Regular Expression Matching

GRUPO Q

Alysson Araújo - 474084
Davi José - 469934
Henricky Lima - 475075

18 Novembro 2022

- **Regular Expression Matching**
- **Problema**
- **Algoritmo Recursivo**
- **Algoritmo Programação Dinâmica**
- **Corretude e Comparação**
- **Exemplos de Execução**

Problema

Problema

Implementar verificação se uma determinada string é formada por uma expressão regular com suporte a '.' e '*'.

Definição: Dadas as strings **text** e **pattern** com tamanhos T e P, tal que t_i e p_j são elementos dada as posições, onde

- $0 \leq i < T$
- $0 \leq j < P$
- **text** é formada por caracteres `{a..z}`
- **pattern** é formada por caracteres (`a..z, *, .`).
    - `.` define um caractere qualquer em (`a..z`)
    - * define 0 ou n repetições do caractere antecessor, este pertencendo a `.`

Temos que `isMatch(text,pattern)` que retorna `Match(i,j)`, para $i = 0$ e $j = 0$, e este retorna se **text** é formada por **pattern** através do algoritmo.

Problema

Exemplos:

Input: s = "aa", p = "a"
Output: false
Explanation: "a" não forma a string "aa".

Input: s = "aaa", p = "a*"
Output: true
Explanation :'*' significa zero ou mais do elemento precedente, 'a'. Portanto, repetindo 'a' três vezes, torna-se "aaa".

Input: s = "abc", p = ".*"
Output: true
Explanation: ".*" significa "zero ou mais (*) de qualquer caractere (.)".

Problema

Restrições:

- 1 <= **s**.length <= 20
- 1 <= **p**.length <= 30
- **s** contém somente letras minúsculas.
- **p** contém somente letras, '.', e '*'.
- É garantido que a cada aparição do caractere '*', haverá um caractere anterior válido para corresponder.

Recursivo

Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.   def isMatch(text: str, pattern: str):
2.     def match(i: int, j: int)...
3.     return match(0, 0)
```

Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.   def isMatch(text: str, pattern: str):
2.     //definição da função match
3.     return match(0, 0)
```

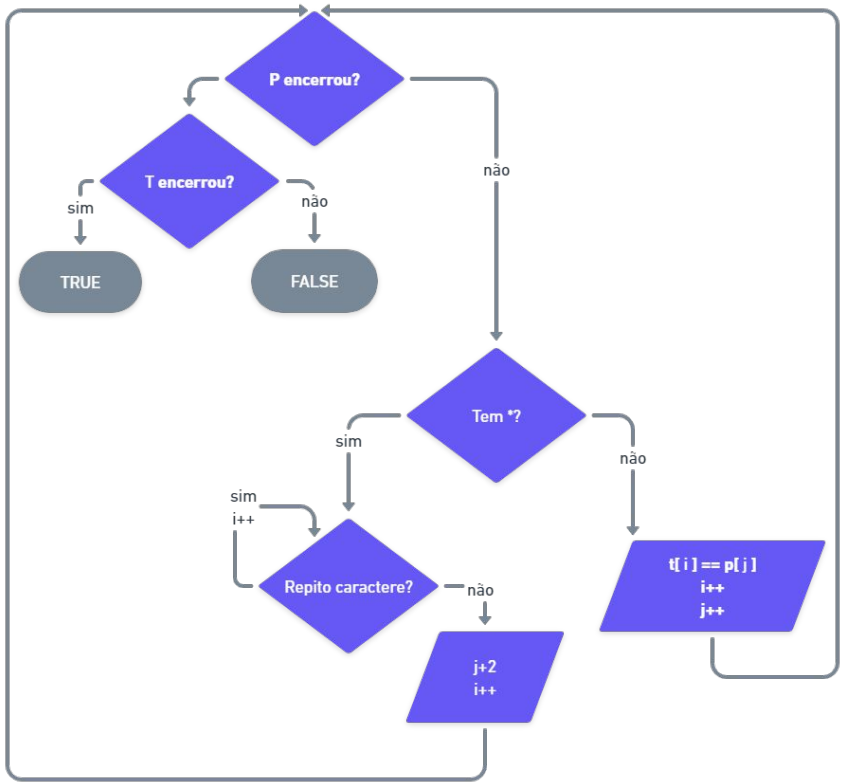
Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.   def isMatch(text: str, pattern: str):
2.     //definição da função match
3.     return match(0, 0)
```



Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    //definição da função match
3.    return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na rodada atual

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)):
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or (first_match and match(i+1, j))
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo

Algoritmo: isMatch(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    //definição da função match
3.    return match(0, 0)
```

Algoritmo: match(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na rodada atual

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)): //Verificar se as duas Str terminam juntas
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or (first_match and match(i+1, j))
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    //definição da função match
3.    return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na **rodada atual**[1]

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)): //Verificar se as duas Str terminam juntas
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}//Verificar primeiro match[1]
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or (first_match and match(i+1, j))
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    //definição da função match
3.    return match(0, 0)
```

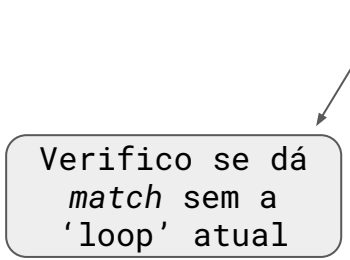
Algoritmo: **match**(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

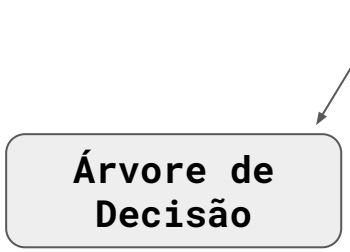
Saída: booleano - Verificação match na **rodada atual**[1]

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)): //Verificar se as duas Str terminam juntas
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}//Verificar primeiro match[1]
6.        if (j+1 < len(pattern) and pattern[j+1] == '*')://Tem repetição
7.          return match(i, j+2) or (first_match and match(i+1, j))
8.        else:
9.          return first_match and match(i+1, j+1)
```

```
match(i, j+2) or (first_match and match(i+1, j))
```



```
match(i, j+2) or (first_match and match(i+1, j))
```

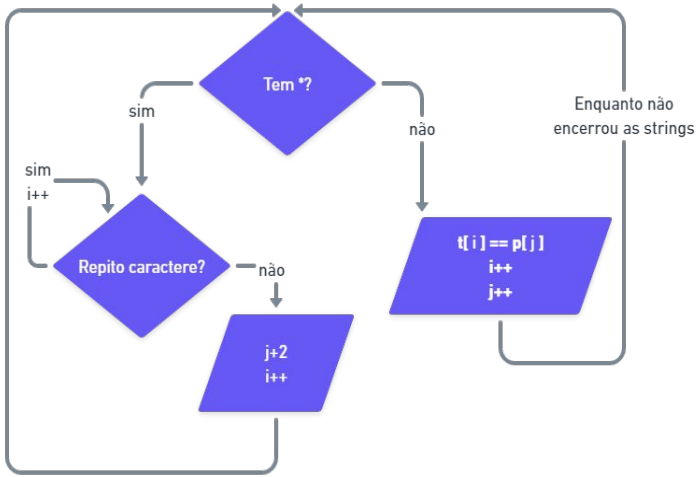


p/ T: [**A**]AB P: [**C**] * A * B

1. Repito **C** ? [i+1]
2. Usa string vazia e pulo pro próximo? [j+2]

```
match(i, j+2) or (first_match and match(i+1, j))
```

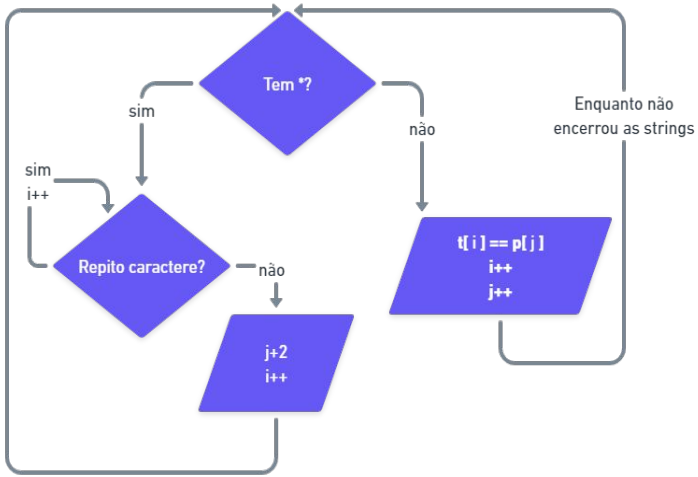
Árvore de Decisão



```
match(i, j+2) or (first_match and match(i+1, j))
```

Árvore de Decisão

```
text não terminou and
(t[i] == p[j] or p[j] == '.')
```

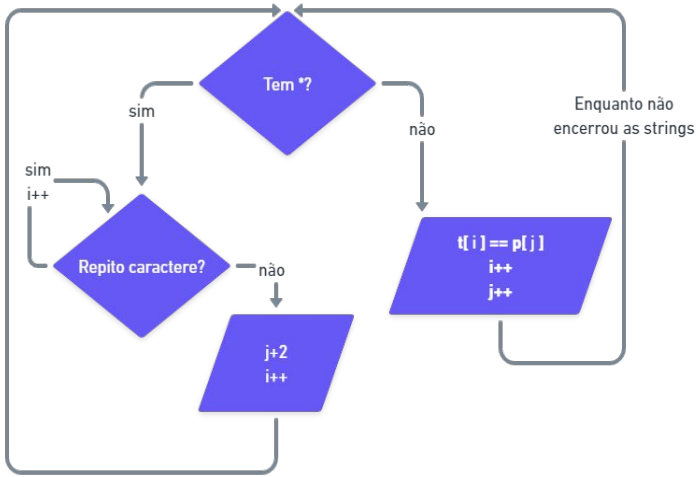


```
match(i, j+2) or (first_match and match(i+1, j))
```

Árvore de Decisão

`text não terminou` **and** `(t[i] == p[j]` **or** `p[j] == '.')`

Continuação da Repetição



Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    //definição da função match
3.    return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na **rodada atual**[1]

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)): //Verificar se as duas Str terminam juntas
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}//Verificar primeiro match[1]
6.        if (j+1 < len(pattern) and pattern[j+1] == '*')://Tem repetição
7.          return match(i, j+2) or (first_match and match(i+1, j))//Árvore de Decisão
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    //definição da função match
3.    return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na **rodada atual**[1]

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)): //Verificar se as duas Str terminam juntas
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}//Verificar primeiro match[1]
6.        if (j+1 < len(pattern) and pattern[j+1] == '*')://Tem repetição
7.          return match(i, j+2) or (first_match and match(i+1, j))//Árvore de Decisão
8.        else:
9.          return first_match and match(i+1, j+1)//Verificação sem repetição
```

Recursivo > Corretude

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.   def isMatch(text: str, pattern: str)-> bool:
2.     def match(i: int, j: int)...
3.     return match(0, 0)
```

Recursivo > Corretude

Algoritmo: isMatch(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.   def isMatch(text: str, pattern: str)-> bool:
2.     def match(i: int, j: int)...
3.     return match(0, 0)
```

Recursivo > Corretude

Algoritmo: isMatch(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.  def isMatch(text: str, pattern: str)-> bool:
2.    def match(i: int, j: int)...
3.    return match(0, 0)
```

Algoritmo: match(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na rodada atual

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)):
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or (first_match and match(i+1, j))
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo > Corretude

```
def isMatch(text: str, pattern: str)-> bool
```

- Pré-condições & Pós-condições (invariante do loop)
- Tamanho das instâncias
- Entrada Geral

Algoritmo: match(i, j)

<pré-condição>:

<pós-condição>:

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)):
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or first_match and match(i+1, j)
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo > Corretude

```
def isMatch(text: str, pattern: str)-> bool
```

- **Pré-condições & Pós-condições (invariante do loop)**
- Tamanho das instâncias
- Entrada Geral

Algoritmo: match(i, j)

<pré-condição>: **i,j** ,maiores ou iguais a 0 e menores que os tamanhos de **text** e **pattern**, respectivamente

<pós-condição>: Processado até o momento **text** é formada ou não por **pattern** (em toda rodada i ou j se aproximam do tamanho das entradas -> Caso base)

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)):
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or first_match and match(i+1, j)
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo > Corretude

```
def isMatch(text: str, pattern: str)-> bool
```

- Pré-condições & Pós-condições (invariante do loop)
- Tamanho das instâncias
- Entrada Geral

Algoritmo: match(i, j)

<pré-condição>: **i,j** ,maiores ou iguais a 0 e menores que os tamanhos de **text** e **pattern**, respectivamente

<pós-condição>: Processado até o momento **text** é formada ou não por **pattern** (em toda rodada i ou j se aproximam do tamanho das entradas -> Caso base)

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)):
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):
7.          return match(i, j+2) or first_match and match(i+1, j)
8.        else:
9.          return first_match and match(i+1, j+1)
```

Recursivo > Corretude

```
def isMatch(text: str, pattern: str)-> bool
```

- Pré-condições & Pós-condições (invariante do loop)
- Tamanho das instâncias
- Entrada Geral

Algoritmo: match(i, j)

<pré-condição>: **i,j** ,maiores que 0 e menores que os tamanhos de **text** e **pattern**, respectivamente

<pós-condição>: Processado até o momento **text** é formada ou não por **pattern** (em toda rodada i ou j se aproximam do tamanho das entradas -> Caso base)

```
1.  def match(i: int, j: int)-> bool:
2.      if (j == len(pattern)):          // CASO BASE
3.        return ( i == len(text) )
4.      else:
5.        first_match = i < len(text) and pattern[j] in {text[i], '.'}
6.        if (j+1 < len(pattern) and pattern[j+1] == '*'):          // CHAMADA DA
7.          return match(i, j+2) or first_match and match(i+1, j)      RECURSÃO
8.        else:
9.          return first_match and match(i+1, j+1)
```

Problema

Implementar verificação se uma determinada string é formada por uma expressão regular com suporte a '.' e '*'.

Definição: Dadas as strings **text** e **pattern** com tamanhos T e P, tal que t_i e p_j são elementos dada as posições, onde

- $0 \leq i < T$
- $0 \leq j < P$
- **text** é formada por caracteres `{a..z}`
- **pattern** é formada por caracteres (`a..z, *, .`).
    - `.` define um caractere qualquer em (`a..z`)
    - * define 0 ou n repetições do caractere antecessor, este pertencendo a `.`

Temos que `isMatch(text,pattern)` que retorna `Match(i,j)`, para $i = 0$ e $j = 0$, e este retorna se **text** é formada por **pattern** através do algoritmo.

- i, j sempre são incrementados, logo sempre vão se aproximar do caso base ($j = P$? [$i = T$? True : False] : chamada da recursão)
- **Caso Base:**
  - Se **pattern** encerrou ***And*** **Text** encerrou, e contando que os passos anteriores operaram de forma efetiva, Então **Text** é formada por **pattern** (**TRUE**).
  - Se **pattern** encerrou ***And*** **Text** não encerrou, e contando que os passos anteriores operaram de forma efetiva, Então **Text** não formada por **pattern** (FALSE).
- **Chamada da Recursão:** (Caso contrário)
  - Obtém o `first_match`:
    - Verifica se **Text** não encerrou ***And*** ($p_j = t_i$ ***Or*** $p_j =$ '.')
    - Ele vai servir para cancelar o processo quando **text** encerrar e o **pattern** não.
    - Ou para cancelar o processo quando não houver match entre t_i e p_j.
  - **Caso de haver Repetição**: (Se p_{j+1} existir ***&*** $p_{j+1} = *$)
    - Verifica `Match(i, j+2)` ***Or*** (`first_match` ***And*** `Match(i+1, j)`)
    - (Lembrando que o ***Or*** exige que exista pelo menos um dos valores seja verdadeira, ou seja, na primeira aparição de um valor verdadeiro poda as outras verificações)
    - `Match(i, j+2)`: Verifica se sem a repetição atual o o match ainda ocorre, logo ele não é necessário para formar **text**. Ele trás a necessidade de uma vefiação (`first_match` ***And*** `Match(i+1, j)`) quando falso.
    - `first_match`: Ele encerra a necessidade de uma verificação de `Match(i+1, j)`, repetições de p_j, em caso de valor falso.
    - `Match(i+1, j)`: Verifica o match do loop de p_j em t_i.
  - **Caso não houver repetição:**
    - Verifica `first_match` ***And*** `Match(i+1, j+1)`
    - (lembrando que o ***And*** exige que todos as suas verificações sejam verdadeiras podando na primeira aparição de uma valor Falso)
    - `first_match`: exclui a necessidade da verificação `Match(i+1, j+1)`, se o mesmo for falso, ou seja, se tratarmos de um **text** que encerrou ou se não houve match p_j e t_i.
    - `Match(i+1, j+1)`: Verifica o match p_j e t_i termo a termo para os casos onde não há 'loop'(*).

Programação Dinâmica

Programação Dinâmica

Algoritmo: **isMatch**(text, pattern)

Entrada: string ***Text*** a ser verificada e string ***Pattern*** um padrão.

Saída: booleano - se ***Text*** é formado por ***Pattern***

```
1.    def isMatch(text: str, pattern: str)-> bool:
2.      memo = {}
3.      def match(i: int, j: int)...
4.      return match(0, 0)
```

Programação Dinâmica

Algoritmo: **isMatch**(text, pattern)

Entrada: string ***Text*** a ser verificada e string ***Pattern*** um padrão.

Saída: booleano - se ***Text*** é formado por ***Pattern***

```
1.    def isMatch(text: str, pattern: str)-> bool:
2.      memo = {}
3.      def match(i: int, j: int)...
4.      return match(0, 0)
```

Programação Dinâmica

Algoritmo: **isMatch**(text, pattern)

Entrada: string ***Text*** a ser verificada e string ***Pattern*** um padrão.

Saída: booleano - se ***Text*** é formado por ***Pattern***

```
1.    def isMatch(text: str, pattern: str)-> bool:
2.      memo = {}
3.      def match(i: int, j: int)...
4.      return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de ***Text*** e posição j de ***Pattern***

Saída: booleano - Verificação match na rodada atual

```
1.    def match(i: int, j: int)-> bool:
2.            if((i,j) in memo): return memo[(i,j)]
3.            if (j == len(pattern)):
4.                memo[(i,j)] = ( i == len(text) )
5.                return memo[(i,j)]
6.            first_match = i < len(text) and pattern[j] in {text[i], '.'}
7.            if (j+1 < len(pattern) and pattern[j+1] == '*'):
8.                memo[(i,j)] = match(i, j+2) or (first_match and match(i+1, j))
9.                return memo[(i,j)]
10.           else:
11.               memo[(i,j)] = first_match and match(i+1, j+1)
12.               return memo[(i,j)]
```

Programação Dinâmica

Algoritmo: **isMatch**(text, pattern)

Entrada: string *Text* a ser verificada e string *Pattern* um padrão.

Saída: booleano - se *Text* é formado por *Pattern*

```
1.    def isMatch(text: str, pattern: str)-> bool:
2.      memo = {}
3.      def match(i: int, j: int)...
4.      return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de *Text* e posição j de *Pattern*

Saída: booleano - Verificação match na rodada atual

```
1.    def match(i: int, j: int)-> bool:
2.            if((i,j) in memo): return memo[(i,j)]
3.            if (j == len(pattern)):
4.                memo[(i,j)] = ( i == len(text) )
5.                return memo[(i,j)]
6.            first_match = i < len(text) and pattern[j] in {text[i], '.'}
7.            if (j+1 < len(pattern) and pattern[j+1] == '*'):
8.                memo[(i,j)] = match(i, j+2) or (first_match and match(i+1, j))
9.                return memo[(i,j)]
10.           else:
11.               memo[(i,j)] = first_match and match(i+1, j+1)
12.               return memo[(i,j)]
```

Programação Dinâmica

Algoritmo: **isMatch**(text, pattern)

Entrada: string ***Text*** a ser verificada e string ***Pattern*** um padrão.

Saída: booleano - se ***Text*** é formado por ***Pattern***

```
1.    def isMatch(text: str, pattern: str)-> bool:
2.      memo = {}
3.      def match(i: int, j: int)...
4.      return match(0, 0)
```

Algoritmo: **match**(i, j)

Entrada: posição i de ***Text*** e posição j de ***Pattern***

Saída: booleano - Verificação match na rodada atual

```
1.    def match(i: int, j: int)-> bool:
2.            if((i,j) in memo): return memo[(i,j)]
3.            if (j == len(pattern)):
4.                memo[(i,j)] = ( i == len(text) )
5.                return memo[(i,j)]
6.            first_match = i < len(text) and pattern[j] in {text[i], '.'}
7.            if (j+1 < len(pattern) and pattern[j+1] == '*'):
8.                memo[(i,j)] = match(i, j+2) or (first_match and match(i+1, j))
9.                return memo[(i,j)]
10.           else:
11.               memo[(i,j)] = first_match and match(i+1, j+1)
12.               return memo[(i,j)]
```

Casos de Teste

Testes

Text	Pattern	Resultado
a	a*	**V**
aab	c*a*b	**V**
aaaa	c*a*	**V**
abcx	.*	**V**
abcxa	abcxa	**V**
z	a*b*c*zk*	**V**
xccc	xc*	**V**
	.*	**V**
dasdasjda	.*	**V**
aaaax	.*a*	**V**

Text	Pattern	Resultado
aaaa	a	F
bbb	a*	F
bbb	ca*	F
bbb	b*c	F
aaaab	.*a*x	F
	b*ac*	F
xaxaxa	x*a*	F
dotatres	valve	F