HW 4 - Solutions

Problem 1: 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 c_j has a start time 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?

Algorithm CourseScheduler(C):

Input: A set C of n courses, such that each course has a start time s_i and a finish time f_i
Output: A nonconflicting schedule of courses in C using a minimum number lecture halls L.
m= 0 Optimal number of lecture halls.
C <- a heap of courses with earliest start time at top of heap
L <- an empty heap of lecture halls with earliest finish time at the top
While (heap C not empty)
remove from C the course i with earliest start time s_i
remove from L the lecture hall j with the earliest availability a_j
if s_i >= a_j then
schedule course i in lecture hall j
update the availability time of lecture hall j by setting $a_j = f_i$
else
m = m + 1
schedule course i on lecture hall m
set $a_m = f_i$ and insert into heap L

The key here is to use two heaps: The first, C, to represent the set of courses to be scheduled as shown in the algorithm description above. The second, L, represents the lecture halls on which courses have been scheduled and the finishing time of the last scheduled course on each lecture hall is the next availability of that lecture hall. The formation of the heap can be done in time n log n as was discussed with the heap data structure. The while loop is executed n times (once for each course added to the schedule). The removal of the top element from the heap takes time O(log n) and together this is O(nlgn). Sorting is Θ(nlgn).

The overall running time is Θ(nlgn) depending on the data structure used Θ (n^2) is a less efficient solution

HW 4 - Solutions

2. **Scheduling jobs intervals with penalties**:

For each $1 \le i \le n$ job j_i is given by two numbers d_i and, where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute. 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 d, we should pay its penalty. Design a greedy algorithm to find a schedule which minimizes the sum of penalties.

Observation: We can assume that all jobs finish after n minutes.

Proof: Suppose not. So there is an empty minute starting at say $0 \le k \le n-1$ (where no job is scheduled) and there is a job j_i for some $1 \le i \le n$ which is scheduled some time after n minutes. If we schedule job j_i to start at minute k, then this can only be a better solution since everything remains same except j_i might now be able to meet its deadline.

We assign time intervals M_i for $1 \le i \le n$ where M_i starts at minute $i-1$ and ends at minute i. The greedy algorithm is as follows:

- **Arrange the jobs in decreasing order of the penalties $p1 \ge p2 \ge \cdots \ge pn$ and add them in this order**
- **To add job j_i ,**
  - **if any time interval M_l is available for $1 \le l \le d_i$, then schedule j_i in the last such available interval.**
  - **else schedule j_i in the first available interval starting backwards from Mn**

What is the running time of your algorithm? Explain.

1. The sorting of by penalties takes Θ(nlgn)

2. To find the spot in the schedule can take in worst case 1+2+...+ n = O(n^2) depending on the data structure used this could be reduced so you can use Big O instead of theta.

Overall O(n^2) or Θ(n^2).

Note: By using a disjoint set data structure the running time can be improved to O(nlgn). This solution was not necessary for full credit.

HW 4 - Solutions

3. **CLRS 16-1-2 Activity Selection Last-to-Start**

Sample Solution 1: Must prove the greedy choice property for Last-to-Start greedy choice criteria.

Greedy Choice Property:

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

Proof: Let A_k be the maximum-size subset of mutually compatible activates in Sk, and let a_j be the activity in A_k with the last start time.

- **If $a_j = a_m$, we are done since we have shown that a_m is in a maximum-size subset of mutually compatible activities.**
- **If $a_j \neq a_m$, then $s_m \geq s_j$ since a_m was the last-to-start of all activities. Let $A_k' = A_k - \{a_j\} \cup \{a_m\}$. The activities in A_k' are mutually compatible because a_j was the last to start in A_k which implies that $s_j \geq s_i$ for all a_i in A_k. Therefore $s_m \geq s_i$ for all a_i in A_k since a_m started after a_j. Since $|A_k| = |A_k'|$, we conclude that A_k' is a maximum-size subset of mutually compatible activities in S_k and it include the greedy choice a_m.**

Alternative Solution 2: Last-to-Start is equivalent to First-to-Finish

The proposed approach—selecting the last activity to start that is compatible with all previously selected activities—is really the greedy algorithm but starting from the end rather than the beginning.

Another way to look at it is as follows. We are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of activities, where $a_i = [s_i, f_i)$, and we propose to find an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. Instead, let us create a set $S' = \{a'_1, a'_2, \ldots, a'_n\}$, where $a'_i = [f_i, s_i)$. That is, a'_i is a_i in reverse. Clearly, a subset of $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\} \subseteq S$ is mutually compatible if and only if the corresponding subset $\{a'_{i_1}, a'_{i_2}, \ldots, a'_{i_k}\} \subseteq S'$ is also mutually compatible. Thus, an optimal solution for S maps directly to an optimal solution for S' and vice versa.

The proposed approach of selecting the last activity to start that is compatible with all previously selected activities, when run on S, gives the same answer as the greedy algorithm from the text—selecting the first activity to finish that is compatible with all previously selected activities—when run on S'. The solution that the proposed approach finds for S corresponds to the solution that the text's greedy algorithm finds for S', and so it is optimal.

HW 4 - Solutions

4. a) description & pseudo code – samples below

This algorithm begins by sorting the activities in descending order by start time. We send those activities to our function that determines the order of activities that produces the maximum number of non-conflict activities. The function has an empty list to store the activities in. We append the activity with the latest start time to the list right away because we are starting with that activity. Then we loop through the list of activities and compare the start time of our first activity to the finish time of the next activity until we find an activity that does not conflict with our first one. We put that activity into our list and search for the next activity that doesn't conflict with our current activity. After we are done searching, we reverse the list of activities because we started at the end. This gives us our solution.

Pseudocode:

```
lastToStart(activity)
        answer = []
        i = 0
        answer.append(activity[0])

        for j = 1 to number of activities
                if activity[i].start >= activity[j].finish
                        ans.append(activity[j])
                        i = j

        answer.reverse()
```

The theoretical running time of this algorithm is Theta(nlgn) because our most costly operation is sorting the list by decreasing start time. Looping through the list and reversing the list are both O(n) which is out competed by sorting.

Input a_n activities, each with a start time s_i and a finish time f_i

Output: Maximum activities A

Sort the data using merge sort, or equivalent, from last finish time to first.
count ← 0
check that the array isn't empty – return error if it is
A[count]=a_n // Add the first element to the array
j← n-1
while j < n
        if f_j <= s_{count} // If the finish time is before the start time of an accepted
        A[count+1] = a_i
        count++;
return count+1 // One more since the array starts at index 0.

To display the solution, A needs to be printed in reverse. Alternatively, the array could be reversed here, or a stack could be used which would produce the correct behavior as well.

HW 4 - Solutions

b) running time Θ(nlgn)

Test cases in act.txt

11
1 1 4
2 3 5
3 0 6
4 5 7
5 3 9
6 5 9
7 6 10
8 8 11
9 8 12
10 2 14
11 12 16
3
3 1 8
2 1 2
1 3 9
6
1 1 5
2 3 6
3 6 10
4 8 11
5 11 15
6 12 15

Solutions:
Test set 1: 2 4 8 11 (alternative 2 4 9 11)
Maximum number of Activities = 4
Test set 2 : 2 1
Maximum number of Activities = 2
Test set 3 : 2 4 6
Maximum number of Activities = 3