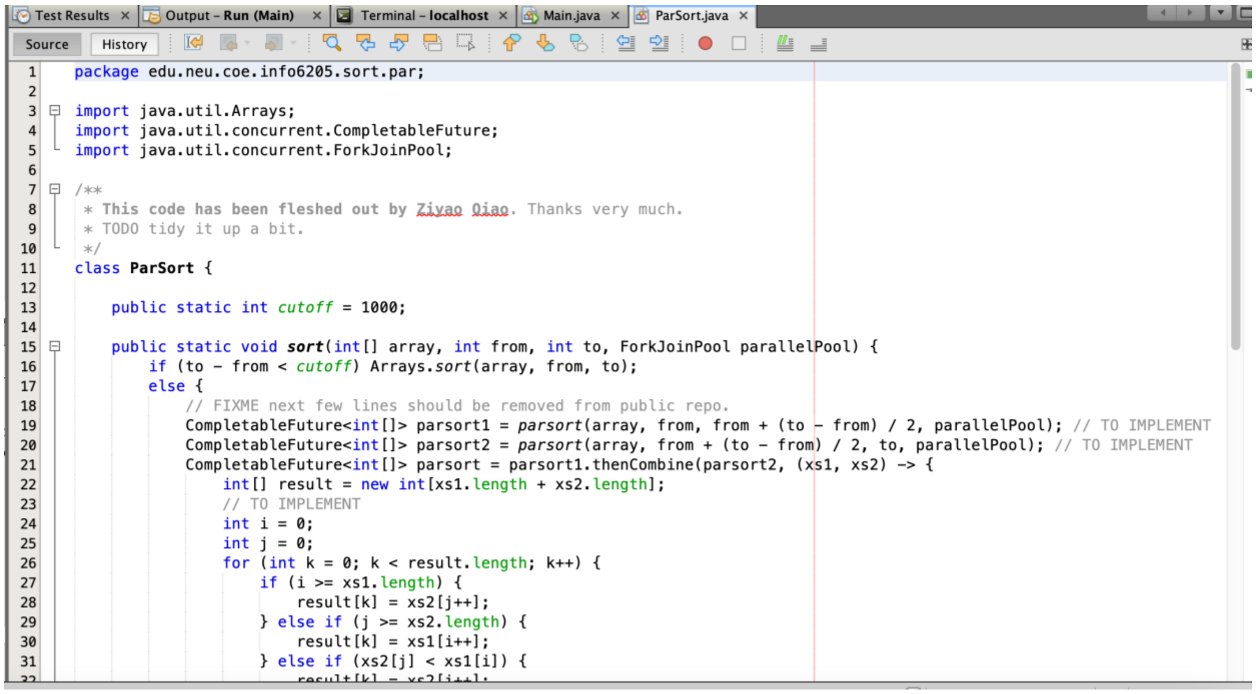
Program Structures and Algorithms
Spring 2022
Assignment -4
Parallel Sorting

Name: Sangram Vuppala
NU ID: 2958963
Section: 1

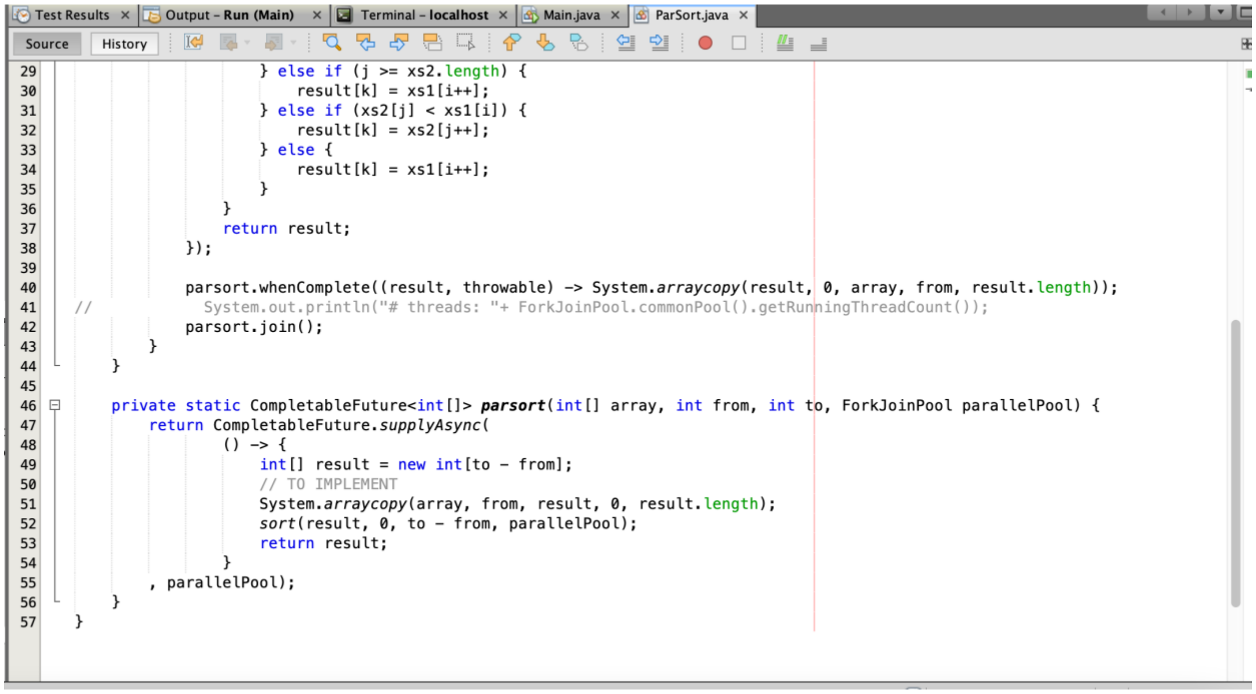
Task:

1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. Recursion depth or the number of available threads. Using this determination, you might decide on an ideal number (*t*) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of *lg t* is reached).
3. An appropriate combination of these.

Code Changes:

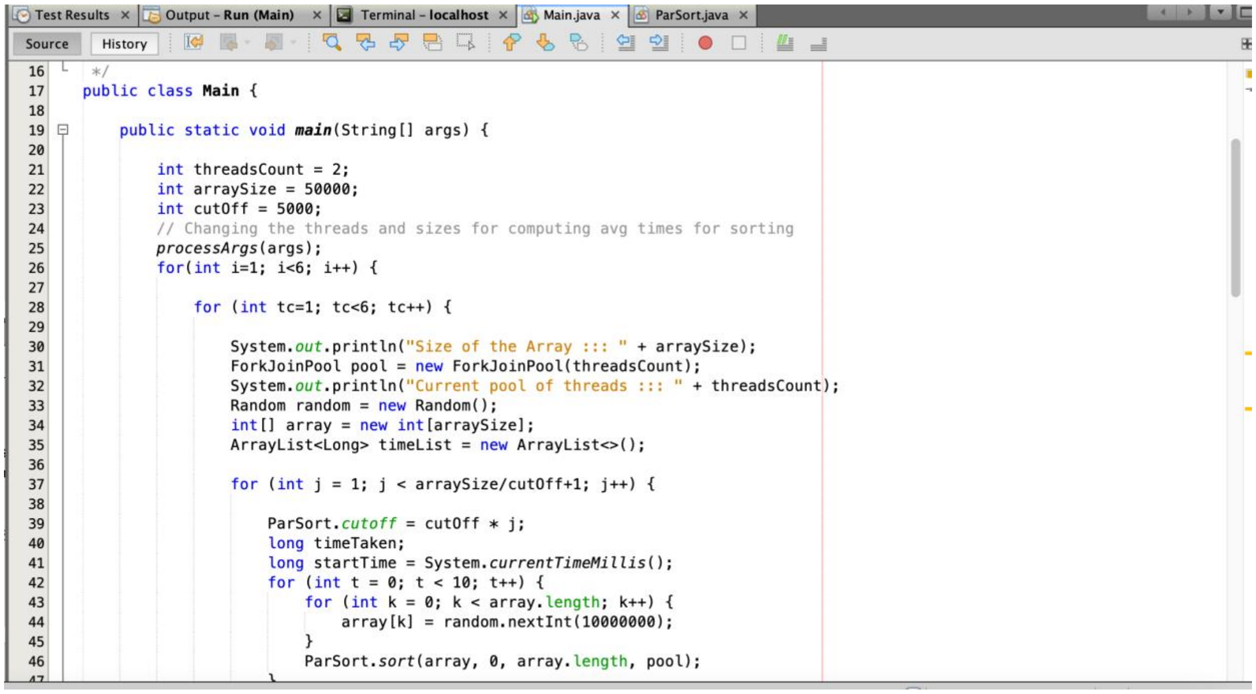


```
package edu.neu.coe.info6205.sort.par;

import java.util.Arrays;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ForkJoinPool;

/**
 * This code has been fleshed out by Ziyao Qiao. Thanks very much.
 * TODO tidy it up a bit.
 */
class ParSort {

    public static int cutoff = 1000;

    public static void sort(int[] array, int from, int to, ForkJoinPool parallelPool) {
        if (to - from < cutoff) Arrays.sort(array, from, to);
        else {
            // FIXME next few lines should be removed from public repo.
            CompletableFuture<int[]> parsort1 = parsort(array, from, from + (to - from) / 2, parallelPool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, from + (to - from) / 2, to, parallelPool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {
                int[] result = new int[xs1.length + xs2.length];
                // TO IMPLEMENT
                int i = 0;
                int j = 0;
                for (int k = 0; k < result.length; k++) {
                    if (i >= xs1.length) {
                        result[k] = xs2[j++];
                    } else if (j >= xs2.length) {
                        result[k] = xs1[i++];
                    } else if (xs2[j] < xs1[i]) {
```

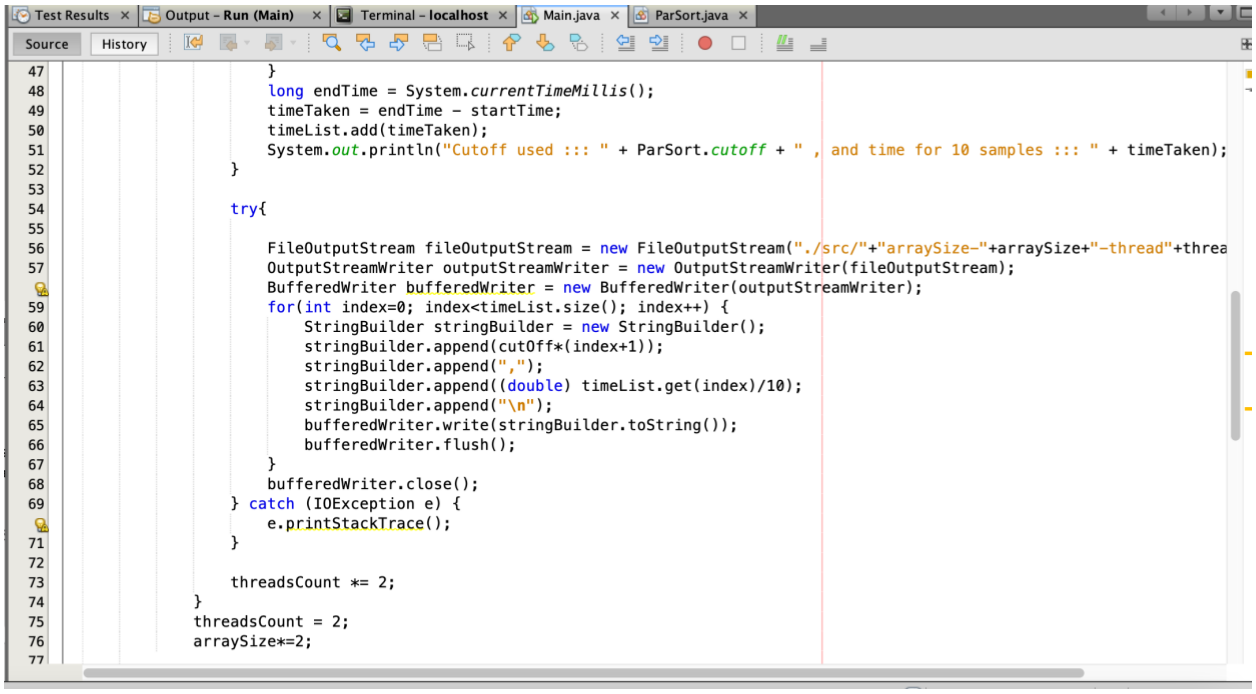


```java
                    } else if (j >= xs2.length) {
                        result[k] = xs1[i++];
                    } else if (xs2[j] < xs1[i]) {
                        result[k] = xs2[j++];
                    } else {
                        result[k] = xs1[i++];
                    }
                }
                return result;
            });

            parsort.whenComplete((result, throwable) -> System.arraycopy(result, 0, array, from, result.length));
//            System.out.println("# threads: "+ ForkJoinPool.commonPool().getRunningThreadCount());
            parsort.join();
        }
    }

    private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool parallelPool) {
        return CompletableFuture.supplyAsync(
                () -> {
                    int[] result = new int[to - from];
                    // TO IMPLEMENT
                    System.arraycopy(array, from, result, 0, result.length);
                    sort(result, 0, to - from, parallelPool);
                    return result;
                }
        , parallelPool);
    }
}
```

Main file



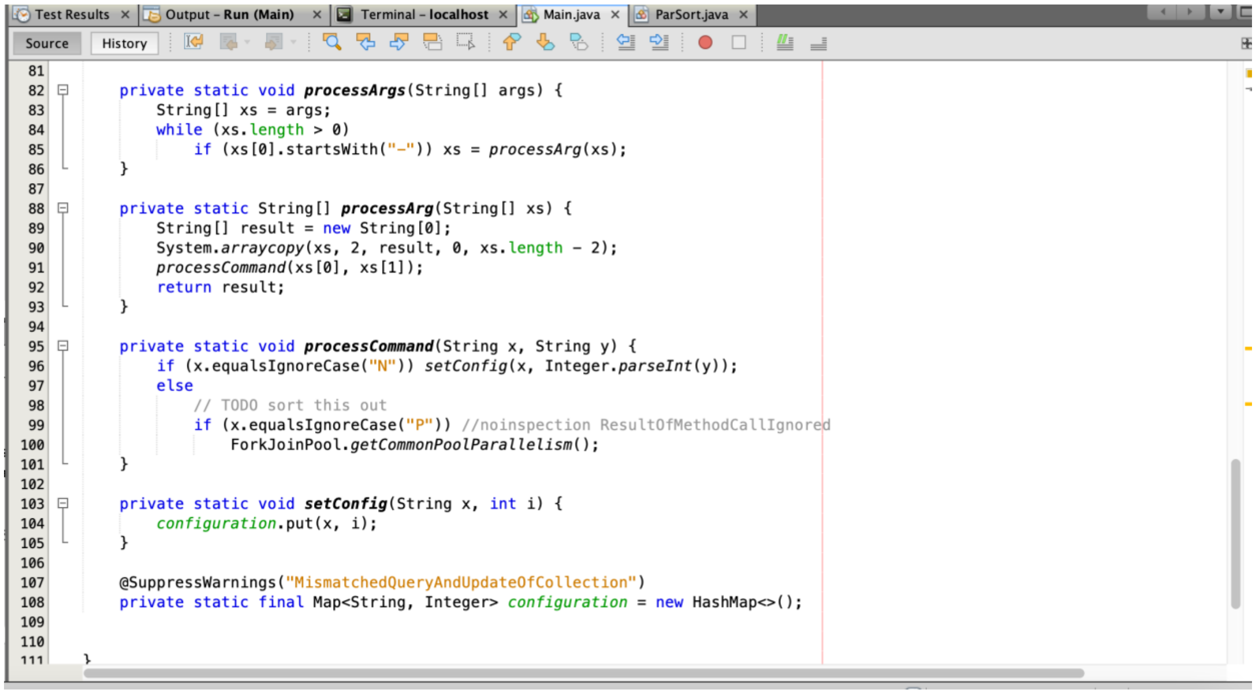
```java
 */
public class Main {

    public static void main(String[] args) {

        int threadsCount = 2;
        int arraySize = 50000;
        int cutOff = 5000;
        // Changing the threads and sizes for computing avg times for sorting
        processArgs(args);
        for(int i=1; i<6; i++) {

            for (int tc=1; tc<6; tc++) {

                System.out.println("Size of the Array ::: " + arraySize);
                ForkJoinPool pool = new ForkJoinPool(threadsCount);
                System.out.println("Current pool of threads ::: " + threadsCount);
                Random random = new Random();
                int[] array = new int[arraySize];
                ArrayList<Long> timeList = new ArrayList<>();

                for (int j = 1; j < arraySize/cutOff+1; j++) {

                    ParSort.cutoff = cutOff * j;
                    long timeTaken;
                    long startTime = System.currentTimeMillis();
                    for (int t = 0; t < 10; t++) {
                        for (int k = 0; k < array.length; k++) {
                            array[k] = random.nextInt(10000000);
                        }
                        ParSort.sort(array, 0, array.length, pool);
```



Test Results | Output - Run (Main) | Terminal - localhost | Main.java | ParSort.java

Source | History

```java
                }
                long endTime = System.currentTimeMillis();
                timeTaken = endTime - startTime;
                timeList.add(timeTaken);
                System.out.println("Cutoff used ::: " + ParSort.cutoff + " , and time for 10 samples ::: " + timeTaken);
            }

            try{

                FileOutputStream fileOutputStream = new FileOutputStream("./src/"+"arraySize-"+arraySize+"-thread"+threa
                OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream);
                BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);
                for(int index=0; index<timeList.size(); index++) {
                    StringBuilder stringBuilder = new StringBuilder();
                    stringBuilder.append(cutOff*(index+1));
                    stringBuilder.append(",");
                    stringBuilder.append((double) timeList.get(index)/10);
                    stringBuilder.append("\n");
                    bufferedWriter.write(stringBuilder.toString());
                    bufferedWriter.flush();
                }
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            threadsCount *= 2;
        }
        threadsCount = 2;
        arraySize*=2;
```



Test Results | Output - Run (Main) | Terminal - localhost | Main.java | ParSort.java

Source | History

```java

    private static void processArgs(String[] args) {
        String[] xs = args;
        while (xs.length > 0)
            if (xs[0].startsWith("-")) xs = processArg(xs);
    }

    private static String[] processArg(String[] xs) {
        String[] result = new String[0];
        System.arraycopy(xs, 2, result, 0, xs.length - 2);
        processCommand(xs[0], xs[1]);
        return result;
    }

    private static void processCommand(String x, String y) {
        if (x.equalsIgnoreCase("N")) setConfig(x, Integer.parseInt(y));
        else
            // TODO sort this out
            if (x.equalsIgnoreCase("P")) //noinspection ResultOfMethodCallIgnored
                ForkJoinPool.getCommonPoolParallelism();
    }

    private static void setConfig(String x, int i) {
        configuration.put(x, i);
    }

    @SuppressWarnings("MismatchedQueryAndUpdateOfCollection")
    private static final Map<String, Integer> configuration = new HashMap<>();


}
```

Observations:
after plotting the values of different cutoffs and threads after loading the outputs onto sheets I've arrived to the opinion that 4 threads is the best amount for processing because the performance does not change much as the number of threads increases. Also, when the cutoff is exactly 1/4 of the array's size, the performance is the lowest.

P.S: Remaining graphs and console outputs are saved in different files in the same git repository.