1 Using Big-O Notation:

Find c_0, n_0 such that $\exists f(n) \leq c_0 * g(n); (\forall n > n_0)$, if $f(n) = log_2(9n)$ and $g(n) = O(log_2 n)$.

USING SUBSTITUTION, if $n_0 = 3$ and $c_0 = 3$,

$$log_2(9n) \leq c_0(log_2\ n); (\forall n > n_0)$$

$$log_2(9(3)) \leq 3(log_2\ 3)$$

$$3.29 \leq 3.29$$

...which is true.

THEREFORE,

$$n_0 = 3, c_0 = 3$$

2 By Growth Rate, Order the Following:

$$n,\ n^2,\ n\ log\ n,\ \frac{2}{n},\ 2^n,\ 92,\ n^2\ log\ n,\ n!,\ n^{1.5},\ n^3,\ log\ n,\ n\ log(n^2),\ 4^{log\ n}$$

PROVING by substitution...

	n	n^2	$n\ log\ n$	$\frac{2}{n}$	2^n	92	$n^2\ log\ n$	$n!$	$n^{1.5}$	n^3	$log\ n$	$n\ log(n^2)$	$4^{log\ n}$
for n	2	4	1.38	1	4	92	2.77	2	2.82	8	.69	2.77	2.61
for n	3	9	3.29	.66	8	92	9.88	6	5.19	27	1.09	6.59	4.58
for n	5	25	8.04	.4	32	92	40.23	120	11.1	125	1.60	16.09	9.3
growth rate	.66	1.77	1.44	$-.39$	3	0	3.07	19	1.13	3.62	.46	1.44	1.03
rank	4	9	8	1	10	2	11	13	6	12	3	7	5

THEREFORE, the order of functions, from smallest to largest, is:

$$\frac{2}{n}, 92, log\ n, n, 4^{log\ n}, n^{1.5}, n\ log(n^2), n\ log\ n, n^2, 2^n, n^2\ log\ n, n^3, n!$$

...where $n\ log\ n$ and $n\ log\ (n^2)$ grow at the same rate, due to the nature of logarithms, because:

$$n\ log\ (n^2) = 2n\ log\ n$$

...which, in Big-O notation, is approximated without loss of generality to:

$$O(n) \in n\ log\ n$$

3 For the below, prove true or false:

ASSUMING that...

$$T_1(N) = O(f(N))$$

and

$$T_2(N) = O(f(N))$$

3.1 a)

BY THE DEFINITION of Big-O, we know that $\exists f(n) \leq c_0 * g(n); (\forall n > n_0)$. THEREFORE we know that there are two numbers c such that...

$$T_1(N) \leq c_0 * f(n)$$

...and...

$$T_2(N) \leq c_1 * f(n)$$

...and, thus, by extension:

$$T_1(N) + T_2(N) \leq (c_0 * g(n)) + c_1 * f(n))$$
$$\therefore T_1(N) + T_2(N) \leq (c_0 + c_1) * f(n)$$

THEREFORE, because ANY two numbers c will result in $T_1(N) + T_2(N) \leq f(n)$, and because $O(f(n))$ is the upper bound of that problem, without loss of generality, we can assume that...

$$T_1(N) + T_2(N) = O(f(N))$$

THEREFORE, this answer is TRUE.

3.2 b)

$$T_1(N) - T_2(N) = o(f(N))$$

...if, in the above, $T_1(N) = n^2$ and $T_2(N) = n^2$,

$$n^2 - n^2 = o(f(N))$$
$$0 \neq o(f(N))$$

THEREFORE, this is FALSE.

3.3 c)

$$\frac{T_1(N)}{T_2(N)} = O(1)$$

...if, in the above, $T_1(N) = n^3$ and $T_2(N) = n^2$,

$$\frac{n^3}{n^2} = O(1)$$
$$n \neq 0(1)$$

THEREFORE, this is FALSE.

3.4 d)

$$T_1(N) = O(T_2(N))$$

...if, in the above, $T_1(N) = n^3$ and $T_2(N) = n$,

$$O(n^3) \neq O(n)$$

THEREFORE, this is FALSE.

4 Give the Big-O for each of the below:

4.1 a)

```
sum = 0;
for (i = 1; i <= n; i++) {
   for (j = 1; j <= n; j++) {
      sum++;
      sum++;
   }
}
```

...for the above, the problem $f(n)$ is represented by the dual incrementing of *var* sum, nested inside two loops, for every value n. Considering the dual incrementing to be two primitive operations, we can assume that the Analytical Time Complexity of the function is:

$$f(n) = 2 * n * n$$

or,

$$f(n) = 2n^2$$

...which reduces down (for significantly large values of n, and without loss of generality) to:

$$f(n) \in O(n^2)$$

4.2 b)

```
sum = 0;
for (i = 1; i <= n; i++) {
   for (j = 1; j <= 3 * i; j++) {
      sum++;
   }
   for (k = 1; k <= 100000; k++) {
      sum++
   }
}
```

...for the above, the problem $f(n)$ is represented by one primitive operation in the "j" loop, which is executed $3n$ times for every iteration of n, as well as one primitive operation in the "k" loop, which is executed 100,000 times for every iteration of n. Therefore, the Analytical Time Complexity of the function is:

$$f(n) = 3n + 100000$$

...FOR BIG-O PURPOSES, because 100000 in this context is a constant,

$$f(n) = O(3n)$$

HOWEVER, we can assume, without loss of generality, that the true $O(n)$ in this situation, for significantly large values of n, is:

$$f(n) \in O(n)$$

4.3 c)

```
sum = 0;
for (i = 1; i <= n; i++) {
   for (j = 1; j <= i * i; j++) {
      sum++;
```

```
   }
}
```

...for the above, the problem $f(n)$ is represented by one primitive operation which is in a nested "for" loop, whose (as in exercise a) outer loop has a cost of $n * n$, or n^2. However, there is an additional cost in the inner loop of executing $i * i$ times per iteration of i (NOT n). Since that loop will be instantiated roughly half of the occurrences of n, but execute i^2 times, the cost of that loop becomes $\frac{n^2}{2}$. Therefore, the Big-O of the function is:

$$n * \frac{n^2}{2}$$

$$\therefore \frac{n^3}{2}$$

...furthermore, because there exists a constant c which represents the primitive operation, it follows from our definition of Big-O that:

$$c * \frac{n^3}{2} = \frac{c}{2} * n^3$$

...and, because we can safely ignore the constant in Big-O notation, without loss of generality and for sufficiently large values of n, the Big-O for this function is:

$$f(n) \in O(n^3)$$

4.4 d)

```
sum = 0;
for (i = 1; i <= n; i++) {
   for (j = 1; j <= i * i; j ++) {
      if (j % i == 0) {
         for (k = 1; k <= j; k++) {
            sum++;
         }
      }
   }
}
```

...for the above, the problem $f(n)$ is represented by the exact scenario as in exercise c, however, the inner condition instantiates *another* for loop only when the j iteration is a multiple of i. Therefore, the Analytical Time Complexity is:

$$f(n) = \frac{n^3 + n^2}{2}$$

...which, using the same principle of constants as above, we can reduce to:

$$f(n) = \frac{1}{2}(n^2 + n^3)$$

...and, because for all sufficiently large values of n, the constant can be ignored in favor of the worst case:

$$f(n) \in O(n^3)$$

To further prove via substitution, we rely again on our definition of $O(g(n))$, which is:

$$\exists f(n) \leq c_0 * g(n); (\forall n > n_0)$$

USING SUBSTITUTION, we can presume that for $f(n) = \frac{n^3+n^2}{2}$, $g(n) = O(n^3)$ when $c_0 = 3$ and $n_0 = 5$, as below:

$$\frac{n^3 + n^2}{2} \leq c_0(n^3); (\forall n > n_0)$$

$$\therefore \frac{5^3 + 5^2}{2} \leq 3(5^3)$$

$$\therefore \frac{125 + 25}{2} \leq 375$$

$$\therefore 75 \leq 375$$

THEREFORE,

$$f(n) \in O(n^3)$$

5 Using Java, write a method:

...which takes in an array, in sorted order, and a value, and returns the smallest possible index of an element that is equal to or larger than the given value, or -1 if the value is larger than the max. The method must run in $O(log\ n)$ time.

5.1 Basic Test Code:

```
public static void main(String[] args) {
        int[] testArray = {1,2,3,3,3,4,5,5,14,17};

        System.out.println("The smallest index where 3 is located is: " +
            firstNonSmallerIndex(testArray, 3));
```

```
            System.out.println("The smallest index where 4 is located is: " +
                firstNonSmallerIndex(testArray, 4));
            System.out.println("The smallest index where -1 is located is: " +
                firstNonSmallerIndex(testArray, -1));
            System.out.println("The smallest index where 23 is located is: " +
                firstNonSmallerIndex(testArray, 23));
            System.out.println("The smallest index where 15 is located is: " +
                firstNonSmallerIndex(testArray, 15));
      }

      public static int firstNonSmallerIndex(int[] array, int value) {
            int valueMin = -1;
            for (int i = array.length - 1; i >= 0; i--) {
                  if (value <= array[i]) {
                        valueMin = i;
                  }
            }
            return valueMin;
      }
```

5.2 Expanded Code Showing Runtime Testing:

Below is the full code, including timing testing functions, and the result set.

```
package fistNonSmallerIndexPackage;

import java.util.Arrays;
import java.util.Random;
import java.util.concurrent.TimeUnit;

public class FirstNonSmallerIndex {
      public static void main(String[] args) {
            //the max size of the array
            final int maxSize = 10000000;

            //the interval by which to increase the array for each test loop
            final int intervalSize = 1000000;

            for (int j = 10; j <= maxSize; j+=intervalSize) {

                  //create a random sorted array
                  int[] testArray = createRandomSortedArray(j);
```

```
            //pick a random number from the above sorted array
            int testNum = getRandom(testArray);

            //find the starting time
            long startTime =
                TimeUnit.NANOSECONDS.toMillis(System.nanoTime());

            //invoke the reverse search method
            System.out.println("The smallest index where " + testNum + "
                (a random number) is located is: " +
                firstNonSmallerIndex(testArray, testNum));

            //find the end time.
            long endTime =
                TimeUnit.NANOSECONDS.toMillis(System.nanoTime());;

            //find the difference between start and end time
            long algTime = endTime - startTime;

            //evangelize the algorithm's run time.
            System.out.println("The above algorithm, searching through an
                array of size " + j + ", took " + algTime + " ms to
                compute.");
      }
}

//the below is my function to find the smallest
// element of an array containing value
public static int firstNonSmallerIndex(int[] array, int value) {
      int valueMin = -1;
      for (int i = array.length - 1; i >= 0; i--) {
            if (value <= array[i]) {
                  valueMin = i;
            }
      }
      return valueMin;
}

//the below function comes from the homework assignment
public static int[] createRandomSortedArray(int size) {
      Random rand = new Random();
      int[] array = new int[size];

      for (int i = 0; i < size; i++) {
            array[i] = rand.nextInt(size * 3) - size / 4;
      }
```

```
            Arrays.sort(array);

            return array;
        }

        //the below function is borrowed code to find a random number from an array
        public static int getRandom(int[] array) {
            int rnd = new Random().nextInt(array.length);
            return array[rnd];
        }
}
```

Finally, the result set:

```
The smallest index where 5 (a random number) is located is: 1
The above algorithm, searching through an array of size 10, took 1 ms to compute.
The smallest index where 2139703 (a random number) is located is: 796340
The above algorithm, searching through an array of size 1000010, took 2 ms to
    compute.
The smallest index where 1870371 (a random number) is located is: 789214
The above algorithm, searching through an array of size 2000010, took 2 ms to
    compute.
The smallest index where 5965356 (a random number) is located is: 2238854
The above algorithm, searching through an array of size 3000010, took 3 ms to
    compute.
The smallest index where 4966627 (a random number) is located is: 1988784
The above algorithm, searching through an array of size 4000010, took 4 ms to
    compute.
The smallest index where 7876902 (a random number) is located is: 3042430
The above algorithm, searching through an array of size 5000010, took 6 ms to
    compute.
The smallest index where 4248695 (a random number) is located is: 1916338
The above algorithm, searching through an array of size 6000010, took 4 ms to
    compute.
The smallest index where 15193318 (a random number) is located is: 5644799
The above algorithm, searching through an array of size 7000010, took 6 ms to
    compute.
The smallest index where 5054424 (a random number) is located is: 2352232
The above algorithm, searching through an array of size 8000010, took 7 ms to
    compute.
The smallest index where 8397384 (a random number) is located is: 3547717
The above algorithm, searching through an array of size 9000010, took 8 ms to
    compute.
```

5.3 Big-O Discussion

FOR THE ALGORITHM ABOVE, we see that, although the "for" loop of the code iterates down for every value of the array, the primitive operation (storing variable i) is only executed if the given value is less than or equal to the array value at index i. Combined with the instantiation of var "valueMin", that makes the Big-O:

$$f(n) = (log_2\ n) + 1$$

...or, reduced down for significantly large values of n, and without loss of generality:

$$f(n) \in O(log\ n)$$