Christopher Matian
CS325 - Spring 2019
04/26/2019

Problem One: Class Scheduling
For this problem we know three things: the class (C), the start time (S), and the finish time (F).

The best method of solving this scheduling problem is to sort the list of classes by S to F, then loop through each class and compare the S time of the current class with the F time of the previous class.

If the current classes start time is less than the finish time of the previous class we know we need another lecture hall represented by a counter starting at 1 (we need at least one lecture hall). We just increment the counter each time there's overlap.

Breaking this down:

1) Sort the list of classes
2) For each class
   a) Check if there's overlap in the S and F times (is F > S)
   b) If there is increment the counter
3) Update the previous class
4) Repeat step 2 - 3 until there's no more classes left
5) Return the number of halls you need (the counter).

The running time of the algorithm would usually be **O(n)** but because we're sorting before evaluating the list the run time is **O(n logn)**.

Problem Two: Scheduling Jobs with Penalties
The important thing to evaluate the problem against is the penalties of each job. So to start we sort the list of jobs by penalties in descending order. The penalty is described as j and if j = 1 we know that the job needs to happen now. If it's j > 1 then we know that j - 1 jobs can be scheduled without incurring a penalty.

Breaking this down:

1) Your inputs are two arrays deadlines (d) and penalties (p).
2) Sort the jobs by penalty in desc. order.
3) For each job (j) where j is a list of jobs: {j1, j2, …., jn}:
   a) If the deadline of j is equal to or i is equal to n:
      i) Schedule the job j at this time (i)
   b) Else call the algorithm iteratively on the list of jobs: {ji, ji+1, …. , jn}

The run time of the algorithm will be similar to problem one… **O(n logn)** because of the sorting that's handled at the beginning. One linear loop through the rest of the job list will take care of the problem.

Problem Three: CLRS 16-1-2 Activity Selection Last-to-Start
We need to figure out the largest set of activities that have no overlap, starting with the the activity ending instead of the beginning. The following considerations can be made:

- S = {a_1, a_2, … , a_n}
- Each a_i is equal to [s_i, f_i)

An algorithm that can find the optimal solution would be as follows:

```
last_to_start(s, f):
    n = length of s
    A = {a_n}
    k = n
    while (m = n - 1 and m > 1):
        if (f[m] <= s[k] ):
            A = A UNION A_m (add A_m to activity)
            k = m
    return A
```

The algorithm will take an input of A and evaluate its start (S) and finish (F) times. It then looks through each activity in descending order (starting from the last activity to start last) and finds a compatible activity to add to A. The algorithm is trying to find an optimal solution at **each stage** so it's essentially a greedy algorithm.

Problem Four: Last-to-start implementation (written portion)
My algorithm is nearly verbatim to the pseudo-code above except for the addition of code for reading from the input file. Also, instead of accepting start,finish times (usually 2 arrays) separately, it accepts a list of arrays which are grouped together from reading in the text file.

Instead of a while-loop it reverses the passed in array and loops through it (using python's native for-in-range loop). It then compares the finish time of the current object to the previous item's start time by checking the grouped element's indexes. If a compatibility is found it pushes the group number to the array which is returned when the loop completes.

The run time of the algorithm should be similar to above **O(n logn)**. One loop of the array suffices and because it checks each stage of the data it functions as a Greedy Algorithm. The sorting at the beginning of the algorithm dominates the linear run-time, hence why it's O(n logn).