Interleaving String

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.

For example,
Given:
s1 = `"aabcc"`,
s2 = `"dbbca"`,

When *s3* = `"aadbbcbcac"`, return true.
When *s3* = `"aadbbbaccc"`, return false.

Solution 1

```
 bool isInterleave(string s1, string s2, string s3) {

    if(s3.length() != s1.length() + s2.length())
        return false;

    bool table[s1.length()+1][s2.length()+1];

    for(int i=0; i<s1.length()+1; i++)
        for(int j=0; j< s2.length()+1; j++){
            if(i==0 && j==0)
                table[i][j] = true;
            else if(i == 0)
                table[i][j] = ( table[i][j-1] && s2[j-1] == s3[i+j-1]);
            else if(j == 0)
                table[i][j] = ( table[i-1][j] && s1[i-1] == s3[i+j-1]);
            else
                table[i][j] = (table[i-1][j] && s1[i-1] == s3[i+j-1] ) || (table[
i][j-1] && s2[j-1] == s3[i+j-1] );
        }

    return table[s1.length()][s2.length()];
}
```

Here is some explanation:

DP table represents if s3 is interleaving at (i+j)th position when s1 is at ith position, and s2 is at jth position. 0th position means empty string.

So if both s1 and s2 is currently empty, s3 is empty too, and it is considered interleaving. If only s1 is empty, then if previous s2 position is interleaving and current s2 position char is equal to s3 current position char, it is considered interleaving. similar idea applies to when s2 is empty. when both s1 and s2 is not empty, then if we arrive i, j from i-1, j, then if i-1,j is already interleaving and i and current s3 position equal, it s interleaving. If we arrive i,j from i, j-1, then if i, j-1 is already interleaving and j and current s3 position equal. it is interleaving.

written by sherryxmhe original link here

Solution 2

If we expand the two strings s1 and s2 into a chessboard, then this problem can be transferred into a path seeking problem from the top-left corner to the bottom-right corner. The key is, each cell (y, x) in the board corresponds to an interval between y-th character in s1 and x-th character in s2. And adjacent cells are connected with like a grid. A BFS can then be efficiently performed to find the path.

Better to illustrate with an example here:

Say s1 = "aab" and s2 = "abc". s3 = "aaabcb". Then the board looks like

```
o--a--o--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--o--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--o--b--o--c--o
|     |     |     |
b     b     b     b
|     |     |     |
o--a--o--b--o--c--o
```

Each "o" is a cell in the board. We start from the top-left corner, and try to move right or down. If the next char in s3 matches the edge connecting the next cell, then we're able to move. When we hit the bottom-right corner, this means s3 can be represented by interleaving s1 and s2. One possible path for this example is indicated with "x"es below:

```
x--a--x--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--x--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--x--b--x--c--x
|     |     |     |
b     b     b     b
|     |     |     |
o--a--o--b--o--c--x
```

Note if we concatenate the chars on the edges we went along, it's exactly s3. And we went through all the chars in s1 and s2, in order, exactly once.

Therefore if we view this board as a graph, such path finding problem is trivial with BFS. I use an `unordered_map` to store the visited nodes, which makes the code look a bit complicated. But a `vector` should be enough to do the job.

Although the worse case timeis also O(mn), typically it doesn't require us to go through every node to find a path. Therefore it's faster than regular DP than average.

```
struct MyPoint {
    int y, x;
    bool operator==(const MyPoint &p) const {
        return p.y == y && p.x == x;
    }
};
namespace std {
    template <>
    struct hash<MyPoint> {
        size_t operator () (const MyPoint &f) const {
            return (std::hash<int>()(f.x) << 1) ^ std::hash<int>()(f.y);
        }
    };
}

class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s1.size() + s2.size() != s3.size()) return false;

        queue<MyPoint> q;
        unordered_set<MyPoint> visited;
        bool isSuccessful = false;
        int i = 0;

        q.push(MyPoint { 0, 0 });
        q.push(MyPoint { -1, -1 });
        while (!(1 == q.size() && -1 == q.front().x)) {
            auto p = q.front();
            q.pop();
            if (p.y == s1.size() && p.x == s2.size()) {
                return true;
            }
            if (-1 == p.y) {
                q.push(p);
                i++;
                continue;
            }
            if (visited.find(p) != visited.end()) { continue; }
            visited.insert(p);

            if (p.y < s1.size()) { // down
                if (s1[p.y] == s3[i]) { q.push(MyPoint { p.y + 1, p.x }); }
            }
            if (p.x < s2.size()) { // right
                if (s2[p.x] == s3[i]) { q.push(MyPoint { p.y, p.x + 1 }); }
            }
        }
        return false;
    }
};
```

written by grapeot original link here

Solution 3

```
public boolean isInterleave(String s1, String s2, String s3) {

    if ((s1.length()+s2.length())!=s3.length()) return false;

    boolean[][] matrix = new boolean[s2.length()+1][s1.length()+1];

    matrix[0][0] = true;

    for (int i = 1; i < matrix[0].length; i++){
        matrix[0][i] = matrix[0][i-1]&&(s1.charAt(i-1)==s3.charAt(i-1));
    }

    for (int i = 1; i < matrix.length; i++){
        matrix[i][0] = matrix[i-1][0]&&(s2.charAt(i-1)==s3.charAt(i-1));
    }

    for (int i = 1; i < matrix.length; i++){
        for (int j = 1; j < matrix[0].length; j++){
            matrix[i][j] = (matrix[i-1][j]&&(s2.charAt(i-1)==s3.charAt(i+j-1)))
                    || (matrix[i][j-1]&&(s1.charAt(j-1)==s3.charAt(i+j-1)));
        }
    }

    return matrix[s2.length()][s1.length()];

}
```

written by ECLAT original link here