

BRUTE  
UDESC

Eliton Machado da Silva, Enzo de Almeida Rodrigues, Eric Grochowicz,  
João Vitor Frölich, João Marcos de Oliveira e Rafael Granza de Mello

7 de janeiro de 2024

# Índice

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Estruturas de Dados</b>                      | <b>6</b> |
| 1.1      | Disjoint Set Union . . . . .                    | 6        |
| 1.1.1    | DSU Completo . . . . .                          | 6        |
| 1.1.2    | DSU Rollback . . . . .                          | 6        |
| 1.1.3    | DSU Simples . . . . .                           | 7        |
| 1.1.4    | DSU Bipartido . . . . .                         | 7        |
| 1.2      | Operation Queue . . . . .                       | 8        |
| 1.3      | Interval Tree . . . . .                         | 8        |
| 1.4      | Segment Tree . . . . .                          | 9        |
| 1.4.1    | Segment Tree Lazy . . . . .                     | 9        |
| 1.4.2    | Segment Tree . . . . .                          | 9        |
| 1.4.3    | Segment Tree 2D . . . . .                       | 10       |
| 1.4.4    | Segment Tree Beats Max And Sum Update . . . . . | 10       |
| 1.4.5    | Segment Tree Beats Max Update . . . . .         | 11       |

ÍNDICE2

1.4.6Segment Tree Esparsa . . . . .

1.4.7Segment Tree Persistente . . . . .

1.4.8Segment Tree Kadani . . . . .

1.5Operation Stack . . . . .

1.6Fenwick Tree . . . . .

1.7LiChao Tree . . . . .

1.8Kd Fenwick Tree . . . . .

1.9Ordered Set . . . . .

1.10MergeSort Tree . . . . .

1.11Sparse Table . . . . .

1.11.1Disjoint Sparse Table . . . . .

1.11.2Sparse Table . . . . .

2Grafos19

2.1Matching . . . . .

2.1.1Hungaro . . . . .

2.2LCA . . . . .

2.3HLD . . . . .

2.4Kruskal . . . . .

2.5Bridge . . . . .

2.6Stoer–Wagner Min Cut . . . . .

2.7Shortest Paths . . . . .

2.7.1Dijkstra . . . . .

|                                       |           |
|---------------------------------------|-----------|
| <i>ÍNDICE</i>                         | 3         |
| 2.7.2 SPFA . . . . .                  | 22        |
| 2.8 Binary Lifting . . . . .          | 22        |
| 2.9 Fluxo . . . . .                   | 23        |
| 2.10 Inverse Graph . . . . .          | 23        |
| 2.11 2 SAT . . . . .                  | 24        |
| 2.12 Graph Center . . . . .           | 24        |
| <b>3 String</b>                       | <b>25</b> |
| 3.1 Aho Corasick . . . . .            | 25        |
| 3.2 Patricia Tree . . . . .           | 25        |
| 3.3 Prefix Function . . . . .         | 26        |
| 3.4 Hashing . . . . .                 | 27        |
| 3.5 Trie . . . . .                    | 27        |
| 3.6 Manacher . . . . .                | 27        |
| 3.7 Lyndon . . . . .                  | 28        |
| 3.8 Suffix Array . . . . .            | 28        |
| <b>4 Paradigmas</b>                   | <b>29</b> |
| 4.1 Mo . . . . .                      | 29        |
| 4.2 Exponenciação de Matriz . . . . . | 30        |
| 4.3 Busca Binaria Paralela . . . . .  | 31        |
| 4.4 Divide and Conquer . . . . .      | 32        |
| 4.5 Busca Ternaria . . . . .          | 32        |

|  |           |
|--|-----------|
| <i>ÍNDICE</i>                              | 4         |
| 4.6 DP de Permutacao . . . . .             | 33        |
| 4.7 Convex Hull Trick . . . . .            | 33        |
| 4.8 All Submasks . . . . .                 | 33        |
| <b>5 Matemática</b>                        | <b>35</b> |
| 5.1 Sum of floor( $n \div i$ ) . . . . .   | 35        |
| 5.2 Primos . . . . .                       | 35        |
| 5.3 NTT . . . . .                          | 36        |
| 5.4 Eliminação Gaussiana . . . . .         | 37        |
| 5.5 GCD . . . . .                          | 37        |
| 5.6 Fatoração . . . . .                    | 38        |
| 5.7 Teorema do Resto Chinês . . . . .      | 39        |
| 5.8 FFT . . . . .                          | 39        |
| 5.9 Exponenciação Modular Rápida . . . . . | 39        |
| 5.10 Totiente de Euler . . . . .           | 40        |
| 5.11 Inverso Modular . . . . .             | 40        |
| <b>6 Theoretical</b>                       | <b>42</b> |
| 6.1 Some Prime Numbers . . . . .           | 42        |
| 6.1.1 Left-Truncatable Prime . . . . .     | 42        |
| 6.1.2 Mersenne Primes . . . . .            | 42        |
| 6.2 C++ constants . . . . .                | 43        |
| 6.3 Linear Operators . . . . .             | 44        |

ÍNDICE

5

6.3.1

Rotate counter-clockwise by  $\theta^\circ$

44

6.3.2

Reflect about the line  $y = mx$

44

6.3.3

Inverse of a 2x2 matrix A

44

6.3.4

Horizontal shear by K

44

6.3.5

Vertical shear by K

44

6.3.6

Change of basis

45

6.3.7

Properties of matrix operations

45

# Capítulo 1

## Estruturas de Dados

### 1.1 Disjoint Set Union

#### 1.1.1 DSU Completo

DSU com capacidade de adicionar e remover vértices.

**EXTREMAMENTE PODEROSO!**

Funciona de maneira off-line, recebendo as operações e dando as respostas das consultas no retorno da função **solve()**

- Complexidade de tempo:  $O(Q * \log(Q) * \log(N))$ ; Onde  $Q$  é o número de consultas e  $N$  o número de nodos

Roda em 0,6ms para  $3 * 10^5$  queries e nodos com printf e scanf.

Possivelmente aguenta  $10^6$  em 3s

#### 1.1.2 DSU Rollback

Desfaz as últimas  $K$  uniões

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

É possível usar um checkpoint, bastando chamar **rollback()** para ir até o último checkpoint.

O rollback não altera a complexidade, uma vez que  $K \leq \text{queries}$ .

### Só funciona sem compressão de caminho

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

#### 1.1.3 DSU Simple

Verifica se dois itens pertencem a um mesmo grupo.

- Complexidade de tempo:  $O(1)$  amortizado.

Une grupos.

- Complexidade de tempo:  $O(1)$  amortizado.

#### 1.1.4 DSU Bipartido

DSU para grafo bipartido, é possível verificar se uma aresta é possível antes de adicioná-la.

Para todas as operações:

- Complexidade de tempo:  $O(1)$  amortizado.



## 1.2 Operation Queue

Fila que armazena o resultado do operador dos itens.

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

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

## 1.3 Interval Tree

**Por Rafael Granza de Mello**

Estrutura que trata intersecções de intervalos.

Capaz de retornar todos os intervalos que intersectam  $[L, R]$ . **L e R inclusos**

Contém funções  $\text{insert}(L, R, \text{ID})$ ,  $\text{erase}(L, R, \text{ID})$ ,  $\text{overlaps}(L, R)$  e  $\text{find}(L, R, \text{ID})$ .

É necessário inserir e apagar indicando tanto os limites quanto o ID do intervalo.

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

Podem ser usadas as operações em Set:

- $\text{insert}()$
- $\text{erase}()$
- $\text{upper\_bound}()$
- etc

## 1.4 Segment Tree

### 1.4.1 Segment Tree Lazy

Implementação padrão de Seg Tree com lazy update

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

### 1.4.2 Segment Tree

Implementação padrão de Seg Tree

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

### 1.4.3 Segment Tree 2D

Segment Tree em 2 dimensões.

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

### 1.4.4 Segment Tree Beats Max And Sum Update

Seg Tree que suporta update de maximo, update de soma e query de soma.

Utiliza uma fila de lazy para diferenciar os updates

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

### 1.4.5 Segment Tree Beats Max Update

Seg Tree que suporta update de maximo e query de soma

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

### 1.4.6 Segment Tree Esparsa

Consultas e atualizações em intervalos.

#### Seg Tree

Implementação padrão de Seg Tree

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

#### Seg Tree Lazy

Implementação padrão de Seg Tree com lazy update

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

### Sparse Seg Tree

Seg Tree Esparsa:

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

### Persistent Seg Tree

Seg Tree Esparsa com histórico de Updates:

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

### Seg Tree Beats

Seg Tree que suporta update de maximo e query de soma

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

### Seg Tree Beats Max and Sum update

Seg Tree que suporta update de maximo, update de soma e query de soma.

Utiliza uma fila de lazy para diferenciar os updates

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

#### 1.4.7 Segment Tree Persistente

Seg Tree Esparsa com histórico de Updates:

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

- Complexidade de tempo (Update em ponto):  $O(\log(N))$
- **Para fazer consulta em um tempo específico basta indicar o tempo na query**

#### 1.4.8 Segment Tree Kadani

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

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

### 1.5 Operation Stack

Pilha que armazena o resultado do operador dos itens.

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

## 1.6 Fenwick Tree

Consultas e atualizações de soma em intervalo.

O vetor precisa obrigatoriamente estar indexado em 1.

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

\* Complexidade de tempo (Consulta em intervalo):  $O(\log(N))$

\* Complexidade de tempo (Update em ponto):  $O(\log(N))$

\* Complexidade de espaço:  $2 * N = O(N)$

## 1.7 LiChao Tree

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

Para retornar o mínimo deve-se inserir o negativo da função e pegar o negativo do resultado.

Está pronta para usar função linear do tipo  $F(x) = mx + b$ .

Funciona para funções com a seguinte propriedade, sejam duas funções  $f(x)$  e  $g(x)$ , uma vez que  $f(x)$  ganha/perde de  $g(x)$ ,  $f(x)$  vai continuar ganhando/perdendo de  $g(x)$ ,

ou seja  $f(x)$  e  $g(x)$  se intersectam apenas uma vez.

\* Complexidade de consulta :  $O(\log(N))$

\* Complexidade de update:  $O(\log(N))$

### LiChao Tree Sparse

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

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

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



## 1.8 Kd Fenwick Tree

### KD Fenwick Tree

Fenwick Tree em K dimensões.

\* Complexidade de update:  $O(\log^k(N))$ .

\* Complexidade de query:  $O(\log^k(N))$ .

## 1.9 Ordered Set

Set com operações de busca por ordem e índice.

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

- `find_by_order(x)`: retorna o item na posição `x`.
- `order_of_key(k)`: retorna o número de elementos menores que `k`. (o índice de `k`)

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

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

```
using namespace __gnu_pbds;
```

```
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;
```

```
ordered_set X;
```

```
X.insert(1);
```

```
X.insert(2);
```

```
X.insert(4);
```

```
X.insert(8);
```

```

X.insert(16);

cout<<*X.find_by_order(1)<<endl; // 2
cout<<*X.find_by_order(2)<<endl; // 4
cout<<*X.find_by_order(4)<<endl; // 16
cout<<(end(X)==X.find_by_order(6))<<endl; // true

cout<<X.order_of_key(-5)<<endl; // 0
cout<<X.order_of_key(1)<<endl; // 0
cout<<X.order_of_key(3)<<endl; // 2
cout<<X.order_of_key(4)<<endl; // 2
cout<<X.order_of_key(400)<<endl; // 5

```

## 1.10 MergeSort Tree

Árvore que resolve queries que envolvam ordenação em range.

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

### MergeSort Tree com Update Pontual

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

**1 segundo para vetores de tamanho  $3 * 10^5$**

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

## 1.11 Sparse Table

### 1.11.1 Disjoint Sparse Table

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

Pré-processamento em  $O(n \log n)$ .

### 1.11.2 Sparse Table

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

Responde consultas de maneira eficiente em um conjunto de dados estáticos.

Realiza um pré-processamento para diminuir o tempo de cada consulta.

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

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

## Capítulo 2

# Grafos

### 2.1 Matching

#### 2.1.1 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:  $O(n^2 * m)$

### 2.2 LCA

Algoritmo de Lowest Common Ancestor usando EulerTour e Sparse Table

Complexidade de tempo:

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

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

## 2.3 HLD

Técnica usada para otimizar a execução de operações em árvores.

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

## 2.4 Kruskal

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

Utiliza [DSU](../Estruturas%20de%20Dados/DSU/dsu.cpp) - (disjoint set union) - para construir MST - (minimum spanning tree)

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

## 2.5 Bridge

Algoritmo que acha pontes utilizando uma dfs

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

## 2.6 Stoer–Wagner Min Cut

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

O algoritmo de Stoer-Wagner é um algoritmo para resolver o problema de corte mínimo em grafos não direcionados com pesos não negativos. A ideia essencial deste algoritmo é encolher o grafo mesclando os vértices mais intensos até que o grafo contenha apenas dois conjuntos de vértices combinados

Complexidade de tempo:  $O(V^3)$

## 2.7 Shortest Paths

### 2.7.1 Dijkstra

Computa o menor caminho entre nós de um grafo.

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

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

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

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

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

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

### 2.7.2 SPFA

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

Detecta ciclos negativos.

Complexidade de tempo:  $O(|V| * |E|)$

## 2.8 Binary Lifting

Usa uma sparse table para calcular o k-ésimo ancestral de u.

Pode ser usada com o algoritmo de EulerTour para calcular o LCA.

Complexidade de tempo:

- Pré-processamento:  $O(N * \log(N))$
- Consulta do k-ésimo ancestral de u:  $O(\log(N))$
- LCA:  $O(\log(N))$

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

## 2.9 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:  $O(V^2 * E)$ , mas em grafo bipartido a complexidade é  $O(\sqrt{V} * E)$

Útil para grafos com poucas arestas

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

Computa o fluxo máximo com custo mínimo

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

## 2.10 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

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



## 2.11 2 SAT

Resolve problema do 2-SAT.

- Complexidade de tempo (caso médio):  $O(N + M)$

$N$  é o número de variáveis e  $M$  é o número de cláusulas.

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

Em relação ao sinal, tanto faz se 0 liga ou desliga, apenas siga o mesmo padrão.

## 2.12 Graph Center

Encontra o centro e o diâmetro de um grafo

Complexidade de tempo:  $O(N)$

## Capítulo 3

# String

### 3.1 Aho Corasick

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

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

### 3.2 Patricia Tree

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

Implementação PB-DS, extremamente curta e confusa:

- Criar: `patricia_tree pat;`
- Inserir: `pat.insert("sei la");`

- Remover: `pat.erase("sei la");`
- Verificar existência: `pat.find("sei la") != pat.end();`
- Pegar palavras que começam com um prefixo: `auto match = pat.prefix_range("sei");`
- Percorrer `*match*` : `for(auto it = match.first; it != match.second; ++it);`
- Pegar menor elemento lexicográfico `*maior ou igual*` ao prefixo: `*pat.lower_bound("sei");`
- Pegar menor elemento lexicográfico `*maior*` ao prefixo: `*pat.upper_bound("sei");`

TODAS AS OPERAÇÕES EM  $O(|S|)$

NÃO ACEITA ELEMENTOS REPETIDOS

### 3.3 Prefix Function

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

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

#### KMP

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

#### Autômato de KMP

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

#### Prefix Count

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

### 3.4 Hashing

Hashing para testar igualdade de duas strings.

A função **\*range(i, j)\*** retorna o hash da substring nesse range.

Pode ser necessário usar pares de hash para evitar colisões.

\* Complexidade de tempo (Construção):  $O(N)$

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

### 3.5 Trie

Estrutura que guarda informações indexadas por palavra.

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

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

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

### 3.6 Manacher

Encontra todos os palindromos de uma string.

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

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

## 3.7 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:  $O(N)$

### Min Cyclic Shift

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

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

## 3.8 Suffix Array

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

Tambem Constroi a tabela LCP(Longest common prefix).

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

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

## Capítulo 4

# Paradigmas

### 4.1 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).

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

#### Mo com Update

Resolve Queries Complicadas Offline de forma rápida.

Permite que existam **UPDATES PONTUAIS!**

É preciso manter uma estrutura que adicione e remova elementos nas extremidades de um range (tipo janela).

- Complexidade de tempo:  $O(Q * N^{2/3})$

4.2 Exponenciação de Matriz

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

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

É preciso mapear a DP para uma exponenciação de matriz.

• –

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:

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

Exemplo de DP:

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

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

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

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

### 4.3 Busca Binária Paralela

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

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



### 4.4 Divide and Conquer

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

É preciso fazer a função `query(i, j)` que computa o custo do subgrupo

$$i, j$$

.  
\* Complexidade de tempo:  $O(n * k * \log(n) * O(\text{query}))$

#### Divide and Conquer com Query on demand

<!-- \*Read in [English](README.en.md)\* -->

Usado para evitar queries pesadas ou o custo de pré-processamento.

É preciso fazer as funções da estrutura **janela**, eles adicionam e removem itens um a um como uma janela flutuante.

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

### 4.5 Busca Ternaria

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

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

#### Busca Ternária em Espaço Discreto

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

Versão para espaços discretos.

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

## 4.6 DP de Permutacao

Otimização do problema do Caixeiro Viajante

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

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

## 4.7 Convex Hull Trick

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

Só funciona quando as retas são monotônicas. Caso não forem, usar LiChao Tree para guardar as retas

Complexidade de tempo:

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

## 4.8 All Submasks

Percorre todas as submáscaras de uma máscara.

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

## Capítulo 5

# Matemática

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

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

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

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

### 5.2 Primos

Algoritmos relacionados a números primos.

#### Crivo de Eratóstenes

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

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

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

### Miller-Rabin

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

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

### Teste Ingênuo

Computa a primalidade de um número  $N$ .

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

## 5.3 NTT

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

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

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

Constantes finais devem ser menor do que  $10^9$ .

Para constantes entre  $10^9$  e  $10^{18}$  é necessário codar também [big\_convolution](big\_convolution.cpp).

## 5.4 Eliminação Gaussiana

Método de eliminação gaussiana para resolução de sistemas lineares.

- Complexidade de tempo:  $O(n^3)$ .

Dica: Se os valores forem apenas 0 e 1 o algoritmo `[gauss_mod2](gauss_mod2.cpp)` é muito mais rápido.

## 5.5 GCD

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

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

### Algoritmo de Euclides

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

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

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

### Algoritmo de Euclides Estendido

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

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

## 5.6 Fatoração

Algoritmos para fatorar um número.

### Fatoração Simples

Fatora um número  $N$ .

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

### Crivo Linear

Pré-computa todos os fatores primos até  $MAX$ .

Utilizado para fatorar um número  $N$  menor que  $MAX$ .

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

### Fatoração Rápida

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

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

### Pollard-Rho

Descobre um divisor de um número  $N$ .

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

## 5.7 Teorema do Resto Chinês

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

## 5.8 FFT

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

Computa convolução (multiplicação) de polinômios.

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

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

## 5.9 Exponenciação Modular Rápida

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

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



## 5.10 Totiente de Euler

Código para computar o totiente de Euler.

### Totiente de Euler (Phi) para um número

Computa o totiente para um único número  $N$ .

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

### Totiente de Euler (Phi) entre 1 e $N$

Computa o totiente entre 1 e  $N$ .

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

## 5.11 Inverso Modular

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

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

### Modular Inverse

Calculates the modular inverse of  $a$ .

Uses the [exp\_mod](/Matemática/Exponenciação%20Modular%20Rápida/exp\_mod.cpp) algorithm, thus expects MOD to be prime.

\* Time Complexity:  $O(\log(MOD))$ .

\* Space Complexity:  $O(1)$ .

### Modular Inverse by Extended GDC

Calculates the modular inverse of  $a$ .

Uses the `[extended_gcd](/Matemática/GCD/extended_gcd.cpp)` algorithm, thus expects  $MOD$  to be coprime with  $a$ .

Returns  $-1$  if this assumption is broken.

\* Time Complexity:  $O(\log(MOD))$ .

\* Space Complexity:  $O(1)$ .

### **Modular Inverse for 1 to MAX**

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

expects  $MOD$  to be prime.

\* Time Complexity:  $O(MAX)$ .

\* Space Complexity:  $O(MAX)$ .

### **Modular Inverse for all powers**

Let  $b$  be any integer.

Calculates the modular inverse for all powers of  $b$  between  $b^0$  and  $b^{MAX}$ .

Needs you calculate beforehand the modular inverse of  $b$ , for 2 it is always  $(MOD+1)/2$ .

expects  $MOD$  to be coprime with  $b$ .

\* Time Complexity:  $O(MAX)$ .

\* Space Complexity:  $O(MAX)$ .

## Capítulo 6

# Theoretical

### 6.1 Some Prime Numbers

#### 6.1.1 Left-Truncatable Prime

Prime number such that any suffix of it is a prime number

357,686,312,646,216,567,629,137

#### 6.1.2 Mersenne Primes

Prime numbers of the form  $2^m - 1$

| Exponent ( <i>m</i> ) | Decimal representation |
|-----------------------|------------------------|
| 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                   | $1,7 * 10^{38}$        |

6.2 C++ constants

| Constant       | C++ Name   | Value       |
|----------------|------------|-------------|
| $\pi$          | 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... |

## 6.3 Linear Operators

### 6.3.1 Rotate counter-clockwise by $\theta^\circ$

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

### 6.3.2 Reflect about the line $y = mx$

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

### 6.3.3 Inverse of a 2x2 matrix A

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

### 6.3.4 Horizontal shear by K

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

### 6.3.5 Vertical shear by K

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

**6.3.6 Change of basis**

$\vec{a}_\beta$  are the coordinates of vector  $\vec{a}$  in basis  $\beta$ .

$\vec{a}$  are the coordinates of vector  $\vec{a}$  in the canonical basis.

$\vec{b}_1$  and  $\vec{b}_2$  are the basis vectors for  $\beta$ .

$C$  is a matrix that changes from basis  $\beta$  to the canonical basis.

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

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

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

**6.3.7 Properties of matrix operations**

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

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

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

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

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

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

Let  $A$  be an  $N \times N$  matrix:

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