Previous Next

135. Candy

March 16, 2017 | 56.3K views

Average Rating: 4.74 (49 votes)

There are *N* children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

Example 1:

```
Input: [1,0,2]
Output: 5
Explanation: You can allocate to the first, second and third child with 2, 1, 2 candie
```

Example 2:

```
Input: [1,2,2]
Output: 4
Explanation: You can allocate to the first, second and third child with 1, 2, 1 candie
             The third child gets 1 candy because it satisfies the above two conditior
```

Solution

Approach 1: Brute Force

The simplest approach makes use of a 1-d array, $candies$ to keep a track of the candies given to the students. Firstly, we give 1 candy to each student. Then, we start scanning the array from left-to-right. At every element encountered, firstly, if the current element's ratings, $ratings[i]$, is larger than the previous element $ratings[i-1]$ and $candies[i] <= candies[i-1]$, then we update $candies[i]$ as $candies[i] = candies[i-1]+1$. Thus, now the candy distribution for these two elements $candies[i-1]$ and $candies[i]$ becomes correct for the time being(locally). In the same step, we also check if the current element's ratings, $ratings[i]$, is larger than the next element's ratings, i.e. $ratings[i] > ratings[i+1]$. If so, we again update $candies[i] = candies[i+1]+1$. We continue this process for the whole $ratings$ array. If in any traversal, no updation of the $candies$ array occurs, it means we've reached at the final required distribution of the candies and we can stop the traversals. To keep a track of this we make use of a $flag$ which is set to True if any updation occurs in a traversal.

At the end, we can sum up all the elements of the $candies$ array to obtain the required minimum number of candies.

Java

```java
public class Solution {
    public int candy(int[] ratings) {
        int[] candies = new int[ratings.length];
        Arrays.fill(candies, 1);
        boolean flag = true;
        int sum = 0;
        while (flag) {
            flag = false;
            for (int i = 0; i < ratings.length; i++) {
                if (i != ratings.length - 1 && ratings[i] > ratings[i + 1] && candies[i] <= candies[i + 1]) {
                    candies[i] = candies[i + 1] + 1;
                    flag = true;
                }
                if (i > 0 && ratings[i] > ratings[i - 1] && candies[i] <= candies[i - 1]) {
                    candies[i] = candies[i - 1] + 1;
                    flag = true;
                }
            }
        }
        for (int candy : candies) {
            sum += candy;
        }
        return sum;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. We need to traverse the array at most n times.
- Space complexity : $O(n)$. One $candies$ array of size n is used.

Approach 2: Using two arrays

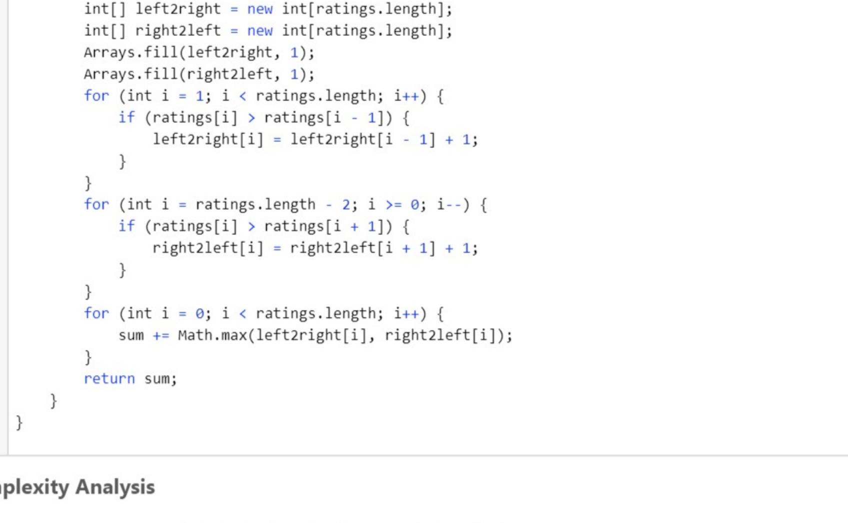
Algorithm

In this approach, we make use of two 1-d arrays $left2right$ and $right2left$. The $left2right$ array is used to store the number of candies required by the current student taking care of the distribution relative to the left neighbours only. i.e. Assuming the distribution rule is: The student with a higher ratings than its left neighbour should always get more candies than its left neighbour. Similarly, the $right2left$ array is used to store the number of candies candies required by the current student taking care of the distribution relative to the right neighbours only. i.e. Assuming the distribution rule to be: The student with a higher ratings than its right neighbour should always get more candies than its right neighbour. To do so, firstly we assign 1 candy to each student in both $left2right$ and $right2left$ array. Then, we traverse the array from left-to-right and whenever the current element's ratings is larger than the left neighbour we update the current element's candies in the $left2right$ array as $left2right[i] = left2right[i-1]+1$, since the current element's candies are always less than or equal candies than its left neighbour before updation. After the forward traversal, we traverse the array from left-to-right and update $right2left[i]$ as $right2left[i] = right2left[i+1]+1$, whenever the current (i^{th}) element has a higher ratings than the right ($(i+1)^{th}$) element.

Now, for the i^{th} student in the array, we need to give $\max(left2right[i], right2left[i])$ to it, in order to satisfy both the left and the right neighbour relationship. Thus, at the end, we obtain the minimum number of candies required as:

\text{minimum_candies}=\sum_{i=0}^{n-1} \text{max}(left2right[i], right2left[i]), \text{where } n = \text{length of the ratings array.}

The following animation illustrates the method:



Java

```java
public class Solution {
    public int candy(int[] ratings) {
        int sum = 0;
        int[] left2right = new int[ratings.length];
        int[] right2left = new int[ratings.length];
        Arrays.fill(left2right, 1);
        Arrays.fill(right2left, 1);
        for (int i = 1; i < ratings.length; i++) {
            if (ratings[i] > ratings[i - 1]) {
                left2right[i] = left2right[i - 1] + 1;
            }
        }
        for (int i = ratings.length - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                right2left[i] = right2left[i + 1] + 1;
            }
        }
        for (int i = 0; i < ratings.length; i++) {
            sum += Math.max(left2right[i], right2left[i]);
        }
        return sum;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. $left2right$ and $right2left$ arrays are traversed thrice.
- Space complexity : $O(n)$. Two arrays $left2right$ and $right2left$ of size n are used.

Approach 3: Using one array

Algorithm

In the previous approach, we used two arrays to keep track of the left neighbour and the right neighbour relation individually and later on combined these two. Instead of this, we can make use of a single array $candies$ to keep the count of the number of candies to be allocated to the current student. In order to do so, firstly we assign 1 candy to each student. Then, we traverse the array from left-to-right and distribute the candies following only the left neighbour relation i.e. whenever the current element's ratings is larger than the left neighbour and has less than or equal candies than its left neighbour, we update the current element's candies in the $candies$ array as $candies[i] = candies[i-1]+1$. While updating we need not compare $candies[i]$ and $candies[i-1]$, since $candies[i] \leq candies[i-1]$ before updation. After this, we traverse the array from right-to-left. Now, we need to update the i^{th} element's candies in order to satisfy both the left neighbour and the right neighbour relation. Now, during the backward traversal, if $ratings[i] > ratings[i+1]$, considering only the right neighbour criteria, we could've updated $candies[i]$ as $candies[i] = candies[i+1]+1$. But, this time we need to update the $candies[i]$ only if $candies[i] \leq candies[i+1]$. This happens because, this time we've already altered the $candies$ array during the forward traversal and thus $candies[i]$ isn't necessarily less than or equal to $candies[i+1]$. Thus, if $ratings[i] > ratings[i+1]$, we can update $candies[i]$ as $candies[i] = \max(candies[i], candies[i+1]+1)$, which makes $candies[i]$ satisfy both the left neighbour and the right neighbour criteria.

Again, we need sum up all the elements of the $candies$ array to obtain the required result.

\[\text{minimum_candies} = \sum_{i=0}^{n-1} candies[i], \text{where } n = \text{length of the ratings array.}
\]

Java

```java
public class Solution {
    public int candy(int[] ratings) {
        int[] candies = new int[ratings.length];
```

```java
public class Solution {
    public int candy(int[] ratings) {
        int[] candies = new int[ratings.length];
        Arrays.fill(candies, 1);
        for (int i = 1; i < ratings.length; i++) {
            if (ratings[i] > ratings[i - 1]) {
                candies[i] = candies[i - 1] + 1;
            }
        }
        int sum = candies[ratings.length - 1];
        for (int i = ratings.length - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                candies[i] = Math.max(candies[i], candies[i + 1] + 1);
            }
            sum += candies[i];
        }
        return sum;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. The array $candies$ of size n is traversed thrice.
- Space complexity : $O(n)$. An array $candies$ of size n is used.

Approach 4: Single Pass Approach with Constant Space

Algorithm

This approach relies on the observation(as demonstrated in the figure below as well) that in order to distribute the candies as per the given criteria using the minimum number of candies, the candies are always distributed in terms of increments of 1. Further, while distributing the candies, the local minimum number of candies given to a student is 1. Thus, the sub-distributions always take the form: $1, 2, 3, ..., \text{n}$ or $\text{n}, ..., 2, 1$, whose sum is simply given by $n(n+1)/2$.

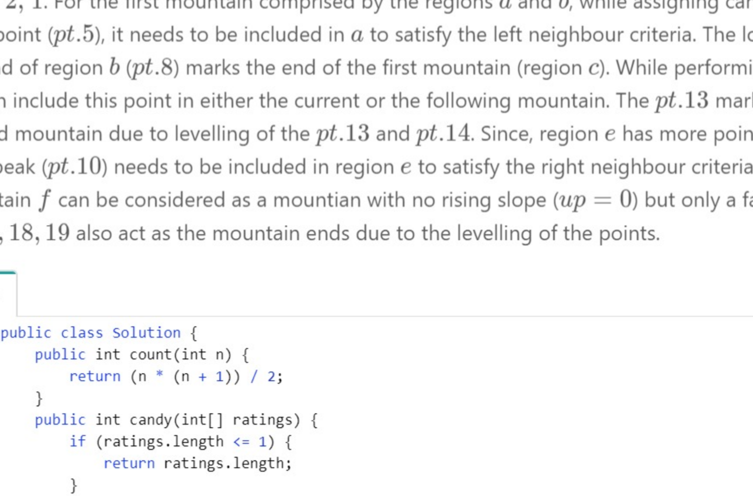
Now, we can view the given $rankings$ as some rising and falling slopes. Whenever the slope is rising, the distribution takes the form: $1, 2, 3, ..., \text{m}$. Similarly, a falling slope takes the form: $\text{k}, ..., 2, 1$. An issue that arises now is that the local peak point can be included in only one of the slopes. Whether to include the local peak point(n) in the rising slope or the falling slope?

In order to decide it, we can observe that in order to satisfy both the right neighbour and the left neighbour criteria, the peak point's count needs to be the max. of the counts determined by the rising and the falling slopes. Thus, in order to determine the number of candies required, the peak point needs to be included in the slope which contains more number of points. The local valley point can also be included in only one of the slopes, but this issue can be resolved easily, since the local valley point will always be assigned a candy count of 1(which can be subtracted from the next slope's count calculations).

Coming to the implementation, we maintain two variables old_slope and new_slope to determine the occurence of a peak or a valley. We also use up and $down$ variables to keep a track of the count of elements on the rising slope and on the falling slope respectively(without including the peak element). We always update the total count of $candies$ at the end of a falling slope following a rising slope (or a mountain). The leveling of the points in $rankings$ also works as the end of a mountain. At the end of the mountain, we determine whether to include the peak point in the rising slope or in the falling slope by comparing the up and $down$ variables up to that point. Thus, the count assigned to the peak element becomes: $\max(up, down)+1$. At this point, we can reset the up and $down$ variables indicating the start of a new mountain.

The following figure shows the cases that need to be handled for this example:

`rankings: [1 2 3 4 5 3 2 1 2 6 5 4 3 3 2 1 1 3 3 3 4 2]`



From this figure, we can see that the candy distributions in the subregions always take the form $1, 2, ...\text{n}$ or $\text{n}, ..., 2, 1$. For the first mountain comprised by the regions a and b, while assigning candies to the local peak point ($pt.5$), it needs to be included in a to satisfy the left neighbour criteria. The local valley point at the end of region b ($pt.8$) marks the end of the first mountain (region c). While performing the calculations, we can include this point in either the current or the following mountain. The $pt.13$ marks the end of the second mountain due to levelling of the $pt.13$ and $pt.14$. Since, region e has more points than region d, the local peak ($pt.10$) needs to be included in region e to satisfy the right neighbour criteria. Now, the third mountain f can be considered as a mountian with no rising slope ($up = 0$) but only a falling slope. Similarly, $pt.16, 18, 19$ also act as the mountain ends due to the levelling of the points.

Java

```java
public class Solution {
    public int count(int n) {
        return (n * (n + 1)) / 2;
    }
    public int candy(int[] ratings) {
        if (ratings.length <= 1) {
            return ratings.length;
        }
        int candies = 0;
        int up = 0;
        int down = 0;
        int old_slope = 0;
        for (int i = 1; i < ratings.length; i++) {
            int new_slope = (ratings[i] > ratings[i - 1]) ? 1 : (ratings[i] < ratings[i - 1] ? -1 : 0);
            if ((old_slope > 0 && new_slope == 0) || (old_slope < 0 && new_slope >= 0)) {
                candies += count(up) + count(down) + Math.max(up, down);
                up = 0;
                down = 0;
            }
            if (new_slope > 0)
                up++;
            if (new_slope < 0)
                down++;
            if (new_slope == 0)
                candies++;

            old_slope = new_slope;
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse the $rankings$ array once only.
- Space complexity : $O(1)$. Constant Extra Space is used.

Rate this article: ★★★★★

Previous Next

Comments: 27 Sort By

Type comment here... (Markdown is supported)

Preview Post

meng789987 ★ 1085 June 3, 2018 11:38 AM
@Administrator, can you collect my solution? it's cleaner than all the above and the tops in the discussion.
https://leetcode.com/problems/candy/discuss/135698/Simple-solution-with-one-pass-using-O(1)-space
Read More
43 Share Reply
SHOW 1 REPLY

root5 ★ 30 September 3, 2017 5:35 AM
Will an interviewer really expect someone to come up with the slope based solution during an interview?
24 Share Reply
SHOW 3 REPLIES

vvstej ★ 43 September 21, 2018 5:11 AM
There is one more nlogn solution with sorting.
9 Share Reply

minweny ★ 219 June 23, 2018 12:52 PM
I like your approach 4, which is so amazing, but I implemented it in a different way.

```java
public int candy(int[] ratings) {
    if(ratings==null || ratings.length==0) return 0;
    int start=0 sum=0 len=ratings length;
```

Read More
8 Share Reply
SHOW 1 REPLY

harry0107100 ★ 17 September 8, 2019 7:16 AM
Can someone please explain why in the first solution, we need to traverse the array at most n times?
3 Share Reply
SHOW 1 REPLY

madno ★ 302 March 9, 2018 1:54 AM
@bonsairobo
The apparent difficulty is because of the biggish number of different cases raised by the various interconnection of hills, valleys and plateaus.

By systematically writing out the cases upfront, one can reduce the
Read More
3 Share Reply

mgh ★ 112 May 10, 2017 12:46 PM
Inside of the for loop why not added by 1: candies += count(up) + count(down) + Math.max(up, down);
3 Share Reply
SHOW 1 REPLY

hello_world_cn ★ 279 December 26, 2018 4:51 PM
Add some comments for Approach 4:

```java
public class Solution {
    public int count(int n) {
        return (n * (n + 1)) / 2;
```

Read More
3 Share Reply
SHOW 3 REPLIES

NideeshT ★ 579 July 16, 2019 6:28 AM
Java Code + Whiteboard Youtube Video Explanation accepted - https://www.youtube.com/watch?v=ntXYNe4D8m0 (clickable link)

Read More
2 Share Reply
SHOW 1 REPLY

bigdogs ★ 1 January 31, 2018 8:34 AM
Is the figure match the ranking array?
1 Share Reply

‹ 1 2 3 ›