Student ID: ______________________ **Student Name:** ____________________

CS203 Data Structure and Algorithm Analysis **Quiz 1**

Note 1: Write all your solutions in the question paper directly. You can ask additional answer paper if necessary

Note 2: If a question asks you to design an algorithm, full marks will be given if your algorithm runs with optimal time complexity

Note 3: If a question asks you to design an algorithm, you should **first** describe your ideas in general words, **then** write the pseudocode, and **end** with time complexity analysis.

Problem 1 [20 points] Recursive Algorithm.

```
RecursivePrint (int n)
1. if n ≤ 0 then
2.     return
3. if n%2=0 then
4.     Print(n) // print the integer n into the standard output
5.     RecursivePrint (n-1)
6. else
7.     RecursivePrint (n-1)
8.     Print(n)
```

(a)[4 points] The time complexity of the algorithm is ______O(n)_____

(b)[4 points] Let n = 5, write down its output: _________42135_________________

(c)[4 points] Let n = 8, write down its output: _______86421357_______________

(d)[8 points] Please modify the above pseudocode such that the output is "531246" when n=6. (You only can use the used functions and statements)

One possible solution:

Modify line 4, 5, 7 and 8

```
Line 4:    RecursivePrint(n-1)
Line 5:    Print(n)
Line 7:    Print(n)
Line 8:    RecursivePrint(n-1)
```

Problem 2 [30 points] Given two sorted arrays **A** and **B** (in ascending order), with n and m integers respectively. All (n+m) integers are distinct. Design an algorithm to find the median in all (n+m) integers.

Median definition: If (n+m) is an odd number, median is the (n+m+1)/2-th smallest number. If (n+m) is an even number, median is the (n+m)/2-th smallest number. For example, A = {1,3}, B = {2, 4}. The median in these four integers is "2".

[Brute Force] [10 Points]
Idea: Merge two sorted arrays into one, and then find the median.
Time complexity: O(n+m)

[Binary Search] [20 Points]
Idea: Use binary search to list numbers in either array, and then find their positions in all (n+m) integers. If the current number's position is latter than median, we seek ahead, otherwise we seek back, till we find the median or realize that it is not here, if so, repeating the above procedure in the other array will lead you to find it.

How to efficiently find one's position in all (n+m) integers? Find its predecessor (the maximum value that is smaller than it) in the other array by binary search.

Time complexity: O(log(n)log(m)).

Select O(log(n)) numbers to check whether it is median, the checking procedure (PredecessorSearch) cost O(log(m)) time. If the median is not in A, then we need to search it in B. Total time cost: O(log(n)log(m)) + O(log(m)log(n)), which is O(log(n)log(m))

[Smart algorithm] [25 Points]
Idea: The key idea is to get rid of all those numbers which can not be the median rather than directly find it.

Let k = (n+m+1)/2 (Integer division), which shows the position of the median in all (n+m) integers. There are (k/2-1) elements smaller than A[k/2-1] and B[k/2-1] respectively. Suppose that A[k/2-1] < B[k/2-1], then there are at most (k-2) elements smaller than A[k/2-1], which implies that A[k/2-1] can not be the k-th smallest number. Further more, all A[0..k/2-1] can not be the median and we can abandon all those k/2 numbers. The question turns to that we want to find (k-k/2) smallest number in the remaining arrays. Repeating the above procedure till we want to find the smallest number in the remaining arrays(A', B'), which is the smaller one between A'[0] and B'[0].

Pseudocode: (Array here starts with index 0, e.g. A[0..n-1])

```
int FindMedian(int A[ ], int B[ ])
1. n ← A.size, m ← B.size
2. k ← (n+m+1)/2    //Integer division
3. i ← 0, j ← 0    //i, j maintain the first index of A, B
4. while(true) do
5.      if i = n then
6.          return B[j + k - 1]
7.      end if
8.      if j = m then
9.          return A[i + k - 1]
10.     end if
11.     if k = 1 then
12.         return min(A[i], B[j])
13.     end if
14.     i' ← min(i + k / 2 - 1, n - 1)
15.     j' ← min(j + k / 2 - 1, m - 1)
16.     if A[i'] < B[j'] then
17.         k -= i' - i +1
18.         i = i' + 1
19.     else
20.         k -= j' - j + 1
21.         j = j' + 1
22.     end if
23. end while
```

Time complexity: O(log(n+m))
The search space is O((n+m+1)/2), which is O(n+m).
During each iteration, the search space will be cut in half. (Recall: We abandon (k/2) numbers in the first iteration, which is a good example to illustrate that.) Thus, we will have at most O(log(n+m)) iterations. For line 5 to 22, all the operations cost O(1) so that the total complexity is O(log(n+m)).

[Optimal Solution] [30 Points]

Idea:

Suppose that (n+m) is an odd number (even case can follow similar process):

We can split A and B into following parts:

Left Part	Right Part
A[0..i]	A[i+1..n-1]
B[0..j]	B[j+1..m-1]

max(Left Part) is the median if and only if:
(1) size of Left Part = size of Right Part + 1
(2) max(Left Part) < min(Right Part)

To satisfy condition (1):

Size of left part = i +1 + j +1
Size of right part = n - i + 1 + m - j + 1
⇒ i + j = (n+m+1)/2

Suppose that $n \le m$ (otherwise, we can simply exchange A and B)
for any $i \in [0, n]$, we can always find $j = (n+m+1)/2 - i$, where $j \in [0, m]$

To satisfy condition (2):

⇒ A[i] < B[j+1] and B[j] < A[i+1]

In a conclusion:

we want to find such pair (i, j) satisfying that

① A[i] < B[j+1] and B[j] < A[i+1] where $i \in [0, n]$, $j = (n+m+1)/2 - i$

The above condition can be reduced to:

② Find the biggest $i \in [0, n]$ such that A[i] < B[j+1] where $j = (n+m+1)/2 - i$

We can use binary search to find suitable i rather than enumerate each number.

Time complexity: O(log(min(n,m)))

We use binary search in the array with fewer elements and then check whether it satisfies the condition Thus, the time complexity is O(log(min(n,m))).

Problem 3 [20 points] Sorting Algorithm

ShellSort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

(a) [2 points] Records A[1], A[2], A[3],..., A[N] are said to be h-sorted, if _(D)_
(A) A[i] <= A[i+h] for 1<= i*h <= N
(B) A[h] <= A[i+h] for 1<= i <= N
(C) A[i] <= A[h] for 1<= i <= h
(D) A[i] <= A[i+h] for 1<= i <= N-h

(b) [2 points] An array that is first 7-sorted, then 5-sorted becomes ___(D)____
(A) 7-ordered
(B) 5-ordered
(C) both 2-ordered and 5-ordered
(D) both 7-ordered and 5-ordered

(c) [4 points] In the worst case, the quick sort algorithm and shell sort algorithm will degenerate to _bubble_ and _insertion_ sort algorithm, respectively.

(d) [3 points] Shell sort is more efficient than insertion sort if the length of input arrays is small. True or False? Why?

False. The main idea of shellSort is to avoid too many movements that insertion sort does by repeatedly applying h-sorted procedures. Since that the length of input array is small, the distance of elements' moving is quite limited, which implies that shellSort cannot be more efficient in this case.

(e) [9 points] Fill the following table to show the running steps of Shell-Sort Algorithm.

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
Input	13	26	18	53	7	17	95	86
4-Sorted	7	17	18	53	13	26	95	86
2-Sorted	7	17	13	26	18	53	95	86
1-Sorted	7	13	17	18	26	53	86	95

Problem 4 [7 points] Proof $5n^3 + 7n^2\sqrt{n} = O(n^3)$.

We need to find constants c1, c2 such that for all n ≥ c2, it holds

$$5n^3 + 7n^2\sqrt{n} \le c1 \cdot n^3$$

Towards this purpose, inspect the inequality:

$$\Leftrightarrow 7n^2\sqrt{n} \le (c1 - 5)\, n^3$$

$$\Leftrightarrow 7 \le (c1 - 5)\sqrt{n}$$

By setting c1 = 12 and c2 = 1, the above holds. This concludes the proof.

Problem 5 [5 points] Which of the following function is not $O(n^{2.5})$ (B)

A. $53179546n^2$ B. $n^{2.7}/\log^2 n$ C. $\frac{n^{100}}{2^n}$ D. $(\log_2 n)^{101}$

Problem 6 [5 points] Which of the following functions is $O(n \log \sqrt{n})$ (C)

A. $(1.03)^n$ B. $n \cdot (\log_2 n)^{1.0001}$ C. $358 \cdot n\log_2 n$ D. $n^{1.2}/\log^5 n$

Problem 7 [5 points] Suppose you receive n numbers in stream (one by one). Once a number arrives, you need to sort it with all numbers you have received and then output the sorted list. Which sort algorithm you should use (A)

A. Insertion Sort B. Selection Sort C. Merge Sort D. Quick Sort

Problem 8 [8 points] Let S1 be an unsorted array of n integers, and S2 is another sorted array of $\log_2 n$ integers (n is a power of 2, **S2 is in ascending order**). Suppose Bo asked you to find the number of all pairs (x, y) satisfying x ∈ S1, y ∈ S2, and x > y. Please tell him the time complexity of your designed algorithm and explain why. (You do not need to write pseudocode!)

Idea: Since that S2 is a sorted array, for each integer in S1, we can perform binary search to find the index of its predecessor in S2. The sum of all those indexes is the number of all pairs satisfying x ∈ S1, y ∈ S2, and x > y.

Time complexity Analysis:

There are O(n) elements in S1, for each element, the binary search on S2 costs O(log log n) time as the size of S2 is O(log n). Find the sum of predecessors' indexes cost O(n) time. Thus, the total cost is O(n log log n) + O(n), which is O(n log log n).