1416. Restore The Array

Problem Explanation

Suppose, we have a string containing digits and an integer `k`. We can restore the string to an array of integers, each being between `[1, k]` inclusive. There are some conditions:

1. All integers in the array are between `[1, k]` inclusive.
2. The array elements do not consist of leading zeros.
3. The sequence of array elements form the original string after stripping off the whitespaces.

Given these constraints, we have to find the number of various arrays that we can form and return that number `%` `10^9+7`.

Let's walk through an example:

```
plaintext
Input: s = "1317", k = 2000
Output: 8
Explanation: Possible arrays are [1317],[131,7],[13,17],[1,317],[13,1,7],[1,31,7],[1,3,17],[1,3,1,7].
```

Here, the string is "1317" and `k` is 2000. Since the digits' sequence does not contain leading zeros and each possible integer formed from the sequence is between `[1, k]` inclusive, we can form 8 different arrays that satisfy the above conditions.

Approach

This problem uses a form of dynamic programming to solve it. The algorithm works with the idea of a `dp` array used to keep the different ways to restore parts of the original string, calling the function recursively to iterate through the string and break it down into possible integers within the range of `1` and `k`.

If there is an element in the `dp` array for the current index `i`, we skip to the next index. Otherwise, concatenating the current digit with the previously formed number if it is not greater than `k` and recursively call the function for subsequent digits. Finally, we store the results in the `dp` array to avoid processing the same data more than once.

It's essential since the sum might be very great, we will return the sum modulo `10^9 + 7`, a common way in algorithmic questions to handle scenarios where the result could be huge.

Solution in Python

```
python
mod = 10 ** 9 + 7

class Solution:
    def numberOfArrays(self, s: str, k: int) -> int:
        dp = [0] * (len(s) + 1)
        dp[-1] = 1

        for i in range(len(s) - 1, -1, -1):
            dp[i], curr = 0, 0
            for j in range(i, len(s)):
                if curr == 0 and s[i] == '0':  # Leading zeros is not allowed
                    break
                curr = 10 * curr + int(s[j])
                if curr > k:  # If it exceeds, end the inner loop
                    break
                dp[i] += dp[j + 1]
                dp[i] %= mod

        return dp[0]
```

In the Python solution, we are looping from the end of the string, and when we find a number `curr` that can be used, we add `dp[j+1]` to `dp[i]`. And finally, return the first value of `dp` which contains the result. The `if curr == 0 and s[i] == '0': break` condition is for the cases where there's a leading zero. We break the for loop when we encounter a leading zero.

Solution in Java

```
java
public int numberOfArrays(String s, int k) {
    int n = s.length(), mod = (int)1e9+7;
    int[] dp = new int[n + 1];
    dp[n] = 1;  // base case
    for (int i = n - 1; i >= 0; --i) {
        long num = 0;
        for (int j = i; j < n; ++j) {
            if (j > i && s.charAt(i) == '0' || num > k) break;
            num = num * 10 + s.charAt(j) - '0';
            dp[i] = (dp[i] + dp[j + 1])%mod;
        }
    }
    return dp[0];
}
```

In the Java solution, we follow the same approach, but we have to take care of converting the character into an integer using `s.charAt(j) - '0'`.

Solution in JavaScript

```
javascript
const numberOfArrays = (s, k, pre = s.length, cache = {}) => {
    if (pre in cache) {
        return cache[pre];
    }
    if (pre === 0) {
        return 1;
    }
    let count = 0;
    for (let len = 1; pre - len >= 0; ++len) {
        const num = Number(s.slice(pre - len, pre));
        if (num > k) {
            break;
        }
        if (num.toString().length !== len) {
            continue;
        }
        count = (count + numberOfArrays(s, k, pre - len, cache)) % (1e9 + 7);
    }
    cache[pre] = count;
    return count;
};
```

In JavaScript, we convert the substring into an integer using `Number()`, continuing if a leading zero is present, increment count recursively, and store cache results for pruning.

Solution in C++

```
C++
const int MOD = 1e9+7;
class Solution {
public:
    int numberOfArrays(string s, int k) {
        int dp[100001] = {0};
        int n= s.size();
        dp[n] = 1;
        for(int i = n-1; i >= 0; --i){
            if(s[i] == '0') continue;
            long long num = 0;
            for(int j = i; j < n; ++j){
                num = num*10 + (s[j] - '0');
                if(num > k) break;
                dp[i] = (dp[i] + dp[j+1]) % MOD;
            }
        }
        return dp[0];
    }
};
```

In this C++ solution, we calculate the number by multiplying the prior number by 10 and add the current digit. If the number exceeds `k`, we stop processing.

Solution in C#

```
C#
public class Solution {
    int[] dp;
    int mod = 1000000000+7, k;
    string s;
    public int NumberOfArrays(string s, int k) {
        if(s.Length == 1)
            if(Convert.ToInt32(s) <= k)
                return 1;
            else
                return 0;
        this.k = k;
        dp = new int[s.Length];
        this.s = s;
        return FindSolution(0);
    }

    int FindSolution(int index) {
        if(index >= s.Length)
            return 1;
        if(s[index] == '0')
            return 0;
        if(dp[index] != 0)
            return dp[index];
        int ans = 0, num = 0;
        for(int i = index; i < s.Length; i++) {
            num = num * 10 + (s[i] - '0');
            if(num > k)
                break;
            ans = (ans + FindSolution(i+1)) % mod;
        }
        return dp[index] = ans;
    }
}
```

In C#, we are providing conditions to solve the problem with some constraints provided by this language.

Through this approach, we can figure out the different arrays that satisfy the conditions and constraints. With the dynamic programming approach and recursion, we have an efficient algorithm.## Key Takeaways

To solve this problem the Dynamic Programming Approach is an ideal strategy to utilize. This approach assists in avoiding unnecessary computations by storing intermediate results. In addition, recursion plays a pivotal role in breaking down the problem into simpler sub-problems.

Each solution provided above follows the same core logic. To summarize:

1. We declare a DP array, initialized with zeroes. The size of the array is the length of the string plus one. The last element in the array is set to one as the base case.
2. Iterating backwards, we calculate the total number for every substring within the range (that is equal to or less than `k`). We add the number of ways to get to the remaining part of the string (future substring) to the DP array at the current index.
3. In cases where the substring starts with zero or the number exceeds `k`, we break the loop.
4. For each valid substring, we add the DP value at the next index to the current index in the DP array.
5. In the end, we return the value at the first index of the DP array. This value gives us the total number of valid distinct arrays that can be formed from the string.
6. We utilize memoization to avoid computation of the same sub-problem again and again, improving efficiency.

Remember: Always handle edge cases like leading zeroes and keep the constraints in mind while developing your solution. Also, when the output can be very large, using modulo arithmetic can prevent integer overflow.