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

Lecture 11

October 14, 2019

Instructor: Sepehr Assadi

1 Greedy Algorithms

In the previous lectures, we learned a strong technique for algorithm design: dynamic programming. A key feature common to almost all dynamic programming algorithms is that the algorithm one way or another examines *all* possible options for solving a problem: even though storing the intermediate solutions helps us with not revisiting these options again and again, we still need to consider these options *at least once.*

We now consider a different technique for algorithm design that deviates from this general approach: *the greedy algorithms* – we no longer examine the space of all possible options and instead *greedily* decide to simply pick some of the options and fix them without ever considering the alternatives.

Before we move, let us make an important remark: this approach *rarely works* and more often than not, greedy algorithms are actually wrong! However, in this lecture and the next one, we are going to see some examples of greedy algorithms that *do work* – the main objective here is to learn *when* should we expect greedy algorithms to work and more importantly, *how* can we make sure our greedy algorithm indeed works (hint: this involves proving the correctness of the algorithms!).

2 Knapsack Problem with Unit Weights

Let us consider a twist on the knapsack problem by assuming the weights of all items are *equal to one.*

Problem 1 (**Knapsack problem with unit weights**)**.** We consider the following problem:

- **Input:** A collection of n items where item i has a weight of 1 and positive value v_i plus a knapsack of size W.
- **Output:** The maximum value we can obtain by picking a subset S of the items such that the total weight of the items in S is at most W, i.e.,

$$\max_{S \subseteq \{1,\ldots,n\}} \sum_{i \in S} v_i$$
$$\text{subject to} \quad \sum_{i \in S} 1 \leq W.$$

We can certainly use dynamic programming to solve Problem 1 in $O(nW)$ time– after all, this is only a special case of the more general problem of knapsack we already studied. However, dynamic programming seems like an *overkill* here: both the algorithms is rather complicated (not really though!) and more importantly it can have a very large runtime (it is $O(nW)$ time; with a simple twist, we can make it $O(n^2)$ also on a knapsack problem with *unit* weights; do you see how?) Let us see how greedy algorithms can help.

A greedy algorithm for knapsack with unit weights: There is a simple observation here that suggests a greedy strategy should indeed work for this problem: between items i and j, if value v_i of item i is larger than value v_j of item j, then item i is *better than* item j *literally in every aspect* (since both have weight of 1). In other words, there does not seem to be any reason to even consider picking item j over item i. Let us turn this observation into an algorithm.

Algorithm:

1. Sort the items in the *decreasing* (more accurately, non-increasing) order of their values.
2. Pick the W items with largest value in the knapsack, i.e., the first W items in this sorted order.

We now prove the correctness of this algorithm. Even though proving correctness of algorithms is always an important task, for greedy algorithms it is certainly *the most important step*: most greedy algorithms are wrong and so unless we can prove the correctness of our greedy algorithm beyond any doubt, we should never trust a greedy algorithm.

The typical way we can prove the correctness of a greedy algorithm is to use an **exchange argument**: roughly speaking, we show that if we pick an optimal solution and the solution output by the greedy algorithm, we can start *exchanging* the elements of the optimal solution to the elements of the greedy algorithm *without hurting* the value of the solution: hence, by switching the elements one by one, the optimal solution changes to the greedy solution, without ever hurting its value: this can only mean that the greedy solution is also optimal for the problem!

Let us see this general approach in action in the context of the knapsack problem with unit weights.

Proof of Correctness: Firstly, we can assume that $W < n$; if not, i.e., when $W \geq n$, the greedy algorithm correctly picks every item in the knapsack (as they all fit together) and so its solutions is clearly correct.

We now prove the correctness of the algorithm via an exchange argument:

- Let $G = \{g_1, g_2, \ldots, g_W\}$ be the greedy solution. Note that by design, $v_{g_1} \geq v_{g_2} \geq \cdots \geq v_{g_W}$.
- Let $O = \{o_1, o_2, \ldots, o_W\}$ be any optimal solution to the problem; we sort the items picked by the optimal solution also in the non-increasing order of their values so that $v_{o_1} \geq v_{o_2} \geq \cdots \geq v_{o_W}$.

  (we can be sure that any optimal solution picks exactly W items since every item has exactly weight 1: if it picks any less, by adding any item to that solution which still fits the knapsack we can increase the value contradicting the fact the original solution was optimal; it can also never be the case that any valid solution picks more than W items as they will not fit the knapsack).

We are now going to start exchanging O into G without decreasing the value of the solution. The way to do this is to find the *first place* where the solution G and O differ from each other. More formally, we let j be the *index* such that $g_1 = o_1, g_2 = o_2, \cdots, g_{j-1} = o_{j-1}$ but $g_j \neq o_j$ (note that j can be equal to 1; also, if we cannot find such a j, then it means that $G = O$ and we are done). In other words, we can write $O = \{g_1, \ldots, g_{j-1}, o_j, o_{j+1}, \ldots, o_W\}$ (because $g_1 = o_1, g_2 = o_2, \cdots, g_{j-1} = o_{j-1}$).

Let us now examine what happens if we exchange o_j with g_j in O to obtain a solution O',

$$O' = \{g_1, \ldots, g_j, o_{j+1}, \ldots, o_W\}.$$

We would like to argue that the value we obtain by picking the items in O' is *at least as large as* the value we obtain by picking the items in O. To prove this, we only need to show that $v_{g_j} \geq v_{o_j}$ because all other items are the same in O and O'. To argue $v_{g_j} \geq v_{o_j}$, we have,

1. For all items $i \notin \{g_1, \ldots, g_{j-1}\}$, $v_{g_j} \geq v_i$: this is because in the greedy algorithm, when picking g_j, we in fact pick the item with maximum value among the items not picked yet (by sorting them beforehand).
2. $o_j \notin \{g_1, \ldots, g_{j-1}\}$: this is because $\{g_1, \ldots, g_{j-1}\} = \{o_1, \ldots, o_{j-1}\}$ by definition of the index j and o_j clearly does not belong to the second set, hence, it cannot belong to the first set as well.

3. By combining (1) and (2), we have $v_{g_j} \geq v_{o_j}$.

This implies that the value of O' is at least as large as value of O. So O' is also another optimal solution (note that value of O' cannot be strictly larger than O since O is an optimal solution; hence, we actually have that value of O and O' are equal).

We are now almost done. We can continue the previous exchange, this time with G and O', and switch the item $j+1$, and continue like this until we switch all the items originally in O to become the items in G; the important thing is that throughout the whole process, we never decrease the value of the intermediate solutions, hence at the end, once we entirely exchanged O with G, we still have that the value of items in G is at least as large as the items in O. This implies that G is also an optimal solution, hence proving the correctness of the algorithm.

Remark: The above argument is yet another "induction in disguise" (similar-in-spirit to the case of proof of correctness of dynamic programming). This is because to formally show that we can exchange O with G, we show that if we already exchanged the first $j-1$ items, we can also exchange the j-th item without decreasing the value (this corresponds to the induction step; the induction base is for the "0-th" index, i.e., when we have not exchanged any item). The induction hypothesis is also that throughout these exchanges, we never decrease the value of the solution.

Again, similar to the case of dynamic programming, we actually do not need to *explicitly* call the above approach induction every time, since the induction hypothesis and induction step are almost always the same: the hypothesis says that we can exchange the optimal solution to the greedy solution one step at a time without hurting the value, and the step (using logical statements and other tools) proves that if we have exchanged the first $j-1$ part of the solution, we can also exchange the j-th part (nevertheless, it is still worth knowing that again we are simply doing induction in this proof).

Runtime Analysis: Finally, we should analyze the runtime of the algorithm. We can use any fast sorting algorithm, say, merge sort, to sort the items in the first step. This takes $O(n \log n)$ time. The second step can be done in $O(n)$ time, making the total runtime of the algorithm $O(n \log n)$.

This concludes the analysis of our first greedy algorithm in this course.

Before we move on, let us quickly remind the reader that in Lecture 8, we showed a simple example where a more general version of this greedy algorithm *fails* to work for the original knapsack problem. Roughly speaking, the general greedy algorithm aimed to sort the items based on their value/weight ratio and continue picking the items in this order. However, it is easy to see that when we have general weights, we can no longer directly compare two items i, j based on their value/weight ratio: even if $v_i/w_i > v_j/w_j$ (so per one unit of weight, i worths more than j) it is not true that item i is strictly better than item j, since w_i can be much larger than w_j and hence item i may not fit the knapsack in certain cases, while item j still does. As such, our original observation no longer holds for this case (you are encouraged to follow the proof above and see where it "breaks" for this new greedy algorithm in the general case).

3 The Pattern of Greedy Algorithms

We are going to see many more greedy algorithms in this course, so it is worth mentioning what is typically the general pattern between all of them:

1. Make an *observation* about the problem that allows you to *ignore* some of the options available to you entirely. At this point, you are simply building intuition and not proving anything.

2. Design your greedy algorithm based on this observation.

3. The main step is to prove the correctness of the greedy algorithm. This is typically (but *not* always) done via an *exchange argument*:

   (a) Fix an optimal solution that is different from the greedy solution.

   (b) Find the “first” difference between this solution and the greedy solution.

   (c) Show that exchanging the choice of the optimal solution with the choice of the greedy solution does not hurt the value of the solution (we will be done now because we repeat this argument again and again until the optimal solution is exchanged to the greedy solution entirely).

4. If you failed to prove the correctness of the algorithm, go back to the first step and make a new observation, use that to design a new algorithm, and repeat – alternatively, entirely give up on using a greedy solution and go with dynamic programming or some other technique; it may very well be the case that the problem you are solving does not have a greedy solution (do *not* try to “force” a greedy solution to a problem that does not have such a solution).