Count Possible Decodings of a given Digit Sequence

Let 1 represent ‘A’, 2 represents ‘B’, etc. Given a digit sequence, count the number of possible decodings of the given digit sequence.

Examples:

```
Input:  digits[] = "121"
Output: 3
// The possible decodings are "ABA", "AU", "LA"

Input: digits[] = "1234"
Output: 3
// The possible decodings are "ABCD", "LCD", "AWD"
```

An empty digit sequence is considered to have one decoding. It may be assumed that the input contains valid digits from 0 to 9 and there are no leading 0’s, no extra trailing 0’s and no two or more consecutive 0’s.

We strongly recommend to minimize the browser and try this yourself first.

This problem is recursive and can be broken in sub-problems. We start from end of the given digit sequence. We initialize the total count of decodings as 0. We recur for two subproblems.
1) If the last digit is non-zero, recur for remaining (n-1) digits and add the result to total count.
2) If the last two digits form a valid character (or smaller than 27), recur for remaining (n-2) digits and add the result to total count.

Following is C++ implementation of the above approach.

```
// A naive recursive C++ implementation to count number
// that can be formed from a given digit sequence
#include <iostream>
#include <cstring>
```

```
using namespace std;

// Given a digit sequence of length n, returns count of
// decodings by replacing 1 with A, 2 woth B, ... 26 wit
int countDecoding(char *digits, int n)
{
    // base cases
    if (n == 0 || n == 1)
        return 1;

    int count = 0;  // Initialize count

    // If the last digit is not 0, then last digit must
    // the number of words
    if (digits[n-1] > '0')
        count =  countDecoding(digits, n-1);

    // If the last two digits form a number smaller than
    // then consider last two digits and recur
    if (digits[n-2] < '2' || (digits[n-2] == '2' && digi
        count +=  countDecoding(digits, n-2);

    return count;
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecoding(digits, n);
    return 0;
}
```

Output:

```
Count is 3
```

The time complexity of above the code is exponential. If we take a closer look at the above program, we can observe that the recursive solution is similar to Fibonacci Numbers. Therefore, we can optimize the above solution to work in O(n) time using Dynamic Programming. Following is C++ implementation for the same.

```
// A Dynamic Programming based C++ implementation to cou
#include <iostream>
#include <cstring>
using namespace std;

// A Dynamic Programming based function to count decodin
int countDecodingDP(char *digits, int n)
{
    int count[n+1]; // A table to store results of subpr
    count[0] = 1;
    count[1] = 1;

    for (int i = 2; i <= n; i++)
    {
        count[i] = 0;

        // If the last digit is not 0, then last digit m
        // the number of words
        if (digits[i-1] > '0')
            count[i] = count[i-1];

        // If second last digit is smaller than 2 and la
        // smaller than 7, then last two digits form a v
        if (digits[i-2] < '2' || (digits[i-2] == '2' && 
            count[i] += count[i-2];
    }
    return count[n];
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecodingDP(digits, n);
    return 0;
}
```

Output:

```
Count is 3
```

Time Complexity of the above solution is O(n) and it requires O(n) auxiliary space. We can reduce auxiliary space to O(1) by using space optimized version discussed in the Fibonacci Number Post.