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

Homework #3 Solution

October 27, 2019

Instructor: Sepehr Assadi

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. *Specification (in plain English):* For every $0 \leq i \leq n$ and $0 \leq j \leq \sum_{i=1}^{n} v_i \leq n \cdot V$, we define:

- $S(i, j)$: the *minimum* size of a knapsack needed to collect j values from a subset of items $\{1, \ldots, i\}$.

Having this, the answer to our problem is the maximum value of j such that $S(n, j) \leq W$. This is because, for any $j' > j$, $S(n, j') > W$ implies that we need a knapsack larger than W and so j' cannot be the answer.

Recursive formula: We write the following recursive formula for $S(i, j)$:

$$S(i,j) = \begin{cases} 0 & \text{if } j = 0 \\ +\infty & \text{if } i = 0 \text{ but } j > 0 \\ S(i-1, j) & \text{if } v_i > j \\ \min\{S(i-1, j), S(i-1, j - v_i) + w_i\} & \text{otherwise.} \end{cases}$$

The proof of correctness of this formula is as follows. For the base cases: when $j = 0$, we do not need to collect anymore value and hence a knapsack of size 0 would be enough. When $i = 0$ but $j > 0$, we still need to collect some value but now there is no item left; this means that no matter what size of a knapsack we bring, we cannot solve the problem at this point, thus $+\infty$ is a correct answer (another option would be to return 'impossible' here and then make sure that when we are updating other values of $S(i, j)$, whenever taking minimum between 'impossible' and another number, we return that number, and so on. However, by using $+\infty$ we automatically achieve this and so we do not need to worry about it).

For the other two cases: If $v_i > j$, we should not pick this item because our goal is to pick *exactly* j values, so we should simply go to the first $i - 1$ items and among those, pick the best solution, captured by $S(i - 1, j)$. When $v_i \leq j$, we have two options: (1) either ignore item i and pick the best solution among remaining items which would be $S(i - 1, j)$, or (2) pick i which decreases our target value to $j - v_i$ so we only need to look for a solution for $S(i - 1, j - v_i)$ but now we also need to account for the weight of item i which is w_i, so this option costs us $S(i - 1, j - v_i) + w_i$. As our goal is to minimize the size of knapsack, we should return minimum of these two options.

Algorithm: We can turn this formula into an algorithm using either memoization or bottom-up dynamic programming (as per instructions, we do not need to write the resulting algorithms however). Either way, there are at most $(n + 1) \cdot (n \cdot V + 1)$ subproblems and each one takes constant time to compute, so solving all $S(i, j)$ problems takes $O(n \cdot (nV)) = O(n^2V)$ time. Computing the final solution from these subproblems also takes at most $O(nV)$ time to go over all values of j, hence leading to an $O(n^2V)$ time algorithm overall.

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. We first do a preprocessing step: we sort the boxes in non-decreasing order of their volume (sorting based on area or any of the dimension will do the trick – one can even skip the sorting part entirely and write a slightly different dynamic programming but we will not cover that in this solution). Then, we define the following dynamic programming solution.

Specification (in plain English): For every $0 \leq i \leq n$, we define:

- $B(i)$: the *maximum length* of a sequence boxes from $\{1, \ldots, i\}$ so that each box in the sequence can fit into the next one and the last box of the sequence is box i itself.

The answer to our original problem is then the maximum value of $B(i)$ for i ranging from 1 to n. This is because any optimal sequence of boxes should end in one of the n boxes, say box j, and since we have sorted the boxes, we know that that sequence cannot contain any box from $\{j+1, \ldots, n\}$ as box j does not fit into any of these boxes (this is why any reasonable sorting measure would work as long as it guarantees that a box i does not fit into a box $i+1$).

Recursive formula: We write the following recursive formula for every $0 \leq i \leq n$ (almost identical to that of the LIS problem):

$$B(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max \begin{cases} \max \{B(j) + 1 \mid 1 \leq j < i \text{ and } L[j] > L[i] \ , \ W[j] > W[i] \ , \ H[j] > H[i]\} \\ 1 \end{cases} & \text{otherwise} \end{cases}$$

The proof of the correctness is as follows: the base case of $i = 0$ holds since there is no box. For the other case, we have the following options to pick a sequence that ends in box i from $\{1, \ldots, i\}$: we should pick a box $j \in \{1, \ldots, i-1\}$ such that box i fits inside box j (all its dimensions are strictly smaller) and in that case, we should find the longest sequence ending in box j which is $B(j)$. Hence, taking the maximum over all valid choices of j in the formula ensures the correctness.

Algorithm: We can again turn this formula into an algorithm using either memoization or bottom-up dynamic programming (and again we do not need to write the resulting algorithms for this homework). Either way, there are $(n+1)$ subproblems and subproblem $B(i)$ takes $O(i)$ time to go over all indices $j < i$. Similar to LIS, adding the runtime of all subproblems results in runtime of $\sum_{i=1}^{n} c \cdot i = O(n^2)$. There is also the step for returning the final answer which can be done in $O(n)$ time and a sorting step which takes $O(n \log n)$ time using, say, merge sort. Hence the total runtime is $O(n^2)$.

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. *Algorithm:*

1. Iterate over the list of friends and insert any friend i with greed factor $g_i \leq m$ inside a new array A (we ignore all friends with greed factor $> m$).
2. Sort the array A in increasing (non-decreasing) order using counting sort.
3. Let $t \leftarrow m$. Iterate over the array A and if $A[i] \leq t$, allocate $A[i]$ cookies to this friend and update $t \leftarrow t - A[i]$; otherwise, if $A[i] > t$, terminate the algorithm (t is the number of remaining cookies).

Proof of Correctness: An obvious observation is that minimizing the number of the friends that are complaining is equivalent to maximizing the number of friends that are "satisfied", i.e., no longer complain. So we focus on the second task instead. Another simple observation is that since we only have m cookies, removing any friend i with $g_i > m$ (as done in the first step) does not change the optimal solution as anyway we could not satisfy those friends. Hence, focusing on the friends in array A instead of all friends is without loss of generality. We now prove the correctness of the algorithm using an exchange argument (there are multiple ways to do an exchange argument for this problem; we simply pick one of them here).

Let $G = \{x_1, x_2, \ldots, x_k\}$ denote the name of the friends that were satisfied by the greedy algorithm, sorted in increasing (non-decreasing) order of their greed factor, so $g_{x_1} \leq g_{x_2} \leq \ldots \leq g_{x_k}$. Let $O = \{o_1, o_2, \ldots, o_\ell\}$ denote the name of the friends that are satisfied in *some optimal* solution, again sorted in increasing (non-decreasing) order of their greed factor so $g_{o_1} \leq \ldots \leq g_{o_\ell}$ (note that we cannot assume $k = \ell$; this is in fact precisely what we want to prove in the first place).

We now show how to exchange O to G. Let j be the first index where O and G differ, i.e., $x_1 = o_1, \ldots, x_{j-1} = o_{j-1}$ but $x_j \neq o_j$. Since both G and O are sorted in increasing order of their greed, and by definition of the greedy, we know that $g_{x_j} \leq g_{o_j}$. We can thus take the cookies given to o_j in O and instead give them to x_j and make x_j satisfied. This means that $O' = \{x_1, \ldots, x_j, o_{j+1}, \ldots, o_\ell\}$ is also a valid solution and since size of O' is equal to size of O, it is also optimal.

We can thus continue doing the exchange with index $j+1$ and so on until we entirely exchanged O to G; in particular, note that once we exchanged x_k and o_k, we can be sure that there are no friends left in the resulting solution since by definition of the greedy algorithm, the reason we did not satisfy some friend x_{k+1} was that we ran out of cookie for satisfying *anyone* else and hence this new solution cannot also satisfy anyone after x_k. This means that $\ell = k$ and G is also optimal.

Runtime analysis: The first line of the algorithm can be done in $O(n)$ time. After removing the friends with greed factor larger than m, the maximum number in A is at most m and hence counting sort, sorts this array in $O(n+m)$ time. The final step also takes $O(n)$ time in total leading to total runtime of $O(n+m)$.

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. We say that a billboard b *covers* a house a if $|a - b| \leq d$. A house is *covered* if some billboard covers it, and is *uncovered* otherwise.

Algorithm. The algorithm can be described in one sentence: while some house is uncovered by a billboard, find the left-most uncovered house y and place a billboard at $y + d$. An efficient implementation of this algorithm is as follows.

1. Sort the input locations A in non-decreasing order.
2. Initialize a linked list B for the output and insert $A[1]$ to B.
3. For $i = 2$ to n: if $A[i] - B.last > d$, append $A[i] + d$ to B.
4. Return B.

Proof of correctness. Let $G = \{x_1, \ldots, x_b\}$ be the solution output by the algorithm above, let $O = \{y_1, \ldots, y_c\}$ be an optimal solution, and let $a_1 \leq \cdots \leq a_n$ be the sorted house locations. The algorithm's output satisfies $x_1 \leq \cdots \leq x_b$ and without loss of generality we assume $y_1 \leq \cdots \leq y_c$. Note that we are *not* allowed to assume $b = c$ (indeed this is what we want to prove!). We now use an exchange argument.

Let i be the first index for which $x_i \neq y_i$. Let j be the smallest index such that house j is uncovered by the first $i - 1$ billboards in G (which are the same as the first $i - 1$ billboards of O by definition of i). Our algorithm sets $x_i = A[j] + d$, and since y_i must also cover $A[j]$ (otherwise O will not be a correct solution – remember that O is also sorted), we know $y_i \leq A[j] + d$. Thus $y_i \leq x_i$, and we can safely exchange x_i for y_i without leaving any houses uncovered (all houses on the left of j are already covered).

We can continue doing this until we entirely exchange O with G. In particular, since every billboard in G covers at least one house not covered by the other billboard, it follows that after exchanging everything up until y_c, we are done (namely, there cannot be a billboard at x_{c+1} since this billboard will not cover any house and so greedy would have not picked it). Hence, G is also an optimal solution.

Runtime analysis. Sorting the input array takes $O(n \log n)$ time. In Step 3, the algorithm performs a single pass over the input array and does constant work in each iteration, taking a total of $O(n)$ time. Thus the algorithm runs in $O(n \log n)$ time.