Arthur Liou
CS325

HW 4

Problem 1: (5 points) Class Scheduling:

Suppose you have a set of classes to schedule among a large number of lecture halls, where any class can class place in any lecture hall. Each class cj has a start time sj and finish time fj. 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?

Consider lectures in increasing order of start time: assign lecture to any compatible classroom. Number of lecture halls needed is the number of "queues" we need to keep track of

- Sort the intervals from earliest start time to latest
- Keep the list of needed and current lecture halls in a priority queue.
- Iterate through the list of classes. If class can be added (and optimized) to a current lecture hall, then do so. If not, then add a new lecture hall to the priority queue.
- For each classroom k, maintain the finish time of the last job added.

Time Complexity / Running Time: O(n log n).

Problem 2: (5 points) Scheduling jobs with penalties:

For each $1 \leq i \leq n$ job, ji is given by two numbers di and pi, where di is the deadline and pi 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 pi. 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?

Set up a "queue" of minutes. Example would be an array, where each element is a minute [first minute, second minute], etc. Consider array[0] to be *M0, Mi.* Thus, these are time intervals Mi for $1 \leq i \leq n$ where Mi starts at minute $i - 1$ and ends at minute i

Arrange the jobs in decreasing order of the penalties $p1 \geq p2 \geq \cdots \geq pn$ and add them in this order.

To add a job *ji*, if any time interval *Mi* is available for $1 \leq l \leq di$, then schedule ji in the last such available interval. Else schedule ji in the first available interval starting backwards from Mn. If this is not possible, then track the sum of penalties which are accrued.

Important to note that only one job can run at a time & sorting in a certain order, similar to a QuickSort or an Insertion Sort

Time Complexity / Running Time: O(n^2).

Problem 3: (5 points) CLRS 16-1-2 Activity Selection Last-to-Start

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

This approach is still a greedy algorithm; it just starts from the end instead of the beginning. If we use the Associative Property (or some form thereof), we can thus prove this approach is still an optimal solution.

Problem 4: (15 points) Activity Selection Last-to-Start Implementation

Submit a copy of all your files including the txt files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named act.txt. You may use any language you choose to implement the activity selection last-to-start algorithm described in problem 3. **Include a verbal description of your algorithm, pseudocode and analysis of the theoretical running time.**

You do not need to collected experimental running times. The program should read input from a file named "act.txt". The file contains lists of activity sets with number of activities in the set in the first line followed by lines containing the activity number, start time & finish time.

Your results including the number of activities selected and their order should be outputted to the terminal.

Verbal Description: The algorithm will take in start and finish times of activities as inputs, which are sorted by finish time. The algo will select last activity to start first. Then it will scan through activities in descending order and finds a compatible activity to add to set A. Return Set A at the very end. This return (put into number of activities selected and their order) will be outputted to console.

PseudoCode: S is a set of activities, f is the finish time, A = set of activities where [number, s, f]

```
function greedy (a, s,f)
        n = length of A
        A = results array
        index = n
        for m = n-1 down to 1
                if f[m] <= s[index]
                        add a[m] to results array
                        index = m
        return A
```

Analysis of the theoretical running time: Assuming that f is already sorted, then O(n) since it's iterating through the array. Otherwise, if inputs of f were unsorted, then it would be O(n log n) since we would use an efficient sorting algo (like MergeSort), then run through the greedy algo.
If F is sorted: O(n)
If F is unsorted: O(n log n)
Since in this problem the activities are not in any sorted order, it will be **O(n log n):**