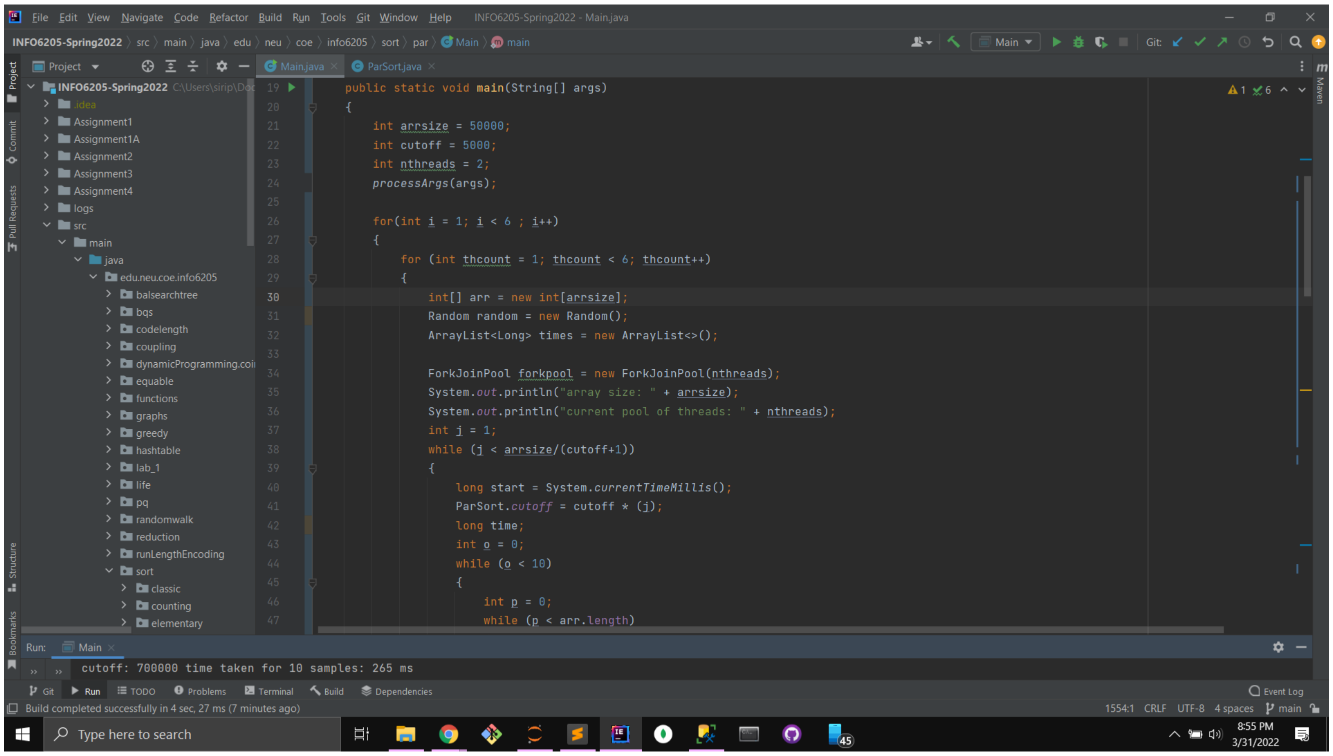
Program Structures & Algorithms
Spring 2022
Assignment No. 4

Name: Shashank Siripragada
NUID: 002193773

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: Main.java



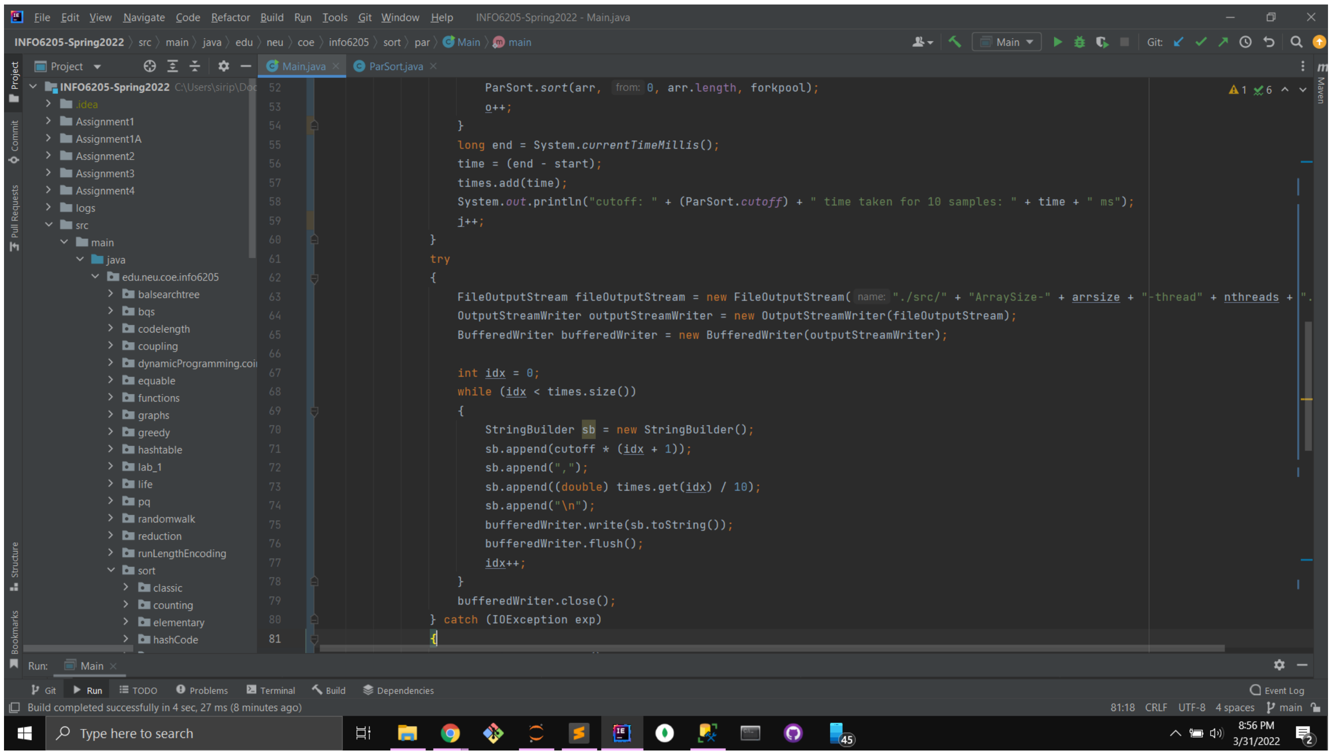
```java
public static void main(String[] args)
{
    int arrsize = 50000;
    int cutoff = 5000;
    int nthreads = 2;
    processArgs(args);

    for(int i = 1; i < 6 ; i++)
    {
        for (int thcount = 1; thcount < 6; thcount++)
        {
            int[] arr = new int[arrsize];
            Random random = new Random();
            ArrayList<Long> times = new ArrayList<>();

            ForkJoinPool forkpool = new ForkJoinPool(nthreads);
            System.out.println("array size: " + arrsize);
            System.out.println("current pool of threads: " + nthreads);
            int j = 1;
            while (j < arrsize/(cutoff+1))
            {
                long start = System.currentTimeMillis();
                ParSort.cutoff = cutoff * (j);
                long time;
                int o = 0;
                while (o < 10)
                {
                    int p = 0;
                    while (p < arr.length)
```

```
cutoff: 700000 time taken for 10 samples: 265 ms
```

Build completed successfully in 4 sec, 27 ms (7 minutes ago)

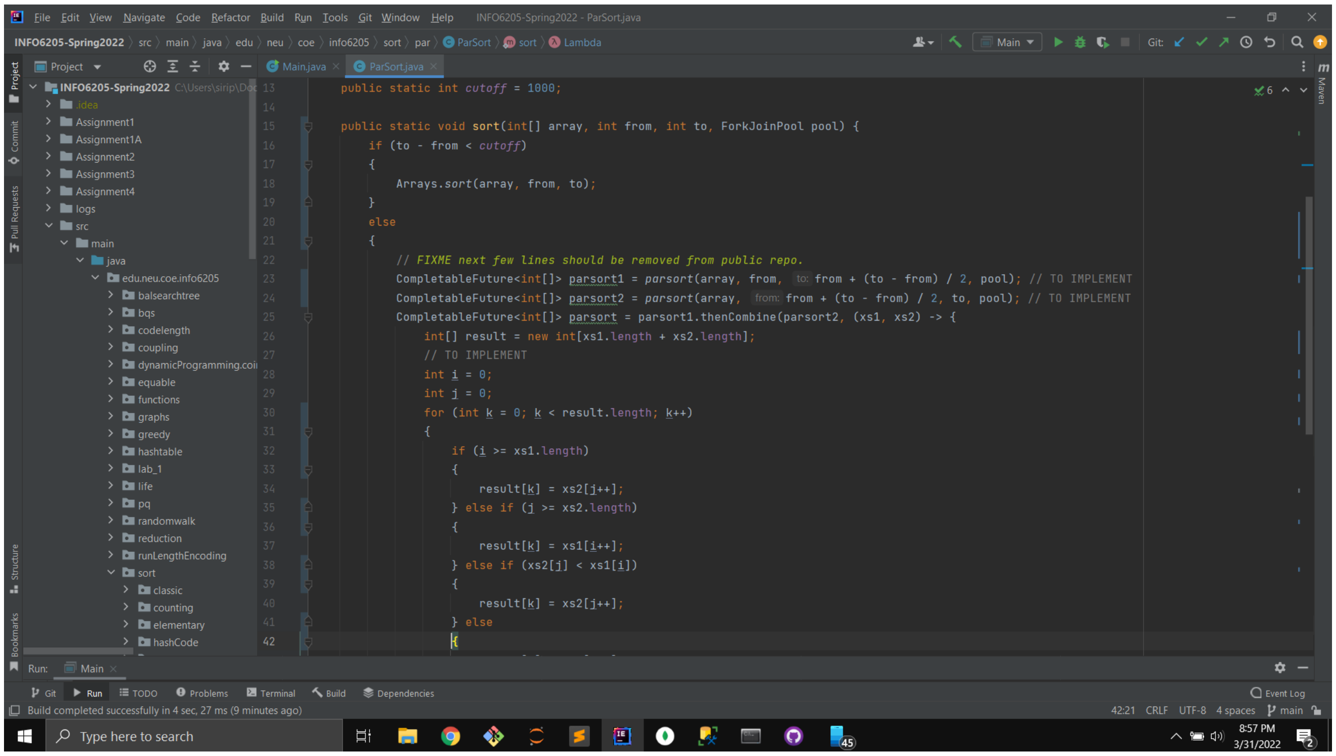


```java
                ParSort.sort(arr, 0, arr.length, forkpool);
                o++;
            }
            long end = System.currentTimeMillis();
            time = (end - start);
            times.add(time);
            System.out.println("cutoff: " + (ParSort.cutoff) + " time taken for 10 samples: " + time + " ms");
            j++;
        }
        try
        {
            FileOutputStream fileOutputStream = new FileOutputStream("./src/" + "ArraySize-" + arrsize + "-thread" + nthreads + ".
            OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream);
            BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);

            int idx = 0;
            while (idx < times.size())
            {
                StringBuilder sb = new StringBuilder();
                sb.append(cutoff * (idx + 1));
                sb.append(",");
                sb.append((double) times.get(idx) / 10);
                sb.append("\n");
                bufferedWriter.write(sb.toString());
                bufferedWriter.flush();
                idx++;
            }
            bufferedWriter.close();
        } catch (IOException exp)
        {
```

Build completed successfully in 4 sec, 27 ms (8 minutes ago)

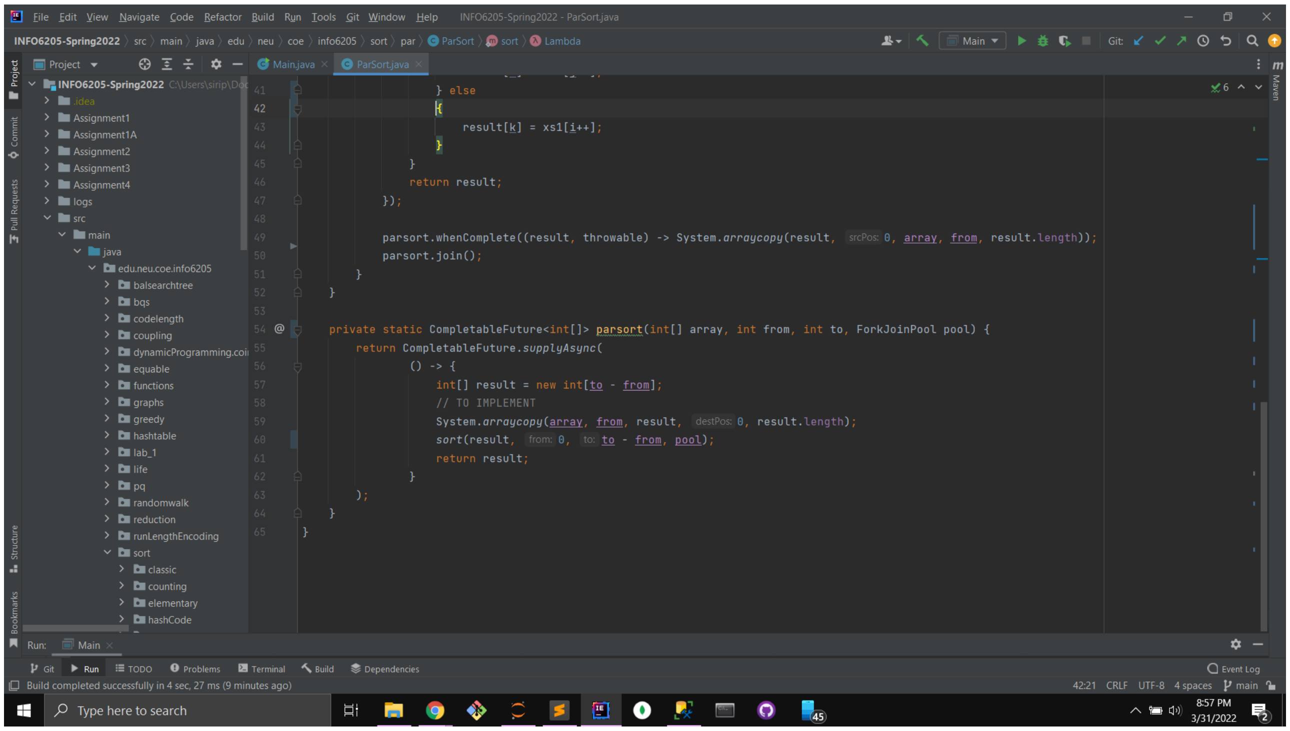
ParSort.java:



```java
    public static int cutoff = 1000;

    public static void sort(int[] array, int from, int to, ForkJoinPool pool) {
        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, pool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, from + (to - from) / 2, to, pool); // 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])
                    {
                        result[k] = xs2[j++];
                    } else
                    {
```

Build completed successfully in 4 sec, 27 ms (9 minutes ago)



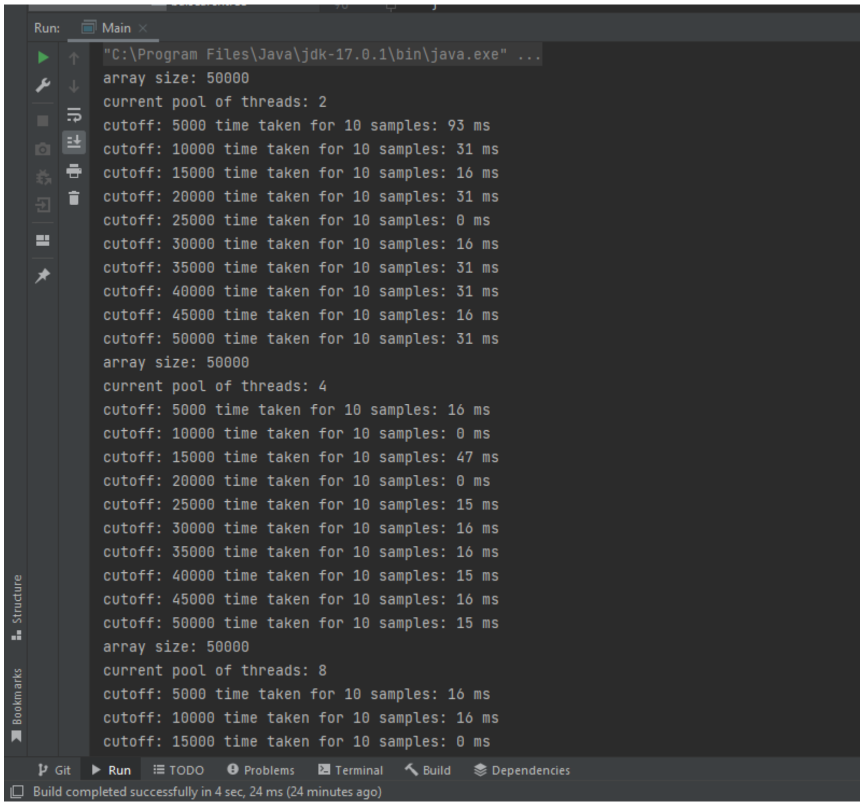
```java
                    } else
                    {
                        result[k] = xs1[i++];
                    }
                }
                return result;
            });

            parsort.whenComplete((result, throwable) -> System.arraycopy(result, srcPos: 0, array, from, result.length));
            parsort.join();
        }
    }

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

Build completed successfully in 4 sec, 27 ms (9 minutes ago)

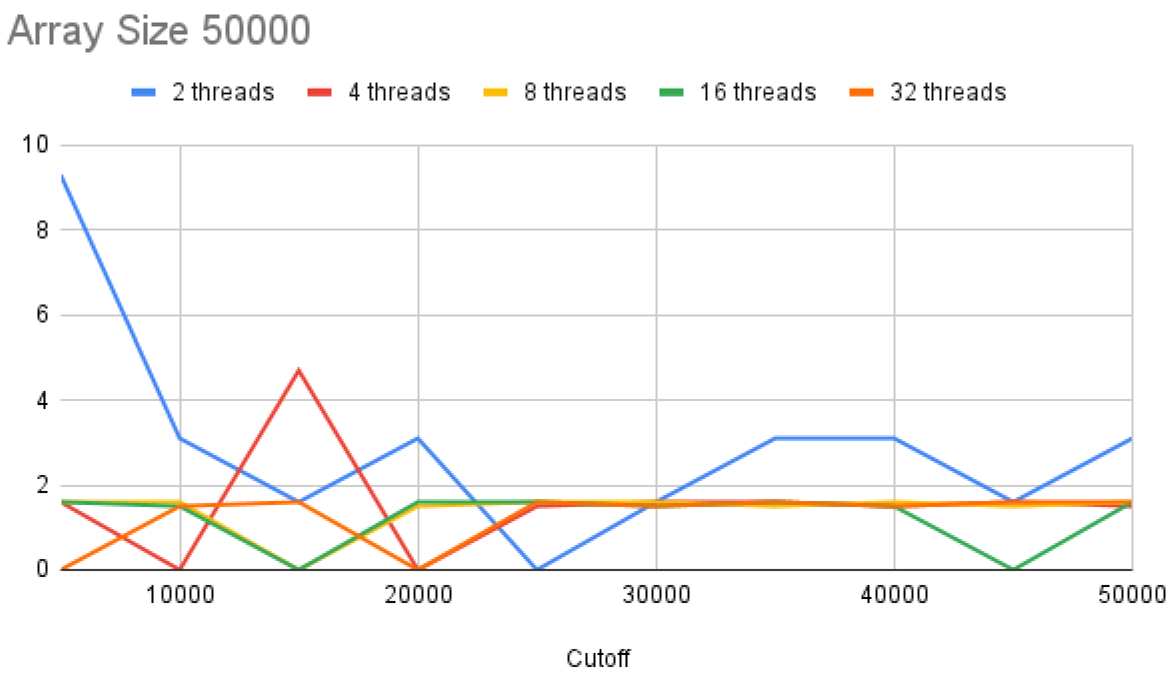
Console Output and CSV files: The csvs containing the observations are added to the src folder.

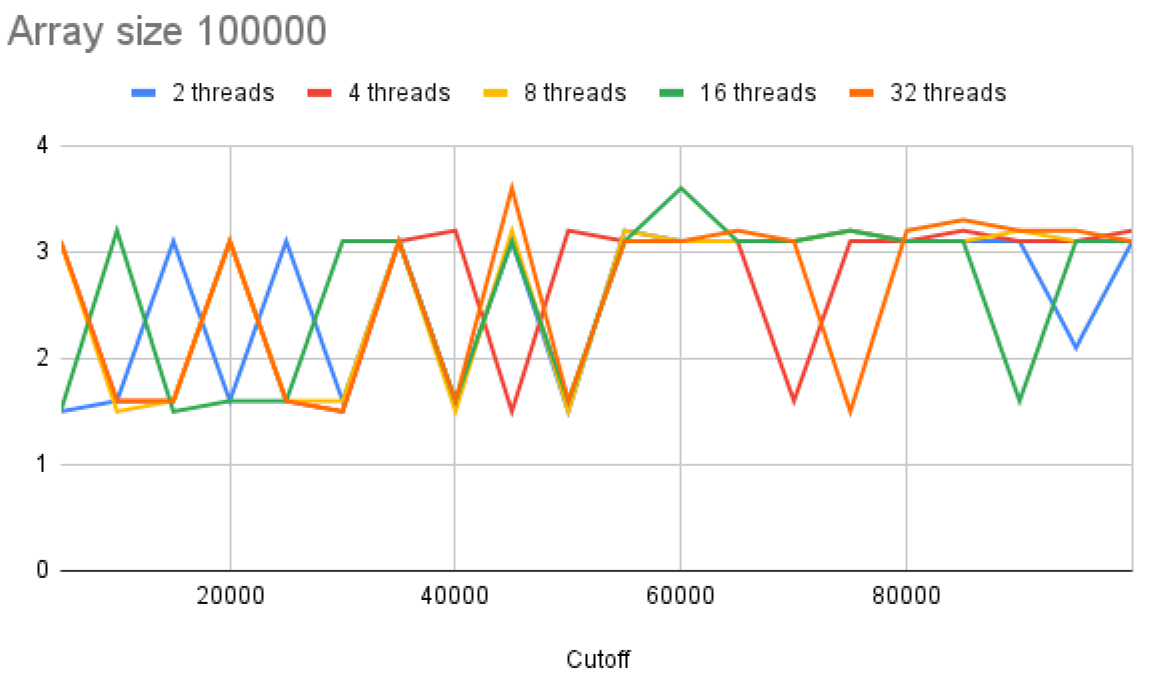


```
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" ...
array size: 50000
current pool of threads: 2
cutoff: 5000 time taken for 10 samples: 93 ms
cutoff: 10000 time taken for 10 samples: 31 ms
cutoff: 15000 time taken for 10 samples: 16 ms
cutoff: 20000 time taken for 10 samples: 31 ms
cutoff: 25000 time taken for 10 samples: 0 ms
cutoff: 30000 time taken for 10 samples: 16 ms
cutoff: 35000 time taken for 10 samples: 31 ms
cutoff: 40000 time taken for 10 samples: 31 ms
cutoff: 45000 time taken for 10 samples: 16 ms
cutoff: 50000 time taken for 10 samples: 31 ms
array size: 50000
current pool of threads: 4
cutoff: 5000 time taken for 10 samples: 16 ms
cutoff: 10000 time taken for 10 samples: 0 ms
cutoff: 15000 time taken for 10 samples: 47 ms
cutoff: 20000 time taken for 10 samples: 0 ms
cutoff: 25000 time taken for 10 samples: 15 ms
cutoff: 30000 time taken for 10 samples: 16 ms
cutoff: 35000 time taken for 10 samples: 16 ms
cutoff: 40000 time taken for 10 samples: 15 ms
cutoff: 45000 time taken for 10 samples: 16 ms
cutoff: 50000 time taken for 10 samples: 15 ms
array size: 50000
current pool of threads: 8
cutoff: 5000 time taken for 10 samples: 16 ms
cutoff: 10000 time taken for 10 samples: 16 ms
cutoff: 15000 time taken for 10 samples: 0 ms
```

Build completed successfully in 4 sec, 24 ms (24 minutes ago)

Plots:


Array Size 50000
2 threads
4 threads
8 threads
16 threads
32 threads
10
8
6
4
2
0
10000
20000
30000
40000
50000
Cutoff


Array size 100000
2 threads
4 threads
8 threads
16 threads
32 threads
4
3
2
1
0
20000
40000
60000
80000
Cutoff

Observations and Conclusion:

The plots are generated from the csv files containing different values of the cutoffs and threads.
It can be concluded that 4 will be the optimal number of threads as there is no change in the performance as we increase the threads.
The lowest performance is when the cutoff is ¼ size of the array.

For recursion depth and number of threads available

$$t=2^d$$

Maximum depth possible:

$$\lg(\text{arr size.} / \text{cutoff})$$