CISC2006: Algorithm Design and Analysis
Homework 2 **Due Date:** 23:59, 04 March 2022

Rules: Discussion of the problems is permitted, but writing the assignment together is not (i.e. you are not allowed to see the actual pages of another student). You can get at most 100 points if attempting all problems. Please make your answers precise and concise.

1. (15 pts) Given two sorted arrays A and B, design a linear ($O(|A|+|B|)$) time algorithm for computing the set C containing elements that are in A or B, but not in both. That is, $C = (A \cup B) \setminus (A \cap B)$. You can assume that elements in A have different values and elements in B also have different values. Please state the steps of your algorithm clearly, prove that it is correct, and analyze its running time.

   Solution: The following algorithm is modified from the merging algorithm of merge-sort. The only difference is that when the two pointer point to elements with the same value, both of them will be removed, instead of being included in C. Since in each while-loop either i or j is increased by one, the algorithm has $O(|A|+|B|)$ while loops. Therefore, the complexity of the algorithm is $O(|A| + |B|)$.

Algorithm 1: ExclusiveUnion(A, B)

suppose $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_m\}$ and the two arrays are sorted from minimum value to maximum.
initialize $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$ and C be an empty set.
let $a_{n+1} = \infty$ and $b_{m+1} = \infty$.
while $i \leq n$ *or* $j \leq m$ **do**
    if $a_i < b_j$ **then**
        set $c_k \leftarrow a_i$, $k \leftarrow k + 1$ and $i \leftarrow i + 1$.
    else if $a_i > b_j$ **then**
        set $c_k \leftarrow b_j$, $k \leftarrow k + 1$ and $j \leftarrow j + 1$.
    else if $a_i = b_j$ **then**
        set $i \leftarrow i + 1$ and $j \leftarrow j + 1$.
Output: C.

Correctness. The correctness of the algorithm follows from the fact that if there exists element $a_x = b_y$, then for the first time when a pointer point to a_x or b_y happens, the pointed element will not be included into C because the other element is strictly smaller. Then when the two pointers point to a_x and b_y, both elements will be skipped without being included into C.s

2. (20 pts) Given a sequence of numbers A, design an algorithm that counts the number of inversions, where an inversion is a pair (a_i, a_j) such that $i < j$ and $a_i > a_j$. Please state the steps of your algorithm clearly and analyze its running time.

   (a) (10 pts) Given two sorted arrays L and R, design a linear ($O(|L| + |R|)$) time algorithm that counts the number of pairs (l, r) such that $l \in L$, $r \in R$ and $l > r$.

   Solution: Suppose l_i is the i-th element of L and c_i is the number of pair (l_i, r) such that $r \in R$ and $l_i > r$. The solution is given by $\sum_{i=1}^{n} c_i$, where $n = |L|$. The key observation is that $c_1 \leq c_2 \leq \cdots \leq c_n$. Therefore the computation of $c_1, \ldots, c_n$ can be done in linear time by scanning L and R once.

Algorithm 2: CountPair(L, R)

```
suppose L = {l_1, ..., l_n} and R = {r_1, ..., r_m} are sorted from min to max.
initialize i ← 1, j ← 1, c ← 0.
while i ≤ n do
    if j = m + 1 or l_i ≤ r_j then
        set i ← i + 1 and c ← c + (j − 1).
    else if l_i > r_j then
        set j ← j + 1.
Output: c.
```

Since in each while-loop $i + j$ increases by one and we have $i + j \leq n + m$, there are at most $n + m$ while-loops. Since each of while-loop executes in constant time, the complexity of the algorithm is $O(n + m) = O(|L| + |R|)$.

   (b) (10 pts) Suppose we have a linear time algorithm for question (a), design a $O(n \log n)$ time algorithm that computes the number of inversion in A.

   Solution: We design CountInversion(A) that returns (c, B) as follows, where c is the number of inversions in A, and B is the sorted version of A:

   - if $|A| \leq 1$ then we return $(0, A)$;
   - otherwise let $k = \lfloor |A|/2 \rfloor$ and partition A into L and R, where L contains the first k elements of A and R contains the remaining elements. Call CountInversion(L) and get (c_L, B_L); call CountInversion(R) and get (c_R, B_R). By definition c_L and c_R are the number of inversions within L and R, respectively, and B_L and B_R are their sorted versions. Then we run the algorithm from question (a) to count the number of inversions c between B_L and B_R, and use the merging algorithm of Merge-Sort to merge B_L and B_R into a sorted version B of A. Return $(c_L + c + c_R, B)$.

   Since excluding the recursive calls, CountInversion(A) executes in $O(n)$ time, the complexity of the algorithm is $T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$.

3. (10 pts) Suppose we have $T(n) \leq c = O(1)$ for all $n \leq 3$, and for every $n \geq 4$, we have

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + c \cdot n.$$

Use Mathematical Induction to prove that $T(n) = O(n \log n)$ for all $n \geq 4$.

Solution:

We use Mathematical Induction to prove that $T(n) \leq 2c \cdot n \log(2n)$ on $n \geq 1$.

Base Case: when $n = 1, 2, 3$, we have $T(n) \leq c \leq 2c \cdot n$; when $n = 4$, we have

$$T(4) \leq T(1) + T(3) + c \cdot 4 \leq 6c \leq 2c \cdot 4 \log 4.$$

Induction Step: (**Induction Hypothesis**) Assume that the statement is true for all $k \leq n - 1$, we consider the case of n:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + c \cdot n \\ &\leq 2c \cdot \frac{n}{4} \log(\frac{n}{2}) + 2c \cdot \frac{3n}{4} \log(\frac{3n}{2}) + c \cdot n \\ &\leq 2c \cdot \frac{n}{4} \log(2n) - c \cdot n + 2c \cdot \frac{3n}{4} \log(2n) + c \cdot n = 2c \cdot n \log(2n). \end{aligned}$$

Therefore, the statement is also true for n. By Mathematical Induction, for all $n \geq 1$ we have $T(n) \leq 2c \cdot n \log(2n)$, which implies $T(n) = O(n \log n)$.

4. (15 pts) Given an array $A = \{a_1, a_2, \ldots, a_n\}$ of n integers in the range $[0, n^2-1]$, design an algorithm for sorting A in $O(n)$ time. Please state the steps of your algorithm clearly and analyze its running time.

   Solution: Note that we can express each element $a_i \in A$ as $x_i \cdot n + y_i$, for some $x_i, y_i \in \{0, 1, \ldots, n-1\}$. The algorithm consists of two phases.

   In the first phase, we sort elements in increasing order of y_i. Since y takes values in $\{0, 1, \ldots, n-1\}$, this can be done in $O(n)$ time using the linear time sorting algorithm from the lecture nodes. Specifically, we initialize n empty doubly-linked lists and organize their heads in to an array $L[0, \ldots n-1]$. Then we scan the elements once: for each element $a_i = x_i \cdot n + y_i$, we put it into the y_i-th list (whose head is $L[y_i]$). After scanning through all elements (using $O(n)$ time), we output elements in $L[y]$, for $y = 0, 1, \ldots, n-1$ to get a list of elements B sorted by y.

   In the second phase, we use the linear time sorting algorithm to sort elements in B by x, which does not change the ordering of elements with the same x value. Specifically, we do a similar processing as above by scanning elements in B one-by-one and inserting $a_i = x_i \cdot n + y_i$ into the x_i-th list. Then we output elements in $L[x]$, for $x = 0, 1, \ldots, n-1$ into a sorted array C; for each $L[x]$, elements that are inserted first are output first.

   Since the algorithm only makes use of the linear sorting algorithm twice, the running time is $O(n)$. The correctness follows from the fact that if $a_i = x_i \cdot n + y_i < a_j = x_j \cdot n + y_j$, then either (1) $x_i < x_j$, in which case a_i will appear in the sorted array C before a_j because elements in $L[x_i]$ are output first; or (2) $x_i = x_j$ and $y_i < y_j$, in which case a_i and a_j will be inserted into the same list $L[x_i]$ but a_i will be inserted and output first because $y_i < y_j$, i.e., a_i appear before a_j in B.

5. (20 pts) You are given n numbers $a_1, a_2, \ldots, a_n$. It takes constant time to check whether two numbers a_i and a_j are of the same value. The goal is to check whether more than half of the numbers have the same value. Please state the steps of your algorithm clearly, prove that it is correct, and analyze its running time.

   (1) (10 pts) Design an $O(n \log n)$ time algorithm to solve the problem.

   Solution: We first sort the elements in $O(n \log n)$ time. Then we scan the element while maintaining a counter, which is initialized to be 1. When we find an element that is the same as the previous one, add one to the counter; otherwise reset it to 1. Throughout the process we record the maximum value of the counter. Then we know that more than half of the numbers have the same value if and only if this maximum value is greater than $n/2$. Since scanning each element takes $O(1)$ time, the total running time is $O(n \log n)+O(n) = O(n \log n)$. The correctness of the algorithm follows from the fact that the counter $= k$ if and only if the algorithm scans k consecutive elements with the same value.

   (2) (10 pts) Design an $O(n)$ time algorithm to solve the problem. (Hint: Show that using linear time, the problem size can be reduced by at least half.)

   Solution: If n is odd, then pick an arbitrary number, e.g., a_n, and compare it with all other numbers to find out whether more than $n/2$ numbers have the same value as a_n. If yes, then we terminate and output "Yes"; otherwise remove a_n. Hence we can assume without loss of generality that n is even. Partition the numbers into $\frac{n}{2}$ pairs arbitrarily. Scan through all pairs and do the following:

   – if the two numbers in a pair are different, then remove both two numbers;
   – if the two numbers in a pair are the same, keep one of them.

   The scanning can be done in $O(n)$ time since we only need to perform $n/2$ comparisons. Then in linear time, we can reduce the problem size to at most $n/2$, because for each pair of numbers, at most one of them are kept. Moreover, we show that if the original instance has more than half of the numbers having the same value, so does the new instance: suppose more than half of $a_1, a_2, \ldots, a_n$ have value x, and k numbers are left after the above procedure. We show that out of the k numbers, more than $k/2$ numbers have value x:

   Suppose at most $k/2$ numbers out of the k numbers have value x. Note that each of the k numbers correspond to a pair of numbers having the same value. Hence we know that $n/2-k$ pairs are removed in the procedure. Moreover, each of these pairs contains at most one number having value x. Thus in the original instance, the number of numbers having value x is at most $2 \cdot \frac{k}{2} + \frac{n}{2} - k = \frac{n}{2}$, which is a contradiction.

   Repeat the above procedure until there are constant numbers left. Then we can check if more than half of them have the same value in constant time. Note that if more than half of them have the same value x, then we should return to the original instance and check whether more than $n/2$ numbers have value x, because we have only proved one direction of implication in the previous analysis: if the original instance has more than half of the numbers having the same value,

so does the new instance. In other words, if the original instance **does not** have more than half of the numbers having the same value, the new instance may have.

Hence there exists a constant c such that the total time spent is

$$T(n) = T(\lfloor n/2 \rfloor) + c \cdot n \leq 2c \cdot n = O(n).$$

(We can prove the above using mathematical induction: proof omitted.)

6. (40 pts) **Comparison of Sorting Algorithms**.

   In this problem you need to implement the different sorting algorithms, and compare their running times on different inputs. Implement each of the following algorithms as a function that takes as input an array (which can be very long), and outputs the sorted version of the array (from minimum to maximum).

   - InsertionSort : based on Insertion-Sort from the lecture notes;
   - BubbleSort : based on Bubble-Sort from the lecture notes;
   - SelectionSort : based on Selection-Sort from the lecture notes;
   - HeapSort : use heap implementation of priority queue for sorting;
   - MergeSort : based on Merge-Sort from the lecture notes;
   - QuickSort : use median of three random elements as the pivot.

   In the main function, we read an array $A = \{a_1, a_2, \ldots, a_n\}$ of different integers from a file, and use different sorting algorithm to do sorting. For each algorithm, test whether the returned array is sorted or not, and output its running time.

   Several test cases of array A will be provided.