**CMSC 451 Homework 3**

1. Shown below is the code for the insertion sort consisting of two recursive methods that replace the two nested loops that would be used in its iterative counterpart:

void insertionSort(int array[]) {

insert(array, 1);

}

void insert(int[] array, int i) {

if (i < array.length) {

int value = array[i];

int j = shift(array, value, i);

array[j] = value;

insert(array, i + 1);

}

}

int shift(int[] array, int value, int i) {

int insert = i;

if (i > 0 && array[i - 1] > value) {

array[i] = array[i - 1];

insert = shift(array, value, i - 1);

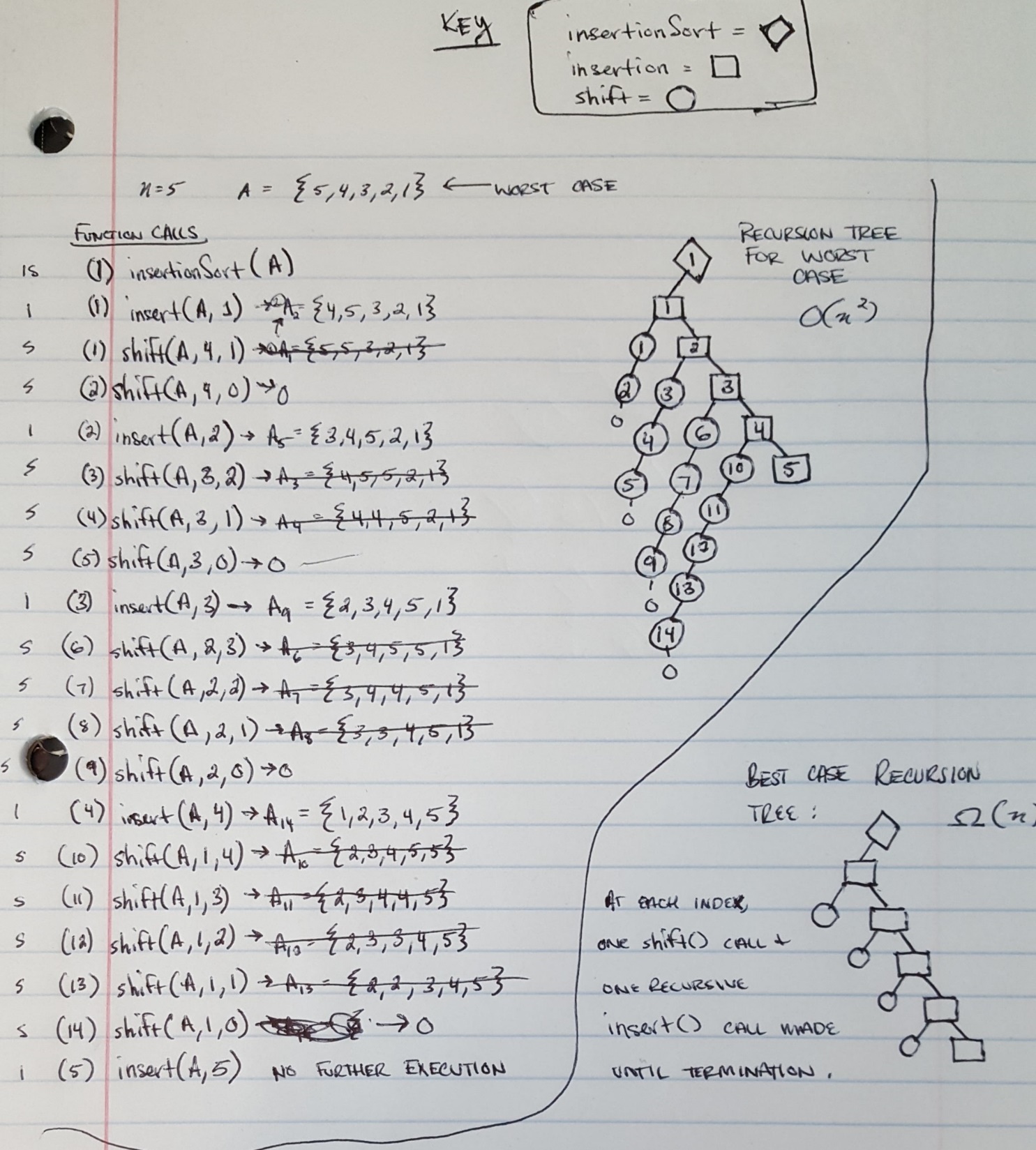
}

return insert;

}

Draw the recursion tree for insertionSort when it is called for an array of length 5 with data that represents the worst case. Show the activations of insertionSort insert and shift in the tree. Explain how the recursion tree would be different in the best case.

**Picture of trees below. Best case makes same amount of *insert()* recursive calls, but only one *shift()* call per *insert()* call as opposed to worst case which has a growing number of shift calls each step down the tree.**



2. Refer back to the recursion tree you provided in the previous problem. Determine a formula that counts the numbers of nodes in that tree. What is Big-Θ for execution time? Determine a formula that expresses the height of the tree. What is the Big-Θ for memory?

* Formula for number of nodes in tree

Base cases: T(0) = 2 nodes

1 for insertionSort and 1 for it’s call to insert

insertionSort = 1 node

insert = n nodes

switch = ((n(n+1)/2) -1)

switch could contribute anywhere from n-1 to ((n(n+1)/2) -1) nodes

number of nodes for switch in worst case is:

(∑ni=2 i) = (n(n+1)/2) – 1 = (n2 + n)/2 – 1

(number of nodes for switch in best case is: n-1)

since switch gets called once per insert except the last insert call

putting it all together we have the following formula:

T(n) = 1 + n + (n2 + n)/2 – 1

= n + (n2 + n)/2

= (n2 + n + 2n)/2

= **(n2 + 3n) / 2**

And for best case… 1 + n + n – 1 = 2n

So the number of nodes in a tree for insertionSort will be**:** 2n <= T(n) <= (n2 + 3n) / 2

* Formula for height of tree

Base case: h(0) = 1

This is assuming that the root or the level for insertionSort call is 0.

**h(n) = 2n – 1** for worst case since height of tree will always be 1 less double input size

h(n) = n – 1 for best case

And if root is level 1, then worst case is 2n and best case is n.

So the height of the tree for insertionSort will be in the range n-1 to 2n-1 if node is level 0

* Big-Θ for execution time: **Θ(n2) since T(n) = (n2+3n)/2 and n2 is the most significant**
* Big-Θ for memory: **Θ(n) since h(n) = 2n – 1**

3. Provide a generic Java class named SortedPriorityQueue that implements a priority queue using a sorted list implemented with the Java ArrayList class. Make the implementation as efficient as possible.

(Java file separately attached for convenience as well)

import java.util.ArrayList;

/\*\*

\* SortedPriorityQueue

\*

\* Implements a priority queue

\* uses a sorted list implemented with the Java ArrayList class

\* Adds elements in sorted DESC order

\* So, max elements are priority

\*

\* @author matthew.towles

\* @param <T>

\* @date Jun 6, 2019

\*/

public class SortedPriorityQueue<T extends Comparable> {

/\*\*

\* Sorted ArrayList

\*/

private ArrayList<T> queue = new ArrayList<>();

/\*\*

\* Adds a new element to queue

\* Maintains sorted property of queue

\*

\* @param el

\*/

void add(T el) {

if (this.isEmpty()) {

queue.add(el);

} else {

// save element in temp variable

T tmp = el;

// keep track if we added el or not

boolean added = false;

// look for next element that is larger or same size as el

for (int i = 0; i < queue.size(); i++) {

// if larger/same size found, set to el

if (queue.get(i).toString().compareTo(el.toString()) < 0) {

// save former value to shift up

tmp = queue.get(i);

// actually set slot to el

queue.set(i, el);

// set added flag to true since el added

added = true;

break;

}

}

// el not added, just add to the end

if (!added) {

queue.add(el);

} else {

// recursive call to keep shifting as necessary

this.add(tmp);

}

}

}

/\*\*

\* Remove min el from queue

\* Min el will be first el since queue is sorted

\*

\* @return removed element

\*/

T removeMax() {

T el = queue.remove(0);

return el;

}

/\*\*

\* Look at next element without doing anything to it

\*

\* @return next element in queue

\*/

T findMax() {

return queue.get(0);

}

/\*\*

\* True for empty, false for not empty

\* @return true | false

\*/

boolean isEmpty() {

return queue.isEmpty();

}

/\*\*

\* Main method

\*

\* @param args

\*/

public static void main(String[] args) {

int[] arr = {6,4,9,2,1};

SortedPriorityQueue<Integer> spq = new SortedPriorityQueue();

System.out.println("- - - - - - - -");

System.out.println("Unsorted array contents:");

System.out.println("- - - - - - - -");

for (int i = 0; i < arr.length; i++) {

spq.add(arr[i]);

System.out.println("added: " + arr[i]);

}

System.out.println("- - - - - - - -");

System.out.println("Sorted Order:");

System.out.println("- - - - - - - -");

int n = spq.queue.size();

for (int i = 0; i < n; i++) {

System.out.println("removed: " + spq.removeMax());

}

}

}

4. Consider the following sorting algorithm that uses the class you wrote in the previous problem:

void sort(int[] array) {

SortedPriorityQueue<Integer> queue = new SortedPriorityQueue();

for (int i = 0; i < array.length; i++)

queue.add(array[i]);

for (int i = 0; i < array.length; i++)

array[i] = queue.remove();

}

Analyze its execution time efficiency in the worst case. In your analysis you may ignore

the possibility that the array list may overflow and need to be copied to a larger array.

Indicate whether this implementation is more or less efficient than the one that uses the

Java priority queue.

**Worst case is an array in reverse sorted order. In the case of my implementation, this is an array, which is sorted, in ascending order.**

**In *sort*(), two methods are called: *add*() and *remove*().**

**Let’s start by analyzing the performance for *remove*() (*removeMax*() in my implementation):**

**Each call to *remove*() simply removes the first element in the queue and returns it.**

***remove*() ∈ O(1)**

**Now let’s take a look at *add*():**

**If the array size, *n* <= 1, then add simply checks and sees that the SortedPriorityQueue, *q* , is empty and adds the element to *q*. So we know if *n* <= 1, *add*() ∈ O(1).**

**When n > 1:**

* **+1 for tmp variable {equation so far: 1 }**
* **+1 for added variable {1+1}**
* **for loop executes one time only in worst case since the first iteration will run into an element smaller than the element being added. {1+1+()}**
  + **if statement in for loop has +2 comparisons and +1 get method {1+1+(3)}**
  + **+1 tmp variable set to queue.get(i) +1 {1+1+(3+1)}**
  + **queue.set +1 {1+1+(3+1+1)}**
  + **added variable +1 {1+1+(3+1+1+1)} = {8 + \_\_}**
* **in worst case, added will be true each call, so we will execute the final else which includes a recursive call to *add*()**
* **Let *m* represent the size of the *q:***
  + **each recursive call will add 8 more of the above executions**
  + **number of recursive calls is function of size of *q*, *m***
  + **so if an element is added and there are 5 elements already in the *q* then there will be 5 additional calls to *add*()**
  + **To make this easier to illustrate, on the last iterative call to *add*() in *sort*(), *add*() will call itself *n*-1 + the one time it is called in *sort*() = n times**
  + **Each successive recursive call to *add*() will result in the for loop to execute one more time than the previous, eventually up to the *q* size, which will be *n* times by the final recursive call, so *add*() ∈O(n2)**
  + **We will also have to call *add* *n* times in *sort*, so we will be calling an O(n2) \* n**
  + **So effectively, to add a sorted array into this SortedPriorityQueue, it will have a worst case running time ∈ O(n3) – *remove* and the other calls are insignificant**
* **remove() ∈ O(1)**
* **add() ∈ O(n2)**
* **sort() calls each n times**
* **sort() = n \* O(1) + n \* O(n2)**
* **= n + n3**
* **sort() ∈ O(n3)**

**This is less efficient than the implementation that uses java priority queue.**