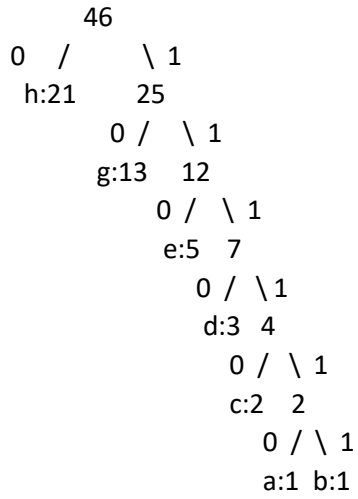
George Lenz
7/20/2018
HW 4

Problem 1:



An effective pattern emerges in the tree for Fibonacci numbers. Essentially the two ones are at the bottom and each successive number is up one branch to the right so you could generalize the algorithm for a Fibonacci sequence to just say put the two ones at the bottom and build the tree up in order with the i-th term at the top right . Another method would be to say that the first 1 has i-1 1's in it and every successive term has one less 1 and a 0 tacked on, with the final term just being 0. For example for 7 terms, the first number has 6 1s, the next has 5 1s and a 0, the next has 4 1s and a 0, then 3 1s and a 0, then 2 1s and a 0, then a single 1 and a 0, and then the final term is just 0.

Problem 2:

You can sort the cities by the distance to each. If the hotel is less than the amount of days, than you can take the hotel with the longest distance traveled. From there you can take now choose the hotels less than a day of travel from the previous hotel and choose the one with the longest distance covered. This should work because for any optimal subset from j..k that does not include i (our initial optimal choice) you can replace j with i and days traveled of i has to be <= j so the total days traveled is either equally optimal or better than the previous solution. The running time would be O(n lgn) because the sorting would take the longest amount of time and thus dominate the growth. Since the items are sorted then selecting the longest day that would fit in a day for each successive interval should only take at most O(n) running time because we only every have to move forward in the list. And since the sorting function dominates the growth it would be O(nlgn)

Problem 3:

If we assume the activites are already sorted by start time (if not we can sort them) then we can implement it by selecting the item with the latest start time first. By selecting the latest start time, we

then select the next latest start time such that it fits within the subset that ends before our previous item. So essentially we just move down the list, and if the end time is less than or equal to the start time of our previous item, we take it. This is a greedy algorithm because we are making a local choice for each selection that ultimately leads to our optimum solution. Since we always choose the one with the latest start time that ends before our previous one starts we can always ensure that the solution is optimal. In an optimum solution for every item i that we have, it has an optimal subset j…k such that the end times for j…k are less than or equal to the start time for our item i. if we do not use the item with the latest start time, then we have a solution that includes the subset j…k that does not include our next optimum choice. Since our optimum item had a later start time than our item j, but still ends before activity i starts, it can replace j and still not only fit within our item, but leave and equal amount of time or more left for the next activity selections.

Problem 4:

Description:
To implement you first have to sort the solution by start times. Then you can add the item with the latest start time to the solution. From there you can iterate from the latest start time to the earliest, and if the end time of next activity is less than or equal to the start time of the last item in your solution it can be added, otherwise you keep iterating through the list.

PseudoCode:

```
Sort array by start times
i <-0
last item in array is first in solution
from listLength – 1 to 0
   if array[i].finishTime < lastItemInSolution.startTime
      add to solution
   else
      continue loop
```

Analysis:

The algorithm runs through the array of items once and never goes back so the running time of the algorithm would be theta(n).