Lab 2

1. Compute the number of primitive operations for each of the following algorithm fragments. The "increment counter" step has not been mentioned explicitly – be sure to take this into account in your computations. *Hint*: When faced with nested `for` loops, compute the number of steps required by the inner loop first, and then figure out the effect of the outer loop.

```
A. sum ← 0
   for i ← 0 to n-1 do
      sum ← sum + 1

B. sum ← 0
   for i ← 0 to n-1 do
      for j ← 0 to n-1 do
         sum ← sum + 1
```

2. Determine the asymptotic running time of the following procedure (an exact computation of number of basic operations is not necessary):

```
int[] arrays(int n) {
   int[] arr = new int[n];
   for(int i = 0; i < n; ++i){
      arr[i] = 1;

   }
   for(int i = 0; i < n; ++i) {
      for(int j = i; j < n; ++j){
         arr[i] += arr[j] + i + j;
      }
   }
   return arr;

}
```

3. Consider the following problem: As input you are given two sorted arrays of integers. Your objective is to design an algorithm that would merge the two arrays together to form a new sorted array that contains all the integers contained in the two arrays. For example, on input

   `[1, 4, 5, 8, 17], [2, 4, 8, 11, 13, 21, 23, 25]`

   the algorithm would output the following array:

   `[1,2,4,4,5,8,8, 11, 13, 17, 21, 23, 25]`

   For this problem, do the following:

   A. Design an algorithm `Merge` to solve this problem and write your algorithm description using the pseudo-code syntax discussed in class.
   B. Examining your pseudo-code, count the number of basic operations used

C. Implement your pseudo-code as a Java method `merge` having the following signature:

```
int[] merge(int[] arr1, int[] arr2)
```

Be sure to test your method in a `main` method to be sure it really works!

4. **Big-oh and Little-oh.** Use the definitions of `O(f(n))` and `o(f(n))` given in class to decide whether each of the following is true or false, and in each case, prove your answer.

A. $1 + 4n^2$ is $O(n^2)$
B. $n^2 - 2n$ is *not* $O(n)$
C. $\log(n)$ is $o(n)$
D. n is *not* $o(n)$

5. **Big-oh Rules.** Suppose you know that *f(n)* and *g(n)* are both O(*h(n)*). Does it follow that the function *f(n)* +*g(n)* is also O(*h(n)*)? Prove your answer.

6. **Power Set Algorithm**. Given a set X, the power set of X, denoted P(X), is the set of all subsets of X. Below, you are given an algorithm for computing the power set of a given set. This algorithm is used in the "naïve" solution to the Knapsack Problem, discussed in the first lecture. Implement this algorithm in a Java method:

```
List powerSet(List X)
```

Algorithm: PowerSet(X)

Input: A list X of elements
Output: A list P consisting of all subsets of X – elements of P are *Sets*

```
P ← new list
S ← new Set     //S is the empty set
P.add(S)        //P is now the set { S }
T  ← new Set
while (!X.isEmpty() ) do
    f  ← X.removeFirst()
    for each x in P do
        T ← x ∪ {f}   // T is the set containing f & all elements of x
        P.add(T)
return P
```

7. Prove by induction that for all $n > 4$, $F_n > (4/3)^n$. Then use this result to explain the approximate asymptotic running time of the recursive algorithm for computing the Fibonacci numbers. Is the recursive Fibonacci algorithm fast or slow? Why?

   Hint #1. First prove by induction that for all n > 1,

   $$(4/3)^{n-2} + (4/3)^{n-1} > (4/3)^n$$

   Hint #2. For the base case of your proof, the following table of values may be useful:

n	F_n	$(4/3)^n$
0	0	1
1	1	1.33
2	1	1.78
3	2	2.37
4	3	3.16
5	5	4.21
6	8	5.62

8. Develop an algorithm that does the following: Given an integer n, the algorithm outputs a randomly generated array in which each of the integers 0, 1, 2, …, n-1 occurs exactly once. Express your algorithm in pseudo-code. Determine (or estimate) the asymptotic running time of your algorithm. *Note:* "Randomly generated array" in this context means (1) arrays or array values are obtained from a random process, and (2) the likelihood of any particular arrangement of {0, 1, 2, . . . n – 1} being generated is exactly the same as the likelihood of any other arrangement being generated.

9. More big-oh: (Work with someone in your group who is familiar with limits)
   a. True or false: 4^n is $O(2^n)$. Prove your answer.
   b. True or false: log n is $\Theta(\log_3 n)$. Prove your answer.
   c. True or false: (n/2) log(n/2) is Θ(nlog n). Prove your answer.

10. Below, pseudo-code is given for the recursive factorial algorithm `recursiveFactorial`.

```
Algorithm recursiveFactorial(n)
  Input: A non-negative integer n
  Output: n!
   if (n = 0 || n = 1) then
       return 1
   return n * recursiveFactorial(n-1)
```

    Do the following:

A. Use the Guessing Method to determine the worst-case asymptotic running time of this algorithm. Then verify correctness of your formula.
B. Prove the algorithm is correct.

11. Devise an iterative algorithm for computing the Fibonacci numbers and compute its running time. Prove your algorithm is correct.

12. Find the asymptotic running time using the Master Formula:

```
T(n) = T(n/2) + n; T(1) = 1
```

13. You are given a length-n array A consisting of 0s and 1s, arranged in sorted order. Devise an algorithm that counts the total number of 0s and 1s in the array that runs *faster than* $\Theta(n)$ time. (In other words, you should be able to prove that the running time of your algorithm is O(n) but not $\Theta(n)$. Intuitively, this means that your algorithm, when run on A, examines *far fewer than* n slots in the array in order to output a result.) Your algorithm may not make use of auxiliary storage such as arrays or hashtables (more precisely, the only additional space used, beyond the given array, is O(1)). You must give an argument to show that your algorithm runs in O(n) time but not in $\Theta(n)$ time.