COL106: Data Structures, I semester, 2016-17
Minor I
1 PM to 2 PM, 29th August 2016.

Attendance Sheet Serial Number: 186

Question	1	2	3	4	Total
	(5 marks)	(5+2 marks)	(5 marks)	(5 marks)	(20+2 marks)
Marks	4	0+0	2.5	0	6.5

1. Write your answers on the **printed question paper** in the space provided. **ROUGH SHEETS WILL NOT BE COLLECTED.**
2. Please write your name on every page and enter your serial number in the box above. **If you miss out any of these: -1 and no rechecking.**
3. **No pseudocode** means: For every loop you use, you must explain what it will do to the input. You cannot write things like "x x + y", you must explain the significance of every step in words. You cannot write "for i = 1 to ..." or "while <condition>...", you must *explain* what the loop achieves. In summary: if we need to interpret how your description will treat a particular input then it is pseudocode.

Q1. (Trees. marks = 5) Marks: 4

Suppose we are given a java implementation which has a `Tree` class and a `Node` class. The `Tree` class stores integer values as data at the nodes and has the following methods available:

- `public Node root()`: This returns the root of the tree.
- `public Boolean isEmpty()`: This returns 1 if the tree is empty, 0 otherwise.

The `Node` class has the following methods available:

- `public int data()`: This returns the integer data value stored in the node.
- `public int no_children()`: This returns the number of children of the node.
- `public Node child(int i)`: This returns the `Node` of the ith child, returning `null` if there are less than i children.
- `public Tree subtree(int i)`: This returns the `Tree` rooted at the ith child, returning an empty tree if there are less than i children.

Consider the following function:

```
public int f1 (Tree T) {
    if (T.isEmpty()) {return 0;}
    else {
       Node curr = T.root();
       int temp = curr.data();
       if (temp < 0) { temp = 0;}
       int i = 0;
       temp2 = 0;
       while (i < curr.no_children()) {
          temp2 = f1(curr.subtree(i));
          if temp2 > temp then {temp = temp2;}
          i++;
       }
       return temp;
    }
}

public int f2 (Tree T) {
     if (T.isEmpty()) { return 1;}
     else {
        Node curr = T.root();
```

```
        int i = 0;
        int flag = 1;
        while (i < curr.no_children() && (flag > 0)) {
           temp = curr.subtree(i);
           if (curr.data() > temp.root().data()) {flag = 0;}
             else {flag = flag*f2(temp);}
           i++;
          }
          return flag;
        }
}
```

Q1.1. (2 marks): Given a tree T explain in words what the function f1(T) outputs? Your answer must be as precise as possible.

f1(T) outputs the maximum positive data stored in all nodes of T

f1(T) returns 0 if all nodes data is negative

(1)

Q1.2. (3 marks): Given a tree T explain in words what the function f2(T) outputs? Your answer must be as precise as possible.

f2(T) outputs 1 if value in each node is greater than value of parent node, else it returns 0.

(3)

Q2. (Amortised analysis. Total marks = 5 + 2 extra credit). Marks: 0 + 0

A *n-counter* is a an array of $\log n$ bits that can store the binary representation of any number from 0 to $n-1$. Typically such n-counters are used to store values from 0 to $n-1$ in sequence. i.e., they are initialised with all 0s and at every step the value stored is incremented by 1, i.e., if at any time the value stored is i, it is incremented by 1 to $i+1 \mod n$. To increment a binary number stored in an n-counter, A which is an array with indexes 0 to $\log n - 1$, we perform the following algorithm:

- Set $i = \log n - 1$
- While ($A[i]$ is not 0) and ($i \geq 0$)
  - Set $A[i]$ to 0
  - $i = i - 1$
- if $i \geq 0$ then set $A[i]$ to 1.

This algorithm starts from the last entry and moves left and flipping all the 1s it encounters until it encounters the first 0, which it changes to 1. If it reaches all the way to the beginning of the array without encountering a 0 (which means that the counter contains all 1s) it does nothing, which is correct since when we increment $n-1$ we get n which is $0 \mod n$.

We will now analyse the running time of this algorithm. We will neglect all operations apart from array write operations, i.e. we will count only the number of times the algorithm flips a bit.

Q2.1. (1 mark) What is the worst case time of any increment operation in terms of bit flips? On the basis of this answer, what is the worst case running time of the the algorithm applied n times starting from 0 and counting up $n-1$ and then back to 0?

O(n) is worst case time for ~~~~ of bit flips

~~O(n) is worst case time~~

Q2.2. (4 marks) Prove that the total time taken to count from 0 back to 0 (i.e. 0 to $n-1$ and then, with one increment, back to 0) is $O(n)$. You may use an argument similar to that for either emulating a queue using two stacks (the argument uses chips) or for growable stacks (averaging argument)

Q2.3* Extra credit. Only attempt if you have time left (2 marks) An alternative way to prove the result of Q2.2 is to consider the time taken to increment a random number and then average it over all possible numbers. A way to generate a random binary number between 0 and $n-1$ is to set each bit to 0 with probability 1/2 and 1 with probability 1/2. Use this method of generating a random binary number to prove that the time taken to count from 0 back to 0 is $O(n)$ on average.

Q3. (Time Complexity and O notation: Total marks = 5) [illegible]

Suppose we have some already defined [illegible] the following [illegible] of the function f(n)

```
void f (int n){
   c = 1;
   for i = 1 to n {
      print g(i);
      if (i == c)
        then {
            print h(i);
            c = 2*c;
        }
   }
}
```

Q3.1. (2 marks) Calculate the *exact* number of time units and time complexity of **f(n)**, if **g(i)** takes $\log i$ units of time and **h(i)** takes i units of time. Write the time complexity in simplified big-Oh notation, e.g. if the number of time units turns out to be $3n^3 + 2\log n$, you must give the final answer as $O(n^3)$. Assume that each assignment statement (e.g. **c = 1**) takes 1 unit of time, each time the **for** statement is executed it takes 2 units of time, each time a print statement is executed it takes 1 unit of time, each multiplication takes 1 unit of time, and each comparison operation (**i == c**) takes 1 unit of time.

No. of units $= 1 + 2n + n + \sum_{i=1}^{n} \log i + n + \sum_{j=1}^{\lfloor \log n \rfloor} 2^j + \lfloor \log n \rfloor + \lfloor \log n \rfloor$

(assignment, for, print, g(i), (i == c), h(i), print, multiplication)

Since, $\sum_{i=1}^{n} \log i$ is $O(n \log n)$ and $\sum_{j=1}^{\lfloor \log n \rfloor} 2^j$ is $O(n)$,

therefore the above expression is $O(n \log n)$ ✓

(2)

Q3.2. (3 marks) Now consider the case where we run this program locally on a machine where all local operations are free but [illegible] to send **h(i)** and **g(i)** to be computed on a machine that charges money to compute them. The machine is available in two configurations. In configuration C_1, execution of **g(i)** costs Rs $1/i$ and **h(i)** costs Rs i. In configuration C_2, **g(i)** costs Rs 1 per execution and **h(i)** also costs Rs 1 per execution. *Asymptotically* which of the two configurations is cheaper in rupee terms? Argue your answer. If you just give the answer without any *mathematical* argument, you will receive a 0.

~~let it~~ For C_1

$f(n)$: Cost 1 $= \sum_{i=1}^{n} \frac{1}{i} + \sum_{j=1}^{\lfloor \log n \rfloor} 2^j = O(n^2)$

$O(n)$ $\log n = 1 + \frac{1}{2} + \frac{1}{3} + \dots$ — $O(n^3)$

(1/2)

$g(n)$: Cost 2 $= n + ~~(1+2 \dots + \lfloor \log n \rfloor)~~ + n \lfloor \log n \rfloor$ — $O(n)$

$= O(n^2)$

$f(n) > g(n)$

$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n^2}{n^3}$ ~~log~~ $\frac{g(n)}{f(n)} < \log \frac{n^2}{n^3}$ [Using definition of little o, $g(n) \in o(f(n))$]

Hence, ~~cost for~~ C_2 → C_2 is cheaper

Scanned by CamScanner

Q4. (Sorting with stacks) (5 marks) You are given two stacks S_1 and S_2 and a queue Q. There are n numbers stored in the queue. You have to sort the numbers in ascending order and put them back in Q in sorted order. You can use S_1 and S_2 for this purpose. You may use no other extra storage but you can use operations like *top* to look at the top of a stack or *front* to look at the front of the queue. Describe in English how to sort the elements of Q in ascending order using S_1 and S_2. **No pseudocode. Write your answer without using any variable names except S_1, S_2 and Q. If you write pseudocode you get an automatic 0 without your answer being read at all.**

Marks | 0

1) ~~Dequeue~~ ~~Avail~~

~~If front~~ · Push() front() to S_1