ASSIGNMENT 2 – 10%

Submit the compulsary answers to this worksheet, which are parts A (all), B (4,6,10), and C(4, 6, 10).

Part A: Theory

The first part of this worksheet focuses on understanding pseudo code. For each problem, you must do three things: (i) carry out a dry run with test data, (ii) calculate the running time using operation counting, and (iii) approximate the running time from part (ii) identify the running time in "Big-Oh" notation (some pointers on how to do this are outlined in the notes at the end of this worksheet) from first principles.

NOTE: Some of the algorithms will not have a variable table because the algorithm contains no variables.
NOTE: For array-based algorithms, please use an array of length 5 in the dry runs.

By way of example, I have included a complete answer to question 1 below.

1) **Algorithm** Increment(a):

```
    Input: An integer a
    Output: a + 1

    output ← a + 1
    return output
```

Answer (note: i have included a variable here only so that you see a sample variable table):

(i) The test data for this algorithm is a=5, and the dry run produces the following output:

Output
6

(ii) Operation Count for algorithm:

Algorithm Increment(a):

```
    Input: An integer a
    Output: a + 1

    output ← a + 1                          2 operations
    return output                           1 operation
                                  Total =   3 operations
```

(iii) "Big-Oh" for this algorithms is O(1)

2) **Algorithm** Average(a, b):

```
    Input: Two integers a and b
    Output: The average of a and b

    return (a + b) / 2
```

3) **Algorithm** MaxInt(a, b):

```
    Input: Two integers a and b
    Output: The larger of the two integers

    if a > b then
        return a
    else
        return b
```

4) **Algorithm** Difference(a, b):

```
    Input: Two integers a and b
    Output: The difference between a and b

    if (a > b) then
        return a – b
    else
        return b - a
```

5) **Algorithm** MinValue(A, n):

```
    Input: An integer array A of size n
    Output: The smallest value in A

    minValue ← A[0]
    for k=1 to n-1 do
        if (minValue > A[k]) then
            minValue ← A[k]

    return minValue
```

6) **Algorithm** MinValueIndex(A, n):

```
    Input: An integer array A of size n
    Output: The position of the smallest value in A

    minValueIndex ← 0
    for k=1 to n-1 do
        if (A[minValueIndex] > A[k]) then
            minValueIndex ← k

    return minValueIndex
```

7) **Algorithm** PrefixAverages2(A, n):

```
Input: An integer array A of size n.
Output: An array X of size n such that X[j] is the average of A[0], ..., A[j].

Let X be an integer array of size n
runningSum ← 0
for j=0 to n-1 do
   runningSum ← runningSum + A[k]
   X[j] ← runningSum / (j+1)

return X
```

8) **Algorithm** Factorial(a):

```
Input: An integer a
Output: The value of a factorial (a!)

factorial ← 1
for k=1 to a do
   factorial ← factorial * k

return factorial
```

9) **Algorithm** Power(a, b):

```
Input: Two integers a and b
Output: The value of a to the power b

power ← 1
for k=1 to b do
   power ← power * a

return power
```

10) **Algorithm** LinearSearch(A, n, q):

```
Input: An integer array A of size n and a query q that we wish to search the
       array for.
Output: The position of q in A or -1 if q is not in A

index ← 0
while (index < n) and (A[index] <> q) do
   index ← index + 1

if (index = n) then
   return -1
else
   return index
```

Part B: Translating Pseudo Code

This next section of the worksheet requires that you write Java code that implements the algorithms whose pseudo code was given in Part A. Remember that because none of these algorithms are associated with a larger concept (i.e. a data structure), they should be implemented as static methods.

To verify that you have implmented these algorithms correctly, you are required to add additional output statements (System.out.println) that replicate the variable tables from the dry runs you carried out in Part A. This means that your main method should call the corresponding algorithm with the relevant test data. By way of illustration, I have completed question 1 for you below.

NOTE: The class name for each program should take the for Bx.java where x is the question number that corresponds to the question in Part A (see question 1 below for an illustration).

1) Java Code (B1.java):

```
public class B1 {
    public static void main(String[] args) {
        increament(5);
    }

    public static int increment(int a) {
        System.out.println("output");
        int output = a + 1;
        System.out.println(output);
        return output;
    }
}
```

Example Output:

```
Output
6
```

Part C: Evaluation

The final part of this worksheet requires that you carry out an experimental evaluation of each of the algorithms using the experimental framework outlined in the lectures. For each algorithm, you need to instrument some code so that you can generate timing information. For non-fixed time algorithms plot the results as a line chart using MS Excel where the horizontal axis corresponds to n (here n is used to represent the parameter that affects the performance) and the vertical axis corresponds to time. Again, sample code and output for question 1 is given below.

NOTES:

- To carry out results smoothing, you should average the performance of the algorithm over 5 runs.
- For arrays, use random data over the following values of N: 100000, 200000, 500000, 1000000, 2000000.
- As with Part B, Java files should have a name of the form Cx.java, where x is the question number.

1) Instrumented Code (C1.java):

```
public class B1 {
    public static void main(String[] args) {
        double duration = 0;
        for (int i=0; i<5; i++) {
            long start = System.currentTimeMillis();
            increment(5);
            long end = System.currentTimeMillis();
            duration += end-start;
        }
        duration = duration / 5;
        System.out.println("Running Time=" + duration);
    }

    public static int increment(int a) {
        int output = a + 1;
        return output;
    }
}
```

Example Output:

```
Running Time=0.0
```

EXCEL GRAPH IS REQUIRED

NOTES: “Big-Oh” Notation

“Big-Oh” Notation is a technique for approximating the running time of algorithms so that they are more easily compared. The need for a “Big-Oh” type technique arises because operation counting is not always the best way of approximating the running time of an algorithm.

Another technique, discussed later in the course, is to use inductive proofs. These proofs are effective at approximating the running time of algorithms that contain loops whose operations are not easy to count. An example of this is the Binary Search Algorithm, which operates on sorted arrays (see below):

```
Algorithm BinarySearch(A, n, q):
  Input: An integer array A of size n and a query q that we wish to search the
         array for.
  Output: The position of q in A or -1 if q is not in A

  if (q < A[0] or q > A[n-1]) then return -1

  lower ← 0
  upper ← n-1

  repeat
    midpoint ← (upper + lower) / 2
    if (A[midpoint] < q) then
      lower ← midpoint+1
    if (A[midpoint] < q) then
      upper ← midpoint-1
  until A[midpoint] = q or lower > upper

  if (lower > upper) then
    return -1

  return midpoint
```

Basically, the running time of this algorithm is easier to work out if we consider how the range [lower, upper] changes with each iteration. We will discuss this in the lectures, however, for illustration purposes, we find out that the running time is proportional to log n.

The problem is that, we want to be able to compare the results of our inductive proof with the operation counting results that we have obtained. To do this, we need to further approximate the running time, and this is where “Big-Oh” comes in.

Basically, “Big-Oh” notation states that:

Given functions f(n) and g(n), we say that
f(n) is **O(**g(n)**)** if and only if
there are positive constants c and n_0 such that
$f(n) \leq c * g(n)$ for all $n \geq n_0$

What does this mean? Well, basically, lets consider the function $f(n) = 3n+2$. Any function, $g(n)$, that satisfies the above constraints can be associated with $f(n)$ via "Big-Oh":

$g(n) = n^2 + 2$

Need to choose some c and n_0 so that:

$$3n+2 \leq c(n^2 + 2)$$

Lets choose c=1 and rewrite the inequality as follows

$$3n+2 \leq n^2 + 2$$
$$0 \leq n^2 - 3n$$
$$0 \leq n\ (n-3)$$

Critical values of n are 0 and 3. Lets choose $n_0 = 3$, and see that $n(n-3)$ will be ≥ 0 for all values of $n \geq 3$. This means that $f(n) \leq n^2 + 2$ for all $n \geq 3$, so $3n+2$ is $O(n^2+2)$.

Similarly, lets see what happens if we choose $g(n) = n$...

We need to choose some c and n_0 so that:

$$3n+2 \leq c(n)$$

Lets choose c=4 and rewrite the inequality as follows

$$3n+2 \leq 4n$$
$$2 \leq n$$

This means that, if we choose c=4 and n_0=2, we find that 3n+2 is O(n).

So what I hear you cry!!! Well, all of this is useful is we can find some value in comparing the "Big-Oh" value, and this is where it gets interesting.

Mathematically speaking, there is a hierarchy of functions:

$$1 < \log n < n < n \log n < n^2 < n^2 \log n < n^3 < \ldots$$

So, if we can approximate a function to one of the functions in this hierarchy, then we can compare algorithms based on their runningt time algorithms position in the hierarchy. For example, I mentioned above that binary search algorithm has running time O(log n). As you will see in the problem sheet, the linear search algorithm has a running time of O(n). This means that binary search is better than linear search because binary search is closer to the summit of the function hierarchy (1) than linear search. More importantly, we are able to state this even though we used two different evaluation techinques for the two algorithms...

While any function g(n) can be used with "Big-Oh" the basic guideline is to choose one from the hierarchy of functions. As a general rule, you should try to match the highest order term in f(n) and making a judicious choice of c.