James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Homework 4

Due Date: Sunday, April 29, 2018

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 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?

ANSWER:

First, we need to arrange the list of classes by their start time. We can implement a sorting algorithm like merge sort to accomplish this task since it operates faster than insertion sort. As discussed in week 1, merge sort will receive the list (array) of classes and their start and finish times. It will then segment the array in half recursively until we finally have array lengths of a single element so that comparisons can be performed. The merge sort will exchange elements so that they are arranged in ascending order based on class start time and then merge the array back into a single list with the new assigned order. Next, the sorted list of classes will be sent to the greedy class scheduling algorithm where the first class in the list is selected automatically and assigned a classroom. The algorithm will process through each class in the list to compare its start time with the previous course's end time. If it starts before the previous class ends then a new classroom will need to be allocated, if not then the class can use the same classroom once it becomes available again. The classroom will remain assigned until the finish time of its current class has been reached.

Class Scheduling (Theoretical Running Time):

The theoretical running time for this algorithm is $\theta(n \log n)$. To exclusively process the greedy selection algorithm, it would only be linear $\theta(n)$ but the list of classes must first be sorted into ascending order based on their start time before we can perform our selections. Implementing merge sort to arrange the list of classes by start time adds a time complexity of $\theta(\log n)$. When combining the running times of the greedy selection algorithm and merge sort, we receive a final running time complexity of $\theta(n \log n)$.

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Problem 2: *(5 points)* **Road Trip:**

Suppose you are going on a road trip with friends. Unfortunately, your headlights are broken, so you can only drive in the daytime. Therefore, on any given day you can drive no more than d miles. You have a map with n different hotels and the distances from your start point to each hotel $x_1 < x_2 < \cdots < x_n$. Your final destination is the last hotel. Describe an efficient greedy algorithm that determines which hotels you should stay in if you want to minimize the number of days it takes you to get to your destination. What is the running time of your algorithm?

ANSWER:

The design of this algorithm is to attempt to travel as close to the d mileage limitation each day before stopping at the furthest possible hotel from that day's starting point (since our headlights are broken, we are unable to drive at night). This means that we need to find a balance between reaching maximum mileage without going past the last available hotel within d miles. The next day, this process will be repeated until we have reached our final destination. Fortunately, the hotels are already sorted from closest to furthest (in miles) because of the default linear nature of a map. Therefore, merge sort does not need to be implemented initially to arrange the hotels by distance before making our selections. The greedy road trip algorithm will need to process through each hotel in the list and select the last possible hotel that is still closer than the possible d miles allotted each day. This process is then repeated each day until our final destination is within the d miles and we've completed our road trip.

Road Trip (Theoretical Running Time):

The theoretical running time for this algorithm is linear $\theta(n)$ because we need to process through each of the hotels in our list between the start and the finish destinations.

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Problem 3: *(5 points)* **Scheduling jobs with penalties:**

For each $1 \leq i \leq$ 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?

ANSWER:

A greedy algorithm for scheduling jobs with penalties would first need to call a sorting algorithm such as merge sort, to arrange the job in descending order by penalty. The greedy algorithm will initially process through each element (job) in the sorted array returned from the merge sort function. If there is an unscheduled time slot available then we will schedule the current job that is being processed in the outer loop until all time slots are filled. Once the time slots are full, the algorithm will need to process through each minute from the deadline down to zero and assign the job in the first empty timeslot that becomes available.

Scheduling Jobs with Penalties (Theoretical Running Time):

The theoretical running time complexity for this algorithm would be quadratic $O(n^2)$. The merge sort algorithm necessary for organizing jobs in descending order by potential penalties incurs a $\theta(n \log n)$ running time complexity. This, however, is dominated by the job scheduling algorithm with a quadratic $O(n^2)$ running time complexity due to the for loop and the nested for loop necessary for all the processing that occurs.

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Problem 4: *(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.

ANSWER:

The Last-to-Start Activity selection utilizes a greedy approach because it first sorts the activities into descending order based on the start time. The algorithm initially selects the first activity in the array which will have the latest start time after our sort. The algorithm then works through the list and compares that start time to earlier activities finishing time. If the finish time is less than or equal to the current selections start time, then we'll add that activity to the selections list and move the comparison index to the current element being analyzed. The algorithm will continue this process until it's finished processing through all activities in the list. This is definitively the structure of a greedy algorithm as it is making decisions based on the current state of the scenario and not based on results demonstrated from subproblems as in Dynamic Programming. To prove that it yields an optimal solution, Theorem 16.1 from the required text (page 418) could be utilized. I followed it pretty much verbatim since it so closely resembled the problem presented here in the assignment.

Theorem 16.1 Proof (Page 418)
S_k = nonempty subproblem
a_m = activity in S^k
A_k = maximum-size subset of mutually compatible activities in S_k
a_j = the activity in A_k with the *latest* start time

If $a_j = a_m$
Then the proof is complete as we have adequately shown that a_m is in some maximum-size subset of mutually compatible activities of S_k

Otherwise, if $a_j \neq a_m$
Let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k, substituting a_m for a_j
A'_k activities are disjointed
Because the A_k activities are disjointed, a_j is the *last* activity to *start,*
and, $f_m \leq f_j$
$|A'_k| = |A_k|$
Therefore, A'_k is a maximum-size subset of mutually compatible activities of S_k, including a_m

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Problem 5: *(10 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 4. 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.

Example 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 6 8
1 7 9
2 1 2
```

In the above example the first activity set contains 11 activities with activity 1 starting at time 1 and finishing at time 4, activity 2 starting at time 3 and finishing at time 5, etc.. The second activity set contains 3 activities with activity 3 starting at time 6 and finishing at time 8 etc. The activities in the file are not in any sorted order.

Your results including the number of activities selected and their order should be outputted to the terminal. For the above example the results are:

```
Set 1
Number of activities selected = 4
Activities: 2 4 9 11
```

Set 2
Number of activities selected = 2
Activities: 2 1

Note: There is an alternative optimal solution for Set 1. Since activities 8 and 9 have the same start time of 8, a2 a4 a8 a11 would be an alternative solution. Your program only needs to find one of the optimal solution. For either solution the activities differ from the solution presented in the text which uses the earliest-finish time criteria.

ANSWER:

My algorithm for this problem first receives a list of the activities, sorted by latest start time using a separate merge sort function (I chose merge sort because while it has greater storage overheard, it processes the sort algorithm faster than insertion sort). Merge sort is called before the selection algorithm to rearrange the initial list of activities by start time in descending order. Within the function for the greedy algorithm, we establish an empty array that will ultimately hold our activity selections. Since this is a greedy implementation, the first element in the list is arbitrarily selected and added to the selections array. The program then iterates through each element in the sorted activity array, starting with the second element and performs comparisons on the start and finish times. It first checks if the start time of element 0 is greater than the finish time of element 1. If this is true, then the element with the smaller finish time is added to the selections array and the index is moved to the current element in the array. This continues for the length of the activities array.

Below is the pseudocode for my implementation of the Last to start activity greedy algorithm, an analysis of the theoretical running time, and screen captures of the actual code implementation in python with execution results on my terminal confirming its validity. The complete version of this program has been submitted to the university TEACH servers.

Activity Selection Last-to-Start Implementation (Pseudocode):

```
# Call the sort function first to sort array and send result to greedy algorithm
sorted_activities := activity_sort (amount, activities)
activity_selection (amounts, sorted_activities)

activity_selection (amount, activities)
        selections := []
        i := 0
        selections [0] := activities[i][0]
        for x in range (1, amount)
                if activities[i][1] >= activities[x][2]
                        selections.insert(0, activities[x][0])
                        i := x
        end for loop
        return selections
```

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Activity Selection Last-to-Start Implementation (Theoretical Running Time):

The actual implementation of the greedy last to first activity selection algorithm would be linear $\theta(n)$ but since the list of activity must first be sorted by latest start time, and since I opted for the implementation of Merge sort, the actual running time of the program is $\theta(n \log n)$.

Activity Selection Last-to-Start Implementation (Python):

```
"""
/*************************************************************************
* Description: activity_selection function
* This is my greedy sort algorithm used for selecting tasks based on
* those with the latest start times.  It receives amounts and activities
* arrays from main and compares the start and finish times to find an
* optimal solution.  Returns the selection array back to main in reverse
* order (this was the only way that I could figure out how to match the example
* answers provided in the homework)
*************************************************************************/
"""

def activity_selection(amounts, activities):
    selections = []
    compare = 0
    selections.append(activities[compare][0])
    for x in range (1, amounts):
        if activities[compare][1] >= activities[x][2]:
            selections.insert(0, activities[x][0])
            compare = x
    return selections
```

Activity Selection Last-to-Start Merge Sort (Python):

```
"""
/*****************************************************************************
* Description: activity_sort function
* Functions receives a list of arrays from main and performs a merge sort
* to place the numbers in descending order based on start time.  The function
* takes and array and recursively devides until it reaches array sizes of one.
* It passes data to a merge function that combines the elements in
* decending order into the original array
*****************************************************************************/
"""

def activity_sort (amounts, activities):

    if amounts < 1:
        print("Array Size {} is invalid".format(arraySize))
    elif amounts == 1:
        return activities
    else:
        first = activities[:(amounts // 2)]
        last = activities[(amounts // 2):]
        firstArray = activity_sort(len(first), first)
        secondArray = activity_sort(len(last), last)
    return merge (activities, firstArray, secondArray)

"""
/*****************************************************************************
* Description: merge function
* Called by the mergeSort function to evaluate and merge the split array
* elements back into a single sorted array.  Receives the original array
* the split right and left arrays and their respective lengths.  Returns
* a sorted integer array to mergerSort
*****************************************************************************/
"""

def merge (dataArray, first, second):
    firstcount = secondcount = originalcount = 0
    while firstcount < len(first) and secondcount < len(second):
        if first[firstcount][1] > second[secondcount][1]:
            dataArray[originalcount] = first[firstcount]
            firstcount += 1
        else:
            dataArray[originalcount] = second[secondcount]
            secondcount += 1
        originalcount += 1
    while firstcount < len(first):
        dataArray[originalcount] = first[firstcount]
        firstcount += 1
        originalcount += 1
    while secondcount < len(second):
        dataArray[originalcount] = second[secondcount]
        secondcount += 1
        originalcount += 1
    return dataArray
```

Program Results: