CSC 225 Assignment 2,
Julian Rocha, V00870460
A01, B02

1) Insertion sort is composed of 2 loops (one nested in the other). The first iterates through every element of an array. The second will compare the current element with an ever-growing sorted sequence and insert it in the correct position. Two important things to observe is the outer loop will run n times for an array of size n, and **every iteration of the inner loop (comparing and switching current item with an item from sorted sequence) will eliminate a single inversion.** So, if we know the total number of inversion beforehand, k, then we know the inner loop will run a total of k times. **Therefore the runtime will be O(n + k)**. This makes sense to us because if there are no inversions (already sorted) then insertion sort must simply pass through the array to insure this. This best-case scenario is O(n) since k = 0. In a reverse sorted order array, we get a shrinking sum of inversions for each element, namely: $k = \sum_{i=0}^{n} i = n(n+1)/2$ In this worst-case scenario, we get $O(n + n(n+1)/2)$ which is $O(n^2)$.

2) **We can recursively define the number of inversions in any array to be the number of inversions in it's left half + number of inversions in its right half + the number of inversions found when merging the two.** An O(n log n) algorithm for counting array inversions could then be created by using a modified merge sort. First the array is recursively divided until we get n arrays of size 1. Then the subarrays are recursively merged (to form an ascending order sort), only we **keep a counter of how many times an element from a left subarray was greater than an element from a right subarray** when comparing the left and right sides. This modification of merge sort will add a counter to the algorithm but the runtime is still O(n log n).

3) First, pass through Ranking 1, and create a new array containing the index locations of every element. The array will be called the comparison array.

   For example:

   Ranking 1

Index	0	1	2	3	4	5	6
Value	0	3	1	6	2	5	4

   Becomes…

   Comparison Array

Index	0	1	2	3	4	5	6
Value	0	2	4	1	6	5	3

   This will take O(n) time. This array can now be used as a means of comparing elements from Ranking 2. Now we use the same modified merge sort algorithm as Question 2, only now we compare two elements using the Comparison Array. For example:

   - When comparing elements 4 and 6 from Ranking 2
   - We compare CompArray[4] vs CompArray[6]
   - = 6 vs 3
   - Therefore, we know 6 comes before 4

Eventually, Ranking 2 will be ordered in the same way as Ranking 1, and we will have a count of how many inversions it took to get there.

This new way of comparing will add lines but since no loops were added other than the O(n) Comparison Array creation at the start, the total runtime is unchanged.

The total inversions are equal to the Kendall tau distance.

4) Here since we know the range of integers, we can use **radix sort** to very efficiently sort the array in O(n) time. We first sort by least significant digit and end with the most significant digit. Since the number of passes through the n numbers is equal to the number of digits k in the largest element, we know that this radix sort will have a run time of O(n * k).
   Since $n^2 - 1$ is largest number:
   - $k = \log_{10}(n^2 - 1)$
   - $k = \log_{10}(n^2)$ *approx. for large n
   - $k = 2\log_{10}(n)$

So the sort will be $O(n * 2\log_{10}(n))$

In general however, we can say the runtime of radix sort is

O(k *(n + b)) where k is # digits in largest number, n is number of elements, and b is the base that the integers are represented in.

****From theorem in class****

From our previous work, we can simplify this to:

- $O(2\log_b(n) * (n + b))$
- $O(\log_n(n) * (n + n))$ * remove constants and setting base b to n
- $O(1 * 2n)$
- **= O(n)**

5) In this scenario, the act of inserting n items into a MinPQ would take O(n log log n) time. The act of now sequentially removing all n items from the MinPQ and adding them to array A would again take O(n log log n) time. The entire process would take O(n log log n) + O(n log log n) which is simply O(n log log n). However, our array A would now be sorted, **meaning we performed a comparison based sort in O(n log log n) time.** This violates the key property that the worst case running time for any comparison-based sort must have a lower bound of at least n log n.

   Now to prove the lower bound Ω(n log n):

   Assume we have a list of n distinct numbers (distinct is the worst-case scenario), we know that **there are n! permutations** for this list, and **only one of these permutations is the list in its fully sorted order.** Since this algorithm is comparison-based, every comparison between two numbers can only yield binary information, namely greater than or less than in the integer case. This idea can be structured into a binary decision tree, where every node is the "state" or current ordering of the array and every "split" represents the two possible decisions after comparing two elements. The most efficient algorithm will always have the shortest path from the unordered "root" to the ordered "leaf". We therefore are trying to minimize the height of tree, since height is determined by the longest path (worst-case scenario). The tree must also contain n! nodes since that is the max number of permutations of the worst case input of size n. **A minimum height binary tree is a "full tree" and the height h of a full binary tree containing n! nodes is $h = \log_2(n!)$.** Finally, we know from Assignment 1 that log (n!) is Ω(n log n).