# Homework 3 – Kyle Guarco

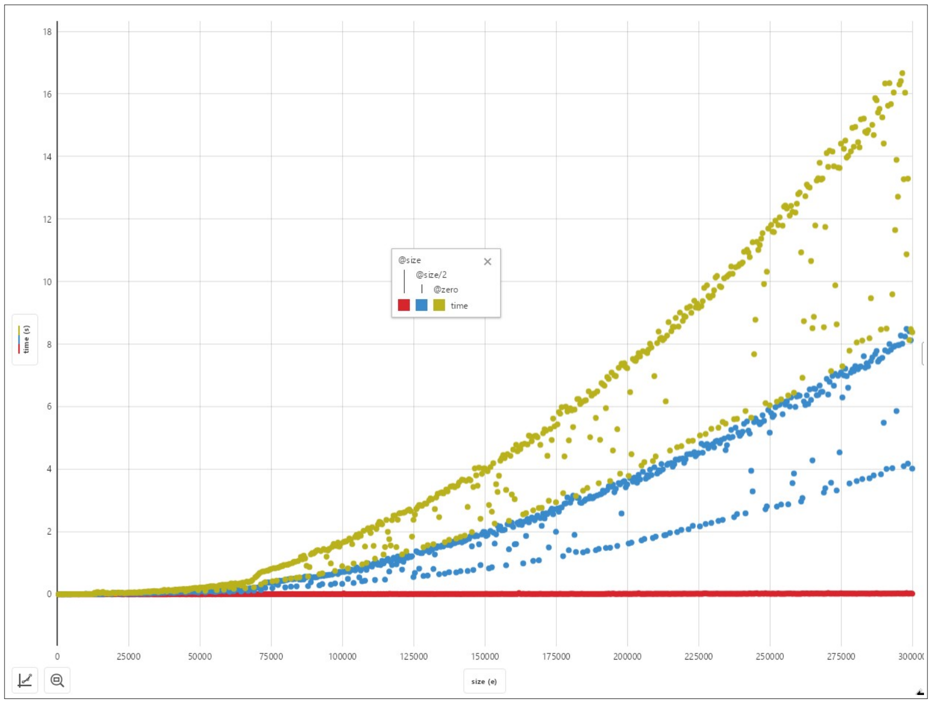
The goal of this assignment is to measure the efficiency of an algorithm with certain inputs. The algorithm efficiency measured in this assignment was the **add(int, int)** function in **java.util.ArrayList**.

The program written below computes the efficiency (measured in nanoseconds, converted to seconds) of **add(int, int)** for the following inputs:

```
                         ArrayList<Integer> list = new ArrayList<>();

        list.add(0);               list.add(list.size()/2, 0);            list.add(0, 0);
 // Insert @ position 'size'      // Insert @ position 'half'      // Insert @ position 'zero'
```

The output provided by the program was in the form of files (called a **Record** in the code). The records were batched together and graphed in *Vernier Graphical Analysis.* The tests were run three times and averaged together.



Ignoring the outliers (the points not in the assumed curve), we can see that inserting an element into the ArrayList at position 'zero' (the yellow-ish curve on the graph) takes the longest as the size of the array gets larger. The red part of the graph, representing the insertion at position 'size', is really close to zero, which leads to the conclusion that it takes almost no time at all to insert at the end of the ArrayList.

**EfficiencyTest.java**

```java
import java.io.IOException;
import java.util.ArrayList;

public class EfficiencyTest {
   public static void main(String[] args) {
      final String FILEPATH = "tests/";
      final int NANOS_SECONDS = 1000000000;

      Job.setMaximumThreadCount(6);

      for (int i = 500; i <= 300000; i += 500) {
         Job job = new Job((jobargs) -> {
            int size = (int) jobargs[0];
            System.out.println("Running job with input size " + size);
            try {
               ArrayList<Integer> testlist = new ArrayList<>();

               Record record = new Record(FILEPATH + size);
               record.createNewHandle("size");
               record.createNewHandle("half");
               record.createNewHandle("zero");

               Long time_size = FunctionTimer.repeatTime(size,
                     (varargs) -> testlist.add(0));
               testlist.clear();
               Long time_half = FunctionTimer.repeatTime(size,
                     (varargs) -> testlist.add(testlist.size()/2, 0));
               testlist.clear();
               Long time_zero = FunctionTimer.repeatTime(size,
                     (varargs) -> testlist.add(0, 0));

               record.write("size", time_size.doubleValue() / NANOS_SECONDS);
               record.write("half", time_half.doubleValue() / NANOS_SECONDS);
               record.write("zero", time_zero.doubleValue() / NANOS_SECONDS);

               record.flush();
               record.close();
            } catch (IOException e) {}
         }, i);

         job.queue();
      }

      Job.process();
      // Waits for the other processes to finish.
      Job.join();
   }
}
```

## FunctionTimer.java

```java
/**
 * This class provides functions that can record the execution time of a given
 * function.
 *
 * @author Kyle Guarco
 */
public class FunctionTimer {

   /**
    * Returns the execution time of the function, in nanoseconds.
    *
    * @param function
    * @param args Arguments for the function.
    * @return
    */
   public static Long time(VaridicFunction function, Object... args) {
      Long starttime = System.nanoTime();

      function.call(args);

      return System.nanoTime() - starttime;
   }

   /**
    * Repeats {@code FunctionTimer.time()} according to {@code count}.}
    *
    * @param count
    * @param function
    * @return
    */
   public static Long repeatTime(int count, VaridicFunction function, Object... args) {
      Long total = 0L;

      for (int i = 0; i < count + 1; i++)
         total += FunctionTimer.time(function, args);

      return total;
   }

   @FunctionalInterface
   static interface VaridicFunction {
      void call(Object... args);
   }
}
```

## Job.java

```java
import java.util.NoSuchElementException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

/**
 * This class schedules any number of tests to be executed, timed and recorded
 * according to a specified function.
 *
 * @author Kyle Guarco
 */
public class Job {
   private static Queue<Job> requestqueue = new ConcurrentLinkedQueue<>();
   private static ThreadGroup testtg = new ThreadGroup("Test ThreadGroup");
   private static Thread processthread;
   private static int threadmax = 1;

   /**
    * Starts the process that starts jobs in threads of their own.
    */
   public static void process() {
      Job.processthread = new Thread(() -> {
         while (requestqueue.size() > 0) {
            if (testtg.activeCount() <= Job.threadmax) {
               try {
                  Job job = requestqueue.remove();

                  new Thread(Job.testtg, () -> job.request.call(job.args)).start();
               } catch (NoSuchElementException e) {
               }
            } else {
               try {
                  Thread.sleep(10L);
               } catch (InterruptedException e) {}
            }
         }
      });

      Job.processthread.start();
   }

   /**
    * Waits for the processing thread to die (blocking).
    */
   public static void join() {
      if (Job.processthread != null) {
         try {
            Job.processthread.join();
         } catch (InterruptedException e) {}

         Job.processthread = null;
      }
   }

   /**
    * Sets the maximum amount of jobs that can run concurrently.
    *
    * @param threadcount
    */
   public static void setMaximumThreadCount(int threadcount) {
      Job.threadmax = threadcount;
   }

   public FunctionTimer.VaridicFunction request;
   public Object[] args;

   public Job(FunctionTimer.VaridicFunction request, Object... args) {
      this.request = request;
      this.args = args;
   }

   /**
    * Queues this test for execution. The test request and test instructions
    * must be specified, or this test will not queue.
    */
   public boolean queue() {
      if (request == null) {
         return false;
      }

      Job.requestqueue.add(this);
      return true;
   }
}
```

## Record.java

```java
import java.io.Closeable;
import java.io.File;
import java.io.Flushable;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;

/**
 * This program logs the given set of data to a number of files in a folder.
 *
 * @author Kyle Guarco
 */
public class Record implements Closeable, Flushable {
   private File dir;
   private HashMap<String, PrintWriter> handles;

   /**
    * Creates a new record pointing to specified path. Records are WRITE-ONLY!
    *
    * @param filepath   The filepath for this record.
    * @param recordname The name for the record.
    * @throws IOException If the record couldn't be created.
    */
   public Record(String filepath) throws IOException {
      this.dir = new File(filepath);

      if (!dir.exists()) {
         dir.mkdirs();
      }

      this.handles = new HashMap<>();
   }

   @Override
   public void close() throws IOException {
      for (PrintWriter writer : handles.values())
         writer.close();
   }

   @Override
   public void flush() throws IOException {
      for (PrintWriter writer : handles.values())
         writer.flush();
   }

   /**
    * Creates a new part of the record
    *
    * @param name The name of this record handle.
    * @return Was the record created successfully?
    * @throws IOException
    */
   public boolean createNewHandle(String name) throws IOException {
      if (handles.containsKey(name))
         return false;

      File file = new File(dir, name);

      if (!file.exists())
         file.createNewFile();

      PrintWriter writer = new PrintWriter(file);
      handles.put(name, writer);

      return true;
   }

   /**
    * Writes to the specified record handle. It must exist.
    * See {@code Record.createNewHandle}.
    *
    * @param name The name of the record handle to write to.
    * @param objects The data to write to the handle (specified by {@code Object.toString()})
    */
   public void write(String name, Object... objects) {
      PrintWriter writer = handles.get(name);

      for (Object obj : objects)
         writer.println(obj.toString());
   }
}
```