Homework 4 (15 pts)[[1]](#footnote-1)

1. [5 pts]: Suppose that you have two programs that work correctly.
   * Program A solves the 0-1 knapsack problem with no replacement.
   * Program B solves the 0-1 knapsack problem with unlimited replacement.

At a very high level, how would you solve the 0-1 knapsack problem with limited replacement? This is the problem where you have a finite, known number of each type of item. Describe an answer that either involves modifying one of the programs at a high level, and/or modifying the input data.

***Solution:***

There are a couple very good answers here, and I don’t have a preference between them.

One is to use the 0-1 knapsack program without replacement, and change the data so that each copy of the same item is considered as a separate item.

The other is to use the 0-1 knapsack program with replacement, but keep track separately of the number of remaining items of each type. When you run out of some type of item, you remove it from the list of available items.

1. [5 pts]: Consider the standard greedy algorithm for making change: give the user change by giving them as many as possible of the highest denomination coin or bill, then as many as possible of the next highest coin or bill, etc. We know that this will always give correct change (assuming that there is a 1-“cent” coin defined). We also know that for some sets of coins (such as American coins) it’s an optimal algorithm, in the sense that it minimizes the total number of coins given out. But we know that there are some cases (making change for 48p with classic British coins [1p, 2p, 3p, 6p, 12p, 24p, 36p, 60p] ) for which it doesn’t work.

It has been hypothesized that if every coin is at least twice as valuable as the next smaller coin, this greedy algorithm always provides minimal number of coins for a given amount of change (again, assuming the existence of a 1-“cent” piece).

Either show that this is true by outlining a proof that the greedy choice property holds,[[2]](#footnote-2) or prove that it isn’t true in general by giving a single counterexample.

***Solution:***

There are many counterexamples. 1, 7, 15 cent coins.

If I want to make change for 21 cents, the best way is to do three 7 cent coins not any 15 cent coin.



1. [5 pts] One problem that we will see multiple times is the Traveling Salesperson Problem (TSP) that is the underlying problem for all programs in this course this semester. Some ways of solving the TSP involve first finding some solution to the TSP then modifying it to make it into a better solution. But to do this, first you have to have a “quick and dirty” way to get a solution to the TSP.

Propose a greedy algorithm that can give you a solution to the TSP problem quickly. Describe your algorithm in careful English or pseudocode, and explain the running time of your algorithm as a function of the number of cities *n*.

***Solution:***

Order the cities in some way. Maybe east to west, maybe alphabetically, in any way. Then visit the cities in order and return from the last city to the first city. This will take O(n lg n) time to do the sort and O(n) time to do the TSP tour, for a total running time of O(n lg n).

1. [4 pts] Consider the job scheduling problem in which you have *N* possible jobs that you can do from among a set *J* of jobs. Each job *j* ∈ *J* has a deadline *dj*, will take you a number of hours *hj*, and will pay you revenue *rj*. Define *maxRev(N, J, H)* as the maximum revenue you could obtain from doing a set of these jobs, working no more than *H* hours total. The optimal solution to this problem cannot be done by a greedy algorithm, so you will use dynamic programming.
   1. [2 pts] Write a recurrence that expresses the maximum revenue you could earn doing these jobs while working no more than *H* hours. Note that the recurrence that I am envisioning does not explicitly include the variables *dj*, they would come in when deciding whether some set of jobs is compatible. (A set of jobs would be compatible if all of them could be finished within their deadlines.) That is, *maxRev(N, J, H) =* <xomething>.
   2. [2 pts] Would you attack this problem with bottom-up problem solving or top-down memoization? Why?

***Solution:***

Part A: maxRev(N, J, H) = maxk(rk + maxRev(N-1, J-jk, H-h­­k). In other words, pick one job, and find the maximum revenue of doing that job with the maximm revenue of doing the other jobs.

Part B: It’s really going to depend on the situation. In a case where the amount of time available *H* is enough to do most of the jobs, then bottom up would be better, since you’ll have to calculate most of them anyway. In the case where there are a lot of jobs with relatively little time, top down would be better.

***Note***: Any reasonable explanation for Part B got full credit, whichever you answered.

1. Note that 17 points are possible, so basically this homework includes a little extra credit. [↑](#footnote-ref-1)
2. We know that the coin changing problem has optimal substructure, you don’t need to address that if you want to prove that a greedy algorithm is optimal for this problem. [↑](#footnote-ref-2)