**Brian Allison**

**10/22/2017**

**CS325**

**Assignment 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?

**Answer:**

We start with an array of classes with start and end times. Also, we start with an empty array that will have a min-heap structure representing lecture halls, prioritized by the end time of the lecture. We will use m to represent the total number of lecture halls needed.

First, we sort the array of classes by start time, using merge sort. Then, we loop through the array of classes. Since the min-heap is empty on the first iteration, we increment m and add the first class’s end time fj to the priority queue. On all remaining iterations, we check if the start time of the current class sj conflicts with (is earlier than) the end time of the earliest ending class, the root node of the min-heap. If there is a conflict, we increment m and add fj to the min-heap. If there is no conflict, we remove the root of the min-heap, and we add fj to the min-heap without incrementing m. At the end of execution, m will be the fewest number of lecture halls needed to accommodate all of the classes.

Time complexity: Merge sort takes nlgn time. Also, it takes n time to cycle through the array of classes. And, in the worst case, it takes lgn time to add an element to the min-heap. We then have nlgn + nlgn = 2nlgn. Based on the rules of asymptotic analysis, the runtime for this algorithm is ϴ(nlgn).

**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 x1< x2< ... < xn. 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:**

We call a road\_trip function passing in the set of distances x and the max distance d. Within this function, we would call a \*modified\_binary\_search (described below) to locate the largest possible distance in x that is not greater than d. When a distance xi is returned by our modified binary search, that is the distance to the next destination that we travel to. We then take the sum of d and xi and that becomes our new max (d = d + xi). The road\_trip function is then executed again using d and the subset of distances to the left of xi. The recursive calls continue until our modified binary search returns xn. When this happens, we are close enough to our final destination to get there without driving more than d miles from the previous location, and we now know which locations we need to travel to in order to minimize the number of days it takes to get to our final destination.

\*Modified\_binary\_search: Binary search is called with parameters d and the array of distances x. If the middle index (size/2) holds a value that is greater than d, we recursively call the modified binary search on all values to the left of the current index. If the middle index is less than d, we check the index to the right. If the index to the right is greater than d, we have found our destination. If the index is less than d, we continue the search on the left side of the current index. We continue the search until: we arrive at an index that has a value that is less than d with the one to the right being more than d, or we arrive at an index that has a value that is less than d and no value to the right of it, or we reach the end of the binary search without finding a value.

Time-complexity: In a worst-case scenario, we would need to travel to every location x before reaching our final destination. Also, binary search is lgn time. Then, in the worst case, the time to execute this algorithm would be ϴ(nlgn).

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

For each 1 ≤ 𝑖 ≤ 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?

**Answer:**

Assume we have an empty set z that will contain the jobs in the order they should be executed. We will use merge sort to sort jobs by penalty pi from largest to smallest. This sorted array will be x. We will loop through x, starting with the first index i, and use the deadline di­ of the job at index i to calculate which indexes we check in z. We will then check for free indexes in z, starting at j = di – 1 and moving toward j = 0. We want to find the highest index in z that is less than our deadline di. If an index in z is free, we assign the job to execute during that minute by adding it to that index of z. If the index is not free, we continue checking for a free index that is less than our deadline, moving towards the 0th index. If there are no free indexes, we add di to a penalty set y, to be executed only after the jobs that will not incur a penalty. We would continue looping through x until all jobs have either been added to z or y.

Time-complexity: The outer loop will iterate n times and the inner loop will iterate n or less times. Since we are dealing with a nested loop, the run-time will be O(n2).

We do not need to consider the run-time of merge sort in this case, since it is a lower order term.

**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.

References: Followed lecture 3 from this week to help understand proofs for this problem.

**Answer:**

We must show that the algorithm uses the greedy choice property and has optimal substructure.

Greedy Choice Property: We must show that there is an optimal solution that begins with a greedy choice (the activity with the latest start time). Suppose we have a set S of activities, and A is a subset of S that is an optimal solution to the Activity Selection problem for S. The activities in A are ordered by start time in descending order (later start times come first). Also, suppose that k is the first activity in set A. If ks  (the start time of k) is the latest start time, then obviously this is the greedy choice.

Now, suppose ks is less than or equal to the start time of another activity xs. Since we know that xs ≥ ks, we could remove k and replace it with x for an alternative optimal solution.

B = A – {k} U {x}

xs ≥ ks so B is also an optimal solution

Optimal substructure: Suppose that set A is an optimal solution for set S. Also, A’ = A – {1} and S’ is all elements in S with lower start times than the first item in A. Then, A’ must be an optimal solution to S.

If there was a solution to S’ that was more optimal than A’, then we could add {1} to that solution and have a solution to S that was more optimal to A. However, this contradicts what we already know, that A is an optimal solution to S. Therefore, this problem has optimal substructure.

1

**CS 325- Homework Assignment 4**

**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.*

Answer:

Description:

First we want to sort the set of activities using merge sort. We will sort in descending order by start time, since we are working backwards starting from latest start time. Let the sorted set of activities be S. Once S is sorted, we can add the activity with the latest start time to our solution set. Let the solution set be A. A now contains the first element of S. We will then compare the start time of the most recent (last) element in A to the finish time of the next element in our sorted set S, starting from the element in S that occurs after the first. If the most recent element in A is greater than or equal to the next element of S, it means that the times are overlapping and we need to move on to the next element of S. Otherwise, if the times are not overlapping, we can append that element of S to A because it is part of our solution. This continues until we reach the last element in S. After that, A is a complete and optimal solution.

Pseudocode:

**activity\_selector(sorted\_array)**

current = 0

solution\_array = sorted\_array[current]

j = 1

for j to length of sorted\_array

if start time of solution\_array[i] >= finish time of sorted\_array[j]

append sorted\_array[j] to solution\_array[i]

current = j

**Read list from file and pass to merge sort and activity selector:**

until end of file is reached:

array = [] (new array for each list)

for i = number of items in activity set

activity = new activity object(id, start, finish)

append activity to array

sorted\_array = merge sort(array) (sort by start time)

Reverse order of list to get descending order

solution\_array = activity\_selector(sorted\_array)

Running Time:

The activities are sorted by start time using merge sort which sorts in nlgn time. We then do one iteration for each activity in the sorted set. This gives us nlgn + n. Based on the rules of asymptotic analysis, the worst case for this algorithm is ϴ(nlgn).