

Final Exam Practice Problems

CS1027A

University of Western Ontario

1. Compute the time complexity of the given code. You **must explain how you computed the time complexity** and you must give the order (big-Oh) of the time complexity.

```
1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j++) {
3          if (i % 2 == 0) {
4              System.out.println("Hi");
5          }
6      }
7  }
```

The `System.out.println()` call represents a single primitive operation ($O(1)$). The if statement from lines 3-5 occurs every other iterations of the outer loop; therefore, the outer loop runs $\frac{n}{2}$ times ($O(n)$). The inner loop runs $O(n)$ times. Therefore, the time complexity of this code fragment is $O(n^2)$.

2. Write **exactly two** for loops that are nested inside of each other such that your code fragment has a time complexity of $O(n \cdot \log(n))$.

```
1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j*=2) {
3          System.out.println("Hi");
4      }
5  }
```

3. Draw the binary tree from the given level-order traversal. We have not included "null" for every empty position in every level of the tree; instead, we have written null when an **existing** node has no child in that position.

- A, B, C, null, D, null, null, null, F

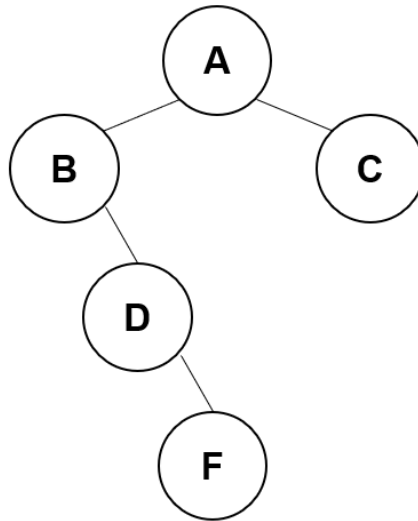



Figure 1

4. Draw the binary tree such that:

- a pre-order traversal visits the nodes in this order: 5, 1, 4, 2, 3, and
- an in-order traversal visits the nodes in this order: 1, 5, 2, 4, 3.

pre-order:


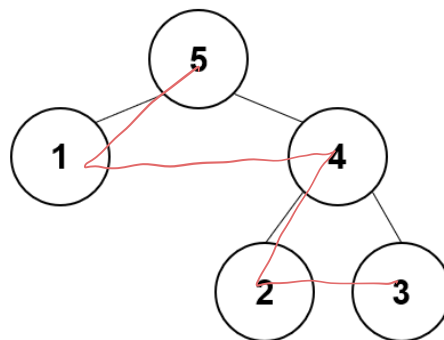


Figure 2

5. Consider the binary tree shown in Figure 3. Perform the iterative pre-order tree traversal using a stack as shown to you in class. Write your answers in the U-shaped diagrams in Figure 6. For the first stack, show the value on the stack before entering the loop. For the rest of the stacks, show the values on the stack at the end of each loop iteration.

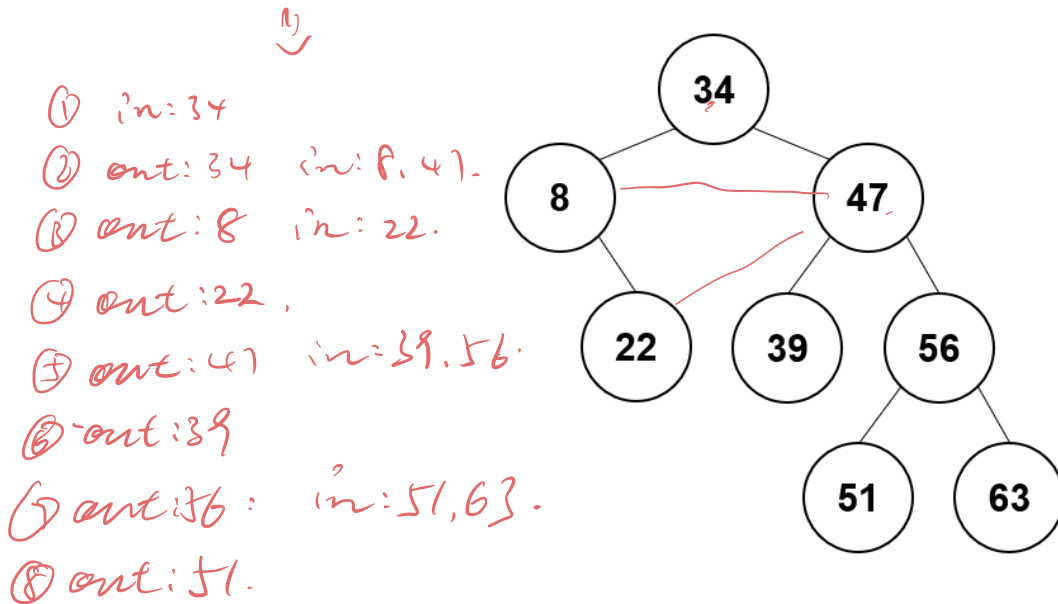


Figure 3

Visited nodes: **34 8 22 47 39 56 51 63** _ _ _

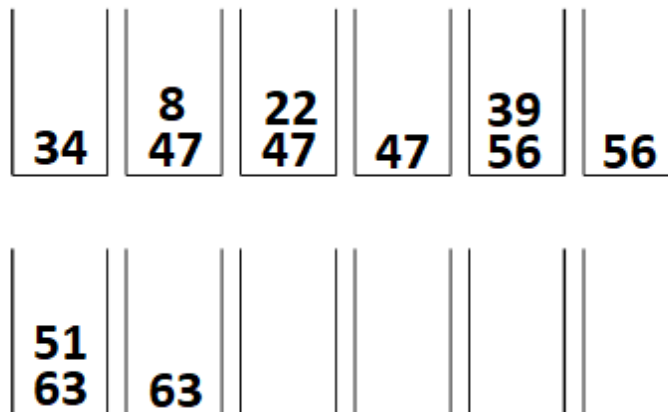


Figure 4

6. Consider the array $A = [1, 2, 3, 4]$. Trace through a stack-based implementation of insertion sort on this array. During each step, draw 3 stacks: (1) the *sorted* stack with the new number in it, (2) the *temp* stack holding whatever numbers are necessary to have drawn the previous *sorted* stack, and (3) the *sorted* stack with everything from *temp* in it.

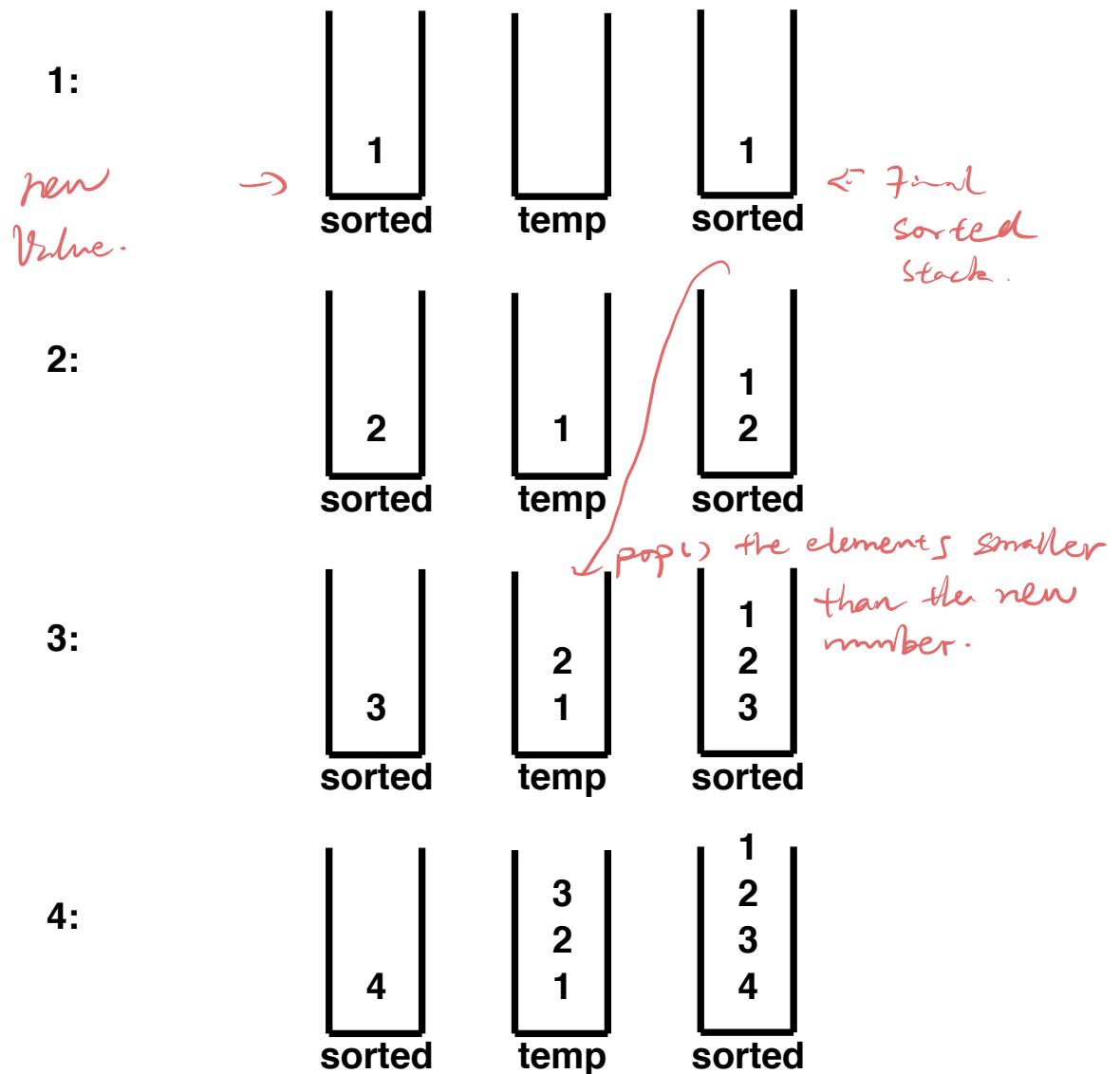


Figure 5

7. Consider the array $A = [5, 4, 3, 2, 1]$. Trace through the quicksort algorithm on this array. Use the **middle (rounded down)** element as the pivot. For each recursive call, draw the execution stack (don't worry about other methods such as *main()*), show the pivot, and show the arrays smaller, equal, and larger. Show the sorted sub-arrays at the appropriate times in the recursion.

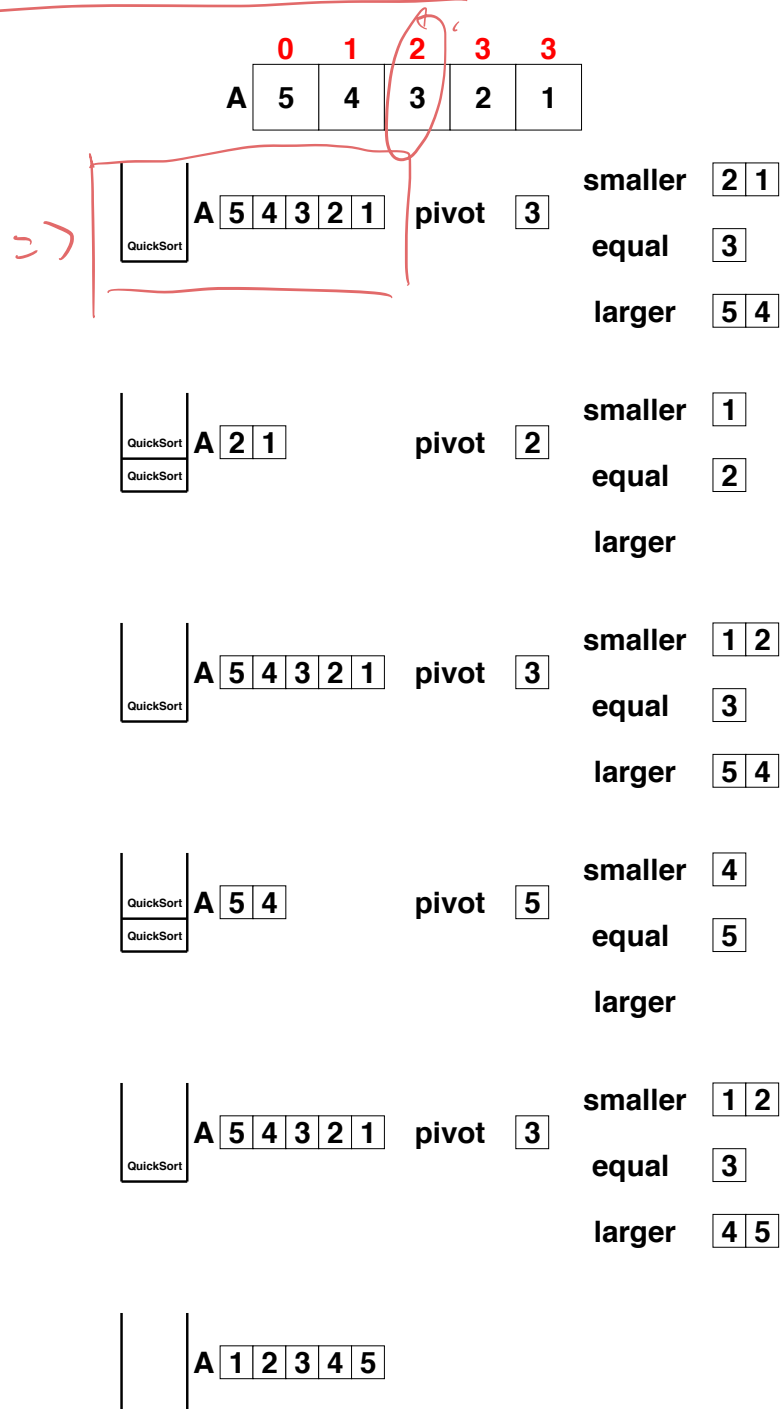


Figure 6

$\boxed{8/2}$

$\Rightarrow 39$

$2\frac{1}{2} \rightarrow 4\frac{1}{2} \rightarrow 6\frac{1}{2}$

39x2
= 780

=> 50

low 抽 (2.5)

8. Consider a binary search tree formed by linking together node objects of class *BinaryTreeNode*. The *BinaryTreeNode* class provides methods *getLeft()*, *setLeft()*, *getRight()*, *setRight()*, and *getElement()*. You can create a new node by calling the constructor of this class using *BinaryTreeNode<>(element)* to create a new node storing value *element* whose left and right children are *null*.

Write an algorithm *public BinaryTreeNode<T> find(BinaryTreeNode<T> node, T element)* in Java or in **detailed** Java-like pseudocode that searches the binary search tree for the node containing *element*. If *element* is in the tree, you must return the *BinaryTreeNode<T>* containing *element*; otherwise, you must throw a *NonExistentKeyException* (you can assume this exception has already been created).

You can assume that the variable *element* is of a class that implements *Comparable<T>*.

```
1 public BinaryTreeNode<T> find(BinaryTreeNode<T> node, T element) {
2     Comparable<T> e = (Comparable) element;
3     if (node == null)
4         throw new NonExistentKeyException();
5     else if (e.compareTo(node.getElement()) == 0)
6         return node;
7     else if (e.compareTo(node.getElement()) < 0)
8         return find(node.getLeft(), element);
9     else
10        return find(node.getRight(), element);
11 }
```

9. A binary tree is *symmetric* if for every internal node the number of nodes in its left subtree is the same as the number of nodes in its right subtree. Given a node p , let $p.size()$ return the number of nodes in the subtree with root p , and $p.getLeftChild()$ and $p.getRightChild()$ return the left and right children of p , respectively. Write a recursive algorithm `public boolean isSymmetric(BinaryTreeNode<T> r)` that receives as parameter the root r of a binary tree and it returns *true* if the tree is symmetric and it returns *false* otherwise. For example for the tree below with root r the algorithm must return *true*, but for the tree with root s it must return *false*.

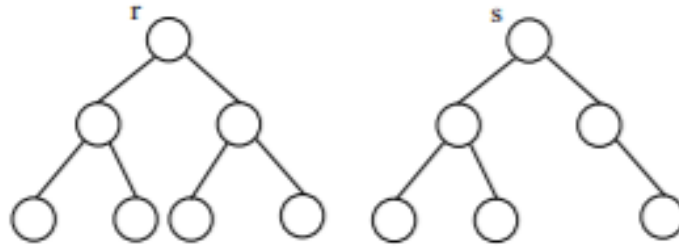


Figure 7

```

1 public boolean isSymmetrical(BinaryTreeNode<T> node) {
2     if (node == null)
3         return true;
4     else if (node.getLeft() == null && node.getRight() == null)
5         return true;
6     else if (node.getLeft() == null || node.getRight() == null)
7         return false;
8     else if (node.getLeft().size() != node.getRight().size())
9         return false;
10    else if (!isSymetric(node.getLeft()))
11        return false;
12    else
13        return isSymmetric(node.getRight());
14 }

```