CS 577: Introduction to Algorithms Fall 2019

Homework 6 Solutions

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

You are given n jobs, each comprised of a time to complete t_i and a value w_i denoting its importance. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. Your objective is to schedule the jobs in a way that minimizes the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$. Design an $O(n \log n)$ algorithm to solve this problem.

We'll try to solve this problem using a greedy approach. This means that we'll order the jobs by choosing, at each step, the job that looks best at the moment. But how will we define 'best'? We could try just choosing the job with the greatest weight first, reasoning that the most important jobs should be done first. But this could get us into trouble if the job of greatest weight is so much longer than the other jobs that it leads to tremendous dissatisfaction among our other customers. We might also try choosing the shortest job first, but this would not work if we had a longer job whose weight was dramatically higher than the shorter jobs. Next, we could consider both the weight and the duration of a job in deciding which one is the current best choice.

We'd like to favor jobs with high weights, but we'd also like to favor jobs with short times. Thus, we should look for a quantity that increases as a job's weight increases, and also increases as a job's duration decreases. To help us find the right such quantity, let's analyze the problem with two jobs. Scheduling job 1 first is best iff

$$w_1 t_1 + w_2(t_1 + t_2) < w_2 t_2 + w_1(t_1 + t_2). \tag{2}$$

In case of equality in (2), both schedules are equally good. We can rewrite (2) as

$$w_2 t_1 < w_1 t_2$$
$$\frac{w_2}{t_2} < \frac{w_1}{t_1}.$$

Thus, for the two-job problem, the jobs are in the proper order if the *ratio* of weight to duration is no bigger for the later job than for the earlier job. This suggests that we use the greedy approach of always choosing the job with the greatest ratio of weight to duration. That is, if J is the list of jobs remaining to be run, we'll choose the first job i (that is, the earliest job in our list) such that

$$\frac{w_i}{t_i} \quad = \quad \max_{j \in J}\left(\frac{w_j}{t_j}\right)$$

We'll have to show that this approach minimizes the weighted sum $\sum_{i=1}^{n} w_i C_i$. We'll do this with an exchange argument, in which we show that any ordering can be converted into the greedy ordering without increasing the cost.

Let us number the jobs from 1 to n in the order of the greedy algorithm. An inversion in a schedule is a pair $i > j$ such that job i is scheduled before job j. Because of our choice of the greedy criterion, this means that either $w_i/t_i < w_j/t_j$, or else that $w_i/t_i = w_j/t_j$ but j comes earlier in the original list of jobs than i (remember, we decided to break ties by choosing the earlier job first). By definition, an ordering different from the greedy ordering has at least one inversion, and by the argument given in the text, any sequence with an inversion has a pair of consecutive elements that constitute an inversion.

Now, consider any pair of consecutive jobs in some schedule, say job i and job j, such that $i > j$. How does swapping jobs i and j affect the cost of the schedule? First of all, it does not affect the contributions of the jobs other than i and j to the overall cost. As for the contribution of jobs i and j, let C_{i-1} be the cumulative times of all jobs before job i (and say $C_0 = 0$). Before the swap, the contribution of jobs i and j to the cost was

$$w_i(C_{i-1} + t_i) + w_{i+1}(C_{i-1} + t_i + t_{i+1}),$$

whereas after the swap it is

$$w_{i+1}(C_{i-1} + t_{i+1}) + w_i(C_{i-1} + t_{i+1} + t_i).$$

Working out the difference (as for (3) above), we see that the contribution does not increase provided

$$\frac{w_j}{t_j} \leq \frac{w_i}{t_i},$$

which is the case by our choice of greedy criterion. This way, we can undo all inversions and end up with the greedy schedule without increasing the cost. Thus, the greedy schedule has the minimum cost.

We should also consider the efficiency of our algorithm. We'll need to compute the ratio of weight to duration for each job, which will take $O(n)$ time. We can then sort the jobs based on this ratio, which will take $O(n \log n)$ time. This gives us a total running time of $O(n)+O(n \log n) = O(n \log n)$.

Problem 2

You see the following special offer by the convenience store: “A bottle of cola for every 3 empty bottles returned.”, and you want to find out the maximum number of colas you can drink if you buy N bottles of cola. You are able to borrow empty bottles from a friend (as many as you want), but you have to give them back at the end. Design an algorithm for this problem that runs in time bounded by a polynomial in the bit length of N, i.e., a polynomial in $\log N$.

The key observation is the following: Whenever there are three or more non-borrowed bottles, it never hurts to postpone borrowing empty bottles. We can formally use an exchange argument to justify this. Suppose we use $k \leq 3$ borrowed bottles to get another one. Because we have three or more non-borrowed bottles, we can replace these k borrowed bottles with non-borrowed ones and still get another one. In both cases, we get the same extra cola, so nothing is lost via this exchange.

Let $\mathrm{OPT}(N)$ denote the maximum number of colas we can drink with N bought bottles. For $N \geq 3$, the above observation shows that

$$\mathrm{OPT}(N) = 3 + \mathrm{OPT}(N-2) \tag{1}$$

That is, drinking 3 bought bottles gives one back, and we can continue the process in the next round as if we had bought $N-2$ bottles, namely $N-3$ that were actually bought and the one that we got back in return for the 3 empty bottles. For the base cases, since $N \geq 1$, we need to consider $N = 1, 2$.

$N=1$ We need to borrow at least two empty bottles in order to get at least one extra bottle. However, if we borrow $k \geq 2$ empty bottles and use at least two of them to get an extra bottle, we have $(k+1)-3+1 = k-1$ bottles in total in the next round. Since our total number of bottles cannot increase over time, this means we'll be able to return at most $k-1$ empty bottles at the end, whereas we need to return k. Thus, we cannot use any borrowed empty bottles, and the best we can do is just drink the one bottle we bought: $\mathrm{OPT}(1) = 1$.

$N=2$ We can drink our two bottles, borrow an empty and turn in the three empty bottles for a new full one, drink the latter, and return it empty. This shows that $\mathrm{OPT}(2) \geq 3$. In order to do better, we'd need to turn in three empty bottles for a new full one at least twice (so we need at least $k \geq 3$ borrowed bottles if we buy 2), which would reduce our total number of bottles from $k+2$ (k borrowed empty bottles and 2 full bought ones) to $(k+2)-3+1 = k$, which then becomes $k-3+1 = k-2$. This makes it impossible to return k empty bottles at the end. Thus, $\mathrm{OPT}(2) = 3$.

Solving (1) gives that

$$\mathrm{OPT}(N) = \begin{cases} 3n & \text{if } N = 2n \\ 3n+1 & \text{if } N = 2n+1 \end{cases}$$

for some $n \geq 1$. A single formula for this is $\lfloor \frac{3N}{2} \rfloor$ since

$$\left\lfloor \frac{3N}{2} \right\rfloor = \begin{cases} \left\lfloor \dfrac{3 \cdot 2n}{2} \right\rfloor = 3n & \text{if } N = 2n \\ \left\lfloor \dfrac{3 \cdot (2n+1)}{2} \right\rfloor = \left\lfloor 3n + \dfrac{3}{2} \right\rfloor = 3n+1 & \text{if } N = 2n+1. \end{cases}$$

Therefore given N, one can directly return $\lfloor \frac{3N}{2} \rfloor$.

Complexity Arithmetic operations can be done in time polynomial in the length of N (encoded in binary), therefore one can compute $\lfloor \frac{3N}{2} \rfloor$ in time polynomial in $\log N$, as desired.

Problem 3

In a city there are n bus drivers. There are also n morning bus routes and n afternoon bus routes, each with various lengths. Each driver is assigned one morning route and one evening route. For any driver, if his total route length for a day exceeds d, he has to be paid overtime for every hour after the first d hours at a fixed rate per hour. Your task is to assign one morning route and one evening route to each bus driver so that the total overtime amount that the city authority has to pay is minimized. Design an $O(n \log n)$ algorithm for this problem.

Since each bus driver is allowed to work exactly d hours before requiring overtime pay, it makes intuitive sense that in order to minimize total overtime pay, the available hours of driving should be spread as evenly as possible among the n drivers. Each morning route and each evening route has to be taken by some driver, so the problem then becomes how to pair each morning route with a corresponding evening route in such a way that the total driving times are as even as possible. The natural way to do this is to pair the longest morning route with the shortest evening route, the second longest with the second shortest, and so on where the i-th longest morning route is paired with the i-th shortest evening route for all $i \in [n]$ (and ties in route length are broken arbitrarily)–we will call the schedule generated by this strategy the "greedy schedule". As it turns out, this strategy works, and we will use an exchange argument to prove that this is the case.

First, we give a high-level overview of the proof technique we will use. Sort the morning routes in nondecreasing order of driving time. Any schedule can be uniquely written as an ordered list of evening routes (where the first evening route is paired implicitly with the first morning route, and so on). Then, given any schedule that is not the greedy schedule, there must be at least one pair of consecutive evening routes that is inverted with respect to greedy schedule (that is, the two routes appear in the opposite order in the greedy schedule, though not necessarily consecutively). Exchanging the order of these two evening routes would reduce the total number of inversions in the schedule by 1, so since the schedule can have only a finite number of inversions to begin with, there must be a finite sequence of such exchanges that transforms the schedule into the greedy schedule. Hence, if we can show that no such exchange increases the total overtime, the greedy schedule can be no worse than any other schedule (and hence must be optimal).

Claim 1. *Given a schedule which contains a pair of consecutive evening routes that is inverted with respect to the greedy schedule, swapping the pair of routes does not increase the total overtime that must be paid in the schedule.*

Proof. Let e and e' be the lengths of the evening routes to be exchanged, and let $m \leq m'$ be the lengths of the corresponding morning routes, where we assume that m precedes m' in the fixed order. Since in the greedy schedule the evening routes are sorted in nonincreasing order, in order for e and e' to be inverted we must have $e \leq e'$. In order to determine the effect that swapping has on total overtime, we consider a number of cases.

- $m' + e' \leq d$. Note that $m + e \leq d$ as well since $m \leq m'$ and $e \leq e'$, so both pairs of routes fall under the overtime threshold before the exchange. After the exchange, $m + e'$ and $m' + e$ are both upper bounded by $m' + e'$ (also since $m \leq m'$ and $e \leq e'$) and hence d, so neither contributes overtime after the exchange either.

- $m + e \geq d$. In this case, both pairs of routes require overtime pay, so the total overtime is $m + e + m' + e' - 2d$ before the exchange. Afterwards, $m + e'$ and $m' + e$ are both at least as large as $m + e$, so the total overtime is still $m + e + m' + e' - 2d$.

- $m + e < d < m' + e'$. Refer to Figure 1 for a pictorial representation of this case. Imagine that the $(m' + e')$-length pairing is in fact made up of a segment of length $m' + e$ and another of length $e' - e$, while the $(m + e)$-length pairing is a single segment of length $m + e$. Then, swapping e and e' is equivalent to moving the segment of length $e' - e$ from the first pairing to the second. The $(m' + e)$- and $(m + e)$-length segments still contribute the same amount to total overtime, but the $(e' - e)$-length segment contributes the same or less, regardless of how much of it contributed to overtime before the swap, since $m + e \leq m' + e$. Hence, the total overtime stays the same or decreases.

Hence, in every case swapping does not increase the total overtime, so the claim follows. $\square$

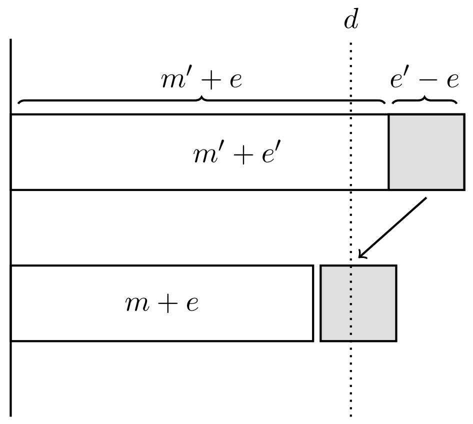

Figure 1: Effect on total overtime of swap when $m + e < d < m' + e'$.

Now, armed with claim, we can follow the exchange argument discussed above to get that the greedy schedule is optimal, as desired. To actually construct the greedy schedule, it suffices to sort the morning routes in increasing order and the evening routes in decreasing order, which takes $O(n \log n)$ time.

Problem 4

The route from your house to work is composed of n parts of length $l_i > 0$ with different speed limits $v_i > 0$. You are given positive integers k and v, where k indicates how many times you are allowed to break the speed limit by v. Your objective is to find at which parts of the route you should break the speed limit to minimize the total time required to get to work. You have some ideas of how to do this.

(a) A reasonable choice is to use it on the longest parts of the route, that is sort the list with respect to l_i (largest first) and use the bonus speed on the first k parts of the sorted list. Provide a counter-example for this greedy strategy.

(b) Another option is to use it on the parts of the route with smallest speed limit v_i, that is sort the list with respect to v_i and use the bonus speed on the first k parts of the sorted list. Provide a counter-example for this greedy strategy.

(c) Design an $O(n \log n)$ algorithm that takes as input the list of lengths and speed limits

$$L = [(l_1, v_1), \dots (l_n, v_n)]$$

and two positive integers k, v and computes the minimum time to go from your house to work. You need to provide a proof that the algorithm finds the minimum travel time.

Part (a)

Consider the instance with $k = 1$, bonus speed $v = 10$ and $L = [(100, 40), (50, 10)]$. The proposed greedy strategy uses the bonus speed on the first part and in this case the total time is $100/(40 + 10) + 50/10 = 7$. On the other hand, if we use the bonus speed on the second part we get a total time of $100/40 + 50/(10 + 10) = 5$.

Part (b)

Consider the instance with $k = 1$, $v = 10$, and $L = [(40, 10), (180, 20)]$. The proposed greedy strategy uses the bonus speed on the first part of the route, making the total travel time $40/(10 + 10) + 180/20 = 11$. However, if we use the bonus on the second part we obtain that the travel time is $40/10 + 180/(20 + 10) = 10$.

Part (c)

We first investigate the case with only two intervals and $k = 1$. In this case, we have that in order to choose the first interval it must be true that

$$\frac{\ell_1}{v_1 + v} + \frac{\ell_2}{v_2} \leq \frac{\ell_1}{v_1} + \frac{\ell_2}{v + v_2} \Leftrightarrow \frac{\ell_1}{v_1(v_1 + v)} \geq \frac{\ell_2}{v_2(v_2 + v)}$$

Therefore, a reasonable choice is to sort the intervals according to non-increasing $\ell_i/(v_i(v_i + v))$, and use the bonus speed on the first k intervals of the sorted list. We can prove that this strategy is optimal using an exchange argument. We simply denote a solution to this problem by the set of the indices of the intervals of the trip that the solution uses the bonus speed on, that is

$S \subseteq [n] \doteq \{1, \ldots, n\}$. Moreover, let t_i be the time that it takes us to travel part i *without* using the bonus speed, namely $t_i = \ell_i/v_i$. The time to travel part i using the bonus speed is $t'_i = \ell_i/(v + v_i)$. Without loss of generality, we may assume that S contains exactly k elements. That is because if S contains fewer, then we can use the bonus speed on any of the remaining intervals and get a solution with smaller travel time. Now, assume that there exist $i \in S$ and $j \in [n] \setminus S$ such that

$$\frac{\ell_j}{v_j(v_j + v)} > \frac{\ell_i}{v_i(v_i + v)}$$

From the same computation as for 2 intervals and $k = 1$ we have that this inequality is equivalent to $t'_i + t_j > t_i + t'_j$. Since the time that we spent on the intervals other that i and j did not change we have that the total time of $(S \setminus \{i\}) \cup \{j\}$ is not worse than the total time of S. Now, we can continue making these changes until there exists no pair i, j with $i \in S$ and $j \in [n] \setminus S$ such that $\frac{\ell_j}{v_j(v_j+v)} > \frac{\ell_i}{v_i(v_i+v)}$. Then S contains the k largest elements of L with respect to our measure, which is exactly what the greedy strategy does. Notice, that if two parts i, j of the route have $\frac{\ell_j}{v_j(v_j+v)} = \frac{\ell_i}{v_i(v_i+v)}$, it does not matter which one of them we pick to use the bonus speed on.

Alternate solution Another, way to see this is by using the objective that we are trying to minimize, namely the total time of the trip. Let $\delta_i = t_i - t'_i$. The total time for this solution is

$$T_S = \sum_{i \in S} t'_i + \sum_{i \in [n] \setminus S} t_i = \sum_{i \in S} (t_i - \delta_i) + \sum_{i \in [n] \setminus S} t_i = \sum_{i=1}^{n} t_i - \sum_{i \in S} \delta_i$$

So, in order to minimize the total travel time, we want to maximize the $\sum_{i \in S} \delta_i$. To do this we can sort the intervals with respect to δ_i in decreasing order and choose the first k to use the bonus speed. In particular, we have that

$$\delta_i = t_i - t'_i = v \frac{\ell_i}{v_i(v + v_i)}.$$

Since, v is an absolute constant we have that the sorted version of L decreasingly with respect to δ_i is the same as when we sort with respect to $\frac{\ell_i}{v_i(v+v_i)}$.

Analysis To implement this greedy strategy we just need to sort the list of the tuples (ℓ_i, v_i) by decreasing value of $\ell_i/(v(v + v_i))$. This can be done in $O(n \log n)$ time using Merge Sort. Then we can do a linear pass over the sorted list to compute the total travel time by adding the time that we spend on each part of the route, for the first k parts we compute the time as $t'_i = \ell_i/(v_i + v)$ since for these we are using the bonus speed, for the others we compute it as t_i. This takes $O(n)$ time. Overall, the run time of our algorithm is $O(n \log n)$.

Problem 5

Your friend is organizing a mini-triathlon exercise: each contestant must swim, bike, and then run a fixed distance for each activity. Only one participant can use the pool at any given time, and as soon as one participant finishes swimming and starts biking, another one is able to start swimming. Each participant has a projected swimming, biking and running time. Your objective is to determine a schedule in which the participants start the activity that minimizes its completion time (the time required for all participants to finish all three activities if they spend exactly their projected times for each one). Note that participants can bike and run simultaneously, but at most one person can be in the pool at the same time.

Design an $O(n \log n)$ algorithm that produces a schedule whose completion time is as small as possible.

The situation described in this problem is similar to some scheduling problems we've seen. The only resource for which there is any conflict is the pool so in essence our problem is simply to schedule the swimming times. The fact that it's the finishing time of the *last* finisher that we're interested in (as opposed to, say, the average finishing time) should ring a bell – it sounds like the problem of minimizing overall lateness. If we give every contestant a deadline of 0, their lateness equals the time by which they finish swimming. That's not really the quantity we would like to min-maximize since a contestant doesn't finish when he/she leaves the pool but only after the additional stretch of time he/she needs for biking and running. However, if we give every contestant a deadline of minus his/her combined biking and running times, then the lateness equals the time by which he/she finishes the race. Thus, we obtain a reduction to the problem of minimizing the maximum lateness.

Let's go over the reduction in more detail. Given a set of n contestants, and projected swimming, biking and running times s_i, b_i, and r_i, respectively, for $i \in \{1..n\}$, we'll construct an instance of the minimizing-lateness problem. Remember that for an instance of the minimizing lateness problem, we'll need to specify n jobs, each with an execution time t_i and a deadline d_i. We'll specify those as follows: for each contestant i, let $t_i = s_i$, and $d_i = -(b_i + r_i)$. This, of course, gives us negative deadlines but that's no problem – if, for some reason, we'd prefer not to have negative deadlines, we could always make them nonnegative by adding the absolute value of the smallest (i.e., most negative) deadline to all of the deadlines. The addition would not affect the optimality of a schedule.

We now must argue that a solution to this instance of the minimizing lateness problem will be an optimal triathlon scheduling. First, we point out that, since we're using the swimming times as the execution times, the lateness-minimizing algorithm will prevent two contestants from being in the pool at the same time, so we can be sure that the schedule it produces will at least be valid. But what quantity will actually be minimized? Let w_i be the time that contestant i must wait before getting into the pool. Then the time that contestant i 'finishes' (as far as the minimizing lateness problem is concerned) will be $w_i + s_i$. Since contestant i's deadline was $-(b_i + r_i)$, the 'lateness' of contestant i will be $w_i + s_i + b_i + r_i$, which is precisely the (actual) finishing time of contestant i.

Thus, minimizing the lateness in the instance we constructed is equivalent to minimizing the overall finishing time of the triathlon. Furthermore, this gives an efficient solution to the triathlon scheduling problem: We can construct the instance of the minimizing lateness problem by simply running through the times of each contestant and computing the contrived deadlines, which takes time $O(n)$, and then run our $O(n \log n)$ time greedy algorithm for minimizing lateness. The overall running time will be $O(n \log n)$.

We should also note that this solution suggests another one. The algorithm that we know for solving the minimizing lateness problem proceeds by choosing the earliest deadline first. Given how we've set up the deadlines, this means it will always choose the contestant with the longest combined biking and running times to enter the pool first. Thus, we have the following greedy algorithm for the problem: put the contestants into the pool in decreasing order of combined biking and running times. To prove the correctness of this approach, though, we'd need to go through a full exchange argument – it's easier to prove correctness with the reduction.