Decode Ways

Solution

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).

C++

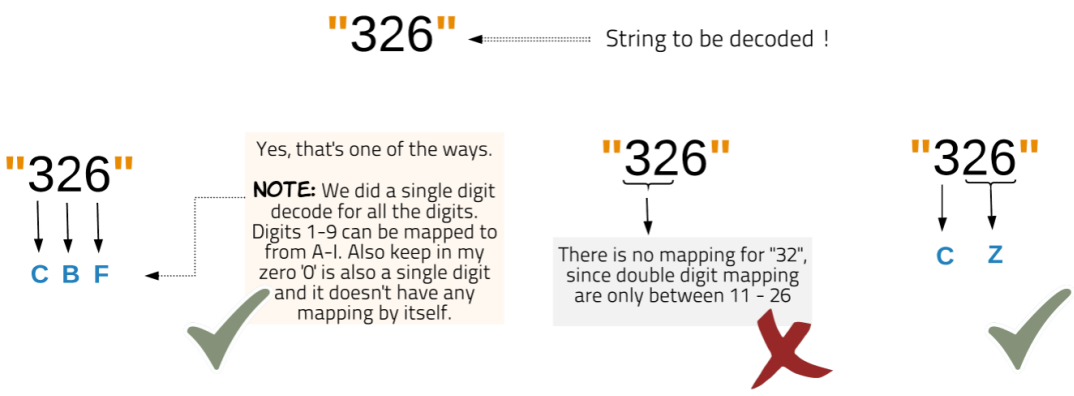
```
class Solution {
public:
    int numDecodings(string s) {

    }
};
```

Video link:

Solution Article

The most important point to understand in this problem is that at any given step when you are trying to decode a string of numbers it can either be a single digit decode e.g. `1` to `A` or a double digit decode e.g. `25` to `Y`. As long as it's a valid decoding we move ahead to decode the rest of the string.

The subproblem could be thought of as number of ways decoding a substring.

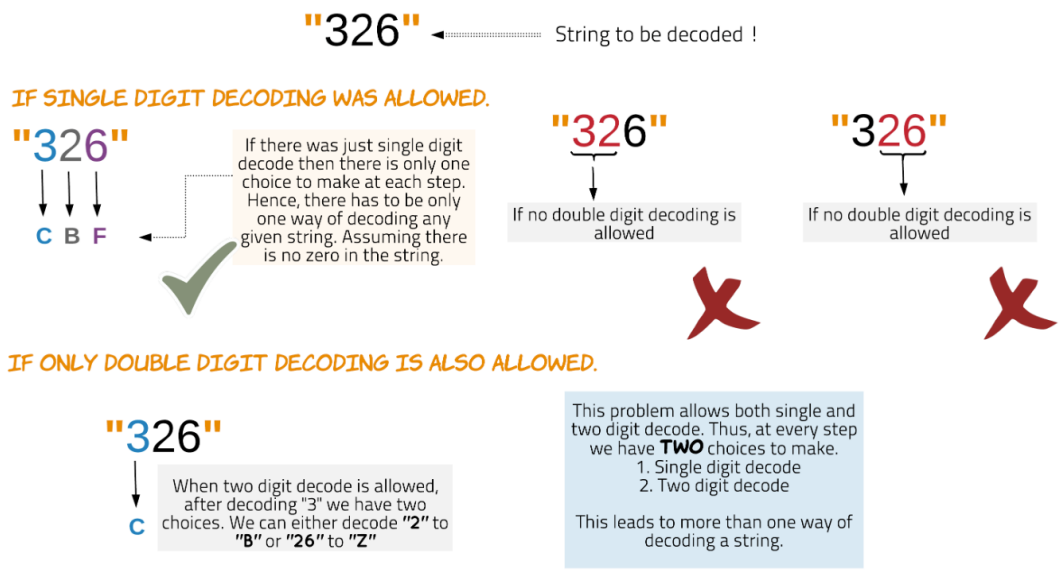


The above diagram shows string "326" could be decoded in two ways.

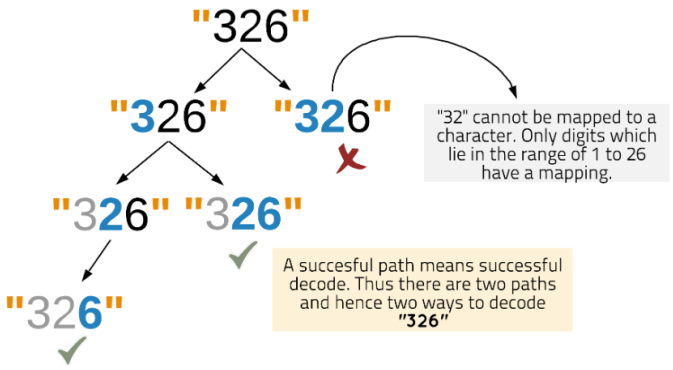
Approach 1: Recursive Approach with Memoization

Intuition

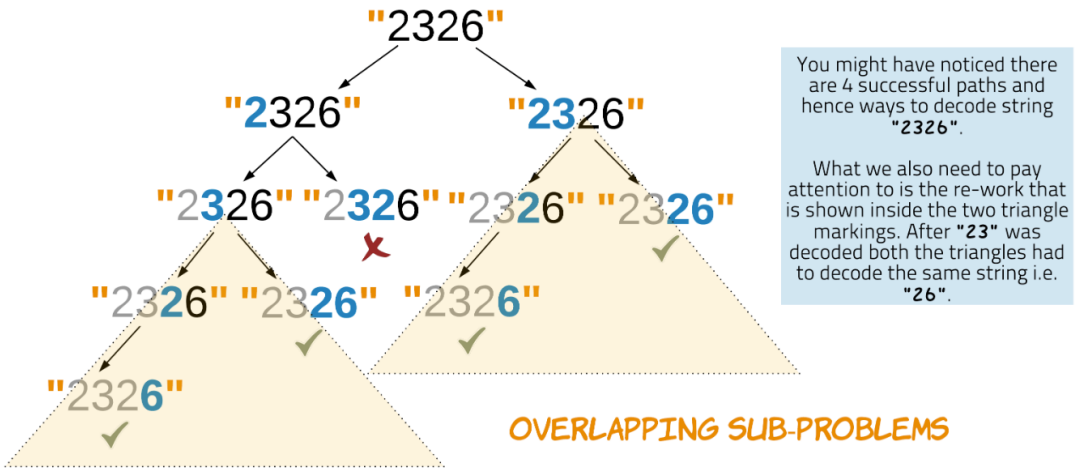
The problem deals with finding number of ways of decoding a string. What helps to crack the problem is to think why there would be many ways to decode a string. The reason is simple since at any given point we either decode using `two digits` or `single` digit. This choice while decoding can lead to different combinations.



Thus at any given time for a string we enter a recursion after successfully decoding two digits to a single character or a single digit to a character. This leads to multiple paths to decoding the entire string. If a given path leads to the end of the string this means we could successfully decode the string. If at any point in the traversal we encounter digits which cannot be decoded, we backtrack from that path.



In the following diagram we can see how the paths have to deal with similar subproblems. Overlapping subproblems means we can reuse the answer. Thus, we do memoization to solve this problem.



Algorithm

1. Enter recursion with the given string i.e. start with index 0.
2. For the terminating case of the recursion we check for the end of the string. If we have reached the end of the string we return `1`.
3. Every time we enter recursion it's for a substring of the original string. For any recursion if the first character is `0` then terminate that path by returning `0`. Thus this path won't contribute to the number of ways.
4. Memoization helps to reduce the complexity which would otherwise be exponential. We check the dictionary `memo` to see if the result for the given substring already exists.
5. If the result is already in `memo` we return the result. Otherwise the number of ways for the given string is determined by making a recursive call to the function with `index + 1` for next substring string and `index + 2` after checking for valid 2-digit decode. The result is also stored in `memo` with key as current index, for saving for future overlapping subproblems.

C++

```
class Solution {
public:
    map<int, int> memo;

    int recursiveWithMemo(int index, string& str) {
        // Have we already seen this substring?
        if (memo.find(index) != memo.end()) {
```

```
            return memo[index];
        }

        // If you reach the end of the string
        // Return 1 for success.
        if (index == str.length()) {
            return 1;
        }

        // If the string starts with a zero, it can't be decoded
        if (str[index] == '0') {
            return 0;
        }

        if (index == str.length() - 1) {
            return 1;
        }

        int ans = recursiveWithMemo(index + 1, str);
        if (stoi(str.substr(index, 2)) <= 26) {
            ans += recursiveWithMemo(index + 2, str);
        }

        // Save for memoization
        memo[index] = ans;

        return ans;
    }

    int numDecodings(string s) {
        return recursiveWithMemo(0, s);
    }
};
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is length of the string. Memoization helps in pruning the recursion tree and hence decoding for an index only once. Thus this solution is linear time complexity.
- Space Complexity: $O(N)$. The dictionary used for memoization would take the space equal to the length of the string. There would be an entry for each index value. The recursion stack would also be equal to the length of the string.

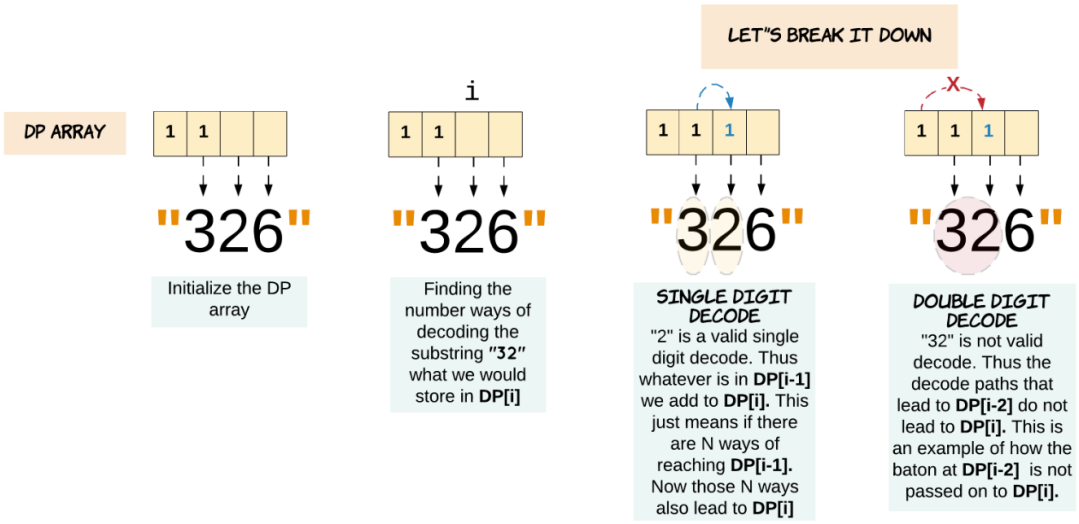
Approach 2: Iterative Approach

The iterative approach might be a little bit less intuitive. Let's try to understand it. We use an array for DP to store the results for subproblems. A cell with index `i` of the `dp` array is used to store the number of decode ways for substring of `s` from index `0` to index `i-1`.

We initialize the starting two indices of the `dp` array. It's similar to relay race where the first runner is given a `baton` to be passed to the subsequent runners. The first two indices of the `dp` array hold a baton. As we iterate the `dp` array from left to right this baton which signifies the number of ways of decoding is passed to the next index or not depending on whether the decode is possible.

`dp[i]` can get the baton from two other previous indices, either `i-1` or `i-2`. Two previous indices are involved since both single and two digit decodes are possible.

Unlike the relay race we don't get only one baton in the end. The batons add up as we pass on. If someone has one baton, they can provide a copy of it to everyone who comes to them with a success. Thus, leading to number of ways of reaching the end.



`dp[i]` = Number of ways of decoding substring `s[:i]`. So we might say `dp[i] = dp[i-1] + dp[i-2]`, which is not always true for this decode ways problem. As shown in the above diagram, only when the decode is possible we add the results of the previous indices. Thus, in this race we don't just pass the baton. The baton is passed to the next index or not depending on possibility of the decode.

Algorithm

1. If the string `s` is empty or null we return the result as `0`.
2. Initialize `dp` array. dp[0] = 1 to provide the baton to be passed.
3. If the first character of the string is zero then no decode is possible hence initialize dp[1] to `0`, otherwise the first character is valid to pass on the baton, dp[1] = 1.
4. Iterate the `dp` array starting at index 2. The index `i` of dp is the i-th character of the string `s`, that is character at index `i-1` of s.
5. We check if valid single digit decode is possible. This just means the character at index `s[i-1]` is non-zero. Since we do not have a decoding for zero. If the valid single digit decoding is possible then we add `dp[i-1]` to `dp[i]`. Since all the ways up to (i-1)-th character now lead up to i-th character too.
6. We check if valid two digit decode is possible. This means the substring `s[i-2]s[i-1]` is between 10 to 26. If the valid two digit decoding is possible then we add `dp[i-2]` to `dp[i]`.
7. Once we reach the end of the `dp` array we would have the number of ways of decoding string `s`.

C++

```
class Solution {
public:
    int numDecodings(string s) {
        // DP array to store the subproblem results
        vector<int> dp(s.length() + 1);
        dp[0] = 1;

        // Ways to decode a string of size 1 is 1. Unless the string is '0'.
        // '0' doesn't have a single digit decode.
        dp[1] = s[0] == '0' ? 0 : 1;

        for (size_t i = 2; i < dp.size(); i++) {
            // Check if successful single digit decode is possible.
            if (s[i - 1] != '0') {
                dp[i] = dp[i - 1];
            }

            // Check if successful two digit decode is possible.
            int two_digit = stoi(s.substr(i - 2, 2));
            if (two_digit >= 10 && two_digit <= 26) {
                dp[i] += dp[i - 2];
            }
        }
        return dp[s.length()];
    }
};
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is length of the string. We iterate the length of `dp` array which is $N + 1$.
- Space Complexity: $O(N)$. The length of the DP array.

Approach 3: Iterative, Constant Space

Intuition

In Approach 2 we are using an array `dp` to save the results for future. As we move ahead character by character of the given string, we look back only two steps. For calculating `dp[i]` we need to know `dp[i-1]` and `dp[i-2]` only. Thus, we can easily cut down our $O(N)$ space requirement to $O(1)$ by using only two variables to store the last two results.

C++

```
class Solution {
public:
    int numDecodings(string s) {
        if (s[0] == '0') {
            return 0;
        }

        size_t n = s.length();
        int two_back = 1;
        int one_back = 1;

        for (size_t i = 1; i < n; i++) {
            int current = 0;
            if (s[i] != '0') {
                current = one_back;
            }
            int two_digit = stoi(s.substr(i - 1, 2));
            if (two_digit >= 10 and two_digit <= 26) {
                current += two_back;
            }

            two_back = one_back;
            one_back = current;
        }
        return one_back;
    }
};
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is length of the string. We're essentially doing the same work as what we were in Approach 2, except this time we're throwing away calculation results when we no longer need them.
- Space Complexity: $O(1)$. Instead of a `dp` array, we're simply using two variables.