CS4311 Design and Analysis of Algorithms

Homework 2 (Suggested Solution)

1. (a) **Ans.** Let ℓ be the length of the input, and $m = \lfloor \ell/3 \rfloor$. The correctness can be shown by induction on m.

   (Base Case:) When $m = 0$, ℓ is either 0, 1, or 2. It is obvious that the algorithm is correct in all cases.

   (Inductive Case:) Suppose that the algorithm is correct for all ℓ such that $m = 0, 1, 2, \ldots, k-1$. In other words, the algorithm is correct for all $\ell = 0, 1, 2, \ldots, 3k-1$. When $m = k$, ℓ is either $3k$, $3k+1$, or $3k+2$. In all cases, the minimum ℓ % 3 numbers will be placed correctly before the three recursive calls. It remains to show that after the three recursive calls, the remaining $3k$ numbers will be sorted correctly.

   First, the induction hypothesis guarantees that each of the three recursive calls will perform correctly (because each is called with length less that $3k$). Thus, after the first recursive call, the largest m numbers *cannot* be in the leftmost m entries, so that the largest m numbers must then be within the rightmost $2m$ entries. This implies that after the second recursive call, the m largest numbers must be placed in the rightmost m entries and sorted. Finally, the third recursive call guarantees the remaining $2m$ smaller numbers are sorted. So, the algorithm performs correctly.

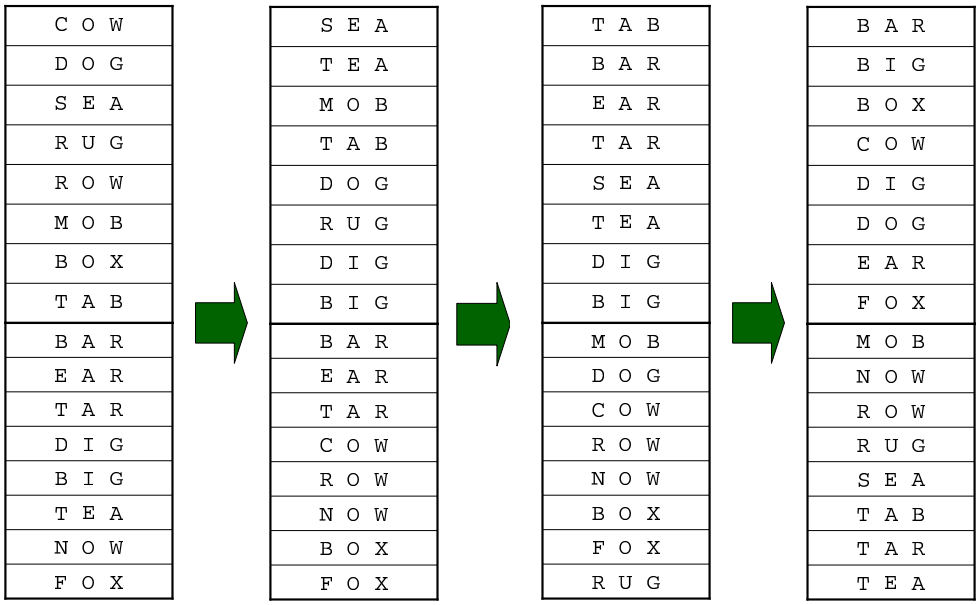
   (b) **Ans.** $T(n) = 3\,T(2n/3) + O(n)$

   (c) **Ans.** By Master Theorem,

   $$T(n) = \Theta(n^{\log_{1.5} 3}) \approx \Theta(n^{2.7095}) = \omega(n^2).$$

   Thus, in terms of the asymptotic running time, `JohnSort` is worse than both insertion sort and merge sort.

2. **Ans.**



3. **Ans.** Let x and y be two integers, and let ℓ_x and ℓ_y denote the lengths of the strings representing x and y, respectively. A key observation is that: if the length ℓ_x is shorter than the length ℓ_y, then the corresponding integer x must be smaller than the corresponding integer y.

   This observation gives us a way to sort our input numbers, using the "bucketing by length" scheme, as follows:

   (Step 1:) Create n buckets, where Bucket i is used for holding integers whose length is exactly i.

   (Step 2:) Distribute the k input integers to the corresponding bucket.

   (Step 3:) Sort each bucket separately using Radix Sort.

   (Step 4:) Collect the sorted list of integers in each bucket, from Bucket 1, Bucket 2, ..., up to Bucket n.

   The correctness of the above algorithm follows from the key observation and from the correctness of Radix Sort. For the running time, it takes $O(n + k)$ time to create the buckets and distribute the numbers into the correct bucket (Steps 1 and 2). After that, the time spent in Radix Sort (Step 3) is proportional to the total number of digits in the k numbers, so the time is $O(n)$. Finally, the collection phase in Step 4 takes $O(n + k)$ time. Since k is smaller than n, the total running time is $O(n)$.

4. (a) **Ans.** There are many possible answers. Some examples are shown below:

2	3	4	∞
5	8	12	∞
9	14	∞	∞
16	∞	∞	∞

2	3	4	5
8	9	12	14
16	∞	∞	∞
∞	∞	∞	∞

2	5	8	∞
3	12	∞	∞
4	14	∞	∞
9	16	∞	∞

   (b) **Ans.** The Young tableau property guarantees that $Y[i, j] \leq Y[x, y]$ if $x \geq i$ and $y \geq j$. It is because:

$$Y[i, j] \leq Y[i + 1, j] \leq \cdots \leq Y[x, j], \quad \text{and}$$

$$Y[x, j] \leq Y[x, j + 1] \leq \cdots \leq Y[x, y].$$

   This implies that any entry $Y[x, y]$ must be at least $Y[1, 1]$. So, if $Y[1, 1] = \infty$, the table cannot contain a finite value, and must therefore be empty. Similarly, any entry $Y[x, y]$ must be at most $Y[m, n]$. So, if $Y[m, n] < \infty$, the table cannot contain ∞ and must therefore be full.

   (c) **Ans.** To perform `Extract-Min`, we first replace the entry $Y[1, 1]$ by ∞. This step guarantees that the table will be storing exactly the same set of values as the desired table. Next, we will restore the ordering of the entries so that each row and each column become sorted.

   The restoration step is very similar to the `Extract-Min` in a heap. We say an entry e satisfy the "sorted-property" if its right adjacent entry and its bottom adjacent entry both have values greater than the value of e. We claim that after each swap, at most one entry may violate the "sorted-ordering" property.

   Then, after each swap, while there is a violating entry x, we will compare the values in the adjacent entries to the right and to the bottom (if exist) of x. Let y be the

entry whose value is smaller. We then swap the value of y with the value of x. After the swapping, y becomes the single entry which may violate the "sorted-ordering" property.

Suppose that our claim is true. Then, there can be at most $m + n - 1$ swaps in the restoration step, because each swap must move the violating entry one step right or one step bottom. Once no entry violate the "sorted-ordering" property, each row and each column in the tableau must be sorted. Thus, if our claim is true, the time for `Extract-Min` in the tableau is $O(m + n)$.

It remains to prove the claim. To show that, we shall use induction to show that after each swap, the following two statements are simultaneously correct:

(i) At most one entry may violate the "sorted-ordering" property;

(ii) if x is the violating entry, let t, ℓ, r, b denote the adjacent entries (if exist) to the top, left, right, and bottom of x. Then, both the values of t and ℓ are greater than both the values of r and b.

(Base Case:) It is easy to check that after the first swap, both statements are true.

(Inductive Case:) Suppose the two statements are true after the kth swap. During the $(k + 1)$th swap, x either swaps with r or b, whichever contains the smaller value. Let us call the swapped entry y. After this swapping, the value in x will satisfy the "sorted-ordering" property *because it now contains the smaller of the original r or b.* Also, u and ℓ satisfy the "sorted-ordering" property *because of induction hypothesis that x now contains a value larger than both u and ℓ.* All remaining entries (except y) must follow the "sorted-ordering" property because values in their right and bottom neighbors are not changed. Thus, y becomes the only entry which may violate the "sorted-ordering" property, and both entries to the top and left of y have values smaller than both entries to the right and bottom of y, *due to induction hypothesis that the original y satisfy the "sorted-ordering" property after the kth swap.*

(d) **Ans.** We start with an empty tableau. Then, we use n^2 `Insert` to insert the input numbers into the tableau. After that, we call `Extract-Min` n^2 times to obtain the sorted sequence of the input numbers. Since each `Insert` or `Extract-Min` take $O(n)$ time, the total time is $O(n^3)$.

5. **Ans.** Given a $m \times n$ Young tableau, for any number K, we define the boundary of K in the ith row, denoted $Bound_i(K)$, the the position of the rightmost entry whose value is at most K. Then, to check if K exists in the tableau, it is sufficient to check for each row i, whether the values of the $Bound_i(K)$th entry is K.

   A key observation is that: for any row, $Bound_i(K) \geq Bound_{i+1}(K)$. In other words, the boundaries of K is monotonically shifting towards the left side if we proceed from the top row to the bottom row. This observation immediately gives us an efficiently way to find *all* the boundaries of K: Start scanning the first row, from rightmost entry towards left, and find $Bound_1(K)$. Then, iteratively, once $Bound_i(K)$ is obtained, we scan the $(i+1)$th row from the $Bound_i(K)$th entry towards left, and find $Bound_{i+1}(K)$. Using this procedure, all the boundaries of K are found in $O(m+n)$ time (because by considering the first move in each row as a 'vertical' move, we have made a total of n left moves, and a total of m vertical moves). Once the boundaries are obtained, we can decide if K is in the tableau in $O(m)$ time.

   In conclusion, the total time taken is $O(m + n)$.