LeetCode 10: Regular Expression Matching

- Tong Wu
- Sizhi Chen

Northeastern University

Question Definition

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '*' where:
•'.' Matches any single character.
•'*' Matches zero or more of the preceding element.
The matching should cover the **entire** input string (not partial).

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 (.)".

Constraints:
•1 <= s.length <= 20
•1 <= p.length <= 20
•s contains only lowercase English letters.
•p contains only lowercase English letters, '.', and '*'.
•It is guaranteed for each appearance of the character '*', there will be a previous valid character to match. [4]

Introduction to Regular Expressions

- Regular Expressions (Regex) are sequences of characters that define a search pattern. They are widely used in:
    1. **String matching**
    2. **Text processing**
    3. **Pattern recognition.**
- Common Regex Symbols [1]

Symbol	Meaning
.	Matches any single character.
*	Matches zero or more of the preceding character.
+	Matches one or more of the preceding character.
?	Matches zero or one of the preceding character.
[ ]	Matches any one character inside the brackets.

Dynamic Programming

Dynamic Programming (DP) is an **optimization technique** used to solve problems by **breaking them down into smaller overlapping subproblems** and storing their results to avoid redundant computations.

Instead of solving the same subproblem multiple times (as in recursion), **DP stores the results** (memorization) or builds solutions iteratively (bottom-up approach).

Why Use Dynamic Programming?

a. **Avoids Redundant Computation**
   - Recursion may recompute the same subproblem multiple times, leading to **exponential time complexity (O(2^n))**.
   - DP **stores results** to prevent recomputation, reducing complexity to **polynomial time (O(n^2) or O(n))**.

b. **Optimizes Recursive Solutions**
   - **Memorization (Top-Down DP)**: Uses recursion + caching.
   - **Tabulation (Bottom-Up DP)**: Uses iterative table-filling. [2]

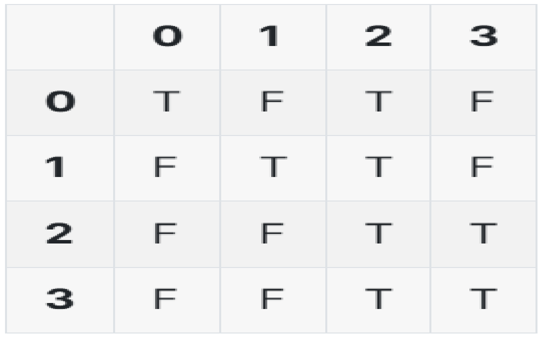
Overall Strategy

Solution1: Top-down with memorization (recursive DFS):

- The key here is to realize that the problem breaks down into smaller subproblems. If we know how smaller parts of the string and pattern match, we can use those results to solve for larger parts.

Solution2: Bottom-up dynamic programming (iterative)

- Builds up from smaller subproblems to the full problem.
- Create a 2D table f where f[i][j] will represent whether the first i characters of string match the first j characters of pattern. And fill it with nested loop. [3]



	0	1	2	3
0	T	F	T	F
1	F	T	T	F
2	F	F	T	T
3	F	F	T	T

Connection to what we know

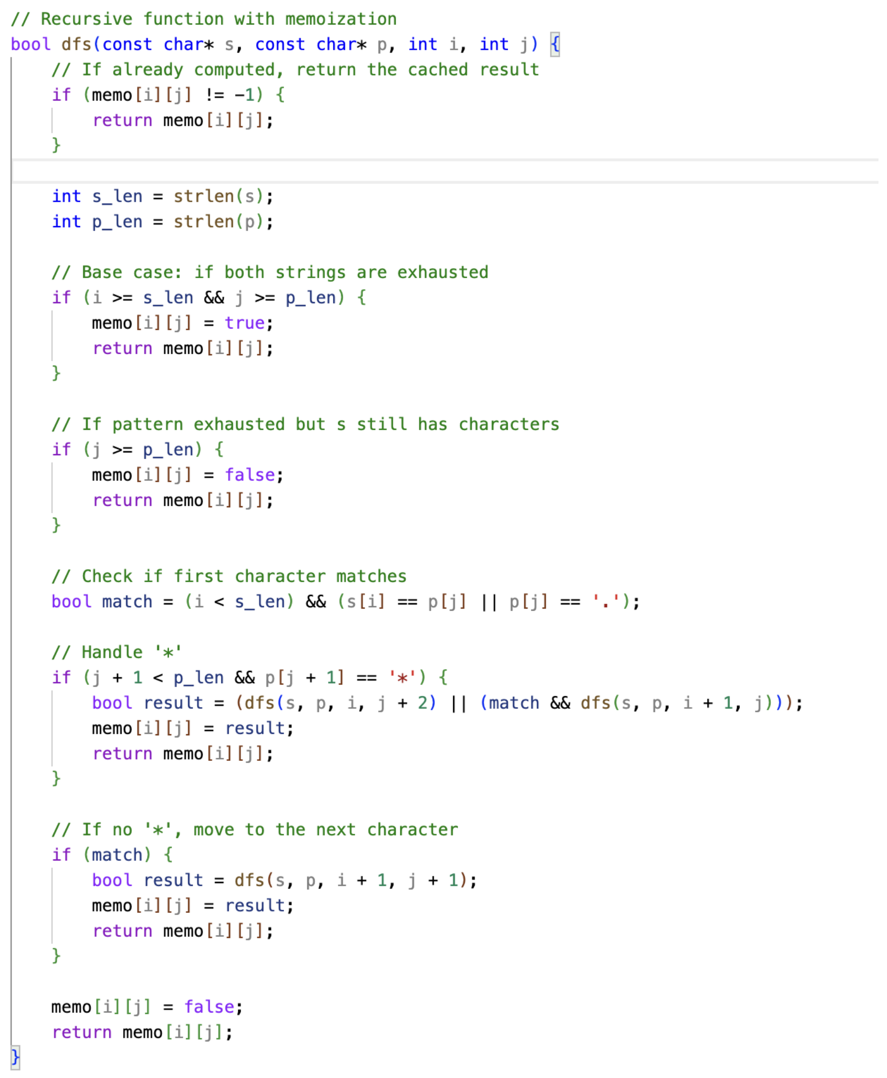
Solution1:

- Using depth-first search which creates a decision tree that explores all possible matching paths between the string and pattern.
- Using recursive method to go through decision tree
- Using pointer to traverse through both string and pattern

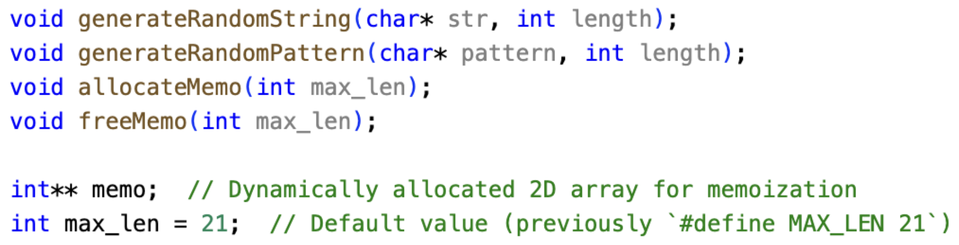
Solution2:

- Using 2d array to contain dynamic programming table
- Using nested loop to fill out the table

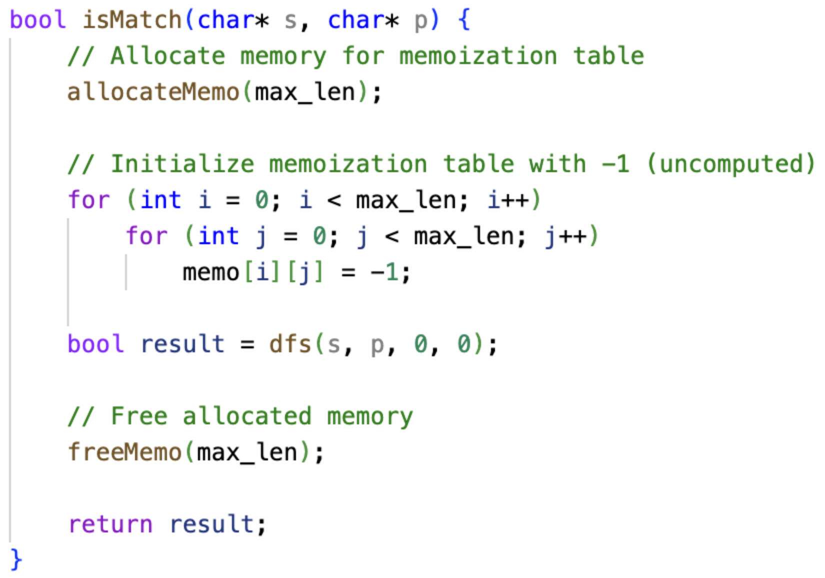
Solution 1



```c
// Recursive function with memoization
bool dfs(const char* s, const char* p, int i, int j) {
    // If already computed, return the cached result
    if (memo[i][j] != -1) {
        return memo[i][j];
    }

    int s_len = strlen(s);
    int p_len = strlen(p);

    // Base case: if both strings are exhausted
    if (i >= s_len && j >= p_len) {
        memo[i][j] = true;
        return memo[i][j];
    }

    // If pattern exhausted but s still has characters
    if (j >= p_len) {
        memo[i][j] = false;
        return memo[i][j];
    }

    // Check if first character matches
    bool match = (i < s_len) && (s[i] == p[j] || p[j] == '.');

    // Handle '*'
    if (j + 1 < p_len && p[j + 1] == '*') {
        bool result = (dfs(s, p, i, j + 2) || (match && dfs(s, p, i + 1, j)));
        memo[i][j] = result;
        return memo[i][j];
    }

    // If no '*', move to the next character
    if (match) {
        bool result = dfs(s, p, i + 1, j + 1);
        memo[i][j] = result;
        return memo[i][j];
    }

    memo[i][j] = false;
    return memo[i][j];
}
```

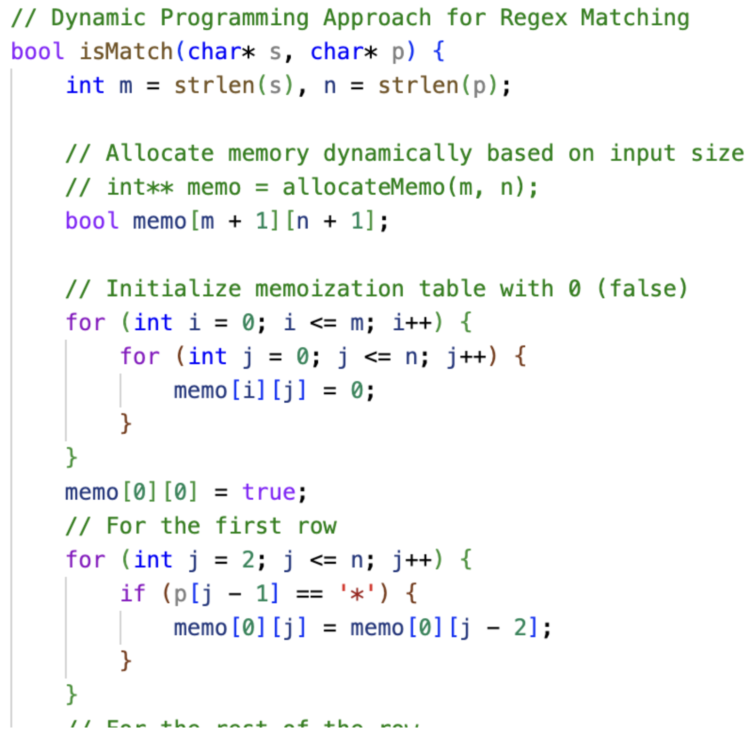


```c
void generateRandomString(char* str, int length);
void generateRandomPattern(char* pattern, int length);
void allocateMemo(int max_len);
void freeMemo(int max_len);

int** memo;  // Dynamically allocated 2D array for memoization
int max_len = 21;  // Default value (previously `#define MAX_LEN 21`)
```

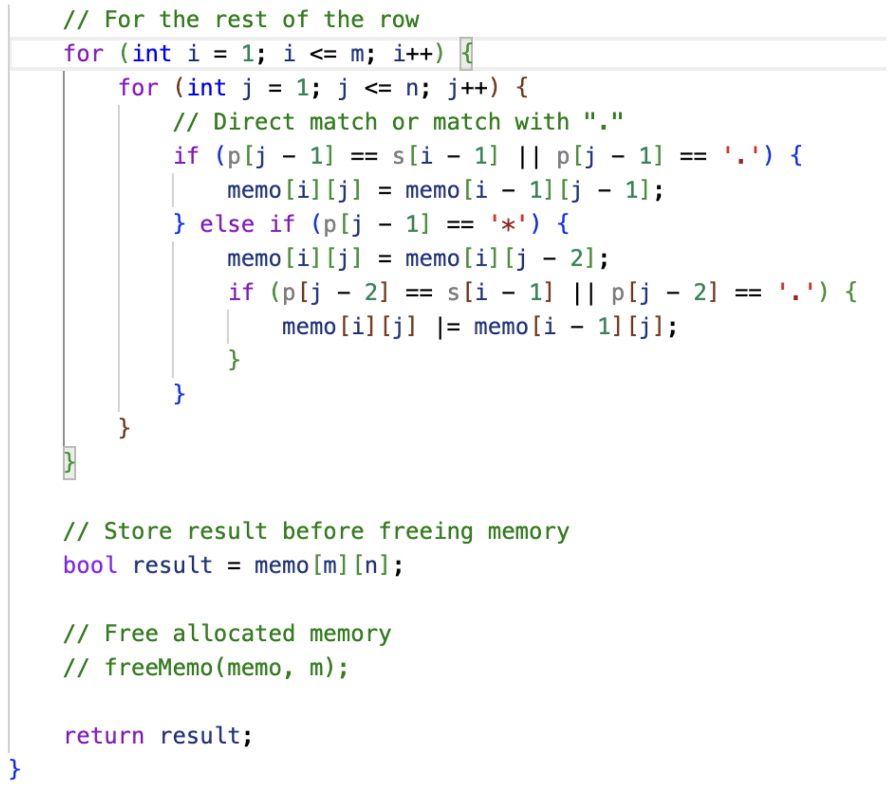


```c
bool isMatch(char* s, char* p) {
    // Allocate memory for memoization table
    allocateMemo(max_len);

    // Initialize memoization table with -1 (uncomputed)
    for (int i = 0; i < max_len; i++)
        for (int j = 0; j < max_len; j++)
            memo[i][j] = -1;

    bool result = dfs(s, p, 0, 0);

    // Free allocated memory
    freeMemo(max_len);

    return result;
}
```

Solution 2



```
// Dynamic Programming Approach for Regex Matching
bool isMatch(char* s, char* p) {
    int m = strlen(s), n = strlen(p);

    // Allocate memory dynamically based on input size
    // int** memo = allocateMemo(m, n);
    bool memo[m + 1][n + 1];

    // Initialize memoization table with 0 (false)
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            memo[i][j] = 0;
        }
    }
    memo[0][0] = true;
    // For the first row
    for (int j = 2; j <= n; j++) {
        if (p[j - 1] == '*') {
            memo[0][j] = memo[0][j - 2];
        }
    }
    // For the rest of the row
```



```
    // For the rest of the row
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            // Direct match or match with "."
            if (p[j - 1] == s[i - 1] || p[j - 1] == '.') {
                memo[i][j] = memo[i - 1][j - 1];
            } else if (p[j - 1] == '*') {
                memo[i][j] = memo[i][j - 2];
                if (p[j - 2] == s[i - 1] || p[j - 2] == '.') {
                    memo[i][j] |= memo[i - 1][j];
                }
            }
        }
    }

    // Store result before freeing memory
    bool result = memo[m][n];

    // Free allocated memory
    // freeMemo(memo, m);

    return result;
}
```

Bibliography

- [1] Wikipedia contributors. (2025, March 20). Regular expression. Wikipedia. https://en.wikipedia.org/wiki/Regular_expression
- [2] BasuMallick, C. (2024, May 13). A Simplified Guide to Dynamic Programming - Spiceworks Inc. Spiceworks Inc. https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/
- [3] "Regular Expression Matching," Algo.Monster. [Online]. Available: https://algo.monster/liteproblems/10. [Accessed: Mar. 22, 2025].
- [4] "Regular Expression Matching," LeetCode. [Online]. Available: https://leetcode.com/problems/regular-expression-matching/description/. [Accessed: Mar. 22, 2025].
- **[5] github repository: https://github.com/SizhiChen/5008-inClass-Presentation.git**