Homework 4 - training problems

Pratyay Dutta (Codeforces ID : pdutt005)

February 23, 2024

Contents

1 Longest Unimodal Subsequence : Submission ID - 247809597

This problem is a variant of the LIS problem and the approach to solve this is very similar as well.
State : We specify two arrays as our dp arrays.
lis[i] = Longest Increasing Subsequence till element i in the array
lds[i] = Longest decreasing subsequence from element i to the end of the array.
Recurrence:

- For the LIS: We check for every element j in the array before element i. If any element arr[j] is less than arr[i] then it can be a part of the LIS till i and hence : lis[i] = lis[j]+1. For any other case, we dont update the lis array.
- For LDS : We check for every element j in the array after element i till the and of the array. If any element is arr[j] is greater than arr[i] then it can be a part of the longest decreasing subsequence from i to the end. Hence : lds[i] = lds[j]+1. For any other case,we dont update the lds array.

Base case : lis[0] = 1 : No other element before first element so lis ending at that element is just that element therefore length is 1.
lds[n-1] = 1 : Lds starting from the last element is 1.
In any case, we initialise our dp array to 1 because the minimum length ending at each element is the element itself so at least length has to be 1.
Answer : We have to find the element i where the sum of lis[i] and lds[i] is the max. The maximum unimodal subsequence length is $max(lds[i] + lis[i]) - 1$ checking for all i in range(0,n). We subtract 1 because we are counting the element i twice while adding the lengths.
Time Complexity : To create the dp arrays : For every cell i, we fill in, in the dp arrays, we make i comparisons(From 0 to i) for lis and n-i-1 comparisons for lds(from i+1 to n). Therefore, total nub. Therefore we make O(n) comparisons for every element. Therefore, to fill up both the arrays, we make $O(n^2)$ comparisons. To find the max, we go through the lis and lds arrays once and find the sum and find the max of the sum. This takes O(n) time. Therefore, total complexity of our algorithm = $O(n^2) + O(n) = O(n^2)$

2 Cake Cutting : Submission ID - 247729102

The cake is a circular object so in order to model it as a data structure, we need a circular data structure. Hence, in this implementation, we have used a circular list. In a circular list, any index¿length of the array, gets rounded back to th beginning and the counting starts again. Therefore, we can make a circular list easily by changing the way we access the indices.
For example, in the given figure, where the numbers outside the circle represent the indexes and the numbers inside represent the size of each cake element.

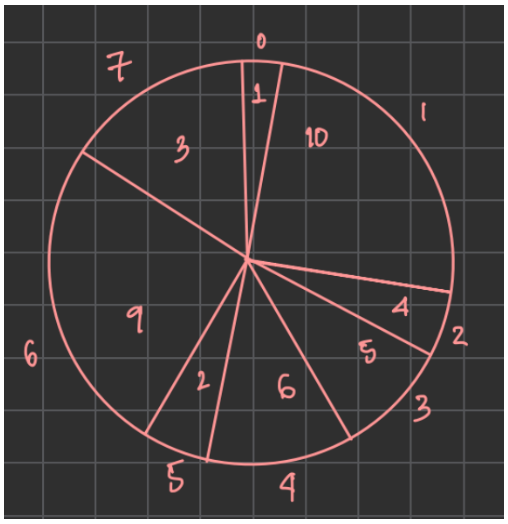

Figure 1: Example test case 2 demonstration

In this figure, if we want to access element 0 from element 7 then we have to add 1 to 7 and take the modulus of the value with respect to the number of cake slices. Therefore s[0] = s[(7+1)mod n]. If any element is lesser than the length, then the modulus will give the same value, but for any value overshooting the length, the modulus turns it back to the start. Therefore s[i] becomes s[i mod n]. Similarly, if we want to access element 7 from element 0, we will have to subtract 1 from the first index which makes it -1. An index cannot be -1, therefore, we add n to it which makes it -1+8 = 7. Therefore we establish two rules here:

- To access the element to the right of an element i, we will use s[(i+1) mod n].
- To access the element to the left of an element i, s[(i-1+n) mod n].

State : We define the state s[i][j] as the maximum amount of cake Preston can take if pieces i to j are available.
Recurrence : We create a list *slices* which contains the size of each piece i in slices[i]. When considering from the pieces of cake between i and j,there can be 2 cases:

1. When the length of the portion considered is 2. When there are 2 pieces from which Preston has to choose, he will always choose the one which is bigger(greedy choice) to maximise his value because the other one will be taken by Calvin and there will be no more pieces to choose from. Therefore if length==2: s[i][j] = max(slices[i], slices[j]).
2. When the length considered is more than 1. In this scenario, the greedy choice is not optimal. Two cases appear again:
    - Preston chooses ith piece : We check which piece is bigger among pieces i+1 and j because these are the only available pieces where Calvin can choose from, since always the pieces on the edges

are available for consideration. Whichever piece is bigger, Calvin takes it because we know he chooses greedily. So, if:

  - The i+1th piece is bigger, Calvin takes it and $left$ (a temporary variable) = slices[i] + s[(i+2) mod n][j] (Since, remaining pieces are from piece i+2 to j since i is taken by Preston and i+1 by Calvin).
  - The jth piece bigger. calvin takes it and $left$ = slices[i] + s[(i+1) mod n][(j-1+n) mod n] (i taken by preston and j taken by Calvin)

- Preston chooses the jth piece : We check between pieces i and j-1.
  - ith piece is bigger : Calvin takes it and $right$ = s[(i+1) mod n][(j-1+n) mod n] + slices[j](Preston takes j and Calvin takes i).
  - j-1th piece bigger : Calvin takes it and $right$ = s[i][(j-2+n) mod n] + slices[j] (Preston takes j and Calvin takes j-1)

s[i][j] = max(left,right). Choosing the max option after picking ith or jth item.

Base case : s[i][i] = slices[i]. Only one item which is picked.
Answer : We have stored the maximum value Preston can get after starting from each piece in s[i][(i+n-1) mod n] which means from i to the item right next to it on its immediate left. We find the maximum of all this values for i=0 to n-1. The maximum of all these values *maxcakevalue* gives us the answer of the maximum amount of cake Preston can get from a given cake cut unevenly.
Time complexity : For every starting point i, we are checking for all possible lengths from 2 to n. For each given length and starting point i, we are checking twice for each item Preston picks. For each item Preston picks, we do a check once i.e in O(1) time. Therefore, for a given length and starting point we make O(1) operations. There are n different lengths to consider starting from each point i. Therefore:
For each starting point i : O(n)
Therefore, for all starting points : $O(n^2)$. Time complexity to find the answer from the dp matrix = O(n), we check for all possible starting points and for the full length each time.
Total time complexity : $O(n^2)$