**Pontifícia Universidade Católica do Rio Grande do Sul**

**Projeto e Otimização de Algoritmos**

**Engenharia de Software**

**Carolina Ferreira, Luiza Heller e Mateus Caçabuena**

**Pattern Matching**

**Porto Alegre**

**2024**

Sumário

[**1. Introdução 3**](#_Toc184166150)

[**2. Fundamentação Teórica 4**](#_Toc184166151)

[**2.1. Algoritmo Rabin-Karp 4**](#_Toc184166152)

[**2.2. Função de Hash no Rabin-Karp 4**](#_Toc184166153)

[**2.3. Rolling Hash 5**](#_Toc184166154)

[**3. Metodologia 6**](#_Toc184166155)

[**3.1. Escolha da Função Rolling Hash 6**](#_Toc184166156)

[**3.2. Implementação dos Algoritmos 6**](#_Toc184166157)

[**3.2.1 Código Base (RabinKarpAula) 6**](#_Toc184166158)

[**3.2.2 Código Otimizado (RabinKarpTrabalho) 7**](#_Toc184166159)

[**3.3. Casos de Teste 8**](#_Toc184166160)

[**4. Resultados 9**](#_Toc184166161)

[**4.1. Implementação da Aula (RabinKarpAula) 9**](#_Toc184166162)

[**4.2. Implementação com Rolling Hash (RabinKarpTrabalho) 9**](#_Toc184166163)

[**4.3. Comparação de Desempenho 10**](#_Toc184166164)

[**5. Discussão 11**](#_Toc184166165)

[**6. Conclusão 12**](#_Toc184166166)

[**7. Referências 13**](#_Toc184166167)

[**8. Apêndices 14**](#_Toc184166168)

[**8.1. Código Fonte 14**](#_Toc184166169)

[**8.1.1 RabinKarpAula 14**](#_Toc184166170)

[**8.1.2. RabinKarpTrabalho 15**](#_Toc184166171)

[**8.2. Resultados Detalhados 16**](#_Toc184166172)

[**8.2.1. RabinKarpAula 16**](#_Toc184166173)

[**8.2.2. RabinKarpTrabalho 17**](#_Toc184166174)

# Introdução

O presente trabalho aborda o problema de *pattern matching* ou busca de padrões, que consiste em localizar todas as ocorrências de um padrão específico dentro de um texto. Este problema é amplamente encontrado em diversas áreas, como bioinformática, busca textual e análise de dados, sendo, no entanto, desafiador devido à alta demanda computacional, especialmente ao lidar com textos extensos e padrões complexos.

Um dos algoritmos clássicos para resolver esse problema é o Rabin-Karp, que utiliza uma função de hash para realizar comparações eficientes entre substrings do texto e o padrão desejado. Diferentemente de métodos mais simples que realizam verificações de caractere a caractere, o Rabin-Karp pode alcançar um desempenho médio significativamente melhor, particularmente em cenários onde múltiplos padrões precisam ser localizados em um mesmo texto.

Um aspecto fundamental do desempenho do algoritmo Rabin-Karp é a técnica de *rolling hash*. Essa técnica permite o reaproveitamento do valor do hash de uma janela anterior para calcular o hash da próxima janela, reduzindo o número de operações necessárias. Essa característica não só melhora a eficiência, mas também diferencia o Rabin-Karp de outros algoritmos de busca de padrões que recalculam hashes independentemente para cada posição.

O objetivo deste trabalho é aprofundar o entendimento sobre a função de hash utilizada no algoritmo Rabin-Karp, implementar uma versão prática do algoritmo com uma função *rolling hash* escolhida, e avaliar o desempenho dessa implementação em comparação com a versão apresentada em aula. Além disso, busca-se explorar o impacto dessa técnica em diferentes casos de teste, fornecendo uma análise experimental robusta.

# Fundamentação Teórica

## 2.1. Algoritmo Rabin-Karp

O algoritmo Rabin-Karp é uma solução clássica para o problema de busca de padrões (*pattern matching*), que busca localizar todas as ocorrências de um padrão PPP em um texto TTT. Ele se baseia na comparação de hashes das substrings do texto com o hash do padrão, evitando comparações caractere a caractere em todas as posições do texto.

A principal ideia por trás do Rabin-Karp é que, se dois valores de hash são iguais, as strings correspondentes provavelmente também são iguais. Essa característica torna o algoritmo eficiente para localizar padrões em grandes volumes de texto ou em situações onde múltiplos padrões precisam ser buscados simultaneamente.

O algoritmo segue os seguintes passos básicos:

1. Calcula o valor de hash do padrão PPP.
2. Calcula os valores de hash de todas as janelas de tamanho ∣P∣|P|∣P∣ no texto TTT.
3. Compara o hash do padrão com os hashes das janelas do texto. Caso haja uma correspondência, verifica caractere a caractere (para evitar colisões de hash).

A complexidade computacional do Rabin-Karp, em média, é ***O(n+m)***, onde ***n*** é o comprimento do texto e ***m*** é o comprimento do padrão. Contudo, no pior caso, a complexidade pode ser ***O(m \* n)***, dependendo da função de hash utilizada.

## 2.2. Função de Hash no Rabin-Karp

A função de hash desempenha um papel central no desempenho do algoritmo Rabin-Karp. Em termos gerais, uma função de hash transforma uma string em um valor numérico único, reduzindo a comparação de strings à comparação de números.

Uma função de hash ingênua recalcula o hash de cada substring do texto de forma independente, resultando em uma complexidade elevada. A técnica de *rolling hash* resolve esse problema de maneira eficiente.

## 2.3. Rolling Hash

O conceito de *rolling hash* é uma inovação fundamental no algoritmo Rabin-Karp, permitindo o reaproveitamento do cálculo de hash de uma janela anterior ao mover para a próxima. Essa técnica reduz significativamente o número de operações necessárias.

Considere uma janela de tamanho ***m*** em um texto ***T***. O valor do hash para a janela atual pode ser calculado a partir do hash da janela anterior, subtraindo a contribuição do primeiro caractere da janela e adicionando a do próximo caractere. Isso elimina a necessidade de recalcular o hash do zero para cada nova posição.

A fórmula geral para o cálculo incremental do hash é:



Onde:

* ***base*** é uma constante (normalmente 256, para representar o número de caracteres ASCII).
* ***prime*** é um número primo grande usado para minimizar colisões.
* ***T[i]*** e ***T[i+m]*** são, respectivamente, o primeiro caractere da janela anterior e o próximo caractere da nova janela.

A eficiência dessa técnica é o que torna o Rabin-Karp competitivo em comparação com outros algoritmos, especialmente em textos extensos.

# Metodologia

## 3.1. Escolha da Função Rolling Hash

Para atender aos requisitos do trabalho, optou-se por utilizar a técnica de *rolling hash* na implementação do algoritmo Rabin-Karp, pois ela permite o reaproveitamento de cálculos de hash ao deslizar o padrão sobre o texto. Essa abordagem reduz a necessidade de recomputar hashes completamente para cada posição do texto, contribuindo para maior eficiência em comparação com a abordagem que calcula o hash de cada substring de forma independente.

Na função escolhida, os valores de hash são calculados iterativamente usando a fórmula:



* ***d***: Base numérica usada para calcular o hash, geralmente relacionada ao conjunto de caracteres (neste caso, 10).
* ***q***: Número primo grande utilizado para evitar colisões.
* ***h***: Valor pré-calculado equivalente a ***dM−1 mod  q***, que ajuda a remover o impacto do caractere mais à esquerda da janela.

## 3.2. Implementação dos Algoritmos

### 3.2.1 Código Base (RabinKarpAula)

O código implementado em aula realiza o cálculo do hash de cada substring do texto de forma independente. Sua estrutura básica é:

1. Calcula o hash do padrão.
2. Para cada janela no texto, calcula o hash da substring correspondente.
3. Compara os hashes do padrão e da substring. Caso coincidam, realiza uma verificação caractere a caractere para confirmar.

**Principais Limitações**:

* Não utiliza *rolling hash*, resultando em recalculação completa para cada nova janela.
* Pode ser ineficiente em textos longos ou padrões extensos.

Exemplo de teste do código:

Texto

Descrição gerada automaticamente

### 3.2.2 Código Otimizado (RabinKarpTrabalho)

Nesta versão, foi introduzida a técnica de *rolling hash*, permitindo calcular os hashes incrementais ao longo do texto. Sua estrutura básica é:

1. Pré-calcula o hash do padrão e da primeira janela do texto.
2. Desliza o padrão sobre o texto, reaproveitando o valor de hash da janela anterior para calcular o próximo.
3. Caso os hashes coincidam, verifica caractere a caractere.

**Benefícios**:

* Reduz o custo de recalcular o hash para cada nova posição.
* É mais eficiente em textos grandes e padrões longos.

Exemplo de teste do código:

Texto

Descrição gerada automaticamente

## 3.3. Casos de Teste

Para garantir uma análise robusta, com diversos cenários, a seguir estão alguns exemplos:

1. **Padrões curtos em textos curtos** (e.g., "ana" em "banana").
2. **Padrões que não ocorrem no texto** (e.g., "bye" em "hello world").
3. **Padrões iguais ao texto inteiro** (e.g., "pattern" em "pattern").
4. **Casos com caracteres especiais** (e.g., "$c d#" em "a\_b$c d#e%f^").
5. **Textos menores que o padrão** (e.g., "muchlongerpattern" em "short").
6. **Padrões sensíveis a maiúsculas/minúsculas** (e.g., "sensitive" em "CaseSensitiveTest").
7. **Padrões repetidos em sequência** (e.g., "aa" em "aaaaaa").

# 4. Resultados

## 4.1. Implementação da Aula (RabinKarpAula)

A versão desenvolvida em aula apresentou os seguintes resultados nos testes:

* **Testes bem-sucedidos**: O algoritmo encontrou corretamente o padrão em casos simples (e.g., "ana" em "banana") e em casos mais complexos (e.g., caracteres especiais).
* **Limitações observadas**:
  + Não detecta múltiplas ocorrências do padrão no mesmo texto (e.g., no teste "ana" em "banana", encontra apenas a primeira).
  + Retorna um índice incorreto (igual ao tamanho do texto) quando o padrão não é encontrado, o que pode gerar confusão.

## 4.2. Implementação com Rolling Hash (RabinKarpTrabalho)

A versão otimizada com *rolling hash* apresentou os seguintes resultados:

* **Testes bem-sucedidos**:
  + Detectou todas as ocorrências do padrão em textos simples e complexos (e.g., múltiplas ocorrências de "ana" em "banana").
* **Problemas encontrados**:
  + Falha em lidar com padrões maiores que o texto (Teste 5) devido à exceção StringIndexOutOfBoundsException.
  + Não apresenta resultados quando o padrão não é encontrado, sugerindo que faltou implementar uma mensagem clara para esses casos.

## 4.3. Comparação de Desempenho

Embora ambos os algoritmos tenham apresentado resultados semelhantes em termos de corretude nos testes básicos, a implementação com *rolling hash* mostrou-se mais eficiente em detectar múltiplas ocorrências do padrão. Entretanto, sua robustez foi comprometida pela falta de validação para casos extremos, como padrões maiores que o texto.

# 5. Discussão

* **Impacto da Técnica Rolling Hash**: A técnica de *rolling hash* trouxe ganhos de eficiência, permitindo detectar múltiplas ocorrências de padrões com maior precisão. No entanto, a implementação apresentou limitações de robustez.
* **Análise dos Resultados**:
  + O algoritmo da aula, embora funcional para casos básicos, mostrou limitações na capacidade de identificar múltiplas ocorrências e em casos de padrões ausentes.
  + A implementação com *rolling hash* foi mais eficiente, mas requer melhorias para lidar com casos extremos.
* **Limitações Gerais**:
  + Ambos os algoritmos dependem fortemente da escolha de parâmetros como o valor da base e o número primo ***q***, que podem impactar na probabilidade de colisões de hash.

# 6. Conclusão

O trabalho demonstrou o impacto da técnica de *rolling hash* na eficiência do algoritmo Rabin-Karp. Enquanto a implementação com *rolling hash* foi superior em identificar múltiplas ocorrências, ela apresentou problemas que limitam sua robustez. O algoritmo desenvolvido em aula, embora menos eficiente, foi mais robusto em termos de manuseio de casos especiais.

Esse estudo reforça a importância de balancear eficiência e robustez na escolha de algoritmos, especialmente ao lidar com diferentes tipos de entradas. Para estudos futuros, melhorias na implementação do *rolling hash* poderiam ampliar sua aplicabilidade prática.

# 7. Referências

* <https://www.programiz.com/dsa/rabin-karp-algorithm>
* <https://www.wikiwand.com/en/articles/Rabin%E2%80%93Karp_algorithm>
* <https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>

# 8. Apêndices

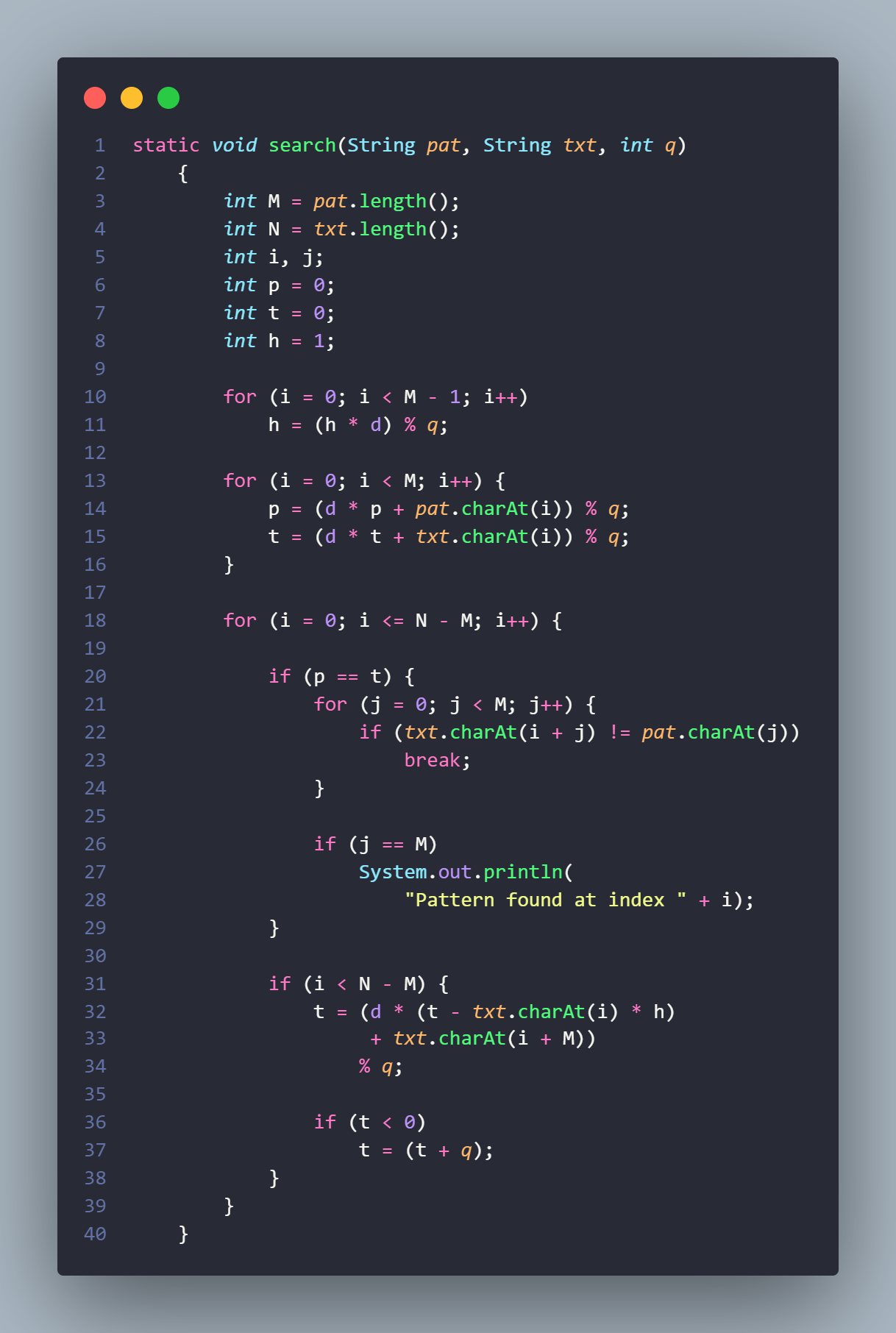
## 8.1. Código Fonte

### 8.1.1 RabinKarpAula

Texto

Descrição gerada automaticamente

### 8.1.2. RabinKarpTrabalho



## 8.2. Resultados Detalhados

### 8.2.1. RabinKarpAula

Texto

Descrição gerada automaticamente

### 8.2.2. RabinKarpTrabalho

Texto

Descrição gerada automaticamente