Homework 2 - training problems

Pratyay Dutta (Codeforces ID : pdutt005)

January 25, 2024

Contents

1 Training problems

1.1 Merge the candies (Submission ID : 242804089)

Explanation: We employ a greedy strategy to solve this problem. To get the minimum number of counts, we always have to merge the two smallest piles of candies. To implement this idea, we first used a very naive approach. Till the array is empty, we:

- Take out the two smallest elements.
- Update the sum
- Insert the sum into the array
- Sort the array

However, the best a sorting algorithm can do is nlogn time where n is the size of the array. We are doing this O(nlogn) operation n times till our array runs out. This means the total time of our algorithm was $O(n^2 log(n))$
To optimize our implementation while keeping the idea intact, we needed to do something which would sort the new array formed after introducing the sum of the previous two elements dynamically in less than O(nlogn) time. Therefore, we build a minimum heap.
We first take our unsorted array of the size of piles and make a min heap. According to the same logic almost, we:

- We take out the minimum element and heapify our heap.
- We take out the second smallest element and heapify our heap.
- We calculate the sum and update our count
- We insert the sum into our heap.
- We return the value $count * 2$ since we are counting it twice once before merging and once after.

Time Complexity : Deleting minimum element from a heap and heapifying the new heap takes O(logn) time. Inserting a new element into the heap in order to preserve the property of the heap takes O(logn) time. Therefore, for each iteration, we do computations which are O(2logn)+O(logn) = O(logn). We do these computations till our array runs out of elements. Therefore we do these computations n times.
Therefore, total complexity of our algorithm = O(nlogn)
This is clearly an improvement over the naive method.
Algorithm:

```
function MERGE(arr)
    count ← 0
    heapify(arr)
while len(arr) > 1 do
    i ← remove(arr)
    j ← remove(arr)
    s ← i + j
    count ← count + s
    insert(arr, s)
    return 2 * count =0
```

1.2 Number of candles (Submission ID : 242830247)

I used a dynamic programming approach to this problem as i found it very similar to an unlimited knapsack problem[1].
Instead of initialising the dp matrix with zeros, we initialise it with an empty string so that we can append the new candle as a string to keep track of the length of the string because the longer the length the higher the age of the individual.
For every capacity from 1 to the given weight, we traverse through all the elements and see if adding the new candle to the existing candles satisfies the following two conditions:

- If the candle under question is of higher value than the existing value of the last candle we considered. This makes sure that the first digit of the formed candle sequence is highest which will give us the highest age.
- If the previous condition is satisfied, we check if adding the new candle increases the length of the candle sequence. This makes sure that we consider the largest length sequence all the time which is going to maximise the value of the age formed by the candles.

If both the conditions are satisfied, we add the candle under question to the previous sequence of questions.
Time complexity: There are two for loops in our implementation of filling up the dynamic programming array. The first for loop runs from 1 to the given capacity W and the nested for loop runs for each of the candles that we have i.e n=9. So the time complexity of our algorithm is O(nW) and since n=9 always, we can say that our algorithm has time complexity = O(9W) = O(W).

Algorithm 1 Maximize age

```
function MaxValue(capacity, weight)
    n ← 9
    value ← [1, 2, 3, 4, 5, 6, 7, 8, 9]
    dp ← array of empty strings of length capacity + 1 for w ← 1 to capacity do
        for i ← 0 to n − 1 do
            if weight[i] ≤ w then
                wi ← dp[w − weight[i]] + str(value[i])
                wo ← dp[w] if wi ≠ '' then
    return dp[capacity][:: −1]
```

References

[1] Cormen, Leiserson, Rivest, and Stein Introduction to algorithms (CLRS). Third Edition Section 16.4, Lemma 16.7