Lab 4

1. A 10-sided die—having values 1, 2, . . ., 10 on its faces—is tossed repeatedly.
   A. What is the expected number of tosses required to get a 2?
      -> 1 / (1 / 10) = 10
   B. What is the expected number of tosses required to get a total of three 2's? Assume that each of the values 1, 2, . . ., 10 is equally likely to appear. Explain your answer.
      -> 3 / (1 / 10) = 30

2. Is there a comparison-based sorting algorithm which, when run on an array containing exactly 4 distinct integers, requires only 4 comparisons to sort the integers, even in the worst case? Can it be done with 5 comparisons? Prove your answers.
   - By the Lower Bound theorem, any comparison-based sorting algorithm requires at least $\lceil \log n! \rceil$. With n = 4: $\lceil \log 4! \rceil = \lceil \log 24 \rceil = 5$

3. Devise an algorithm that rearranges the elements of an input array of size n in a random way. What is the running time of your algorithm? Hint. There is a way to do this in O(n) time.
   - Algorithm rearrangeArr(arr)
     Input: An array with number elements
     Output: an array with elements arranged in random order

     ```
     n <- arr.size()
     for index <- 0 to n-1 do
             randomIndex = random(index, n-1)
             swap(arr, index, randomIndex)
     return arr
     ```

     Running time: O(n)

4. Goofy has thought of a new way to sort an array arr of n distinct integers:
   a. Step 1: Check if arr is sorted. If so, return.
   b. Step 2: Randomly arrange the elements of arr (see part 3(A))
   c. Step 3: Repeat Steps 1 and 2 until there is a return.
      Answer the following:
   A. Will Goofy's sorting procedure work at all? Explain
      -> Yes, after rearranging, there is a chance that all elements are randomly placed in the correct order of a sorted array.
   B. What is a best case for GoofySort?
      -> when the initial array is already sorted. In this case, Step 1 will immediately return, as the array is already sorted. Therefore, no further steps are required.
   C. What is the running time in the best case?
      -> O(1) because Step 1 will return immediately without any further iterations or rearrangements.
   D. What is the worst-case running time?
      -> O(∞) since there is no guarantee that the array will eventually become sorted
   E. What is the average case running time? Hint: What is the probability that a randomly arranged array of integers happens to be in sorted order? Then think about the coin-flipping model introduced in the slides.
      -> Possible number of distinct arrays: n!
         Number of tries to get a success: 1 / p = 1 / (1/n!) = n!
         Average case running time: O(n * n!)
   F. Is the algorithm inversion-bound?
      -> Yes. Each step of rearranging the elements has a random component, so it is likely to introduce new inversions or maintain existing ones. As a result, the number of

inversions can potentially increase with each iteration, making it difficult for the array to become sorted.

5. In our average case analysis of QuickSort, we defined a good self-call to be one in which the pivot x is chosen so that number of elements < x is less than 3n/4, and also the number of elements > x is less than 3n/4. We call an x with these properties a good pivot. When n is a power of 2, it is not hard to see that at least half of the elements in an n-element array could be used as a good pivot (exactly half if there are no duplicates). For this exercise, you will verify this property for the array A = [5, 1, 4, 3, 6, 2, 7, 1, 3] (here, n = 9). Note: For this analysis, use the version of QuickSort in which partitioning produces 3 subsequences L, E, R of the input sequence S.
   a. Which x in A are good pivots? In other words, which values x in A satisfy:
      i. the number of elements < x is less than 3n/4=6.75, and also
      ii. the number of elements > x is less than 3n/4
      -> 2, 3, 4. 5
   b. Is it true that at least half the elements of A are good pivots?
      -> yes, 5 elements of A (2, 3, 3, 4, 5) are good pivots.

6. Devise an algorithm that performs sideways sorting on the elements of a length-n integer array. When an array is sideways-sorted, the elements are arranged as follows: position 0: the smallest integer position 1: the largest integer position 2: the second smallest integer position 3: the second largest integer etc. For example, when you sideways sort the input array {1, 2, 17, -4, -6, 8} you get: {-6, 17, -4, 8, 1, 2}. (Notice that –6 is the smallest, 17 the largest, -4 second smallest, 8 second largest, etc.) Answer the following:
   A. What is the asymptotic running time of your algorithm?
   - Algorithm sidewaysSort(arr)
     Input: A length-n integer array
     Output: an array is sideways sorted

```
sortedArr = sort(arr)
n <- arr.size()
ans <- new array[n]
leftPointer <- 0
rightPointer <- n - 1
for index <- 0 to n-1 do
    if index is odd do
        ans[index] = sortedArr[leftPointer]
        leftPointer = leftPointer + 1
    else do
        ans[index] = sortedArr[rightPointer]
        rightPointer = rightPointer - 1
return ans
```

     -> It takes O(nlogn) to sort the array, O(n) to do sideways sorting.
     Therefore, the asymptotic running time is O(blogs) + O(n) = O(nlogn)

   B. Prove that it is impossible to obtain an algorithm to do sideways sorting of an integer array that runs asymptotically faster than the algorithm you created in Part A.
      -> In order to perform sideways sorting, the algorithm needs to sort the array in non-decreasing order. Sorting an array of n elements requires at least Ω(n log n) operations in the average case. Since the sorting step alone takes Ω(n log n) operations, any additional operations required for sideways sorting will not significantly affect the overall asymptotic running time.

Therefore, any algorithm that achieves an asymptotically faster running time than O(n log n) for sideways sorting would contradict the lower bound of Ω(n log n) required for sorting the array itself. Hence, it is impossible to obtain an algorithm that performs sideways sorting with an asymptotically faster running time than the algorithm described in Part A.