Previous Next

552. Student Attendance Record II

April 15, 2017 | 13.3K views

Average Rating: 3.92 (25 votes)

Given a positive integer **n**, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod $10^9 + 7$.

A student attendance record is a string that only contains the following three characters:

1. **'A'** : Absent.
2. **'L'** : Late.
3. **'P'** : Present.

A record is regarded as rewardable if it doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.

Example 1:

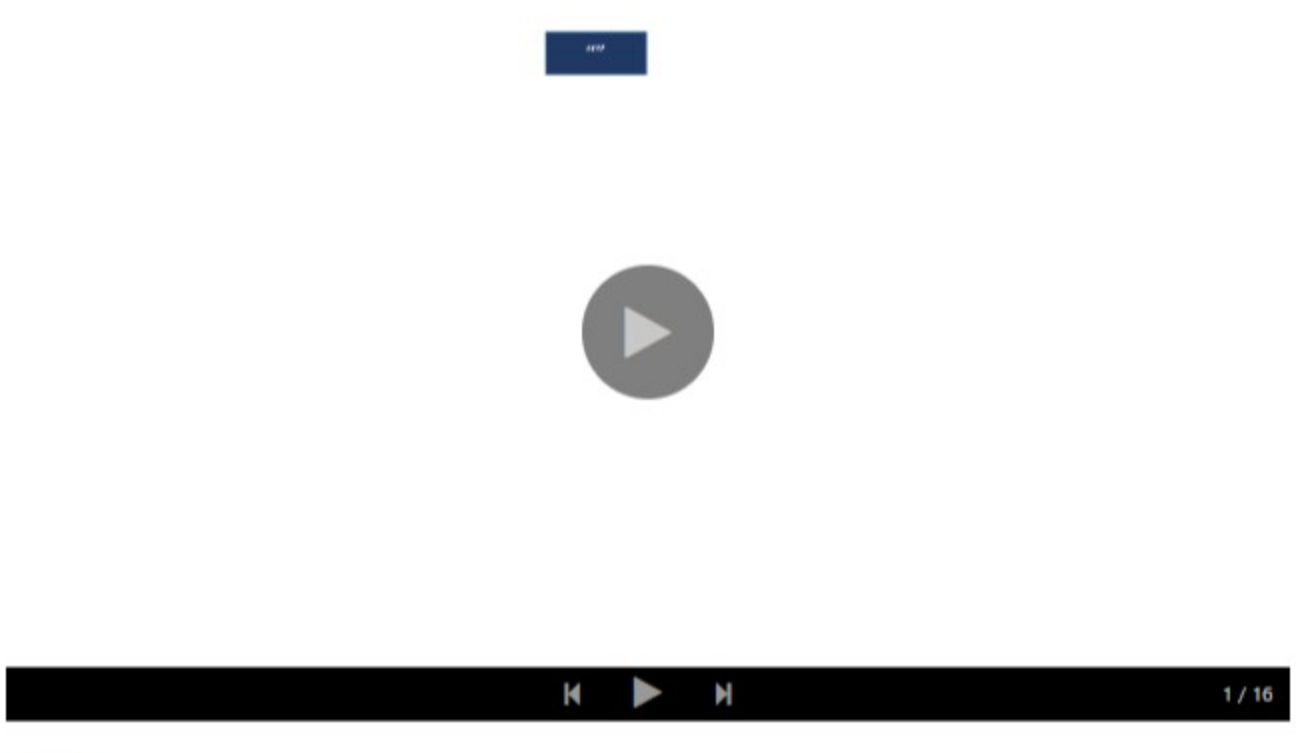
```
Input: n = 2
Output: 8
Explanation:
There are 8 records with length 2 will be regarded as rewardable:
"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"
Only "AA" won't be regarded as rewardable owing to more than one absent times.
```

Note: The value of **n** won't exceed 100,000.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

In the brute force approach, we actually form every possible string comprising of the letters "A", "P", "L" and check if the string is rewardable by checking it against the given criterias. In order to form every possible string, we make use of a recursive `gen(string, n)` function. At every call of this function, we append the letters "A", "P" and "L" to the input string, reduce the required length by 1 and call the same function again for all the three newly generated strings.



Java

```java
public class Solution {
    int count,M=1000000007;
    public int checkRecord(int n) {
        count = 0;
        gen("", n);
        return count;
    }
    public void gen(String s, int n) {
        if (n == 0 && checkRecord(s))
            count=(count+1)%M;
        else if (n > 0) {
            gen(s + "A", n - 1);
            gen(s + "P", n - 1);
            gen(s + "L", n - 1);
        }
    }
    public boolean checkRecord(String s) {
        int count = 0;
        for (int i = 0; i < s.length() && count < 2; i++)
            if (s.charAt(i) == 'A')
                count++;
        return s.length() > 0 && count < 2 && s.indexOf("LLL") < 0;
    }
}
```

Complexity Analysis

- Time complexity : $O(3^n)$. Exploring 3^n combinations.
- Space complexity : $O(n^n)$. Recursion tree can grow upto depth n and each node contains string of length $O(n)$.

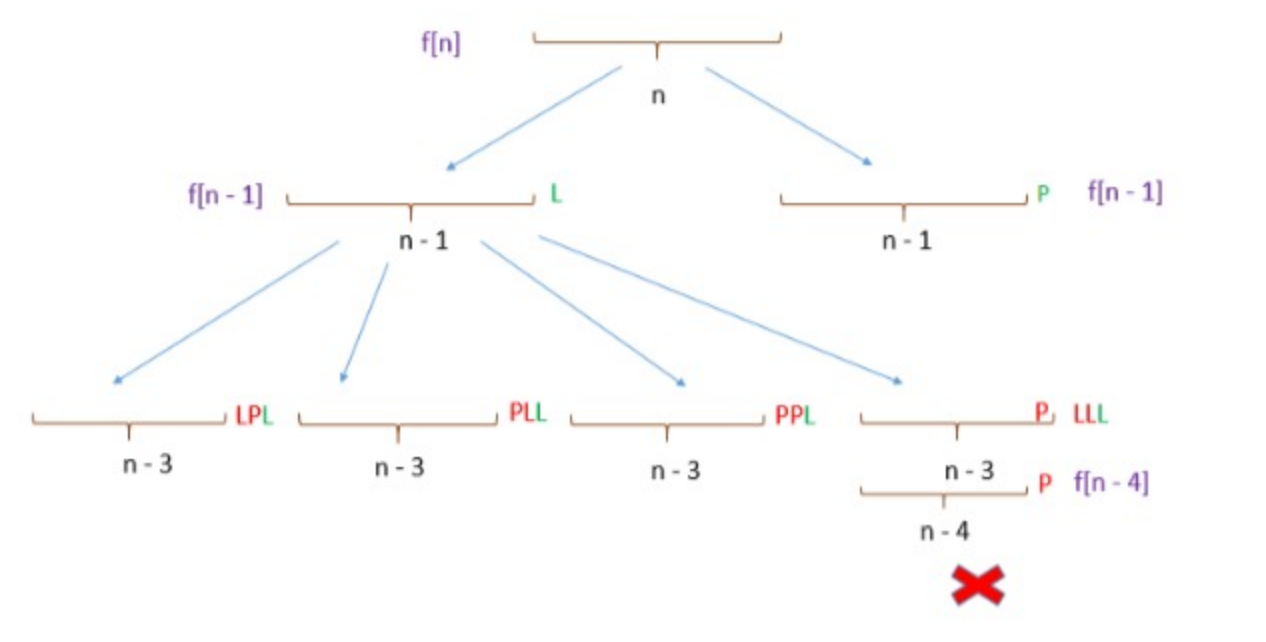
Approach #2 Using Recursive formulae [Time Limit Exceeded]

Algorithm

The given problem can be solved easily if we can develop a recurring relation for it.

Firstly, assume the problem to be considering only the characters L and P in the strings. i.e. The strings can contain only L and P. The effect of A will be taken into account later on.

In order to develop the relation, let's assume that $f[n]$ represents the number of possible rewardable strings(with L and P as the only characters) of length n. Then, we can easily determine the value of $f[n]$ if we know the values of the counts for smaller values of n. To see how it works, let's examine the figure below:



The above figure depicts the division of the rewardable string of length n into two strings of length $n-1$ and ending with L or P. The string ending with P of length n is always rewardable provided the string of length $n-1$ is rewardable. Thus, this string accounts for a factor of $f[n-1]$ to $f[n]$.

For the first string ending with L, the rewardability is dependent on the further strings of length $n-3$. Thus, we consider all the rewardable strings of length $n-3$ now. Out of the four combinations possible at the end, the fourth combination, ending with a LL at the end leads to an unawardable string. But, since we've considered only rewardable strings of length $n-3$, for the last string to be rewardable at length $n-3$ and unawardable at length $n-1$, it must be preceded by a P before the LL.

Thus, accounting for the first string again, all the rewardable strings of length $n-1$, except the strings of length $n-4$ followed by PLL, can contribute to a rewardable string of length n. Thus, this string accounts for a factor of $f[n-1]-f[n-4]$ to $f[n]$.

Thus, the recurring relation becomes:

$$f[n] = 2f[n-1] - f[n-4]$$

We store all the $f[i]$ values in an array. In order to compute $f[i]$, we make use of a recursive function `func(n)` which makes use of the above recurrence relation.

Now, we need to put the factor of character A being present in the given string. We know, atmost one A is allowed to be presnet in a rewardable string. Now, consider the two cases.

1. No A is present: In this case, the number of rewardable strings is the same as $f[n]$.
2. A single A is present: Now, the single A can be present at any of the n positions. If the A is present at the i^{th} position in the given string, in the form: "<(i-1) characters>, A, <(n-i) characters>", the total number of rewardable strings is given by: $f[i-1] * f[n-i]$. Thus, the total number of such substrings is given by: $\sum_{i=1}^{n} (f[i-1] * f[n-i])$.

Java

```java
public class Solution {
    int M=1000000007;
    public int checkRecord(int n) {
        int[] f =new int[n+1];
        f[0]=1;
        for(int i=1;i<=n;i++)
            f[i]=func(i);
        int sum=func(n);
        for(int i=1;i<=n;i++){
            sum+=(f[i-1]*f[n-i])%M;
        }
        return sum%M;
    }
    public int func(int n)
    {
        if(n==0)
            return 1;
        if(n==1)
            return 2;
        if(n==2)
            return 4;
        if(n==3)
            return 7;
        return (2*func(n-1) - func(n-4))%M;
    }
}
```

Complexity Analysis

- Time complexity : $O(2^n)$. method $func$ will take 2^n time.
- Space complexity : $O(n)$. f array is used of size n.

Approach #3 Using Dynamic Programming [Accepted]

Algorithm

In the last approach, we calculated the values of $f[i]$ everytime using the recursive function, which goes till its root depth everytime. But, we can reduce a large number of redundant calculations, if we use the results obtained for previous $f[j]$ values directly to obtain $f[i]$ as $f[i] = 2f[i-1] + f[i-4]$.

Java

```java
public class Solution {
    long M = 1000000007;
    public int checkRecord(int n) {
        long[] f = new long[n <= 5 ? 6 : n + 1];
        f[0] = 1;
        f[1] = 2;
        f[2] = 4;
        f[3] = 7;
        for (int i = 4; i <= n; i++)
            f[i] = ((2 * f[i - 1]) % M + (M - f[i - 4])) % M;
        long sum = f[n];
        for (int i = 1; i <= n; i++) {
            sum += (f[i - 1] * f[n - i]) % M;
        }
        return (int)(sum % M);
    }
}
```

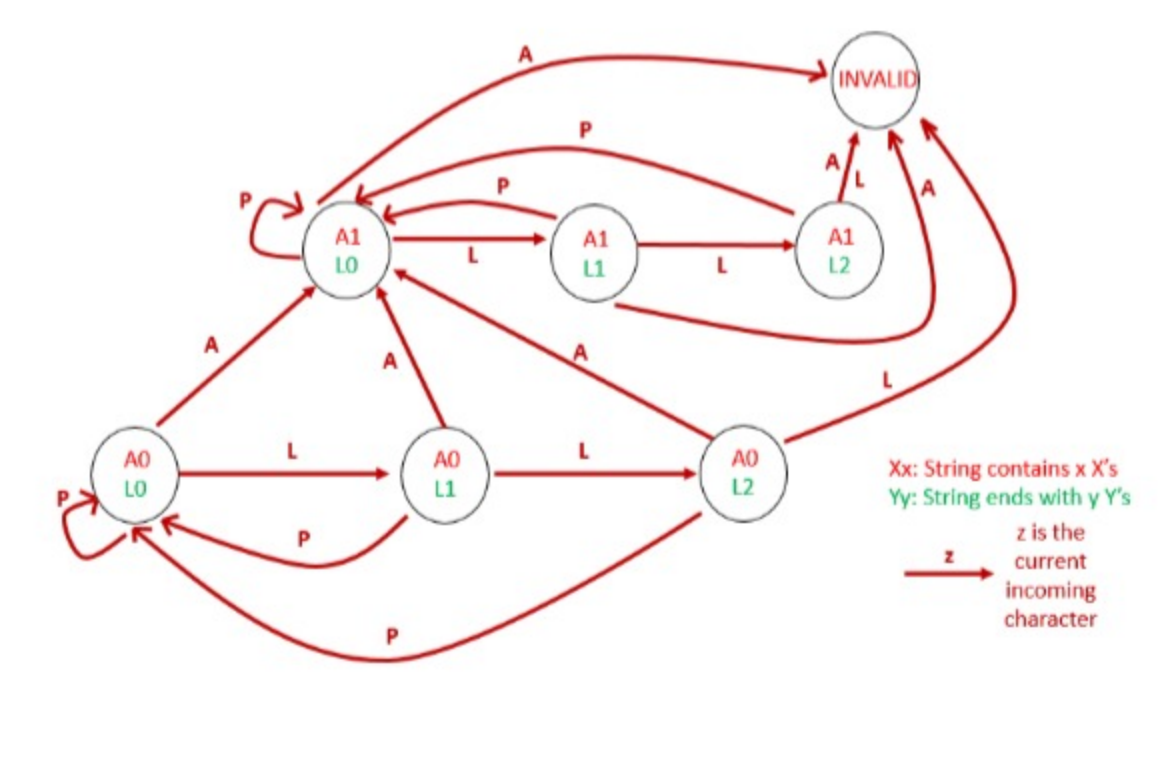
Complexity Analysis

- Time complexity : $O(n)$. One loop to fill f array and one to calculate sum
- Space complexity : $O(n)$. f array of size n is used.

Approach #4 Dynamic Programming with Constant Space [Accepted]

Algorithm

We can observe that the number and position of P's in the given string is irrelevant. Keeping into account this fact, we can obtain a state diagram that represents the transitions between the possible states as shown in the figure below:



This state diagram contains the states based only upon whether an A is present in the string or not, and on the number of L's that occur at the trailing edge of the string formed till now. The state transition occurs whenver we try to append a new character to the end of the current string.

Based on the above state diagram, we keep a track of the number of unique transitions from which a rewardable state can be achieved. We start off with a string of length 0 and keep on adding a new character to the end of the string till we achieve a length of n. At the end, we sum up the number of transitions possible to reach each rewardable state to obtain the required result.

We can use variables corresponding to the states. $axly$ represents the number of strings of length i containing x $a's$ and ending with y $l's$.

Below code is inspired by @stefanpochmann.

Java

```java
public class Solution {
    long M = 1000000007;
    public int checkRecord(int n) {
        long a0l0 = 1;
        long a0l1 = 0, a0l2 = 0, a1l0 = 0, a1l1 = 0, a1l2 = 0;
        for (int i = 0; i < n; i++) {
            long new_a0l0 = (a0l0 + a0l1 + a0l2) % M;
            long new_a0l1 = a0l0;
            long new_a0l2 = a0l1;
            long new_a1l0 = (a0l0 + a0l1 + a0l2 + a1l0 + a1l1 + a1l2) % M;
            long new_a1l1 = a1l0;
            long new_a1l2 = a1l1;
            a0l0 = new_a0l0;
            a0l1 = new_a0l1;
            a0l2 = new_a0l2;
            a1l0 = new_a1l0;
            a1l1 = new_a1l1;
            a1l2 = new_a1l2;
        }
        return (int)((a0l0 + a0l1 + a0l2 + a1l0 + a1l1 + a1l2) % M);
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Single loop to update the states.
- Space complexity : $O(1)$. Constant Extra Space is used.

Approach #5 Using less variables [Accepted]

Algorithm

In the last approach discussed, we've made use of six extra temporary variables just to keep a track of the change in the current state. The same result can be obtained by using a lesser number of temporary variables too.

Java

```java
public class Solution {
    long M = 1000000007;
    public int checkRecord(int n) {
        long a0l0 = 1, a0l1 = 0, a0l2 = 0, a1l0 = 0, a1l1 = 0, a1l2 = 0;
        for (int i = 0; i <= n; i++) {
            long a0l0_ = (a0l0 + a0l1 + a0l2) % M;
            a0l2 = a0l1;
            a0l1 = a0l0;
            a0l0 = a0l0_;
            long a1l0_ = (a0l0 + a1l0 + a1l1 + a1l2) % M;
            a1l2 = a1l1;
            a1l1 = a1l0;
            a1l0 = a1l0_;
        }
        return (int) a1l0;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Single loop to update the states.
- Space complexity : $O(1)$. Constant Extra Space is used.

Rate this article:

Previous Next

Comments: 17

Sort By

Type comment here... (Markdown is supported)

Preview Post

elle ★ 22 April 28, 2017 8:26 PM

The simpler way, IMHO, and compliments to N. Shade at math.stackexchange.com, is to consider the cases (a) ------P, (b) ------PL, and (c) ------PLL. A minute of thought will reveal that these are mutually exclusive. It's also realized that going further, e.g. (d) ------PLLL, becomes useless because the string will always be unrewardable. So the whole of f[n] may be summarized by (a) + (b) + (c), in other words, f[n] = f[n-1] + f[n-2] + f[n-3]. This means also that f[n-1] = f[n-2] + f[n-3] + f[n-4]. If you subtract the latter

Read More

16 Share Reply

Kinsapoon ★ 35 July 20, 2018 2:50 AM

why we cannot just do - f[i - 4] % M but need to + (M - f[i - 4]) % 4

4 Share Reply

SHOW 1 REPLY

wangzi6147 ★ 3745 April 11, 2018 9:12 AM

In approach 3, it should be f[i] = 2f[i−1] - f[i−4]. Both in description and the code.

4 Share Reply

SHOW 2 REPLIES

nileshdagrawal ★ 2 August 30, 2017 7:42 AM

Space complexity of brute force approach is wrong it cannot be O(n^n) unless you store all the entries, which you are not doing.
Space complexity will remain O(n) which is equal to depth of the tree.

2 Share Reply

SHOW 1 REPLY

wqmbisheng ★ 8 October 21, 2017 3:48 AM

Can anyone prove the mod operation in detail? (f[i] = ((2 * f[i - 1]) % M + (M - f[i - 4])) % M;)
why is the result still right after we did three mod operations each time ?

1 Share Reply

SHOW 1 REPLY

giftwei ★ 171 June 7, 2019 1:26 AM

Outdated, long-winded, and not clear at key point as always. If such articles were released right after contest, I would give 5 stars rating:

1 Share Reply

Galileo_Galilei ★ 477 January 4, 2019 10:11 AM

@vinod23 Hi I am still confused why string of length `n-3` ending with `PLL` should be considered instead of `n-2` ending with `LL` in the second approach. Because whatever character behind `LL` would be, it could always form `LLL` if we append `L` to `LL`, right? In my case, in 2nd approach, this line `f[i] = ((2 * f[i - 1]) % M + (M - f[i - 4])) % M;` should be changed into

Read More

0 Share Reply

elle ★ 22 April 28, 2017 8:03 PM

Right. So consider the 1st of the 4 branches: [n-3]LPL. A string like PPPPPPPPPPPPPPPLLLPL would be rewardable at [n-3] but fail at [n-1], would it not? That was my original question: Under your approach, why doesn't the 1st branch get considered as a "rewardable at [n-3] but unrewardable at [n-1]" case? Maybe your criterion is more specific but not fully stated; I would like to understand that part.

0 Share Reply

SHOW 1 REPLY

vinod23 ★ 461 April 27, 2017 1:29 AM

@elle This is because, we need to consider only those strings which are rewardable at the length (n-3) but become unrewardable at a length(n-1)(for left branch). Only the strings ending with a P at length (n-3) and ending with PLL of the left branch at length (n-1) follow this criteria. Could you please share your simpler explanation?

0 Share Reply

elle ★ 22 April 26, 2017 7:47 PM

Thanks for your reply. Why fix an additional P to make ---PLLL? Isn't ---LLLL also invalid? The final recurrence relation is absolutely correct, but I'm not sure this is the correct explanation. I can explain a simpler way to get to 2f[n-1]-f[n-4] but first I would like to understand yours. By the way, that's a typo in the graphic, writing "N" for the right branch instead of "P", right?

0 Share Reply

‹ 1 2 ›