CS 577: Introduction to Algorithms Fall 2022

Homework 6

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers the greedy paradigm and exchange arguments in particular. **Problem 3 must be submitted for grading by 2:29pm on 11/1.** Please refer to the homework guidelines on Canvas for detailed instructions.

Warm-up problems

1. A small photocopying service with a single large machine to do its photocopying faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight $w_i \geq 0$ that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize 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. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

   Example: Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

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.

   For example, consider the case where $N = 8$. The straightforward strategy is as follows: After finishing your 8 bottles of cola, you have 8 empty bottles. Take 6 of them and you get 2 new bottles of cola. Now after drinking them you have 4 empty bottles, so you take 3 of them to get yet another new cola. Finally, you have only 2 bottles in hand, so you cannot get new cola any more. Hence, you have enjoyed $8 + 2 + 1 = 11$ bottles of cola.

   You can actually do better! You first borrow an empty bottle from your friend, then you can enjoy $8 + 3 + 1 = 12$ bottles of cola! Of course, to be fair, you have to return your remaining empty bottle back to your friend which you can do because you will have one left over bottle after drinking the final cola you get from the store.

   Design an algorithm to finds the largest number of bottles you can get when buying N of them and can borrow as many bottles as you want. Your algorithm should run in time bounded by a polynomial in the bit length of N, i.e., a polynomial in $\log N$.

Regular problems

3. [Graded] You're about to buy a new phone and want to determine the minimum amount of memory that is required to download and install your n favorite apps. The i-th app has a download size of d_i and a storage size of s_i. To download the app, your phone must have at least d_i megabytes of free disk space. After installation the app consumes s_i megabytes of disk space on the phone. The download size d_i is always at least as large as the storage size s_i but may be larger due to material that might not get used such as translations to different languages.

   The order in which you install the apps matters. Consider an example with $n = 2$, $(d_1, s_1) = (7, 4)$, and $(d_2, s_2) = (10, 6)$.

   - First downloading app 1 and then app 2 requires: 7 megabytes for downloading app 1, after which app 1 permanently consumes 4 megabytes of disk space, and then an additional 10 megabytes for downloading app 2, for a total of 14 megabytes. Thus, the maximum amount of memory needed at any point in time is 14 megabytes.
   - First downloading app 2 and then app 1 requires: 10 megabytes for downloading app 2, after which app 2 permanently consumes 6 megabytes of disk space, and then an additional 7 megabytes for downloading app 1, for a total of 13 megabytes. In this case the maximum amount of memory needed is 13 megabytes.

   For this example the answer would be $\min(14, 13) = 13$.

   Design on $O(n \log n)$ algorithm to determine the amount of memory needed to download and install all n apps in an optimal order.

4. In a city there are n bus drivers. There are also n morning bus routes and n evening 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.

5. Each morning you drive from your house to work. The route decomposes to n parts each of length $l_i > 0$. At each part there is a different speed limit $v_i > 0$. As usual, you are late for work and decide to break the speed limit in order to arrive faster. Since you do not want to risk much you decide to only break the speed limit by v at each part and only for k parts overall. For example, your route may be the list

   $$[(30\text{mi}, 20\text{mph}), (40\text{mi}, 70\text{mph}), (10\text{mi}, 30\text{mph})],$$

   where each tuple contains the length l_i and the speed limit v_i of each part. You decide that you only want to break the speed limit 2 times, that is $k = 2$ and by speed $v = 10$mph. A possible way to do this route is for example to drive 30 mph at l_1 and 40 mph at l_3. Remember that since you always travel at a *constant* speed during each part of your route you the time t_i to travel part i is l_i/v_i if you travel according to the speed limit or $l_i/(v_i + v)$ if you decide to use the extra speed v. Since you want to optimize the time of your commute, you want to find at which parts of the route you should break the speed limit and use the extra speed v. You have many ideas on how you should use the extra speed.

   (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

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

   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.

Challenge problem

6. Suppose you are given a connected graph G in which the edge weights depend on a parameter t, where t ranges over the reals. More specifically, the cost of an edge e is given by a function of the form $c_e(t) = a_e t^2 + b_e t + d_e$, where a_e, b_e, d_e are reals depending on e, and $a_e > 0$. Your goal is to find the minimum cost of a minimum spanning tree of G over all values of t. Your algorithm should run in time polynomial in the number of nodes and edges of G. You may assume that you can perform standard arithmetic operations (including computing square roots) at unit cost.

   Would your approach also work for the shortest paths problem (assuming the weight functions are nonnegative over their entire range)?

Programming problem

7. SPOJ problem Island Hopping (problem code ISLHOP)

CS 577: Introduction to Algorithms Fall 2022

Homework 6 Solutions to Warm-up Problems

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

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

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.

CS 577: Introduction to Algorithms Fall 2022

Homework 6 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 3

You're about to buy a new phone and want to determine the minimum amount of memory that is required to download and install your n favorite apps. The i-th app has a download size of d_i and a storage size of s_i. To download the app, your phone must have at least d_i megabytes of free disk space. After installation the app consumes s_i megabytes of disk space on the phone. The download size d_i is always at least as large as the storage size s_i but may be larger due to material that might not get used such as translations to different languages.

Design on $O(n \log n)$ algorithm to determine the amount of memory needed to download and install all n apps in an optimal order.

Let us first consider the case with $n = 2$ apps. The amount of memory space needed to first download app 1 and then app 2 is: d_1 for downloading app 1, and $s_1 + d_2$ for downloading app 2 while app 1 is in memory, so the amount of memory needed for the entire process is $\max(d_1, s_1 + d_2)$. The other order similarly requires space $\max(d_2, s_2 + d_1)$. The first order is optimal if and only if

$$\max(d_1, s_1 + d_2) \leq \max(d_2, s_2 + d_1), \tag{1}$$

i.e., the largest value among d_1, $s_1 + d_2$, d_2, $s_2 + d_1$ is among d_2 and $s_2 + d_1$. Since $s_2 + d_1 \geq d_1$ and $s_1 + d_2 \geq d_2$, the largest value among those four is always one of $s_1 + d_2$ and $s_2 + d_1$. If follows that (1) is equivalent to $s_1 + d_2 \leq s_2 + d_1$, which can be rearranged as

$$d_2 - s_2 \leq d_1 - s_1. \tag{2}$$

If the inequality in (2) goes the other way around, then first downloading app 2 and then app 1 is optimal. In case of equality, both orders are optimal. This suggests the greedy strategy of downloading the apps in the order of non-increasing value of $d_i - s_i$, with ties broken arbitrarily.

The criterion (2) can also be gleaned from Figure 1, where the top represents the situation where app 1 is downloaded first and then app 2, and the bottom the other order. The solid area represents the permanent memory needed for a given app, s_i, and the hatched area the additional memory needed for downloading the app, $d_i - s_i$. The amount of memory needed for a download order corresponds to the furthest a block in that order sticks out to the right.

Note that is possible in the top line for the hatched block of app 1 to stick out the furthest, but in that case the corresponding hatched block in the other order sticks out even further. Thus, we only need to compare how far the blocks corresponding to the second downloads in both orders stick out. (This is the equivalent of the above observation that two of four quantities involved always contain the maximum.) Since the memory that is permanently needed for both orders is the same, the starting point for the hatched blocks in both orders is the same, namely $s_1 + s_2 = s_2 + s_1$. Thus, the order that sticks out the furthest is the one for which the hatched block of the second download is the largest. The other order is the preferred one.

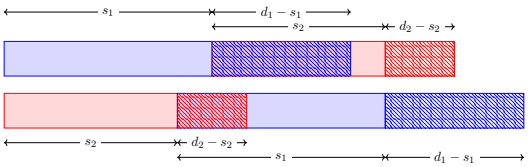

Figure 1: Geometric comparison of download orders for two apps.

Correctness We use an exchange argument based on the above two-app analysis. Consider any order π of the n apps that is not our greedy order. Such an order has to contain two consecutive apps, say app 1 and app 2, that are out of our greedy order: App 2 is downloaded before app 1 while $d_2 - s_2 \leq d_1 - s_1$. Consider the order π' obtained from π by swapping apps 1 and 2. Doing so:

- Does not affect the space needed for downloading the apps that come before both app 1 and app 2 as there is no difference between π and π' up to that point.
- Changes the memory space required to download apps 1 and 2 from $\max(s+d_2, s+s_2+d_1) = s + \max(d_2, s_2+d_1)$ in π to $\max(s+d_1, s+s_1+d_2) = s + \max(d_1, s_1+d_2)$ in π', where s denotes the sum of the storage sizes of all apps that come before both app 1 and app s.
- Does not affect the space needed for downloading the apps that come after both app 1 and app 2 as the total storage size of all prior apps in π and π' is the same for each such app.

Thus, the swap does not increase the memory space required as long as $s + \max(d_1, s_1 + d_2) \leq s + \max(d_2, s_2 + d_1)$, which simplifies to $\max(d_1, s_1 + d_2) \leq \max(d_2, s_2 + d_1)$, and by the above analysis is equivalent to our hypothesis $d_2 - s_2 \leq d_1 - s_1$.

The above swap reduces the number of inversions of π with respect to the greedy order by 1. Thus, after a finite number of swaps we end up at our greedy order without every having increased the memory space needed. As we started with an arbitrary order π other than our greedy order, this implies that our greedy order requires the least amount of memory space.

Algorithm and running time Implementing the above strategy involves sorting the apps in nonincreasing order of $d_i - s_i$, and then computing the memory space required for that order. The first phase can be done in time $O(n \log n)$. The second phase can be done in time $O(n)$ by going over the apps in the sorted order, keeping track of the sum s of the storage sizes of the apps downloaded thus far, and keeping track of the maximum value of $s + d_i$. The finale value of the latter quantity is the answer. See Algorithm 1 below for pseudocode. The resulting overall running time is $O(n \log n) + O(n) = O(n \log n)$.

Alternate solution The problem can be solved by a reduction to minimizing the maximum lateness. For each app i, create a job i with duration s_i and deadline $D_i \doteq s_i - d_i$ (note that

Algorithm 1

```
Sort the apps in nonincreasing order of d_i − s_i
m ← 0; s ← 0
for i = 1 to n do
    m ← max(m, s + d_i); s ← s + s_i
return m
```

$D_i \leq 0$, which is OK). Suppose that the jobs are indexed the way they are scheduled, starting with $i = 1$. The finish time of job i equals $T_i \doteq \sum_{j=1}^{i} s_j$, so the lateness of job i equals $\max(0, T_i - D_i) = d_i + \sum_{j=1}^{i-1} s_j$, which is exactly the memory space needed for downloading app i in this order. The maximum lateness over all jobs equals the amount of memory space needed for downloading all apps, which is the quantity we want to minimize. Thus, a schedule that minimizes the maximum lateness is equivalent to a download order that minimizes the memory required. This argues the correctness of the reduction. As the reduction takes time $O(n)$ and the resulting instance of minimizing maximum lateness can be solved in time $O(n \log n)$ using the algorithm from class, the overall running time of this approach is also $O(n \log n)$. In fact, this approach ends up doing exactly the same elementary operations as the direct approach above.

Problem 4

In a city there are n bus drivers. There are also n morning bus routes and n evening 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 2 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. □

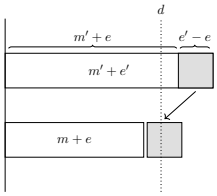

Figure 2: Effect on total overtime of swap when $m + e < d < m' + e'$.

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 5

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

$S \subseteq [n] \doteq \{1, \dots, n\}$. 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 $O(n \log n)$.

COMP SCI 577 Homework 06 Problem 3

Greed

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

1 November 2022

Algorithm

Explanation

We first sort the apps $[d_i, s_i]$ descendingly by the value of $d_i - s_i$. Then we loop through the sorted apps to calculate the total space needed in this order: if there is not enough space (free space = total space − consumed space) to download the current app, expand the space first; then after we have enough space, we increase the consumed space after installation. By downloading and installing the apps in this order, we get the smallest total space needed.

Code (Python) of $O(n \log n)$

```python
import input
import typing

if __name__ == "__main__":
    n: int = input.nextInt()
    d: typing.List[int] = [input.nextInt() for _ in range(n)]
    s: typing.List[int] = [input.nextInt() for _ in range(n)]
    apps: typing.List[typing.Tuple[int, int]] = [(d[i], s[i]) for
       i in range(n)]
    apps.sort(key=lambda x: x[0] - x[1], reverse=True)
    d = [app[0] for app in apps]
    s = [app[1] for app in apps]
```



```python
    total: int = 0
    consumed: int = 0
    for i in range(n):
        if total - consumed < d[i]:
            total += (d[i] - (total - consumed))
        consumed += s[i]
    print(total)
```

Proof

Correctness

Let an ordering of apps be $A = (D, S)$ with $a_i = (d_i, s_i)$, D be downloading size, and S be installing size. Let A be the order to download and install apps by our algorithm, and A^* be any order which could minimize the total space used. If $A = A^*$ then we are done. Otherwise, $A \neq A^*$, i.e., $\exists i \in [n]$ such that $a_i \neq a_i^*$. Denote updated spaces at the i^{th} round as S_t^i and S_c^i. The base case $n = 0$ is always true, as when there is no app installed, there is no space that should be cost.

First, we can prove that we have correct total and consumed costs by the counting part, as the total space S_t could be increased to fit any d_i, and the consumed space S_c always has $S_c \leq S_t$. We start an induction for the loop invariants: (1) $S_t^i \geq S_c^i$, (2) $S_t^i - S_c^{i-1} = d_i$, and (3) $S_c^i - S_c^{i-1} = s_i$. We have the inductive step: by $S_t^i = S_t^{i-1} + d_i - (S_c^{i-1} - S_c^{i-1}) = d_i + S_c^{i-1}$, $S_c^i = S_c^{i-1} + s_i$, and $d_i \geq s_i \geq 0$; we have $S_t^i - S_c^i = d_i - (S_c^i - S_c^{i-1}) \geq 0$ by loop invariant (3), $S_t^i - S_c^{i-1} = d_i + S_c^{i-1} - S_c^{i-1} = d_i$, and $S_c^i - S_c^{i-1} = S_c^{i-1} + s_i - S_c^{i-1} = s_i$.

Then, noting that any A (including which redefined later), A^*, and G are feasible orders with arrangements as (d_i, s_i) are bundled together. We redefine A be a set transformed from A^* by that there are two consecutive elements in A^* denoted as a_i^* and a_{i+1}^* in a different order than they are in G, i.e., there is an inversion such that $a_i = a_{i+1}^*$ and $a_{i+1} = a_i^*$. Denote the comparison between two elements a_i and a_j in A be based on their differences, i.e., $d_i - s_i$ and $d_j - s_j$, as well as similarly in A^*. Then we have $a_i \geq a_{i+1}$ from our algorithm such that $a_{i+1}^* \geq a_i^*$.

As A^* is an optimal solution which is in a different order, to keep the optimality (or equality) of the space yielded by our algorithm, we should have $s_i + d_{i+1} \leq s_i^* + d_{i+1}^* = s_{i+1} + d_i$ for the greedy-like order which could be proved by $d_{i+1} - s_{i+1} \leq d_i - s_i$, i.e., $a_i^* \leq a_{i+1}^*$, with $s_i + d_{i+1}$ (similar for $*$) be the minimum space occupancy of the concatenation by appending a_{i+1} after a_i

if there is no enough space for d_{i+1} after installing s_i. Another situation is that $d_i - s_i \geq d_{i+1}$, so that a_i^* is fit within the remaining space from a_{i+1}^*, and the total space consumption is d_i, then after transformation we have $s_i + d_{i+1} \leq d_i$ which also satisfies $s_i + d_{i+1} \leq s_{i+1} + d_i$ mentioned before. We use the signs $\leq$ and $\geq$ to ensure that tie are maintained that should not be . Thus, we have proven in both cases, the space consumed does not increase from A^* to A.

As the number of inversions of A with respect to the true greedy solution G is one less than the number of inversions of A^* with respect to G. We end up in G eventually. □

Termination

The algorithm must terminate, as it is in loops without any recursion. Also, no edge case should be taken care of.

Complexity

Data reading costs $O(n)$ time, sorting costs $O(n \log n)$ time, and counting costs $O(n)$ time. The overall time cost is $O(n \log n)$.

Test Cases

Input: $n \leftarrow 2$; $d \leftarrow [7, 10]$; $s \leftarrow [4, 6]$;
Output: 13;
Explanation: Same as the example in the write-out of this problem.

Appendix

Code (Python) for utility `input`

```python
from typing import List, Optional

file = None
queue = []

def openFile(filename: Optional[str]):
    global file
    if filename ≠ None:
        file = open(filename, "r")
    else:
        file = None

def next() → Optional[str]:
    while len(queue) == 0:
        if file == None:
            line: str = input()
        else:
            line: str = file.readline()
        if len(line) == 0:
            return None
        lineArr: List[str] = line.split(" ")
        for lineElem in lineArr:
            queue.append(lineElem)
    return queue.pop(0)

def nextInt() → int:
    result = next()
    if result ≠ None:
        return int(result)
    raise Exception("no input")
```