Problem Set 2

All parts are due Thursday, October 13 at 11:59PM.

Name: Dimitris Koutentakis

Collaborators: Carl Unger

Part A

Problem 2-1.

(a) $T(n) = \Theta(n^2 logn)$ is of the form $\Theta(n^k logn)$, which implies that this refers to the second case of the Master Theorem.
Thus:

$$\Theta(n^2) = \Theta(n^{log_2 a}) \iff a = 4$$

(b) $T(n) = \Theta(n^2)$ and $f(n) = \Theta(n)$ imply that this refers to the first case of the Master Theorem. Thus:

$$n^{log_b a} = n^2, b = 3 \iff a = 9$$

The master theorem holds for $\epsilon = 1$.

(c) $f(n) = \Theta(n^2)$ and $T(n) = \Theta(n^2)$ Imply that this recurrence is of the type of the third case of the Master Theorem. Thus, for $a = 4$, we have:

$$f(n) = \Omega(n^{log_b 4+\epsilon)}) \iff n^{log_b 4} < n^2 \iff log_b 4 < 2 \iff b > 2$$

(d) $f(n) = \Theta(n^5)$ and $T(n) = \Theta(n^2)$ imply that we are in the first case of the Master Theorem. Thus, we have:

$$n^6.06 = n^{log_b 5} \iff 6.006 = log_b 5 \iff b = 5^{\frac{1}{6.006}} \iff b = 5^{\frac{500}{3003}} = 1.3073$$

(e) For a$a = 6$ and $b = 6$, the only possibility to have $T(n) = \Theta(n^2)$ is to have the third case of the Master Theorem. In that case,

$$f(n) = T(n) = \Theta(n^2)$$

Problem 2-2.

(a) The algorithm for this process, would be a divide and conquer algorithm. An easy way to solve this in time, would be to always divide the array in half along the smaller direction (m). After that, we would have two resulting arrays of n rows and $\frac{m}{2}$ columns each.

We would then recurse on each of these two arrays until we only get column vectors of length n. Since we already have sorted 1-D arrays, the easier way to merge them in a sorted way would be to use merge sort. Namely, we would have to compare each element of two arrays at a time and add the smallest element to the sorted array of length 2n. We would repeat that process until we result in a 1-D sorted array of length mn.

For the runtime analysis of this algorithm, we can use the recursion tree method. The tree would have two branches from every node and the time will be evenly divided along the branches, always resulting in time m at every level. The height of the tree would be logm. Since merge sort would take linear time for pre-sorted arrays, the merging would be done in linear time O(mn). Thus the total runtime of this algorithm would be:

$$O(nmlog(m))$$

Problem 2-3.

(a) If the trucks we want to switch are within distance k, we can just switch them. If not, then we will have to follow a sequence of steps, according to the following algorithm:

Let T1 and T2 be the two trucks we want to swap at positions x and y. Assume y>x.
If y is not a multiple of k plus x, that is if (y-x) mod k != 0, then swap T1 with the truck at position x+(y-x) mod k = r.
Now keep swapping truck T1 (at its new place) with the one at k spots further than T1, until it is swapped with T2. That would be $\frac{\lfloor y-x \rfloor}{k}$ swaps. The result would be T1 at T2's place and T2 would be at position y-k.
In order to return the rest of the trucks to the original order, we just have to do the opposite swap $\frac{\lfloor y-x \rfloor}{k} - 1$ times. If r was nonzero, we have to end by switching T2 with the truck at position (T2-r).

The algorithm above will switch any two trucks without affecting the order of the trucks inbetween in time O(n/k).

(b) In order to compare the sizes of two trucks, they must be less than distance k away. In order to achieve that, we will use part of the algorithm described above.

That is we will perform $\frac{\lfloor y-x \rfloor}{k} - 1$ swaps after the first one, so that both trucks end up exactly k trucks afar. After performing the comparison, we have to return the truck back, by repeating the same exact switches in the opposite order.

This algorithm will compare the size of two trucks (not necessarily at distance at most k) in O(n/k), while not affecting the position of any truck.

(c) In order to sort the trucks by size, we will have to use our answer to part (b) to compare the sizes of the trucks and the answer to part (a) to move our trucks into position. The way to sort the trucks in place and quickly, would be using heap-sort.

In order to use heapsort, we will have to first order our trucks into a heap. To do that, we would essentially run max-heapify on the sizes of the trucks. That would result in a max-heap, where each parent is bigger than both of its children. This comparison will be done by our answer to part (a) We would then represent that heap as a list, instead of a tree. That list will be the positions of the trucks (moved by our answer to b) before the sorting begins.
Now, for the actual sorting, at each instance we will move the root, which is the largest truck, and located at the left-most truck in the parking lot to the end, re-heapifying between every iteration. So in every iteration, we remove the root (largest unsorted truck, located at the smallest indexed spot), switch it with the last unsorted truck (n-removals), then heapifying the heap again to maintain the heap property, and repeating. This will end up sorting all the trucks by size, by largest to smallest.

The algorithm described above will sort all trucks by smallest to largest in $O((n^2 logn)/k)$ truck swaps. In specific, heap sort takes $O(nlogn)$, broken down, that is $O(n)$ for building the heap, and $O(lgn)$ for heapifying in every iteration. In addition, each switching of trucks takes $O(n/k)$ truck swaps, as does every comparison for building the heap. Thus the final order of truck swaps would be:

$$T(n) = O(\frac{n^2 logn}{k})$$

(d) Part d

Part B

Problem 2-4.

(a) An algorithm of time $O(N^2)$ to find Bowser's rank is easily implemented as following:

First we have to create the behind and ahead dictionaries we can do that in $O(n^2)$ time, just by going through the contestants, and for every contestant find the one exactly ahead and update the ahead and behind dictionaries. Once those are updated, we can calculate the losetime for every pair (ahead, behind) and insert that into a list of tuples including the losetime and the pair.
Now for the recursion, we just create a loop in which in every iteration the first loser is removed (we should also update the ahead and behind dictionaries and the losetime list). This loop will end once the loser is Bowser. The number of iterations the loop has run will be the number of people who lost before Bowser, and thus it will also be his rank.

Therefore, the above algorithm describes how to find the rank of Bowser in $O(N^2)$ time.

(b) *On alg.csail.mit.edu*

(c) *On alg.csail.mit.edu*

(d) Once Charlie passes Bowser, Bowser is eliminated. Hence we should remove from consideration any future event that involves Bowser passing anyone. We should also not consider anyone else passing Bowser. The event that would now be added to our consideration, is that of Charlie passing Alice only if $v_c > v_a$. Since Alice now is the one directly ahead of Charlie, she might get passed if $v_c > v_a$. The consideration for events involving Deborah remain the same.

(e) *On alg.csail.mit.edu*