Look-and-Say Sequence or Count and Say

Find the n'th term in Look-and-say (Or Count and Say) Sequence. The look-and-say sequence is the sequence of the below integers:
1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ...

How is the above sequence generated?

n'th term is generated by reading (n-1)'th term.

```
The first term is "1"
Second term is "11", generated by reading first term as "One 1"
(There is one 1 in previous term)
Third term is "21", generated by reading second term as "Two 1"
Fourth term is "1211", generated by reading third term as "One 2 One
1"
and so on
```

How to find n'th term?

Example:

```
Input: n = 3
Output: 21
Input: n = 5
Output: 111221
```

The idea is simple, we generate all terms from 1 to n. First, two terms are initialized as "1" and "11", and all other terms are generated using previous terms. To generate a term using the previous term, we scan the previous term. While scanning a term, we simply keep track of the count of all consecutive characters. For a sequence of the same characters, we append the count followed by the character to generate the next term.
Below is an implementation of the above idea.

```cpp
// C++ program to find n'th term in look and say
// sequence
#include <bits/stdc++.h>
using namespace std;

// Returns n'th term in look-and-say sequence
string countnndSay(int n)
{
```

```
// Base cases
if (n == 1)      return "1";
if (n == 2)      return "11";

// Find n'th term by generating all terms from 3 to
// n-1.  Every term is generated using previous term
string str = "11"; // Initialize previous term
for (int i = 3; i<=n; i++)
{
    // In below for loop, previous character
    // is processed in current iteration. That
    // is why a dummy character is added to make
    // sure that loop runs one extra iteration.
    str += '$';
    int len = str.length();

    int cnt = 1; // Initialize count of matching chars
    string  tmp = ""; // Initialize i'th term in series

    // Process previous term to find the next term
    for (int j = 1; j < len; j++)
    {
        // If current character doesn't match
        if (str[j] != str[j-1])
        {
            // Append count of str[j-1] to temp
            tmp += cnt + '0';

            // Append str[j-1]
            tmp += str[j-1];

            // Reset count
            cnt = 1;
        }

        //  If matches, then increment count of matching
        // characters
        else cnt++;
    }

    // Update str
    str = tmp;
}
```

```
    return str;
}

// Driver program
int main()
{
    int N = 3;
    cout << countnndSay(N) << endl;
    return 0;
}
```


Output

```
21
```

Time Complexity : O(n2)
Auxiliary Space : O(1)
Another Approach(Using STL): There is one more idea where we can use unordered_map from c++ stl to track the count of digits. Basic idea is to use a generator function that will generate a string from the previous string. In the count and say function we will iterate over integers from 1 to n-1 and keep updating our result.

- C++
- Java
- Python3
- C#
- Javascript

```
#include <bits/stdc++.h>
```

```
using namespace std;

// generator function returns int string from prev int
// string e.g. -> it will return '1211' for '21' ( One 2's
// and One 1)
string generator(string str)
{
    string ans = "";

    unordered_map<char, int>
        tempCount; // It is used to count integer sequence

    for (int i = 0; i < str.length() + 1; i++) {
        // when current char is different from prev one we
        // clear the map and update the ans
        if (tempCount.find(str[i]) == tempCount.end()
            && i > 0) {
            auto prev = tempCount.find(str[i - 1]);
            ans += to_string(prev->second) + prev->first;
            tempCount.clear();
        }
        // when current char is same as prev one we increase
        // it's count value
        tempCount[str[i]]++;
    }

    return ans;
}

string countnndSay(int n)
{
    string res = "1"; // res variable keep tracks of string
                      // from 1 to n-1

    // For loop iterates for n-1 time and generate strings
    // in sequence "1" -> "11" -> "21" -> "1211"
    for (int i = 1; i < n; i++) {
        res = generator(res);
    }

    return res;
}
```

```
int main()
{

    int N = 3;
    cout << countnndSay(N) << endl;
    return 0;
}
```


Output

```
21
```

Thanks to Utkarsh and Neeraj for suggesting the above solution.
Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

Using dynamic programming:

The **idea** behind the dynamic programming solution to this problem is to generate each row of the look-and-say pattern based on the previous row, and store all the intermediate results in a vector of strings. By doing this, we avoid repeating the same calculations for multiple rows and reduce the overall time complexity of the solution.

The **intuition** behind this approach is that the look-and-say pattern is self-referential, meaning that each row can be generated from the previous row by counting the consecutive occurrences of each character. Therefore, if we have the ‘i’th row of the look-and-say pattern, we can generate the ‘i+1’th row by counting the consecutive occurrences of each character in the ‘i’th row.

Follow the steps to implement the above approach:

1. Initialize a vector of strings to store the intermediate results. Set the first element of the vector to “1”, which represents the first row of the look-and-say pattern.

2. Use a for loop to generate each row of the look-and-say pattern, starting from the second row (i = 1) up to the nth row (i < n).
3. For each iteration of the loop, initialize an empty string temp to store the next row of the look-and-say pattern.
4. Use a for loop to iterate through the current row of the look-and-say pattern, stored in the i-1th element of the vector.
5. Within the inner for loop, use a while loop to count the number of consecutive occurrences of the same character.
6. After the while loop, append the count of consecutive occurrences and the current character to the temp string.
7. After the inner for loop, add the temp string to the vector of strings as the ith element, which represents the next row of the look-and-say pattern.
8. After the outer for loop, return the nth element of the vector of strings, which represents the nth row of the look-and-say pattern
Below is the implementation of the above approach

- C++
- Java
- Python3
- C#
- Javascript

```
// C++ code to implement the above approach
#include <iostream>
#include <vector>

using namespace std;

// Function to generate the nth row of the look-and-say
// pattern

string generateNthRow(int n)
{
  //vector to store all the intermediary results
    vector<string> dp(n + 1);
  //initialization
    dp[1] = "1";
    for (int i = 2; i <= n; i++) {
        string prev = dp[i - 1];
```

```
        string curr = "";
        for (int j = 0; j < prev.size(); j++) {
            int count = 1;
            while (j + 1 < prev.size()
                   && prev[j] == prev[j + 1]) {
                count++;
                j++;
            }
            curr += to_string(count) + prev[j];
        }
        dp[i] = curr;
    }
    return dp[n];
}

int main()
{
    int n = 3;
    cout << generateNthRow(n) << endl;
    return 0;
}
//This code is contributed by Veerendra Singh Rajpoot
```


Output

21

Explanation of the above code: here we are using a vector of strings dp to store the generated rows of the look-and-say pattern. We initialize the first element of the dp vector with the string “1”.
In the for loop, we iterate from the 2nd row to the nth row and generate each row based on the previous row. We use a nested for loop to iterate over each character of the previous row. The outer for loop keeps track of the current

character, and the inner while loop counts the number of consecutive occurrences of the same character. After counting the number of occurrences, we add the count and the current character to the curr string, which represents the current row of the look-and-say pattern. Finally, we update the dp[i] with the curr string.
After the loop, we return the nth row stored in dp[n]
Time Complexity :O(n*m) where n is the number of rows to generate, and m is the maximum length of a row in the look-and-say pattern.
space complexity : O(n * m), where n is the number of rows to generate and m is the maximum length of a row in the look-and-say pattern.
Another Approach(Using Stacks):
Intuition:
the problem is simpler if we understand it as just feeding the output of the counting function into itself n times.
f(1) => 11 // base case
f(11) => 21
f(21) => 1211
f(1211) => 111221
.
.
.
n times

each time we are passing a number into the function we are just counting the number of same digits in a sequence and replacing the digit with the count and the digit
eg 33333 -> there are 5 threes so we return 53(count + digit)
Approach:
we use a stack to count the digits.
if the stack is empty we push to the stack
if the current digit is same as the digit on the top of the stack then we push to the stach.
if the current digit is not the same then we get the length of the stack and the digit in top of the stack return the len + digit as a string, empty the stack and push the current element to the top of the stack
we do this n times. each time the output of the function will be the input to the next iteration.

```cpp
// C++ code to implement the above approach
#include <bits/stdc++.h>

using namespace std;

// Function to generate the nth row of the look-and-say
// pattern

string countAndSay(int n)
{
    if (n == 1)
        return "1";

    std::string ret{ "" };
    std::string strToCount{ countAndSay(
        n - 1) }; // we need to count and say the n-1th term
    std::stack<char> stack;

    for (int i{ 0 }; i <= strToCount.length(); ++i) {
        // if we're at the end OR we find a num that's
        // different... we add the length of the stack to ret
        // and also the num we've been counting then, we
        // empty the stack to count the next number
        if (i == strToCount.length()
            || !stack.empty()
                   && stack.top() != strToCount[i]) {
            std::stringstream ss;
            std::string toAdd{ "" };
            ss << stack.size();
            ss >> toAdd;
            ret += toAdd;
            ret += stack.top();
            while (!stack.empty())
                stack.pop();
        }
        if (i != strToCount.length())
            stack.push(strToCount[i]);
    }
    return ret;
}

int main()
{
    int n = 3;
```

```
    cout << countAndSay(n) << endl;
    return 0;
}
```

Output

```
21
```

Time Complexity : O(n2)
Auxiliary Space : O(1)

Another Approach(Using STL): There is one more idea where we can use unordered_map from c++ stl to track the count of digits. Basic idea is to use a generator function that will generate a string from the previous string. In the count and say function we will iterate over integers from 1 to n-1 and keep updating our result.

```
#include <bits/stdc++.h>

using namespace std;


// generator function returns int string from prev int

// string e.g. -> it will return '1211' for '21' ( One 2's

// and One 1)

string generator(string str)

{

        string ans = "";


        unordered_map<char, int>

                tempCount; // It is used to count integer sequence


        for (int i = 0; i < str.length() + 1; i++) {

                // when current char is different from prev one we

                // clear the map and update the ans

                if (tempCount.find(str[i]) == tempCount.end()

                        && i > 0) {
```

```
                auto prev = tempCount.find(str[i - 1]);
                ans += to_string(prev->second) + prev->first;
                tempCount.clear();
            }
            // when current char is same as prev one we increase
            // it's count value
            tempCount[str[i]]++;
        }

        return ans;
}

string countnndSay(int n)
{
        string res = "1"; // res variable keep tracks of string
                                        // from 1 to n-1

        // For loop iterates for n-1 time and generate strings
        // in sequence "1" -> "11" -> "21" -> "1211"
        for (int i = 1; i < n; i++) {
            res = generator(res);
        }

        return res;
}

int main()
{
```

```
    int N = 3;

    cout << countnndSay(N) << endl;

    return 0;

}
```

Using dynamic programming:

The **idea** behind the dynamic programming solution to this problem is to generate each row of the look-and-say pattern based on the previous row, and store all the intermediate results in a vector of strings. By doing this, we avoid repeating the same calculations for multiple rows and reduce the overall time complexity of the solution.

The **intuition** behind this approach is that the look-and-say pattern is self-referential, meaning that each row can be generated from the previous row by counting the consecutive occurrences of each character. Therefore, if we have the 'i'th row of the look-and-say pattern, we can generate the 'i+1'th row by counting the consecutive occurrences of each character in the 'i'th row.

Follow the steps to implement the above approach:

1. Initialize a vector of strings to store the intermediate results. Set the first element of the vector to "1", which represents the first row of the look-and-say pattern.
2. Use a for loop to generate each row of the look-and-say pattern, starting from the second row (i = 1) up to the nth row (i < n).
3. For each iteration of the loop, initialize an empty string temp to store the next row of the look-and-say pattern.
4. Use a for loop to iterate through the current row of the look-and-say pattern, stored in the i-1th element of the vector.
5. Within the inner for loop, use a while loop to count the number of consecutive occurrences of the same character.
6. After the while loop, append the count of consecutive occurrences and the current character to the temp string.
7. After the inner for loop, add the temp string to the vector of strings as the ith element, which represents the next row of the look-and-say pattern.
8. After the outer for loop, return the nth element of the vector of strings, which represents the nth row of the look-and-say pattern

Below is the implementation of the above approach

```cpp
// C++ code to implement the above approach
#include <iostream>
#include <vector>

using namespace std;

// Function to generate the nth row of the look-and-say
// pattern

string generateNthRow(int n)
{
  //vector to store all the intermediary results
    vector<string> dp(n + 1);
  //initialization
    dp[1] = "1";
    for (int i = 2; i <= n; i++) {
        string prev = dp[i - 1];
        string curr = "";
        for (int j = 0; j < prev.size(); j++) {
            int count = 1;
            while (j + 1 < prev.size()
                   && prev[j] == prev[j + 1]) {
                count++;
                j++;
            }
            curr += to_string(count) + prev[j];
        }
        dp[i] = curr;
    }
    return dp[n];
}

int main()
{
    int n = 3;
    cout << generateNthRow(n) << endl;
    return 0;
}
//This code is contributed by Veerendra Singh Rajpoot
```

Output
21

Explanation of the above code: here we are using a vector of strings dp to store the generated rows of the look-and-say pattern. We initialize the first element of the dp vector with the string “1”.
In the for loop, we iterate from the 2nd row to the nth row and generate each row based on the previous row. We use a nested for loop to iterate over each character of the previous row. The outer for loop keeps track of the current character, and the inner while loop counts the number of consecutive occurrences of the same character. After counting the number of occurrences, we add the count and the current character to the curr string, which represents the current row of the look-and-say pattern. Finally, we update the dp[i] with the curr string.
After the loop, we return the nth row stored in dp[n]
Time Complexity :O(n*m) where n is the number of rows to generate, and m is the maximum length of a row in the look-and-say pattern.
space complexity : O(n * m), where n is the number of rows to generate and m is the maximum length of a row in the look-and-say pattern.

Another Approach(Using Stacks):
Intuition:
the problem is simpler if we understand it as just feeding the output of the counting function into itself n times.
f(1) => 11 // base case
f(11) => 21
f(21) => 1211
f(1211) => 111221
.
.
.
n times

each time we are passing a number into the function we are just counting the number of same digits in a sequence and replacing the digit with the count and the digit

eg 33333 -> there are 5 threes so we return 53(count + digit)
Approach:
we use a stack to count the digits.
if the stack is empty we push to the stack
if the current digit is same as the digit on the top of the stack then we push to the stach.
if the current digit is not the same then we get the length of the stack and the digit in top of the stack return the len + digit as a string, empty the stack and push the current element to the top of the stack
we do this n times. each time the output of the function will be the input to the next iteration.

```
// C++ code to implement the above approach
#include <bits/stdc++.h>

using namespace std;

// Function to generate the nth row of the look-and-say
// pattern

string countAndSay(int n)
{
    if (n == 1)
        return "1";

    std::string ret{ "" };
    std::string strToCount{ countAndSay(
        n - 1) }; // we need to count and say the n-1th term
    std::stack<char> stack;

    for (int i{ 0 }; i <= strToCount.length(); ++i) {
        // if we're at the end OR we find a num that's
        // different... we add the length of the stack to ret
        // and also the num we've been counting then, we
        // empty the stack to count the next number
        if (i == strToCount.length()
            || !stack.empty()
                   && stack.top() != strToCount[i]) {
            std::stringstream ss;
```

```
            std::string toAdd{ "" };
            ss << stack.size();
            ss >> toAdd;
            ret += toAdd;
            ret += stack.top();
            while (!stack.empty())
                stack.pop();
        }
        if (i != strToCount.length())
            stack.push(strToCount[i]);
    }
    return ret;
}

int main()
{
    int n = 3;
    cout << countAndSay(n) << endl;
    return 0;
}
```


Output

21

Time complexity: O(kn),since we are calling the function n times and for k length of the string

Auxiliary Space: O(kn),stack space needed.