

## Metadata

Note\_Type::Literature Note

Topics::[Dynamic Programming](#)

Source\_Url::[LeetCode-Problem 10, Dynamic Programming - Top down and Bottom up](#)

---

## 說明

### 題目簡述

鼎鼎大名的正則表示法(Regular Expression Matching)，幾乎包含於各個熱門程式語言中。此題目並未囊括所有表示法，但是包含 `.`、`*` 兩種：

- `*`：表示可以比對前一個字元**零次至多次**
- `.`：表示可以比對單一任意字元

題目範例：

#### 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 repeating 'a' once, it becomes "aa".

#### Example 3:

**Input:** s = "ab", p = ".\*"

**Output:** true

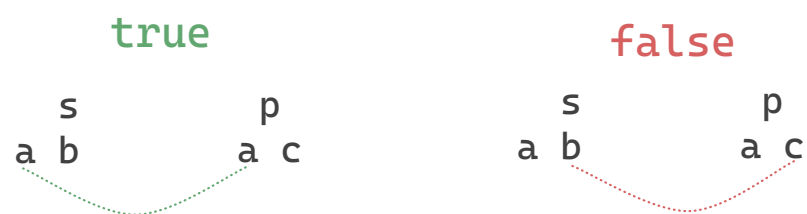
**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

## 解題思路

須考慮的情境如下：

1. `s` 當前字元與 `p` 是否相同
2. 若 `p` 當前字元為 `.`，`s` 是否為**有效字元**
3. 若 `p` 下個字元為 `*`，須額外配對前一個字元的情境

情境一：s 當前字元與 p 是否相同

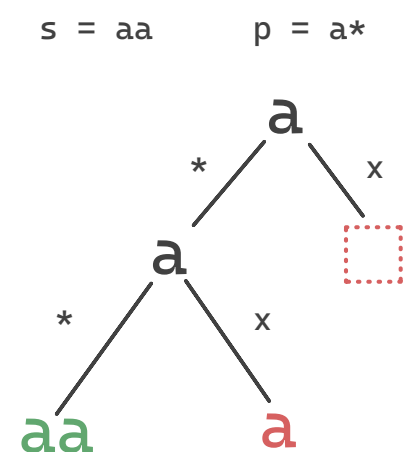


情境二：若 p 當前字元為 .，s 是否為有效字元



情境三：若 p 下個字元為 \*，須額外配對前一個字元的情境

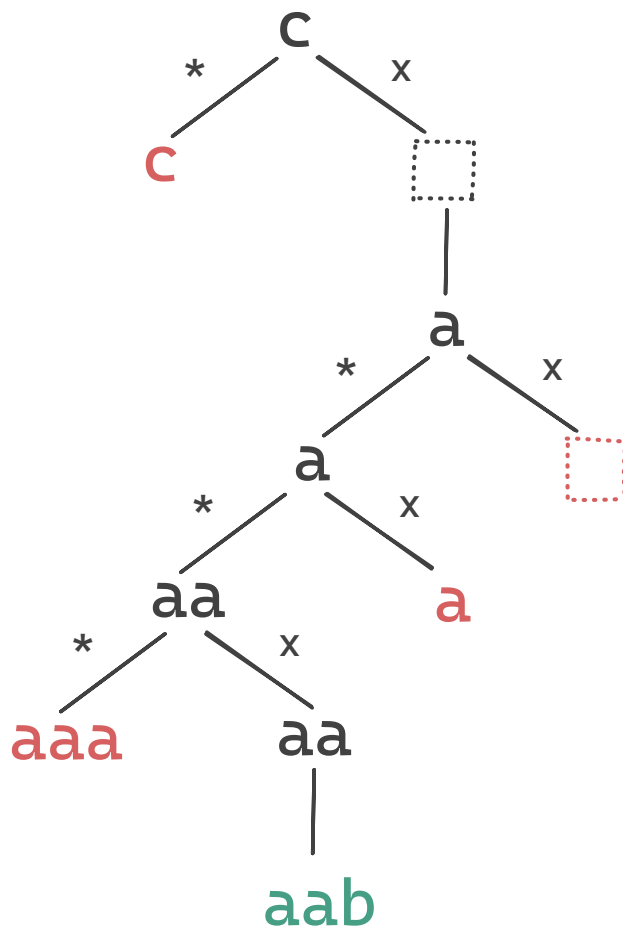
每當我們配對到 \* 字元時，可以選擇**零次或一次**的前一個字元。此種二元選擇可以歸納為一個決策樹來展示我們的配對：



\* 後方還有字符時，也會因為前面的選擇更動：

s = aab

p = c\*a\*b



## 演算法設計

### 建立考慮未來結果的演算法？

要將未來結果視為條件進行判斷，可以考慮的方向：

1. 把現有的結果暫存起來，直到最後將結果全部一起看 (Top down)
2. 將演算的順序顛倒，直接引用未來的值 (Bottom Up)

## Dynamic Programming - Bottom Up

- 建立動態規劃陣列

預計是以陣列最後一位開始，由右至左、由下到上遍歷，而動態規劃的最終結果便會出現在第一位。

```
class Solution
{
public:
    bool isMatch(string s, string p)
```

```

{
    // 建立動態規劃陣列
    vector<vector<int>> dp(s.length() + 1, vector<int>(p.length() + 1, 0));
    // 初始值為 1(true)，將前一筆匹配紀錄持續演算下去
    dp[s.length()][p.length()] = 1;

    // 遍歷所有`s`與`p`字元配對
    // ...

    // 當前`s`與`p`相同?
    // ...

    // `p`下個字元為`*`?
    // ...

    // 前幾筆的配對也是成功?
    // ...

    return dp[0][0];
}
};

```

- 遍歷所有 s 與 p 字元配對

s 的索引初始值(s.length())之所以比 p 的初始值(p.length() - 1) 多一，是考慮配對 s 當前字元為空的情況。

```

class Solution
{
public:
    bool isMatch(string s, string p)
    {
        // 建立動態規劃陣列
        vector<vector<int>> dp(...);

        // 遍歷所有`s`與`p`字元配對
        for (int i = s.length(); i ≥ 0; i--)
        {
            for (int j = p.length() - 1; j ≥ 0; j--)
            {
                // 當前`s`與`p`相同?
                // ...

                // `p`下個字元為`*`?
                // ...

                // 前幾筆的配對也是成功?
                // ...
            }
        }

        return dp[0][0];
    }
};

```

- s 與 p 當前字元相同?

```
class Solution
{
public:
    bool isMatch(string s, string p)
    {
        // 建立動態規劃陣列
        vector<vector<int>>> dp(...);

        // 遍歷所有`s`與`p`字元配對
        for (int i = s.length(); i ≥ 0; i--)
        {
            for (int j = p.length() - 1; j ≥ 0; j--)
            {
                // 當前`s`與`p`相同?
                bool curr_match = i ≤ s.length() && p[j] == '.' || p[j] == s[i];

                // `p`下個字元為`*`?
                // ...

                // 前幾筆的配對也是成功?
                // ...
            }
        }

        return dp[0][0];
    }
};
```

- 前幾筆的配對也是成功?

```
class Solution
{
public:
    bool isMatch(string s, string p)
    {
        // 建立動態規劃陣列
        vector<vector<int>>> dp(...);

        // 遍歷所有`s`與`p`字元配對
        for (int i = s.length(); i ≥ 0; i--)
        {
            for (int j = p.length() - 1; j ≥ 0; j--)
            {
                // 當前`s`與`p`相同?
                bool curr_match = i < s.length() && p[j] == '.' || p[j] == s[i];

                // `p`下個字元為`*`?
                if (j + 1 < p.length() && p[j + 1] == '*')
                {
                    // ...
                }
                // 前幾筆的配對也是成功?
                else
                {

```

```

        dp[i][j] = curr_match && dp[i + 1][j + 1]; // 當前的配對結果
        // 前一筆的配對結果
    }
}

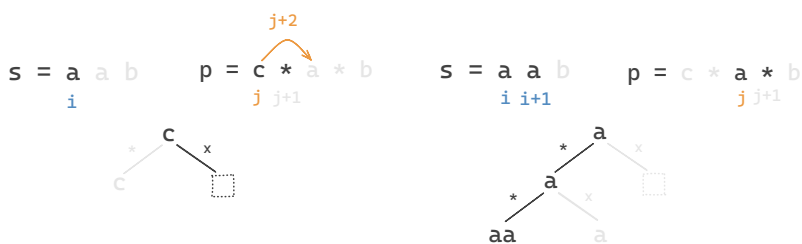
return dp[0][0];
};

```

- p 下個字元為 \* 的情況

\* 匹配數量為 0

\* 匹配數量為多筆



\* 規則是依據前個字元來決定配對是否成功

所以當配對 s 與 p 當前字元(i、j)時，會觀察 p 下個字元(j + 1)是否為 \*，以考慮所有組合。

**需要 \* 配對零次**情境下，不需要當前字元配對，所以跳過 \* 找下個 p 字元(j + 2)，並固定 s 當前字元(i)配對。

**需要 \* 配對多筆**情境下，需要當前字元持續配對，所以持續找 s 下個字元(i + 1)，並固定當前 p 字元(j)配對。

```

class Solution
{
public:
    bool isMatch(string s, string p)
    {
        // 建立動態規劃陣列
        vector<vector<int>>> dp(...);

        // 遍歷所有`s`與`p`字元配對
        for (int i = s.length(); i >= 0; i--)
        {
            for (int j = p.length() - 1; j >= 0; j--)
            {
                // 當前`s`與`p`相同?
                bool curr_match = i < s.length() && p[j] == '.' || p[j] == s[i];

                // `p`下個字元為`*`?
                if (j + 1 < p.length() && p[j + 1] == '*')
                {
                    dp[i][j] = dp[i][j + 2] || // `*` 配對零筆
                               (curr_match && dp[i + 1][j]); // `*` 配對一筆
                }
                // 當前字元匹配相同時
            }
        }
    }
};

```

```

        else
        {
            dp[i][j] = curr_match && dp[i + 1][j + 1];
        }
    }
}

return dp[0][0];
}
};

```

## Dynamic Programming - Top down

```

class Solution
{
    string ss, pp;
public:
    bool dfs(int i, int j)
    {
        // 兩字串配對成功?
        if (j == pp.length())
        {
            return i == ss.length();
        }

        // 當前字元配對成功?
        bool curr_match = i < ss.length() && pp[j] == '.' || pp[j] == ss[i];

        if (j + 1 < pp.length() && pp[j + 1] == '*')
        {
            return dfs(i, j + 2) || // `*`配對零次
                curr_match && dfs(i + 1, j); // `*`配對一次
        }

        return curr_match && dfs(i + 1, j + 1); // 現在、下次配對成功?
    }

    bool isMatch(string s, string p)
    {
        ss = s, pp = p;
        return dfs(0, 0);
    }
};

```