**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Fall 2019**

# Homework #3

October 24, 2019

*Name: Himesh Buch* *Extension: No*

**Problem 1.** Recall that in the knapsack problem, we are given $n$ items with positive integer weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack of size $W$; we want to pick a subset of items with maximum total value that fit the knapsack, i.e., their total weight is not larger than the size of the knapsack. In the class, we designed a dynamic programming algorithm for this problem with $O(nW)$ runtime. Our goal in this problem is to design a different dynamic programming solution.

Suppose you are told that the *value* of each item is a *positive integer* between 1 and some integer $V$. Design a dynamic programming algorithm for this problem with worst case $O(n^2 \cdot V)$ runtime. Note that there is no restriction on the value of $W$ in this problem. **(25 points)**

**Solution.** Solution to problem one goes here.

---

**Problem 2.** You are given a set of $n$ boxes with dimensions specified in the (length, width, height) format. A box $i$ fits into another box $j$ if *every* dimension of the box $i$ is *strictly smaller* than the corresponding dimension of the box $j$. Your goal is to determine the *maximum length* of a sequence of boxes so that each box in the sequence can fit into the next one.

Design an $O(n^2)$ time dynamic programming algorithm that given the arrays $L[1:n]$, $W[1:n]$, and $H[1:n]$, where for every $1 \leq i \leq n$, $(L[i], W[i], H[i])$ denotes the (length, width, height) of the $i$-th box, outputs the length of the longest sequence of boxes $i_1, \ldots, i_k$ so that box $i_1$ can fit into box $i_2$, $i_2$ can fit into $i_3$, and so on and so forth. Note that a box $i$ can fit into another box $j$ if $L[i] < L[j]$, $W[i] < W[j]$, and $H[i] < H[j]$ (you are *not* allowed to rotate any box). **(25 points)**

**Solution.**

**Objective:** To design an algorithm that can identify the number of boxes that can fit into each other

**Algorithm:**

- The base case is when there are no boxes, meaning n = 0, in that case we return 0
- If no box can fit into each other we will return 1
- if n $\neq$ 0, then we can recursivley call the algorithm and check if the length, height, and width of boxes are less than the other box which fits in it. We will make sure that we compare the length with length, height with height, and width with width.

  **Recursive formula:**

- **Specification:** if the dimensions of first box is lesser than the second box, then only first box will fit in second box and so on. While comparing the dimensions, we have to make sure that we compare the corresponding sizes. Hence, the goal is to find the maximum length of sequence/maximum number of boxes in the given input

- **Recursive solution:**

$$Box(ans, arr, b, l) = \begin{cases} 0 & \text{; if } i = 0 \\ 1 + Box(ans, arr, b+1, l) & \text{; If the value is greater than below cell} \\ max(1 + Box(ans, arr, b+1, l), Box(ans, arr, b, l+1)) & \text{; If the value is greater than left cell} \end{cases} \tag{1}$$

**Proof of Correctness:**

- We will try to utilize information from the Longest increasing subsequence algorithm we did in class
- As we can see from the recursive formula, we have three cases, ans contains the result and it will be updated after every recursive call, and along with that, we will have two more variables b (represents below) and l (represents left) of the item which will also be updated in every recursive call
- we have two main cases here. We represent the array of dimensions as a 2D matrix, and we compare the cell below and the cell to the left with the main cell

1. Check the value of the cell below. If the value is greater than below cell, we will recursively call the function again and increase b by one and add 1 to the entire result
2. Check the value to the left of the cell. If the value is greater than left cell, we will recursively call the function again and increase l by one and take the maximum of result found above and make another recursive call

- The entire algorithm here depends on n, which is the number of boxes. or the size of our input array ("arr" in this case), and the two recursive calls on a 2D matrix makes it an O($n^2$) algorithm

**Memoization:**

- since we are representing the input array as a matrix, we have p and q as its dimensions
- Box(ans, arr, b, l):

*if no boxes can fit in each other then we return 1*

1. if (x == a - 1 or y == b - 1) then ans = 1

   *this is the case for checking below cell*

2. if (b + 1 < p and arr[b][l] < arr[b + 1][l]) then ans = 1 + Box(ans, arr, b + 1, l)

   *this is the case for checking left cell*

3. if (l + 1 < q and arr[b][l] < arr[b][l + 1]) then result = max(1 + Box(ans, arr, b + 1, l), Box(ans, arr, b, l + 1))
4. return ans

---

**Problem 3.** You have a bag of $m$ cookies and a group of $n$ friends. For each of your friends, you know the "greed factor" of your friend (denoted by $g_i$ for your $i$-th friend): this is the minimum number of cookies you should give to this friend to make them stop complaining. Of course, you would like to find a way to distribute your cookies in a way to minimize the number of your friends that are still complaining.

Design an $O(m + n)$ time greedy algorithm to find an assignment of the cookies to your friends so as the minimize the number of the friends that are still complaining: recall that a friend $i$ stops complaining if we assign them $c_i$ cookies and $c_i \geq g_i$. **(25 points)**

**Solution.**

**Objective:** To design an algorithm that can efficiently provide cookies to each friend and if can't, provide the cookies that way so that least of the friends complain

**Algorithm:**

- We will first need to sort the array of greed factor, which is basically an array of how many cookies required by each friend
- Once we have the sorted array, we will look at m, and there could be three possibilities for m as compared to gi

1. m = 0; in that case we will return null, as every friend will be complaining
2. m > gi; in that case, we will return the entire array, as each friend will get the cookies they asked for
3. m < gi; this will be the most comprehensive case, as we have to figure out a way so that least number of friends complain. Since, m < sum(gi), we will not be able to satisfy each friend's demand. So, we will start from the begining and give out cookies to each friends untill no cookies are left. In the end, we will return those friends who get the cookies

**Proof of Correctness:**

- As explained in the algorithm above, we will have to sort the array of greed factor first
- Since the entire algoritm should be done in more than linear time, we will use counting sort for this
- Sorting of the array is important because we need the least friends complaining. Because of that, if first friend requires more than any other friends, we will end up having more amount of friends complaining which is not the goal here
- Once, the array is sorted, we will loop through (seperate loop, not nested) it and give out cookies to friends
- We will have three cases here. The first one is us having no cookies, in that case we will return null because we won't be able to satisfy any friend's demand. In second case, if we have more cookies than required, we will be able to satify each friend's requst and hence return the entire array, as each friend will be satisfied
- In the case of us having less cookies than the total needed, we will loop through the sorted array of greed factor and give out cookies to friends. Now, we will be going from least cookies required than to most. We will keep doing it until we get a friend's demand which is larger than the total cookies left (break the loop in that case and return the remaining indices of friends)
- While looping through, after giving out cookies to friends, we will keep deducting that amount from the total number of cookies, we basically be comparing that amount with the greed factor of the next friend

**Time complexity:**

- As explained in the algorithm above, we will have two loops. One to sort the array of greed factor and the other to hand out cokies to friends
- Sorting takes O(n + k), where k is the non-negative range of the output array.

- The second loop basically compares each element from the sorted array with cookies and deducts gi from the number of cookies, the loop breakes if gi > (number of cookies left), which will take O(m) time
- Hence, the entire algortihm will take O(m + n) time.

---

**Problem 4.** There is a straight highway with $n$ houses alongside it. You have developed a brilliant new product and would like to advertise it by placing billboards alongside this highway. Since constructing billboards is expensive, you would like to choose as few billboard locations as possible such that every house is within at most $d$ units of a billboard.

Design an $O(n \log n)$ time greedy algorithm which given an array $A$ of real numbers representing the locations of the houses alongside the highway (house one is placed on location $A[1]$, house two on $A[2]$, etc.), and a maximum distance $d$, outputs a *minimum* set of locations (i.e., real numbers) for placing the billboards such that each house is at most $d$ units from some billboard. **(25 points)**

**Solution.**

**Objective:** To design an algorithm that can find the least amount of billboards that needs to be placed so that every house can see it

**Algorithm:**

- We will have to sort the array of houses first and we will use merge sort for that
- Once we have a sorted array of houses, we will randomly select a pivot and divide the array into parts
- There could be two possibilities, either the difference of d and house location is greter than d or it is less than or equal to d
- If the difference of d and house location is less than or equal to d, then we can find the range of that array (from index 0 to pivot or randomly chosen house location) and divide it by 2, and that is where we put the billboard
- If the difference of d and the house location is greater than d, then we move one element left and check again. If it is less than d now, then we can again take the range of that array and divide it by 2 and place the billboard there. If we can't find any element for which the difference is not less than d, then again we find the range of that array, divide it by 2 and see if the first element is less than or greater than the result of (range)/2. If it is less than the first element, then we place the billboard at (first element) + (range/2), if it is greater than that then we need to assign separate billboard for all the houses in that array

**Proof of Correctness:**

- The first step of the algorithm is to sort the houses of array. It will not be possible to design an alogorithm in O(nlogn) time if the house locations are not sorted
- Once we have the sorted array, we can pick a location at random which will be our pivot. Picking a pivot will be done in constant time
- In order to get the required time complexity, we will trim down the array and look at the left of the pivot first and repeat the same process for the other/right side

- We will check if the difference of d and house location (pivot or otherwise) is less than or equal to d. If so, we can take the range of that array and divide that number by 2 (we will make sure it is an integer), which will be the location of the billboard, or for all of those houses, the number (range/2) will be the billboard location
- If the difference of d and house location (pivot or otherwise) is not less than d, then we will move our pivot one to the left and repeat the same process as above
- If we can not find any element for which the difference of d and house location (pivot or otherwise) is less than or equal to d, then we will take the range of that array and divide it by 2 (we will make sure it is an integer), and check if that number is less than d. If so, that is where we place the billboard. But if that number is not less than d, then we will place seperate billboard for all the houses
- By this way, we will satisfy the requirement

**Time complexity:**

- As explained in the algorithm above, we will sort the array first using merge sort, and it will take O(nlogn) time
- Now, just like binary search algorithm, we will find a pivot (O(1) time), and divide the array into a subset or a smaller arrays.
- We will keep dividing the array until the billboard are placed properley. Dividing array in halves and checking for the proper position of billboard will take O(nlogn) time
- As mentioned in the algorithm, finding range and dividing it by 2 will be done in constant time
- So, as a whole the entire algorithm will take O(nlogn) time

---

**Challenge Yourself.** Let us revisit Problem 4. Instead of minimizing the number of billboards so that every house is within distance $d$ of some billboard, we now have a budget of $b$ billboards in total. Our goal is again to place the $b$ billboards in proper locations such that the *maximum* distance of every house from some billboard is *minimized*.

Design an algorithm for this problem that given the locations of the houses in the array $A[1:n]$ and the budget $b$, outputs an assignment of these billboards to the locations so that the maximum distance, among all houses, from their closest billboard is minimized.

*Example:* If the houses are placed in $[1, 5, 2, 7, 11]$ and $b = 2$, we can place one billboard at location 3 and another at location 9 and the answer would be 2 – every house is within distance 2 of some billboard. For the same houses, if instead we have $b = 1$, we can place a billboard in location 6 and have the distance of at most 5 for every house.