10) int sum = 0;

for (int counter = n; counter > 0; counter = counter - 2){

sum = sum + counter;

}

The dominant operation is the for loop which iterates from n to 0 by decrementing n by 2 each iteration. inside this for loop is a single operation, so the operation will be performed n/2 times during the course of this algorithm. Since we can eliminate constants, we can say this is n \* 1/2 and remove the 1/2 which leaves n. Since the operation is growing proportionally with n, we can say that it is linear growth or O(n) complexity.

12) for (int pass = 1; pass <= n; pass++) {

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

for (int count = 1; count < 10; count++) {

...

}

}

}

The dominant operations in this algorithm is a series of loops that are nested. Starting from the inner most for loop we have and int count that will increment from 1 to 9 which will run 9 times no matter what n is, so this will be constant time, O(1) and can be put aside for now. The next loop up has int index that increments by one from 0 to n-1, so this will run n times before finishing its iterations. This will be linear time, O(n). The top level is a for loop where int pass will iterate from 1 to n, so this one will also run n times. With a nested loop that has two linear time constant functions, we can see that our actual time complexity will be O(n)\*O(n) or O(n^2)

16) Program A = 1000=n^2 operations. Program B = 2^n operations. for what n will A < B.

n = 10, Program A - 100,000 > Program B - 1,024

n = 11, Program A - 121,000 > Program B - 2,048

n = 12, Program A - 144,000 > Program B - 4,096

n = 13, Program A - 169,000 > Program B - 8,192

n = 14, Program A - 196,000 > Program B - 16,384

n = 18, Program A - 324,000 > Program B - 262,144

n = 19, Program A - 361,000 < Program B - 524,288

n = 20, Program A - 400,000 < Program B - 1,048,576

At 19 operations, Program A will run less operations than Program B and will be more efficient.

24) Write a program in java that takes in an array of unique integers in a random order in range 1 to n+1, then performs a search to locate the missing integer from the array.

**package** CriticalThinking.Module3;

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** IntegerLocator {

**public** **static** **void** main(String[] args) {

Set<Integer> ints = **new** HashSet();

**int** n = *randomIntGen*(0, 25);

**int** min = 1;

System.***out***.println("Number of integers: " + n + "\n");

**for** (**int** i = 0; i < n - 1; i++) {

**int** randomInt = *randomIntGen*(min, n);

**if** (ints.contains(randomInt)) {

i--;

} **else** {

ints.add(randomInt);

}

}

String dispInts = ints.toString();

System.***out***.println("Given numbers: " + dispInts + "\n");

**for** (**int** i = 1; i <= n; i++) {

**if** (!ints.contains(i)) {

System.***out***.println("Number missing: " + i);

}

}

}

**private** **static** **int** randomIntGen(**int** min, **int** max) {

**return** (**int**) (Math.*random*() \* (max - min + 1)) + min;

}

}