756. Pyramid Transition Matrix

Solved

Medium | Topics | Companies

You are stacking blocks to form a pyramid. Each block has a color, which is represented by a single letter. Each row of blocks contains **one less block** than the row beneath it and is centered on top.

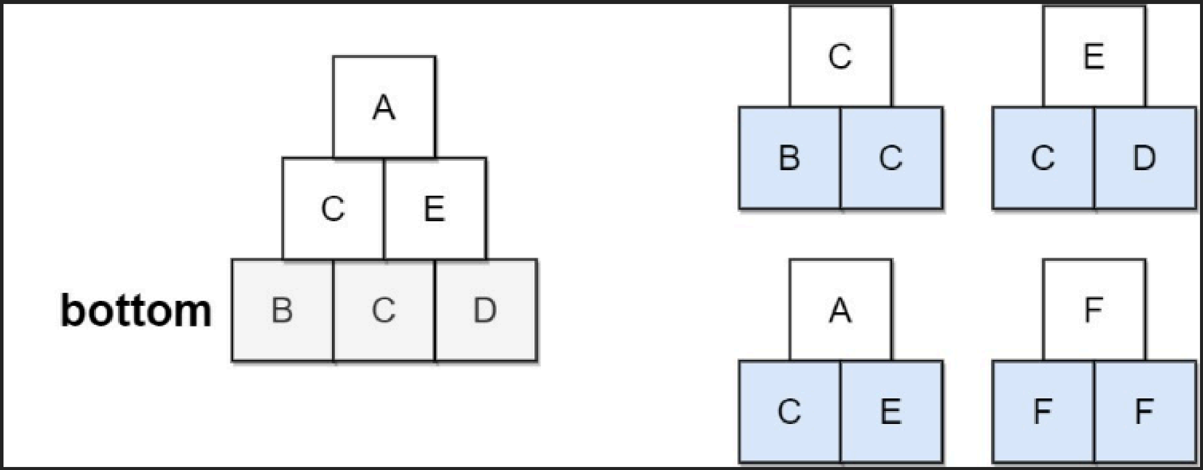
To make the pyramid aesthetically pleasing, there are only specific **triangular patterns** that are allowed. A triangular pattern consists of a **single block** stacked on top of **two blocks**. The patterns are given as a list of three-letter strings `allowed`, where the first two characters of a pattern represent the left and right bottom blocks respectively, and the third character is the top block.

- For example, `"ABC"` represents a triangular pattern with a `'C'` block stacked on top of an `'A'` (left) and `'B'` (right) block. Note that this is different from `"BAC"` where `'B'` is on the left bottom and `'A'` is on the right bottom.

You start with a bottom row of blocks `bottom`, given as a single string, that you **must** use as the base of the pyramid.

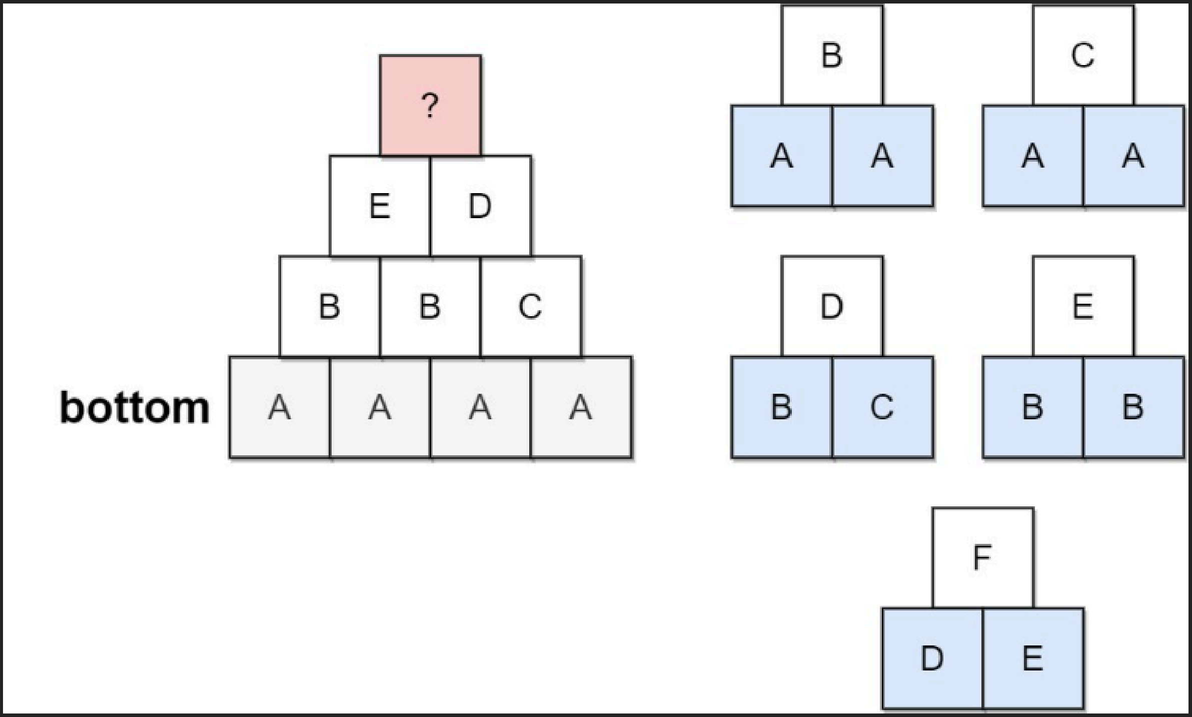
Given `bottom` and `allowed`, return `true` *if you can build the pyramid all the way to the top such that **every triangular pattern** in the pyramid is in* `allowed`, *or* `false` *otherwise.*

Example 1:



```
Input: bottom = "BCD", allowed = ["BCC","CDE","CEA","FFF"]
Output: true
Explanation: The allowed triangular patterns are shown on the right.
Starting from the bottom (level 3), we can build "CE" on level 2 and
then build "A" on level 1.
There are three triangular patterns in the pyramid, which are "BCC",
"CDE", and "CEA". All are allowed.
```

Example 2:



```
Input: bottom = "AAAA", allowed = ["AAB","AAC","BCD","BBE","DEF"]
Output: false
Explanation: The allowed triangular patterns are shown on the right.
Starting from the bottom (level 4), there are multiple ways to build
level 3, but trying all the possibilites, you will get always stuck
before building level 1.
```

Constraints:

- `2 <= bottom.length <= 6`
- `0 <= allowed.length <= 216`
- `allowed[i].length == 3`
- The letters in all input strings are from the set `{'A', 'B', 'C', 'D', 'E', 'F'}`.
- All the values of `allowed` are **unique**.

Python:

```
class Solution:
    def pyramidTransition(self, bottom: str, allowed: List[str]) -> bool:
        allowed_map = defaultdict(list)
        for a in allowed:
            allowed_map[a[:2]].append(a[2])
        memo = {}

        def rec(bottom):
            if len(bottom) == 1:
                return True
            if bottom in memo:
                return memo[bottom]

            possible_tops = []
            for i in range(1, len(bottom)):
                pair = bottom[i-1] + bottom[i]
                if pair in allowed_map:
                    possible_tops.append(allowed_map[pair])
                else:
                    # If any pair has no valid top, this row is a dead end
                    memo[bottom] = False
                    return False

            # Cartesian product to generate all possible next rows
            # This is the "transition" step where we branch out
            for next_top_tuple in product(*possible_tops):
                next_top = "".join(next_top_tuple)
                if rec(next_top):
                    memo[bottom] = True
                    return True

            # If no combination works
            memo[bottom] = False
            return False

        return rec(bottom)
```

JavaScript:

```
var pyramidTransition = function(bottom, allowed) {
    const map = new Map();
    for (const triplet of allowed) {
        const [a, b, c] = triplet;
        const key = a + b;
        if (!map.has(key)) map.set(key, []);
        map.get(key).push(c);
    }
    const memo = new Map();

    const dfs = (row) => {
        if (row.length === 1) return true;
        if (memo.has(row)) return memo.get(row);

        for (let i = 0; i < row.length - 1; i++) {
            if (!map.has(row[i] + row[i + 1])) {
                memo.set(row, false);
                return false;
            }
        }

        const helper = (i, cur) => {
            if (i === row.length - 1) return dfs(cur);
            const pair = row[i] + row[i + 1];
            for (const c of map.get(pair)) {
                if (helper(i + 1, cur + c)) return true;
            }
            return false;
        };

        const res = helper(0, "");
        memo.set(row, res);
        return res;
    };

    return dfs(bottom);
};
```

Java:

```
class Solution {
    Map<String, List<Character>> map = new HashMap<>();
    Map<String, Boolean> memo = new HashMap<>();

    public boolean pyramidTransition(String bottom, List<String> allowed) {
        for (String s : allowed) {
            String key = s.substring(0, 2);
            map.computeIfAbsent(key, k -> new ArrayList<>()).add(s.charAt(2));
        }
        return solve(bottom);
    }

    private boolean solve(String bottom) {
        if (bottom.length() == 1) return true;
        if (memo.containsKey(bottom)) return memo.get(bottom);

        // Try to generate a valid next row and solve recursively
        boolean result = generateNextRow(bottom, 0, new StringBuilder());
        memo.put(bottom, result);
        return result;
    }

    // Helper to generate next row candidates (DFS within the level)
    private boolean generateNextRow(String bottom, int idx, StringBuilder currentNext) {
        if (idx == bottom.length() - 1) {
            // Full next row built, proceed to next level
            return solve(currentNext.toString());
        }

        String key = bottom.substring(idx, idx + 2);
        if (!map.containsKey(key)) return false;

        for (char val : map.get(key)) {
            currentNext.append(val);
            if (generateNextRow(bottom, idx + 1, currentNext)) return true;
            currentNext.deleteCharAt(currentNext.length() - 1); // backtrack
        }

        return false;
    }
}
```