VE 281 Homework 1

Name: , ID No.:

- **Exercise 1.**

  Definition 1 (o-Notation)**.** *Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$, written as $f(n) = o(g(n))$, if*

  $$\forall c > 0. \exists n_0. \forall n \geq n_0. f(n) < cg(n).$$

  An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows:

  $$f \mathcal{R} g \text{ if and only if } f(n) = \Theta(g(n)).$$

  A complexity class is an equivalence class of $\mathcal{R}$.

  The equivalence classes can be ordered by $\prec$ defined as: $f \prec g$ iff $f(n) = o(g(n))$.

  Example: $1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{\frac{3}{4}} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$.

  Please order the following functions by $\prec$ and give your (verbal or math) explanation:

  $$(\sqrt{2})^{\log n}, (n+1)!, ne^n, (\log n)!, n^3, n^{1/\log n}.$$

- **Exercise 2**

  Prove that $\log(n!) = \Omega(n \log n)$. Notice that after you prove this, you will understand the lower bound of comparison sort is $\Theta(n \log n)$.

- **Exercise 3**

  Show how QUICKSORT can be made to run in $O(n \log n)$ time in the worst case.

- **Exercise 4**

  Let A be a list of n (not necessarily distinct) integers. Describe an $O(n)$ -algorithm to test whether any item occurs more than $\lceil n/2 \rceil$ times in A.

- **Exercise 5**

  Modify the $Merge(int * \ a, int\ left, int\ mid, int\ right)$ function taught in class and make it in-place (provide pseudo code). Explain why the new mergeSort might still run slower than quickSort in practice. Hint: Piazza.

- **Exercise 6**

  Let A be a sorted array of integers and S a target integer. Design algorithms for determining if there exist a pair of indices i, j(not necessarily distinct) such that A[i]+A[j]=S.

  1. Design an $O(n^2)$ algorithm (stating it in plain English is OK).
  2. Design an $O(nlogn)$ algorithm (stating it in plain English is OK).

- **Exercise 7**

  What is the most efficient sorting algorithm for each of the following situations and briefly explain:

  1. A small array of integers.
  2. A large array of integers that is already almost sorted.
  3. A large collection of htegers that are drawn from a very small range.
  4. Stability is reqired, i.e., the relative order of two records that have the same sorthg key should not be changed.

- **Exercise 8**

  Suppose you are given a set of mames and your job is to produce a set of unique first names. If you just remove the last name from all the names, you may have some duplicate first names. How would you create a set of first names that has each name occurring only once? Specifically, design an efficient algorithm for removing all the duplicates from an array.

- **Exercise 9**

  Suppose $\lim_{n\to\infty} f_1(n)/g_1(n)$ and $\lim_{n\to\infty} f_2(n)/g_2(n)$ exist, and we assume that all the functions are larger than 0 when $n > 0$. Judge whether the following statement is correct or not when $n \to \infty$. Justify your answers.

  1. $n \log n = O(n)$.
  2. $2^n = O(n!)$.
  3. If $f_1(n) = \Omega(g_1(n))$, $f_2(n) = \Omega(g_2(n))$, then $f_1(n)f_2(n) = \Omega(g_1(n)g_2(n))$.
  4. If $f_1(n) = \Theta(g_1(n))$, $f_2(n) = \Theta(g_2(n))$, then $f_1(n) + f_2(n) = \Theta(\max\{g_1(n), g_2(n)\})$.

- **Exercise 10**

  We want to find the 6th largest element, which is 6, in the following array, Insertion sort is a simple and fast sorting algorithm when the length of array n is short. However, when n goes large, insertion sort may not be the best choice, as the worst case time complexity is $O(n^2)$. We can speed up insertion sort by combining it with `merge` in `mergeSort` we learnt in the lectures,

  Alg. 1: `timSort`$(a[\cdot], x)$

  Input : an array a of n elements, an integer $x > 0$ (you can assume that $x \ll n$)
  Output: the sorted array of a

  ```
  for i ← 0; i < n; i += x do
      insertionSort(a, i, min(i + x − 1, n − 1));
  for step ← x; step < n; step *= 2 do
      for left ← 0; left < n; left += 2 × step do
          mid ← left + step − 1;
          right ← min(left + 2 × step − 1, n − 1);
          merge(a, left, mid, right);
  ```

  This algorithm is used as the default sorting algorithm in Java and Python. Here, we assume that $x \ll n$ is a known constant.

  1. Suppose $n = 1000$ and $x = 32$. How many times will `insertionSort` be performed?
  2. Suppose $x = 32$. Express in terms of n how many comparisons in the worst case will be performed in `insertionSort`.
  3. Express the worst case running time of the whole algorithm in terms of the big-Oh notation.
  4. Is this algorithm in-place? If not, express the additional space needed in terms of the big-Oh notation.