CS3510-A: Design and Analysis of Algorithms, Spring 2021

Homework-6

February 23, 2021

DUE DATE: Tuesday, March 2, 11:59pm

Note-1: Your homework solutions should be electronically formatted as a single PDF document that you will upload on Gradescope. If you have to include some handwritten parts, please make sure that they are very clearly written and that you include them as high resolution images.

Note-2: Please think twice before you copy a solution from another student or resource (book, web site, etc). It is not worth the risk and embarrassment.

Note-3: You need to **explain/justify** your answers. Do not expect full credit if you just state the correct answer.

Note-4: You will get 2 extra points if you submit electronically typed solutions instead of hand-written.

Problem-1 (30 points)

You are consulting for Cyclotron, a company that does a large amount of shipping between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed maximum amount of weight they are allowed to carry, C. Boxes arrive at the New York facility one by one, and each package i has a weight p_i. The facility is quite small, so at most one truck can be at the station at any time. Cyclotron requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after theirs make it to Boston faster. The company uses a simple greedy algorithm for packing: Boxes are packed in the order they arrive, and whenever the next box does not fit they send the truck on its way.

The board of directors is wondering if they might be using too many trucks, and they have hired you to see whether the situation can be improved. Their thinking is as follows: "Maybe one could decrease the number of trucks needed by *sometimes* sending off a truck that was *less* full, and in this way allow the next few trucks to be better packed."

You must prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use minimizes the number of trucks needed to ship these boxes. Your proof should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other possible solutions.

Solution

For a solution from the greedy algorithm to not be optimal it has to be because the optimal solution has a lower number of total trucks. In order for there to be a lower number of trucks the optimal solution must either involve a solution where a truck is left less full (as described in the problem description by the directors' suggestion) or a solution where the optimal solution involves moving onto the next truck later in the list of packages and filling the current truck more.

In the first case (leaving a truck slightly empty / moving onto the next truck before the greedy solution would have), this would require moving one package k from the truck it appears in for the greedy solution t_k to the next truck t_{k+1} for the supposedly optimal solution. However this would result in either overpacking t_{k+1} or overflowing to a new truck down the line, which would contradict the opitimality of this proposed solution.

In the second case (an optimal solution that involves filling a truck further than the greedy solution did before moving onto the next truck), this is not possible because the greedy solution fills up the trucks as much as possible while still filling the trucks in the order the packages arrived. Therefore all trucks will be filled as much as possible without skipping over packages in the list.

Problem-2 (40 points)

Your friend is working as a camp counselor, and they are in charge of organizing activities for a set of campers. One of their plans is the following mini-triathlon exercise: each camper must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the campers out in a staggered fashion, via the following rule: the campers must use the pool one at a time. In other words, first one camper swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second camper begins swimming the 20 laps; as soon as they are out and starts biking, a third camper begins swimming...and so on.

Each camper has a projected swimming time (the expected time it will take them to complete the 20 laps), a projected biking time (the expected time it will take them to complete the 10 miles of bicycling), and a projected running time (the time it will take them to complete the 3 miles of running). Your friend wants to decide on an order in which to sequence the starts of the campers. Let's say that the completion time of an order is the earliest time at which all campers will be finished with all three legs of the triathlon, assuming they each spend exactly their projected swimming, biking, and running times on each of these three parts. Give an algorithm that produces an order whose completion time is as small as possible in $O(nlog(n))$ time.

Note: You must show your algorithm's running time with respect to the number of campers.
Note: You must show your algorithm's optimality.

Solution

Process:

1 - Sort the list of campers in decreasing order of $b + r$ (where b, r are expected bike times and expected running times)
2 - Start the campers in the order of this sorting

Explanation:

The total sum of all expected swim times will be less than the total possible duration of the race, and by nature of only one camper being able to use the pool at a time this full sum is inevitably added to the total duration. Therefore the absolute minimum possible duration will be the total swim time of all campers plus the biking and running time of the camper with the shortest combined biking and running time. In order to optimize for this value the fastest $b+r$ camper should go last and all other campers will need a slight head start to correct for the difference in $b + r$ times. By this method the only way that the total race duration can exceed the sum of swim times plus the smallest $b + r$ is if there is a camper has a longer $b + r$ time than the sum of all swimmers after them plus the $b+r$ of the last/fastest camper. If such a camper exists then they will need to be as far up in the order as possible. By placing them only after any campers with a longer/slower $b+r$, then they will be placed in a position where they are extending the total duration by a minimum amount possible without forcing the other campers above them to extend the total duration as well.

Runtime:

The only operation over the list of campers is sorting them in reverse order by $b + r$. This can be done in $O(nlogn)$ (where n is the total number of campers), therefore this algorithm has a runtime of: $O(nlogn)$

Problem-3 (30 points)

Consider the following variation on the Interval Scheduling Problem. You have a supercomputer that can operate 24 hours a day, every day. People submit requests to run **daily** jobs on the supercomputer. Each such job request includes a fixed and pre-determined *start time* and an *end time*. If a job request is accepted it must run continuously, every day, for the period between the requested start and end times. Jobs can begin before midnight and end after midnight. (This makes for a type of situation different from what we saw in the Interval Scheduling Problem.) The supercomputer can run at most one job at any given point in time; no two jobs can run concurrently.

Given a list of n such job requests, provide an algorithm to accept as many job requests as possible (regardless of their length). Your algorithm should have a running time that is $O(n^2)$. You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by (start-time, end-time) pairs.

$$(6P.M., 6A.M.), (9P.M., 4A.M.), (3A.M., 2P.M.), (1P.M., 7P.M.).$$

The optimal solution would be to pick the two jobs (9P.M.,4A.M.) and (1P.M., 7P.M.), which can be scheduled without overlapping.

Note: You must show your algorithm's running time is $O(n^2)$.

Solution

Algorithm 1 24 Hour Schedule

```
J = all jobs sorted by finish time
S = empty set
for j in J do
    A = copy of J
    remove j from A
    remove jobs that overlap with j from A
    x = start time for j
    split/flatten the schedule cycle at time x
    c = {j} (the current schedule)
    for t in A do
        if t has no conflicts in c then
            add t to c
        end if
    end for
    if c is larger than S then
        S = c
    end if
end for
return: S
```

Explanation:

The algorithm finds the schedule with the maximum number of jobs when splitting the 24 hour cycle at each job and starting the schedule with that job. First the jobs are sorted by their finish time. Then for each job j in the set of jobs J the cycle is split at the start time of j and j becomes the first job in output set. Now new jobs are added to the output set by considering them in order of finish time (where the order by finish time has been adjusted/wrapped to reflect the newly split/flattened 24 hour schedule). This will result in an optimal schedule that includes the job j. If the output set is larger than the sets generated for any other starting job j then the current output set is tracked as the maximum and then returned once all jobs j have been considered and had optimal schedules built for them. By considering every job j and splitting the cycle at the start of j the problem is turned into every relevant version of the interval scheduling problem.
An optimal solution to the 24 hour cycle problem will have some "starting" job a. By splitting the cycle/jobs at the start of a then the optimal solution starting with job a will be the same solution or an equally optimal solution to the optimal solution for the 24 hour cycling problem. By considering every possible task as the start of the normal interval schedule problem the optimal solution involving a will inevitably be found.

Runtime:

The algorithm starts by sorting all jobs by their finish time - $O(nlogn)$
The algorithm then has two nested for loops that iterate over all jobs - $O(n^2)$
- NOTE: All items inside the two nested for loops can be done in $O(1)$

Therefore, the runtime for this algorithm has a runtime of: $O(n^2)$, where n is the total number of jobs given