10. Regular Expression Matching

Sept. 13, 2017 | 325.4K views

Given an input string (`s`) and a pattern (`p`), implement regular expression matching with support for `'.'` and `'*'`.

```
'.' Matches any single character.
'*' Matches zero or more of the preceding element.
```

The matching should cover the **entire** input string (not partial).

Note:

- `s` could be empty and contains only lowercase letters `a-z`.
- `p` could be empty and contains only lowercase letters `a-z`, and characters like `.` or `*`.

Example 1:

```
Input:
s = "aa"
p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input:
s = "aa"
p = "a*"
Output: true
Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by repea
```

Example 3:

```
Input:
s = "ab"
p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any character (.)".
```

Example 4:

```
Input:
s = "aab"
p = "c*a*b"
Output: true
Explanation: c can be repeated 0 times, a can be repeated 1 time. Therefore, it matche
```

Example 5:

```
Input:
s = "mississippi"
p = "mis*is*p*."
Output: false
```

Solution

Approach 1: Recursion

Intuition

If there were no Kleene stars (the `*` wildcard character for regular expressions), the problem would be easier - we simply check from left to right if each character of the text matches the pattern.

When a star is present, we may need to check many different suffixes of the text and see if they match the rest of the pattern. A recursive solution is a straightforward way to represent this relationship.

Algorithm

Without a Kleene star, our solution would look like this:

Python | Copy

```
def match(text, pattern):
    if not pattern: return not text
    first_match = bool(text) and pattern[0] in {text[0], '.'}
    return first_match and match(text[1:], pattern[1:])
```

If a star is present in the pattern, it will be in the second position $\text{pattern}[1]$. Then, we may ignore this part of the pattern, or delete a matching character in the text. If we have a match on the remaining strings after any of these operations, then the initial inputs matched.

Java | Python | Copy

```
class Solution(object):
    def isMatch(self, text, pattern):
        if not pattern:
            return not text

        first_match = bool(text) and pattern[0] in {text[0], '.'}

        if len(pattern) >= 2 and pattern[1] == '*':
            return (self.isMatch(text, pattern[2:]) or
                    first_match and self.isMatch(text[1:], pattern))
        else:
            return first_match and self.isMatch(text[1:], pattern[1:])
```

Complexity Analysis

- Time Complexity: Let T, P be the lengths of the text and the pattern respectively. In the worst case, a call to `match(text[i:], pattern[2j:])` will be made $\binom{i+j}{i}$ times, and strings of the order $O(T - i)$ and $O(P - 2*j)$ will be made. Thus, the complexity has the order $\sum_{i=0}^{T}\sum_{j=0}^{P/2}\binom{i+j}{i}O(T+P-i-2j)$. With some effort outside the scope of this article, we can show this is bounded by $O\big((T+P)2^{T+\frac{P}{2}}\big)$.
- Space Complexity: For every call to `match`, we will create those strings as described above, possibly creating duplicates. If memory is not freed, this will also take a total of $O\big((T+P)2^{T+\frac{P}{2}}\big)$ space, even though there are only order $O(T^2+P^2)$ unique suffixes of P and T that are actually required.

Approach 2: Dynamic Programming

Intuition

As the problem has an **optimal substructure**, it is natural to cache intermediate results. We ask the question dp(i, j): does text[i:] and pattern[j:] match? We can describe our answer in terms of answers to questions involving smaller strings.

Algorithm

We proceed with the same recursion as in Approach 1, except because calls will only ever be made to `match(text[i:], pattern[j:])`, we use dp(i, j) to handle those calls instead, saving us expensive string-building operations and allowing us to cache the intermediate results.

Top-Down Variation

Java | Python | Copy

```
class Solution(object):
    def isMatch(self, text, pattern):
        memo = {}
        def dp(i, j):
            if (i, j) not in memo:
                if j == len(pattern):
                    ans = i == len(text)
                else:
                    first_match = i < len(text) and pattern[j] in {text[i], '.'}
                    if j+1 < len(pattern) and pattern[j+1] == '*':
                        ans = dp(i, j+2) or first_match and dp(i+1, j)
                    else:
                        ans = first_match and dp(i+1, j+1)

                memo[i, j] = ans
            return memo[i, j]

        return dp(0, 0)
```

Bottom-Up Variation

Java | Python | Copy

```
class Solution(object):
    def isMatch(self, text, pattern):
        dp = [[False] * (len(pattern) + 1) for _ in range(len(text) + 1)]

        dp[-1][-1] = True
        for i in range(len(text), -1, -1):
            for j in range(len(pattern) - 1, -1, -1):
                first_match = i < len(text) and pattern[j] in {text[i], '.'}
                if j+1 < len(pattern) and pattern[j+1] == '*':
                    dp[i][j] = dp[i][j+2] or first_match and dp[i+1][j]
                else:
                    dp[i][j] = first_match and dp[i+1][j+1]

        return dp[0][0]
```

Complexity Analysis

- Time Complexity: Let T, P be the lengths of the text and the pattern respectively. The work for every call to `dp(i, j)` for $i = 0, ..., T$; $j = 0, ..., P$ is done once, and it is $O(1)$ work. Hence, the time complexity is $O(TP)$.
- Space Complexity: The only memory we use is the $O(TP)$ boolean entries in our cache. Hence, the space complexity is $O(TP)$.