Analysis of Algorithms

AA Lab 1

1 Empirical Analysis

In today's lab, we're going to write some programs to help us understand the implications of the complexity of an algorithm.

1.1 Setup

1. In Eclipse, create a project called "Lab 1". This project will have the following classes.

1.2 Timer

This class is going to be used to keep track of the time elapsed in running our algorithms.

1. Create a new class in your project, called "Timer"
2. This class must have the following attributes, both of which are of type **long**:
   - startTime
   - endTime
3. The class must have a **start** method which sets the startTime of the object to the current time, a **stop** method which sets the endTime of the method to the current time and a **getTime** method which returns the difference between the two.
4. **System.currentTimeMillis()** is the builtin function that can be used to get the current time. It return a value of type **long**.

1.3 Algorithms

1. Now create another class in the project, called "Algorithms"
   Our Algorithms class will contain many different algorithms to use for testing
2. These algorithms make use of an array of integers. We must thus first write a static method called **initialize** that randomly generates integers. This method is given the number of items to be generated. It will then create an array of that size, fill it with random integers between 0 and 1000000, and return the array. Code that generates a random number between 0 and 1000000 follows:

   ```
   Random generator = new Random();
   int number = generator.nextInt(1000000);
   ```

   Be sure to import java.util.Random

3. Next, create a static method called **linearCount**, which is given an array and an integer, and searches through the array for the integer, returning the number of times the integer occurs in the array. Note that the complexity of the linear count is $\theta(n)$

4. Now, create a main method which looks as follows:

```
public static void main(String args[]){
   int size=10000;
   Timer myTimer=new Timer();
   int[] data = initialize(size);
   myTimer.start();
   int position = linearCount(data, 50);
   myTimer.stop();
   System.out.println("Time : " + myTimer.getTime());
}
```

   This method will record how long it takes to perform a linear search on 10000 items.

5. Now, to compare it to an algorithm whose complexity is $\theta(n^2)$, we are going to implement a bubble sort. Note that the complexity of the bubble sort algorithm is $\theta(n^2)$. Implement a static **bubbleSort** method that takes an array as input and sorts it using the following algorithm:

```
for every array position i
    examine every array position j where j>i
        if the element in position i is greater than the element
        in position j, swap the elements in positions i and j
return the array
```

6. Now we need to compare the real world effect of the complexity difference between these two algorithms.

7. We will now modify our **main** method so that it runs both algorithms on the same data set, and outputs both times.

```
public static void main(String[] args) {
   int size=10000;
   Timer myTimer=new Timer();
   int[] data = initialize(size);
   myTimer.start();
   int position = linearCount(data, 50);
   myTimer.stop();
   System.out.println("Search Time : " + myTimer.getTime());

   myTimer.start();
   bubbleSort(data);
   myTimer.stop();
   System.out.println("Sort Time : " + myTimer.getTime());
}
```

8. Submit your code to the marking system. Make sure your source files are not in a package or folder when you make your zip file.

9. Now, change the *size* value, and record the times taken for input sizes of 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000 and 100000 items.

10. For interests sake, try running the program for a *size* of 1000000.

11. Note how the time taken to perform the linear search grows dramatically slower than the time taken to perform the bubble sort.

12. This comparison is not very useful except for illustrative purposes, as the algorithms perform completely different tasks. Often many algorithms are available that could perform the same operation, and this exercise illustrates why the complexity of these algorithms is the primary criterion when selecting one.