This is an optional problem to work on for extra credit. You can submit this without finishing all of it, and you will be awarded points based on how much you (correctly) complete.

Algorithm 1 Bubble Sort

1: procedure BubbleSort($X$)
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  
12:  
13:  
14:  
15: end procedure

In this problem, we study the worst-case and expected time complexity of two sorting algorithms. Algorithm 1 is called bubble sort because each element in turn “bubbles” up through the sorted part of the list to its proper place. It is simple to understand and analyze, but as we will soon see, it often requires many comparisons. Algorithm 2 is called quicksort; it is more complicated than bubblesort, but often requires fewer comparisons. You will need to study their pseudocode to answer some of the below analysis questions, but we provide Matlab implementations for the implementation questions, where we use Monte-Carlo techniques to study their performance characteristics.

(a) Compute the maximum possible number of comparisons between list elements made by the deterministic bubble sort in Algorithm 1 in terms of $n$.

(b) Compute the maximum possible number of comparisons between list elements made by the deterministic quick sort in Algorithm 2 in terms of $n$. *Hint*: the worst case occurs when this list is already sorted.

(c) To avoid the worst-case, randomized sorts first randomly permute the list, and then sort. Let $c$ be an integer random variable equal to the number of comparisons made by randomized bubble sort. Compute $E[c]$ for randomized bubble sort in terms of $n$. You may want to use the expression $x(randperm(n))$, which generates a random permutation of some length $n$ vector $x$.

(d) Monte Carlo sampling can be used to estimate the cumulative distribution function of $c$. Use Matlab to execute the provided randomized quicksort 1000 times for shuffled lists of the integers 1..10, and plot the empirical CDF $F_c(x)$, for both bubble sort and quicksort. Compute and report the average and maximum number of comparisons across your 1000 Monte Carlo trials.

(e) Repeat part (c) for lists of length $n \in \{10, 20, \ldots, 100\}$, and plot both the average and worst-case number of comparisons (over 1000 runs), as a function of $n$. For both the average and
Algorithm 2 Quicksort

1: procedure PARTITION($X, i, j, \text{randomize}$)  
2:     Input: Array $X$, start, end indices $i < j$, whether to randomize.  
3:     Output: Pivot index $p$ such that $i \leq p \leq j$, number of comparisons made $c$.  
4:     Side Effects: $X_{i:j}$ are reordered such that $\forall a < p, X_a \leq X_p$ and $\forall b > p, X_b \geq X_p$.  
5: if randomize then  
6:     $u \sim \text{Uniform}(i, \ldots, j)$  \hfill $\triangleright$ Randomly select a pivot value  
7: else  
8:     $v \leftarrow X_i$  \hfill $\triangleright$ Pivot on the first value.  
9: end if  
10: $p \leftarrow i$  \hfill $\triangleright$ Initialize pivot index to $i$.  
11: for $k \in i..j$ do  
12:     if $X_k < v$ then  \hfill $\triangleright$ Check if this element is smaller than the pivot value.  
13:         swap($X_p, X_k$)  \hfill $\triangleright$ Swap this to earlier in the list.  
14:         $p \leftarrow p + 1$  \hfill $\triangleright$ Increment the pivot position.  
15: end if  
16: end for  
17: swap($X_p, X_j$)  \hfill $\triangleright$ Swap the pivot into the correct position.  
18: $c \leftarrow j - i + 1$  \hfill $\triangleright$ # of comparisons is always $|i, i + 1, \ldots, j|$  
19: return $p, c$  \hfill $\triangleright$ Return the pivot, and the number of comparisons made.  
20: end procedure

21: procedure QUICKSORTSUBLIST($X, i, j, \text{randomize}$)  
22:     Input: Array $X$, start index $i$, end index $j$, whether to randomize.  
23:     Output: $c$, the number of comparisons made during this sorting routine.  
24:     Side Effect: The sublist $X_{i:j}$ is sorted.  
25: if $j < i$ then  
26:     return 0  \hfill $\triangleright$ Base case: Single-element or empty sublist is already sorted.  
27: end if  
28: $p, cp \leftarrow \text{Partition}($$X, i, j, \text{randomize})$  \hfill $\triangleright$ Select pivot and partition.  
29: $cl \leftarrow \text{QuickSortSUBLIST}($$X, i, p - 1)$  \hfill $\triangleright$ Quicksort $X_{i:p-1}$.  
30: $cr \leftarrow \text{QuickSortSUBLIST}($$X, p + 1, j)$  \hfill $\triangleright$ Quicksort $X_{p+1:j}$.  
31: return $cp + cl + cr$  \hfill $\triangleright$ Total comparisons required during partition and recursion.  
32: end procedure

33: procedure QUICKSORT($X, \text{randomize}$)  
34:     Input: Array $X$, whether to randomize.  
35:     Output: $X$ sorted, the number of comparisons $c$ made during sorting.  
36:     $X \leftarrow \text{copy}(X)$  \hfill $\triangleright$ Make a copy of the list  
37: $c \leftarrow \text{QUICKSORTSUBLIST}($$X, 0, \text{length}(X), \text{randomize})$  \hfill $\triangleright$ Sort the copy  
38: return $X, c$  
39: end procedure
Algorithm 3 Perturb

1: **procedure** Perturb(X, p)
2: **Input:** Array X, number of perturbations p.
3: **Output:** Array Y, equal to X after making p random perturbations.
4: Y ← copy(X)
5: n ← length(X)
6: for i ∈ [1..p] do
7:   j ∼ Uniform(1..n − 1)
8:   swap(Y[i], Y[i+1])
9: end for
10: return Y
11: **end procedure**

worst-case number of comparisons, does this function appear to be quadratic \(c(n) \in \Theta(n^2)\), thus \(c(n) \propto n^2\), *linearithmic* \(c(n) \in \Theta(n), \) thus \(c(n) \propto n \log(n)\), or *linear* \(c(n) \in \Theta(n), \) thus \(c(n) \propto n\)? How does this empirical evidence fit in with your analysis for the expected and worst-case complexities of quick sort and bubble sort?

(f) An adaptive sort performs well when inputs are approximately sorted, whereas a non-adaptive sort doesn’t generally benefit from approximately-sorted input. For both bubble sort and quick sort, answer whether you think the sort is adaptive, and explain why or why not.

(g) We supply the function Perturb(X, p) (pseudocode given in Algorithm 3) that “perturbs” a list \(X_1, X_2, \ldots, X_n\) by uniformly randomly selecting an index \(i\) between 1 and \(n − 1\), swapping \(X_i\) with \(X_{i+1}\), and repeating this process \(p\) times. Starting with the sorted list \(X = 1, \ldots, 100\), apply Perturb(X, p), for \(p \in \{10, 100, 1000\}\). Now use bubble sort and quick sort to the resulting lists. Repeat this process 1000 times, and report the average number of comparisons made in each case. Do these results support your answer to part (f), and why or why not?

**Answer:**