CS 218, Fall 2025, Assignment #1 **Due: 7:00am, 10/14, 2024**
Name: First Name Last Name **Student ID: XXX**

1 A Complex Complexity Problem (24 pts)

Yourae recently learned asymptotical analysis. The key idea is to evaluate the *growth* of a function. For example, she now knows that n^2 grows faster than n, but for more complicated functions, she feels confused. Can you help Yourae compare the functions below? We use $\log n$ to denote $\log_2 n$. $n! = 1 \times 2 \times \cdots \times n$. **Explain each of your answers briefly.**

For the following questions, use o, Θ or ω to fill in the blanks.

1. $\sqrt{n} =$ ___$(\log^2 n)$
2. $3^n =$ ___$(n!)$
3. $3^n =$___(2^n)
4. $\log\sqrt{n} =$___$(\log n^2)$

2 Solve Recurrences (18 pts)

For all of them, you can assume the base case is when n is a constant, $T(n)$ is also a constant. Use $\Theta(\cdot)$ to present your answer. Show your work.

(1) $T(n) = T(\sqrt{n}) + 1$

(2) $T(n) = T(n-1) + n^2$

(3) $T(n) = T(n/2) + n\log n$

3 (42 pts) Special Sorting

We are all familiar with the sorting algorithms. The algorithm will take an input array $A[]$ with size n, and reorder the elements to put them in order. In many cases, a sorting algorithm will find two elements in the input and swap them. By performing a series of "swaps", all elements are put in order. For simplicity, let's assume all input elements are distinct.

Now let's consider a special sorting algorithm: this algorithm can only move elements by swapping *adjacent* elements. Let's call such a swap an adjacent swap. In other words, to reorder the elements, the only operation that is allowed is to swap $A[i]$ with $A[i+1]$, for some i. It can read and compare elements arbitrarily (but only move elements by adjacent swaps). We are interested in studying that, how "powerful" such an algorithm is, and how does that affect the upper and lower bounds of sorting algorithms. In general, we will consider a computational model, such that we only count the number of adjacent swaps as the cost of the algorithm.

Example: Given a sequence $6, 4, 1, 3$, we can use five adjacent swaps to sort it: $1 \leftrightarrow 4, 1 \leftrightarrow 6, 4 \leftrightarrow 3, 6 \leftrightarrow 3, 6 \leftrightarrow 4$. This is also the fewest number of adjacent swaps needed to sort the sequence.

1. (5 pt) To sort n elements by an algorithm, what is the lower bound of the *number of comparisons* needed? In other words, how many comparisons do you need to guarantee to sort any input sequence with n elements? You should use $\Theta(\cdot)$ to present your answer.

   Note that, by these number of comparisons, we can actually know the ordering of all elements, but we still need to reorder them, which have to be done by the adjacent swaps.

2. (8 pt) If we are allowed to swap any pair of elements in the array, can you come up with an algorithm with $O(n\log n)$ swaps to sort an array of size n?

3. (8 pt) Design an algorithm, such that it always uses $O(n^2)$ adjacent swaps to sort an array of size n.
4. (6 pt) Prove that, the number of "adjacent swaps" needed to sort any n elements is $\Omega(n^2)$ (i.e., $\Omega(n^2)$ is a lower bound). This means that, given an arbitrary input sequence with size n, no algorithm can guarantee to sort the sequence in $o(n^2)$ adjacent swaps.
5. (15 pt) Given a sequence of size n, design an algorithm with $O(n \log n)$ time complexity to compute the smallest number of adjacent swaps needed to sort the input. This means that, although the number of adjacent swaps can be $\Omega(n^2)$, you can compute this number in a faster way. Obviously, simulating the algorithm (e.g., using the algorithm in subproblem 3) is not fast enough.

Hint

For subproblems 2 and 3, one way you can try is to see if any *existing* sorting algorithm can satisfy the requirements.

4 (16 pt) Being Unique

You are given an array A of n numbers from the set $\{1, 2, \ldots, n\}$. The array has the "unique-in-range" property if for every range $[i, j]$ there exists an element $A[k]$ (where $i \leq k \leq j$) such that the number $A[k]$ occurs just once in that range.

For example, the sequence: 1 2 1 3 1 2 1 4 1 2 1 3 1 has the unique-in-range property. As does 1 2 3 4 5 6. But the sequence 1 2 3 1 2 3 does not (it fails on the whole range: every number appears twice).

Give an algorithm to determine if a given array has the property. It should have runtime $O(n \log n)$ or better. Prove the runtime and the correctness of your algorithm.