Problem Set 2

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

Name: Milo H. Knowles

Collaborators: Gillian P. Belton

Part A

Problem 2-1.

(a) $T(n) = \theta(n^2 * \log n)$ is a solution to $T(n) = aT(\frac{n}{2}) + \theta(n^2)$.
Consider $n^{\log_2 a}$ vs. n^2
Compare the exponents: $\log_2 a$ and 2 must be equal.
$\implies a = 4$

(b) $T(n) = \theta(n^2)$ is a solution to $T(n) = aT(\frac{n}{3}) + \theta(n)$.
Consider $n^{\log_3 a}$ vs. n^1
Because the runtime is $\theta(n^2)$, $n^{\log_3 a}$ must be the dominant term, and equal to n^2
$n^{\log_3 a} = 2$
$\implies a = 9$

(c) $T(n) = \theta(n^2)$ is a solution to $T(n) = 4T(\frac{n}{b}) + \theta(n^2)$.
Consider $n^{\log_b 4}$ vs. n^2
Because the runtime is $\theta(n^2)$, the $\theta(n^2)$ term must be dominant.
This will occur when $\log_b 4 < 2$
$b^{\log_b 4} < b^2$
$b^2 > 4$
$|b| > 2$. But we are only considering positive real b, so $b > 2$.

(d) $T(n) = \theta(n^{6.006})$ is a solution to $T(n) = 5T(\frac{n}{b}) + \theta(n^5)$.
Consider $n^{\log_b 5}$ vs. n^5. The left side must be dominant in order for .$T(n) = \theta(n^{6.006})$
$\implies \log_b 5 = 6.006$
$b^{6.006} = 5$
$\log b = \frac{\log 5}{6.006}$
$b = 1.3073$

$T(n) = \theta(n^2)$ is a solution to $T(n) = 6T(\frac{n}{6}) + f(n)$.

(e) Consider $n^{\log_6 6}$ vs. $f(n)$
Equivalent to: $nvs.f(n)$
In order for the runtime to be $\theta(n^2)$, $f(n)$ must be asymptotically similar to n^2.
$\implies f(n) = n^2$ is one possible $f(n)$.

Problem 2-2.

(a) Sorting a Rectangle

$$\begin{bmatrix}1\\2\\3\\4\\4\\5\\5\\6\\8\\10\\19\\30\end{bmatrix}$$

$$\begin{bmatrix}1\\2\\3\\4\\5\\6\end{bmatrix} \qquad \begin{bmatrix}4\\5\\8\\10\\19\\30\end{bmatrix}$$

$$\begin{bmatrix}1\\3\\4\end{bmatrix} \quad \begin{bmatrix}2\\5\\6\end{bmatrix} \quad \begin{bmatrix}4\\5\\19\end{bmatrix} \quad \begin{bmatrix}8\\10\\30\end{bmatrix}$$

This algorithm is essentially a modified version of merge sort. The array A can be split up into individual columns, each of which is a leaf in the binary tree above.

The "merging" subroutine: To merge two length n columns into a single sorted array, keep taking the largest element from either column until the merged array is complete (has length $2n$). This takes $\theta(n)$ time, because there are n elements in each column.

At each layer of the tree, there are $\frac{m}{2^l}$ arrays to merge, where l is the layer in the tree. However, the size of each array is $n \cdot 2^l$. So each layer will take the same amount of

time to merge: $\theta(mn)$.

The height of the tree is $\log_2 m$, and each layer takes $\theta(mn)$ time, so the runtime will be $\theta(mn \cdot \log_2 m)$.

Problem 2-3.

(a) Algorithm to swap two arbitrary trucks and return all the trucks that were moved in the process to their original positions:

1. Given an arbitrary pair of trucks A and B, their distance from each other is, at most,$n-1$ spaces. Let Truck A have lower index and Truck B have higher index. While the distance between the two trucks (in spaces) is $\geq k$, swap Truck A with the truck k spaces ahead. This will take, at most, $\frac{n}{k} - 1$ swaps.
2. Once the Truck A is within k spaces of the Truck B, swap them with each other. Now, Truck A is in the correct position and there have been, at most, $\frac{n}{k}$ swaps.
3. Excluding the final swap, make every swap again in reverse chronological order. This will leave Truck A in place, put Truck B in the original space of Truck A, and return every other truck to its original space. This will require, at most, $\frac{n}{k} - 1$ swaps, because we exclude the final swap.

At this point, there have been at most, $\frac{2n}{k} - 1$ swaps. This means that the algorithm has taken $O(\frac{n}{k})$ time.

(b) Algorithm to compare two arbitrary trucks, and return all trucks that were moved in the process to their original positions:

1. Given an arbitrary pair of trucks A and B, their distance from each other is, at most,$n-1$ spaces. Let Truck A have lower index and Truck B have higher index. While the distance between the two trucks (in spaces) is $> k$, swap Truck A with the truck k spaces ahead. This will take, at most, $\frac{n}{k} - 1$ swaps.
2. Once the Truck A is within k spaces of the Truck B, the two trucks can be compared.
3. Make every swap again in reverse chronological order. This will return every truck to its original space. This will require, at most, $\frac{n}{k} - 1$ swaps, because this is the maximum number of swaps we made in step 1.

At this point, there have been at most, $\frac{2n}{k} - 2$ swaps. This means that the algorithm has taken $O(\frac{n}{k})$ time.

(c) To sort the trucks in $O(\frac{n^2 \log n}{k})$ time, we can use a modified version of heapsort. Building the heap takes $O(\frac{n^2}{k}$ time, because heapify will do, at most, n comparison-based swaps, each of which take $O(\frac{n}{k}$ time. Although comparisons and swaps each take

$O(\frac{n}{k})$ time, they can be easily combined into a single $O(\frac{n}{k})$ process that compares two trucks and then swaps them before restoring other trucks to their correct positions.

Heapsort runs the extract_min subroutine n times on the heap, taking the minimum element in constant time and running min_heapify after every extraction. The subroutine min_heapify runs in $O(\frac{n \log n}{k})$, because at most $\log n$ violations of the MHP must be fixed in the heap, and each fix requires an $O(\frac{n}{k})$ comparison-swap, as we saw previously. Therefore, the runtime of heapsort is $O(\frac{n^2 \log n}{k})$.

(d) Extra credit.

Part B

Problem 2-4.

(a) The following is an $O(n^2)$ algorithm that computes Bowser's final rank. Consider the pseudocode below:

```
losetimes = { } # this dict stores players as keys, and each player's "lose-time" as a value

for s in competitors:
    for f in competitors:
        min_losetime = 1,000,000,000 # initialized with a really high integer
        if s == f:
          continue
        else:
          if vel_f > vel_s:
            time = calculate_losetime(f, s)
            if time < min_losetime:
              min_losetime = time
    losetimes[s] = min_losetime
```

With the list of minimum losetimes for each player, we can build a list of race events by extracting the event with the smallest losetime, and removing all of the possible events where the player who was just taken out of the race overtakes another racer. We must extract the smallest losetime at most n times, and each extraction costs $O(n)$ time. Therefore, this step takes $O(n^2)$ time.

Finally, we iterate over the list of events (of which there are n, to find the position in the events list that Bowser is in. His rank will be his index in the list plus one, due to zero indexing.

The total cost of this algorithm is $O(n^2)$, because building the losetimes dictionary takes $O(n^2)$ time, and getting a sorted race events list takes $O(n^2)$ time.

This algorithm will take $O(n^2)$ time because it takes $O(n)$ time to compute the minimum lose-time for each competitor, and there are n competitors.

(b) *Submit your implementation on alg.csail.mit.edu*

(c) *Submit your implementation on alg.csail.mit.edu*

(d) We assume that Charlie does pass Bowser. In this case, we must add 1 event and remove 1 event:

- Because Bowser is eliminated from the race, the event of Bowser passing Alice is no longer possible and must be removed. If we are considering all possible passing events, not just players passing the player directly in front of them, then we must remove all events in which Bowser passes a competitor.
- Once Bowser is gone from the race, Alice will be now be the player in front of Charlie. If $v_c > v_a$, then we must add the event of Charlie passing Alice. If we are considering all possible passing events, not just players passing the player directly in front of them, then we must remove all events in which Bowser is passed by a competitor.

(e) *Submit your implementation on alg.csail.mit.edu*