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 [15 marks] Please analysis the time complexity of the following algorithm line by line.

```
Selection-Sort ( A[1..n] )
1. for integer i ← 1 to n−1
2.       k ← i
3.       for integer j ← i+1 to n
4.             if  A[k] > A[j]  then
5.                   k ← j
6.       swap A[i] and A[k]
```

Atomic operations per line: **(2 marks per line)**

Line 1: ______ $n-1$ ______

Line 2: ______ $n-1$ ______

Line 3: ______ $\sum_{i=1}^{n} n-i$ = (n−1)*n/2 ______

Line 4: ______ (n−1)*n/2 ______

Line 5: ______ (n−1)*n/2 ______

Line 6: ______ (n−1) ______

Total cost (in terms of Big-O notation) **T(*n*)** is: **(3 marks)**

______ = O(n) + O(n) + O(n^2) + O(n^2) + O(n^2) + O(n) = O(n^2) ______

Problem 2 [25 marks] 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 descending order**). Describe an algorithm to output the number of pairs (x, y) satisfying x ∈ S1, y ∈ S2, and x ≤ y. Your algorithm must terminate in $O(n \log \log n)$ time. For example, if S1 = {10, 7, 12, 18} and S2 = {15, 7}, then you should output 4 because 4 pairs satisfy the required conditions: (10, 15),(7, 15),(12, 15),(7, 7).

Solution:

Idea: (10 marks)

For every element x ∈ S1, perform binary search on S2 to find the number t_x of elements in S2 that are larger than or equal to x. Return $\sum_{x \in S1} t_X$.

Pseudocode: (10 marks)

```
Algorithm CountPairs(S1, S2)
    n ← len(S1)
    sum ← 0 // the total number of pairs
    for i ← 0 to n-1
        sum += findPairs(S1[i], S2)
    return sum
```

Algorithm findPairs(t, S2)

```
left ← 0, right ← len(S2)
repeat
    mid ← (left+right)/2
    if (t > S2[mid]) then
        right ← mid - 1
    else
        left ← mid + 1
until left > right
return right
```

Time complexity analysis: (5 marks)

There are O(n) elements in S1, for each element, the binary search on S2 costs O(log log n) time, thus, the total cost is therefore O(n log log n).

Problem 3 [30 marks] Design an algorithm to convert infix expression to postfix expression. (You can omit time complexity analysis in this problem).
Then, show the running steps of your algorithms for the following expression:
5 * ((9 + 3) * (4*2) + 7)

Solution

Idea and pseudocode: 20 marks
Idea
1. Read all the symbols one by one from left to right in the given Infix Expression, denote the reading symbol as **token**
2. If **token** is operand, then directly add it to Postfix string.
3. If **token** is left parenthesis '(', then Push it on to the Stack.
4. If **token** is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and add each poped token to Postfix.
5. If **token** is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and add them to the Postfix string.

Pseudocode:
```
Algorithm InfixtoPostfix( string Infix )
n ← len(S),
string Postfix ← empty string,
operator stack ops ← empty stack
for i ← 0 to n-1
    token ← Infix[i]
    if (token is operand) then
        Postfix += token
    else
        if (ops is empty) then
            ops.push(token)
        else if (token is '(')
            ops.push(token)
        else if (token is ')')
            topS ← ops.top()
            repeat
                add to topS to Postfix string
                ops.pop()
                topS ← ops.top()
            until  (topS is '(')
        else
            topS ← ops.top()
            while (topS precedence over token)
                add to topS to Postfix string
                ops.pop()
                topS ← ops.top()
            ops.push(token)
```

```
while (ops is not empty)
     topS ← ops.top()
     add to topS to Postfix string
     ops.pop()
return Postfix
```

Example: (10 marks)

5 * ((9 + 3) * (4*2) + 7)

Step	Token	Stack	Postfix
1	5		5
2	*	*	5
3	(	* (	5 9
4	(	* ((	5 9
6	+	* ((+	5 9
7	3	* ((+	5 9 3
8	)	* (	5 9 3 +
9	*	* (*	5 9 3 +
10	(	* (* (	5 9 3 +
11	4	* (* (	5 9 3 + 4
12	*	* (* (*	5 9 3 + 4
13	2	* (* (*	5 9 3 + 4 2
14	)	* (*	5 9 3 + 4 2 *
15	+	* (+	5 9 3 + 4 2 * *
16	7	* (+	5 9 3 + 4 2 * *
17	)	*	5 9 3 + 4 2 * * +
18			5 9 3 + 4 2 * * + *

Problem 4 [30 marks] Design a function to check if a linked list is a palindrome. For example:

Linked list **A**: 1->2->3 is not a palindrome, return No.

Linked list **B**: 1->2->3->2->1 is a palindrome, return Yes.

Solution

Idea: (15 marks)

We want to detect linked lists where the front half of the list is the reverse of the second half. How would we do that? By reversing the front half of the list. A stack can accomplish this.

We need to push the first half of the elements onto a stack. Since we do not know the length of the linked list, we can do like this way: we use a slow runner (go one step per iteration) and a fast runner (go two step per iteration). At each step in the loop, we push the data from the slow runner onto a stack. When the fast runner hits the end of the list, the slow runner will have reached the middle of the linked list. By this point, the stack will have all the elements from the front of the linked list, but in reverse order.

Pseudocode (10 marks)

```
Algorithm isPalindrome(LinkedListNode head) {
LinkedListNode fast ← head;
LinkedListNode slow ← head;
Stack s ← empty stack;
while (fast != null && fast->next != null) {
     s.push(slow->data);
     slow ← slow->next;
     fast ← fast->next->next;
}
if (fast != null) { //Has odd number of elements, so skip the middle element
     slow ← slow->next;
}
while (slow != null) {
     top = s.pop()->data;
     if (top != slow.data) {
               return false;
     }
     slow ← slow->next;
}
return true
```

Time complexity analysis: (5 marks)

Since slow runner only pass the linked list once, thus the time complexity is **O(n)**, where n is the length of given liked list.