Decode Ways II

A message containing letters from `A-Z` is being encoded to numbers using the following mapping way:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Beyond that, now the encoded string can also contain the character '*', which can be treated as one of the numbers from 1 to 9.

Given the encoded message containing digits and the character '*', return the total number of ways to decode it.

Also, since the answer may be very large, you should return the output mod $10^9 + 7$.

Example 1:

```
Input: "*"
Output: 9
Explanation: The encoded message can be decoded to the string: "A", "B", "C", "D", "E
", "F", "G", "H", "I".
```

Example 2:

```
Input: "1*"
Output: 9 + 9 = 18
```

Note:

1. The length of the input string will fit in range [1, 10^5].
2. The input string will only contain the character '*' and digits '0' - '9'.

Solution 1

The idea is DP. One of the hints is that you need mod the answer with a huge prime number.

1. For any string s longer than 2, we can decode either the last 2 characters as a whole or the last 1 character. So dp[i] = dp[i-1]* f(s.substr(i,1)) + dp[i-2]* f(s.substr(i-1, 2)). f() is the number of ways to decode a string of length 1 or 2. f() could be 0, for example f("67").
2. There is a lot of cases and corner cases for f(string s). For example, * cannot be '0', so ** has 15 instead of 16 possibilities, because "20" is excluded. But the time complexity is still O(n).
   The code is as below.

```
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size(), p = 1000000007;
        // f2 is the answer to sub string ending at position i; Initially i = 0.
        long f1 = 1, f2 = helper(s.substr(0,1));
        // DP to get f2 for sub string ending at position n-1;
        for (int i = 1; i < n; i++) {
            long f3 = (f2*helper(s.substr(i, 1)))+(f1*helper(s.substr(i-1, 2)));
            f1 = f2;
            f2 = f3%p;
        }
        return f2;
    }
private:
    int helper(string s) {
        if (s.size() == 1) {
            if (s[0] == '*') return 9;
            return s[0] == '0'? 0:1;
        }
        // 11-26, except 20 because '*' is 1-9
        if (s == "**")
            return 15;
        else if (s[1] =='*') {
            if (s[0] =='1') return 9;
            return s[0] == '2'? 6:0;
        }
        else if (s[0] == '*')
            return s[1] <= '6'? 2:1;
        else
            // if two digits, it has to be in [10 26]; no leading 0
            return stoi(s) >= 10 && stoi(s) <= 26? 1:0;
    }
};
```

written by zestypanda original link here

Solution 2

I found '*' in this question can only be 1 to 9. But things may be more complicated when it can also be 0;

Input: "1*"
there should be 19 ways:
(1) from 10 to 19 : 10 ways;
(2) from A1 to A9: 9 ways;
then the total ways are 10 + 9 = 19

Input: "*"
the first " can not be set to 0, as 0 is not any character.

written by Xingxing original link here

Solution 3

The idea for DP is simple when using two helper functions
ways(i) -> that gives the number of ways of decoding a single character
and
ways(i, j) -> that gives the number of ways of decoding the two character string formed by i and j.
The actual recursion then boils down to :

```
f(i) = (ways(i) * f(i+1)) + (ways(i, i+1) * f(i+2))
```

The solution to a string f(i), where i represents the starting index,

f(i) = no.of ways to decode the character at i, which is ways(i) + solve for remainder of the string using recursion f(i+1)
and
no.of ways to decode the characters at i and i+1, which is ways(i, i+1) + solve for remainder of the string using recursion f(i+2)

The base case is ,

```
return ways(s.charAt(i)) if(i == j)
```

The above recursion when implemented with a cache, is a viable DP solution, but it leads to stack overflow error, due to the depth of the recursion. So its better to convert to memoized version.

For the memoized version, the equation changes to

```
f(i) = ( f(i-1) * ways(i) ) + ( f(i-2) *ways(i-1, i) )
```

This is exactly the same as the previous recursive version in reverse,

The solution to a string f(i), where i represents the ending index of the string,

f(i) = solution to the prefix of the string f(i-1) + no.of ways to decode the character at i, which is ways(i)
and
solution to the prefix f(i-2) + no.of ways to decode the characters at i - 1 and i, which is ways(i-1, i)

```
public class Solution {
    public static int numDecodings(String s) {
        long[] res = new long[2];
        res[0] = ways(s.charAt(0));
        if(s.length() < 2) return (int)res[0];

        res[1] = res[0] * ways(s.charAt(1)) + ways(s.charAt(0), s.charAt(1));
        for(int j = 2; j < s.length(); j++) {
            long temp = res[1];
            res[1] = (res[1] * ways(s.charAt(j)) + res[0] * ways(s.charAt(j-1), s.c
harAt(j))) % 1000000007;
            res[0] = temp;
        }
        return  (int)res[1];
    }

    private static int ways(int ch) {
        if(ch == '*') return 9;
        if(ch == '0') return 0;
        return 1;
    }

    private static int ways(char ch1, char ch2) {
        String str = "" + ch1 + "" + ch2;
        if(ch1 != '*' && ch2 != '*') {
            if(Integer.parseInt(str) >= 10 && Integer.parseInt(str) <= 26)
                return 1;
        } else if(ch1 == '*' && ch2 == '*') {
            return 15;
        } else if(ch1 == '*') {
            if(Integer.parseInt(""+ch2) >= 0 && Integer.parseInt(""+ch2) <= 6)
                return 2;
            else
                return 1;
        } else {
            if(Integer.parseInt(""+ch1) == 1 ) {
                return 9;
            } else if(Integer.parseInt(""+ch1) == 2 ) {
                return 6;
            }
        }
        return 0;
    }
}
```

written by johnyrufus16 original link here