {
6     int u, v;
7     ll cap, flow = 0;
8     FlowEdge(int u, int v, ll cap) : u(u), v(v), cap(cap) { }
9 };
10
11 struct Dinic {
12     vector<FlowEdge> edges;
13     vector<vector<int>> adj;
14     int n, s, t, m = 0;
15     vector<int> level, ptr;
16     queue<int> q;
17     Dinic(int n, int s, int t) : n(n), s(s), t(t) {
18         adj.resize(n);
19         level.resize(n);
20         ptr.resize(n);
21     }
22     void add_edge(int u, int v, ll cap) {
23         edges.emplace_back(u, v, cap);
24         edges.emplace_back(v, u, 0);
25         adj[u].push_back(m);
26         adj[v].push_back(m + 1);
27         m += 2;
28     }
29     bool bfs() {
30         while (!q.empty()) {
31             int u = q.front();
32             q.pop();
33             for (int id : adj[u]) {
34                 if (edges[id].cap - edges[id].flow < 1) continue;
35                 int v = edges[id].v;
36                 if (level[v] != -1) continue;
37                 level[v] = level[u] + 1;
38                 q.push(v);
39             }
40         }
41         return level[t] != -1;
42     }
43     ll dfs(int u, ll f) {
44         if (f == 0) return 0;
45         if (u == t) return f;
46         for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++) {
47             int id = adj[u][cid];
48             int v = edges[id].v;
49             if (level[u] + 1 != level[v] || edges[id].cap - edges[id].flow < 1)
50                 continue;
51             ll tr = dfs(v, min(f, edges[id].cap - edges[id].flow));
52             edges[id].flow += tr;
53             edges[id ^ 1].flow -= tr;
54             if (tr == 0) continue;
55             return tr;
56         }
57     }
58     ll flow() {
59         ll maxflow = 0;
60         while (true) {
61             fill(level.begin(), level.end(), -1);
62             level[s] = 0;
63             q.push(s);
64             if (!bfs()) break;
65             fill(ptr.begin(), ptr.end(), 0);
66             while (ll f = dfs(s, INF)) maxflow += f;
67         }
68         return maxflow;
69     }
70     void min_cut() {
71         vector<bool> vis(n);
72         function<void(int)> dfs = [&](int u) {
73             vis[u] = 1;
74             for (int id : adj[u]) {
75                 int v = edges[id].v;
76                 if (!vis[v] && edges[id].cap - edges[id].flow > 0) dfs(v);
77             }
78         };
79         dfs(s);
80         for (int id = 0; id < (int)edges.size(); id++) {
81             auto [u, v, cap, flow] = edges[id];
82             if (vis[u] ^ vis[v] && cap > 0) {
83                 // this edge is in the min cut
84                 // do something here
85             }
86         }
87     }
88 };

```

Codigo: MinCostMaxFlow.cpp

```

1 struct MinCostMaxFlow {
2     int n, s, t, m = 0;
3     ll maxflow = 0, mincost = 0;
4     vector<FlowEdge> edges;
5     vector<vector<int>> adj;
6

```

```

7     MinCostMaxFlow(int n, int s, int t) : n(n), s(s), t(t) { adj.resize(n); }
8
9     void add_edge(int u, int v, ll cap, ll cost) {
10        edges.emplace_back(u, v, cap, cost);
11        edges.emplace_back(v, u, 0, -cost);
12        adj[u].push_back(m);
13        adj[v].push_back(m + 1);
14        m += 2;
15    }
16
17    bool spfa() {
18        vector<int> pego(n, -1);
19        vector<ll> dis(n, INF);
20        vector<bool> inq(n, false);
21        queue<int> fila;
22        fila.push(s);
23        dis[s] = 0;
24        inq[s] = 1;
25        while (!fila.empty()) {
26            int u = fila.front();
27            fila.pop();
28            inq[u] = false;
29            for (int id : adj[u]) {
30                if (edges[id].cap - edges[id].flow < 1) continue;
31                int v = edges[id].v;
32                if (dis[v] > dis[u] + edges[id].cost) {
33                    dis[v] = dis[u] + edges[id].cost;
34                    pego[v] = id;
35                    if (!inq[v]) {
36                        inq[v] = true;
37                        fila.push(v);
38                    }
39                }
40            }
41        }
42
43        if (pego[t] == -1) return 0;
44        ll f = INF;
45        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
46            f = min(f, edges[id].cap - edges[id].flow);
47            mincost += edges[id].cost;
48        }
49        for (int id = pego[t]; id != -1; id = pego[edges[id].u]) {
50            edges[id].flow += f;
51            edges[id ^ 1].flow -= f;
52        }
53        maxflow += f;

```

```

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

```

9.9 HLD

9.9.1 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.

Código: `hld_edge.cpp`

```

1 const int N = 3e5 + 5;
2
3 vector<pair<int, ll>> adj[N];
4
5 namespace HLD {
6     int t, sz[N], pos[N], par[N], head[N];
7     SegTree seg; // por padrão, a HLD está codada para usar a SegTree lazy,
8                  // mas pode usar qualquer estrutura de dados aqui

```

```

9   void dfs_sz(int u, int p = -1) {
10    sz[u] = 1;
11    for (int i = 0; i < (int)adj[u].size(); i++) {
12      auto &[v, w] = adj[u][i];
13      if (v != p) {
14        dfs_sz(v, u);
15        sz[u] += sz[v];
16        if (sz[v] > sz[adj[u][0].first] || adj[u][0].first == p)
17          swap(adj[u][0], adj[u][i]);
18      }
19    }
20  }
21  void dfs_hld(int u, int p = -1) {
22    pos[u] = t++;
23    for (auto &[v, w] : adj[u]) {
24      if (v != p) {
25        par[v] = u;
26        head[v] = (v == adj[u][0].first ? head[u] : v);
27        dfs_hld(v, u);
28      }
29    }
30  }
31  void build_hld(int u) {
32    dfs_sz(u);
33    t = 0, par[u] = u, head[u] = u;
34    dfs_hld(u);
35  }
36  void build(int n, int root) {
37    build_hld(root);
38    vector<ll> aux(n, seg.neutral);
39    for (int u = 0; u < n; u++) {
40      for (auto &[v, w] : adj[u])
41        if (u == par[v]) aux[pos[v]] = w;
42    }
43    seg.build(aux);
44  }
45  ll query(int u, int v) {
46    if (u == v) return seg.neutral;
47    if (pos[u] > pos[v]) swap(u, v);
48    if (head[u] == head[v]) {
49      return seg.query(pos[u] + 1, pos[v]);
50    } else {
51      ll qv = seg.query(pos[head[v]], pos[v]);
52      ll qu = query(u, par[head[v]]);
53      return seg.merge(qu, qv);
54    }
55  }

```

```

56  ll query_subtree(int u) {
57    if (sz[u] == 1) return seg.neutral;
58    return seg.query(pos[u] + 1, pos[u] + sz[u] - 1);
59  }
60  // a flag repl diz se o update é de soma ou de replace
61  void update(int u, int v, ll k, bool repl) {
62    if (u == v) return;
63    if (pos[u] > pos[v]) swap(u, v);
64    if (head[u] == head[v]) {
65      seg.update(pos[u] + 1, pos[v], k, repl);
66    } else {
67      seg.update(pos[head[v]], pos[v], k, repl);
68      update(u, par[head[v]], k, repl);
69    }
70  }
71  void update_subtree(int u, ll k, bool repl) {
72    if (sz[u] == 1) return;
73    seg.update(pos[u] + 1, pos[u] + sz[u] - 1, k, repl);
74  }
75  int lca(int u, int v) {
76    if (pos[u] > pos[v]) swap(u, v);
77    return head[u] == head[v] ? u : lca(u, par[head[v]]);
78  }
79 }

```

9.9.2 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.

Código: hld.cpp

```

1 const int N = 3e5 + 5;
2
3 vector<int> adj[N];
4
5 namespace HLD {
6     int t, sz[N], pos[N], par[N], head[N];
7     SegTree seg; // por padrão, a HLD está codada para usar a SegTree lazy,
8                 // mas pode usar qualquer estrutura de dados aqui
9     void dfs_sz(int u, int p = -1) {
10         sz[u] = 1;
11         for (int &v : adj[u]) {
12             if (v != p) {
13                 dfs_sz(v, u);
14                 sz[u] += sz[v];
15                 if (sz[v] > sz[adj[u][0]] || adj[u][0] == p) swap(v, adj[u][0]);
16             }
17         }
18     }
19     void dfs_hld(int u, int p = -1) {
20         pos[u] = t++;
21         for (int v : adj[u]) {
22             if (v != p) {
23                 par[v] = u;
24                 head[v] = (v == adj[u][0] ? head[u] : v);
25                 dfs_hld(v, u);
26             }
27         }
28     }
29     void build_hld(int u) {
30         dfs_sz(u);
31         t = 0, par[u] = u, head[u] = u;
32         dfs_hld(u);
33     }
34     void build(vector<ll> v, int root) { // pra buildar com valores nos nodos
35         build_hld(root);
36         vector<ll> aux(v.size());
37         for (int i = 0; i < (int)v.size(); i++) aux[pos[i]] = v[i];
38         seg.build(aux);
39     }
40     void build(int n, int root) { // pra buildar com neutro nos nodos
41         build(vector<ll>(n, seg.neutral), root);
42     }
43     void build(ll *bg, ll *en, int root) { // pra buildar com array de C

```

```

44         build(vector<ll>(bg, en), root);
45     }
46     ll query(int u, int v) {
47         if (pos[u] > pos[v]) swap(u, v);
48         if (head[u] == head[v]) {
49             return seg.query(pos[u], pos[v]);
50         } else {
51             ll qv = seg.query(pos[head[v]], pos[v]);
52             ll qu = query(u, par[head[v]]);
53             return seg.merge(qu, qv);
54         }
55     }
56     ll query_subtree(int u) { return seg.query(pos[u], pos[u] + sz[u] - 1); }
57     // a flag repl diz se o update é de soma ou de replace
58     void update(int u, int v, ll k, bool repl) {
59         if (pos[u] > pos[v]) swap(u, v);
60         if (head[u] == head[v]) {
61             seg.update(pos[u], pos[v], k, repl);
62         } else {
63             seg.update(pos[head[v]], pos[v], k, repl);
64             update(u, par[head[v]], k, repl);
65         }
66     }
67     void update_subtree(int u, ll k, bool repl) {
68         seg.update(pos[u], pos[u] + sz[u] - 1, k, repl);
69     }
70     int lca(int u, int v) {
71         if (pos[u] > pos[v]) swap(u, v);
72         return head[u] == head[v] ? u : lca(u, par[head[v]]);
73     }
74 }

```

9.10 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 $\mathcal{O}(N \cdot \log N + N \cdot \log M)$.

Código: inverse_graph.cpp

```

1 set<int> nodes;
2 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 }
```

9.11 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`.

Código: 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 }
```

9.12 Kruskal

Algoritmo 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.

Código: kruskal.cpp

```

1 vector<tuple<int, int, int>> edges; // {u, v, w}
2
3 void kruskal(int n) {
4     DSU dsu(n); // DSU da seção Estruturas de Dados
5
6     sort(edges.begin(), edges.end(), [](auto a, auto b) {
7         return get<2>(a) < get<2>(b);
8     });
9
10    for (auto [u, v, w] : edges) {
11        if (dsu.unite(u, v)) {
12            // edge u-v is in the MST
13        }
14    }
15 }
```

9.13 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 $\mathcal{O}(1)$.

Código: lca.cpp

```

1 const int N = 5e5 + 5;
2 int timer, tin[N];
3 vector<int> adj[N];
4 vector<pair<int, int>> prof;
5
6 struct SparseTable {
7     int n, LG;
8     using T = pair<int, int>;
9     vector<vector<T>> st;
10    T merge(T a, T b) { return min(a, b); }
11    const T neutral = {INT_MAX, -1};
12    void build(const vector<T> &v) {
13        n = (int)v.size();
14        LG = 32 - __builtin_clz(n);
15        st = vector<vector<T>>(LG, vector<T>(n));
16        for (int i = 0; i < n; i++) st[0][i] = v[i];
17        for (int i = 0; i < LG - 1; i++)
18            for (int j = 0; j + (1 << i) < n; j++)
19                st[i + 1][j] = merge(st[i][j], st[i][j + (1 << i)]);
20    }
21    T query(int l, int r) {
22        if (l > r) return neutral;
23        int i = 31 - __builtin_clz(r - l + 1);
24        return merge(st[i][l], st[i][r - (1 << i) + 1]);
25    }
26 } st_lca;
27
28 void et_dfs(int u, int p, int h) {
29     tin[u] = timer++;
30     prof.emplace_back(h, u);
31     for (int v : adj[u]) {
32         if (v != p) {
33             et_dfs(v, u, h + 1);
34             prof.emplace_back(h, u);
35         }
36     }
37     timer++;
38 }
39
40 int lca(int u, int v) {
41     int l = tin[u], r = tin[v];
42     if (l > r) swap(l, r);
43     return st_lca.query(l, r).second;
44 }
```

```

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

```

9.14 Matching

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

Código: `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 }

```

9.15 Pontes

9.15.1 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.

Código: `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     // Opcão 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 // Opcão 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 }
```

9.15.2 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.

Código: 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 }
```

9.16 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

```

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(int u, int p = -1) {
7     low[u] = tin[u] = ++timer;
8     int child = 0;
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(v, u);
14             low[u] = min(low[u], low[v]);
15             if (p != -1 && low[v] >= tin[u]) {
16                 // vertice u eh um ponto de articulacao
17             }
18             child++;
19         }
20     }
21     if (p == -1 && child > 1) {
22         // vertice u eh um ponto de articulacao
23     }
24 }
25
26 void find_articulation_points() {
27     timer = 0;
28     for (int i = 0; i < n; i++) tin[i] = low[i] = 0;
29     for (int i = 0; i < n; i++)
30         if (tin[i] == 0) dfs(i);
31 }
```

9.17 Shortest Paths

9.17.1 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

```

1 const int N = 3e5 + 5;
2 const int INF = 1e9;
3
4 int n;
5 vector<pair<int, int>> adj[N];
6
7 vector<int> bfs01(int s) {
8     vector<int> dist(n, INF);
9     deque<int> q;
10    dist[s] = 0;
11    q.emplace_back(s);
12    while (!q.empty()) {
13        int u = q.front();
14        q.pop_front();
15        for (auto [w, v] : adj[u]) {
16            if (dist[u] + w < dist[v]) {
17                dist[v] = dist[u] + w;
18                if (w == 0) q.push_front(v);
19                else q.push_back(v);
20            }
21        }
22    }
23    return dist;
24 }
```

9.17.2 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

```

1 const int N = 3e5 + 5;
2
3 int n;
4 vector<int> adj[N];
5
6 vector<int> bfs(int s) {
7     vector<int> dist(n, -1);
8     queue<int> q;
9     dist[s] = 0;
10    q.emplace(s);
11    while (!q.empty()) {
12        int u = q.front();
13        q.pop();
14        for (auto v : adj[u]) {
15            if (dist[v] == -1) {
16                dist[v] = dist[u] + 1;
17                q.emplace(v);
18            }
19        }
20    }
21    return dist;
22 }
```

9.17.3 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

```

1 const ll INF = 1e18;
2
3 int n;
4 vector<tuple<int, int, int>> edges;
5
6 vector<ll> bellman_ford(int s) {
7     vector<ll> dist(n, INF);
8     dist[s] = 0;
9     for (int i = 0; i < n; i++) {
10         for (auto [u, v, w] : edges)
11             if (dist[u] < INF) dist[v] = min(dist[v], dist[u] + w);
12     }
13     for (int i = 0; i < n; i++) {
14         for (auto [u, v, w] : edges)
15             if (dist[u] < INF && dist[u] + w < dist[v]) dist[v] = -INF;
16     }
17     // dist[u] = -INF se tem um ciclo negativo que chega em u
18     return dist;
19 }
```

9.17.4 Dial

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

Útil quando o maior peso de uma aresta D não é muito grande (inutilizável se D puder ser até 10^9 , vide a complexidade abaixo).

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

Muito semelhante a uma BFS, mas simula $D + 1$ filas, uma para cada distância a cada passo do algoritmo, ao invés de uma só fila.

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

Codigo: dial.cpp

```

1 const int N = 3e5 + 5;
2 const int INF = 1e9;
3 // D é o maior peso + 1
```

```

4 const int D = 61;
5
6 int n;
7 vector<pair<int, int>> adj[N];
8
9 vector<int> dial(int s) {
10    vector<int> dist(n, INF);
11    vector<basic_string<int>> q(D);
12    dist[s] = 0;
13    q[0].push_back(s);
14    int prev = -1;
15    while (true) {
16        int x = -1;
17        for (int i = 1; i <= D; i++) {
18            int cur = (prev + i) % D;
19            if (!q[cur].empty()) {
20                x = cur;
21                break;
22            }
23        }
24        if (x == -1) break;
25        prev = x;
26        while (!q[x].empty()) {
27            int u = q[x].back();
28            q[x].pop_back();
29            if (x != dist[u] % D) continue;
30            for (auto [w, v] : adj[u]) {
31                int d = dist[u] + w;
32                if (d < dist[v]) {
33                    q[d % D].push_back(v);
34                    dist[v] = d;
35                }
36            }
37        }
38    }
39    return dist;
40 }
```

9.17.5 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

```

1 const int N = 3e5 + 5;
2 const ll INF = 1e18;
3
4 int n;
5 vector<pair<int, int>> adj[N];
6
7 vector<ll> dijkstra(int s) {
8    vector<ll> dist(n, INF);
9    using T = pair<ll, int>;
10   priority_queue<T, vector<T>, greater<>> pq;
11   dist[s] = 0;
12   pq.emplace(dist[s], s);
13   while (!pq.empty()) {
14       auto [d, u] = pq.top();
15       pq.pop();
16       if (d != dist[u]) continue;
17       for (auto [w, v] : adj[u]) {
18           if (dist[v] > d + w) {
19               dist[v] = d + w;
20               pq.emplace(dist[v], v);
21           }
22       }
23   }
24   return dist;
25 }
```

9.17.6 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

```

1 const int N = 3e3 + 5;
2 const ll INF = 1e18;
3 int n;
4
5 ll adj[N][N]; // adj[u][v] = peso da aresta u-v, INF se não existe
6 ll dist[N][N];
7
8 void floydwarshall() {
9     for (int u = 0; u < n; u++)
10        for (int v = 0; v < n; v++) dist[u][v] = adj[u][v];
11    for (int k = 0; k < n; k++) {
12        for (int i = 0; i < n; i++)
13            for (int j = 0; j < n; j++)
14                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
15    }
16 }

```

9.17.7 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

```

1 const int N = 1e4 + 5;
2 const ll INF = 1e18;
3
4 int n;
5 vector<pair<int, int>> adj[N];
6
7 vector<ll> spfa(int s) {
8     vector<ll> dist(n, INF);
9     vector<int> cnt(n, 0);
10    vector<bool> inq(n, false);
11    queue<int> q;
12    q.push(s);
13    inq[s] = true;
14    dist[s] = 0;
15    while (!q.empty()) {
16        int u = q.front();
17        q.pop();
18        inq[u] = false;

```

```

19        for (auto [w, v] : adj[u]) {
20            ll newd = (dist[u] == -INF ? -INF : max(w + dist[u], -INF));
21            if (newd < dist[v]) {
22                dist[v] = newd;
23                if (!inq[v]) {
24                    q.push(v);
25                    inq[v] = true;
26                    cnt[v]++;
27                    if (cnt[v] > n) dist[v] = -INF; // negative cycle
28                }
29            }
30        }
31    }
32    return dist;
33 }

```

9.18 Stoer–Wagner Min Cut

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

O algoritmo de Stoer-Wagner é um algoritmo para resolver o problema de corte mínimo em grafos não direcionados com pesos não negativos. A ideia essencial deste algoritmo é encolher o grafo mesclando os 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

```

1 const int MAXN = 555, INF = 1e9 + 7;
2
3 int n, e, adj[MAXN][MAXN];
4 vector<int> bestCut;
5
6 int mincut() {
7     int bestCost = INF;
8     vector<int> v[MAXN];
9     for (int i = 0; i < n; i++) v[i].assign(1, i);
10    int w[MAXN], sel;
11    bool exist[MAXN], added[MAXN];
12    memset(exist, true, sizeof(exist));

```

```

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

```

7     sort(S.begin(), S.end(), [&](int i, int j) { return bl::tin[i] < bl::tin[j]; });
8     for (int i = 1; i < (int)S.size(); i++) S.emplace_back(bl::lca(S[i - 1], S[i]));
9     sort(S.begin(), S.end(), [&](int i, int j) { return bl::tin[i] < bl::tin[j]; });
10    S.erase(unique(S.begin(), S.end()), S.end());
11    vir_nodes = S;
12    for (auto u : S) vir_tree[u].clear();
13    vector<int> stk;
14    for (auto u : S) {
15        while (stk.size() && !bl::ancestor(stk.back(), u)) stk.pop_back();
16        if (stk.size()) vir_tree[stk.back()].emplace_back(u);
17        stk.emplace_back(u);
18    }
19 }
```

9.19 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](#).

Código: virtual_tree.cpp

```

1 const int N = 3e5 + 5;
2 #warning nao esquece de copiar o código de LCA
3 vector<int> vir_tree[N];
4 vector<int> vir_nodes;
5
6 void build_virtual_tree(vector<int> S) {
```

Capítulo 10

Estruturas de Dados

10.1 Disjoint Set Union

10.1.1 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 contêm os elementos dados, retornando `true` se eles estavam em conjuntos diferentes e `false` caso contrário.

Codigo: dsu.cpp

```
1 struct DSU {
2     vector<int> par, sz;
3     void build(int n) {
4         par.assign(n, 0);
5         iota(par.begin(), par.end(), 0);
6         sz.assign(n, 1);
7     }
8     int find(int a) { return a == par[a] ? a : par[a] = find(par[a]); }
9     bool unite(int a, int b) {
10         a = find(a), b = find(b);
11         if (a == b) return false;
12         if (sz[a] < sz[b]) swap(a, b);
13         par[b] = a;
14         sz[a] += sz[b];
15         return true;
16     }
17 }
```

```
16     }
17 }
```

10.1.2 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

```
1 struct Bipartite_DSU {
2     vector<int> par, sz, c, bip;
3     bool all_bipartite;
4     void build(int n) {
5         par.assign(n, 0);
6         iota(par.begin(), par.end(), 0);
7         sz.assign(n, 1);
8         c.assign(n, 0);
9         bip.assign(n, 1);
10        all_bipartite = 1;
11    }
12 }
```

```

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

```

10.1.3 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 revertê-las.

Código: `rollback_dsu.cpp`

```

1 struct Rollback_DSU {
2     vector<int> par, sz;
3     stack<stack<pair<int &, int>>> changes;
4     void build(int n) {
5         par.assign(n, 0);
6         sz.assign(n, 1);

```

```

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

```

10.1.4 DSU Rollback Bipartido

DSU com rollback e bipartido.

Código: `bipartite_rollback_dsu.cpp`

```

1 struct BipartiteRollback_DSU {
2     vector<int> par, sz, c, bip;
3     int all_bipartite;
4     stack<stack<pair<int &, int>>> changes;
5     void build(int n) {
6         par.assign(n, 0);
7         iota(par.begin(), par.end(), 0);
8         sz.assign(n, 1);
9         c.assign(n, 0);

```

```

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

```

10.1.5 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.

Código: `offline_dsu.cpp`

```

1 struct Offline_DSU : BipartiteRollback_DSU {
2     int time;
3     void build(int n) {
4         BipartiteRollback_DSU::build(n);
5         time = 0;
6     }
7     struct query {
8         int type, a, b;
9     };
10    vector<query> queries;
11    void askConnect(int a, int b) {
12        if (a > b) swap(a, b);
13        queries.push_back({0, a, b});
14        time++;
15    }
16    void askBipartite(int a) {
17        queries.push_back({1, a, -1});
18        time++;
19    }
20    void askAllBipartite() {
21        queries.push_back({2, -1, -1});
22        time++;
23    }
24    void addEdge(int a, int b) {
25        if (a > b) swap(a, b);
26        queries.push_back({3, a, b});
27        time++;
28    }
29    void removeEdge(int a, int b) {
30        if (a > b) swap(a, b);
31        queries.push_back({4, a, b});
32        time++;
33    }
34    vector<vector<pair<int, int>>> lazy;
35    void update(int l, int r, pair<int, int> edge, int u, int L, int R) {
36        if (R < l || L > r) return;
37        if (L >= l && R <= r) {
38            lazy[u].push_back(edge);

```

```

39         return;
40     }
41     int mid = (L + R) / 2;
42     update(l, r, edge, 2 * u, L, mid);
43     update(l, r, edge, 2 * u + 1, mid + 1, R);
44 }
45 void dfs(int u, int L, int R, vector<int> &ans) {
46     if (L > R) return;
47     checkpoint();
48     for (auto [a, b] : lazy[u]) unite(a, b);
49     if (L == R) {
50         auto [type, a, b] = queries[L];
51         if (type == 0) ans.push_back(find(a) == find(b));
52         else if (type == 1) ans.push_back(bipartite(a));
53         else if (type == 2) ans.push_back(all_bipartite);
54     } else {
55         int mid = (L + R) / 2;
56         dfs(2 * u, L, mid, ans);
57         dfs(2 * u + 1, mid + 1, R, ans);
58     }
59     rollback();
60 }
61 vector<int> solve() {
62     lazy.assign(4 * time, {});
63     map<pair<int, int>, int> edges;
64     for (int i = 0; i < time; i++) {
65         auto [type, a, b] = queries[i];
66         if (type == 3) {
67             edges[{a, b}] = i;
68         } else if (type == 4) {
69             update(edges[{a, b}], i, {a, b}, 1, 0, time - 1);
70             edges.erase({a, b});
71         }
72     }
73     for (auto [k, v] : edges) update(v, time - 1, k, 1, 0, time - 1);
74     vector<int> ans;
75     dfs(1, 0, time - 1, ans);
76     return ans;
77 }
78 };

```

10.2 Fenwick Tree

10.2.1 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 .

Código: fenwick_tree.cpp

```

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

```

```
30     }
31 };
```

10.2.2 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 $\mathcal{O}(2^k \cdot k \cdot \log^k n)$.

Código: kd_fenwick_tree.cpp

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

10.3 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, $a[L], a[L+1], \dots, a[R] \rightarrow a[R], a[R-1], \dots, a[L]$.

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.

Código: implicit_treap.cpp

```
1 mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
2 namespace imp_treap {
3     using T = ll; // mudar pra int se nao precisar pra melhorar a performance
4     T merge(T a, T b) { return a + b; }
5     T neutral = 0;
6     struct node_info {
7         node_info *l, *r;
8         int y, size;
9         T val, acc, add;
10        bool rev;
11        node_info() { }
12        node_info(T _val)
13            : l(0), r(0), y(rng()), size(0), val(_val), acc(0), add(0), rev(false) { }
14    };
15    using node = node_info *;
16    node root = 0;
17    inline int size(node t) { return t ? t->size : 0; }
18    inline T acc(node t) { return t ? t->acc : 0; }
19    inline bool rev(node t) { return t ? t->rev : false; }
20    inline void push(node t) {
```

```

21     if (!t) return;
22     if (rev(t)) {
23         t->rev = false;
24         swap(t->l, t->r);
25         if (t->l) t->l->rev ^= 1;
26         if (t->r) t->r->rev ^= 1;
27     }
28     t->acc += t->add * size(t);
29     // t->acc += t->add se for RMQ
30     t->val += t->add;
31     if (t->l) t->l->add += t->add;
32     if (t->r) t->r->add += t->add;
33     t->add = 0;
34 }
35 inline void pull(node t) {
36     if (t) {
37         push(t->l), push(t->r);
38         t->size = size(t->l) + size(t->r) + 1;
39         t->acc = merge(t->val, merge(acc(t->l), acc(t->r)));
40     }
41 }
42 void merge(node &t, node L, node R) {
43     push(L), push(R);
44     if (!L || !R) {
45         t = L ? L : R;
46     } else if (L->y > R->y) {
47         merge(L->r, L->r, R);
48         t = L;
49     } else {
50         merge(R->l, L, R->l);
51         t = R;
52     }
53     pull(t);
54 }
55 void split(node t, int pos, node &L, node &R, int add = 0) {
56     if (!t) {
57         L = R = nullptr;
58     } else {
59         push(t);
60         int imp_key = add + size(t->l);
61         if (pos <= imp_key) {
62             split(t->l, pos, L, t->l, add);
63             R = t;
64         } else {
65             split(t->r, pos, t->r, R, imp_key + 1);
66             L = t;
67         }
68     }
69     pull(t);
70 }
71 inline void insert(node to, int pos) {
72     node L, R;
73     split(root, pos, L, R);
74     merge(L, L, to);
75     merge(root, L, R);
76 }
77 bool remove(node &t, int pos, int add = 0) {
78     if (!t) return false;
79     push(t);
80     int imp_key = add + size(t->l);
81     if (pos == imp_key) {
82         node me = t;
83         merge(t, t->l, t->r);
84         delete me;
85         return true;
86     }
87     bool ok;
88     if (pos < imp_key) ok = remove(t->l, pos, add);
89     else ok = remove(t->r, pos, imp_key + 1);
90     pull(t);
91     return ok;
92 }
93 inline T query(int l, int r) {
94     if (l > r) return neutral;
95     node L1, L2, R1, R2;
96     split(root, r + 1, L1, R1);
97     split(L1, l, L2, R2);
98     T ans = acc(R2);
99     merge(L1, L2, R2);
100    merge(root, L1, R1);
101    return ans;
102 }
103 inline void update_sum(int l, int r, T val) {
104     if (l > r) return;
105     node L1, L2, R1, R2;
106     split(root, r + 1, L1, R1);
107     split(L1, l, L2, R2);
108     assert(R2);
109     R2->add += val;
110     merge(L1, L2, R2);
111     merge(root, L1, R1);
112 }
113 inline void reverse(int l, int r) {
114     if (l > r) return;

```

```

115     node L1, L2, R1, R2;
116     split(root, r + 1, L1, R1);
117     split(L1, l, L2, R2);
118     R2->rev ^= 1;
119     merge(L1, L2, R2);
120     merge(root, L1, R1);
121 }
122 inline void insert(int pos, int val) { insert(new node_info(val), pos); }
123 inline bool remove(int pos) { return remove(root, pos); }
124 }
```

10.4 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`

Código: `interval_tree.cpp`

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4
5 struct interval {
6     long long lo, hi, id;
7     bool operator<(const interval &i) const {
8         return tuple(lo, hi, id) < tuple(i.lo, i.hi, i.id);
9     }
10 };
11 const long long INF = 1e18;
12
13 template <class CNI, class NI, class Cmp_Fn, class Allocator>
14 struct intervals_node_update {
15     typedef long long metadata_type;
```

```

17     int sz = 0;
18     virtual CNI node_begin() const = 0;
19     virtual CNI node_end() const = 0;
20     inline vector<int> overlaps(const long long l, const long long r) {
21         queue<CNI> q;
22         q.push(node_begin());
23         vector<int> vec;
24         while (!q.empty()) {
25             CNI it = q.front();
26             q.pop();
27             if (it == node_end()) continue;
28             if (r >= (*it)->lo && l <= (*it)->hi) vec.push_back((*it)->id);
29             CNI l_it = it.get_l_child();
30             long long l_max = (l_it == node_end()) ? -INF : l_it.get_metadata();
31             if (l_max >= l) q.push(l_it);
32             if ((*it)->lo <= r) q.push(it.get_r_child());
33         }
34         return vec;
35     }
36     inline void operator()(NI it, CNI end_it) {
37         const long long l_max =
38             (it.get_l_child() == end_it) ? -INF : it.get_l_child().get_metadata();
39         const long long r_max =
40             (it.get_r_child() == end_it) ? -INF : it.get_r_child().get_metadata();
41         const_cast<long long &>(it.get_metadata()) = max((*it)->hi, max(l_max, r_max));
42     }
43 };
44 typedef tree<interval, null_type, less<interval>, rb_tree_tag, intervals_node_update>
45     interval_tree;
```

10.5 LiChao Tree

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

Para retornar o mínimo deve-se inserir o negativo da função ($g(x) = -ax - b$) 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.

Código: lichao_tree.cpp

```

1 const ll INF = ll(2e18) + 10;
2 struct Line {
3     ll a, b;
4     Line(ll a_ = 0, ll b_ = -INF) : a(a_), b(b_) { }
5     ll operator()(ll x) { return a * x + b; }
6 };
7
8 template <ll MINL = ll(-1e9 - 5), ll MAXR = ll(1e9 + 5>
9 struct LichaoTree {
10     vector<Line> tree;
11     vector<int> L, R;
12
13     int newnode() {
14         tree.push_back(Line());
15         L.push_back(0);
16         R.push_back(0);
17         return int(tree.size()) - 1;
18     }
19
20     LichaoTree() {
21         newnode();
22         newnode();
23     }
24
25     int lc(int p, bool create = false) {
26         if (create && L[p] == 0) L[p] = newnode();
27         return L[p];
28     }
29
30     int rc(int p, bool create = false) {
31         if (create && R[p] == 0) R[p] = newnode();
32         return R[p];
33     }
34
35     void insert(Line line, int p = 1, ll l = MINL, ll r = MAXR) {
36         if (p == 0) return;
37         ll mid = l + (r - 1) / 2;

```

```

38         bool bl = line(l) > tree[p](l);
39         bool bm = line(mid) > tree[p](mid);
40         bool br = line(r) > tree[p](r);
41         if (bm) swap(tree[p], line);
42         if (line.b == -INF) return;
43         if (bl != bm) insert(line, lc(p, true), l, mid - 1);
44         else if (br != bm) insert(line, rc(p, true), mid + 1, r);
45     }
46
47     ll query(int x, int p = 1, ll l = MINL, ll r = MAXR) {
48         if (p == 0 || tree[p](x) == -INF || (l > r)) return -INF;
49         if (l == r) return tree[p](x);
50         ll mid = l + (r - 1) / 2;
51         if (x < mid) return max(tree[p](x), query(x, lc(p), l, mid - 1));
52         else return max(tree[p](x), query(x, rc(p), mid + 1, r));
53     }
54 };

```

10.6 Merge Sort Tree

10.6.1 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

```

1 template <typename T = int>
2 struct MergeSortTree {
3     vector<vector<T>> tree;
4     int n;
5     int lc(int u) { return u << 1; }
6     int rc(int u) { return u << 1 | 1; }
7     void build(int u, int l, int r, const vector<T> &a) {
8         tree[u] = vector<T>(r - l + 1);
9         if (l == r) {
10             tree[u][0] = a[l];
11             return;
12         }
13         int mid = (l + r) >> 1;
14         build(lc(u), l, mid, a);
15         build(rc(u), mid + 1, r, a);
16         merge(
17             tree[lc(u)].begin(),
18             tree[lc(u)].end(),
19             tree[rc(u)].begin(),
20             tree[rc(u)].end(),
21             tree[u].begin()
22         );
23     }
24     void build(const vector<T> &a) { // para construir com vector
25         n = (int)a.size();
26         tree.assign(4 * n, vector<T>());
27         build(1, 0, n - 1, a);
28     }
29     void build(T *bg, T *en) { // para construir com array de C
30         build(vector<T>(bg, en));
31     }
32     int count(int u, int l, int r, int L, int R, int a, int b) {
33         if (l > R || r < L || a > b) return 0;
34         if (l >= L && r <= R) {
35             auto ub = upper_bound(tree[u].begin(), tree[u].end(), b);
36             auto lb = upper_bound(tree[u].begin(), tree[u].end(), a - 1);
37             return (int)(ub - lb);
38         }
39         int mid = (l + r) >> 1;
40         return count(lc(u), l, mid, L, R, a, b) + count(rc(u), mid + 1, r, L, R, a, b);
41     }
42     int count(int l, int r, int a, int b) { return count(1, 0, n - 1, l, r, a, b); }
43     int less(int l, int r, int k) { return count(1, r, tree[1][0], k - 1); }
44     int kth(int l, int r, int k) {
45         int L = 0, R = n - 1;
46         int ans = -1;

```

```

47         while (L <= R) {
48             int mid = (L + R) >> 1;
49             if (count(1, r, tree[1][0], tree[1][mid]) > k) {
50                 ans = mid;
51                 R = mid - 1;
52             } else {
53                 L = mid + 1;
54             }
55         }
56         return tree[1][ans];
57     }
58 };

```

10.6.2 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

```

1 template <typename T = int>
2 struct MergeSortTree {
3     vector<ordered_set<pair<T, int>>> tree;
4     vector<T> v;
5     int n;
6     int lc(int u) { return u << 1; }
7     int rc(int u) { return u << 1 | 1; }
8     void build(int u, int l, int r, const vector<T> &a) {
9         if (l == r) {
10             tree[u].insert({a[l], 1});
11             return;
12         }
13         int mid = (l + r) >> 1;
14         build(lc(u), l, mid, a);
15         build(rc(u), mid + 1, r, a);
16         for (auto x : tree[lc(u)]) tree[u].insert(x);
17         for (auto x : tree[rc(u)]) tree[u].insert(x);
18     }
19     void build(const vector<T> &a) { // para construir com vector
20         n = (int)a.size();
21         v = a;
22         tree.assign(4 * n, ordered_set<pair<T, int>>());

```

```

23     build(1, 0, n - 1, a);
24 }
25 void build(T *bg, T *en) { // para construir com array de C
26     build(vector<T>(bg, en));
27 }
28 int count(int u, int l, int r, int L, int R, int a, int b) {
29     if (l > R || r < L || a > b) return 0;
30     if (l >= L && r <= R) {
31         int ub = (int)tree[u].order_of_key({b + 1, INT_MIN});
32         int lb = (int)tree[u].order_of_key({a, INT_MIN});
33         return ub - lb;
34     }
35     int mid = (l + r) >> 1;
36     return count(lc(u), l, mid, L, R, a, b) + count(rc(u), mid + 1, r, L, R, a, b);
37 }
38 int count(int l, int r, int a, int b) { return count(1, 0, n - 1, l, r, a, b); }
39 int less(int l, int r, int k) { return count(l, r, tree[1].begin()>first, k - 1); }
40 void update(int u, int l, int r, int i, T x) {
41     tree[u].erase({v[i], i});
42     if (l == r) {
43         v[i] = x;
44     } else {
45         int mid = (l + r) >> 1;
46         if (i <= mid) update(lc(u), l, mid, i, x);
47         else update(rc(u), mid + 1, r, i, x);
48     }
49     tree[u].insert({v[i], i});
50 }
51 void update(int i, T x) { update(1, 0, n - 1, i, x); }
52 };

```

10.7 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 vazio é indefinido.

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

Código: op_deque.cpp

```

1 template <typename T, auto OP>
2 struct op_deque {
3     op_stack<T, OP> in, out;
4     void push_back(T x) { in.push(x); }
5     void push_front(T x) { out.push(x); }
6     void work() {
7         op_stack<T, OP> to;
8         bool sw = false;
9         if (in.empty()) sw = true, swap(in, out);
10        int m = in.size();
11        for (int i = 0; i < m / 2; i++) to.push(in.top()), in.pop();
12        while (in.size()) out.push(in.top()), in.pop();
13        while (to.size()) in.push(to.top()), to.pop();
14        if (sw) swap(in, out);
15    }
16    T pop_front() {
17        if (out.empty()) work();
18        T ret = out.top();
19        out.pop();
20        return ret;
21    }
22    T pop_back() {
23        if (in.empty()) work();
24        T ret = in.top();
25        in.pop();
26        return ret;
27    }
28    T get() {
29        if (in.empty()) return out.get();
30        if (out.empty()) return in.get();
31        return OP(in.get(), out.get());
32    }
33 };

```

10.8 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).

Código: op_queue.cpp

```

1 template <typename T, auto OP>
2 struct op_queue {
3     op_stack<T, OP> in, out;
4     void push(T x) { in.push(x); }
5     void pop() {
6         if (out.empty()) {
7             while (!in.empty()) {
8                 out.push(in.top());
9                 in.pop();
10            }
11        }
12        out.pop();
13    }
14    T get() {
15        if (out.empty()) return in.get();
16        if (in.empty()) return out.get();
17        return OP(in.get(), out.get());
18    }
19    T front() {
20        if (out.empty()) return in.bottom();
21        return out.top();
22    }
23    T back() {
24        if (in.empty()) return out.bottom();
25        return in.top();
26    }
27};

```

10.9 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:

```

1 int f(int a, int b) { return a + b; }
2
3 void test() {
4     auto g = [](int a, int b) { return a ^ b; };
5
6     op_stack<int, f> st;
7     op_stack<int, g> st2;
8
9     st.push(1);
10    st.push(1);
11    st2.push(1);
12    st2.push(1);
13    cout << st.get() << endl; // 2
14    cout << st2.get() << endl; // 0
15}

```

Pode ser tanto função normal quanto lambda.

Código: op_stack.cpp

```

1 template <typename T, auto OP>
2 struct op_stack {
3     vector<pair<T, T>> st;
4     T get() { return st.back().second; }
5     T top() { return st.back().first; }
6     T bottom() { return st.front().first; }
7     void push(T x) {
8         auto snd = st.empty() ? x : OP(st.back().second, x);
9         st.push_back({x, snd});
10    }
11    void pop() { st.pop_back(); }
12    bool empty() { return st.empty(); }
13    int size() { return (int)st.size(); }
14};

```

10.10 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`:

```
1 ordered_set<int> X;
2 X.insert(1);
3 X.insert(2);
4 X.insert(4);
```

```
5 X.insert(8);
6 X.insert(16);
7 cout << *X.find_by_order(1) << endl; // 2
8 cout << *X.find_by_order(2) << endl; // 4
9 cout << *X.find_by_order(4) << endl; // 16
10 cout << (end(X) == X.find_by_order(5)) << endl; // true
11 cout << X.order_of_key(-5) << endl; // 0
12 cout << X.order_of_key(1) << endl; // 0
13 cout << X.order_of_key(3) << endl; // 2
14 cout << X.order_of_key(4) << endl; // 2
15 cout << X.order_of_key(400) << endl; // 5
```

Exemplo de uso do `ordered_map`:

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

Código: `ordered_set.cpp`

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

10.11 Segment Tree

10.11.1 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 $mid = (l + r)/2$.

Código: seg_tree.cpp

```

1 struct SegTree {
2     ll merge(ll a, ll b) { return a + b; }
3     const ll neutral = 0;
4     inline int lc(int p) { return p * 2; }
5     inline int rc(int p) { return p * 2 + 1; }
6     int n;
7     vector<ll> t;
8     void build(int p, int l, int r, const vector<ll> &v) {
9         if (l == r) {
10             t[p] = v[l];
11         } else {
12             int mid = (l + r) / 2;
13             build(lc(p), l, mid, v);
14             build(rc(p), mid + 1, r, v);
15             t[p] = merge(t[lc(p)], t[rc(p)]);
16         }
17     }
18     void build(int _n) { // pra construir com tamanho, mas vazia
19         n = _n;
20         t.assign(n * 4, neutral);
21     }
22     void build(const vector<ll> &v) { // pra construir com vector
23         n = (int)v.size();
24         t.assign(n * 4, neutral);
25         build(1, 0, n - 1, v);

```

```

26     }
27     void build(ll *bg, ll *en) { // pra construir com array de C
28         build(vector<ll>(bg, en));
29     }
30     ll query(int p, int l, int r, int L, int R) {
31         if (l > R || r < L) return neutral;
32         if (l >= L && r <= R) return t[p];
33         int mid = (l + r) / 2;
34         auto ql = query(lc(p), l, mid, L, R);
35         auto qr = query(rc(p), mid + 1, r, L, R);
36         return merge(ql, qr);
37     }
38     ll query(int l, int r) { return query(1, 0, n - 1, l, r); }
39     void update(int p, int l, int r, int i, ll x, bool repl = 0) {
40         if (l == r) {
41             if (repl) t[p] = x; // substitui
42             else t[p] += x; // soma
43         } else {
44             int mid = (l + r) / 2;
45             if (i <= mid) update(lc(p), l, mid, i, x, repl);
46             else update(rc(p), mid + 1, r, i, x, repl);
47             t[p] = merge(t[lc(p)], t[rc(p)]);
48         }
49     }
50     void update(int i, ll x, bool repl) { update(1, 0, n - 1, i, x, repl); }
51     void sumUpdate(int i, ll x) { update(i, x, 0); }
52     void setUpdate(int i, ll x) { update(i, x, 1); }
53 } seg;

```

10.11.2 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 $\mathcal{O}(\log n \cdot \log m)$.

Código: seg_tree_2d.cpp

```

1 struct SegTree2D {
2     ll merge(ll a, ll b) { return a + b; }
3     ll neutral = 0;
4     int n, m;
5     vector<vector<ll>> t;
6     void build(int _n, int _m) {
7         n = _n, m = _m;

```

```

8     t.assign(2 * n, vector<ll>(2 * m, neutral));
9     for (int i = 2 * n - 1; i >= n; i--)
10        for (int j = m - 1; j > 0; j--)
11            t[i][j] = merge(t[i][j << 1], t[i][j << 1 | 1]);
12        for (int i = n - 1; i > 0; i--)
13            for (int j = 2 * m - 1; j > 0; j--)
14                t[i][j] = merge(t[i << 1][j], t[i << 1 | 1][j]);
15    }
16    ll inner_query(int idx, int l, int r) {
17        ll res = neutral;
18        for (l += m, r += m + 1; l < r; l >>= 1, r >>= 1) {
19            if (l & 1) res = merge(res, t[idx][l++]);
20            if (r & 1) res = merge(res, t[idx][-r]);
21        }
22        return res;
23    }
24    // query do ponto (a, b) ate o ponto (c, d), retorna neutro se a > c ou b > d
25    ll query(int a, int b, int c, int d) {
26        ll res = neutral;
27        for (a += n, c += n + 1; a < c; a >>= 1, c >>= 1) {
28            if (a & 1) res = merge(res, inner_query(a++, b, d));
29            if (c & 1) res = merge(res, inner_query(--c, b, d));
30        }
31        return res;
32    }
33    void inner_update(int idx, int i, ll x) {
34        auto &c = t[idx];
35        i += m;
36        c[i] = x;
37        for (i >= 1; i > 0; i >>= 1) c[i] = merge(c[i << 1], c[i << 1 | 1]);
38    }
39    void update(int i, int j, ll x) {
40        i += n;
41        inner_update(i, j, x);
42        for (i >= 1; i > 0; i >>= 1) {
43            ll val = merge(t[i << 1][j + m], t[i << 1 | 1][j + m]);
44            inner_update(i, j, val);
45        }
46    }
47 } seg;

```

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

Código: seg_tree_beats.cpp

```

1 const ll INF = 1e18;
2 struct node {
3     ll mi, smi, mx, smx, sum, lazy;
4     int fmi, fmw;
5     node() {
6         mi = smi = INF;
7         mx = smx = -INF;
8         fmi = 0, fmw = 0, sum = 0, lazy = 0;
9     }
10    node(ll val) {
11        mi = mx = sum = val;
12        smi = INF, smx = -INF;
13        fmw = fmi = 1;
14        lazy = 0;
15    }
16 };
17
18 node operator+(node a, node b) {
19     node ret;
20     ret.sum = a.sum + b.sum;
21     if (a.mi == b.mi) {
22         ret.mi = a.mi;
23         ret.fmi = a.fmi + b.fmi;
24         ret.smi = min(a.smi, b.smi);
25     } else if (a.mi < b.mi) {
26         ret.mi = a.mi;
27         ret.fmi = a.fmi;
28         ret.smi = min(a.smi, b.mi);
29     } else {
30         ret.mi = b.mi;
31         ret.fmi = b.fmi;
32         ret.smi = min(b.smi, a.mi);
33     }
34 }
35
36 void update(node &t, int l, int r, ll val) {
37     if (l > r) return;
38     if (l == r) {
39         t = node(val);
40         return;
41     }
42     int m = (l + r) / 2;
43     if (t.lazy > 0) {
44         t.mi = t.smi = t.sum = t.fmi = t.laz

```

```

33     }
34     if (a.mx == b.mx) {
35         ret.mx = a.mx;
36         ret.fmx = a.fmx + b.fmx;
37         ret.smx = max(a.smx, b.smx);
38     } else if (a.mx > b.mx) {
39         ret.mx = a.mx;
40         ret.fmx = a.fmx;
41         ret.smx = max(b.mx, a.smx);
42     } else {
43         ret.fmx = b.fmx;
44         ret.mx = b.mx;
45         ret.smx = max(a.mx, b.smx);
46     }
47     return ret;
48 }
49
50 struct SegBeats {
51     vector<node> t;
52     int n;
53     void build(int _n) { // pra construir com tamanho, mas vazia
54         n = _n;
55         t.assign(n * 4, node());
56     }
57     void build(const vector<ll> &v) { // pra construir com vector
58         n = (int)v.size();
59         t.assign(n * 4, node());
60         build(1, 0, n - 1, v);
61     }
62     void build(ll *bg, ll *en) { // pra construir com array de C
63         build(vector<ll>(bg, en));
64     }
65     inline int lc(int p) { return 2 * p; }
66     inline int rc(int p) { return 2 * p + 1; }
67     node build(int p, int l, int r, const vector<ll> &a) {
68         if (l == r) return t[p] = node(a[l]);
69         int mid = (l + r) >> 1;
70         return t[p] = build(lc(p), l, mid, a) + build(rc(p), mid + 1, r, a);
71     }
72     void pushsum(int p, int l, int r, ll x) {
73         t[p].sum += (r - l + 1) * x;
74         t[p].mi += x;
75         t[p].mx += x;
76         t[p].lazy += x;
77         if (t[p].smi != INF) t[p].smi += x;
78         if (t[p].smx != -INF) t[p].smx += x;
79     }
80     void pushmax(int p, ll x) {
81         if (x <= t[p].mi) return;
82         t[p].sum += t[p].fmi * (x - t[p].mi);
83         if (t[p].mx == t[p].mi) t[p].mx = x;
84         if (t[p].smx == t[p].mi) t[p].smx = x;
85         t[p].mi = x;
86     }
87     void pushmin(int p, ll x) {
88         if (x >= t[p].mx) return;
89         t[p].sum += t[p].fmx * (x - t[p].mx);
90         if (t[p].mi == t[p].mx) t[p].mi = x;
91         if (t[p].smi == t[p].mx) t[p].smi = x;
92         t[p].mx = x;
93     }
94     void pushdown(int p, int l, int r) {
95         if (l == r) return;
96         int mid = (l + r) >> 1;
97         pushsum(lc(p), l, mid, t[p].lazy);
98         pushsum(rc(p), mid + 1, r, t[p].lazy);
99         t[p].lazy = 0;
100    pushmax(lc(p), t[p].mi);
101    pushmax(rc(p), t[p].mi);
102
103    pushmin(lc(p), t[p].mx);
104    pushmin(rc(p), t[p].mx);
105}
106
107    node updatemin(int p, int l, int r, int L, int R, ll x) {
108        if (l > R || r < L || x >= t[p].mx) return t[p];
109        if (l >= L && r <= R && x > t[p].smx) {
110            pushmin(p, x);
111            return t[p];
112        }
113        pushdown(p, l, r);
114        int mid = (l + r) >> 1;
115        t[p] = updatemin(lc(p), l, mid, L, R, x) + updatemin(rc(p), mid + 1, r, L, R,
116        x);
117        return t[p];
118    }
119    node updatemax(int p, int l, int r, int L, int R, ll x) {
120        if (l > R || r < L || x <= t[p].mi) return t[p];
121        if (l >= L && r <= R && x < t[p].smi) {
122            pushmax(p, x);
123            return t[p];
124        }
125        pushdown(p, l, r);
126        int mid = (l + r) >> 1;

```

```

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

```

10.11.4 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 re-alocações 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.

Código: `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 - 1) / 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 - 1) / 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)]);

```

```

45     }
46     void update(ll i, ll x, bool repl) { update(1, MINL, MAXR, i, x, repl); }
47     void sumUpdate(ll i, ll x) { update(i, x, 0); }
48     void setUpdate(ll i, ll x) { update(i, x, 1); }
49 } seg;

```

10.11.5 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.

Código: itseg_tree.cpp

```

1 struct SegTree {
2     ll merge(ll a, ll b) { return a + b; }
3     const ll neutral = 0;
4     inline int lc(int p) { return p * 2; }
5     inline int rc(int p) { return p * 2 + 1; }
6     int n;
7     vector<ll> t;
8     void build(int _n) { // pra construir com tamanho, mas vazia
9         n = _n;
10        t.assign(n * 2, neutral);
11    }
12    void build(const vector<ll> &v) { // pra construir com vector
13        n = (int)v.size();
14        t.assign(n * 2, neutral);
15        for (int i = 0; i < n; i++) t[i + n] = v[i];
16        for (int i = n - 1; i > 0; i--) t[i] = merge(t[lc(i)], t[rc(i)]);
17    }
18    void build(ll *bg, ll *en) { // pra construir com array de C
19        build(vector<ll>(bg, en));
20    }
21    ll query(int l, int r) {
22        ll ansL = neutral, ansR = neutral;
23        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
24            if (l & 1) ansL = merge(ansL, t[l++]);

```

```

25                if (r & 1) ansR = merge(t[--r], ansR);
26            }
27            return merge(ansL, ansR);
28        }
29        void update(int i, ll x, bool replace) {
30            i += n;
31            t[i] = replace ? x : merge(t[i], x);
32            for (i >= 1; i > 0; i >>= 1) t[i] = merge(t[lc(i)], t[rc(i)]);
33        }
34        void sumUpdate(int i, ll x) { update(i, x, 0); }
35        void setUpdate(int i, ll x) { update(i, x, 1); }
36    } seg;

```

10.11.6 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 $mid = (l + r)/2$.

Código: seg_tree_kadane.cpp

```

1 struct SegTree {
2     struct node {
3         ll sum, pref, suf, ans;
4     };
5     const node neutral = {0, 0, 0, 0};
6     node merge(const node &a, const node &b) {
7         return {
8             a.sum + b.sum,

```

```

9      max(a.pref, a.sum + b.pref),
10     max(b.suf, b.sum + a.suf),
11     max({a.ans, b.ans, a.suf + b.pref})
12   };
13 }
14 inline int lc(int p) { return p * 2; }
15 inline int rc(int p) { return p * 2 + 1; }
16 int n;
17 vector<node> t;
18 void build(int p, int l, int r, const vector<ll> &v) {
19   if (l == r) {
20     t[p] = {v[1], v[1], v[1], v[1]};
21   } else {
22     int mid = (l + r) / 2;
23     build(lc(p), l, mid, v);
24     build(rc(p), mid + 1, r, v);
25     t[p] = merge(t[lc(p)], t[rc(p)]);
26   }
27 }
28 void build(int _n) { // pra construir com tamanho, mas vazia
29   n = _n;
30   t.assign(n * 4, neutral);
31 }
32 void build(const vector<ll> &v) { // pra construir com vector
33   n = int(v.size());
34   t.assign(n * 4, neutral);
35   build(1, 0, n - 1, v);
36 }
37 void build(ll *bg, ll *en) { // pra construir com array de C
38   build(vector<ll>(bg, en));
39 }
40 node query(int p, int l, int r, int L, int R) {
41   if (l > R || r < L) return neutral;
42   if (l >= L && r <= R) return t[p];
43   int mid = (l + r) / 2;
44   auto ql = query(lc(p), l, mid, L, R);
45   auto qr = query(rc(p), mid + 1, r, L, R);
46   return merge(ql, qr);
47 }
48 ll query(int l, int r) { return query(1, 0, n - 1, l, r).ans; }
49 void update(int p, int l, int r, int i, ll x) {
50   if (l == r) {
51     t[p] = {x, x, x, x};
52   } else {
53     int mid = (l + r) / 2;
54     if (i <= mid) update(lc(p), l, mid, i, x);
55     else update(rc(p), mid + 1, r, i, x);

```

```

56       t[p] = merge(t[lc(p)], t[rc(p)]);
57     }
58   }
59   void update(int i, ll x) { update(1, 0, n - 1, i, x); }
60 } seg;

```

10.11.7 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 $mid = (l + r)/2$.

Código: `seg_tree_lazy.cpp`

```

1 struct SegTree {
2   ll merge(ll a, ll b) { return a + b; }
3   const ll neutral = 0;
4   int n;
5   vector<ll> t, lazy;
6   vector<bool> replace;
7   inline int lc(int p) { return p * 2; }
8   inline int rc(int p) { return p * 2 + 1; }
9   void push(int p, int l, int r) {
10     if (replace[p]) {
11       t[p] = lazy[p] * (r - l + 1);
12     }
13     if (l != r) {
14       lazy[lc(p)] = lazy[p];
15       lazy[rc(p)] = lazy[p];
16       replace[lc(p)] = true;
17       replace[rc(p)] = true;
18     }
19   }
20   else if (lazy[p] != 0) {
21     t[p] += lazy[p] * (r - l + 1);
22   }
23   if (l != r) {
24     replace[lc(p)] = true;
25     replace[rc(p)] = true;
26   }
27 }
28 void update(int p, int l, int r, int i, ll x) {
29   if (l == r) {
30     t[p] = x;
31   } else {
32     int mid = (l + r) / 2;
33     if (i <= mid) update(lc(p), l, mid, i, x);
34     else update(rc(p), mid + 1, r, i, x);
35   }
36 }
37 ll query(int l, int r) { return t[lc(1)].sum; }
38

```

```

21         lazy[lc(p)] += lazy[p];
22         lazy[rc(p)] += lazy[p];
23     }
24 }
25 replace[p] = false;
26 lazy[p] = 0;
27 }
28 void build(int p, int l, int r, const vector<ll> &v) {
29     if (l == r) {
30         t[p] = v[l];
31     } else {
32         int mid = (l + r) / 2;
33         build(lc(p), l, mid, v);
34         build(rc(p), mid + 1, r, v);
35         t[p] = merge(t[lc(p)], t[rc(p)]);
36     }
37 }
38 void build(int _n) { // pra construir com tamanho, mas vazia
39     n = _n;
40     t.assign(n * 4, neutral);
41     lazy.assign(n * 4, 0);
42     replace.assign(n * 4, false);
43 }
44 void build(const vector<ll> &v) { // pra construir com vector
45     n = (int)v.size();
46     t.assign(n * 4, neutral);
47     lazy.assign(n * 4, 0);
48     replace.assign(n * 4, false);
49     build(1, 0, n - 1, v);
50 }
51 void build(ll *bg, ll *en) { // pra construir com array de C
52     build(vector<ll>(bg, en));
53 }
54 ll query(int p, int l, int r, int L, int R) {
55     push(p, l, r);
56     if (l > R || r < L) return neutral;
57     if (l >= L && r <= R) return t[p];
58     int mid = (l + r) / 2;
59     auto ql = query(lc(p), l, mid, L, R);
60     auto qr = query(rc(p), mid + 1, r, L, R);
61     return merge(ql, qr);
62 }
63 ll query(int l, int r) { return query(1, 0, n - 1, l, r); }
64 void update(int p, int l, int r, int L, int R, ll val, bool repl = 0) {
65     push(p, l, r);
66     if (l > R || r < L) return;
67     if (l >= L && r <= R) {

```

```

68         lazy[p] = val;
69         replace[p] = repl;
70         push(p, l, r);
71     } else {
72         int mid = (l + r) / 2;
73         update(lc(p), l, mid, L, R, val, repl);
74         update(rc(p), mid + 1, r, L, R, val, repl);
75         t[p] = merge(t[lc(p)], t[rc(p)]);
76     }
77 }
78 void update(int l, int r, ll val, bool repl) { update(1, 0, n - 1, l, r, val,
79             repl); }
80 void sumUpdate(int l, int r, ll val) { update(l, r, val, 0); }
81 void setUpdate(int l, int r, ll val) { update(l, r, val, 1); }
82 } seg;

```

10.11.8 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.reserve(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.

Código: `seg_tree_sparse_lazy.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, lazy;
6     vector<int> Lc, Rc;
7     vector<bool> replace;
8     inline int newnode() {
9         t.push_back(neutral);
10        Lc.push_back(-1);
11        Rc.push_back(-1);
12        lazy.push_back(0);
13        replace.push_back(false);
14        return (int)t.size() - 1;

```

```

15     }
16     inline int lc(int p) {
17         if (Lc[p] == -1) Lc[p] = newnode();
18         return Lc[p];
19     }
20     inline int rc(int p) {
21         if (Rc[p] == -1) Rc[p] = newnode();
22         return Rc[p];
23     }
24     SegTree() { newnode(); }
25     void push(int p, ll l, ll r) {
26         if (replace[p]) {
27             t[p] = lazy[p] * (r - l + 1);
28             if (l != r) {
29                 lazy[lc(p)] = lazy[p];
30                 lazy[rc(p)] = lazy[p];
31                 replace[lc(p)] = true;
32                 replace[rc(p)] = true;
33             }
34         } else if (lazy[p] != 0) {
35             t[p] += lazy[p] * (r - l + 1);
36             if (l != r) {
37                 lazy[lc(p)] += lazy[p];
38                 lazy[rc(p)] += lazy[p];
39             }
40         }
41         replace[p] = false;
42         lazy[p] = 0;
43     }
44     ll query(int p, ll l, ll r, ll L, ll R) {
45         push(p, l, r);
46         if (l > R || r < L) return neutral;
47         if (l >= L && r <= R) return t[p];
48         ll mid = l + (r - l) / 2;
49         auto ql = query(lc(p), l, mid, L, R);
50         auto qr = query(rc(p), mid + 1, r, L, R);
51         return merge(ql, qr);
52     }
53     ll query(ll l, ll r) { return query(0, MINL, MAXR, l, r); }
54     void update(int p, ll l, ll r, ll L, ll R, ll val, bool repl) {
55         push(p, l, r);
56         if (l > R || r < L) return;
57         if (l >= L && r <= R) {
58             lazy[p] = val;
59             replace[p] = repl;
60             push(p, l, r);
61         } else {

```

```

62             ll mid = l + (r - 1) / 2;
63             update(lc(p), l, mid, L, R, val, repl);
64             update(rc(p), mid + 1, r, L, R, val, repl);
65             t[p] = merge(t[lc(p)], t[rc(p)]);
66         }
67     }
68     void update(ll l, ll r, ll val, bool repl) { update(0, MINL, MAXR, l, r, val, repl); }
69     void sumUpdate(ll l, ll r, ll val) { update(l, r, val, 0); }
70     void setUpdate(ll l, ll r, ll val) { update(l, r, val, 1); }
71 } seg;

```

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

Código: seg_tree_pa.cpp

```

1 struct SegTree {
2     using ii = pair<ll, ll>;
3     ll merge(ll a, ll b) { return a + b; }
4     const ll neutral = 0;
5     int n;
6     vector<ll> t;
7     vector<ii> lazy;
8     inline int lc(int p) { return p * 2; }
9     inline int rc(int p) { return p * 2 + 1; }
10    void push(int p, int l, int r) {
11        if (lazy[p].second) {
12            auto [a, d] = lazy[p];
13            t[p] += a * (r - l + 1) + d * (r - l) * (r - l + 1) / 2;
14            if (l != r) {
15                int mid = (l + r) / 2;
16                lazy[lc(p)].first += a;
17                lazy[lc(p)].second += d;
18                lazy[rc(p)].first += a + (mid + 1 - l) * d;
19                lazy[rc(p)].second += d;
20            }
21        }
22        lazy[p] = ii(0, 0);

```

```

22     }
23 }
24 void build(int p, int l, int r, const vector<ll> &v) {
25     if (l == r) {
26         t[p] = v[l];
27     } else {
28         int mid = (l + r) / 2;
29         build(lc(p), l, mid, v);
30         build(rc(p), mid + 1, r, v);
31         t[p] = merge(t[lc(p)], t[rc(p)]);
32     }
33 }
34 void build(int _n) { // pra construir com tamanho, mas vazia
35     n = _n;
36     t.assign(n * 4, neutral);
37 }
38 void build(const vector<ll> &v) { // pra construir com vector
39     n = (int)v.size();
40     t.assign(n * 4, neutral);
41     build(1, 0, n - 1, v);
42 }
43 void build(ll *bg, ll *en) { // pra construir com array de C
44     build(vector<ll>(bg, en));
45 }
46 ll query(int p, int l, int r, int L, int R) {
47     push(p, l, r);
48     if (l > R || r < L) return neutral;
49     if (l >= L && r <= R) return t[p];
50     int mid = (l + r) / 2;
51     auto ql = query(lc(p), l, mid, L, R);
52     auto qr = query(rc(p), mid + 1, r, L, R);
53     return merge(ql, qr);
54 }
55 ll query(int l, int r) { return query(1, 0, n - 1, l, r); }
56 void update(int p, int l, int r, int L, int R, ii pa) {
57     push(p, l, r);
58     if (l > R || r < L) return;
59     if (l >= L && r <= R) {
60         auto [a, d] = pa;
61         lazy[p] = ii(a + (l - L) * d, d);
62         push(p, l, r);
63     } else {
64         int mid = (l + r) / 2;
65         update(lc(p), l, mid, L, R, pa);
66         update(rc(p), mid + 1, r, L, R, pa);
67         t[p] = merge(t[lc(p)], t[rc(p)]);
68     }

```

```

69     }
70     void update(int l, int r, ll a0, ll d) { update(1, 0, n - 1, l, r, ii(a0, d)); }
71 } seg;

```

10.11.10 Segment Tree Persistente

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 deixa 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(MAX)`; `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.

Código: `seg_tree_persistent.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, roots;
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     roots.push_back(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, int root = -1) {
34     if (root == -1) root = roots.back();
35     else root = roots[root];
36     return query(root, MINL, MAXR, l, r);
37 }
38 void update(int p, int old, ll l, ll r, ll i, ll x, bool repl) {
39     t[p] = t[old];
40     if (l == r) {
41         if (repl) t[p] = x; // substitui
42         else t[p] += x; // soma
43         return;
44     }
45     ll mid = l + (r - l) / 2;
46     if (i <= mid) {
47         Rc[p] = rc(old);
48         update(lc(p, true), lc(old), l, mid, i, x, repl);
49     } else {
50         Lc[p] = lc(old);
51         update(rc(p, true), rc(old), mid + 1, r, i, x, repl);
52     }
53     t[p] = merge(t[lc(p)], t[rc(p)]);
54 }
55 int update(ll i, ll x, bool repl, int root = -1) {
56     // root é qual versão da segtree vai ser atualizada,
57     // -1 atualiza a ultima root
58     int new_root = newnode();
59     if (root == -1) root = roots.back();
60     else root = roots[root];
61     update(new_root, root, MINL, MAXR, i, x, repl);
62     roots.push_back(new_root);
63     return roots.back();
64 }
65 int sumUpdate(ll i, ll x, int root = -1) { return update(i, x, 0, root); }
66 int setUpdate(ll i, ll x, int root = -1) { return update(i, x, 1, root); }
67 } seg;

```

10.12 Sparse Table

10.12.1 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.

Código: dst.cpp

```

1 struct DisjointSparseTable {
2     int n, LG;
3     vector<vector<ll>> st;
4     ll merge(ll a, ll b) { return a + b; }
5     const ll neutral = 0;
6     void build(const vector<ll> &v) {
7         int sz = (int)v.size();
8         n = 1, LG = 1;
9         while (n < sz) n <= 1, LG++;
10        st = vector<vector<ll>>(LG, vector<ll>(n));
11        for (int i = 0; i < n; i++) st[0][i] = i < sz ? v[i] : neutral;
12        for (int i = 1; i < LG - 1; i++) {
13            for (int j = (1 << i); j < n; j += (1 << (i + 1))) {
14                st[i][j] = st[0][j];
15                st[i][j - 1] = st[0][j - 1];
16                for (int k = 1; k < (1 << i); k++) {
17                    st[i][j + k] = merge(st[i][j + k - 1], st[0][j + k]);
18                    st[i][j - 1 - k] = merge(st[0][j - k - 1], st[i][j - k]);
19                }
20            }
21        }
22    }
23    void build(ll *bg, ll *en) { build(vector<ll>(bg, en)); }
24    ll query(int l, int r) {
25        if (l == r) return st[0][l];
26        int i = 31 - __builtin_clz(l ^ r);
27        return merge(st[i][l], st[i][r]);
28    }
29 } dst;

```

10.12.2 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.

Código: sparse_table.cpp

```

1 struct SparseTable {
2     int n, LG;
3     vector<vector<ll>> st;
4     ll merge(ll a, ll b) { return min(a, b); }
5     const ll neutral = 1e18;
6     void build(const vector<ll> &v) {
7         n = (int)v.size();
8         LG = 32 - __builtin_clz(n);
9         st = vector<vector<ll>>(LG, vector<ll>(n));
10        for (int i = 0; i < n; i++) st[0][i] = v[i];
11        for (int i = 0; i < LG - 1; i++)
12            for (int j = 0; j + (1 << i) < n; j++)
13                st[i + 1][j] = merge(st[i][j], st[i][j + (1 << i)]);
14    }
15    void build(ll *bg, ll *en) { build(vector<ll>(bg, en)); }
16    ll query(int l, int r) {
17        if (l > r) return neutral;
18        int i = 31 - __builtin_clz(r - l + 1);
19        return merge(st[i][l], st[i][r - (1 << i) + 1]);
20    }
21 };

```

10.13 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 $\mathcal{O}(\log N)$.

Código: treap.cpp

```

1 mt19937 rng((uint32_t)chrono::steady_clock::now().time_since_epoch().count());
2 namespace treap {
3     struct node_info {
4         node_info *l, *r;
5         int x, y, size;
6         node_info() { }
7         node_info(int _x) : l(0), r(0), x(_x), y(rng()), size(0) { }
8     };
9     using node = node_info *;
10    node root = 0;
11    bool setify = false;
12    inline int size(node t) { return t ? t->size : 0; }
13    inline void upd_size(node t) {
14        if (t) t->size = size(t->l) + size(t->r) + 1;
15    }
16    void merge(node &t, node L, node R) {
17        if (!L || !R) {
18            t = L ? L : R;
19        } else if (L->y > R->y) {
20            merge(L->r, L->r, R);
21            t = L;
22        } else {
23            merge(R->l, L, R->l);
24            t = R;
25        }
26        upd_size(t);
27    }
28    void split(node t, int x, node &L, node &R) {

```

```

29     if (!t) {
30         L = R = 0;
31     } else if (t->x <= x) {
32         split(t->r, x, t->r, R);
33         L = t;
34     } else {
35         split(t->l, x, L, t->l);
36         R = t;
37     }
38     upd_size(t);
39 }
40 void insert(node &t, node to) {
41     if (!t) {
42         t = to;
43     } else if (to->y > t->y) {
44         split(t, to->x, to->l, to->r);
45         t = to;
46     } else {
47         insert(to->x < t->x ? t->l : t->r, to);
48     }
49     upd_size(t);
50 }
51 bool remove(node &t, int x) {
52     if (!t) return false;
53     if (x == t->x) {
54         node rem = t;
55         merge(t, t->l, t->r);
56         upd_size(t);
57         delete rem;
58         return true;
59     }
60     bool ok = remove(x < t->x ? t->l : t->r, x);
61     upd_size(t);
62     return ok;
63 }
64 bool find(node &t, int x) {
65     return t ? (t->x == x || find(x < t->x ? t->l : t->r, x)) : false;
66 }
67 bool find(int x) { return find(root, x); }
68 inline void insert(int x) {
69     if (setify) {
70         if (find(x)) return;
71     }
72     insert(root, new node_info(x));
73 }
74 inline void remove(int x) { remove(root, x); }
75 }

```

10.14 XOR Trie

Uma Trie que armazena os números em binário (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 $\mathcal{O}(\log X)$.

Código: xor_trie.cpp

```

1 struct XorTrie {
2     const int bits = 30;
3     vector<vector<int>> go;
4     int root = 0, cnt = 1;
5     void build(int n) { go.assign((n + 1) * bits, vector<int>(2, -1)); }
6     void insert(int x) {
7         int v = root;
8         for (int i = bits - 1; i >= 0; i--) {
9             int b = x >> i & 1;
10            if (go[v][b] == -1) go[v][b] = cnt++;
11            v = go[v][b];
12        }
13    }
14    int max_xor(int x) {
15        int v = root;
16        int ans = 0;
17        if (cnt <= 1) return -1;
18        for (int i = bits - 1; i >= 0; i--) {
19            int b = x >> i & 1;
20            int good = go[v][!b];
21            int bad = go[v][b];
22            if (good != -1) {
23                v = good;
24                ans |= 1 << i;
25            } else v = bad;
26        }
27        return ans;
28    }
29    int min_xor(int x) {
30        int flipped = x ^ ((1 << bits) - 1);
31        int query = max_xor(flipped);
32        if (query == -1) return -1;

```

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