Candy

29 May 2024 11:50

Problem statement : given 'n' no of children.
Each child has a rating value.
(n children & n rating values)

Our task : to distribute candies to each & every children keeping in mind the rating values.

2 conditions : → each child → at least one candy
→ children with higher rating than should get more candies than its neighbour.

Ex: [1 2 3]
candies: 1 2 3 = 6 → we return this.
↓
rating val of child 1.
(give minimum candies but make sure the condition satisfies)

arr = [1 3 2 1]
candy : [1 2 1 1] → did this distribution keeping in mind the left neighbours.
[1 3 2 1] → right neighbours. (considering)

to satisfy left & right: 1 3 2 1 = 7 → min candy we need to satisfy the two conditions.

Brute force → considering left → no left neigh ∴ only 1 | more than left neigh ∴ give more | more | less than left neigh ∴ give less | less | less | less | more | more | more | less | less | less

Ex:
arr = [0 2 4 3 2 1 1 3 5 6 4 0 0]
Left → considering the left neigh → 1 2 3 1 1 1 1 2 3 4 1 1 1
Right → considering the right neigh → 1 1 4 3 2 1 1 1 1 3 2 1 1 ← right start from here

left start from here

Ex: < greater than its left & also greater than its right.
(Typically try to assign the max of it)
max of left & right.

→ Think about left & think about the right neighbour
→ The first el will always have 1 (as it doesn't have left neighbour)

ans (max) → 1 2 4 3 2 1 1 2 3 4 2 1 1
= add all.

NOTE:
→ left's first el [0] will always be assigned 1.
→ right's first el [n-1] will always be assigned 1.
→ left → iterate in this dir^{n} → [1 - R]
→ right → iterate in this dir^{n} ← [R - 1]
→ answer should be max of left & right.
(addition of it all)

pseudocode

```
func ( ratings[ ] )
{
    left [n], right [n]

    left [0] = 1        right [n-1] = 1      // left
    ( for i = 1 → n-1 )
    {
        if ( ratings[i] > ratings of [i-1] )   // left neighbour
            left [i] = left [i-1] + 1;
        else                                   // not greater than left neigh.
            left [i] = 1
    }
```

doesn't have a left neighbour ← left [0] = 1 ; right [n-1] = 1 → doesn't have a right neighbour

(for i = 1 → n-1) → why from 1 & not 0 ? as 0th index will never have a left neighbour

```
else                                  → not greater than left neigh.
    left[i] = 1
}
// right
for( i = n-2     i >= 0 ; i-- )
    if (ratings[i] > ratings[i+1] )
        right[i] = right[i+1] + 1

    else
        right[i] = 1 ;
}                                     TC → 3 for loops
                                           O(3N)
Sum = 0 .                             SC → O(2N)
                                           └→ left & right array .
for( i = 0 → n-1 )
{
    sum = sum + max( left[i], right[i] )
}
return sum ;
```

⇓

further optimise .

⇓

we can do something like removing right array .

⇓

we had 3 operation .
→ compute left
→ compute high
→ summation of max of left & right .

→ we can compute right & take max at same iteration .

```
func( ratings[ ] )
{
    left[n], right[n]
← left[0] = 1      right[n-1] = 1

( for i = 1 → n-1 ) →
{
    if ( ratings[i] > ratings of [i-1] )
        left[i] = left[i-1] + 1 ;

else
    left[i] = 1
}
curr = 1 , right = 1 , sum = 0      we missed out adding n-1 th index
for( i = n-2     i >= 0 ; i-- )
```

curr = 1 , right = 1 , sum = 0

for (i = n-2 ; i >= 0 ; i--)

if (ratings[i] > ratings[i+1])

curr = right + 1 , right = curr ;

else

curr = 1 .

sum = sum + max [left[i] , curr]

}

return sum ;

we missed out adding n-1 th index

∴ sum = max (1 , left[n-1]) ;

TC → O(2N)
↓ 2 for loops

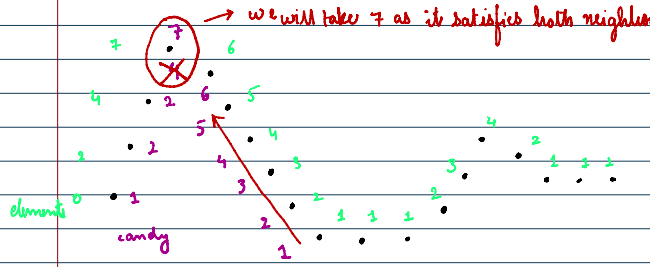
SC → O(N)

optimising the extra space (comp. O(N))

optimal approach

we will use concept of slope.

arr = [0 2 4 7 6 5 4 3 2 1 1 1 2 3 4 2 1 1 1]



→ we will take 7 as it satisfies both neighbour.

→ This is how the slope looks like.

if greater, give more candy (till increasing slope increment)

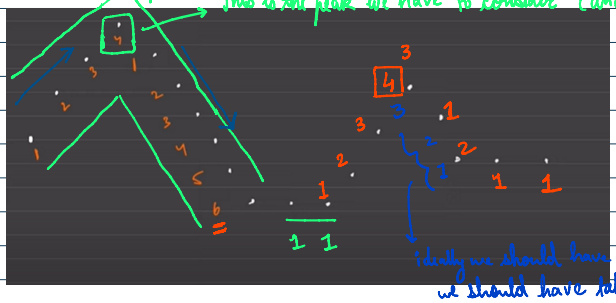
but we cannot go in the same dir ↘ for decreasing ∴ we go in opp dir^n .

1 2 3 4 3 2 1 → if we go it could give minus value.

-1

BUT

(also we don't care about distribution we care about summation ∴ we will go in the same dir^n ↘ (but start from 1 to avoid -ve)



This is the peak we have to consider (and it will take 7 not 4)

as long as ideal giving

ideal giving us 1 + 2 + 3 + 4 + 5 + 6

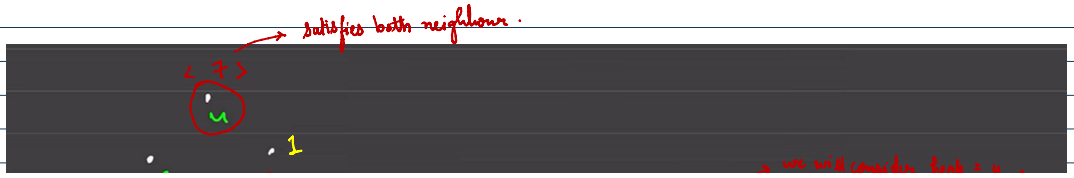
& our method some we don't care about distribution (just summation)

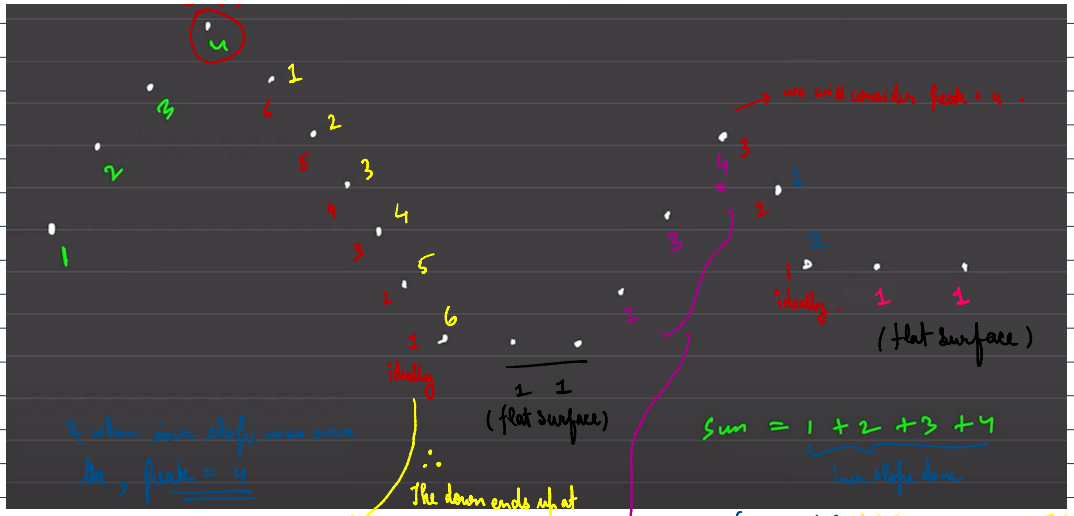
(for the peak value we will discuss below)

(as its diff for ↘ & ↖)
this dir for this dir.

ideally we should have done this & for peak out of 3 & 4 we should have taken 4 (but as long as we are getting 1 & 2) (meaning we care about summation not distribution) (we are accumulating sum not distribution)

→ The first element in increasing slope be 1 .





→ whenever we have flat surface we add 1 to answer .

pseudocode:

```
func ( ratings )
{
  sum = 1        i = 1 .
  while ( i < n )  → O(N)   → inner while loop doesn't matter ( inner while loop is making sure i is moving) because we are traversing every element once.
  {
    if (ratings[i] == ratings[i-1])   → if flat surface
    { sum = sum+1 ; i++ ; continue ;   → add 1 to sum. & do i++ & go ahead to next one.
    }
    // slope
    peak = 1   → assigned 1 to first value (0th index)
    // inc slope
    while (i<n && ratings[i] > ratings[i-1])   → increasing slope.
    {
      peak += 1 ; sum += peak ; i++ ;
    }
    // dec slope
    down = 1
    while ( i<n && ratings[i] < ratings[i-1] )   → decreasing slope.
    {
      sum += down , i++ , down++ ;
    }
    if ( down > peak )   → if down > peak
    {                      sum = sum + (down - peak)
      sum += down - peak ;
    }
  }
  return sum.
```

}
}
return ans;

TC → O(N)
SC → O(1)