Tristan Santiago
CS325_400
October 21, 2018
Homework 4

Question 1
Suppose you have a set of classes to schedule among a large number of lecture halls, where any class can place in ay lecture hall. Each class c_j has a start s_j and finish time f_j. We wish to schedule all classes using as few lecture halls as possible. Verbally describe an efficient greedy algorithm to determine which class should use which lecture hall at any given time. What is the running time of your algorithm?

Input: A set of classes C = $\{c_1,\ldots,c_n\}$, each with a start time and a finish time = $c_1 = (s_j,f_i)$
Two activities are compatible if and only if their interval does not overlap.

Output: A schedule full of non-conflicting classes that uses the minimum number of lecture halls possible, while also using all classes in the given set.

The idea is to go through the classes in order of start time and assign each class that already had a scheduled class and is available at the appropriate starting time. If no such lecture hall is found, the class would then be scheduled in a new lecture hall, which effectively guarantees that the algorithm uses the fewest number of lecture halls while also making sure all classes compatible.

```
Algorithm ClassSchedule(C)        // Adopted from Week 4, Part 3 lecture video
h ← 0 // Number of lecture halls
while C is not empty
    remove class i with smallest $c_i$ // Remove class with earliest start time
    if there's a lecture hall j for i then
        schedule i in lecture hall j
    else
        h ← h + 1
        schedule i in lecture hall m
```

If the classes are already sorted by start time, then the running time of this algorithm would be $\Theta(n)$. However, if the classes are not already sorted, our running time would be $\Theta(n \log(n))$.

Question 2
For each $1 \le i \le n$ job j_i is given by two numbers d_i and p_i where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute and once the job starts it cannot be stopped until completed. We want to

schedule all jobs, but only one job can run at any given time. If job i does not complete on or before its deadline, we will pay its penalty p_i. Design a greedy algorithm to find a schedule such that all jobs are completed and the sum of all penalties is minimized. What is the running time of your algorithm?

Input: An array of deadlines d_i and an array of penalties p_1
Output: A schedule of jobs that minimizes the penalty incurred.

The first thing we would want to do is sort the jobs in decreasing order of penalties.
Next, suppose the jobs in this sorted array are denoted as $\{j_1, j_1,\ldots,j_n\}$

```
If deadline($j_i$ == i) or (i == n)
        Schedule $j_i$ here
Else
        Call the algorithm iteratively on the jobs from $\{j_1, j_{i+1},\ldots,j_n\}$
```

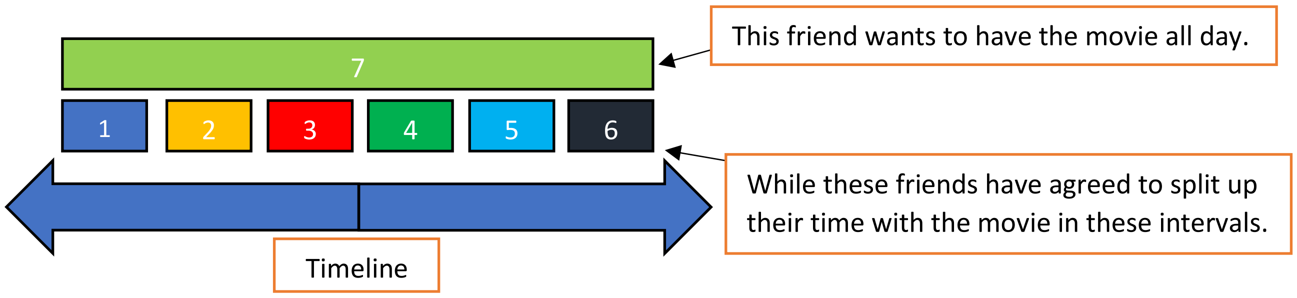
Since we're sorting and sorting takes Θ(n log(n)) time, that would be our overall running time.

Question 3

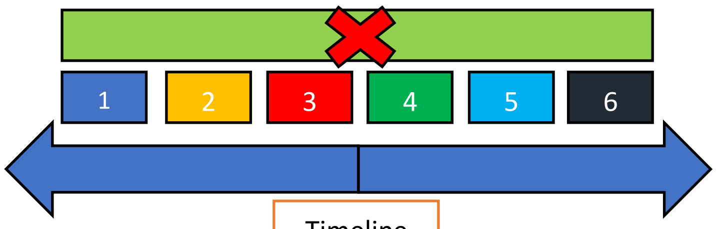
Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible will all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

I referenced the following video to help me solve this problem:
https://www.youtube.com/watch?v=BWlXudP7Unk&t=584s

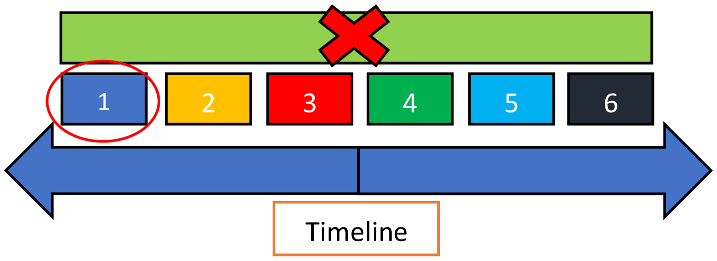
Consider a problem where we make the greedy choice and choose the first to finish (i.e. we choose the earliest intervals) to receive our optimal solution. Suppose we have a movie that we want to loan some friends, and we want to loan the movie to as many friends as possible (represented by the timeline in the following example) as shown here:


7
1
2
3
4
5
6
This friend wants to have the movie all day.
While these friends have agreed to split up their time with the movie in these intervals.
Timeline

In the example provided above, when we choose the first to finish (the earliest interval), we end up choosing friend #1. Then we cross out the overlapping interval of friend #7.

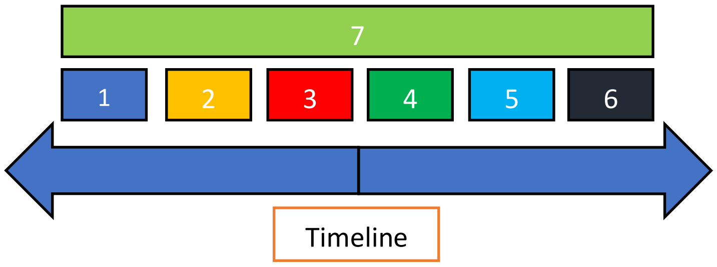


Next, we choose friend #2, because friend #2 has the next earliest interval (next earliest finish time).

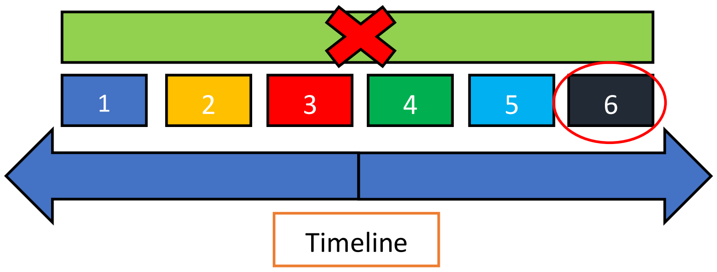


Then friend #3 and so on. We continue this process until eventually receive the optimal solution of allowing six friends to borrow the movie throughout the course of the day.

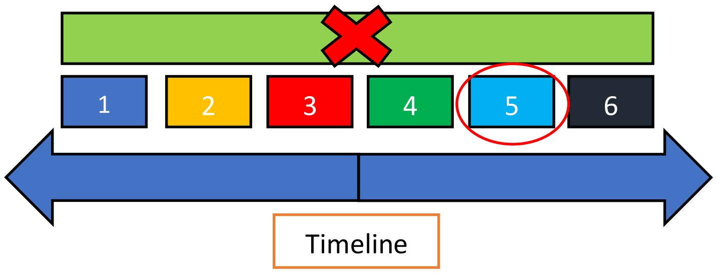
Now, suppose we want to choose the last to start (start the latest) instead. Using the same example from before:



Our first choice here would be friend #6. So, we choose friend #6 and we then cross out the overlapping time interval of friend #7.



Next, we look for the next friend that has the latest start:



Here, that would be friend #5. We continue this process until eventually we reach the optimal solution of six. With both algorithms, we still receive the optimal solution of six because both algorithms are symmetrical.

Problem 4

Include a verbal description of the algorithm, pseudocode, and analysis of the theoretical running time.

I implemented this program using C++. I decided to create a class called Activity to contain the activity number, start time, and finish time. The class also has setter and getter functions, which were incredibly important for getting the algorithm to work correctly. I then read the numbers from the input file into a vector of objects (Activity).

I then sorted the vector containing the set of activities in descending order, according to their start time. When the sort function for vectors encounters an activity with the same start time, it uses the finish time of each activity to order them. For example, in Set 1, both activity 8 and 9 have a start time of 8. The vector sort function places activity 8 before 9, because activity 8's finish time is 11, whereas activity 9's finish time is 12. I believe this vector sort function a O(n) running time, since the vector being sorted only contains integers (in this case, start times of each activity) and the time it takes depends on the number of elements it has to sort.

Since I was working with a sorted vector before doing any comparisons to determine the optimal solution, I knew the first element of the sorted vector would always be the activity with the latest start time. Therefore, my program takes the greedy approach and manually inserts the activity number of the first element of the sorted vector into the new vector of integers containing the optimal solution (i.e. numbers of the activities chosen).

I then begin the algorithm that starts making its comparisons to that first activity, which acts as our "current" activity. The algorithm looks at the finish time of the next activity and compares it to the start time of the current activity. If the finish time of the next activity is less than or equal to the start time of the current activity, that activity can be included in our solution, so it is inserted into the vector containing the results (i.e. the optimal solution). The algorithm updates the current activity to the most recently inserted activity, increments the variable used to track how many activities selected, and this process continues for each activity in the set. Once the algorithm determines the optimal solution, the program re-sorts the activity numbers in ascending order to match the sample output provided with the question. Because the greedy algorithm is comparing integers, the running time would depend on the number of activities in the set. Therefore, I would estimate the running time to be $O(n)$. The initial sorting is done outside the actual greedy algorithm (in $O(n)$ time). If I combined the running times (despite the fact that they are separated in my program), I believe the theoretical running time of both would be $O(n^2)$. Had I been able to successfully been able to implement a merge sort with my program, the running time would likely be a little faster, dominated with a running time of $\Theta(n \log(n))$.

Pseudocode

```
int selected;  // Already set to one, since the first activity has already been inserted.
int i = 0         // Current activity index
int j = 1         // Next activity index
For k = 0; while k less than number of activities per set
     If activity[j] finishTime <= activity[i] startTime // If next F <= current S
          Insert activity[j]       // Place next activity into results vector
          i = j    // Update current index
          selected++;  // Increment selected variable
     Else    // The activity cannot be included in the solution:
          j = j + 1;       // Update index to the next activity.
```