Nalongsone Danddank Student ID : 14958950 StarID: jf3893pd

Email: [nalongsone.danddank@my.metrostate.edu](mailto:nalongsone.danddank@my.metrostate.edu)

Homework 4 (*9.5/15* pts)[[1]](#footnote-1)

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:*** *That’s not a greedy algorithm that you listed. And you don’t address the issue of how to replace elements chosen for the knapsack, but only up to a certain number.*

Solve the 0-1 knapsack problem with limited replacement is the greedy algorithm or greedy choice property. So I would like to use greedy algorithm to solve it.

0, W=0

K(W) =

maxi ( K(w-wi) + vi), wi ) 1<= i <= n

1. [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:***  proof that the greedy choice property holds: *But it doesn’t hold, see answer key.*

**Optimization problem:**



input: n dollars and unlimited coin with values {vi} (a1, a2, a3, …an) where a2 > 2a1… ai > 2ai-1.

Output: the minimum number of coins with the total value n.

C(i): minimal number of coins for the total value *I*

Goal: C(n)

**Optimal substructure:**

Claim: suppose we have optimal solution to C(i), there are:

Case 1: coin1 is an optimal solution of C(i - v1)

We have Ci = minj(1 + C i - vj)

**Greedy choice property:** select the coin with the largest value no more than current total.

Proof via contradiction (use the case 1 <= i < 50 for demo)

Assume that there is no optimal solution including this greedy choice(choose 10)

All solution use 1, 5, 50 to pay i

50 cannot be used.

Coins with value 5 < 2, otherwise we can use a 10 to have a better output

Coins with value 1<5, other wise we can use a 5 to have a better output.

We cannot pay i with the constraints (at most 5 + 4 = 9).

1. [4/5 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:

Greedy algorithm for TSP is not give the optimal solution because it focus only on the local optima and optimizes the local best solution.

So this greedy algorithm searches for the local one and optimizes the local best solution to look for the global optima.

First, begins by sorting all the distance of the each cities and then selects the distance which is the minimum cost.

Second, continuously selects the best next choices that given a condition that on loops are formed.

Finally, until stop at the last city or destination. But there is no guarantee that a global optimum solution is found.

Time complexity is O(n2lg n) *I think your algorithm is O(n2 + lg n) = O(n2)*

Pseudocode:

GreedyTSP (V,E, home):

X[1] := home;

Begin for loop i := 1 TO n-1 do

Select (u,v) with min distance such that (u,v)ϵE, uϵS, vϵS

X[………………… S := Sᴗ{v};…

Until V≠{}

End for loop

Return V;

1. [3.5/4 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:

0, if n = 0

1. *maxRev(N, J, H) =*

ri = max(ri-1, J[i] + maxRev(N, J[n-i], H)), 1<= i <=n

Time complexity is O(2n) *Well, H – hk, the time to do the kth job.*

1. I would like to use memoization to attack this problem, because it is straightforward and easy to figure out for me. Then I can use the array or hash table to memorize or catch some result for saving and reuse again that let my algorithm run more quickly without duplicated and save time.

0, if n = 0

1. *maxRev\_memo(N, J, H) = memo[n] if memo[n] >=0*

ri = max(ri-1, J[i] + maxRev\_memo(N, J[n-i], H)), 1<= i <=n

memo[i] = ri

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)