<https://leetcode.com/problems/decode-ways/>

A message containing letters from A-Z can be **encoded** into numbers using the following mapping:

'A' -> "1"

'B' -> "2"

...

'Z' -> "26"

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

* "AAJF" with the grouping (1 1 10 6)
* "KJF" with the grouping (11 10 6)

Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

Given a string s containing only digits, return *the* ***number*** *of ways to* ***decode*** *it*.

The test cases are generated so that the answer fits in a **32-bit** integer.

**Example 1:**

Input: s = "12"

Output: 2

Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).

**Example 2:**

Input: s = "226"

Output: 3

Explanation: "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

**Example 3:**

Input: s = "06"

Output: 0

Explanation: "06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").

**Constraints:**

* 1 <= s.length <= 100
* s contains only digits and may contain leading zero(s).

**Attempt 1: 2023-09-09**

**Solution 1:  Native DFS (10 min)**

class Solution {

public int numDecodings(String s) {

Set<String> set = new HashSet<>();

for(int i = 1; i <= 26; i++) {

set.add(String.valueOf(i));

}

return helper(s, 0, set);

}

private int helper(String s, int i, Set<String> set) {

// Only when index able to go through the whole string

// means a solution build up, hence return 1

if(i == s.length()) {

return 1;

}

int result = 0;

if(i + 1 <= s.length() && set.contains(s.substring(i, i + 1))) {

result += helper(s, i + 1, set);

}

if(i + 2 <= s.length() && set.contains(s.substring(i, i + 2))) {

result += helper(s, i + 2, set);

}

return result;

}

}

**Solution 2:  DFS + Memoization (10 min)**

class Solution {

public int numDecodings(String s) {

Set<String> set = new HashSet<>();

for(int i = 1; i <= 26; i++) {

set.add(String.valueOf(i));

}

Integer[] memo = new Integer[s.length()];

return helper(s, 0, set, memo);

}

private int helper(String s, int i, Set<String> set, Integer[] memo) {

// Only when index able to go through the whole string

// means a solution build up, hence return 1

if(i == s.length()) {

return 1;

}

if(memo[i] != null) {

return memo[i];

}

int result = 0;

if(i + 1 <= s.length() && set.contains(s.substring(i, i + 1))) {

result += helper(s, i + 1, set, memo);

}

if(i + 2 <= s.length() && set.contains(s.substring(i, i + 2))) {

result += helper(s, i + 2, set, memo);

}

return memo[i] = result;

}

}

**Solution 3:  DP (30 min)**

**Style 1: Fully match the logic from recursion way ["顶" -> "底"一致]**

class Solution {

// 1. Why we have to initialize dp as (n + 1) size ?

// It depends on the problem and what the DP data structure

// is storing. A simple example of when n+1 would be necessary:

// let's say the DP array is storing computations where the index

// represents the length of a string. An array of size n will

// not have enough space for the case of an empty string (index 0),

// the case of the entire string (index n), as well as everything

// between. Size n does not allow indexing beyond n-1, while size

// n+1 allows indexing at n (which many problems call for)

// -------------------------------------------------------------------

// 2. Why we set dp[n] = 1 ?

// This is mapping to the base case from the recursive solution.

// If i == s.length(), return 1

// when we reach end, that means we were able to successfully decode

// the string by using some combination of single and double characters.

// So it resulted in 1 way of decoding.

// And physically dp[n] = 1 is not for corner case like "empty" string

// have 1 way to decode, its just set to 1 only to get the result for

// dp[n - 2]. If this confuses you, think of it this way.

// You have a string "12" , either you can decode it as '2' or '12'.

// Now if you select "12" , then dp[n - 2] += dp[n].

// If dp[n] is 0, you won't count '12' as a way to decode.

// Hence dp[n] needs to be 1.

public int numDecodings(String s) {

Set<String> set = new HashSet<>();

for(int i = 1; i <= 26; i++) {

set.add(String.valueOf(i));

}

int n = s.length();

// dp[i] means the number of ways to decode substring s[i,n)

int[] dp = new int[n + 1];

dp[n] = 1;

dp[n - 1] = set.contains(String.valueOf(s.charAt(n - 1))) ? 1 : 0;

for(int i = n - 2; i >= 0; i--) {

if(i + 1 <= n && set.contains(s.substring(i, i + 1))) {

dp[i] += dp[i + 1];

}

if(i + 2 <= n && set.contains(s.substring(i, i + 2))) {

dp[i] += dp[i + 2];

}

}

return dp[0];

}

}

Time Complexity: should be O(n), where n is the length of String

Space Complexity: should be O(n), where n is the length of String

**Refer to**

<https://leetcode.com/problems/decode-ways/solutions/30358/java-clean-dp-solution-with-explanation/>

I used a dp array of size n + 1 to save subproblem solutions. dp[0] means an empty string will have one way to decode, dp[1] means the way to decode a string of size 1. I then check one digit and two digit combination and save the results along the way. In the end, dp[n] will be the end result.

public class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0) {

return 0;

}

int n = s.length();

int[] dp = new int[n + 1];

dp[0] = 1;

dp[1] = s.charAt(0) != '0' ? 1 : 0;

for (int i = 2; i <= n; i++) {

int first = Integer.valueOf(s.substring(i - 1, i));

int second = Integer.valueOf(s.substring(i - 2, i));

if (first >= 1 && first <= 9) {

dp[i] += dp[i-1];

}

if (second >= 10 && second <= 26) {

dp[i] += dp[i-2];

}

}

return dp[n];

}

}

**I wrote some notes for myself reference, hope it might help someone to understand this solution.**

<https://leetcode.com/problems/decode-ways/solutions/30358/java-clean-dp-solution-with-explanation/comments/29468>

dp[i]: represents possible decode ways to the ith char(include i), whose index in string is i-1

Base case: dp[0] = 1 is just for creating base; dp[1], when there's one character, if it is not zero, it can only be 1 decode way. If it is 0, there will be no decode ways.

Here only need to look at at most two digits before i, cuz biggest valid code is 26, which has two digits.

For dp[i]: to avoid index out of boundry, extract substring of (i-1,i)- which is the ith char(index in String is i-1) and substring(i-2, i)

First check if substring (i-1,i) is 0 or not. If it is 0, skip it, continue right to check substring (i-2,i), cuz 0 can only be decode by being together with the char before 0.

Second, check if substring (i-2,i) falls in 10~26. If it does, means there are dp[i-2] more new decode ways.

Time: should be O(n), where n is the length of String

Space: should be O(n), where n is the length of String

**If dp[0] means an empty string will have only one way to decode, then when we check the corner case (s == null || s.length() == 0) why do we return 0 ? shouldn't we return 1 ? ? Can someone explain that**

<https://leetcode.com/problems/decode-ways/solutions/30358/java-clean-dp-solution-with-explanation/comments/669480>

**There are 0 ways to decode a empty string. dp[0] is set to 1 only to get the result for dp[2].**

**If this confuses you, think of it this way. You have a string "12" , either you can decode it as '2' or '12'.Now if you select "12" , then dp[2] += dp[0].**

**If dp[0] is 0, you wont count '12' as a way to decode. Hence dp[0] needs to be 1.**

**Style 2: To avoid the confusion of dp[0] = 1, no dp[n + 1], just dp[n] (cannot repeat with same pattern)**

**Refer to**

<https://leetcode.com/problems/decode-ways/solutions/30358/java-clean-dp-solution-with-explanation/comments/207715>

public int numDecodings(String s) {

if(s == null || s.length() == 0) {

return 0;

}

int n = s.length();

int[] dp = new int[n];

dp[0] = s.charAt(0) != '0' ? 1 : 0;

for(int i = 1; i < n; i++) {

int first = Integer.valueOf(s.substring(i, i+1));

int second = Integer.valueOf(s.substring(i-1, i+1));

if(first >= 1 && first <= 9) {

dp[i] += dp[i-1];

}

if(second >= 10 && second <= 26) {

dp[i] += i >=2 ? dp[i-2] : 1;

}

}

return dp[n-1];

}

**Refer to**

<https://leetcode.wang/leetcode-91-Decode-Ways.html>

# **解法一 递归**

很容易想到递归去解决，将大问题化作小问题。

比如 232232323232。

对于第一个字母我们有两种划分方式。

2|**32232323232** 和 23|**2232323232**

所以，如果我们分别知道了上边划分的右半部分 32232323232 的解码方式是 ans1 种，2232323232 的解码方式是 ans2 种，那么整体 232232323232 的解码方式就是 ans1 + ans2 种。可能一下子，有些反应不过来，可以看一下下边的类比。

假如从深圳到北京可以经过**武汉**和**上海**两条路，而从**武汉**到北京有 8 条路，从**上海**到北京有 6 条路。那么从深圳到北京就有 8 + 6 = 14 条路。

public int numDecodings(String s) {

return getAns(s, 0);

}

private int getAns(String s, int start) {

//划分到了最后返回 1

if (start == s.length()) {

return 1;

}

//开头是 0,0 不对应任何字母，直接返回 0

if (s.charAt(start) == '0') {

return 0;

}

//得到第一种的划分的解码方式

int ans1 = getAns(s, start + 1);

int ans2 = 0;

//判断前两个数字是不是小于等于 26 的

if (start < s.length() - 1) {

int ten = (s.charAt(start) - '0') \* 10;

int one = s.charAt(start + 1) - '0';

if (ten + one <= 26) {

ans2 = getAns(s, start + 2);

}

}

return ans1 + ans2;

}

# **解法二 递归 memoization**

解法一的递归中，走完左子树，再走右子树会把一些已经算过的结果重新算，所以我们可以用 memoization 技术，就是算出一个结果很就保存，第二次算这个的时候直接拿出来就可以了。

public int numDecodings(String s) {

HashMap<Integer, Integer> memoization = new HashMap<>();

return getAns(s, 0, memoization);

}

private int getAns(String s, int start, HashMap<Integer, Integer> memoization) {

if (start == s.length()) {

return 1;

}

if (s.charAt(start) == '0') {

return 0;

}

//判断之前是否计算过

int m = memoization.getOrDefault(start, -1);

if (m != -1) {

return m;

}

int ans1 = getAns(s, start + 1, memoization);

int ans2 = 0;

if (start < s.length() - 1) {

int ten = (s.charAt(start) - '0') \* 10;

int one = s.charAt(start + 1) - '0';

if (ten + one <= 26) {

ans2 = getAns(s, start + 2, memoization);

}

}

//将结果保存

memoization.put(start, ans1 + ans2);

return ans1 + ans2;

}

# **解法三 动态规划**

同样的，递归就是压栈压栈压栈，出栈出栈出栈的过程，我们可以利用动态规划的思想，省略压栈的过程，直接从 bottom 到 top。

用一个 dp 数组， dp [ i ] 代表字符串 s [ i, s.len-1 ]，也就是 s 从 i 开始到结尾的字符串的解码方式。

这样和递归完全一样的递推式。

如果 s [ i ] 和 s [ i + 1 ] 组成的数字小于等于 26，那么

dp [ i ] = dp[ i + 1 ] + dp [ i + 2 ]

public int numDecodings(String s) {

int len = s.length();

int[] dp = new int[len + 1];

dp[len] = 1; //将递归法的结束条件初始化为 1

//最后一个数字不等于 0 就初始化为 1

if (s.charAt(len - 1) != '0') {

dp[len - 1] = 1;

}

for (int i = len - 2; i >= 0; i--) {

//当前数字时 0 ，直接跳过，0 不代表任何字母

if (s.charAt(i) == '0') {

continue;

}

int ans1 = dp[i + 1];

//判断两个字母组成的数字是否小于等于 26

int ans2 = 0;

int ten = (s.charAt(i) - '0') \* 10;

int one = s.charAt(i + 1) - '0';

if (ten + one <= 26) {

ans2 = dp[i + 2];

}

dp[i] = ans1 + ans2;

}

return dp[0];

}

接下来就是，动态规划的空间优化了，例如[5题](https://leetcode.windliang.cc/leetCode-5-Longest-Palindromic-Substring.html)，[10题](https://leetcode.windliang.cc/leetCode-10-Regular-Expression-Matching.html)，[53题](https://leetcode.windliang.cc/leetCode-53-Maximum-Subarray.html?h=%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92)，[72题](https://leetcode.wang/leetCode-72-Edit-Distance.html)等等都是同样的思路。都是注意到一个特点，当更新到 dp [ i ] 的时候，我们只用到 dp [ i + 1] 和 dp [ i + 2]，之后的数据就没有用了。所以我们不需要 dp 开 len + 1 的空间。

简单的做法，我们只申请 3 个空间，然后把 dp 的下标对 3 求余就够了。

public int numDecodings4(String s) {

int len = s.length();

int[] dp = new int[3];

dp[len % 3] = 1;

if (s.charAt(len - 1) != '0') {

dp[(len - 1) % 3] = 1;

}

for (int i = len - 2; i >= 0; i--) {

if (s.charAt(i) == '0') {

dp[i % 3] = 0; //这里很重要，因为空间复用了，不要忘记归零

continue;

}

int ans1 = dp[(i + 1) % 3];

int ans2 = 0;

int ten = (s.charAt(i) - '0') \* 10;

int one = s.charAt(i + 1) - '0';

if (ten + one <= 26) {

ans2 = dp[(i + 2) % 3];

}

dp[i % 3] = ans1 + ans2;

}

return dp[0];

}

然后，如果多考虑以下，我们其实并不需要 3 个空间，我们只需要 2 个就够了，只需要更新的时候，指针移动一下，代码如下。

public int numDecodings5(String s) {

int len = s.length();

int end = 1;

int cur = 0;

if (s.charAt(len - 1) != '0') {

cur = 1;

}

for (int i = len - 2; i >= 0; i--) {

if (s.charAt(i) == '0') {

end = cur;//end 前移

cur = 0;

continue;

}

int ans1 = cur;

int ans2 = 0;

int ten = (s.charAt(i) - '0') \* 10;

int one = s.charAt(i + 1) - '0';

if (ten + one <= 26) {

ans2 = end;

}

end = cur; //end 前移

cur = ans1 + ans2;

}

return cur;

}

# **总**

从递归，到动态规划，到动态规划的空间复杂度优化，已经很多这样的题了，很经典。