1 Question 1

For this question, I utilized the itertools library in python with the permutations method which returns all the permutations of a list of numbers. I then got the strings for those permutations and iterated through them to make an arraylist of permutation arrays for each type of permutation. I then created arrays for the number of swaps for Insertion sort and Quicksort and compared them. If the value at index i for insertion sort was greater than the value at index i for Quicksort, I incremented the Insertion sort work counter. Then took the number of times Insertion sort beat Quicksort and divided by the total number of permutations for a given list. While not the most simple way to do things, this seemed to give me reasonable answers. (Note: I had to split the perm7 string into two strings because it was too long for a single string. Did not show in pics) Here is my code:

```
print(list(permutations([1, 7, 9])))
print(list(permutations([2,5,7,8,9])))
print(list(permutations([1,2,4,5,6,8,9])))
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Permutations {
    static int swapsIS = 0;
    static int swapsQS = 0;

    public static void main(String[] args) {
        String perm3 = "(1, 7, 9), (1, 9, 7),
        String perm5 = "(2, 5, 7, 8, 9), (2, 5
 8, 5), (2, 8, 5, 7, 9), (2, 8, 5, 9, 7), (2,
 8, 9), (5, 2, 7, 9, 8), (5, 2, 8, 7, 9), (5,
 9, 7), (5, 8, 7, 2, 9), (5, 8, 7, 9, 2), (5,
 5, 9), (7, 2, 8, 9, 5), (7, 2, 9, 5, 8), (7,
 9, 2), (7, 8, 9, 2, 5), (7, 8, 9, 5, 2), (7,
 5, 7), (8, 2, 9, 7, 5), (8, 5, 2, 7, 9), (8,
 5, 2), (8, 9, 2, 5, 7), (8, 9, 2, 7, 5), (8,
 7, 8), (9, 5, 2, 8, 7), (9, 5, 7, 2, 8), (9,
 7, 5), (9, 8, 5, 2, 7), (9, 8, 5, 7, 2), (9,
        String perm7_1 = "(1, 2, 4, 5, 6, 8, 9
 8, 5, 9), (1, 2, 4, 6, 8, 9, 5), (1, 2, 4, 6,
```

```
//create lists of arrays for each permutation for quicksort
List<int[]> p3_IS = permList(perm3,  size: 3);
List<int[]> p3_QS = permList(perm3,  size: 3);

List<int[]> p5_IS = permList(perm5,  size: 5);
List<int[]> p5_QS = permList(perm5,  size: 5);

//must create two separate arrays (because string is too long)
List<int[]> p7_1 = permList(perm7_1,  size: 7);
List<int[]> p7_2 = permList(perm7_2,  size: 7);

List<int[]> p7_12 = permList(perm7_1,  size: 7);
List<int[]> p7_22 = permList(perm7_2,  size: 7);

//merge two separate arrays into one list
List<int[]> p7_IS = new ArrayList<>();
List<int[]> p7_QS = new ArrayList<>();
p7_IS.addAll(p7_1);
p7_IS.addAll(p7_2);
p7_QS.addAll(p7_12);
p7_QS.addAll(p7_22);

//size 3! == 6
int[] p3SwapsIS = new int[6];
int[] p3SwapsQS = new int[6];

for (int i = 0; i < p3_IS.size(); i++) {
    insertionSort(p3_IS.get(i));
    p3SwapsIS[i] = swapsIS;
    swapsIS = 0;

    quickSort(p3_QS.get(i),  p: 0,  r: 3);
    p3SwapsQS[i] = swapsQS;
    swapsQS = 0;
}

//check # of times IS beats QS
int IS_3 = 0;
for (int i = 0; i < p3SwapsIS.length; i++) {
    if (p3SwapsIS[i] > p3SwapsQS[i]) {
        IS_3++;
    }
}


//size 5! == 120
int[] p5SwapsIS = new int[120];
int[] p5SwapsQS = new int[120];

for (int i = 0; i < p5_IS.size(); i++) {
    insertionSort(p5_IS.get(i));
    p5SwapsIS[i] = swapsIS;
    swapsIS = 0;

    quickSort(p5_QS.get(i),  p: 0,  r: 5);
    p5SwapsQS[i] = swapsQS;
    swapsQS = 0;
}
```

```
//check # of times IS beats QS
int IS_5 = 0;
for (int i = 0; i < p5SwapsIS.length; i++) {
    if (p5SwapsIS[i] > p5SwapsQS[i]) {
        IS_5++;
    }
}

//size 7! == 120
int[] p7SwapsIS = new int[5040];
int[] p7SwapsQS = new int[5040];

for (int i = 0; i < p7_IS.size(); i++) {
    insertionSort(p7_IS.get(i));
    p7SwapsIS[i] = swapsIS;
    swapsIS = 0;

    quickSort(p7_QS.get(i),  p: 0,  r: 7);
    p7SwapsQS[i] = swapsQS;
    swapsQS = 0;
}

//check # of times IS beats QS
int IS_7 = 0;
for (int i = 0; i < p7SwapsIS.length; i++) {
    if (p7SwapsIS[i] > p7SwapsQS[i]) {
        IS_7++;
    }
}

//check % of times InsertionSort did worse than QuickSort
double ISwork3 = (double)IS_3/6 * 100;
double ISwork5 = (double)IS_5/120 * 100;
double ISwork7 = (double)IS_7/5040 * 100;

System.out.printf("3 integer permutation: Insertion sort did worse than QuickSort %.2f percent of the time\n", ISwork3);
System.out.printf("5 integer permutation: Insertion sort did worse than QuickSort %.2f percent of the time\n", ISwork5);
System.out.printf("7 integer permutation: Insertion sort did worse than QuickSort %.2f percent of the time\n", ISwork7);
```

```java
// generate list of arrays from formatted string
public static List<int[]> permList(String string, int size) {
    //create new list
    List<int[]> list = new ArrayList<>();

    //create new String array with one character per index
    String[] split = string.split( regex: "");

    //create new array
    int[] A = new int[size];
    int j = 0;

    for (int i = 0; i < split.length; i++) {
        //get String at index i
        String s = split[i];

        //if a new list is forming, reinitialize the array and j
        if (s.equals("(")) {
            A = new int[size];
            list.add(A);
            j = 0;
        } else if (!s.equals(",") && !s.equals(")") && !s.equals(" ")) { // if s is NaN, don't enter loop
            //turn string to an integer and add to list
            int p = Integer.parseInt(s);
            A[j] = p;
            j++;
        }
    }

    return list;
}

public static void insertionSort(int[] A) {

    for (int j = 2; j < A.length; j++) {
        int key = A[j];
        int i = j - 1;

        while (i > 0 && A[i] > key) {
            A[i + 1] = A[i];
            i -= 1;
            swapsIS++;
        }
        A[i+1] = key;
    }
}
```

```
    public static int partition(int[] A, int p, int r) {
        int x = A[r];
        int i = p - 1;

        for (int j = p; j < r; j++) {
            if (A[j] <= x) {
                i++;
                swap(A, i, j);
                swapsQS++;
            }
        }
        swap(A, i: i + 1, r);
        swapsQS++;
        return i + 1;
    }

    public static void quickSort(int[] A, int p, int r) {
        if (p < r-1) {
            int q = partition(A, p, r: r-1);
            quickSort(A, p, r: q - 1);
            quickSort(A, p: q + 1, r: r-1);
        }
    }

    public static void swap(int[] a, int i, int j) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
}
```

1.1 Output

The output for permutations of size 3, I got that 0.00% of Insertion sort invocations did more work than Quicksort.

For the permutations of size 5, I got that 29.17% of Insertion sort invocations did more work than Quicksort.

For the permutations of size 7, I got that 50.85% of Insertion sort invocations did more work than Quicksort.

From this trend, it appears that, on average, as the array length increases, the number of times Insertion sort does less work than Quicksort decreases.

2 Question 2

2.1 a.

The successor of x is the smallest element of x's right subtree in a BST. Let r be x's successor. If r has no right child, it will have 0 children because if it had a left child, it would not be the successor of x. If r has a right child, that child will be r's only child for the reason just stated. Therefore, x's successor can have either 0 or 1 children.

2.2 b.

The predecessor of x is the largest element in the left subtree of x. Let l be x's predecessor. Then, if l has no right child, it will have 0 children because a right child would imply that l is not x's successor. If l has a left child, it will still be the predecessor to x because any child in the left subtree of l will be strictly less than l. Therefore, l can have 0 or 1 children.

3 Question 3

3.1 a.

funA:
Let x be an arbitrary node in a BST. In every instance of comparing x's children to x, each invocation ensures that no left child be greater than $nodeVal(x)$ and no right child can be less than $nodeVal(x)$. This implies that funA is a correct algorithm.

funB:
I will give an example to show that the algorithm is incorrect. Suppose we take an arbitrary node x in the BST. Let l be the left child of of x and l_r be the right child of l. Given this, the algorithm will not check if $nodeVal(l_r) < nodeVal(x)$ because it is only comparing children nodes to their respective parent nodes. So, if $nodeVal(l) < nodeVal(x)$ but $nodeVal(l_r) > nodeVal(x)$, funB will still return true.

3.2 b.

funA:
At each check of a node we are using $tree-get-max()$ and $tree-get-min$ functions which have a runtime of $O(h)$. Additionally, since we are calling both functions for each node, we have a runtime of $2h * n$. Since h is a constant, we drop it to get a complexity of $\theta(hn)$. In the best case, where the BST properties do not hold, funA will have a complexity of $\Omega(h)$ because it will still call both $get - max/min$ functions in order to determine the tree is not a BST.

funB:
For each node in the BST, we are compare it to its parent, left child, and right child which, implies that each node is compared to 3 times. Since we compare every node in the tree, with a size of n, we have a runtime of $3n$. Since 3 is a constant we can drop it, which yields a complexity of $\theta(n)$.

4 Question 4

4.1 a.

From the book, we have that the expected upper bound for the number of probes of an unsuccessful search is $\frac{1}{1-\alpha}$. Using $\alpha = \frac{6}{7}$ we have $\frac{1}{1-\frac{6}{7}} = 7$ probes. This makes sense intuitively because we will have to search every element of the hash table to ensure that the key we are looking for is not in the table.

4.2 b.

Also from the book, we have that the expected upper bound for the number of probes of a successful search is $\frac{1}{\alpha}ln(\frac{1}{1-\alpha})$. Using $\alpha = \frac{6}{7}$ we have $\frac{1}{\frac{6}{7}}ln(\frac{1}{1-\frac{6}{7}}) = \frac{7}{6}ln(7) = 2.27$ probes. This makes sense because if we compare this value to part (a), the expected number of probes for a successful search should be less than the number of probes for an unsuccessful search.

5 Question 5

From (4), we know the equations of the expected number of probes. For this problem, we can set the equation for the unsuccessful search equal to 3.5 times the equation for the successful search. This gives $\frac{1}{1-f} = \frac{3.5}{f}ln(\frac{1}{1-f})$. We can then graph these two equations and see where they intersect. When graphed, both lines intersect when $x = 0.882$. Thus, a load factor f of 0.882 will ensure that the expected number of probes in an unsuccessful search is 3.5 times the expected number of probes in a successful search.