Word Pattern II

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** substring in `str`.

Examples:

1. pattern = `"abab"`, str = `"redblueredblue"` should return true.
2. pattern = `"aaaa"`, str = `"asdasdasdasd"` should return true.
3. pattern = `"aabb"`, str = `"xyzabcxzyabc"` should return false.

Notes:
You may assume both `pattern` and `str` contains only lowercase letters.

Solution 1

We can solve this problem using backtracking, we just have to keep trying to use a character in the pattern to match different length of substrings in the input string, keep trying till we go through the input string and the pattern.

For example, input string is `"redblueredblue"`, and the pattern is `"abab"`, first let's use `'a'` to match `"r"`, `'b'` to match `"e"`, then we find that `'a'` does not match `"d"`, so we do backtracking, use `'b'` to match `"ed"`, so on and so forth ...

When we do the recursion, if the pattern character exists in the hash map already, we just have to see if we can use it to match the same length of the string. For example, let's say we have the following map:

`'a': "red"`

`'b': "blue"`

now when we see `'a'` again, we know that it should match `"red"`, the length is `3`, then let's see if `str[i ... i+3]` matches `'a'`, where `i` is the current index of the input string. Thanks to StefanPochmann's suggestion, in Java we can elegantly use `str.startsWith(s, i)` to do the check.

Also thanks to i-tikhonov's suggestion, we can use a hash set to avoid duplicate matches, if character `a` matches string `"red"`, then character `b` cannot be used to match `"red"`. In my opinion though, we can say apple (pattern `'a'`) is "fruit", orange (pattern `'o'`) is "fruit", so they can match the same string, anyhow, I guess it really depends on how the problem states.

The following code should pass OJ now, if we don't need to worry about the duplicate matches, just remove the code that associates with the hash set.

```
public class Solution {

  public boolean wordPatternMatch(String pattern, String str) {
    Map<Character, String> map = new HashMap<>();
    Set<String> set = new HashSet<>();

    return isMatch(str, 0, pattern, 0, map, set);
  }

  boolean isMatch(String str, int i, String pat, int j, Map<Character, String> ma
p, Set<String> set) {
    // base case
    if (i == str.length() && j == pat.length()) return true;
    if (i == str.length() || j == pat.length()) return false;

    // get current pattern character
    char c = pat.charAt(j);

    // if the pattern character exists
    if (map.containsKey(c)) {
      String s = map.get(c);
```

```
      // then check if we can use it to match str[i...i+s.length()]
      if (!str.startsWith(s, i)) {
        return false;
      }

      // if it can match, great, continue to match the rest
      return isMatch(str, i + s.length(), pat, j + 1, map, set);
    }

    // pattern character does not exist in the map
    for (int k = i; k < str.length(); k++) {
      String p = str.substring(i, k + 1);

      if (set.contains(p)) {
        continue;
      }

      // create or update it
      map.put(c, p);
      set.add(p);

      // continue to match the rest
      if (isMatch(str, k + 1, pat, j + 1, map, set)) {
        return true;
      }

      // backtracking
      map.remove(c);
      set.remove(p);
    }

    // we've tried our best but still no luck
    return false;
  }

}
```

written by jeantimex original link here

Solution 2

```
public class Solution {
Map<Character,String> map =new HashMap();
Set<String> set =new HashSet();
public boolean wordPatternMatch(String pattern, String str) {
    if(pattern.isEmpty()) return str.isEmpty();
    if(map.containsKey(pattern.charAt(0))){
        String value= map.get(pattern.charAt(0));
        if(str.length()<value.length() || !str.substring(0,value.length()).equals
(value)) return false;
        if(wordPatternMatch(pattern.substring(1),str.substring(value.length())))
return true;
    }else{
        for(int i=1;i<=str.length();i++){
            if(set.contains(str.substring(0,i))) continue;
            map.put(pattern.charAt(0),str.substring(0,i));
            set.add(str.substring(0,i));
            if(wordPatternMatch(pattern.substring(1),str.substring(i))) return tr
ue;
            set.remove(str.substring(0,i));
            map.remove(pattern.charAt(0));
        }
    }
    return false;
}
```

}

written by DREAM123 original link here

Solution 3

```
class Solution {
public:
    unordered_map<char, string> pDict;
    unordered_map<string, char> sDict;
    bool wordPatternMatch(string pattern, string str) {
        return match(pattern, 0, str, 0);
    }

    bool match(string &pattern, int i, string &str, int j) {
        int m = pattern.size();
        int n = str.size();
        if (i == m || j == n) {
            if (i == m && j == n)
                return true;
            return false;
        }
        bool ins = false;
        for (int k = j; k < n; k++) {
            string s = str.substr(j, k - j + 1);
            if (pDict.find(pattern[i]) != pDict.end()) {
                if (pDict[pattern[i]] != s)
                    continue;
            } else if (sDict.find(s) != sDict.end()) {
                if (sDict[s] != pattern[i])
                    continue;
            } else {
                pDict[pattern[i]] = s;
                sDict[s] = pattern[i];
                ins = true;
            }
            if (match(pattern, i + 1, str, k + 1))
                return true;
            if (ins) {
                pDict.erase(pattern[i]);
                sDict.erase(s);
            }
        }
        return false;
    }
};
```

C++ backtracking. ins indicates whether current round has inserted new mapping pair. edited with two maps to ensure on-to-one mapping. .

written by lightmark original link here