Name: Jaime Arana-Rochel

Homework #4

Q1: Exercise 1 Chapter 6

(a)

Consider the path G with vertices $V = \{v_1, v_2, v_3\}$ with weights $W = \{60, 150, 100\}$ respectively. The greedy algorithm would choose to return the independent set $S = \{v_2\}$ which has total weight of 150. However, the optimal solution is the set $O = \{v_1, v_3\}$ which has total weight 160. So the algorithm does not always find a max weight independent set.

(b)

Consider the path G with vertices $V = \{v_1, v_2, v_3, v_4, v_5\}$ with weights $W = \{1, 8, 6, 3, 6\}$ respectively. The algorithm creates the set with odd vertices $S_1 = \{v_1, v_3, v_5\}$ which has total weight 13, and it creates the set of even vertices $S_2 = \{v_2, v_4\}$ with total weight of 11. The algorithm would therefore return S_1. However, the optimal solution is the set $O = \{v_2, v_5\}$ which has total weight 14. So this algorithm also fails.

(c)

We define $p(n)$ to be the value of the index $i < n$ that corresponds to the vertex v_i two edges away from v_n. So $p(n) = n - 2$, with the value being 0 for $n \leq 1$.

```
Independent-Set:
   // Array M will hold the values of the optimum solution
   M[0] = 0

   For i = 1 to n
      // w_i refers to the weight for vertex v_i
      M[i] = max(w_i + M[p(i)], M[i-1])

   endfor

   return M[n]
```

To find the actual set of vertices, we can simply trace back through values found starting with $M[n]$, printing the indices of M as we trace back.

Correctness

Looking at path G with n vertices, we observe the following about the optimal solution $Opt(i)$ of smaller paths in G. For any value $1 \leq i \leq n$, the vertex v_i in the path $\{v_1, \ldots, v_i\}$ either belongs or doesn't belong to the optimal solution. If it does, then the optimal maximum independent weight value $= w_i + Opt(p(i))$. If it doesn't then the optimal maximum weight $= Opt(i-1)$.
Thus we get the recurrence relation

$$Opt(i) = max(w_i + Opt(p(i)), Opt(i-1))$$

which gives the value of the maximum weight of the independent set of a path G with i vertices.

We thus aim to prove by induction on n that the algorithm writes $Opt(n)$ in the array entry $M[n]$ that is returned:
Base case: For $n = 1$, clearly the independent set is just the graph itself, in which case the algorithm returns $M[1] = w_1$.

Inductive step: Let our induction hypothesis IH be that for $i < n$, the algorithm correctly writes $Opt(i)$ into the array entry $M[i]$.
Consider $i = n$. By the IH, $Opt(p(n)) = M[p(n)]$, and $Opt(n-1) = M[n-1]$. Thus,

$$Opt(n) = max(w_n + Opt(p(n)), Opt(n-1))$$

$$= max(w_n + M[p(n)], M[n-1]) = M[n]$$

Since the algorithm returns this optimal solution $M[n]$ for $i = n$, by induction we are done.

Running Time

Based on the graph input with n vertices, the algorithm runs explicitly for n iterations of the for loop. Each iteration is spent doing constant time work - comparing the maximum between two array values. Thus, the algorithm runs linear on the input size; Independent-Set $= O(n)$.

Q2: Exercise 3 Chapter 6

(a)

Consider the ordered graph G with vertices $V = \{v_1, \ldots, v_5\}$ and edges $E = \{(v_1, v_2), (v_1, v_3), (v_3, v_4), (v_2, v_5$
The algorithm given would find a path v_1-v_2-v_5 with length 2. However, the actual longest path is v_1-v_3-v_4-v_5 which has length 3.

(b)

Algorithm

`Longest-Path:`

```
// path from v_1 to itself has length 0
M[1] = 0

// for every vertex after v_1
For i = 2 to n
   // a length of -1 indicates that it cannot be reached
   // from the vertex v_1
   len = -1

   For every edge (v_k,v_i)
      if M[k] != -1   //if v_k is reachable from v_1
         if len < M[k]+1 then
            len = M[k]+1
   endfor

   M[i] = len
endfor

return M[n]
```

Correctness

We note that for a set of vertices $v_1, \ldots, v_i$ of the graph G with n vertices, the optimal solution $Opt(i)$ is the longest path from v_1 to v_i. Since v_i may have multiple edges (v_k, v_i) going to it, we see that

$$Opt(i) = Opt(k) + 1$$

where the index k corresponds to the vertex v_k which has the longest path to it from v_1.

With this, we provide a proof by induction on n:
Base case: When $n = 1$, the longest path is of size 0 which the algorithm returns as $M[1]$.

Inductive Step: Let our induction hypothesis IH be the following - assume that for all $i < n$, the algorithm correctly writes the value of the optimal solution $Opt(i)$ into array entry $M[i]$.
Now let $i = n$. The algorithm checks every edge (v_k, v_n) and checks if the current guess for the longest path, gotten from a previous edge check, is less than $M[k] + 1$. If it is, then the value of *len* gets updated to $M[k] + 1$. By the IH, $Opt(k) = M[k]$ meaning that

$$Opt(n) = Opt(k) + 1$$

$$= M[k] + 1 = M[n]$$

The algorithm returns this value $M[n]$, so by induction the algorithm correctly returns the longest path and we are done.

Running Time

There are two for loops in the algorithm. The first one iterates through $n-1$ vertices, while the second one iterates trough at most $n-1$ edges. Since the second for loop is nested in the first one, we get a quadratic running time proportional to n; Longest-Path $= O(n^2)$.

Q3

Algorithm

```
Fractional-Knapsack:
   // w[] will be sorted by decreasing value per pound.
   Sort array w[]
   // a[i] should give the pounds of item i corresponding to the new entry w[i]
   Sort array a[] to correspond to array w[]

   total = 0
   i = 0

   // while the knapsack isn't empty
   while b > 0:
      // if we have used up all our m items
      if i = m then exit while loop

      // calculate how much of item i can take
      amount = min(a[i], b)
      // calculate total value based on that amount
      total = amount*w[i] + total

      b = b - amount
      i = i + 1
   endwhile

   return total
```

Correctness

We will prove that the greedy algorithm stays ahead of other algorithms which give optimal solution O. In the proof, items are sorted by decreasing value per pound. Furthermore, if the values per pound are not distinct, then we can simply perturb the values by small amounts so that they are distinct. First, suppose that the greedy algorithm took a fraction of item 0. If O also contains the same fraction of item 0, then we are done; both knapsacks are full with the same quantity of the highest valued item. If O contains less of a fraction of item 0, then the solution of the algorithm must have an equal or higher total value to O since it contains more of the highest valued item 0.

Now suppose that our algorithm takes amount a[0] of the 0^{th} item. If O also contains a[0] of item 0, then we move on to the next item. If the optimal solution O takes a fraction of the available pounds of item 0, then the solution given by the greedy algorithm must have an equal or higher total value to O since it contains more of the item 0 than O does.

In the case that both solutions took amount a[0] of item 0, we next examine how much each solution takes of item 1. By the same reasoning as above, we either arrive at the case where both solutions contain the same amount of item 1, or where the optimal solution contains less of it. In either case, the greedy algorithm returns a total value which is at least as high as the optimal solution O. We continue this reasoning for all m items if necessary, thus we are done.

Running Time

The algorithm first sorts the arrays a and w such that the n items are in decreasing order of value per pound. The sorting takes running time $O(n\,log\,n)$. The while loop then iterates until the knapsack is full, or we have taken the amounts for all n items. Thus the loop iterates for at most n iterations and is $O(n)$. The total running time is therefore $O((n\,log\,n) + n) = O(n\,log\,n)$.