DECODE WAYS

A PROJECT REPORT

Submitted by

KHUSI TIWARI [RA2111003011709]
VIBHU KAUSHIK [RA2111003011711]

for the course 18CSC204J Design and Analysis of Algorithms

Under the Guidance of

Dr. Sindhuja M

Assistant Professor, Department of Computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603203

APRIL 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that the 18CSC204J Design and Analysis of Algorithms course project report titled **"DECODE WAYS"** is the bonafide work done by **KHUSHI TIWARI [RA2111003011709] & VIBHU KAUSHIK [RA2111003011711]** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE
Faculty In-Charge
Dr. Sindhuja M,
Assistant Professor,
Department of Computing Technologies,
SRMIST.

HEAD OF THE DEPARTMENT
Dr. M. Pushpalatha,
Professor and Head,
Department of Computing Technologies,
SRMIST.

INDEX

Problem statement:

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.

Constraints:

1 <= s.length <= 100

s contains only digits and may contain leading zero(s).

SR NO.	INPUTS	OUTPUTS	EXPLANATION
1.	s = "12"	2	"12" could be decoded as "AB" (1 2) or "L" (12).
2.	s = "226"	3	"226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
3.	s = "06"	0	"06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").

Problem statement explanation:

The problem is to decode an encoded message represented as a string of digits, where each digit can be mapped to a letter from A to Z using a given mapping. The mapping is such that 'A' is represented by '1', 'B' is represented by '2', and so on, up to 'Z' being represented by '26'. For example, the message "123" can be decoded into the letters "ABC" by mapping each digit to its corresponding letter.

However, there can be multiple ways to decode a given message because some digits can be mapped to a single letter (like '10' being mapped to 'J'), and some digits can be mapped to two letters (like '26' being mapped to both 'Z' and 'B'). To account for this, we must consider all possible groupings of digits and their corresponding letter mappings.

The problem asks us to count the number of ways to decode the given message, where a way of decoding is defined as a valid grouping of digits that can be mapped to letters using the given mapping. For example, "11106" can be decoded in two ways: "AAJF" and "KJF", but "1 11 06" is not a valid grouping since '06' cannot be mapped to a letter.

It is important to note that the answer should fit in a 32-bit integer, which means that the total number of ways to decode the message should not exceed 2^31 - 1.

Design Paradigms/techniques:

1. **Greedy Approach:** The greedy approach aims to find the optimal solution to a problem by making locally optimal choices at each step. It chooses the best possible option at each step and hopes to reach the global optimum. However, this approach may not always lead to the optimal solution for all problems.

2. **Backtracking:** Backtracking is a general algorithmic technique that involves trying out all possible solutions to a problem by making choices and undoing them when they lead to a dead end. It is used to solve problems where the solution space is too large to be explored exhaustively by brute force. The backtracking algorithm prunes the search space by eliminating the choices that cannot lead to a valid solution.

3. **Dynamic Programming:** Dynamic programming is an algorithmic technique used to solve problems that can be broken down into overlapping subproblems. It solves each subproblem only once and stores its result for future use. This approach helps to reduce the computational complexity of the problem by avoiding redundant computations.

Greedy Approach

Algorithm:

1. If the input string is empty or starts with '0', return 0 as it cannot be decoded.
2. Initialize a counter variable to 1 to account for the base case where the entire string can be decoded using a single digit.
3. Iterate through each digit in the string from index 1 to n-1.
4. If the current digit is '0', check if the previous digit is '1' or '2'. If not, return 0 as '0' cannot be decoded by itself. Decrement the counter by 1 to exclude the previous digit, and add 1 to the counter if the current digit can be combined with the previous digit to form a valid code.
5. If the previous digit is '1' or '2', check if the current digit can be combined with the previous digit to form a valid code. If so, add 1 to the counter.
6. Return the final value of the counter.

Pseudocode:

```
def numDecodingsGreedy(s: str) -> int:
    if not s or s[0] == '0':
        return 0

    n = len(s)
    count = 1
    i = 1
    while i < n:
        if s[i] == '0':
            if s[i-1] not in ['1', '2']:
                return 0
```

```
            count -= 1
            if i >= 2:
                count += 1
        elif s[i-1] == '1' or (s[i-1] == '2' and s[i] in ['1', '2', '3']):
            if i >= 2:
                count += 1
        i += 1

    return count
```

Analysis of code implementation:

1. The function `numDecodingsGreedy` takes a string `s` as input and returns an integer representing the number of ways to decode the string.

2. The function first checks if the input string is empty or if its first character is '0'. If either condition is true, it returns 0 as there is no way to decode an empty string or a string that starts with '0'.

3. The function then initializes a counter variable `count` to 1 and an index `i` to 1.

4. The while loop iterates through each character in the string starting from the second character (i.e., index 1). At each iteration, it checks if the current character is '0'. If it is, it checks if the previous character is either '1' or '2'. If it is not, it returns 0 as it is not possible to decode the string with the given conditions.

5. If the previous character is '1' or '2', it decrements the counter `count` by 1 and checks if the index is greater than or equal to 2. If it is, it increments `count` by 1.

6. If the current character is not '0', the function checks if the previous character is '1' or '2' and the current character is between '1' and '3' (inclusive). If it is, it increments `count` by 1.

7. Finally, the function returns the counter `count`, which represents the number of ways to decode the input string.

8. The time complexity of this algorithm is O(n), where n is the length of the input string, as it performs a single pass through the string. The space complexity is O(1), as it only uses a constant amount of extra space to store the counter `count` and the index `i`.

Time Complexity Analysis:

The time complexity of the greedy approach is O(n), where n is the length of the input string.

To prove this using the substitution method, we assume that the time complexity of the greedy algorithm for an input of size n is T(n).

At each iteration of the algorithm, we perform constant time operations such as checking the value of the current and previous digits, incrementing or decrementing the counter variable, and updating the index variable.

Therefore, we can express the time complexity of the greedy algorithm as follows:

$T(n) = O(1) + T(n-1)$

Solving this recurrence relation, we get:

$T(n) = O(1) + T(n-1)$
$= O(1) + O(1) + T(n-2)$
$= O(1) + O(1) + O(1) + T(n-3)$
$= ...$
$= O(1) * n$
$= O(n)$
Therefore, the time complexity of the greedy approach is O(n).

Backtracking

Algorithm:

1. Define a helper function backtrack(start, end) that takes the starting index and ending index of a substring as input and returns the number of ways to decode that substring.
2. If the starting index equals the ending index, return 1 as there is only one way to decode a single digit.
3. If the first digit is '0', return 0 as it cannot be decoded by itself.
4. If the substring contains only two digits, return 1 if the two digits can form a valid code, else return 0.
5. Recursively call the helper function for the substring starting from the next digit and add the result to a counter variable.
6. If the first two digits can form a valid code, recursively call the helper function for the substring starting from the second digit and add the result to the counter variable.
7. Return the final value of the counter.

Pseudo Code:

```
def numDecodingsBacktrack(s: str) -> int:
    def backtrack(start: int, end: int) -> int:
        if start == end:
            return 1
        if s[start] == '0':
            return 0
        if start == end - 1:
            return 1

        count = backtrack(start+1, end)
```

```
    if int(s[start:start+2]) <= 26:
        count += backtrack(start+2, end)

    return count

return backtrack(0, len(s))
```

Analysis of code implementation:

1. The code uses the backtracking approach to solve the problem. The main function `numDecodingsBacktrack` takes a string `s` as input and returns the number of ways to decode the string using the given mapping.

2. The function defines a nested function `backtrack` that takes two arguments `start` and `end`. The `start` argument is the index of the current digit being processed, and `end` is the index of the last digit in the string. The function returns the number of ways to decode the substring starting from index `start` and ending at index `end`.

3. The base cases of the recursion are:
    - If `start` and `end` are equal, the function returns 1 as there is only one way to decode a single digit.
    - If the first digit in the substring is '0', there are no possible ways to decode it, so the function returns 0.
    - If the substring has only two digits, the function returns 1 if the substring can be decoded to a valid letter (i.e., the integer value of the substring is less than or equal to 26), otherwise it returns 0.

4. For other cases, the function calculates the number of ways to decode the substring recursively by considering two possibilities:

- Decode the current digit as a single letter, and recursively decode the remaining substring (i.e., call the `backtrack` function with `start+1` and `end`).
- Decode the current digit and the next digit as a single letter, and recursively decode the remaining substring (i.e., call the `backtrack` function with `start+2` and `end`).

5. The function returns the sum of the above two possibilities as the total number of ways to decode the substring.

6. The main function `numDecodingsBacktrack` calls the `backtrack` function with start index 0 and end index `len(s)`. It returns the final count of ways to decode the entire string.

Time Complexity Analysis:

The time complexity of the backtracking approach is O(2^n), where n is the length of the input string.

To prove this using the substitution method, we assume that the time complexity of the backtracking algorithm for an input of size n is T(n).

At each recursive call of the algorithm, we generate two new recursive calls for substrings starting from the next digit and the second digit respectively. Therefore, the number of recursive calls doubles at each level of the recursion tree.

Therefore, we can express the time complexity of the backtracking algorithm as follows:

$T(n) = 2 * T(n-1)$

Solving this recurrence relation, we get:

$T(n) = 2 * T(n-1)$
$= 2 * (2 * T(n-2))$
$= 2^2 * T(n-2)$
$= 2^3 * T(n-3)$
$= ...$
$= 2^n * T(0)$

Since T(0) is a constant, we can express the time complexity of the backtracking algorithm as follows:

$T(n) = O(2^n)$

Therefore, the time complexity of the backtracking approach is $O(2^n)$.

Dynamic Programming

Algorithm:

1. If the input string is empty or starts with '0', return 0 as it cannot be decoded.
2. Initialize an array of size n+1 to store the number of ways to decode each substring of the input string.
3. Set the base cases dp[0] = 1 and dp[1] = 1 if the first digit is not '0', else dp[1] = 0.
4. Iterate through the input string from index 2 to n.
5. If the current digit is not '0', add dp[i-1] to the number of ways to decode the current substring.
6. If the previous two digits form a valid code, add dp[i-2] to the number of ways to decode the current substring.
7. Return dp[n], which represents the total number of ways to decode the entire input string.

Pseudo code:

```
def numDecodingsDP(s: str) -> int:
    if not s or s[0] == '0':
        return 0

    n = len(s)
    dp = [0] * (n + 1)
    dp[0] = 1
    dp[1] = 1 if s[0] != '0' else 0

    for i in range(2, n+1):
        if s[i-1] != '0':
```

```
            dp[i] += dp[i-1]
        if s[i-2:i] >= '10' and s[i-2:i] <= '26':
            dp[i] += dp[i-2]

    return dp[n]
```

Analysis of code implementation:

1. The function first checks if the input string is empty or starts with 0, and returns 0 in those cases since it's impossible to decode the message.

2. It then initializes a list `dp` with length `n+1` to store the number of ways to decode a prefix of the string of length i. It sets the first two elements of the list based on whether the first character of the input string is 0 or not.

3. The function then iterates over the remaining characters in the input string and updates the `dp` list based on whether the current character can be decoded on its own or with the previous character. The value in `dp[i]` is the sum of the value in `dp[i-1]` and `dp[i-2]` if the current two characters can be decoded together.

4. The function returns the last element in the `dp` list, which represents the total number of ways to decode the input string.

5. Overall, this algorithm has a time complexity of O(n) and a space complexity of O(n).

Time Complexity Analysis:

The time complexity of the dynamic programming approach is O(n), where n is the length of the input string.

To prove this using the substitution method, we assume that the time complexity of the dynamic programming algorithm for an input of size n is T(n).

At each iteration of the algorithm, we perform constant time operations such as checking the value of the current digit, updating the values in the dp array, and updating the index variable.

Therefore, we can express the time complexity of the dynamic programming algorithm as follows:

$T(n) = O(1) + T(n-1)$

Solving this recurrence relation, we get:

$T(n) = O(1) + T(n-1)$
$= O(1) + O(1) + T(n-2)$
$= O(1) + O(1) + O(1) + T(n-3)$
$= ...$
$= O(1) * n$
$= O(n)$

Therefore, the time complexity of the dynamic programming approach is O(n).

Conclusion

The backtracking approach has an exponential time complexity of O(2^n), which quickly becomes impractical for larger input sizes.

The dynamic programming and greedy approaches, on the other hand, have linear time complexity of O(n), which makes them more efficient for larger input sizes. The dynamic programming approach has a slightly higher constant factor due to the overhead of initializing and updating the DP table, but the difference becomes negligible as the input size grows.

Overall, the dynamic programming approach is the most efficient of the three for this particular problem.

References:

1. https://leetcode.com/
2. https://www.geeksforgeeks.org/
3. https://hackerrank.com/
4. https://www.codechef.com/