Shannon Jeffers
CS 325 Fall 17
HW 4

1. This algorithm is going to be extremely similar to the machine scheduling algorithm discussed in the lecture. The main difference is going to be what items are called. Each class will need to have a start and finish time with its start time being less than its finish time. Unlike the first scheduling activity we discussed, this one is going to rely on the start times rather than finish times, so the activities need to be sorted by start time. The algorithm will take the class list (with start and finish times) as it's input. This list of classes will need to be sorted, we will do this with merge sort because it is fastest. It will start by setting the number of classrooms to 0, and we will add a classroom to the binary heap only when a new classroom is necessary. The classroom heap will be controlled by the latest finishing times assigned to each classroom. We will loop through the classes in the class list. For each class we will check if there is a classroom in our list of classrooms that is open at that current time. If so, the classroom will be removed from the heap and the time it is available adjusted, then it will be added back to the heap. We will add the class to that classrooms schedule, and if not we will add a new classroom to our classroom heap and add the class to the new classroom's schedule. We will do this until we are out of classes to schedule.
   Psudocode:
```
scheduleClass(classes):
        cr = 0  //Number of classrooms
        mergeSort(classes)
        for class in classes:
                if a classroom is open:
                        pop classroom from heap
                        schedule class to that classroom
                        add classroom back to heap
                else
                        create new classroo
                        schedule class on new classroom
                        add classroom to heap
```
   This algorithm would have a run time of about thetan log n) because the classes would need to be sorted first to get them in class starting time order. MergeSort has a time complexity of theta(n log n). The loop would also have a running time of theta(n log n) because the for loop goes through every value in the classes array and binary heap operations can be done in tetha(log n) time.

2. This algorithm is a little different. It is greedy in the fact that you are going to travel as close as you possibly can to the dmiles each day and stop at the hotel that is as far away as possible. This is done using a couple of loops. Since it states that you already have a map that gives the hotels in distance order, there is no reason to sort your distance list. The first loop in this algorithm is going to be a while loop that keeps going until we reach our destination. There will be another loop inside this loop that steps through the hotels until we find the hotel that is closest to d miles away. This hotel is added to our list of stopped at hotels and this process continues until we reach our destination.
```
roadTrip((hotels):
        curr = 0
        while distance is less than or equal to distance of farthest hotel
            prev = curr
            while curr <=  furthest hotel (hotels.length) & hotels[curr+1] - hotels[prev] <= d
                add one to curr
            add hotel to solution
```

With this algorithm the run time with be theta(n) because every hotel in the list of hotels will be walked through exactly once. It qualifies as greedy because we don't revisit any past hotels. This problem is a little tricky however. Another algorithm which uses a while loop and binary search to find the next hotel to stop at would have a worse case running time of O(n log n). The only way we would get a worse case is if we had to stop at every hotel. The fewer hotels we end up stopping at the closer to O(log n) this algorithm would become. This problem is tricky because when talking about worse case run times the algorithm above with theta(n) definitely takes the cut but it may not always be the algorithm that we want to use to solve this problem.

3. With this algorithm we want to sort the jobs that need to be done by penalty in descending order (where the highest penalty is the first in the list). We will do this sort using merge sort to get a fast sorting time. We will need 2 arrays, one to tell us if something is already scheduled at that time and one to hold which jobs we’ve already added to the list. We will need a for loop that goes through every job in the sorted job list. Nested inside this for loop will be a second for loop that goes from the min of (n, deadline) down to 0. Within nexted for loop the algorithm will check to see if a job is already taking each position from deadline/n down to zero. The first open position will be given to the current job. If there is not room between n/deadline and 0, this is a penalty that will be incurred. We want to do this backwards loop as the nested loop because we want to schedule the item with the highest penalty as late as possible to still meet the deadline. This is because scheduling it earlier may make us incur more penalties from things we could have scheduled but did not, but it doesn’t hurt to do the highest penalty item at the last minute.
   Psudocode:

```
jobSchedule(jobs):
    sort jobs in descending order
    let scheduled be a new array of size jobs.length
    let result be a new array of size jobs.length
    for job in jobs
        count = min(jobs.length, job deadline)
        while count &gt; -1
            if scheduled[i] == 0 //if no job is scheduled at this time
            result[i] = job
            scheduled[i] = 1 // schedule the job at this time
            count -= 1
            break
```

   Due to the double loop structure in this algorithm the run time will be O(n^2). I say this is O(n^2) and not theta because the inner loop only runs until the min of n and deadline. If the deadline is quite a bit before the number of n, then this loop will run far fewer times.

4. This algorithm is set up to start from the end. Since this is a problem from the book, lets explain this using the same terms as the book. This means we have a set o activities that we can call S_{ij} and from that set we can obtain a set of activities A_{ij} which is a maximum subset. This time we

need to have our list of activities arranged by descending start time (the last to start being first). Now we can proceed the exact same way as the original activity selection algorithm but with starting times rather than finish times. We take the last to start and add it to the list, now we skip ahead to the next activity that doesn't overlap with the first one we added and we add that to the list, effectively only taking those that are the "last to start" of what is left over without overlap. The definition of a greedy algorithm states that it always makes the choice that looks best at the moment, which is exactly what this does.

Proof:

Theorem 16.1 – modified to meet our requirements

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the Latest start time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.L

The modifications done to this theorem are basically just changing earliest finish time to latest start time. The algorithm works essentially the same. The main difference is it is comparing the current finish time to the previously added activities start time to make sure they are compatible instead of comparing the current start time to the previously added's finish time.

So let's say we have A_k as a maximum-size subset of compatible activities. Lets let a_j be an activity in A_k

If $a_j = a_m$ then it is already in our set and we are done

If they are not equal then we get $A'_k = A_k – \{a_j\} \cup \{a_m\}$ can be set to A_k but substituting a_m for a_j. The activities do not overlap and a_j is the last activity to start. Since these subsets are the same size, we conclude that A'_k is a maximum-size subset of mutually compatible activities which include a_m

5. This algorithm needs to be able to take in a data set that includes start and finish times. It then sorts the data (merge sort) so that the start times are in descending order with the last start time being the one at the beginning of the list. The algorithm adds the first activity to the list automatically, because this is the activity with the last start time. I then starts searching through the list to find the next compatible activity. This activity will be the activity who's finishing time is before the previously added activity's start time. Once the entire list has been traversed we will be left with a maximum set of activities that can be done.

   Psudocode:

```
greedySelect(s, f):
        sort start (and finish) so that starting times are in descending order using merge sort
        n = start.length
        let A be a new array
        add the activity with last start time to A
        k = 0
        for m in range(1, n)
                if s[k] >= f[m]
                        add the activity at index m to A
                        k=m
        return A
```

The running time of this algorithm as written would be theta(n log n) because the algorithm handles the sorting which can be done in theta(n log n) time and the time it takes to add activities to the list is theta(n). If the input that was passed in was assumed to already be sorted then we would not need the sort function and the algorithm would be theta(n) but because the teacher said we cannot assume the data is sorted, the sort needs to happen in order for the algorithm to run giving us an Theta(n log n) running time.