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 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:

We start with a set of classes with start and end times. Also, we start with an empty set 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 set of classes by start time, using merge sort. Then, we loop through the set of classes. Since the min-heap is empty on the first iteration, we increment m and add the first class's end time f_j to the min-heap. On all remaining iterations, we check if the start time of the current class s_j 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 f_j to the min-heap. If there is no conflict, we remove the root of the min-heap, and we add f_j 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 $\Theta(nlgn)$.

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 $x_1 < x_2 < \ldots < x_n$. 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:

Assuming the list of hotels is already sorted from closest to farthest…

We call a road_trip function passing in the set of distances x and the max distance d. We then loop through the possible destinations and we stop when we get to a distance in x that is greater than d. We then travel to the hotel comes before the one that is further than d, adding that to our solution set. We then take the sum of d and x_i (the distance we just traveled) and that becomes our new max ($d = d + x_i$). We then continue the loop using d and the subset of distances that are larger than (to the right of) the current x_i where we stayed over night. This continues until the loop reaches x_n. 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.

Time-complexity: In a worst-case scenario, we would need to check or travel to every location x before reaching our final destination. So, in the worst case, the time to execute this algorithm would be $\Theta(n)$.

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:

Assume we have a set z with n number of empty indexes that will eventually contain the jobs in the order they should be executed. We will use merge sort to sort jobs by penalty p_i from largest to smallest. Let this sorted set be x. We will loop through x, starting with the first index i, and use the deadline d_i 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 $j = d_i - 1$ and move toward $j = 0$. We want to find the highest index in z that is less than our deadline d_i. 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 0^{th} index. If there are no free indexes, we add d_i 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. All jobs in y will then be added to z, starting with our first non-empty element in z, to complete our solution set z.

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 $O(n^2)$.

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 k_s (the start time of k) is the latest start time, then obviously this is the greedy choice.

Now, suppose k_s is less than or equal to the start time of another activity x_s. Since we know that $x_s \geq k_s$, we could remove k and replace it with x for an alternative optimal solution.
B = A – {k} U {x}
$x_s \geq k_s$ 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.

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.*

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

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 $\Theta(nlgn)$.