Homework 2

Solution Key

Problem 1

1. Assume that we want to sort sequences of numbers with the following property:
   P: "Each element in the input sequence is placed in the input at most k locations away from its placement in the output ordered sequence".
   Design a modification of Selection sort which runs in time $O(nk)$.
2. Analyze the worst-case running time of Insertion Sort for strings with the **P** property.
3. For input sequences exhibiting property **P**, which of Mergesort, Insertion Sort, or Standard Selection sort would you use for better performance? How does your choice change for different values of k? Motivate your answer.
4. Assume now that we are looking to sort input sequences in which there is a subsequence of size k which is already sorted. Show that using one among Selection, Insertion, or Bubble sort it is possible to sort such inputs in $O(n(n-k))$ steps.

1. We propose the following modification of selection sort:

Algorithm 1 k-Selection Sort

```
procedure k_selection_sort(A, k)
    for i ∈ [0, |A| - 1] do
        min_idx ← i
        for j ∈ [i+1, i+k] do    ▷ Next minimum element is only k positions away
            if A[min_idx] > A[j] then
                min_idx = j
        A[i], A[min_idx] = A[min_idx], A[i]    ▷ Swaps the two elements
```

Our input is a list of length n and a value k which denotes how far an element in the unsorted input is from its position in the sorted list. The outer loop iterates through the entire list, running n times. During an iteration of the outer loop, the inner for loop runs at most k times to search for the next minimum element. This is because the next minimum element should be at most k positions away. Since the inner loop runs k times for every iteration of the outer loop which runs a total of n times, the runtime is $O(nk)$.

2. Recall that insertion sort has the following structure:

Algorithm 2 Insertion Sort

```
procedure insertion_sort(arr):
    for i $\in$ [1, |A| - 1] do
        key $\leftarrow$ arr[i]
        j $\leftarrow$ i - 1
        while j $\geq$ 0 and key < arr[j] do
            arr[j + 1] $\leftarrow$ arr[j]
            j $\leftarrow$ j - 1
        arr[j + 1] $\leftarrow$ key
```

Note that in the while loop we are moving elements of the array that are greater than `key` to one position ahead of their current position. Since any element of the unsorted array is at most k positions away from their original position the while loop will run at most k times.

The outer loop will iterate through the entire list, and at each iteration the inner loop will run at most k times. This makes the overall complexity $O(nk)$.

3. Mergesort will always take $O(n \log n)$ time, in the best and worst case scenarios (it is a non-adaptive algorithm). Indeed, for an arbitrary list, mergesort will proceed as follows:

   - We recursively split the list in half until we're down to 1 element lists, taking $O(n)$ time
   - We merge lists recursively, taking $O(n)$ time at each step (one can merge two sorted lists by comparing them pointwise, which **always** takes $O(\text{[size of list]})$ time since one must always compare everything in at least one of the lists to something at least one, and at step k one has $O(2^k)$ lists with size $O(n/2^k)$)
   - We iterate the former step $O(\log n)$ times.

   This, therefore, takes $O(n \log n)$ time in totality.

   On the other hand, we've shown above that for such "k-pseudo sorted" lists, insertion sort takes $O(nk)$ time without modification. On the other hand, we had to modify selection sort to guarantee this $O(nk)$ time, in particular by ensuring we only did k comparisons for each element. But standard selection sort always does $O(n^2)$ comparisons, so without modification it will run in $O(n^2)$ time. This means that if $k < \log(n)$, insertion sort and modified selection sort will asymptotically outperform mergesort(!). If $k \geq \log(n)$, we are either agnostic about the use of mergesort vs. the others (if equality holds) or prefer mergesort.

4. Insertion sort will suffice for such a sequence S. Indeed, consider the number of comparisons which one will have to make for each element, which completely determines the complexity of insertion sort. If an element e is in S but not in our k-sorted subsequence, then in general we will have to do $O(n)$ comparisons when inserting e. There are $(n-k)$ such elements, so this will take $O(n(n-k))$ time. On the other hand, if e is in a k-sorted subsequence, then e will only be compared to at most $(n-k)$ elements: indeed, anytime e would be compared to another element of the k-sorted subsequence, insertion would terminate, since any element before e in said sequence has strictly lower key. Since there are

k such elements, this will take $O(k(n-k))$ time. Thus, insertion sort on such a sequence will take $O(n(n-k)) + O(k(n-k)) = O(n(n-k))$ (since $k \leq n$) time.

Problem 2

Joanna is in charge of a law firm working on an important case, where she needs to find a specific folder which is stored in one of the company filing cabinets. She assigns this task to Rohit the Paralegal.

Rohit has at most N workers at his disposal to complete this task. The filing cabinet has M shelves, and each shelf has F_i folders. Assume that there are always more shelves than Rohit has workers. Joanna wants to minimize chaos, so she asks Rohit to assign workers to a contiguous sub-array of shelves.

Rohit wants to divide the work as evenly as possible among the workers. Help Rohit come up with an algorithm to find the **minimum number of folders** that each of his workers has to look at in order to find the specific folder.

Since Joanna is a strict boss, please give an algorithm that runs in $O(M \log(S))$, where S is the total number of folders in the entire filing cabinet.

Solution: We use binary search on our "output space" (the number of folders a worker can be assigned) to determine the optimal workload for the workers. Notice that the minimum workload a worker can be assigned is 0 folders, while the maximum workload he/she can be assigned is S. Furthermore, a worker can only search a integral number of folders. Therefore, we wish to apply binary search on the set $\{0, 1, \cdots, S\}$.

We set our left extremity, L, of our binary search to $\max\{F_i\}$ and our right extremity R to S. Our candidate optimal workload, W, will then be the middle value between these two extremities. We then verify W by simulating the assignment of contiguous segments of shelves to workers keeping in mind that no worker should be assigned no more than W folders. If we are able to make it through the entire file cabinet by assigning at most W folders to each worker then W is a candidate solution to the problem. In this case we set $R = W$ since any workload greater than W will also satisfy the constraints of the problem. However, if we require more than N workers to search the cabinet with a maximum workload of W for each worker then we need to increase the maximum workload. Therefore, we set $L = W + 1$. We repeat this search until binary search terminates.

Recall that the runtime of binary search is $O(\log n)$ where n is the length of the list we are searching through. Furthermore, at each iteration of our binary search we iterate through our cabinet (array) of folders to simulate the workload. In the worst case we would have to iterate through the entire list at each iteration of binary search which would take $O(M)$ time. For each of the $O(\log S)$ steps we perform a total of $O(M)$ steps for a total runtime of $O(M \log S)$.

Problem 3

Let S be an array of n unique integers. An inversion in S is a pair of indices i and j such that $i < j$ but $S[i] > S[j]$. Describe an algorithm running in $O(n \log n)$ time for determining the number of inversions in S.

Solution: The idea is similar to merge sort. We need to divide the array into two equal halves in each step until the base case is reached. The base case of recursion is when there is only one element in the given half. We will count the number of inversions when two halves of the array are merged. Let i be the index used to iterate through the first array and j be the index used to iterate through the second array. Let $mid - 1$ be the last element in the first array. Then, while merging, if we find a pair (i, j) such that $a[i] > a[j]$ then we will have $(mid–i)$ inversions resulting from this inequality. This is because the left and right subarrays are already sorted, so all the remaining elements in left subarray $(a[i+1], a[i+2] \dots a[mid-1])$ will also be greater than $a[j]$. We can get the total number of inversions by adding the number of inversions in the first half, the number of inversion in the second half and the number of inversions by merging the two.