Encode String with Shortest Length

Given a **non-empty** string, encode the string such that its encoded length is the shortest.

The encoding rule is: `k[encoded_string]`, where the *encoded_string* inside the square brackets is being repeated exactly *k* times.

Note:

1. *k* will be a positive integer and encoded string will not be empty or have extra space.
2. You may assume that the input string contains only lowercase English letters. The string's length is at most 160.
3. If an encoding process does not make the string shorter, then do not encode it. If there are several solutions, return any of them is fine.

Example 1:

```
Input: "aaa"
Output: "aaa"
Explanation: There is no way to encode it such that it is shorter than the input st
ring, so we do not encode it.
```

Example 2:

```
Input: "aaaaa"
Output: "5[a]"
Explanation: "5[a]" is shorter than "aaaaa" by 1 character.
```

Example 3:

```
Input: "aaaaaaaaaa"
Output: "10[a]"
Explanation: "a9[a]" or "9[a]a" are also valid solutions, both of them have the sam
e length = 5, which is the same as "10[a]".
```

Example 4:

```
Input: "aabcaabcd"
Output: "2[aabc]d"
Explanation: "aabc" occurs twice, so one answer can be "2[aabc]d".
```

Example 5:

```
Input: "abbbabbbcabbbabbbc"
Output: "2[2[abbb]c]"
Explanation: "abbbabbbc" occurs twice, but "abbbabbbc" can also be encoded to "2[ab
bb]c", so one answer can be "2[2[abbb]c]".
```

Solution 1

This is the first question I have answered in Leetcode. I hope you guys will like my solution. The approach here is simple. We will form 2-D array of Strings.
dp[i][j] = string from index i to index j in encoded form.

We can write the following formula as:-
dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j]) or if we can find some pattern in string from i to j which will result in more less length.

Time Complexity = O(n^3)

```
public class Solution {
```

```
 public String encode(String s) {
    String[][] dp = new String[s.length()][s.length()];

    for(int l=0;l<s.length();l++) {
        for(int i=0;i<s.length()-l;i++) {
            int j = i+l;
            String substr = s.substring(i, j+1);
            // Checking if string length < 5. In that case, we know that encoding
will not help.
            if(j - i < 4) {
                dp[i][j] = substr;
            } else {
                dp[i][j] = substr;
                // Loop for trying all results that we get after dividing the str
ings into 2 and combine the   results of 2 substrings
                for(int k = i; k<j;k++) {
                    if((dp[i][k] + dp[k+1][j]).length() < dp[i][j].length()){
                        dp[i][j] = dp[i][k] + dp[k+1][j];
                    }
                }

                // Loop for checking if string can itself found some pattern in i
t which could be repeated.
                for(int k=0;k<substr.length();k++) {
                    String repeatStr = substr.substring(0, k+1);
                    if(repeatStr != null
                       && substr.length()%repeatStr.length() == 0
                       && substr.replaceAll(repeatStr, "").length() == 0) {
                          String ss = substr.length()/repeatStr.length() + "[" +
dp[i][i+k] + "]";

                          if(ss.length() < dp[i][j].length()) {
                            dp[i][j] = ss;
                          }
                     }
                }
            }
        }
    }

    return dp[0][s.length()-1];
}
}
```

written by hatella original link here

Solution 2

3 for loop, so complexity O(n^3), bottom up DP with step goes from 1 to n, and for each step calculate all start and end locations. Use the collapse idea from another solution.

```
class Solution {
private:
 vector<vector<string>> dp;
public:
 string collapse(string& s, int i, int j) {
     string temp = s.substr(i, j - i + 1);
  auto pos = (temp+temp).find(temp, 1);
  if (pos >= temp.size()) {
      return temp;
  }
  return to_string(temp.size()/pos) + '['+ dp[i][i+pos-1]+']';
 }

 string encode(string s) {
  int n = s.size();
  dp = vector<vector<string>>(n, vector<string>(n, ""));
  for (int step = 1; step <= n; step++) {
   for (int i = 0; i + step - 1 < n; i++) {
    int j = i + step - 1;
    dp[i][j] = s.substr(i, step);
    for (int k = i; k < j; k++) {
     auto left = dp[i][k];
     auto right = dp[k + 1][j];
     if (left.size() + right.size() < dp[i][j].size()) {
      dp[i][j] = left + right;
     }
    }
    string replace = collapse(s, i, j);
    if (replace.size() < dp[i][j].size()) {
     dp[i][j] = replace;
    }
   }
  }
  return dp[0][n - 1];
 }
};
```

written by yanzhan2 original link here

Solution 3

Either don't encode `s` at all, or encode it as **one** part `k[...]` or encode it as **multiple** parts (in which case we can somewhere split it into two subproblems). Whatever is shortest. Uses @rsrs3's nice trick of searching `s` in `s + s`.

```
def encode(self, s, memo={}):
    if s not in memo:
        n = len(s)
        i = (s + s).find(s, 1)
        one = '%d[%s]' % (n / i, self.encode(s[:i])) if i < n else s
        multi = [self.encode(s[:i]) + self.encode(s[i:]) for i in xrange(1, n)]
        memo[s] = min([s, one] + multi, key=len)
    return memo[s]
```

written by StefanPochmann original link here