Adilson Luis Jonck Junior

# *Uma biblioteca de Estrutura de Dados para Operações em Intervalos de vetor*

Joinville

2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**Adilson Luis Jonck Junior**

# UMA BIBLIOTECA DE ESTRUTURA DE DADOS PARA OPERAÇÕES EM INTERVALOS DE VETOR

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

**Karina Girardi Roggia**
**Orientadora**

Joinville, Novembro de 2019

# UMA BIBLIOTECA DE ESTRUTURA DE DADOS PARA OPERAÇÕES EM INTERVALOS DE VETOR

Adilson Luis Jonck Junior

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora

---

Karina Girardi Roggia - Doutora (orientadora)

---

Gilmario Barbosa dos Santos - Doutor

---

Rui Jorge Tramontin Junior - Doutor

# Agradecimentos

Agradeço a minha família por todo o apoio incondicional que me deram nesses últimos anos, por todas as conversas e pela confiança que me cederam. Agradeço a todos os meus amigos, onde incluo a professora Karina, pela paciência, pelo auxílio, pelas conversas e por outras coisas que eu nem lembro. Todos vocês fizeram parte dessa jornada comigo, e se cheguei onde cheguei e vi tudo que vi foi consequência de estar sobre os ombros de vocês. Beijo, gente.

*"O problema não é errar, é continuar errando."*

*Desconhecido*

# Resumo

As competições de programação são extremamente relevantes para cursos relacionados à computação. Entre tais eventos, destaca-se a maratona de programação, que é uma competição mundial que ocorre anualmente no Brasil desde 1996. O objetivo deste trabalho é implementar uma biblioteca de estruturas de dados para operações em intervalos de vetor uma vez que tais operações são frequentes em problemas de competições de programação. A literatura em português para este tema é escassa e portanto o trabalho também suprirá esta demanda. No contexto deste trabalho, operações em intervalos de vetor são funções/relações aplicadas a elementos de um intervalo contínuo de um vetor na memória. Para cada estrutura analisada, é indicada a sua fundamentação e suas operações, a implementação em C++, além de problemas associados às respectivas estruturas no cenário competitivo. As estruturas de dados que são abordadas para operações em intervalos neste trabalho foram: *Prefix Sum*, *Sparse Table*, *Fenwick Tree*, *Segment Tree*, *Treap*, em ordem de robustez nas capacidades de suas operações.

**Palavras-chaves: estruturas de dados, operação em intervalos de vetor, maratona de programação**

# Abstract

Programming competitions are extremely relevant to courses related to computer science. A bigger focus is given to the ICPC (Maratona de programação), which is an international competition that occurs anually in Brazil since 1996. The objective of this work is to implement a library of data structures for operations on intervals of arrays, once such operations are frequent in the competitive programming scenario. The literature in portuguese is scarse, so this work will also supply this demand. In the context of this work, operation on intervals of arrays are functions/relations applyed to elements of a continuos interval of an array in memory. For each data structure analyzed, an introduction will be made about its definitions and operations, its implementation in C++, followed by problems related to the respective data structure in the competitive scenario. Amongst the data structures that are used for operations on intervals, the following will be approached: *Prefix Sum*, *Sparse Table*, *Fenwick Tree*, *Segment Tree*, *Treap*, in order of robustness of their operations.

**Keywords: data structure, operation on intervals of arrays, international collegiate programming contest**

# Sumário

# Lista de Figuras

# Lista de Tabelas

# 1 Introdução

A maratona de programação é uma competição mundial que ocorre anualmente no Brasil desde 1996. Nela, os participantes são avaliados de acordo com a sua capacidade de trabalhar em equipe, trabalhar sob pressão, e resolver o maior número de problemas no menor tempo possível, de certa forma simulando uma situação real de profissionais da área. Entre os temas mais frequentes da competição encontram-se grandes áreas da computação, como programação dinâmica, algoritmos gulosos, grafos, *ad hoc*, geometria e afins (DUARTE, 2017). Entre as diversas áreas da computação, é recorrente o uso de estruturas de dados para a fundamentação de um algoritmo. Neste trabalho em específico são abordadas as estruturas de dados para operações em intervalos de vetor, pois são amplamente utilizadas dentro do cenário competitivo (FORISEK, 2018). Entende-se como intervalo de vetor uma sequência contígua de valores na memória, onde esses valores representam algo para o problema.

Apesar do alcance da maratona de programação ter crescido consideravelmente nos últimos anos, a literatura brasileira sobre os tema não seguiu essa proporção. O material disponível é majoritariamente estrangeiro e escasso (TOMMASINI, 2014; KAWAKAMI, 2017). Assim, vê-se a necessidade de fundamentar esses conceitos e unificá-los em um só lugar para a construção de uma biblioteca relevante que possa capacitar um leitor com interesse em programação competitiva sobre a fundamentação dos conceitos das estruturas de dados em intervalos de vetor e suas aplicabilidades, tanto para competições quanto para futuras situações no cenário profissional ou acadêmico (COUTO, 2016).

Inicialmente é dada a fundamentação das estruturas que frequentemente são utilizadas para operações em intervalos de vetor no cenário de competições de programação (HALIM et al., 2013, p 4-5). Em seguida, são elencados problemas relacionados a cada uma das estruturas e o código em C++/C++11 das principais operações das estruturas abordadas, devido a sua popularidade (ARCHIVE, 2019). No desenvolvimento deste trabalho, optamos por uma abordagem de linguagem informal, pois acreditamos que isso colabora com os objetivos propostos de uma literatura mais didática.

As estruturas de dados que são exploradas, por ordem de poder computacional,

são:

**Prefix Sum.** A estrutura mais básica. Trata-se de um vetor de prefixo acumulativo, onde sua função depende do problema. De maneira geral, serve como um acumulador. Não suporta modificações depois de sua construção, nem certas operações como máximo ou mínimo de um intervalo.

**Sparse Table.** Semelhante à estrutura anterior, é estático, ou seja, não pode ser modificado depois de construído. Porém, é um pouco mais flexível com as suas operações. Máximo e mínimo do intervalo, por exemplo, são suportadas.

**Fenwick Tree.** É uma estrutura dinâmica, dessa maneira, aceita modificações depois de sua construção. Não suporta operações como máximo e mínimo, assim como o *Prefix Sum*. Portanto, é limitado nas suas operações. Porém, possui constantes de tempo menores que a *Segment Tree* e a *Treap* devido a sua implementação, assim como um código conciso.

**Segment Tree.** É uma estrutura dinâmica assim como a *Fenwick Tree*. Suporta diversas operações, sendo bem robusta computacionalmente. Porém, possui constantes de implementação altas e um código verboso.

**Treap.** É também uma estrutura dinâmica. Suporta todas as operações básicas possíveis em intervalos de vetor. Contudo, possui constantes de implementação altas, pois se trata de uma árvore estatística. Possui também um código verboso.

Uma abordagem sucinta de suas complexidades de tempo em conjunto com a restrição das operações suportadas é dada pela tabela a seguir.

Tabela 1.1: Tabela com a complexidade de tempo das estruturas abordadas e a restrição das operações suportadas

| | Construção | Consulta | Modificação | Associativa | Reversível |
|---|---|---|---|---|---|
| Prefix Sum | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | Sim | Sim |
| Sparse Table | $\mathcal{O}(n \log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(n \log n)$ | Sim | Não |
| Fenwick Tree | $\mathcal{O}(n \log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | Sim | Sim |
| Segment Tree | $\mathcal{O}(n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | Sim | Não |
| Treap | $\mathcal{O}(n \log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | Sim | Não |

A complexidade de espaço para a construção das estruturas é equivalente à complexidade de tempo, exceto pela *Treap* e *Fenwick Tree* que possuem uma complexidade de espaço $\mathcal{O}(n)$. As complexidades de espaço para as consultas e modificações são constantes.

## 1.1 Conceitos Preliminares

Um vetor é uma coleção de dados sequenciais armazenados na memória, onde esses dados são acessados pelos seus respectivos índices (HALIM et al., 2013, p. 34-35). Seja $V$ um vetor, e $|V| = n$ o seu tamanho (o número de elementos que ele possui). Seja $(i, j)$ um par ordenado de dois inteiros tal que $1 \leq i \leq j \leq n$. Um intervalo ou segmento $(i, j)$ desse vetor é a sequência contínua de todos os elementos de $V_i$ até $V_j$, ou seja:

$$V_{i...j} = [V_i, V_{i+1}, V_{i+2}, ..., V_{j-1}, V_j]$$

Dados dois conjuntos $A$ e $B$, uma função ou operação $f : A \rightarrow B$ é um subconjunto de $A \times B$ onde para todo $a \in A$ existe um único $b \in B$ tal que $(a, b) \in f$. Os conjuntos $A$ e $B$ são definidos como o domínio e o contradomínio, respectivamente (CORMEN et al., 2009, p. 1166-1167). Uma operação $\oplus$ binária é uma função interna em que o domínio é um produto cartesiano e é dita:

- associativa, quando: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$; e
- reversível, se existe uma operação $\otimes$ tal que: $(a \oplus b) \otimes b = a$

Uma função reversível é definida também como uma função que a partir do seu resultado é possível obter os operandos originais (MENEZES, 2010, .p 47). Uma operação binária interna possui um elemento neutro quando:

$$a \oplus e = e \oplus a = a \quad \forall a \ \in A$$

onde $e$ é o elemento neutro da operação $\oplus$, e $A$ é seu domínio. Dessa maneira, uma função ou operação $\oplus$ sobre um intervalo $(i, j)$ de um vetor $V$ é uma operação em intervalo. A função $\oplus$ deve ser aplicada a todos os elementos do intervalo $(i, j)$ do vetor $V$. Uma operação em um intervalo pode ser de dois tipos:

1. Operação de consulta: uma função $\oplus$ deve ser aplicada no intervalo, porém não há modificações no vetor $V$ original;
2. Operação de alteração: uma função $\otimes$ de modificação deve ser aplicada no intervalo. A modificação será efetuada no vetor $V$ original.

Um exemplo de operação de consulta $\oplus$ pode ser:

$$\oplus(a, b) = \sum_{i=a}^{b} V_i$$

onde a operação $\oplus(a, b)$ deve calcular o somatório do conteúdo do vetor $V$ nos índices entre o intervalo $a$ e $b$. Já um exemplo de operação $\psi$ de modificação:

$$\psi(x, a, b) = sub(x, a, b)$$

no qual $sub(x, a, b)$ substitui os valores das posições $a$ até $b$ de $V$ por $x$. Dessa maneira, ao aplicar a operação $\psi(x, a, b)$, faremos com que todos os elementos de $V$ entre os índices $a$ e $b$ tenham valor $x$, ou seja, $V_{a...b} = [x, x, x, ..., x, x]$.

## 1.2 Objetivos

**Objetivo geral**: o objetivo desse trabalho é contribuir com a literatura e a difusão do cenário de programação competitiva no Brasil. Com este fim, é necessário o desenvolvimento didático e com certo rigor matemático de um trabalho relacionado a estruturas de dados para intervalos de vetor, visto que se trata de um tema recorrente em competições de programação. A abordagem e o desenvolvimento das estruturas são dados com um foco para programação competitiva.

**Objetivos específicos**:

- Fundamentar as principais estruturas de dados para intervalos de vetor, as análises de complexidade correspondentes e provas de corretude de suas operações;
- Elencar problemas relacionados a cada uma das estruturas abordadas;
- Implementar o código de cada uma das estruturas apresentadas em C++/C++11.

## 1.3 Estrutura do Texto

O Capítulo 2 aborda a *Prefix Sum*, uma estrutura simples e estática para acumular prefixos aplicados a uma função/relação. No Capítulo 3 abordamos a *Sparse Table*, uma

estrutura estática mais robusta, que suporta um número maior de operações. No Capítulo 4 tratamos a respeito da *Fenwick Tree*, uma estrutura dinâmica capaz de processar as mesmas operações da estrutura *Prefix Sum*, mas com a capacidade de trabalhar com operações de modificação. No Capítulo 5 apresentamos a *Segment Tree*, análoga à *Sparse Table*. Contudo é uma estrutura dinâmica que suporta operações mais robustas, como a modificação em intervalo. No Capítulo 6 abordamos a *Treap*, uma estrutura dinâmica ainda mais poderosa computacionalmente. Possui operações mais robustas que a *Segment Tree*, como a inversão de intervalos, mas sua construção não é determinística. No decorrer dos Capítulos 2 a 6 fazemos o desenvolvimento das estruturas de dados abordadas, sendo separado da seguinte maneira:

**Motivação.** Um problema é introduzido e formalizado, algumas soluções iniciais são abordadas e analisadas para elucidarmos a necessidade da estrutura de dados.

**Definição da Estrutura.** Formalizamos a construção da estrutura e exploramos as suas propriedades. É feita também a análise assintótica da sua construção.

**Operações da Estrutura.** Observamos como podemos aproveitar as propriedades da estrutura para a resolução do problema motivacional.

**Implementação.** Fornecemos uma implementação da estrutura de acordo com as definições abordadas.

**Problema Relacionado.** Levantamos um problema recente utilizado em competições de programação relacionado à estrutura proposta e desenvolvemos sua solução em detalhes.

No Capítulo 7 são feitas as considerações finais acerca do desenvolvimento do trabalho.

# 2 Prefix Sum

Seja $V$ um vetor de inteiros, onde $|V| = n$ e seja $q$ o número de operações de consultas que devem ser realizadas. Cada operação possui dois inteiros $a$ e $b$, tal que $1 \leq a \leq b \leq n$, onde cada operação é feita para calcular:

$$\sum_{i=a}^{b} V_i$$

Ou seja, para cada consulta deve-se calcular o somatório do conteúdo do vetor $V$ nos índices entre o intervalo $a$ e $b$. A primeira solução que pode vir a aparecer é: para cada operação $(a, b)$, percorremos o vetor das posições de $a$ até $b$ acumulando o somatório do conteúdo desse intervalo. Ao final deste processo, o valor acumulado seria a resposta para a operação atual. Podemos observar que, no pior caso, as $q$ operações de consultas podem ser tais que $a = 1$ e $b = n$. Assim, para cada consulta teríamos que percorrer o vetor inteiro para acumular a resposta, apresentando uma complexidade de tempo de $\mathcal{O}(n)$. Como temos $q$ consultas a complexidade de tempo final desse algoritmo seria $\mathcal{O}(nq)$. Na Seção 2.2 veremos como podemos reduzir essa complexidade para $O(n + q)$ utilizando o conceito de *Prefix Sum*.

## 2.1 Definição da estrutura

A estrutura *Prefix Sum* é uma estrutura estática simples, que guarda informações sobre os prefixos de um vetor $V$ qualquer. Dessa forma, é possível saber o valor de uma operação aplicada em um intervalo de $V$ em tempo $\mathcal{O}(1)$, caso tal operação seja associativa e reversível. Por ser uma estrutura estática, ela não suporta operações de modificação no vetor. O *Prefix Sum* de um vetor $V$ de tamanho $n$ é um vetor $P$ tal que $|P| = n$. Cada posição $i$ de $P$ é definida como a soma de todos os valores de $V_1$ até $V_i$, isto é, a soma do

$i$-ésimo prefixo de $V$ (QIU; AKL, 1999):

$$
\begin{aligned}
P_1 &= V_1 \\
P_2 &= V_1 + V_2 \\
P_3 &= V_1 + V_2 + V_3 \\
&... \\
P_n &= V_1 + V_2 + ... + V_{n-1} + V_n
\end{aligned}
$$

Formalmente:

$$P_i = \sum_{j=1}^{i} V_j \quad \forall i \in \{1, ..., n\}$$

um exemplo de sua aplicação para um vetor $V = [7, 1, -1, 0, 10]$, resultaria em um vetor $P = [7, 8, 7, 7, 17]$.

Inicialmente, a construção da estrutura pode parecer ser feita em tempo $\mathcal{O}(n^2)$: para cada posição $i$ de $P$, percorremos todos os índices de 1 até $i$ acumulando a soma dos valores do vetor $V$. Observe que para preencher $P_1$ precisamos percorrer somente 1 elemento, para preencher $P_2$ precisamos percorrer 2 elementos, e assim por diante até o $n$-ésimo elemento. Desta maneira, faremos $1 + 2 + 3 + ... + n$ operações totais para preencher o vetor $P$ de tamanho $n$. Tem-se também que $1 + 2 + 3 + ... + n = n(n + 1)/2$, logo a complexidade de tempo para preencher essa estrutura seria de $n(n+1)/2 = \mathcal{O}(n^2)$. Mas note que não estamos utilizando as propriedades existentes dentro do vetor de prefixo $P$ acerca das posições já calculadas. Sabemos que $P_{i-1} = V_1 + V_2 + ... + V_{i-1}$, e sabemos também que $P_i = V_1 + V_2 + ... + V_{i-1} + V_i$. Desse modo, notamos que $P_i = P_{i-1} + V_i$, gerando a seguinte recorrência:

$$P_i = \begin{cases} V_i & \text{se} \quad i = 1 \\ P_{i-1} + V_i, & \text{senão} \end{cases} \tag{2.1}$$

Portanto, para cada posição $P_i$ do vetor de prefixo podemos preenchê-la em tempo $\mathcal{O}(t)$, onde $t$ é a complexidade de tempo de aplicar a operação $\oplus$. Como temos no máximo $n$ posições, o vetor de prefixo $P$ é criado em tempo $\mathcal{O}(nt)$, como a operação de soma é feita em $O(1)$, a complexidade resultante é de $\mathcal{O}(n)$.

## 2.2 Operações da estrutura

### 2.2.1 Consulta de soma em intervalo

Com o vetor $P$ construído podemos definir a soma de um intervalo de $a$ até $b$ da seguinte maneira:

$$\sum_{i=a}^{b} V_i = \begin{cases} P_b & \text{se} \quad a = 1 \\ P_b - P_{a-1}, & \text{senão} \end{cases} \tag{2.2}$$

note que caso $a = 1$ temos:

$$\sum_{i=1}^{b} V_i = P_b = V_1 + V_2 + V_3 + ... + V_b$$

e caso $a \neq 1$:

$$\sum_{i=a}^{b} V_i = P_b - P_{a-1} = V_b + V_{b-1} + ... + V_{a+1} + V_a$$

Por exemplo, seja $V = [7, 1, -1, 0, 10]$, a *Prefix Sum* correspondente $P = [7, 8, 7, 7, 17]$, e as consultas:

- $a = 1$ e $b = 4$: $\sum_{i=1}^{4} V_i = P_4 = (V_1 + V_2 + V_3 + V_4) = 7$
- $a = 2$ e $b = 4$: $\sum_{i=2}^{4} V_i = P_4 - P_1 = (V_1 + V_2 + V_3 + V_4) - (V_1) = V_2 + V_3 + V_4 = 0$
- $a = 5$ e $b = 5$: $\sum_{i=5}^{5} V_i = P_5 - P_4 = (V_1 + V_2 + V_3 + V_4 + V_5) - (V_1 + V_2 + V_3 + V_4) = V_5 = 10$

Figura 2.1: Exemplo de uma consulta de soma no intervalo $a = 3$ e $b = 5$ de um vetor $V = [7, 1, -1, 0, 10]$


Tem-se, portanto, que a soma no intervalo definido é feita em tempo $\mathcal{O}(1)$. Repare que as operações possíveis com essa estratégia não se limitam simplesmente a operação de soma.

Toda operação $\oplus$ tal que:

$$P_b \otimes P_{a-1} = V_b \oplus V_{b-1} \oplus ... \oplus V_{a+1} \oplus V_a$$

onde $\otimes$ seja a operação reversa à $\oplus$, e $a \leq b$, pode ser utilizada nessa estrutura. Outros exemplos seriam $\oplus = \times$ (logo $\otimes = \div$) e $\oplus = xor$ (com $\otimes = xor$), onde $xor$ é definido como o "ou exclusivo" entre os bits de dois números. A função $\oplus$ deve ser:

- Associativa: como $P_{i+1} = P_i + V_{i+1}$ é equivalente à $P_{i+1} = (P_{i-1} \oplus V_i) \oplus V_{i+1}$, logo a seguinte equação deve ser verdadeira, $P_{i+1} = P_{i-1} \oplus (V_i \oplus V_{i+1})$;
- Reversível: é necessária uma operação reversa $\otimes$ tal que $P_b \otimes P_{a-1} = V_a \oplus V_{a+1} \oplus ... \oplus V_b$.

A complexidade de cada consulta em um intervalo se dá por $\mathcal{O}(r)$, onde $r$ é a complexidade de tempo de aplicar a operação $\otimes$.

Caso seja necessário fazer alguma operação de modificação em um valor $V_i$ teríamos que recalcular $P_j$ para todo índice $j$ a partir da posição $i$, pois cada posição $P_j$, tal que $j \geq i$, depende dos valores de $V_1, V_2, ..., V_i, V_{i+1}, ..., V_j$. Para cada modificação feita no vetor original $V$ teremos que reconstruir o vetor de prefixo $P$, concluindo que cada modificação tem complexidade de tempo $\mathcal{O}(n)$ devido ao tempo de construção do vetor de prefixo.

## 2.3 Implementação

Inicialmente, construímos uma função onde receberemos como parâmetro o vetor $V$ e retornaremos a *Prefix Sum* de $V$. Criamos um vetor zerado $P$ de mesmo tamanho, onde será armazenada a *Prefix Sum*.

```
1 vector<int> build_prefix(vector<int> V){
2     vector<int> P(V.size(), 0);
```

Para preencher o vetor $P$, utilizaremos a recorrência 2.1, onde $P_1 = V_1$, e para as próximas posições, acumularemos as posições anteriores de $P$ com a posição atual de $V$. Em seguida, retornamos o vetor $P$ construído.

```
    P[1] = V[1];
    for(int i = 2; i < V.size(); i++)
        P[i] = P[i-1] + V[i];
    return P;
}
```

Como pode ser observado, a construção da estrutura se dá em tempo $\mathcal{O}(n)$ e espaço $\mathcal{O}(n)$.

### 2.3.1 Consulta de soma em intervalo

Para uma consulta de soma em um intervalo $(a, b)$ de $V$, receberemos como parâmetro o intervalo $(a, b)$ e o vetor $P$ da *Prefix Sum* de $V$ como referência. A recorrência utilizada é a 2.2.

```
int query_sum(int a, int b, vector<int>& P){
    if(a == 1) return P[b];
    else return P[b] - P[a-1];
}
```

A complexidade da operação em tempo e espaço é $\mathcal{O}(1)$.

## 2.4 Problema Relacionado

A estrutura *Prefix Sum* geralmente é utilizada em conjunto com outras técnicas de programação, como busca binária, tabelas hash, grafos, etc. Porém, podemos encontrar alguns problemas que podem ser resolvidos utilizando somente do conhecimento dessa estrutura, como é o caso do problema *Manhattan Crepe Cart* da competição *International CodeJam* da Google (GOOGLE, 2019). Uma versão resumida do problema é a seguinte:

Seja $M$ um número de pontos em um plano cartesiano $n \times n$. Cada ponto pode ser definido como uma tripla $(x, y, d)$, onde $x$ corresponde a sua coordenada no eixo das abscissas, $y$ corresponde a sua coordenada no eixo das ordenadas, e $d$ indica a direção para onde esse ponto está apontando (cima, baixo, esquerda, direita).

- Um ponto $(x_0, y_0)$ apontando para cima, cobre todos os pontos $(x, y)$ onde $y > y_0$
- Um ponto $(x_0, y_0)$ apontando para baixo, cobre todos os pontos $(x, y)$ onde $y < y_0$
- Um ponto $(x_0, y_0)$ apontando para direita, cobre todos os pontos $(x, y)$ onde $x > x_0$
- Um ponto $(x_0, y_0)$ apontando para esquerda, cobre todos os pontos $(x, y)$ onde $x < x_0$

Queremos descobrir a coordenada $(x_{res}, y_{res})$ que está coberta pelo maior número de pontos. Em casos de empate escolheremos a menor coordenada do eixo das abscissas e em seguida das ordenadas. As restrições de valores são $1 \leq M \leq 500$, $1 \leq n \leq 10^5$ e $1 \leq x, y \leq 10^5$.

Começamos tentando encontrar o valor de $x_{res}$: fazemos uma análise dos pontos $(x_0, y_0)$ que apontam para a direita. Para cada ponto $(x, y)$ com $x > x_0$ manteremos o número de pontos $(x_0, y_0)$ que cobrem $(x, y)$. Como manter uma matriz de tamanho $n \times n$ para guardar a quantidade de vezes que um ponto foi coberto por outro é inviável, visto que $n \times n = 10^{10}$ para o pior caso onde $n = 10^5$, teremos que recorrer a outro tipo de estratégia.

Seja $(x_0, y_0)$ um ponto qualquer que aponta para a direita. Tem-se que qualquer ponto $(x, y)$ onde $x > x_0$ estará coberto por $(x_0, y_0)$ e que qualquer ponto $(x_1, y_1)$ onde $x_1 > x$ será coberto também pelos mesmos pontos que cobrem $(x, y)$. Partindo disso, podemos concluir que: as variáveis $y$, $y_0$ e $y_1$ são irrelevantes para as condições atuais do problema, e que o número de pontos que cobrem uma coordenada $x_1$ qualquer são acumulativos em relação às coordenadas $x$, onde $x_1 > x$.

Como os valores do eixo das ordenadas são irrelevantes para as situações atuais, podemos levar em consideração somente os valores do eixo das abscissas. Assim,

teremos apenas $n$ valores possíveis para $x$, onde podemos criar um vetor para guardar essa informação. Em suma, seja $P_x$ o número de vezes que os pontos com coordenadas em abscissas igual a $x$ foram cobertos e $dir_x$ o número de pontos $(x_0, y_0)$ que apontam para a direita, onde $x_0 = x$, logo:

$$P_x = \begin{cases} 0, & \text{se} \quad x = 0 \\ dir_{x-1} + P_{x-1} & \text{senão} \end{cases}$$

Repetiremos a mesma estratégia para os pontos $(x_0, y_0)$ que apontam para a esquerda. Porém, desta vez, como os pontos apontam para a esquerda, precisamos fazer a soma acumulada começando em $n$ e descendo até 0. Além disso, precisamos manter os valores dos pontos $x$ que foram cobertos pelos pontos que apontavam para a direita. Temos, portanto, a seguinte recorrência, onde $esq_x$ representa o número de pontos $(x_0, y_0)$ que apontam para a esquerda, tal que $x_0 = x$.

$$P_x \leftarrow \begin{cases} P_x, & \text{se} \quad x = n \\ P_x + esq_{x+1} + P_{x+1} & \text{senão} \end{cases}$$

tal que a operação $a \leftarrow b$ faz com que o valor de $b$ seja atribuido a $a$. Dessa maneira, $x_{res}$ será o índice $i$ onde $P_i$ é máximo, e em caso de empate escolheremos o menor $i$ entre os máximos. Para encontrar o valor de $y_{res}$ a estratégia é análoga, porém, agora para os pontos $(x_0, y_0)$ que apontam para baixo/cima.

# 3 Sparse Table

Seja $V$ um vetor de inteiros, onde $|V| = n$ e seja $q$ o número de operações de consultas que devem ser feitas. Cada consulta possui dois inteiros $a$ e $b$, onde $1 \leq a \leq b \leq n$, com o objetivo de calcular:

$$max(V_a, V_{a+1}, ..., V_{b-1}, V_b)$$

onde a operação $max$ retorna o máximo entre os valores passados como parâmetros. Dessa maneira, em cada consulta queremos saber o valor máximo que aparece entre os valores de $V_a$ e $V_b$. Após a introdução da estrutura *Prefix Sum*, uma ideia natural seria construir uma estrutura de prefixo $Pmax$ mantendo o valor máximo do prefixo em vez da soma do mesmo, assim:

$$Pmax_i = max(V_1, V_2, ..., V_i) \quad \forall i \in \{1, ..., n\}$$

Porém, ao analisarmos a operação de $max$, vemos que não existe uma operação reversa $\otimes$ tal que:

$$Pmax_b \otimes Pmax_{a-1} = max(V_b, V_{b-1}, ..., V_{a+1}, V_a)$$

Um contraexemplo para essa ideia seria tal que $V = [5, 1, 2, 3, 4]$ e $a = 2$, $b = 3$. Usando $Pmax$ como um vetor de prefixo da operação $max$, temos a estrutura $Pmax = [5, 5, 5, 5, 5]$ que, como pode ser observado, não calcula a resposta esperada, pois para computar o resultado no intervalo $(a, b)$ não possuímos uma função reversa $\otimes$ capaz de retornar o valor $max(V_3, V_2)$. É necessária uma estrutura capaz de suportar operações $\oplus$ associativas em intervalos tal que não possuam uma operação reversa $\otimes$ correspondente. Assim como o algoritmo inicial proposto na motivação da estrutura *Prefix Sum*, um algoritmo para resolver esse problema seria que para cada consulta percorremos o vetor $V$ das posições $a$ até $b$ acumulando seu máximo. Para cada consulta, no pior caso, temos a complexidade de $\mathcal{O}(n)$, e uma complexidade total do algoritmo de $\mathcal{O}(nq)$. Na Seção 3.2 veremos como podemos reduzir essa complexidade para $O(n \log n + q \log n)$ utilizando o conceito de *Sparse Table* (*ST* ou tabela esparsa, em português).

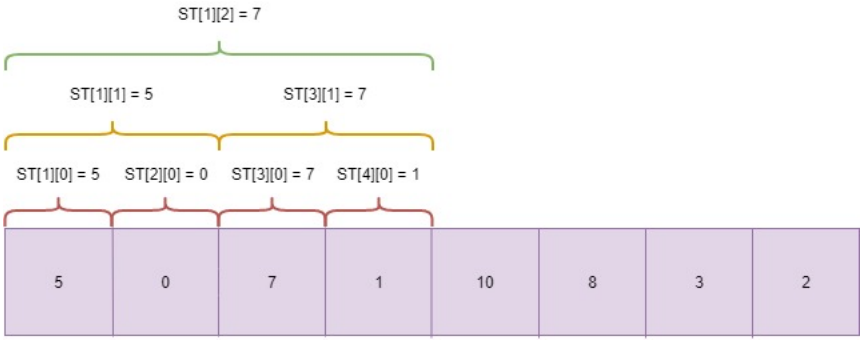
## 3.1 Definição da estrutura

A *Sparse Table* é uma estrutura estática que guarda informações sobre intervalos do vetor que tenham tamanhos de potências de dois. A estrutura é capaz de suportar operações em intervalos de vetor que sejam associativas e não necessariamente reversíveis, em tempo $\mathcal{O}(\log n)$, e sua complexidade de construção em tempo e espaço é de $\mathcal{O}(n \log n)$, onde $n$ é o tamanho do vetor que deve ser avaliado. Além disso, algumas operações que respeitam certas propriedades podem ser calculadas em tempo $\mathcal{O}(1)$. A *Sparse Table* de um vetor $V$ de tamanho $n$ é uma tabela $ST$ que pode ser definida como uma matriz de dimensão $n \times m$, onde $m = \lfloor \log_2 n \rfloor + 1$ e $\lfloor x \rfloor$ é definido como o valor de $x$ arredondado para baixo, e $\lceil y \rceil$ é definido como o valor de $y$ arredondado para cima. Por exemplo: $\lfloor 2.8 \rfloor = 2$ e $\lceil 3.2 \rceil = 4$. Denotaremos por $ST_{i,j}$ o valor da $j$-ésima coluna da $i$-ésima linha da tabela $ST$. A principal ideia da tabela esparsa é pré-calcular para cada posição $i \in \{1, ..., n\}$ a operação $\oplus$ em todos os intervalos que começam em $i$ e tenham tamanho $2^j$ onde $j \in \{0, 1, ..., \lfloor \log_2 n \rfloor\}$ tal que $i + 2^j - 1 \leq n$ (HALIM et al., 2013, p. 388-389), ou seja:

$$ST_{i,j} = V_i \oplus V_{i+1} \oplus ... \oplus ...V_{i+2^j-1}$$

Seja $V = [5, 0, 7, 1, 10, 8, 3, 2]$, onde $\oplus = max$, um exemplo de uma tabela esparsa para o vetor $V$ é a seguinte:

Figura 3.1: Exemplo de algumas posições da tabela esparsa para o vetor $V = [5, 0, 7, 1, 10, 8, 3, 2]$




note que as linhas de $ST$ são indexadas a partir de 1, as colunas são indexadas a partir de 0 e $ST[i][j] = ST_{i,j}$. Como é visto na definição, temos que $ST_{1,0}$ é equivalente ao máximo no intervalo de 1 até $1 + 2^0 - 1 = 1$, logo, $ST_{1,0} = max(V_1...V_1) = 5$. Assim como $ST_{1,1}$ representa o máximo no intervalo de 1 até $1+2^1-1 = 2$, ou seja, $ST_{1,1} = max(V_1...V_2) = 5$, a posição $ST_{1,2}$ representa o máximo no intervalo de 1 até $1 + 2^2 - 1 = 4$, portanto,

$ST_{1,2} = max(V_1...V_4) = 7$.

Sempre que for necessário aplicar a operação $\oplus$ em um intervalo $(a, b)$ de tamanho $2^j$, podemos utilizar o resultado que está pré-calculado na tabela esparsa. Para preencher essa tabela, um algoritmo inicial seria: para cada $j \in \{0, ..., \lfloor \log_2 n \rfloor\}$, calculamos o intervalo de tamanho $2^j$ para todo $i \in \{1, ..., n\}$ tal que $i + 2^j - 1 \leq n$. Porém, como podemos observar, para cada posição teremos que fazer $1 + 2 + 4 + ... + n = \mathcal{O}(n)$ operações. Como temos $\mathcal{O}(n)$ posições, a complexidade final desse algoritmo para construir a estrutura seria $\mathcal{O}(n^2)$.

Podemos tirar proveito do fato que estamos calculando as respostas de tamanho potência de dois e, assim como fizemos com a estrutura *Prefix Sum*, podemos reutilizar os dados que já foram calculados para calcular os próximos. Temos calculado para todo $i \in \{1, ..., n\}$ a potência $j - 1$, assim, podemos calcular a potência $j$ da seguinte maneira:

$$ST_{i,j} = ST_{i,j-1} \oplus ST_{i+2^{j-1},j-1} \quad \forall i \in \{1, ..., n\}$$

como $2^j = 2^{j-1} + 2^{j-1}$, fazemos a operação $\oplus$ em $(i, i + 2^{j-1} - 1)$ e em seguida em $(i + 2^{j-1}, i + 2^{j-1} + 2^{j-1} - 1) = (i + 2^{j-1}, i + 2^j - 1)$, isto é, cobrimos todo o intervalo $(i, i + 2^j - 1)$ referente a $ST_{i,j}$. Por exemplo, na figura 3.1, observe que a posição $ST_{1,2}$ pode ser calculada como $ST_{1,2} = max(ST_{1,1}, ST_{3,1})$, pois $ST_{1,1} = max(V_1, V_2)$ e $ST_{3,1} = max(V_3, V_4)$, assim, $ST_{1,2} = max(V_1, V_2, V_3, V_4)$. Como já temos todos os intervalos de tamanho $2^j$ calculados podemos fazer essa operação em $\mathcal{O}(1)$ para $\oplus = max$, consultando na tabela $ST$ as respectivas posições. Assim, para cada posição $ST_{i_j}$ da tabela, podemos calculá-la em tempo $\mathcal{O}(t)$, onde $t$ é a complexidade da operação $\oplus$. Como temos no máximo $\mathcal{O}(\log n)$ potências $j$ possíveis, e para cada potência preenchemos todos os índices $i \in 1, ..., n$, a complexidade da construção dessa estrutura é $\mathcal{O}(tn \log n)$. Segue a recorrência para a construção de $ST$:

$$ST_{i,j} = \begin{cases} V_i, & \text{se} \quad j = 0 \\ ST_{i,j-1} \oplus ST_{i+2^{j-1},j-1}, & \text{se} \quad i + 2^{j-1} \leq n \\ -\infty & \text{senão} \end{cases} \tag{3.1}$$

Note que em nosso caso base ($j = 0$), a resposta para toda posição $i$ é o próprio valor $V_i$, pois como estamos lidando com um intervalo de tamanho $2^0 = 1$ o intervalo

avaliado é o próprio elemento. Aqui o valor $-\infty$ é o elemento neutro da operação $max$. Para aplicações em problemas reais, o valor de $-\infty$ pode ser substituído pelo valor mínimo que $V_i$ pode assumir.

## 3.2 Operações da estrutura

### 3.2.1 Consulta de máximo em intervalo

A motivação por trás da estratégia é devida à base binária, pois caso o tamanho do intervalo que devemos avaliar não seja uma potência de dois, podemos quebrar um intervalo $(a, b)$ de tamanho $k = b - a + 1$ em suas respectivas potências de dois e em seguida uni-las aplicando a operação $\oplus$. Formalmente, seja a decomposição de $k$ a seguinte:

$$k = 2^{c_1} + 2^{c_2} + ... + 2^{c_m}$$
$$c_1 > c_2 > ... > c_m \geq 0$$

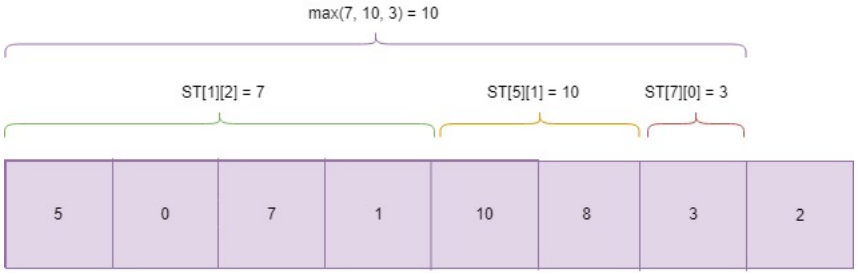
podemos calcular o intervalo $(a, b)$ pela seguinte recorrência, à qual pode ser inicializada com $query\_interval(a, 1)$:

$$query\_interval(i, j) = \begin{cases} ST_{i,j} & \text{se} \quad j = m \\ ST_{i,j} \oplus query\_interval(i + 2^{c_j}, j + 1) & \text{senão} \end{cases} \tag{3.2}$$

como estamos cobrindo todas as potências de dois de $k$, tem-se um intervalo de tamanho $k$, ou seja, o intervalo $(a, b)$ por completo. Por exemplo, seja $V = [5, 0, 7, 1, 10, 8, 3, 2]$, e as consultas:

- $a = 1$ e $b = 4$: $max_{i=1}^{4} V_i = ST_{1,2} = max(V_1, V_2, V_3, V_4) = 7$
- $a = 1$ e $b = 7$: $max_{i=1}^{7} V_i = max(ST_{1,2}, ST_{5,1}, ST_{7,0}) = max\ (max(V_1, V_2, V_3, V_4), max(V_5, V_6), max(V_7)) = 10$
- $a = 3$ e $b = 6$: $max_{i=3}^{6} V_i = max(ST_{3,2}) = max(V_3, V_4, V_5, V_6) = 10$
- $a = 7$ e $b = 7$: $max_{i=7}^{7} V_i = max(ST_{7,0}) = max(V_7) = 3$

Figura 3.2: Exemplo de uma consulta no intervalo $a = 1$ e $b = 7$ em uma tabela esparsa $ST$ de um vetor $V = [5, 0, 7, 1, 10, 8, 3, 2]$




O valor $k$ é quebrado em no máximo $\mathcal{O}(\log n)$ potências de dois, logo, avaliaremos a tabela $ST$ em no máximo $\mathcal{O}(\log n)$ intervalos, e como para cada intervalo que avaliamos sabemos que seu tamanho é uma potência de dois, temos a resposta para esse intervalo salva em $ST$ podendo consultá-la em $\mathcal{O}(1)$. Dessa maneira, a complexidade de uma consulta na tabela esparsa é de $\mathcal{O}(\log n)$, e, assim como na *Prefix Sum*, para toda operação de modificação feita no vetor original $V$ teremos que reconstruir a tabela $ST$ no pior caso. Logo, a complexidade de uma modificação na *Sparse Table* é de $\mathcal{O}(n \log n)$ devido ao tempo de construção da estrutura. A operação $\oplus$ utilizada deve ser associativa, pois a recorrência 3.2 depende da associatividade do operador $\oplus$.

## 3.3 Implementação

Seja a seguinte função que recebe um vetor $V$ de inteiros e retorna uma matriz $ST$ referente à tabela esparsa de $V$, onde $n$ é o tamanho de $V$ e $m$ é o número de potências de dois que são necessárias calcular.

```
1 vector< vector<int> > build_sparse_table(vector<int> v){
2     int n = v.size(), m = log2(n) + 1;
3     vector< vector<int> > ST(n, vector<int>(m, 0));
```

Utilizamos a recorrência abordada na Equação 3.1 para a construção da tabela, onde o caso base é definido como $ST_{i,0} = V_i$, ou seja, os intervalos começando em $i$ de tamanho 1 são definidos como o próprio valor do vetor na posição $i$. A partir dos casos bases, os próximos valores são preenchidos baseados nos valores anteriores, pois

$ST_{i,j} = max(ST_{i,j-1}, ST_{i+2^{j-1},j-1})$.

```
1     for(int i = 1; i < v.size(); i++)
2         ST[i][0] = v[i];
3     for(int j = 1; j < m; j++)
4         for(int i = 1; i+(1 << j) < v.size(); i++)
5             ST[i][j] = max(ST[i][j-1], ST[i+(1 << (j-1))][j-1]);
6     return ST;
7 }
```

Note que na linha 3 o comando $(1 << j)$ corresponde a um inteiro formado por um bit deslocado $j$ vezes à esquerda, ou seja, equivale a $2^j$. A complexidade da construção da estrutura é dada por $\mathcal{O}(n \log n)$ em tempo e espaço.

### 3.3.1 Consulta de máximo em intervalo

Para o cálculo da função de $max$ em um intervalo de $V$, a seguinte função será implementada, onde recebemos a tabela esparsa $ST$ de $V$ como parâmetro, assim como o intervalo $(a, b)$ desejado. Inicializamos $k$ como o tamanho do intervalo desejado, $m$ como a potência máxima que aparece na decomposição de $k$ e $res$ como 0, onde 0 é definido como o elemento neutro da operação de $max$, pois assumimos que $V_i \geq 0 \; \forall i \in \{1, ..., n\}$.

```
1 int query_max(vector< vector<int> >& ST, int a, int b){
2     int k = b-a+1, res = 0;
3     int m = log2(k);
```

Utilizamos a decomposição binária de $k$ para avaliar os sub-intervalos de tamanhos de potências de dois presentes em $[a...b]$, como demonstrado na Equação 3.2. Note que o comando da linha 2 é utilizado para verificar se a $i$-ésima potência de dois está ligada na representação binária de $k$, e $a$ é utilizado como acumulador do deslocamento.

```
1     for(int i = m; i >= 0; i--)
2         if((1 << i)&k){
```

```
3             res = max(res, ST[a][i]);
4             a += (1 << i);
5         }
6     return res;
7 }
```

A complexidade da operação de *max* em intervalo é dada por $\mathcal{O}(\log n)$ em tempo e $\mathcal{O}(1)$ em espaço.

## 3.4 Problema Relacionado

Assim como boa parte das estruturas de dados, a *Sparse Table* normalmente é utilizada em conjunto com outras estratégias de programação, como algoritmos gulosos, teoria dos grafos e programação dinâmica, como é o caso do problema *Imperial Roads* da etapa nacional da maratona de programação de 2017 (NIÑO, 2017). Uma versão simplificada do problema é:

> Seja $G$ uma árvore geradora mínima não direcionada, onde cada aresta $x - y$ em $G$ possui um peso $w$ associado a ela, e seja $q$ o número de consultas que devem ser feitas. Em cada consulta recebemos dois inteiros $u$ e $v$, e devemos calcular qual a maior aresta no caminho de $u$ até $v$, note que a raiz da árvore é definida pelo nodo 1. As restrições de valores são $1 \leq n, q \leq 10^5$ e $1 \leq w \leq 10^4$.

Este problema pode ser quebrado em 2 partes: (1) Achar o menor ancestral em comum de $u$ e $v$, que chamaremos de $l$; e (2) Achar a maior aresta de $u$ até $l$ e de $v$ até $l$. Podemos achar o nodo $l$ em tempo $\mathcal{O}(\log n)$ com uma tabela esparsa que possui tempo de construção de $\mathcal{O}(n \log n)$ (BENDER; FARACH-COLTON, 2000). A estratégia que será utilizada para resolver a primeira parte do problema será diferente da demonstrada inicialmente na concepção da estrutura (BENDER; FARACH-COLTON, 2000), porém, será fundamentalmente equivalente, pois construiremos uma tabela esparsa em tempo $\mathcal{O}(n \log n)$ e manteremos consultas em $\mathcal{O}(\log n)$. Contudo, não manteremos uma lista do *tour euleriano* do grafo $G$. Com a primeira parte construída poderemos construir uma segunda tabela esparsa para fazer operações de máximo em um determinado caminho no grafo $G$.

Seja *Ancestral* a primeira tabela esparsa, mantendo para cada nodo $u$ quem é o seu $2^j$-ésimo ancestral, a seguinte recorrência gera a estrutura desejada:

$$Ancestral_{u,j} = \begin{cases} Pai_u, & \text{se} \quad j = 0 \wedge u \neq 1 \\ Ancestral_{Ancestral_{u,j-1},j-1} & \text{se} \quad Ancestral_{u,j-1} \neq 0 \wedge u \neq 1 \\ 0 & \text{senão} \end{cases}$$

onde $Pai_u$ é o ancestral imediato de $u$. Para o caso base em que $j = 0$ devemos calcular o pai imediato de $u$, indicado por $Pai_u$ por sua própria definição. Caso não exista o $2^{j-1}$-ésimo ancestral de um nodo $u$ qualquer, utilizaremos o valor 0 como o padrão para a sua não existência. Repare que a ocorrência de $u = 1$ está contida nesse caso, já que como o nodo 1 é a raiz da árvore ele não possui qualquer ancestral. Caso exista o $2^{j-1}$-ésimo ancestral de $u$, podemos tirar proveito de que o $2^j$-ésimo ancestral de $u$ é equivalente ao $2^{j-1}$-ésimo ancestral do $2^{j-1}$-ésimo ancestral de u, visto que para chegar no $2^j$-ésimo ancestral precisamos subir $2^j$ arestas, logo, subir $2^j$ arestas é equivalente a subir $2^{j-1}+2^{j-1}$ arestas.

Com a tabela *Ancestral* criada, criamos a segunda tabela $ArestaMaxima$ de maneira análoga, onde manteremos para cada nodo $u$ qual a maior aresta entre seus $2^j$-ésimo ancestrais, construída pela seguinte recorrência:

$$ArestaMaxima_{u,j} = \begin{cases} PesoPai_u, & \text{se} \quad j = 0 \wedge u \neq 1 \\ max(ArestaMaxima_{u,j-1}, & \\ ArestaMaxima_{Ancestral_{u_{j-1},j-1}}) & \text{se} \quad Ancestral_{u,j-1} \neq 0 \wedge u \neq 1 \\ 0 & \text{senão} \end{cases}$$

onde $PesoPai_u$ mantém o peso $w$ da aresta de $u - Pai_u$. Podemos agora calcular o menor ancestral em comum para qualquer par de nodos. Ao calcular $l$, o menor ancestral em comum de $u$ e $v$, teremos um dos dois casos: $dep_u = dep_v$ ou $dep_u \neq dep_v$. Tal que $dep_u$ é a profundidade do nodo $u$, ou seja, o número de arestas que existem entre a raiz e o nodo $u$, que pode ser pré-calculada para todo nodo através de uma busca em profundidade em tempo $\mathcal{O}(n)$. Caso $dep_u = dep_v$, temos que achar o nodo $l$ tal que:

- $l$ é um ancestral de $u$;

- $l$ é um ancestral de $v$;
- $dep_l$ é mínima.

É necessário subir a maior quantidade de arestas possíveis enquanto $u$ e $v$ compartilham um ancestral, e faremos isso em potências de dois, pois, caso $l$ se encontre a $k$ arestas de distância de $u$ e $v$, poderemos subir as potências de dois respectivas à decomposição binária de $k$. Como $k$ é quebrado em no máximo $\mathcal{O}(\log n)$ potências de dois, e como para cada potência podemos acessar o resultado armazenado na tabela *Ancestral* em $\mathcal{O}(1)$, podemos achar o nodo $l$ em tempo $\mathcal{O}(\log n)$. É possível calcular a maior aresta de $u$ até $l$ enquanto subimos as potências de $k$ utilizando a tabela $ArestaMaxima$. Aplicamos a mesma estratégia para achar o máximo de $v$ até $l$.

Caso $dep_u \neq dep_v$, onde $dep_u > dep_v$ e $k = dep_u - dep_v$. Seja $z$ o $k$-ésimo ancestral de $u$ (note que podemos calcular o nodo $z$ através da tabela *Ancestral*). Podemos aplicar a mesma lógica para $z$ e $v$, pois como $z$ é um ancestral de $u$ não menor que o ancestral mínimo de $u$ e $v$, pois $dep_z = dep_v$, logo, o ancestral mínimo de $u$ e $v$ também será ancestral mínimo de $z$ e $v$.

# 4 Fenwick Tree

Seja $V$ um vetor de inteiros com $|V| = n$ e seja $q$ o número de consultas que devem ser feitas. Uma consulta pode ser definida por uma das seguintes operações, as quais podem ser feitas em uma ordem qualquer:

1. Dado um par $(a, b)$, onde $1 \leq a \leq b \leq n$, calcular $\sum_{i=a}^{b} V_i$.
2. Dado um par $(i, x)$, onde $1 \leq i \leq n$, alterar o valor $V_i$ para $x$.

Note que esse problema motivacional é análogo ao problema motivacional da estrutura *Prefix Sum*, porém, com a restrição adicional de possíveis mudanças no conteúdo do vetor original $V$. O seguinte algoritmo pode ser proposto: construímos o vetor de *Prefix Sum* $P$ para $V$. Para cada consulta do tipo 1 sabemos como respondê-la utilizando as propriedades da estrutura *Prefix Sum*, e para cada consulta do tipo 2 podemos fazer a alteração no valor $V_i$ e reconstruir o vetor $P$ baseado no novo vetor $V$ após a alteração. Essa solução possui complexidade de $\mathcal{O}(1)$ para consultas do tipo 1 e $\mathcal{O}(n)$ para consultas do tipo 2, portanto a complexidade de sua construção e execução teria complexidade de tempo $\mathcal{O}(nq + n)$ em um pior caso. Veremos que é possível utilizar outra estrutura com melhor performance, com construção em $\mathcal{O}(n \log n)$ e capaz de suportar tais consultas em tempo $\mathcal{O}(\log n)$. Para isso é empregado o conceito de *Fenwick Tree*.

## 4.1 Definição da estrutura

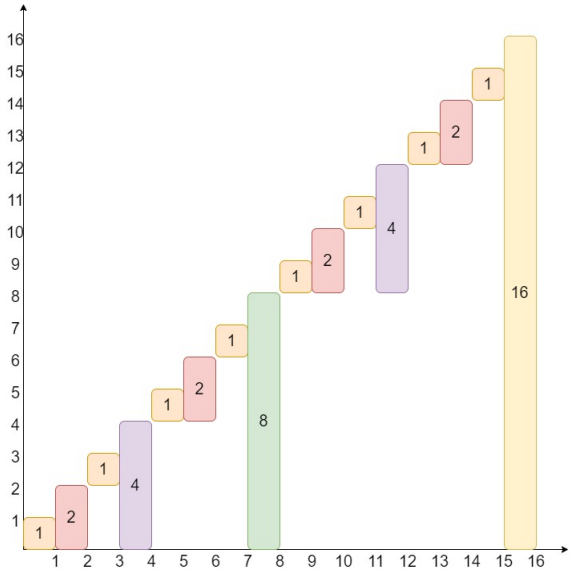
A *Fenwick Tree* é uma estrutura dinâmica que guarda dados sobre os subintervalos de um vetor, capaz de suportar uma função que seja associativa e reversível, e modificações no vetor em tempo $\mathcal{O}(\log n)$. Sua construção é feita em $\mathcal{O}(n \log n)$ e sua complexidade de espaço é de $\mathcal{O}(n)$. Seja $V$ um vetor, onde $|V| = n$. Uma *Fenwick Tree* ($FT$) de $V$, como definida originalmente em seu artigo original (FENWICK, 1994), é um vetor de tamanho $n$, onde cada posição de $FT$ representa um subintervalo de $V$ aplicado à operação $\oplus$. Análoga à *Sparse Table*, a estrutura utiliza-se do fato de que, assim como um inteiro pode ser decomposto em potências de dois, uma operação $\oplus$ aplicada em um intervalo

de $V$ também pode ser decomposta na operação $\oplus$ acumulada em subintervalos de $V$. Formalmente:

$$FT_i = V_{i-LSO(i)+1} \oplus V_{i-LSO(i)+2} \oplus ... \oplus V_i$$

Tem-se que a operação $LSO(x)$ retorna a $j$-ésima potência de dois, onde $j$ indica a posição (indo da direita para a esquerda) do bit menos significativo da representação binária de $x$ que tenha valor 1. Por exemplo, a representação de 52 em 8 bits é 00110100, portanto $LSO(52) = 00000100 = 4$. Para a construção de $FT$, iremos adicionar iterativamente cada elemento $V_i$ tal que $i \in \{1, ..., n\}$.

Figura 4.1: Representação visual dos intervalos de cada posição de $FT$. O número em um intervalo representa seu tamanho. O eixo das abcissas representa os índices de $FT$, enquanto o eixo das ordenadas representa os índices de $V$ cobertos pelo índice de $FT$ correspondente.




Inicialmente a estrutura $FT$ deve ser inicializada com o elemento neutro da operação $\oplus$. Pela definição da estrutura, um elemento $V_i$ estará nas posições $P_1$, $P_2$, ..., $P_k$, onde $P_{k+1} > n$ e $P_k \leq n$. A recorrência de $P$ é dada por:

$$P_k = \begin{cases} i, & \text{se} \quad k = 1 \\ P_{k-1} + LSO(P_{k-1}), & \text{senão} \end{cases} \tag{4.1}$$

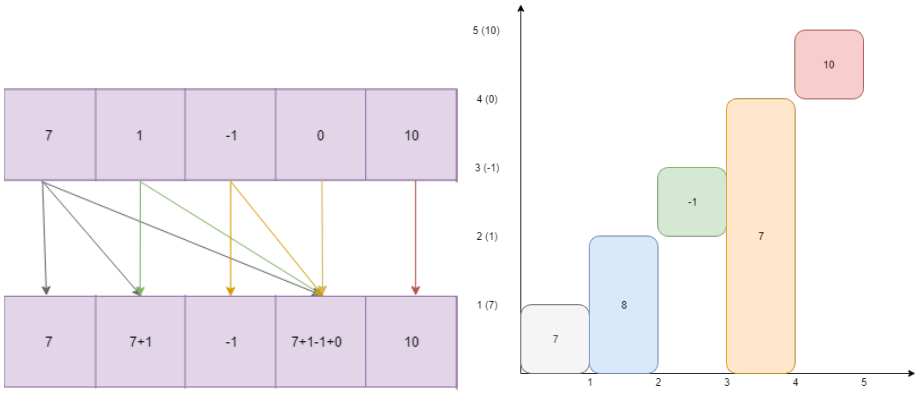
Assim, para cada elemento $V_i$, temos que as posições $FT_{P_1}, FT_{P_2}, ..., FT_{P_k}$ devem ser

modificadas/incrementadas:

$$FT_j \leftarrow FT_j \oplus V_i \quad \forall j \in \{P_1, P_2, ..., P_k\} \tag{4.2}$$

Note que as posições $P_1, P_2, ..., P_k$ são dependentes da posição $i$. Um exemplo de $FT$ a partir de um vetor $V = [7, 1, -1, 0, 10]$, pode ser representado da seguinte maneira:

Figura 4.2: Visualização da construção de uma $FT$ a partir de um vetor $V = [7, 1, -1, 0, 10]$




(a) Visualização em vetor das posições de $FT$

(b) Visualização dos subintervalos das posições de $FT$

na figura 4.2(b), no eixo das ordenadas temos as posições de $FT$, no eixo das abscissas temos as posições de $V$ e cada retângulo indica a área que aquela determinada posição de $FT$ cobre e a sua soma. Por exemplo, a posição 2 de $FT$ cobre os valores de $[1, 2]$ com soma igual a 8 e a posição 5 de $FT$ cobre o valor $[5, 5]$ com soma igual a 10. Já na figura 4.2(a), as arestas indicam as posições que um elemento irá aparece. Por exemplo, o valor de $V_1$ estará nas posições 1, 2 e 4, enquanto que o valor de $V_3$ estará nas posições 3 e 4.

Podemos analisar quatro propriedades referentes à construção dessa estrutura. A primeira propriedade que podemos observar, é que os subintervalos dos nodos de $FT$ possuem tamanhos em potências de dois. Para cada posição $i \in \{1, ..., n\}$ de $FT$, o seu subintervalo é definido por $[i - LSO(i) + 1...i]$, assim, o tamanho $L$ do subintervalo de $i$ é $L = i - (i - LSO(i) + 1) + 1 = LSO(i)$. Como $LSO(i)$ é uma operação que retorna a menor potência de dois ativa dentro da decomposição binária de $i$, temos que para cada posição o tamanho de seu subintervalo é $2^k$, para $k \geq 0$.

A segunda propriedade é que a operação de modificação/construção é feita em tempo $\mathcal{O}(\log n)$, já que temos que a operação de modificação aplica a operação $LSO(i)$

consecutivamente nas posições de $FT$ até chegar em um nodo que seja maior que $n$. Assim, seja $i_1$ a posição que receberá as modificações inicialmente, seja $i_1 = 2^{a_1} + 2^{a_2} + ... + 2^{a_p}$ sua decomposição binária, onde $a_1 < a_2 < ... < a_n$, e $LSO(i_1) = 2^a$. Com isso, temos que a próxima posição que será visitada será $i_2$, onde $i_2 = 2^{a_1+1} + 2^{a_2} + ... + 2^{a_p}$, e temos duas possibilidades:

1. Se $2^{a_1+1} < 2^{a_2}$: então o $LSO(i_2) = 2^{a_1+1} = 2 * 2^{a_1}$, logo, o tamanho do subintervalo de $i_2$ é duas vezes maior que o subintervalo de $i_1$.

2. Caso contrário, se $2^{a_1+1} \geq 2^{a_2}$: então o $LSO(i_2) = 2^{a_2}$, como $a_2 > a_1$ por definição, temos que $2^{a_2}$ é pelo menos duas vezes maior que $2^{a_1}$.

Assim, temos que com cada iteração da operação de modificação o subintervalo da posição avaliada é pelo menos duas vezes maior que o subintervalo anterior, logo, temos no máximo $\mathcal{O}(\log n)$ operações de modificação.

A terceira propriedade é que a operação de modificação/construção deve ser feita em elementos com subintervalos crescentes: seja $i = 2^{a_1} + 2^{a_2} + ... + 2^{a_n}$ a posição que deve ser modificada, e $j = 2^{b_1} + 2^{b_2} + ... + 2^{b_m}$ uma posição que possui $i$ em seu subintervalo e que possui um subintervalo de tamanho menor ou igual a $i$, logo $i < j$ e $LSO(i) \geq LSO(j)$. Como $i < j$, temos que $i$ e $j$ possuem um sufixo de sua decomposição binária iguais até uma potência $k$ onde $2^k$ pertence à decomposição de $j$ e não pertence à decomposição de $i$ e $2^k \neq LSO(j)$, pois caso contrário $i$ seria maior que $j$, o que contraria nossa suposição de $i < j$. Como $2^k$ está ativo em $j$ e não em $i$, mesmo se $2^0 + 2^1 + ... + 2^{k-1}$ pertencer ao prefixo da decomposição binária de $i$, ainda teremos que $2^0 + 2^1 + ... + 2^{k-1} < 2^k$. Logo, mesmo se retirarmos a menor potência de $j$, ainda teremos que $j - LSO(j) > i$. Dessa maneira, temos que um subintervalo que seja menor ou igual ao subintervalo de $i$ nunca passará por $i$.

A quarta e última propriedade é que a operação de modificação/construção passa por todas as posições que são válidas. Seja $i = 2^{a_1} + 2^{a_2} + ... + 2^{a_k}$ a posição inicial de modificação. Como foi demonstrado, temos que o próximo valor a ser modificado deve ser o menor valor $j > i$ tal que $LSO(j) > LSO(i)$ e $j - LSO(j) < i$. Naturalmente, o próximo tamanho de subintervalo a ser avaliado deve ser o imediatamente maior a $LSO(i)$, dessa maneira, temos que $LSO(j) = 2 * LSO(i)$. Temos de tratar dois casos:

1. Se a potência $LSO(j)$ já está presente em $i$: pelo nosso método de construção,

teremos que $j < i$, pois $j$ é formado de todas as potências de $i$ exceto a $LSO(i)$. Assim, para fazer com que $j$ seja o menor valor possível maior que $i$ e que mantenha $LSO(j) = 2 * LSO(i)$ precisamos achar a primeira potência de dois $k$, tal que $2^k$ seja maior que $LSO(j)$ e não esteja presente em $i$ e adicionamos essa potência em $j$. Como $j$ é formado de todas as potências de $i$ exceto a $LSO(i)$, e como adicionamos a menor potência possível em $j$ para fazer com que $j > i$ sem quebrar a restrição de $LSO(j) = 2 * LSO(i)$, construímos o menor valor $j > i$ tal que $LSO(j) = 2 * LSO(i)$. Porém, perceba que a restrição $j - i < LSO(j)$ não é respeitada, pois $2^k - LSO(i) < LSO(j)$ e $LSO(j) = 2 * LSO(i)$. Temos que $2^k < 3 * 2^{a_1}$, que deve ser falso, pois, por definição temos $k > a_2$. Com isso, o próximo nodo a conter $i$ em seu subintervalo deve ter um subintervalo de tamanho igual à primeira potência maior que LSO(i) que não aparece em $i$, ou seja, um subintervalo de tamanho igual a $2^k$.

2. Caso contrário, o valor $j$ não precisa de nenhuma modificação.

Para cada posição de $FT$ iremos fazer no máximo $\mathcal{O}(\log n)$ modificações. Logo, a complexidade de construção de $FT$ é dada por $\mathcal{O}(nt \log n)$, onde $t$ é a complexidade de tempo da operação $\oplus$.

## 4.2 Operações da estrutura

### 4.2.1 Operação em intervalo

Com a estrutura $FT$ criada, podemos definir a seguinte recorrência para calcular a operação $\oplus$ aplicada no intervalo $(1, n)$:

$$query(n) = \begin{cases} 0, & \text{se} \quad n = 0 \\ FT_n \oplus query(n - LSO(n)), & \text{senão} \end{cases} \tag{4.3}$$

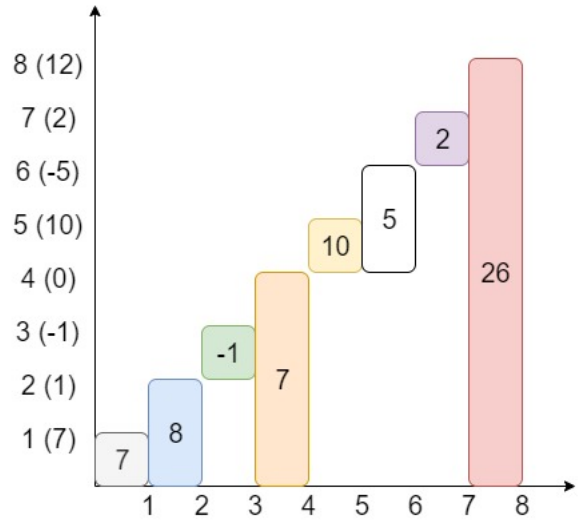
onde 0 representa o elemento neutro da operação $\oplus$. A cada iteração retiramos uma potência de dois crescente, logo, faremos no máximo $\mathcal{O}(\log n)$ operações para calcular $\oplus(1, n)$. Dessa maneira:

$$query(n) = FT_n \oplus FT_{n-LSO(n)} \oplus ... \oplus 0, \quad \text{logo}$$
$$query(n) = (V_n \oplus V_{n-1} \oplus ... \oplus V_{n-LSO(n)+1}) \oplus$$
$$(V_{n-LSO(n)} \oplus V_{n-LSO(n)-1} \oplus ... \oplus V_{n-LSO(n)-LSO(n-LSO(n))+1}) \oplus ... \oplus 0$$

observe que todos os elementos de $V_1 ... V_n$ estão incluídos unicamente em $query(n)$. Seja $V = [7, 1, -1, 0, 10, -5, 2, 12]$, temos que o intervalo:

- $[1, 7]$ pode ser composto pelos nodos $7, 6$ e $4$;
- $[1, 4]$ pode ser composto pelo nodo $4$;
- $[1, 5]$ pode ser composto pelos nodos $5$ e $4$.

Figura 4.3: Visualização da $FT$ de $V = [7, 1, -1, 0, 10, -5, 2, 12]$




Para calcular a operação $\oplus$ em um intervalo $(a, b)$, temos a seguinte definição:

$$queryinterval(a, b) = \begin{cases} query(b), & \text{se} \quad a = 1 \\ query(b) \otimes query(a-1), & \text{senão} \end{cases} \tag{4.4}$$

onde $\otimes$ é a função reversa de $\oplus$. De maneira análoga ao *Prefix Sum*, caso $a = 1$:

$$queryinterval(1, b) = V_b \oplus V_{b-1} \oplus ... \oplus V_1$$

E caso $a \neq 1$:

$$queryinterval(a,b) = (V_b \oplus V_{b-1} \oplus ... \oplus V_1) \otimes (V_{a-1} \oplus V_{a-2} \oplus ., . \oplus V_1), \quad \text{logo}$$
$$queryinterval(a,b) = V_b \oplus V_{b-1} \oplus ... \oplus V_a$$

Como chamamos a função $query(n)$ duas vezes no pior caso, tem-se portanto que uma operação do tipo 1 pode ser calculada em tempo $\mathcal{O}(log\ n)$. Note que a operação $\oplus$ deve ser:

- associativa: $FT_n \oplus FT_{n-LSO(n)} \oplus ... \oplus 0 = ((V_n \oplus V_{n-1} \oplus ... \oplus V_{n-LSO(n)+1}) \oplus (V_{n-LSO(n)} \oplus V_{n-LSO(n)-1} \oplus ... \oplus V_{n-LSO(n)-LSO(n-LSO(n))+1})) \oplus ... \oplus 0$;
- reversível: $queryinterval(a,b) = query(b) \otimes query(a-1)$;
- possuir elemento neutro: a estrutura precisa ser inicializada pelo elemento neutro.

### 4.2.2 Modificação de um elemento

Como um elemento $V_i$ aparece somente $\mathcal{O}(\log n)$ vezes em $FT$, caso uma operação de alteração seja feita em $V_i$, somente as posições onde $V_i$ aparece em $FT$ deverão ser modificadas. Assim, uma operação do tipo 2 pode ser computada em tempo $\mathcal{O}(r \log n)$ mantendo as propriedades da estrutura, onde $r$ é a complexidade de tempo para aplicar a operação inversa $\otimes$. A recorrência para a modificação do elemento $V_i$ para $x$ é dada por:

$$FT_j \leftarrow (FT_j \otimes V_i) \oplus x \quad \forall j \in \{P_1, P_2, ..., P_k\} \tag{4.5}$$

Inicialmente fazemos a operação $FT_j \otimes V_i$, onde $\otimes$ é a operação reversa de $\oplus$, para reverter o valor acumulado de $V_i$ em $FT_j$, e em seguida é feita a modificação de $x$ em $FT_k$ pelo operador $\oplus$. Aqui, $P_1, P_2, \ldots P_k$ são as posições em $FT$ que o elemento $V_i$ está presente e podem ser obtidas pela Equação 4.1.

## 4.3 Implementação

A seguinte função recebe um vetor de inteiro $V$ e retorna $FT$, a *Fenwick Tree* correspondente a $V$. Para cada elemento $V_i$ de $V$ iremos passar por todas as posições de $FT$

que possuem $i$ como um elemento de seu subintervalo e atualizamos aquela posição com a recorrência 4.5. Note que foi utilizado o valor 0 como elemento neutro.

```
1 vector<int> build_fenwick_tree(vector<int> v){
2     int n = v.size();
3     vector<int> ft(n, 0);
4     for(int i = 0; i < n; i++)
5         for(int j = i; j < n; j+=j&(-j))
6             ft[j] += v[i];
7     return ft;
8 }
```

Para cada elemento passamos por no máximo $\mathcal{O}(\log n)$ posições, logo, a construção é feita em tempo $\mathcal{O}(n \log n)$. O comando feito para o incremento de $j$ na linha 5 é equivalente a $j+ = LSO(j)$.

### 4.3.1 Soma em intervalo

Assim como na Equação 4.4, recebemos como parâmetro o intervalo $(a, b)$ e adicionalmente o vetor $FT$, e retornamos a soma do intervalo $[V_a...V_b]$.

```
1 int queryinterval(int a, int b, vector<int> &ft){
2     if(a == 1) return query(b, ft);
3     else return query(b, ft) - query(a-1, ft);
4 }
```

Já a função *query*, recebe como parâmetro a posição $i$ e o vetor $FT$ e retorna o somatório do intervalo $[V_1...V_i]$, como demonstrado na recorrência 4.3. Note que, assim como na Seção 4.3, o valor 0 é usado como valor nulo da operação.

```
1 int query(int i, vector<int> &ft){
2     if(i == 0) return 0;
3     else return ft[i] + query(i - i&(-i));
4 }
```

No pior caso, a função *query* é chamada 2 vezes pela função *queryinterval*. Como a função *query* retira recursivamente potências de dois crescentes, temos que a complexidade da soma em intervalo é feita em tempo $\mathcal{O}(\log n)$.

### 4.3.2 Modificação de um elemento

Recebemos como parâmetros a posição que deve ser modificada, o vetor $V$ original, o vetor $FT$ da *Fenwick Tree*, a posição $i$ de modificação e o valor x que irá substituir o valor de $V_i$. Primeiramente, passamos por todas as posições de $FT$ que possuem a posição $i$ de $V$ como subintervalo, revertemos o antigo valor de $V_i$ e adicionamos o valor de $x$, como é definido na Equação 4.5. Em seguida, atualizamos o valor $x$ como o valor mais recente no vetor $V$.

```
1 void update(int i, vector<int> &ft, vector<int>& v, int x){
2     for(int pos = i; pos < ft.size(); pos += pos&(-pos))
3         ft[pos] = ft[pos]-v[i] + x;
4     v[i] = x;
5 }
```

A função *update* irá passar por todas as posições que possuem o índice $i$ em seu subintervalo. Como só existem $\mathcal{O}(\log n)$ posições válidas, a complexidade de tempo de *update* é de $\mathcal{O}(\log n)$.

## 4.4 Problema Relacionado

No problema *Cortador de Pizza* da etapa regional da maratona de programação 2018 (ANIDO, 2018), temos um exemplo de uma aplicação padrão da estrutura *Fenwick Tree*. Uma versão simplificada do problema é:

> Seja $P$ uma pizza retangular de tamanho $X \times Y$, onde seu ponto esquerdo in-

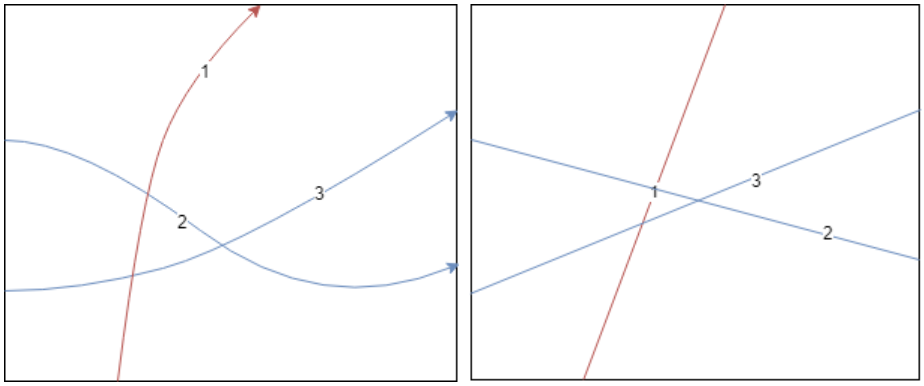
ferior é a coordenada $(0, 0)$ e seu ponto direito superior é a coordenada $(X, Y)$. Serão feitos $n$ cortes horizontais e $m$ cortes verticais na pizza, onde cortes horizontais começam na borda esquerda da pizza e continua monotonicamente até a borda direita e cortes verticais começam na borda inferior da pizza e continuam monotonicamente até a borda superior. Para cada corte horizontal são dados dois inteiros $Y_1$ e $Y_2$ indicando que o corte começa na altura $Y_1$ da borda esquerda da pizza e termina na altura $Y_2$ na borda direita da pizza. Para cada corte na vertical são dados dois inteiros $X_1$ e $X_2$ indicando que o corte começa no ponto $X_1$ na borda inferior e termina no ponto $X_2$ na borda superior. Os cortes respeitam as seguintes restrições:

1. Dois cortes só possuem um ponto de intersecção;
2. Três cortes não se interceptam em um mesmo ponto;
3. Dois cortes não se interceptam na borda da pizza;
4. Um corte não intercepta um canto da pizza.

Encontre o número de pedaços resultantes dos cortes feitos na pizza. As restrições de valores: $1 \leq X, Y, X_i, Y_i \leq 10^9$ e $1 \leq n, m \leq 10^5$.

Nota-se que, devido às restrições, podemos tratar todos os cortes como sendo lineares. Uma pizza com cortes não lineares pode ser convertida em uma pizza com cortes lineares mantendo o número de pedaços da pizza original:

Figura 4.4: Exemplo de linearização dos cortes de uma pizza




(a) Uma pizza de 7 pedaços

(b) Uma pizza de 7 pedaços com cortes lineares

onde o corte 1 representa um corte vertical, e os cortes 2 e 3 representam cortes na

horizontal. Observe que o número de cortes horizontais e verticais acumulados com as intersecções entre as retas é equivalente ao número de pedaços da pizza $-1$, pois inicialmente temos 1 pedaço de pizza (a pizza inteira) e a cada intersecção ou corte um novo pedaço é adicionado. Assim, o problema pode ser resumido a encontrar o número de intersecções de retas que existem na pizza. Cada reta na horizontal possui uma intersecção com cada reta na vertical. Outras possíveis intersecções são as que ocorrem entre as próprias retas na horizontal e as intersecções que ocorrem entre as próprias retas na vertical.

Inicialmente, iremos calcular as intersecções entre as retas horizontais, tal que os cortes serão calculados de maneira iterativa. Seja $V_{hor}$ o vetor de cortes horizontais:

$$V_{hor} = [(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)]$$

onde $a_i$ é a altura do $i$-ésimo corte na borda esquerda, e $b_i$ é a altura do $i$-ésimo corte na borda direita. Iremos ordenar o vetor $V_{hor}$ de maneira crescente pelo valor de $a_i$ (não há casos de empate devido à restrição 3). Assim, durante o processamento do corte $i$ temos que todos os cortes que têm inicio antes de $a_i$ já foram processados. Um corte $j$ possui intersecção com um corte $i$ se (1) $a_j < a_i$ e (2) $b_j > b_i$. A condição (1) é inicialmente satisfeita devido a ordem dos elementos em $V_{hor}$. O número de intersecções do corte $i$ com algum corte $j$, onde $j < i$, será equivalente ao número de elementos que satisfazem (2). Para encontrar este valor utilizaremos um vetor $FT$ baseado na estrutura *Fenwick Tree* apresentada.

Ao processar um corte $i$ iremos adicionar à posição $b_i$ de $FT$ que um corte é finalizado nesta posição, onde $FT$ irá acumular o final de todos os cortes $j$, onde $j < i$. Para encontrar a quantidade de elementos maiores que $b_j$, podemos utilizar $queryinterval(b_j + 1, 10^9)$, pois são contados todos os elementos maiores do que $b_j$ que são menores ou iguais ao intervalo máximo das restrições. Com isso é possível contar todos os cortes horizontais que possuem uma intersecção entre si.

Contudo, a recorrência $queryinterval(b_j + 1, 10^9)$ necessita de um vetor $FT$ de tamanho $10^9$, o que é inviável para o contexto deste problema. Para diminuir a dimensão das coordenadas, o conceito de compressão de coordenadas é utilizado. Utilizaremos uma função para mapear os valores atuais para outros de menor dimensão. Seja $b = [b_1, b_2, ..., b_n]$. O menor valor de $b$ é mapeado para o valor 1, o segundo menor valor de $b$ é mapeado para o valor 2, assim, o $n$-ésimo menor valor de $b$ é mapeado para $n$. Ao

aplicar essa função para todos os valores de $b$, temos que o maior valor de $b$ será no máximo $n$. Note que as restrições ainda se aplicam: um valor $b_i < b_j$ antes da compressão de coordenadas, continuará sendo válido após a aplicação da compressão. De maneira análoga, repetiremos a estratégia utilizada para o cálculo das intersecções entre as retas verticais.

# 5 Segment Tree

Seja $V$ um vetor de inteiros com $|V| = n$ e seja $q$ o número de consultas que devem ser feitas. Uma consulta pode ser definida por uma das seguintes operações, as quais podem ser feitas em uma ordem qualquer:

1. Dado um par $(a, b)$, onde $1 \leq a \leq b \leq n$, calcular $max(V_a, V_{a+1}, ..., V_b)$.
2. Dado um par $(i, x)$, onde $1 \leq i \leq n$, alterar o valor de $V_i$ para $x$.

Repare que este problema motivacional é similar ao problema do Capítulo 3, com a adição de consultas que possibilitam alterações nos valores do vetor original $V$. Veja que:

- *Prefix sum*: não pode ser usada pois há operações de modificação a serem realizadas e a operação desejada não é reversível;
- *Sparse Table*: suporta operações não reversíveis, porém não pode ser usada, pois há operações de modificação a serem realizadas;
- *Fenwick Tree*: suporta operações de modificação, mas não é capaz de suportar operações não reversíveis.

Dessa maneira, é necessária uma estrutura robusta o suficiente para lidar com operações associativas e não reversíveis em intervalo, e capaz de suportar operações de modificação. Veremos como podemos construir essa estrutura em $\mathcal{O}(n \log n)$ e responder consultas do tipo 1 e 2 em tempo $\mathcal{O}(\log n)$. Veremos também como podemos fazer operações de modificação em subintervalos do vetor e não somente em posições, com o conceito de *Segment Tree* (*SegTree* ou Árvore Segmentada, em português).

## 5.1 Definição da estrutura

A *Segment Tree* é uma estrutura dinâmica robusta, capaz de suportar operações em intervalo que sejam associativas e não necessariamente reversíveis, assim como operações de modificação em tempo $\mathcal{O}(\log n)$ (BENTLEY, 1977). Não limitada somente a isso, a

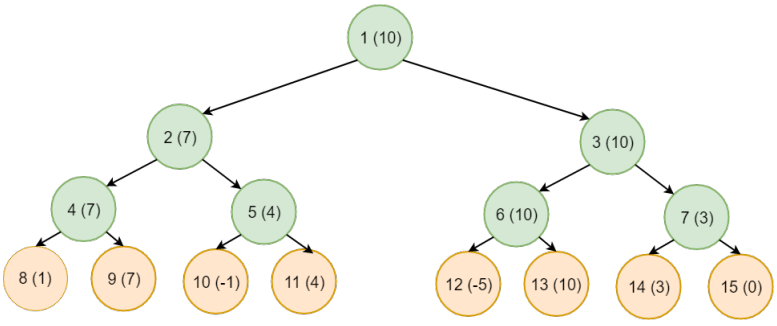
*Segment Tree* suporta operações de modificação em intervalo com tempo $\mathcal{O}(\log n)$ com o conceito de *Lazy Propagation*, sua extensão natural. É uma estrutura flexível e prática para vetores, mas também pode ser escalável para matrizes com a *Segment Tree 2D*, onde cada nodo de uma *Segment Tree* é outra *Segment Tree*, sendo capaz de fazer operações como a soma de um retângulo de uma matriz em tempo $\mathcal{O}(\log^2 n)$. Porém, essa extensão foge do escopo deste trabalho.

Uma *Segment Tree* $T$ de um vetor $V$ de tamanho $n$ é um vetor de tamanho no máximo $4n$, onde cada posição representa um subintervalo do vetor $V$ aplicado a uma operação $\oplus$. Possui uma estrutura recursiva, onde:

1. A posição $T_1$ (raiz) representa o intervalo $[1...n]$ aplicado à operação $\oplus$;
2. As posições que representam um intervalo de tamanho 1 aplicado à $\oplus$, como $[k...k]$, são consideradas as folhas de $T$;
3. Toda posição $T_i$ que não é uma folha representa o intervalo $[L_i...R_i]$ de $V$ aplicado à operação $\oplus$, e terá dois filhos: filho esquerdo, que será a posição $T_{2i}$ e lida com o intervalo $[L_i...mid_i]$, e filho direito, que será a posição $T_{2i+1}$ e lida com o intervalo $[mid_i + 1...R_i]$, onde $mid_i = \lfloor (L_i + R_i)/2 \rfloor$.

Seja $V = [1, 7, -1, 4, -5, 10, 3, 0]$ e $\oplus = max$, a *Segment Tree* correspondente à $V$ seria o vetor $T = [10, 7, 10, 7, 4, 10, 3, 1, 7, -1, 4, -5, 10, 3, 0]$. A Figura 5.1 representa $T$ visualizado como uma árvore.

Figura 5.1: Visualização de $T$ para $V = [1, 7, -1, 4, -5, 10, 3, 0]$




Nessa representação temos que os nodos laranjas são folhas, nodos verdes são não-folhas e o valor $i(x)$ em cada nodo representa a sua posição $i$ e o valor $x$ de $max$ aplicado em seu intervalo $[L_i...R_i]$.

Para a construção de $T$, podemos aproveitar algumas propriedades dessa representação para calcular a operação $max$ em um intervalo de maneira eficiente. Note que o cálculo de $\oplus$ na posição $T_i$, pode ser dado segundo a Equação 5.1, uma vez que $\oplus$ é uma operação associativa.

$$T_i = T_{2i} \oplus T_{2i+1} \tag{5.1}$$

Como $T_{2i}$ cobre o intervalo $[L_i...mid]$ e a posição $T_{2i+1}$ cobre o intervalo $[mid+1...R_i]$, teremos que a operação $T_{2i} \oplus T_{2i+1}$ cobrirá o intervalo $[L_i...R_i]$. Dessa maneira, para calcular o valor de uma posição $T_i$ qualquer, basta calcular os valores de seu filho esquerdo e filho direito. Contudo, quando $T_i$ é uma folha e não possui filhos, temos que $T_i = V_{L_i}$, pois $T_i$ representa o intervalo $[L_i...R_i]$, tal que $L_i = R_i$. Seja $T$ a árvore segmentada de $V$ construída a partir de $build(1, 1, n)$ segundo a recorrência a seguir.

$$build(u, L, R) = \begin{cases} \oplus(V_L) & \text{se} \quad L = R \\ esq \oplus dir & \text{senão} \\ & \text{onde } esq = build(2u, L, (L+R)/2) \\ & \qquad dir = build(2u, (L + R/2) + 1, R) \end{cases} \tag{5.2}$$

No texto que segue, denotaremos $T_u$ o valor de $build(u, L_u, R_u)$. Referente à complexidade de construção dessa estrutura, duas propriedades são observáveis:

1. A altura da árvore é $\mathcal{O}(\log n)$: a cada iteração o intervalo de um nodo é dividido pela metade até chegar em uma folha. Teremos no máximo $\log_2 n$ divisões possíveis para um vetor de tamanho $n$, logo a altura deve ser no máximo $\mathcal{O}(\log n)$.

2. Existem no máximo $4n$ nodos em $T$: como a altura da árvore é $\mathcal{O}(\log n)$ e cada nível da árvore tem no máximo 2 vezes mais nodos que a anterior (cada nodo do nível anterior tem dois filhos), teremos $2^0 + 2^1 + 2^2 + ... + 2^{\log n}$ nodos. Temos também que $2^k - 1 = 2^0 + 2^1 + ... + 2^{k-1}$ e $2^0 + 2^1 + 2^2 + ... + 2^{\lceil \log n \rceil} = 2^{\lceil \log n \rceil} - 1 + 2^{\lceil \log n \rceil} = 4n - 1$.

Portanto, a complexidade da construção da árvore é feita em tempo $\mathcal{O}(nt)$, pois cada nodo é avaliado somente uma vez, onde $t$ é a complexidade de tempo para aplicar a operação $\oplus$. Como demonstrado anteriormente, a complexidade em espaço da árvore é de $\mathcal{O}(n)$.

## 5.2 Operações da estrutura

### 5.2.1 Consulta de máximo em intervalo

Seja $(a, b)$ um intervalo da *Segment Tree* $T$. O cálculo de uma operação de consulta em $T$ é feito por uma travessia por todos os ramos necessários para calcular o intervalo $(a, b)$ e utiliza os valores computados de $T$ para responder em tempo constante a operação de $max$ em um subintervalo coberto pelos nodos. A recorrência para calcular o máximo de $(a, b)$ é dada por $query(1, 1, n, a, b)$ onde:

$$query(u, L, R, a, b) = \begin{cases} 0 & \text{se} \quad R < a \vee L > b \\ T_u & \text{se} \quad L \geq a \wedge R \leq b \\ max(esq, dir) & \text{senão} \\ & \text{onde } esq = query(2u, L, (L + R)/2, a, b) \\ & \qquad dir = query(2u + 1, (L + R)/2 + 1, R, a, b) \end{cases} \tag{5.3}$$

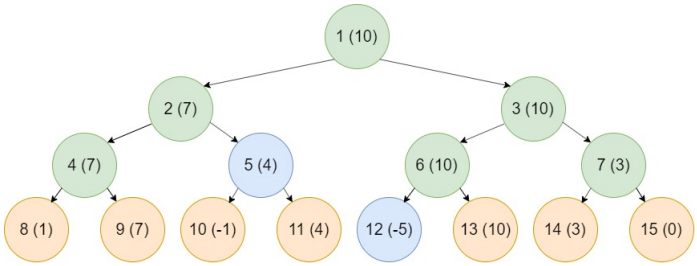
Repare que para o nodo $u$ avaliado em $query(u, L, R, a, b)$ teremos as seguintes opções:

1. Nenhum dos filhos de $T_u$ compreende algum valor do intervalo $(a, b)$. Este caso é coberto pela primeira condição e retornamos o valor neutro da operação de $max$ (esse valor deverá ser o menor valor de acordo com o contexto do problema; aqui o valor 0 foi utilizado);

2. Ambos os filhos de $T_u$ estão totalmente cobertos pelo intervalo $(a, b)$. Portanto, retornamos o valor salvo na posição $T_u$ que corresponde ao intervalo $[L_{T_u}...R_{T_u}]$. Isso é garantido pela segunda condição;

3. Algum filho de $T_u$ (possivelmente ambos) está parcialmente coberto pelo intervalo $(a, b)$. Então expandimos ambos os filhos para encontrar os nodos que estão totalmente cobertos, que acontece na terceira condição. Veja que sempre haverá um nodo que está totalmente coberto, pois no pior dos casos este nodo será uma folha.

Figura 5.2: Exemplo de uma consulta de máximo no intervalo $(3, 5)$. Os nodos em azul são aqueles que serão contabilizados na resposta da consulta. O resultado para essa consulta é $max(FT_5, FT_{12}) = max(4, -5) = 4$.




Veja que sempre teremos no máximo 2 nodos com sobreposição ao intervalo $(a, b)$ descendo pela árvore durante cada nível da consulta, e como cada nodo irá passar por no máximo $\mathcal{O}(\log n)$ filhos (devido à altura da árvore), teremos uma complexidade de tempo $\mathcal{O}(t \log n)$, tal que $t$ é a complexidade de tempo de $\oplus$. A prova pode ser feita por contradição: seja $(a, b)$ o intervalo de consulta e $h$ um nível qualquer da árvore onde temos três nodos com sobreposição ao intervalo $(a, b)$, $T_i$, $T_j$ e $T_k$ com $i \leq j \leq k$, descendo pela árvore. O intervalo atingido pelos três nodos será $[T_{esq_i}...T_{dir_k}]$. Como $T_i$ e $T_k$ são nodos que estão descendo pela árvore, alguma parte de seus subintervalos se encontram em $(a, b)$. Logo, pelo fato de os intervalos de consulta serem contínuos, temos que pelo menos as posições $[T_{dir_i}...T_{esq_k}]$ fazem parte de $(a, b)$ e as posições $[T_{esq_j}...T_{dir_j}]$ estão contidas nesse intervalo. Assim, como o subintervalo do nodo $T_j$ está contido pelos nodos $T_i$ e $T_k$ e devido à nossa segunda condição da recorrência, temos que o nodo $T_j$ não poderá continuar descendo pela árvore, mantendo assim somente dois nodos por nível. Repare que, devido à segunda condição, somente nodos válidos que pertencem ao intervalo de consulta serão avaliados na resposta e, devido à terceira condição, iremos passar por todos os ramos necessários para fazer o cálculo de $(a, b)$.

### 5.2.2 Modificação de um elemento

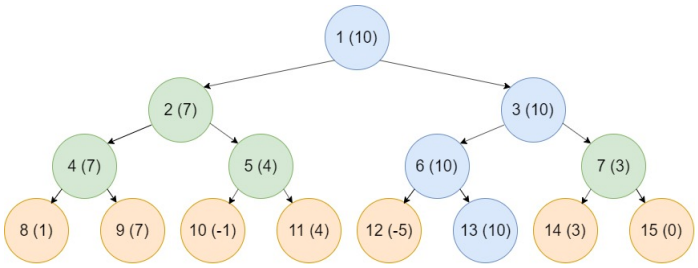
Seja $(i, x)$ uma consulta de modificação da *Segment Tree* $T$ criada a partir de $V$. Uma operação de modificação é um caminho em $T$, partindo da raiz e chegando na folha destino. Ao chegar na folha de modificação, todos os nodos que possuem $i$ em seu subintervalo devem ser recalculados devido à mudança em um de seus descendentes. A recorrência para fazer uma modificação é similar à recorrência 5.2 de construção, e para uma modificação

$(i, x)$ é dada por $update(1, 1, n, i, x)$:

$$update(u, L, R, i, x) = \begin{cases} T_u & \text{se} \quad i > R \vee i < L \\ \oplus(x) & \text{se} \quad L = R \wedge L = i \\ esq \oplus dir & \text{senão} \\ & \text{onde } esq = update(2i, L, (L+R)/2, i, x) \\ & \quad dir = update(2i+1, (L+R)/2+1, R, i, x) \end{cases} \tag{5.4}$$

Um nodo $u$ de $T$ visitado por *update* chamará no máximo uma recursão para um de seus filhos. O filho chamado irá recursivamente chamar o seu filho, atualizar seu valor na árvore e retorná-lo. Como a altura da árvore é no máximo $\mathcal{O}(\log n)$, temos que no máximo $\mathcal{O}(\log n)$ nodos serão visitados e modificados, totalizando uma complexidade de tempo de $\mathcal{O}(t \log n)$, onde $t$ é a complexidade de tempo para aplicar a operação $\oplus$. Após fazer o cálculo de $update(u, L, R, i, x)$, o valor resultante dessa chamada deve ser atualizado no nodo $T_u$, mantendo assim a *Segment Tree* $T$ atualizada.

Figura 5.3: Exemplo de uma consulta de modificação na posição 6. Os nodos em azul são aqueles que serão visitados durante a consulta.




### 5.2.3 Modificação de um intervalo

Até o momento foram abordadas operações de modificação em posições singulares, mas digamos que a operação de modificação do tipo 2 do problema motivador seja alterada para a seguinte:

- Dado uma tripa $(i, j, x)$, onde $1 \leq i \leq j \leq n$, alterar o valor das posições de $i$ a $j$ de $V$ para $x$.

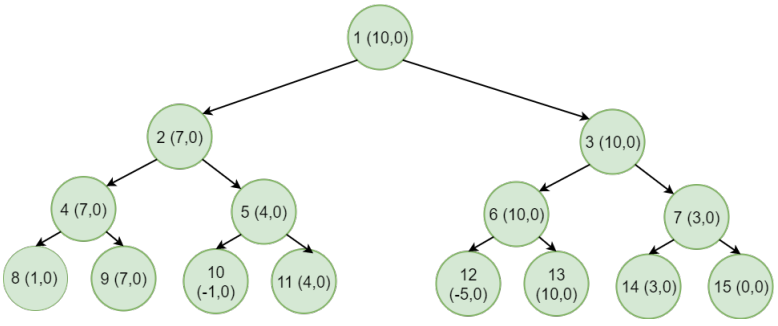
Utilizando a operação de modificação de um elemento, cada consulta teria complexidade de tempo de $\mathcal{O}(n \log n)$ no pior caso, o que é inviável para um contexto de competição

onde $n$ e $q$ podem ser tão grandes quanto $10^5$. Para lidar com esse tipo de consulta em tempo viável de competição, temos que utilizar o conceito de *Lazy Propagation*.

Manipularemos esse tipo de consulta de maneira similar a como manipulamos as consultas de máximo em intervalo. Se um intervalo $(i, j)$ deve ser modificado, podemos quebrá-lo em $\mathcal{O}(\log n)$ nodos que representam esse intervalo e dessa forma podemos fazer as modificações somente nesses intervalos, armazenando uma informação nesses nodos que deve ser propagada para todos os seus filhos. Com essa modificação, é necessária a atualização dos valores da *Segment Tree*. Em um pior caso uma atualização poderia ser feita sobre toda a árvore $(1, n)$ e atualizar todos os nodos da árvore seria uma tarefa custosa. Porém, na realidade só precisamos atualizar os nodos que estão sendo utilizados durante uma consulta. Para isso será necessário um vetor auxiliar chamado de $lazy$ que guarda informações sobre as atualizações que podem ser feitas posteriormente.

Ao descer pela árvore em busca dos nodos que compõem o intervalo $(i, j)$ devemos aplicar as modificações pendentes do vetor $lazy$ em nodos que precisam ser atualizados. Ao encontrar um nodo $u$ que compõe o intervalo $(i, j)$ devemos atualizar o valor do máximo do intervalo de $u$ para $x$, já que todos os valores naquela subárvore devem ser modificados para $x$. Para os filhos de $u$, o vetor de $lazy$ que guarda informações sobre as modificações feitas naquele intervalo deve ser atualizado para $x$. As atualizações em $lazy$ são feitas em tempo $\mathcal{O}(1)$ e o intervalo $(i, j)$ é coberto por no máximo $\mathcal{O}(\log n)$ nodos. Totalizando uma complexidade de $\mathcal{O}(\log n)$ por consulta. Inicialmente, seja $V = [1, 7, -1, 4, -5, 10, 3, -7]$ e sua *Segment Tree* seja $T = [10, 7, 10, 7, 4, 10, 3, 1, 7, -1, 4, -5, 10, 3, -7]$.

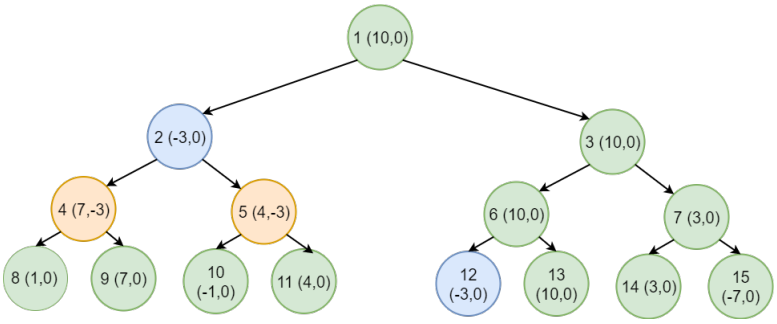
Figura 5.4: Visualização de $T$ para $V = [1, 7, -1, 4, -5, 10, 3, -7]$




Onde os valores $i(x, y)$ representam o nodo $i$ com máximo $x$ e com $lazy$ $y$ (note que aqui o valor 0 no vetor $lazy$ representa o valor nulo, ou seja uma ausência de pendências, para aplicações em problemas é suficiente utilizar um valor fora do domínio

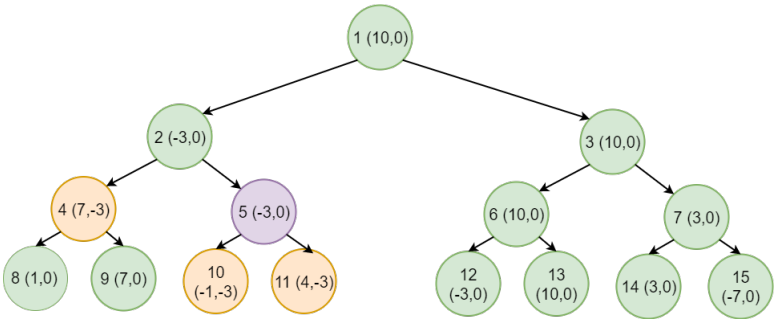
do problema como valor nulo). Ao fazer uma operação de modificação do tipo $(1, 5, -3)$ o estado da árvore é modificado para o seguinte, conforme a Figura 5.5.

Figura 5.5: Estado da árvore após uma modificação em $(1, 5, -3)$



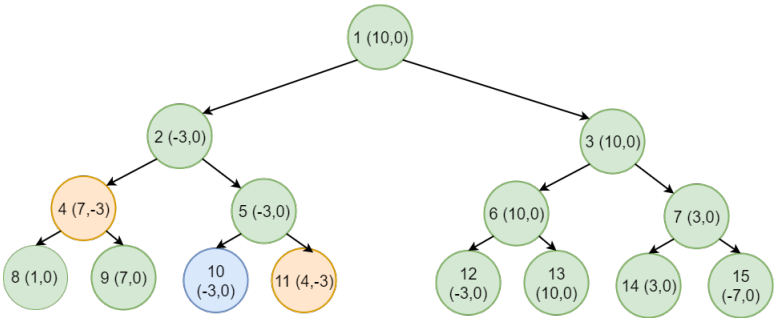

Os nodos em azul compõem o intervalo $(1, 5)$ e foram modificados imediatamente. Já os nodos em laranja precisam ser atualizados, como pode ser notado pelo valor de $lazy = -3$ nos nodos 4 e 5. Mas perceba que essa modificação na subárvore de 2, dos nodos $4, 5, 8, 9, 10, 11$ não precisa ser feita imediatamente. Na verdade, essa modificação só precisa ser feita quando passarmos por algum desses nodos em alguma outra operação. Observe na figura 5.6 o comportamento da árvore ao fazer uma consulta de máximo no intervalo $(3, 3)$.

Figura 5.6: Estado da árvore ao chegar no nodo 5 durante a consulta de máximo em $(3, 3)$




Ao chegar no nodo 5 (denotado pela cor roxa), atualizamos o valor de $T_5$, propagamos o valor de $lazy_5$ para ambos os seus filhos, definimos o valor de $lazy_5$ para o valor nulo (indicando que sua modificação já foi propagada) e continuamos a busca até o nodo 10.

Figura 5.7: Estado da árvore ao chegar no nodo 10 durante a consulta de máximo em $(3, 3)$




Chegando no nodo 10, seu valor é atualizado junto com o valor de $lazy$, e em seguida tem-se a conclusão da consulta de máximo em $(3, 3)$. Dessa forma, conseguimos manter todas as propriedades do intervalo de máximo na árvore e conseguimos reduzir a complexidade de uma operação de modificação em intervalo para $\mathcal{O}(t \log n)$, onde $t$ é a complexidade de tempo para aplicar a operação $\oplus$. Com esse tipo de modificação em intervalo, algumas propriedades devem ser mantidas ao fazer operações na árvore:

1. Ao passar por um nodo $u$ em qualquer tipo de operação (consulta ou modificação) a atualização do nodo $u$ deve ser feita e propagada para seus filhos esquerdo e direito. Dessa maneira, garantimos que o valor de um nodo está correto ao avaliá-lo, e que a sua modificação foi propagada para a sua subárvore.

2. Ao fazer essa atualização, o valor do nodo pode ser alterado. Como foi visto no exemplo anterior, após a atualização do intervalo $(1, 5)$ com o valor $-3$ não necessariamente todos os nodos dessa subárvore possuíam o valor de $-3$ internamente. Essa atualização dos valores é feita quando tais nodos forem necessários em operações seguintes.

## 5.3 Implementação

A construção de uma árvore segmentada para a operação $\oplus = max$ pode ser feita pela Equação 5.2, onde recebemos o vetor representando a árvore segmentada $T$, e o vetor inicial $V$ de tamanho $n$. A construção é feita a partir da posição 1 com intervalo $[1...n]$. O valor de uma posição $T_i$ só será avaliado após a construção do filho esquerdo e direito de $T_i$. Dessa maneira, para fazer a avaliação do valor de $T_i$, basta utilizar os valores de seus filhos. A chamada inicial é feita por $build\_segment\_tree(1, 1, n, V, T)$.

```
int build_segment_tree(int u, int L, int R, vector<int>& V, vector<int>& T){
    if(L == R)
        return T[u] = v[L];
    else{
        int mid = (L+R)/2;
        int esq = build_segment_tree(2*u, L, mid, V, T);
        int dir = build_segment_tree(2*u+1, mid+1, R, V, T);
        return T[u] = max(esq, dir);
    }
}
```

Passamos por cada nodo somente uma vez e, como temos no máximo $\mathcal{O}(n)$ nodos, a construção é feita em tempo $\mathcal{O}(n)$.

### 5.3.1 Consulta de máximo em intervalo

Seja $V$ um vetor de inteiros de tamanho $n$, $(a, b)$ um intervalo de $V$ e $T$ o vetor correspondente à sua *Segment Tree*. Para fazer o cálculo de $max(V_a, V_{a+1}, ..., V_b)$ podemos utilizar a Equação 5.3, onde recebemos como parâmetros o intervalo que deve ser avaliado, e o vetor $T$.

```
    int query(int u, int L, int R, int i, int j, vector<int>& T){
        if(R > i or j < L) return 0;
        if(L >= i and R <= j) return T[u];
        else{
            int mid = (L+R)/2;
            int esq = query(2*u, L, mid, i, j, T);
            int dir = query(2*u+1, mid+1, R, i, j, T);
            return max(esq, dir);
        }
    }
```

Assim como na Equação 5.3, temos que a complexidade é $\mathcal{O}(\log n)$, e uma chamada da função $query(1, 1, n, a, b, T)$ é suficiente para fazer o cálculo de máximo no intervalo $(a, b)$.

### 5.3.2 Modificação de um elemento

Seja $V$ um vetor de inteiros de tamanho $n$, $T$ a *Segment Tree* correspondente à $V$, $pos$ uma posição que deve ser modificada e $x$ o valor que deve ser fixado em $V_{pos}$. Para fazer a atualização de uma posição no vetor $V$ podemos utilizar a Equação 5.4, recebendo como parâmetros iniciais $update(1, 1, n, pos, x, T)$.

```
int update(int u, int L, int R, int pos, int x, vector<int>& T){
    if(pos > R or pos < L) return T[u];
    if(L == R) return T[u] = x;
    else{
        int mid = (L+R)/2;
        int esq = update(2*u, L, mid, pos, x, T);
        int dir = update(2*u+1, mid, R, pos, x, T);
        return T[u] = max(esq, dir);
    }
}
```

Ao fazer uma mudança em algum filho de $T_u$ é necessário recalcular a operação $max$ em seu filho esquerdo e direito e atualizar esse valor para o nodo $T_u$. Como na equação 5.4, no máximo $\mathcal{O}(\log n)$ serão modificados. Repare que a modificação na posição $pos$ não é feita no vetor $V$. Logo, se for necessário computar o valor de $V_i$ após algumas operações, deve-se fazer uma consulta de máximo no intervalo $(i, i)$ de $T$.

### 5.3.3 Modificação de um intervalo

Seja $V$ um vetor de inteiros de tamanho $n$, $T$ a *Segment Tree* correspondente à $V$, $(i, j, x)$ o intervalo $V_{i...j}$ que deve ser modificado para $x$ e $lazy$ o vetor de pendências da árvore (que deve ser inicializado com um valor fora do domínio dos elementos, durante a implementação esse valor será tratado como 0). Inicialmente será construída a função $push$ para fazer a atualização de um nodo, onde recebemos o nodo $u$ que deve ser atualizado, seu intervalo $L_u$ e $R_u$, o vetor de $lazy$ e a árvore $T$.

```
void push(int u, int L, int R, vector<int>& lazy, vector<int>& T){
    if(lazy[u] != 0){
        T[u] = lazy[u];
```

```
4         if(L != R){
5             lazy[2*u] = lazy[u];
6             lazy[2*u+1] = lazy[u];
7         }
8         lazy[u] = 0;
9     }
10 }
```

Caso o nodo $u$ possua uma pendência em $lazy_u$, devemos atualizá-lo e, se $u$ não for uma folha (ou seja, se $L_u \neq R_u$), devemos propagar a mudança para seus filhos. Com isso, devemos remover a pendência de $u$ em $lazy$. Com a função de atualização feita, poucas alterações devem ser aplicadas nas funções de modificação e consulta. Para a função de consulta, as únicas modificações que devem ser efetuadas são na primeira linha da função, onde atualizamos o nodo que a função está percorrendo e a adição do vetor $lazy$ na recursão.

```
1  int query(int u, int L, int R, int i, int j, vector<int>& lazy, vector<int>& T){
2      push(u, L, R, lazy, T);
3      if(R > i or j < L) return 0;
4      if(L >= i and R <= j) return T[u];
5      else{
6          int mid = (L+R)/2;
7          int esq = query(2*u, L, mid, i, j, lazy, T);
8          int dir = query(2*u+1, mid+1, R, i, j, lazy, T);
9          return max(esq, dir);
10     }
11 }
```

Assim como na função de consulta, a função de modificação sofrerá poucas modificações. Deve-se atualizar as pendências de cada nodo percorrido, e ao encontrar um dos nodos $u$ que compõem o intervalo $(i, j)$ deve-se atualizar o nodo $u$ e em seguida propagar as suas novas pendências.

```
1  int update(int u, int L, int R, int i, int j, int x, vector<int>& lazy, vector<int>& T){
2      push(u, L, R, lazy, T);
3      if(R > i or j < L) return T[u];
4      if(L >= i and R <= j){
```

```
5           lazy[u] = x;
6           push(u, L, R, lazy, T);
7           return T[u];
8       }
9       else{
10          int mid = (L+R)/2;
11          int esq = update(2*u, L, mid, i, j, x, lazy, T);
12          int dir = update(2*u+1, mid+1, R, i, j, x, lazy, T);
13          return max(esq, dir);
14      }
15  }
```

## 5.4 Problema Relacionado

No problema *Fundraising* da etapa nacional da maratona de programação 2017 (COSTA, 2017), é necessário o conceito de *Segment Tree* (note que o problema também pode ser resolvido com *Fenwick Tree* com algumas observações) para sua solução. Uma descrição simplificada do problema é a seguinte:

> Dados $n$ possíveis convidados para uma festa, onde o $i$-ésimo convidado possui um nível de beleza $x_i$, um nível de riqueza $y_i$ e uma contribuição de $w_i$, escolha um subconjunto dos convidados de maneira a maximizar a soma de suas contribuições de forma que entre os convidados escolhidos não exista uma relação ruim. Uma relação ruim entre dois convidados escolhidos surge caso uma das seguintes condições sejam verdadeiras:
>
> 1. Para um convidado escolhido $i$, existe um outro convidado escolhido $j$, tal que $i \neq j$ e $x_i < x_j$ e $y_i \geq y_j$;
> 2. Para um convidado escolhido $i$, existe um outro convidado escolhido $j$, tal que $i \neq j$ e $x_i > x_j$ e $y_i \leq y_j$.
>
> Ou seja, um convidado $i$ tem uma relação boa com um convidado $j$ se ele é estritamente mais rico e mais belo que $j$, ou ele é estritamente menos rico e menos belo que $j$, ou $i$ e $j$ são iguais em beleza e riqueza. As restrições de valores são : $1 \leq x_i, y_i \leq 10^9$ e $1 \leq n \leq 10^5$.

A primeira observação que pode ser feita no problema é que convidados que possuem o mesmo nível de beleza e riqueza podem ser fundidos em um novo convidado com suas contribuições somadas. Dessa maneira, temos que para um candidato $i$ ser escolhido ele precisa ser estritamente mais belo e rico que todos os outros, ou estritamente menos belo e menos rico que todos os outros. Em seguida iremos ordenar os convidados pelas suas riquezas em ordem crescente e em caso de empate serão ordenados pelas belezas em ordem decrescente. O desempate é feito dessa maneira para garantir que um convidado $(x, y_1, w_1)$ não utilize um outro convidado $(x, y_2, w_2)$, onde $y_1 > y_2$. Por exemplo, os convidados $(2, 2, 10)$, $(2, 4, 8)$ e $(3, 5, 15)$, onde $(x, y, w)$ representa o convidado com riqueza $x$, beleza $y$ e contribuição $w$, quando ordenados se tornariam $(2, 4, 8)$, $(2, 2, 10)$ e $(3, 5, 15)$.

Com isso, iremos processar cada um dos convidados de acordo com a ordenação utilizada. Isso nos garante que ao processar o convidado $i$ todos os convidados estritamente menores que $i$ já foram processados, e podemos utilizar as respostas dos convidados anteriores para computar o valor de adicionar o convidado $i$ na resposta. Mas note que, ao processar um convidado $i$ temos que todo convidado $j$, onde $j < i$, possui $x_j \leq x_i$ mas não necessariamente $y_j < y_i$. Para resolver esse problema iremos manter um vetor $V$ (inicialmente inicializado com $-\infty$), onde $V_i$ irá guardar o valor da melhor soma de contribuições possíveis quando todos os candidatos utilizados possuem um nível de riqueza menor que $i$ e não existe alguma relação ruim entre os candidatos. Assim, se estamos processando o convidado $i$, sabemos que a melhor soma de contribuição que inclui o convidado $i$ será $w_i + max(V_1, ..., V_{y_i-1})$ e em seguida devemos atualizar a posição $V_{y_i}$ com $max(V_{y_i}, w_i + max(V_1, ..., V_{y_i-1}))$. Isto se mantém válido pois, todos os convidados processados anteriormente possuem uma riqueza menor que $i$, e caso a riqueza de algum valor anterior seja igual a de $i$, devido à nossa ordenação iremos processar primeiro os candidatos de maior beleza. Dessa maneira, os candidatos com uma riqueza igual não irão afetar uns aos outros.

Para fazer as operações de atualização e de máximo no vetor $V$, simularemos o vetor $V$ usando o conceito de *Segment Tree*. Como a coordenada $y$ pode ir até $10^9$, criar um vetor com este tamanho é inviável em um ambiente de competição. Logo, será necessário utilizar o conceito de compressão de coordenada abordado na Seção 4.4. Repare que após a compressão, as relações se mantêm válidas, pois se $x_i < x_j$ antes da compressão, após a compressão esta relação permanece válida. A mesma ideia se aplica para os valores de $y$.

# 6 Treap

Seja $V$ um vetor de inteiros e seja $q$ o número de consultas que devem ser feitas. Uma consulta pode ser definida por uma das seguintes operações, as quais podem ser feitas em uma ordem qualquer:

1. Dado um par $(i, x)$, onde $1 \leq i \leq k+1$ e $k$ é o tamanho atual do vetor $V$, inserir um elemento de valor $x$ na posição $i$, fazendo com que todos os elementos de índice $j \geq i$ se desloquem para a direita;

2. Dada uma posição $i$, onde $1 \leq i \leq k$ e $k$ é o tamanho atual do vetor $V$, remover o elemento da posição $i$, fazendo com que todos os elementos de índice $j \geq i$ se desloquem para a esquerda;

3. Dado um par $(a, b)$, onde $1 \leq a \leq b \leq k$ e $k$ é o tamanho atual do vetor $V$, calcular $max(V_a, V_{a+1}, ..., V_b)$.

Por exemplo, inicialmente seja $V = [3, 5, -1, 0, 3, 1]$. Após uma consulta do tipo 1 com um par $(i, x) = (2, 10)$ o vetor $V$ seria $V = [3, 10, 5, -1, 0, 3, 1]$. Uma consulta do tipo 2 com valor $i = 5$ em seguida faria o vetor $V$ ser $V = [3, 10, 5, -1, 3, 1]$. Por fim, uma consulta do tipo 3 com um par $(a, b) = (4, 6)$ teria resultado $max(-1, 3, 1) = 3$. Note que após uma consulta do tipo 1 o tamanho do vetor $V$ será incrementado em uma posição, e após uma consulta do tipo 2 o tamanho de $V$ seria decrementado por uma posição.

O problema motivacional da estrutura *Treap* é análogo ao problema motivacional da estrutura *Segment Tree*, porém as operações do tipo 1 e 2 podem alterar a estrutura do vetor $V$ e não somente modificar seus valores (repare que uma operação de modificação pode ser simulada como uma operação de remoção seguida de uma operação de inserção com o elemento modificado). Dessa maneira, temos que nenhuma das outras estruturas apresentadas são robustas o suficiente para resolver esse tipo de problema. Uma solução inicial pode ser feita através de uma simulação das operações: mantemos um vetor $V$ durante as operações, para cada operação do tipo 1 inserimos o elemento de maneira trivial em $\mathcal{O}(k)$; para cada operação do tipo 2 removemos o elemento de maneira trivial em $\mathcal{O}(k)$, e; para as operações do tipo 3 calculamos o máximo de um intervalo também

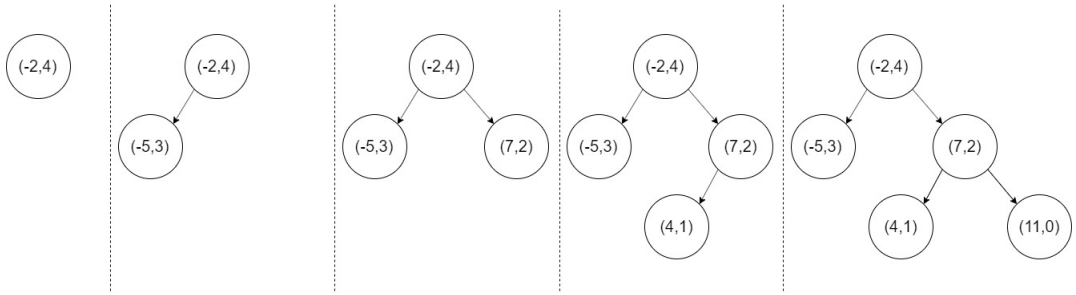
trivialmente em $\mathcal{O}(k)$. No total, temos $q$ consultas onde cada uma delas é respondida em tempo $\mathcal{O}(k)$, logo a complexidade de tempo desse algoritmo é de $\mathcal{O}(qk)$ no pior caso. Veremos em seguida como é possível reduzir a complexidade de tempo desse problema para $\mathcal{O}(q \log k + k \log k)$ e como fazer operações de modificação no intervalo, como inversão de segmentos, com o conceito de *Treap*.

## 6.1 Definição da estrutura

A *Treap* (*Binary Tree* + *Binary Heap*) é uma estrutura de dados robusta e dinâmica capaz de suportar operações em intervalo que sejam associativas e não necessariamente reversíveis. Assim como a *Segment Tree*, suporta operações de modificação em intervalo com a adição do conceito $lazy$, e pode ser generalizada para mais dimensões (apesar de não ser comum em ambientes de competição). Com o conceito de *Treap Implícita*, suporta também outros tipos de operações, como inserção e remoção de elementos, reversão de intervalos do vetor, Fusão de árvores e Quebra de árvores, todas estas em tempo $\mathcal{O}(\log n)$.

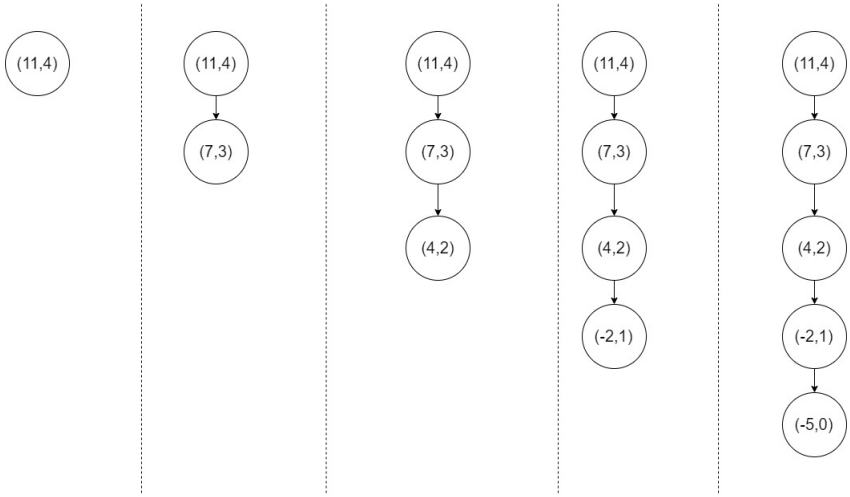
De acordo com sua definição inicial (ARAGON; SEIDEL, 1989), uma *Treap* $T$ é uma estrutura que guarda pares do tipo $(x, y)$, onde os valores de $x$ são chamados de chaves e os valores de $y$ são chamados de prioridade. Denotaremos a chave de um nodo $u$ por $u.chave$ e a prioridade de um nodo $u$ como $u.prioridade$, assim como $u.left$ para seu filho esquerdo e $u.right$ para seu filho direito. A estrutura é construída de modo que $T$ seja uma árvore binária de busca pelos valores de $x$ e uma árvore *heap* pelos valores de $y$. Ou seja, para ser uma árvore binária pelas chaves, para todo nodo $u$ de $T$, temos que $v.chave \leq u.chave$ para todo $v$ na sub-árvore esquerda de $u$, e que $u.chave \leq w.chave$ para todo $w$ na sub-árvore direita de $u$. Já para ser uma árvore *heap* pelas suas prioridades, para todo nodo $u$ de $T$, temos que $u.prioridade \geq u'.prioridade$ para qualquer nodo $u'$ de alguma sub-árvore de $u$. Repare que $T$ é unicamente determinada pela prioridade de seus nodos (assumimos que todas as prioridades são distintas). Assim, $T$ é a árvore de busca binária resultante da adição dos elementos $(x, y)$ em ordem decrescente de suas prioridades e utilizando a comparação de nodos baseados em chaves.

Figura 6.1: Visualização da construção de $T$ utilizando a prioridade de seus pares em ordem decrescente para os seguintes pares: $(-2, 4)$, $(-5, 3)$, $(7, 2)$, $(4, 1)$ e $(11, 0)$




Utilizando as prioridades $[4, 3, 2, 1, 0]$ para as chaves $[-2, -5, 7, 4, 11]$, temos uma *Treap* $T$ de profundidade 2 como é mostrado na Figura 6.1. Porém, note que se as prioridades fossem $[1, 0, 3, 2, 4]$ para as mesmas chaves teríamos a árvore da Figura 6.2 que possui profundidade 4.

Figura 6.2: Visualização da construção de $T$ utilizando a prioridade de seus pares em ordem decrescente para os seguintes pares: $(-2, 1)$, $(-5, 0)$, $(7, 3)$, $(4, 2)$ e $(11, 4)$




Assim, temos que duas propriedades podem ser observadas:

1. Operações em árvore, tais como inserções, remoções, buscas e modificações dependem intrinsecamente da profundidade da árvore que está sendo utilizada;

2. O valor de prioridade utilizado em cada chave pode alterar a profundidade da *Treap* $T$.

Se já são conhecidos os elementos que irão ser adicionados antes da construção de $T$, podemos designar prioridades para fazer com que a profundidade de $T$ seja $\mathcal{O}(\log n)$. Porém, como é visto no problema motivacional, nem sempre é possível ter o conhecimento prévio

de todos os elementos que serão adicionados. Contudo, se ao inserir um novo nodo $u$ na árvore utilizarmos uma prioridade pseudo-aleatória, teremos em média uma profundidade de $\mathcal{O}(\log n)$ (ARAGON; SEIDEL, 1989), tornando-se viável a utilização desse conceito quando não se tem prévio conhecimento dos elementos que serão adicionados.

## 6.2 Operações da estrutura

Para ser possível a inserção/remoção de elemento primeiramente serão definidas as operações de fusão e quebra de árvores, pois essas operações serão úteis posteriormente para outros tipos de operações, tal como o máximo em intervalo. Outras operações que não serão abordadas nesse trabalho também se tornam triviais com essa abordagem, tais como a atualização em intervalo, inversão de segmento e remoção de segmento.

### 6.2.1 Fusão de árvores

Sejam $T_1$ e $T_2$ duas *Treaps*, onde as chaves de $T_1$ são menores que as chaves $T_2$, ou seja para cada nodo $u$ de $T_1$ e para cada nodo $v$ de $T_2$ temos que $u.chave \leq v.chave$. Desejamos fazer a fusão das duas estruturas em uma *Treap* $T$ que possua todos os nodos de $T_1$ e $T_2$. Iremos construir $T$ de cima para baixo, isto é, começaremos pelas raízes das duas árvores e a cada passo do algoritmo iremos decidir a ordem de como cada sub-árvore será adicionada em $T$. Inicialmente as sub-árvores que serão analisadas serão $esq = T_1$ e $dir = T_2$ e a fusão de $esq$ com $dir$ será armazenada no nodo $T$. A cada passo teremos duas possibilidades:

1. $esq$ é um nodo nulo: sabemos que $dir$ é a única opção como sub-árvore;
2. $dir$ é um nodo nulo: sabemos que $esq$ é a única opção como sub-árvore;
3. $esq.prioridade > dir.prioridade$: temos que $T$ deve ser equivalente à $esq$, exceto pela sub-árvore direita de $T$ que deve ser o resultado da fusão de $esq.right$ e $dir$;
4. $esq.prioridade < dir.prioridade$: temos que $T$ deve ser equivalente à $dir$, exceto pela sub-árvore esquerda de $T$ que deve ser o resultado da fusão de $esq$ e $dir.left$.

Figura 6.3: Visualização da fusão de duas *Treaps*. Os nodos laranjas são os que estão sendo avaliados no momento, e os nodos verdes são os nodos fixados na *Treap* resultante. Na primeira etapa temos as duas árvores $esq$ e $dir$. Na segunda etapa, pela prioridade de $esq$ ser superior, temos que $T = esq$ e que $T.right$ é equivalente ao resultado da fusão da sub-árvore de $esq.right$ e $dir$.

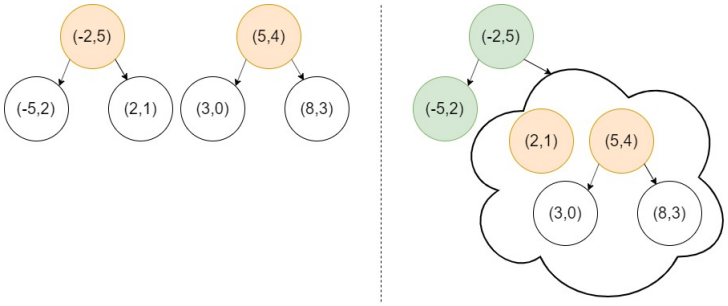
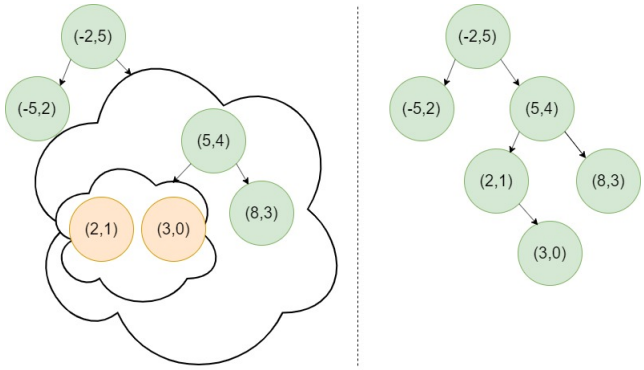



Figura 6.4: Na terceira etapa, pela prioridade de $dir$ ser superior, temos que $T.right = dir$ e $T.right.left$ deve ser equivalente a fusão de $dir.left$ e $esq.right$. Na quarta etapa, após todas as computações serem feitas temos a árvore resultante.




Nos itens 1 e 2, pode-se observar de forma trivial que as propriedades de uma *Treap* serão mantidas, visto que $dir$ e $esq$ já eram *Treaps* válidas antes da fusão. Para o item 3 precisamos manter duas propriedades: as chaves em uma árvore binária de busca, e as prioridades em uma árvore *heap*. Ao fundir uma árvore $esq$ com $dir$ onde $esq.prioridade > dir.prioridade$, temos obrigatoriamente que $esq$ deve estar acima de $dir$, mantendo assim a propriedade de uma árvore *heap*. Com isso, podemos inserir $dir$ na sub-árvore esquerda de $esq$ ou na sub-árvore direita de $esq$. Porém, note que $dir$, por possuir chaves maiores do que $esq$, não pode estar em sua sub-árvore esquerda, e então é inserido na sub-árvore direita de $esq$ onde a fusão de $esq.right$ e $dir$ deve acontecer recursivamente, mantendo a propriedade de uma árvore binária de busca. A prova para o item 4 é simétrica à prova do item 3. Faremos no máximo $k$ recursões, onde $k$ é igual ao máximo entre as profundidades de $T_1$ e $T_2$ no pior caso. Como em média a profundidade de uma *Treap* é $\mathcal{O}(\log n)$, temos uma complexidade de tempo esperada de $\mathcal{O}(\log n)$.

### 6.2.2 Quebra de árvores

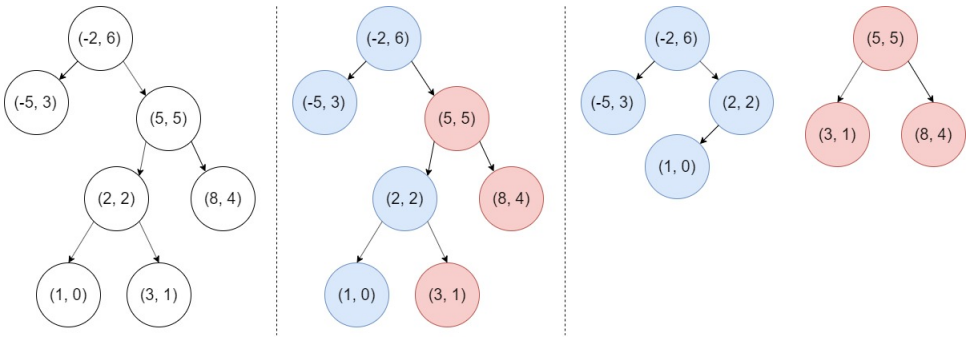
Seja $T$ uma *Treap* e $x$ um inteiro qualquer. Desejamos quebrar a *Treap* $T$ em duas outras *Treaps* $T_1$ e $T_2$, onde todas as chaves dos nodos de $T_1$ são menores que $x$ e todas as chaves dos nodos de $T_2$ são maiores ou iguais a $x$. Além disso, todo nodo $u$ pertencente a $T$ deve pertencer exclusivamente à $T_1$ ou à $T_2$. A construção será feita a partir da raiz de $T$, e a cada etapa iremos construir recursivamente as árvores $T_1$ e $T_2$. Se estamos atualmente em um nodo $u$ (inicialmente $u = T$) durante a Quebra de árvores, temos as seguintes possibilidades:

1. $u$ é um nodo nulo: então sabemos que $(T_1, T_2) = (nulo, nulo)$;

2. $u.chave < x$: temos que $T_1$ deve conter $u$, porém ainda há nodos na sub-árvore direita de $u$ que podem pertencer a $T_2$, logo precisamos das árvores $(T'_1, T'_2)$ referentes à quebra de árvore de $u.right$;

3. $u.chave \geq x$: temos que $T_2$ deve conter $u$, porém ainda há nodos na sub-árvore esquerda de $u$ que podem pertencer a $T_1$, logo precisamos das árvores $(T'_1, T'_2)$ referentes à quebra de árvore de $u.left$.

Figura 6.5: Visualização da quebra de uma *Treap* para uma constante $x = 3$. Na primeira etapa temos a *Treap* $T$. Na segunda etapa os nodos são marcados para suas árvores correspondentes, onde nodos azuis possuem valor de chave menor que $x$ e os nodos vermelhos possuem valor de chave maior ou igual a $x$ . Na terceira etapa a reconstrução é feita mantendo todos os nodos de uma mesma cor em uma mesma árvore.




O item 1 é trivialmente verdadeiro. Para o item 2 temos que $u.chave < k$ e como sabemos que todo nodo $v$ pertencente a $u.left$ possui $v.chave < u.chave < k$, sabemos que $u$ e toda a sua sub-árvore esquerda devem pertencer a $T_1$. Porém, isso não se aplica para $u.right$, visto que pode existir algum elemento $w$ que pertence a $u.right$ tal que $u.chave < k < w.chave$. Para resolver esse tipo de conflito e encontrar os nodos $z$ de

$u.right$ tal que $u.chave < z.chave < k$ é necessário achar a quebra de árvore $(T_1', T_2')$ de $u.right$. Com isso, temos que $T_1 = u$ exceto pela sua sub-árvore direita, que deve ser $T_1.right = T_1'$ pois todo nodo $z$ em $T_1'$ possui $u.chave < z.chave < k$, e que $T_2 = T_2'$ visto que $u$ e toda sua sub-árvore esquerda são menores que $k$ a única sub-árvore possível com chaves maiores ou iguais a $k$ é computada pela quebra de $u.right$. A propriedade de árvore *heap* é sempre mantida, visto que uma sub-árvore $B$ será adicionada a uma sub-árvore $A$ somente se $B$ fazia parte da sub-árvore de $A$ em $T$, logo para todo nodo $b$ que pertence a $B$ deve ser verdade que $b.prioridade < A.prioridade$. Já a propriedade de árvore binária de busca é mantida, dado que $T_1'$ será adicionado à $T_1.right$, e todo nodo $z$ de $T_1'$ respeita $A.chave < z.chave < k$. A prova para o item 3 é simétrica à prova para o item 2. Faremos no máximo $k$ recursões em um pior caso, onde $k$ é a profundidade da *Treap* $T$. Em média, a complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### 6.2.3 Inserção de elemento

Seja $T$ uma *Treap* e $x$ um inteiro qualquer. Desejamos inserir um nodo $(x, y)$ em $T$, onde $y$ é um número pseudo-aleatório qualquer. Com as operações de Fusão e Quebra de árvores definidas, a inserção de um elemento se dá trivialmente por 3 operações. Inicialmente fazemos uma quebra de árvore pelo valor $x$ em $T$ e obtemos duas árvores $T_1$ e $T_2$ referentes aos valores menores que $x$ e os valores maiores ou iguais a $x$, respectivamente. Em seguida, fazemos a fusão de $T_1$ com o nodo $(x, y)$ resultando em uma *Treap* $T'$. Isso é válido, pois todos as chaves de $T_1$ são menores que $x$. Finalmente fazemos a fusão de $T'$ com $T_2$. Como todas as chaves de $T'$ são menores ou iguais as chaves de $T_2$ essa operação também se mantém válida.

Figura 6.6: Visualização da inserção de um nodo $(4, 13)$ em uma *Treap* $T$. O nodo laranja é o que deve ser adicionado, os nodos azuis são os nodos referentes a $T_1$ e os nodos vermelhos são os nodos referentes a $T_2$. Nas duas primeiras etapas fazemos a quebra de $T$ em $T_1$ e $T_2$.

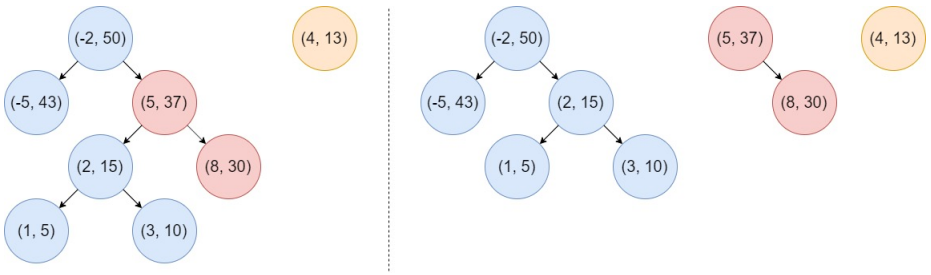
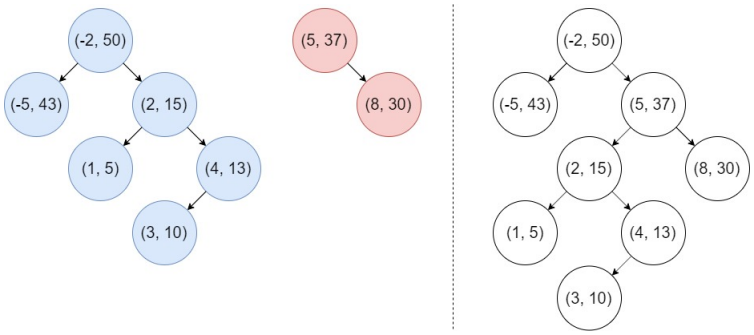


Figura 6.7: Nas duas últimas etapas fazemos a fusão de $T_1$ com $(4, 13)$ resultando em $T'$, e a fusão de $T'$ com $T_2$. A ultima figura representa a *Treap* final após a inserção de $(4, 13)$.

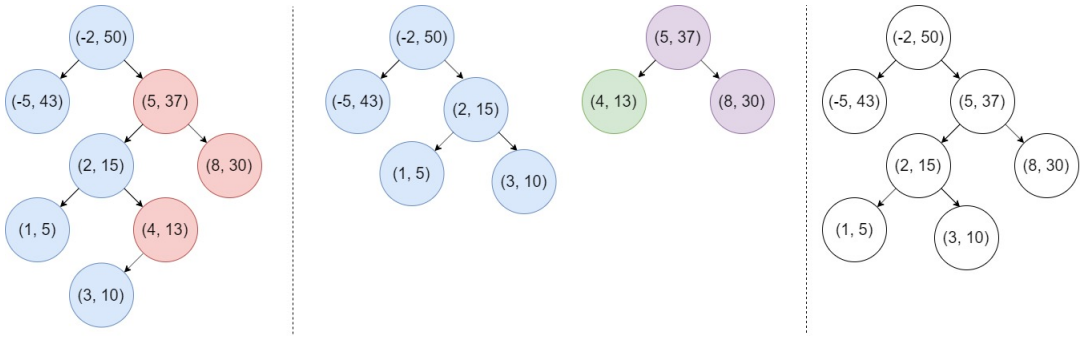



Como faremos 3 operações de quebra/fusão, a complexidade esperada é de $\mathcal{O}(\log n)$.

### 6.2.4 Remoção de elemento

Seja $T$ uma *Treap* e $x$ um inteiro qualquer que representa a chave de algum nodo de $T$. Desejamos obter a *Treap* resultante de remover o nodo que possui chave $x$. Analogamente à inserção de um elemento, podemos desconstruir a tarefa de remoção de um elemento em uma sucessão de operações de Fusão e Quebra de árvore. Inicialmente, quebramos a *Treap* $T$ em duas *Treaps* $T_1$ e $T_2$, as quais contêm os valores de chave menores que $x$ e os valores de chave maiores ou iguais a $x$, respectivamente. Em seguida, quebramos a *Treap* $T_2$ baseado no valor de chave $x + 1$. Assim, teremos duas novas *Treaps*, $T_1'$ e $T_2'$. Observe que $T_1'$ é totalmente composta por nodos que possuem chave igual a $x$. Por fim, fazemos a fusão de $T_1$ com $T_2'$, mantendo todos os nodos que possuem chave diferente de $x$. Caso seja necessário fazer a remoção somente de um nodo que possua chave $x$ e não todos, basta fazer a fusão dos filhos de $T_1'$ com $T_1$ e $T_2'$.

Figura 6.8: Visualização da remoção do nodo com chave $x = 4$ de uma *Treap* $T$. Na primeira etapa temos a quebra de $T$ em $T_1$ em azul e $T_2$ em vermelho. Na segunda etapa temos a quebra de $T_2$ em $T_1'$ em verde e $T_2'$ em roxo. Na ultima etapa temos a *Treap* resultante da fusão de $T_1$ e $T_2'$.




Assim como na inserção de elementos, faremos 3 operações de fusão/quebra de árvore totalizando uma complexidade esperada de $\mathcal{O}(\log n)$.

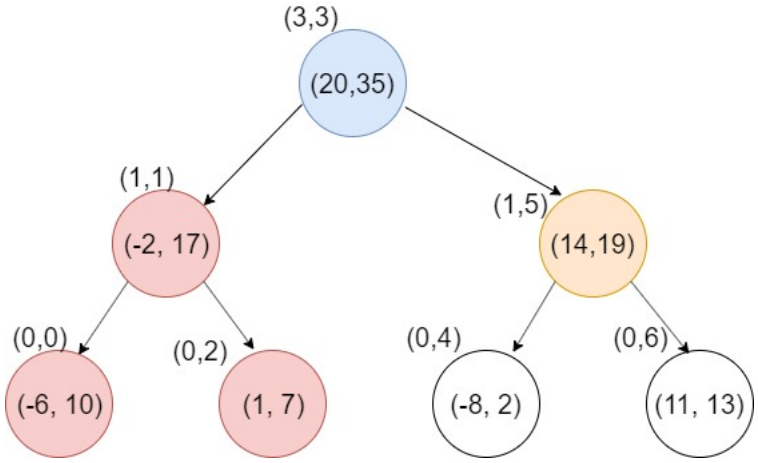
### 6.2.5 *Treap* implícita

Para a resolução do problema motivacional, é necessária uma estrutura capaz de simular um vetor de elementos. Ao contrário das estruturas anteriores, onde uma árvore é simulada por um vetor (como na *Segment Tree* e *Fenwick Tree*), teremos que um vetor $V$ será simulado pela *Treap* implícita $T$. Uma *Treap* implícita $T$ é uma *Treap* onde $T$ simula as posições $[V_1...V_n]$ de $V$ tal que $|V| = n$. Com isso, $T$ é capaz de simular operações em $V$, tais como: inserção de um elemento em uma posição qualquer, remoção de um elemento em uma posição qualquer, máximo em um intervalo, inversão de intervalo e modificação em segmento em tempo $\mathcal{O}(\log n)$.

Para comportar esse tipo de operação, $T$ deve perder a propriedade de ser uma árvore binária de busca por sua chave para ser uma árvore binária de busca pela posição que um nodo representa, já que agora o que define um nodo estar na sub-árvore esquerda ou na sub-árvore direita de um outro nodo é a posição que ele deve assumir no vetor $V$. Assim como em um vetor, onde uma posição $V_i$ é definida pela quantidade de elementos à sua esquerda, ou seja, a $i$-ésima posição de $V$ possui $i - 1$ elementos à sua esquerda, temos que um nodo $u$ de $T$ que corresponde à posição $V_i$ deve possuir $i - 1$ nodos à sua esquerda em $T$. Por exemplo, o nodo que representa $V_1$ não deve possuir nenhum nodo à sua esquerda, já o nodo que representa a posição $V_3$ deve possuir 2 nodos à sua esquerda. Note que as funções de *Treap* devem ser adaptadas para tratar a *Treap* implícita. Dessa

maneira, será necessário manter um valor adicional em cada nodo $u$, além dos valores de chave e prioridade, que será o tamanho de sua sub-árvore denotado por $u.size$.

Figura 6.9: Visualização da *Treap* implícita $T$ de um vetor $V = [-6, -2, 1, 20, -8, 14, 11]$. O par de valores $(a, b)$ acima de cada nodo representa o número de nodos na sub-árvore de seu filho esquerdo e o numero de nodos da árvore $T$ que estão à sua esquerda, respectivamente.




Repare que um nodo $u$ de $T$ que representa uma posição $V_i$ não necessariamente possui $i - 1$ filhos em sua sub-árvore esquerda, como é visto no nodo laranja da Figura 6.9 que representa a posição $V_6$ e possui somente 1 nodo em sua sub-árvore esquerda. Seja $leftsize(u)$ o número de nodos de $T$ que estão à esquerda de $u$ e não fazem parte da sub-árvore esquerda de $u$. Por exemplo, o valor de $leftsize$ do nodo laranja seria 4 (os nodos em vermelhos e o nodo em azul), já o valor de $leftsize$ do nodo azul seria 0 (todos os nodos à sua esquerda em $T$ estão em sua sub-árvore esquerda). Temos que o valor de $leftsize(v)$ onde $v$ é o filho esquerdo de $u$ deve ser $leftsize(v) = leftsize(u)$, pois $leftsize(u)$ contabiliza todos os nodos que estão à esquerda de $u$ e não estão em sua sub-árvore esquerda (por consequência os nodos que estão à esquerda de $v$ e não estão na sub-árvore esquerda de $v$). Já o valor de $leftsize(w)$ onde $w$ é o filho direito de $u$ deve ser $leftsize(w) = leftsize(u) + u.left.size + 1$, pois $leftsize(u)$ contabiliza todos os nodos que estão à esquerda de $u$ e não estão em sua sub-árvore esquerda (por consequência os nodos que estão à esquerda de $w$ e não estão na sub-árvore esquerda de $w$), $u.left.size$ contabiliza os nodos na sub-árvore esquerda de $u$ e 1 para contabilizar o próprio nodo $u$, pois ambos estão à esquerda de $w$. Com isso, a posição que um nodo $u$ representa em $V$ pode ser definida como $leftsize(u) + u.left.size + 1$.

**Fusão de árvores implícitas**

Sejam $T_1$ e $T_2$ duas *Treaps* implícitas, tal que $|T_1| = n$ e $|T_2| = m$, isto é, $T_1$ representa os índices $[1...n]$ de um um vetor $A$ e $T_2$ representa os índices $[1...m]$ de um vetor $B$. A fusão de $T_1$ com $T_2$ deve resultar em uma *Treap* implícita $T$, tal que $|T| = n + m$ e $T$ represente os índices um vetor $V$ que seja a concatenação de $B$ em $A$, ou seja $V = [A_1, A_2, ..., A_n, B_1, B_2, ..., B_m]$. Observe que a Fusão de árvores implícitas se mantém constante à Fusão de árvores, pois sua construção é feita a partir da prioridade dos nodos e não de suas chaves, e assim como a Fusão de árvores, recebemos duas árvores onde temos que uma árvore $T_2$ deve sempre se manter à direita de uma árvore $T_1$ para manter a propriedade de que os elementos do vetor $B$ estejam sempre à direita dos elementos do vetor $A$ em $V$. Como a Fusão de árvores implícitas permanece a mesma que a Fusão de árvores, temos que sua complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

**Quebra de árvores implícitas**

Seja $T$ uma *Treap* implícita e $x$ um inteiro, tal que $T$ represente os índices $[1...n]$ de um vetor $V$ de tamanho $n$, e $1 \leq x \leq n$. A quebra de Árvore implícita de $T$ por $x$ deve retornar duas *Treaps* implícitas $T_1$ e $T_2$ tal que $T_1$ represente um vetor $A = [V_1, V_2, ..., V_{x-1}]$ e $T_2$ represente um vetor $B = [V_x, V_{x+1}, ..., V_n]$. Assim como a Quebra de árvores, a construção de $T_1$ e $T_2$ será feita a partir da raiz $T$ recursivamente. Seja $u$ o nodo que esteja sendo avaliado atualmente (inicialmente $u = T$), temos as seguintes possibilidades:

1. $u$ é um nodo nulo: então sabemos que $(T_1, T_2) = (nulo, nulo)$;
2. $u$ representa um índice $i < x$ de $V$: temos que $T_1$ deve conter $u$, porém ainda há nodos na sub-árvore direita de $u$ que podem pertencer a $T_2$. Logo, precisamos das árvores $(T'_1, T'_2)$ referentes à quebra de árvore implícita de $u.right$;
3. $u$ representa um índice $i \geq x$ de $V$: temos que $T_2$ deve conter $u$, porém ainda há nodos na sub-árvore esquerda de $u$ que podem pertencer a $T_1$. Logo, precisamos das árvores $(T'_1, T'_2)$ referentes à quebra de árvore implícita de $u.left$.

O item 1 é trivialmente verdadeiro. Para o item 2, como $u$ representa um índice $i < x$, tem-se que todos os nodos na sub-árvore esquerda de $u$ também representam índices menores que $x$, mas na sub-árvore direita de $u$ pode existir algum nodo $w$ que represente

uma posição maior ou igual a $x$ e deve pertencer a $T_2$. Consequentemente, deve-se fazer a Quebra de árvores implícitas recursivamente na sub-árvore direita de $u$ para encontrar esses nodos. A prova para o item 3 é simétrica à prova do item 2. Faremos no máximo $k$ recursões em um pior caso, onde $k$ é a profundidade de $T$. Em média, a complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Inserção de elemento implícito

Seja $T$ uma *Treap* implícita, tal que $T$ represente o vetor $V$ de tamanho $n$, $x$ um inteiro qualquer e $i$ a posição em que o valor $x$ deve ser adicionado em $V$. A Inserção de elemento implícito é equivalente à Inserção de elemento, tal que inicialmente é feita a Quebra de árvores implícita de $T$ pela posição $i$ em $T_1$ e $T_2$, onde $T_1$ representa o vetor $A = [V_1, V_2, ..., V_{i-1}]$ e $T_2$ representa o vetor $B = [V_i, V_{i+1}, ..., V_n]$. Em seguida, é feita a Fusão de árvores implícitas de $T_1$ e $x$, resultando em uma *Treap* implícita $T_1'$ que representa o vetor $A' = [V_1, V_2, ..., V_{i-1}, x]$. Por fim, é feita a Fusão de árvores implícitas de $T_1'$ e $T_2$ resultando em uma *Treap* implícita $T'$ que representa o vetor $V' = [V_1, V_2, ..., V_{i-1}, x, V_i, V_{i+1}, ..., V_n]$. Como são feitas 3 operações de quebra/fusão, a complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Remoção de elemento implícito

Seja $T$ uma *Treap* implícita, tal que $T$ represente o vetor $V$ de tamanho $n$, e $i$ a posição do elemento que deve ser removido de $V$. A Remoção de elemento implícito é equivalente à Remoção de elemento, tal que inicialmente é feita a Quebra de árvores implícitas de $T$ pela posição $i$ em $T_1$ e $T_2$, onde $T_1$ representa o vetor $A = [V_1, V_2, ..., V_{i-1}]$ e $T_2$ representa o vetor $B = [V_i, V_{i+1}, ..., V_n]$. Em seguida, é feita a Quebra de árvores implícita em $T_2$ pela posição 2 em $T_1'$ e $T_2'$, onde $T_1'$ representa o vetor $A' = [V_i]$ e $T_2'$ representa o vetor $B' = [V_{i+1}, ..., V_n]$. Por fim, é feita a Fusão de árvores implícitas de $T_1$ e $T_2'$ resultando em uma *Treap* implícita $T'$ correspondente ao vetor $V' = [V_1, V_2, ..., V_{i-1}, V_{i+1}, ..., V_n]$. Como são feitas 3 operações de quebra/fusão, a complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

**Máximo em intervalo implícito**

Seja $(a, b)$ um par ordenado que represente um intervalo $[a...b]$ de um vetor $V$ e $T$ a *Treap* implícita correspondente de $V$, tal que $|V| = n$, $1 \leq a \leq b \leq n$. Para responder consultas de máximo em intervalo será necessário manter um valor adicional em cada nodo $u$ de $T$, que será o valor máximo de chave existente na sub-árvore de $u$, denotado por $u.maxi$. Com isso, basta encontrar a *Treap* implícita $T'$ que corresponde ao vetor $V' = [V_a, V_{a+1}, ..., V_b]$ e obter o valor de $T'.maxi$. Para isso, inicialmente é feita a Quebra de árvores implícitas de $T$ pela posição $a$ em $T_1$ e $T_2$, onde $T_1$ corresponde ao vetor $A = [V_1, V_2, ..., V_{a-1}]$ e $T_2$ corresponde ao vetor $B = [V_a, V_{a+1}, ..., V_b, V_{b+1}, ..., V_n]$. Em seguida, fazemos a Quebra de árvores implícitas de $T_2$ pela posição $b - a + 1$ (note que $|B| = n - a + 1$) em $T'_1$ e $T'_2$, onde $T'_1$ representa o vetor $A' = [V_a, V_{a+1}, ..., V_b]$ e $T'_2$ representa o vetor $B' = [V_{b+1}, V_{b+2}, ..., V_n]$. Por fim, temos que $A' = T'$ e que $max(V_a, V_{a+1}, ..., V_n) = A'.maxi$. Para manter o valor de $u.maxi$ atualizado para um nodo $u$ qualquer é necessário fazer a atualização de $u$ ao passar por ele em uma chamada de fusão/Quebra de árvores implícitas, onde a atualização de $u$ será $u.maxi = max(u.left.maxi, u.right.maxi, u.chave)$, pois $u.left.maxi$ cobre a sub-árvore esquerda de $u$, $u.right.maxi$ cobre a sub-árvore direita de $u$, e $u.chave$ cobre o próprio nodo $u$, consequentemente contemplando toda a sub-árvore de $u$. A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

## 6.3 Implementação

A construção de uma *Treap* $T$ será feita em cima de um vetor $V$ de tamanho $n$, onde um nodo será criado a partir de cada posição $i \in \{1, 2, ..., n\}$, tal que sua chave seja o valor de $V_i$ e sua prioridade um valor inteiro pseudo-aleatório. Inicialmente teremos uma estrutura *nodo* que representará um nodo qualquer de $T$ que será usada durante as implementações das funções a seguir.

```
1 struct nodo{
2     int chave, prioridade, size;
3     nodo *left, *right;
4     nodo(int _chave){
5         chave = _chave;
6         prioridade = rand();
7         left = right = size = 0;
8     }
```

```
};
```

Sua construção será feita por chamadas da função Inserir elemento para cada valor $V_i$ que pertence a $V$. Cada chamada tem complexidade de tempo esperada de $\mathcal{O}(\log n)$, totalizando uma complexidade de tempo esperada de construção de $\mathcal{O}(n \log n)$. A *Treap* $T$ é inicializa pela seguinte linha:

```
nodo* T = NULL;
```

### Fusão de árvores

Sejam $T_1$ e $T_2$ duas *Treaps*, onde para todo nodo $u$ pertencente a $T_1$ e para todo nodo $v$ pertencente a $T_2$ tem-se que $u.chave \leq v.chave$. A Fusão de árvores de $T_1$ com $T_2$ resultará em uma *Treap* implícita $T$ que possua todos os nodos de $T_1$ e $T_2$. A implementação é uma consequência direta das observações da Seção 6.2.1.

```
1  nodo* fundir(nodo* &esq, nodo* &dir){
2      if(esq == NULL) return dir;
3      if(dir == NULL) return esq;
4      if(esq->prioridade > dir->prioridade){
5          esq->right = fundir(esq->right, dir);
6          return esq;
7      }
8      else{
9          dir->left = fundir(esq, dir->left);
10         return dir;
11     }
12 }
```

As linhas 2, 3, 4 e 8 são consequências das condições 1, 2, 3 e 4 da Seção 6.2.1. A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Quebra de árvores

Seja $T$ uma *Treap* e $x$ um inteiro qualquer. A Quebra de árvores deve retornar duas *Treaps* $T_1$ e $T_2$ tal que todos os nodos de $T_1$ devem possuir uma chave menor que $x$ e todos os nodos de $T_2$ devem possuir uma chave maior ou igual a $x$.

```
1  pair<nodo*,nodo*> quebrar(nodo* &T, int x){
2      if(T == NULL) return {NULL, NULL};
3      if(T->chave < x){
4          nodo *t1, *t2; tie(t1, t2) = quebrar(T->right, x);
5          T->right = t1;
6          return {T, t2};
7      } else {
8          nodo *t1, *t2; tie(t1, t2) = quebrar(T->left, x);
9          T->left = t2;
10         return {t1, T};
11     }
12 }
```

A condição da linha 1 é equivalente à primeira possibilidade da Seção 6.2.2. Já as linhas 3 e 7 são consequências diretas das possibilidades 2 e 3 da Seção 6.2.2. A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Inserir elemento

Seja $T$ uma *Treap* e $x$ um inteiro qualquer. A função Inserir elemento deve inserir um nodo $(x, y)$ em *Treap* $T$, tal que $y$ seja um valor pseudo-aleatório. A inserção de um elemento se torna uma consequência de 3 chamadas de Quebra/Fusão de árvores. Inicialmente,quebra-se a *Treap* $T$ pelo valor $x$, resultando em duas *Treaps* $T_1$ e $T_2$ tal que todo nodo de $T_1$ seja menor que $x$ e todo nodo de $T_2$ seja maior ou igual a $x$. Em seguida, cria-se um nodo $u = (x, y)$ e é feita a fusão de $T_1$ com $u$ resultando em uma *Treap* $T_{11}$. Por fim, é feita a fusão de $T_{11}$ com $T_2$, concluido a inserção do elemento.

```
1  void inserir(nodo* &T, int x){
2      nodo* u = new nodo(x);
3      nodo *t1, *t2; tie(t1,t2) = quebrar(T, x);
4      nodo *t11 = fundir(t1, u);
5      T = fundir(t11, t2);
```

```
}
```

A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

**Remover elemento**

Seja $T$ uma *Treap* e $x$ um inteiro qualquer. A função Remover elemento deve remover os nodos que possuem o valor de chave $x$ de $T$. Assim como a função Inserir elemento, a remoção se dá pela sequência de 3 chamadas de Quebra/Fusão de árvores. Inicialmente, quebra-se a *Treap* $T$ pelo valor de $x$, resultando em duas *Treaps* $T_1$ e $T_2$, tal que todo nodo de $T_1$ seja menor que $x$ e todo nodo de $T_2$ seja maior ou igual a $x$. Em seguida, é feita outra Quebra de árvores em $T_2$ pelo valor de $x+1$ gerando duas árvores $T_{21}$ e $T_{22}$, tal que $T_{21}$ será composta por todos os nodos com chave igual a $x$ e todo nodo de $T_{22}$ terá valor de chave estritamente maior que $x$. Com isso, basta fazer a Fusão de árvores de $T_1$ e $T_{22}$.

```
void remover(nodo* &T, int x){
    nodo *t1, *t2; tie(t1,t2) = quebrar(T, x);
    nodo *t21, *t22 = quebrar(t2, x+1);
    T = fundir(t1, t22);
}
```

A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### 6.3.1 *Treap* implícita

A construção de uma *Treap* implícita $T$ que representa um vetor $V$ de inteiros, onde $|V| = n$, pode ser feita pela chamada de Inserção de elemento implícito para cada valor $V_i$ em $T$. Como cada inserção terá tempo esperado de $\mathcal{O}(\log n)$ a complexidade esperada total da construção será de $\mathcal{O}(n \log n)$. Como o tamanho de sub-árvore de cada nodo deve ser mantido, tem-se que um nodo $u$ qualquer deve ser atualizado durante as funções de Quebra e Fusão de árvores, e isso será mantido pela seguinte função:

```
void update(nodo* &T){
    if(T != NULL){
        T->size = 1;
        if(T->left != NULL) T->size += T->left->size;
        if(T->right != NULL) T->size += T->right->size;
    }
}
```

O valor da sub-árvore de T deve ser 1 somado com o valor de sua sub-árvore esquerda (se existir) e o valor de sua sub-árvore direita (se existir).

### Fusão de árvores implícitas

Sejam $T_1$ e $T_2$ duas *Treaps* implícitas, tal que $T_1$ represente o vetor $A$ de tamanho $n$ e $T_2$ represente o vetor $B$ de tamanho $m$, o resultado da Fusão de árvores implícitas é uma *Treap* implícita $T$ que representa um vetor $V = [A_1, A_2, ..., A_n, B_1, B_2, ..., B_m]$. A Fusão de árvores implícitas permanece a mesma da Fusão de árvores, pois o critério de seleção da fusão se dá somente pela prioridade (um valor que não é alterado de *Treap* para *Treap* implícita). A implementação seguinte segue das observações da Seção Fusão de árvores implícita e sua chamada é feita por $T = fundir(T_1, T_2)$.

```
nodo* fundir(nodo* &esq, nodo* &dir){
    if(esq == NULL) return dir;
    if(dir == NULL) return esq;
    if(esq->prioridade > dir->prioridade){
        esq->right = fundir(esq->right, dir);
        update(esq);
        return esq;
    }
    else{
        dir->left = fundir(esq, dir->left);
        update(dir);
        return dir;
    }
}
```

Note que, ao fazer uma modificação em uma das sub-árvores de *esq* ou *dir*, é necessário

fazer a chamada da função *update* para manter o valor de *size* atualizado. A complexidade de tempo esperada para essa função é de $\mathcal{O}(\log n)$.

### Quebra de árvores implícitas

Seja $T$ uma *Treap* implícita e $x$ um inteiro, tal que $T$ represente um vetor $V$ de tamanho $n$ e $1 \leq x \leq n$. A Quebra de árvores implícitas de $T$ por $x$ retorna duas *Treaps* implícitas $T_1$ e $T_2$, tal que $T_1$ representa o vetor $A = [V_1, V_2, ..., V_x]$ e $T_2$ representa o vetor $B = [V_{x+1}, V_{x+2}, ..., V_n]$. Diferentemente da Quebra de árvores, onde a condição de quebra é o valor da chave do nodo avaliado, na Quebra de árvores implícitas temos que essa condição deve ser a posição que o nodo avaliado representa em $V$. O valor de $leftsize$ é calculado recursivamente ao descer pela raiz $T$.

```
1  pair<nodo*,nodo*> quebrar(nodo* &T, int x, int leftsize = 0){
2      if(T == NULL) return {NULL, NULL};
3      int total_left = T->left != NULL ? T->left->len : 0;
4      int position = leftsize + total_left + 1;
5      if(position < x){
6          nodo *t1, *t2; tie(t1, t2) = quebrar(T->right, x, position);
7          T->right = t1;
8          update(T); update(t2);
9          return {T, t2};
10     } else {
11         nodo *t1, *t2; tie(t1, t2) = quebrar(T->left, x, leftsize);
12         T->left = t2;
13         update(t1); update(T);
14         return {t1, T};
15     }
16 }
```

Ao fazer uma Quebra de árvores implícitas deve-se fazer a chamada da função *update* para manter o valor de *size* atualizado após a modificação na árvore. Assim como na Quebra de árvores, tem-se que sua complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Inserção de elemento implícito

Seja $T$ uma *Treap* implícita, $x$ e $i$ inteiros quaisquer, tal que $T$ represente um vetor $V$ de tamanho $n$ e $1 \leq i \leq n$. A Inserção de elemento implícito de $(i, x)$ em $T$ deve modificar

$T$ fazendo com que este represente um novo vetor $V' = [V_1, V_2, ..., V_{i-1}, x, V_i, ..., V_n]$. A Inserção de elemento implícito pode ser quebrada em uma sequência de 3 operações de quebra/Fusão de árvores implícitas. Inicialmente, é feita uma Quebra de árvores implícitas para encontrar a posição de interesse $i$. Em seguida são feitas duas fusões para inserir o novo valor $x$ na posição de interesse em $T$.

```
void inserir(nodo* &T, nodo* novo, int i){
    nodo *t1, *t2; tie(t1,t2) = quebrar(T, i);
    nodo *t11 = fundir(t1, novo);
    T = fundir(t11, t2);
}
```

Repare que a função Inserção de elemento implícito modifica a *Treap* implícita $T$ mas não precisa fazer uma chamada para a função *update*, visto que essas atualizações são feitas dentro das funções de quebra/Fusão de árvores implícitas. A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Remoção de elemento implícito

Seja $T$ uma *Treap* implícita e $i$ um inteiro, tal que $T$ represente um vetor $V$ de tamanho $n$ e $1 \leq i \leq n$. A Remoção de elemento implícito de $i$ em $T$ deve modificar $T$ fazendo com que este represente um novo vetor $V' = [V_1, V_2, ..., V_{i-1}, V_{i+1}, ..., V_n]$. Assim como a Inserção de elemento implícito, a Remoção de elemento implícito também pode ser quebrada em uma sequência de 3 operações de quebra/Fusão de árvores implícitas.

```
void remover(nodo* &T, int i){
    nodo *t1, *t2; tie(t1,t2) = quebrar(T, i);
    nodo *t21, *t22 = quebrar(t2, 2);
    T = fundir(t1, t22);
}
```

A complexidade de tempo esperada é de $\mathcal{O}(\log n)$.

### Máximo em intervalo implícito

Seja $T$ uma *Treap* implícita e $(a, b)$ um par de inteiros ordenados que representam um intervalo, tal que $T$ represente um vetor $V$ de tamanho $n$ e $1 \leq a \leq b \leq n$. O Máximo em intervalo implícito deve retornar um valor $x$ que deve ser o resultado de $max(V_a, V_{a+1}, ..., V_b)$. Deverá ser mantido um valor $maxi$ para cada nodo $u$ de $T$, tal que $u.maxi$ será o valor máximo de toda a sub-árvore de $u$. Inicialmente, fazemos duas quebras para encontrar uma *Treap* implícita $T'$ tal que $T'$ represente o vetor $V' = [V_a, V_{a+1}, ..., V_b]$, com isso $x = T'.maxi$.

```
void maximo_intervalo(nodo* &T, int a, int b){
    nodo *t1, *t2; tie(t1,t2) = quebrar(T, a);
    nodo *t21, *t22 = quebrar(t2, b-a+2);
    int x = t21->maxi;
    t2 = fundir(t21, t22);
    T = fundir(fundir(t1, t2));
}
```

Deve-se manter o valor de $maxi$ atualizado para cada nodo após uma modificação em sua sub-árvore. Para isso, basta fazer uma alteração na função *update* para manter o valor de $maxi$ atualizado.

```
void update(nodo* &T){
    if(T != NULL){
        T->size = 1;
        T->maxi = T->chave;
        if(T->left != NULL){
            T->size += T->left->size;
            T->maxi = max(T->maxi, T->left->maxi);
        }
        if(T->right != NULL){
            T->size += T->right->size;
            T->maxi = max(T->maxi, T->right->maxi);
        }
    }
}
```

A complexidade de tempo esperada da função Máximo em intervalo implícito é de $\mathcal{O}(\log n)$.

## 6.4 Problema Relacionado

No problema *Dog Show* da fase qualificatória da sub-regional da Eurásia da região sul (DOG..., 2019), é necessário o conceito de Treap (note que o problema pode ser feito com outras técnicas com algumas observações). Uma descrição simplificada do problema é a seguinte:

> Seja $A$ um vetor de valores inteiros, tal que $|A| = n$. Seu objetivo é marcar o maior número de posições de $A$ em $K$ minutos. A $i$-ésima posição pode ser marcada somente após $A_i$ minutos. Os valores devem ser marcados da esquerda para a direita (partindo da posição 1 até $n$) e o tempo para ir de um índice $i$ até um índice $i + 1$ é de 1 minuto. Ao avaliar um índice $i$ com tempo atual $t$, tem-se três opções:
>
> 1. Marcar a posição $i$ se $A_i \leq t$;
> 2. Não marcar a posição $i$ se $A_i > t$;
> 3. Esperar até o tempo $A_i$ e então marca-lo se $A_i > t$.
>
> Inicialmente, nenhum índice é avaliado e leva-se 1 minuto para começar a avaliar o índice 1. Note que não é necessário marcar todos os valores. As restrições de valores são: $1 \leq n \leq 200000$ e $1 \leq A_i \leq 10^9$.

A primeira observação que pode ser feita é que esperar $x$ minutos para marcar uma posição $i$ em uma determinada sequência é equivalente a esperar $x$ antes de começar as avaliações e repetir a mesma sequência sem a necessidade de esperar na posição $i$. Consequentemente, a sequência ótima deve esperar $m$ minutos antes de começar as avaliações e em seguida marcar todos os valores possíveis até chegar em $n$ ou o seu tempo exceder $K$. A princípio, deve-se modificar os valores de $A_i$ para $A_i - i$ para todo $i \in \{1, 2, ..., n\}$, assim o valor de $A_i$ representa o tempo máximo de espera que pode ser feito antes de começar as avaliações e que ainda seja possível marcar o índice $i$, e a restrição de ordem (esquerda para direita) dos valores pode ser descartada.

Com isso, tem-se duas possibilidades: (1) $K$ é menor que $n$ e (2) $K$ é maior ou igual a $n$. No caso (1) temos que os valores possíveis de $m$ estão no intervalo $[0...K - 1]$.

Pode-se fazer uma busca completa em todos os valores possíveis, onde a quantia ótima de um valor $x$ é equivalente ao número de posições $j$ tal que $x \geq A_j$. Os valores $A_i$, tal que $t \leq i \leq n$, devem ser removidos antes das buscas por serem inatingíveis (não há tempo suficiente para chegar nesses valores). No caso (2), assim como no caso (1) os valores possíveis de $m$ estão no intervalo $[0...K-1]$. Porém, repare que neste caso sempre é ótimo esperar o máximo de tempo possível antes de ser necessário remover valores inatingíveis. Com isso, o novo intervalo de valores possíveis é $[K-n-1...K-1]$ , que possui tamanho $n$. Novamente, pode-se fazer uma busca completa nos valores possíveis, porém ao avaliar cada um dos valores possíveis deve-se remover as posições inatingíveis. Por fim, construímos uma *Treap* $T$ com todos os valores de $A$ modificados. Ao avaliar um tempo de espera $x$, é suficiente fazer uma Quebra de árvores em $T$ pelo valor de $x+1$ resultante em duas *Treaps* $T_1$ e $T_2$, tal que todos os nodos de $T_1$ possuem valor de chave menores ou igual à $x$, e todos os nodos de $T_2$ possuem valor de chave maior que $x$. O valor ótimo de $x$ deve ser o número de nodos em $T_1$, que pode ser obtido mantendo o tamanho de sub-árvore para cada nodo. Para remover uma posição inatingível $i$ é suficiente fazer a chamada de Remover elemento em um nodo que possua chave de valor $A_i$.

# 7 Considerações Finais

Durante a construção deste trabalho foi utilizada uma abordagem informal com o intuito de desenvolver as estruturas de dados mais recorrentes para operações em intervalos de vetor de uma maneira didática e com certo rigor matemático, valendo ressaltar que o foco desse trabalho é para leitores com interesse em programação competitiva. O desenvolvimento desse trabalho se deu principalmente pela escassez de materiais sobre estruturas de dados avançadas na academia, tanto nacionalmente como internacionalmente, visto que não são temas tão recorrentes no cenário comercial.

Foram fundamentadas as principais operações de cada uma das estruturas abordadas em conjunto com a análise de complexidade de tempo e da corretude de suas operações. Além disso, outras aplicações menos óbvias foram abordadas como a aplicação de *Sparse Table* para encontrar o Ancestral Mínimo Comum em árvores, *Segment Tree* com *Lazy Propagation* para modificação em área e a utilização de *Treap* implícita para simular um vetor de elementos com inserção/remoção de elementos e operações em intervalo em tempo $\mathcal{O}(\log n)$. Adicionalmente, para cada uma das operações abordadas no desenvolvimento das estruturas foram implementados códigos em C++/C++11. Os códigos fornecidos foram testados para problemas onde as estruturas foram necessárias e estão disponíveis no endereço `https://github.com/BRUTEUdesc/datastructures_array_intervals`. Cada uma das estruturas desenvolvidas possui pontos fortes e fracos. De modo geral, uma consequência do aumento de robustez de uma estrutura de dados é um código mais verboso e mais sujeito a erros de implementação. Isso se torna um fator importante principalmente em ambientes de competição, onde tempo é um fator crucial. Consequentemente, ter o conhecimento para discernir qual estrutura será suficiente para a resolução de um problema é uma habilidade que deve ser dominada pelos competidores.

Por fim, a *Segment Tree* $2D$, *Fenwick Tree* $2D$ e *Treap* com *Lazy Propagation* seriam estruturas que poderiam ser desenvolvidas em trabalhos futuros, visto que são extremamente robustas e generalizáveis para diversas operações e não possuem a devida abordagem formalizada dentro da academia.

# Referências


ANIDO, R. *Cortador de Pizza*. 2018. Acessado em 06 de Junho de 2019. Disponível em: <http://maratona.ime.usp.br/hist/2018/primfase18.html>.

ARAGON, C. R.; SEIDEL, R. G. Randomized search trees. In: IEEE. *30th Annual Symposium on Foundations of Computer Science*. 1989. p. 540–545.

ARCHIVE, I. L. *Yearly Stats*. 2019. `https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=23`. Acessado em 31 de Maio de 2019.

BENDER, M. A.; FARACH-COLTON, M. The lca problem revisited. In: SPRINGER. *Latin American Symposium on Theoretical Informatics*. 2000. p. 88–94.

BENTLEY, J. L. Solutions to klee's rectangle problems. *Unpublished manuscript*, p. 282–300, 1977.

CORMEN, T. H. et al. *Introduction to algorithms*. : MIT press, 2009.

COSTA, P. C. P. *Fundraising*. 2017. Acessado em 04 de Junho de 2019. Disponível em: <http://maratona.ime.usp.br/hist/2017/index.html>.

COUTO, Y. S. *Algoritmos em sequências*. 2016. Trabalho de Conclusão de Curso (Graduação), USP (Universidade de São Paulo), São Paulo, Brasil.

DOG Show. 2019. Acessado em 12 de Novembro de 2019. Disponível em: <https://codeforces.com/contest/847/problem/D>.

DUARTE, G. L. *Técnicas De Otimização De Programação Dinâmica*. 2017. Trabalho de Conclusão de Curso (Graduação), UNIFESO (Centro Universitário Serra dos Órgãos), Teresópolis, Brasil.

FENWICK, P. M. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, Wiley Online Library, v. 24, n. 3, p. 327–336, 1994.

FORISEK, M. *The International Olympiad in Informatics Syllabus*. 2018. Acessado em 28 de Maio de 2019. Disponível em: <https://ioinformatics.org/files/ioi-syllabus-2018.pdf>.

GOOGLE. *Manhattan Crepe Cart*. 2019. Acessado em 22 de Maio de 2019. Disponível em: <https://codingcompetitions.withgoogle.com/codejam/round/0000000000051706/000000000012295c>.

HALIM, S. et al. *Competitive Programming 3*. : Lulu Independent Publish, 2013.

KAWAKAMI, M. M. *Algoritmos em redes de fluxo e aplicações*. 2017. Trabalho de Conclusão de Curso (Graduação), USP (Universidade de São Paulo), São Paulo, Brasil.

MENEZES, P. B. *Matemática discreta para computação e informática*. : Bookman, 2010. 47-48 p.

NIÑO, E. *Imperial Roads*. 2017. Acessado em 22 de Maio de 2019. Disponível em: <http://maratona.ime.usp.br/hist/2017/index.html>.

QIU, K.; AKL, S. G. Parallel maximum sum algorithms on interconnection networks. In: *Proceedings of the Eleventh IAESTED Conference on Parallel and Distributed Computing and Systems*. 1999. p. 31–38.

TOMMASINI, S. *Programação Dinâmica*. 2014. Trabalho de Conclusão de Curso (Graduação), USP (Universidade de São Paulo), São Paulo, Brasil.