Assignment 5 - Program Structures and Algorithms Spring 2023(SEC - 1)

NAME: Paurush Batish **NUID: 002755631**

Your task is

Please see the presentation on *Assignment on Parallel Sorting* under the *Exams. etc.* module.
Your task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You will consider two different schemes for deciding whether to sort in parallel.

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.

There is a *Main* class and the *ParSort* class in the *sort.par* package of the INFO6205 repository. The *Main* class can be used as is but the *ParSort* class needs to be implemented where you see "TODO..." [it turns out that these TODOs are already implemented].
Unless you have a good reason not to, you should just go along with the Java8-style future implementations provided for you in the class repository.
You must prepare a report that shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cut-off schemes.

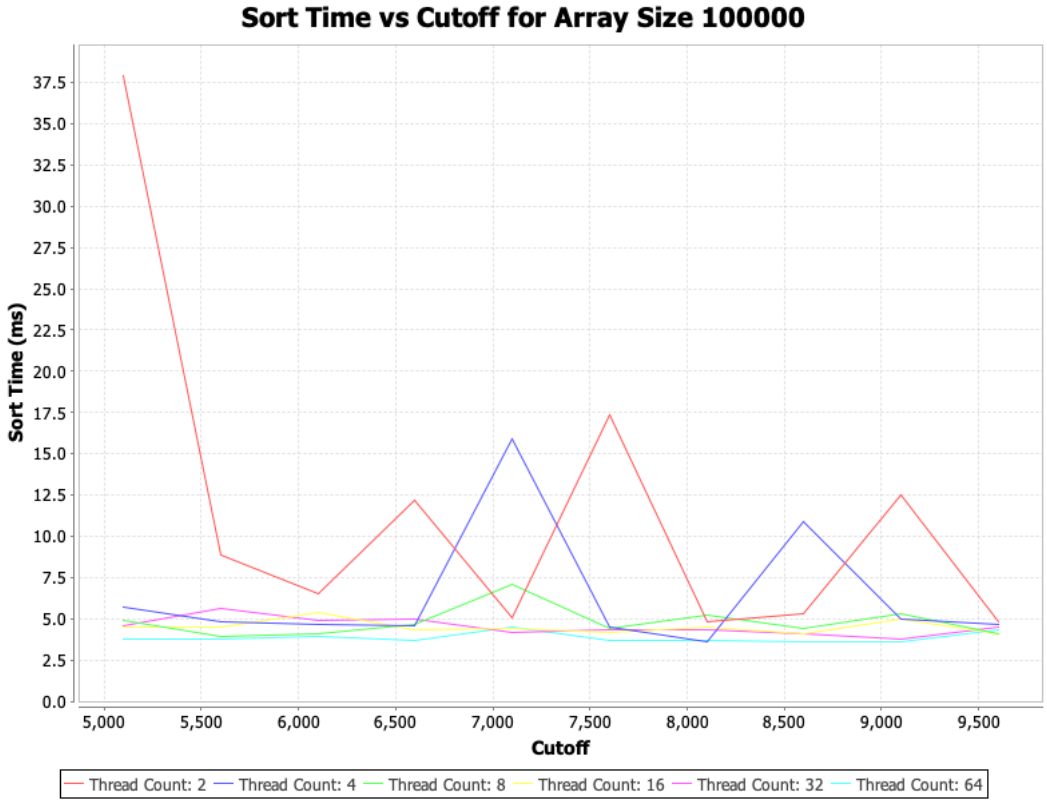
Solution –
Code snippets →

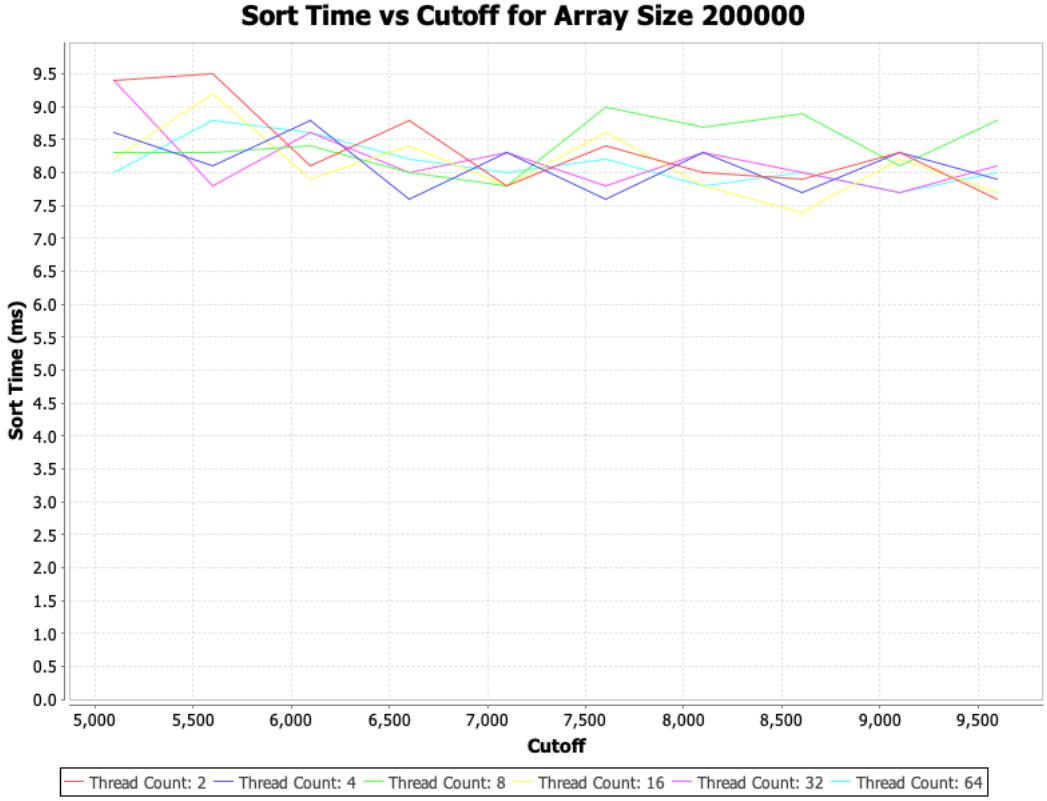
```java
Random random = new Random();

ArrayList<Long> timeList = new ArrayList<>();
int[] array;
for (int arraySize = 100000; arraySize <= 300000; arraySize += 100000) {
    System.out.println("Array size: " + arraySize);
    array = new int[arraySize];
    XYSeriesCollection dataset = new XYSeriesCollection();
    for (int threadCount = 2; threadCount < 65; threadCount = threadCount * 2) {
        System.out.println("Thread count is: " + threadCount);
        XYSeries series = new XYSeries("Thread Count: " + threadCount);
        for (int j = 50; j < 100; j += 5) {
            ParSort.cutoff = 100 * (j + 1);
            for (int i = 0; i < array.length; i++)
                array[i] = random.nextInt(10000000);
            long time;
            long startTime = System.currentTimeMillis();
            for (int t = 0; t < 10; t++) {
                for (int i = 0; i < array.length; i++)
                    array[i] = random.nextInt(10000000);
                ParSort.sort(array, 0, array.length);
            }
            long endTime = System.currentTimeMillis();
            time = (endTime - startTime);
            timeList.add(time);
            series.add(ParSort.cutoff, (double) time / 10);
        }
        dataset.addSeries(series);
    }
    JFreeChart chart = ChartFactory.createXYLineChart(
            "Sort Time vs Cutoff for Array Size " + arraySize, "Cutoff",
            "Sort Time (ms)", dataset, PlotOrientation.VERTICAL, true,
            true, false);

    // set plot colors
    XYPlot plot = (XYPlot) chart.getPlot();
    plot.setBackgroundPaint(Color.WHITE);
    plot.setDomainGridlinePaint(Color.LIGHT_GRAY);
    plot.setRangeGridlinePaint(Color.LIGHT_GRAY);
    plot.setOutlinePaint(Color.GRAY);


  // set axis colors
  NumberAxis domainAxis = (NumberAxis) plot.getDomainAxis();
  domainAxis.setLabelPaint(Color.BLACK);
  domainAxis.setTickLabelPaint(Color.BLACK);
  NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
  rangeAxis.setLabelPaint(Color.BLACK);
  rangeAxis.setTickLabelPaint(Color.BLACK);

  // save chart to file
  try {
      ChartUtilities.saveChartAsPNG(new File("./src/graph_" + arraySize + ".png"), chart, 800, 600);
  } catch (IOException e) {
      e.printStackTrace();
  }
```

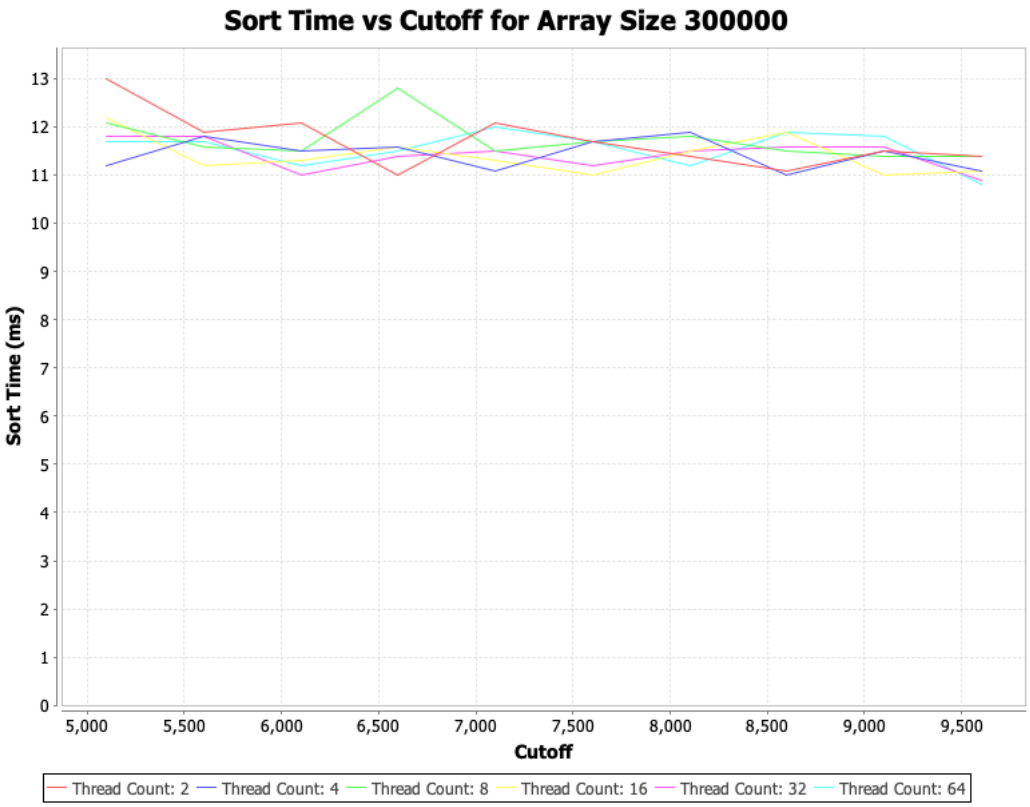
Observation →

1. As we increase the cut-off value, the time required to sort the array decreases, we can see in below graph .
2. As we increase the number of threads, the time required to sort the array also decreases.
3. As we increase the array size, the time required to sort the array also increases. The time complexity of parallel merge sort is O($nlogn$).

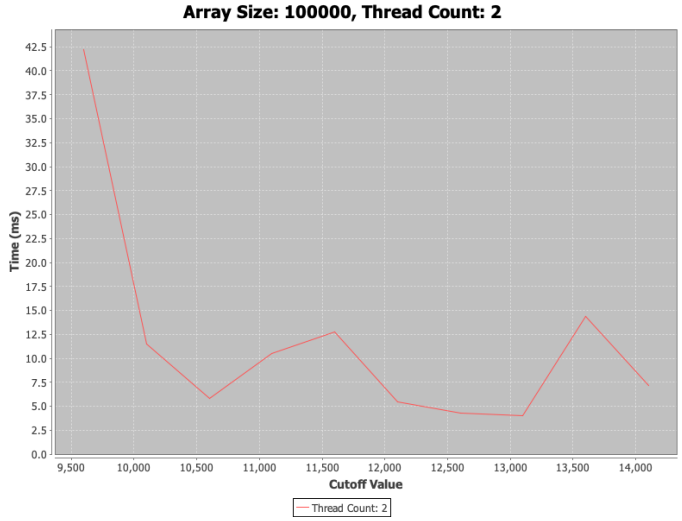
Now we can see this using graphs →

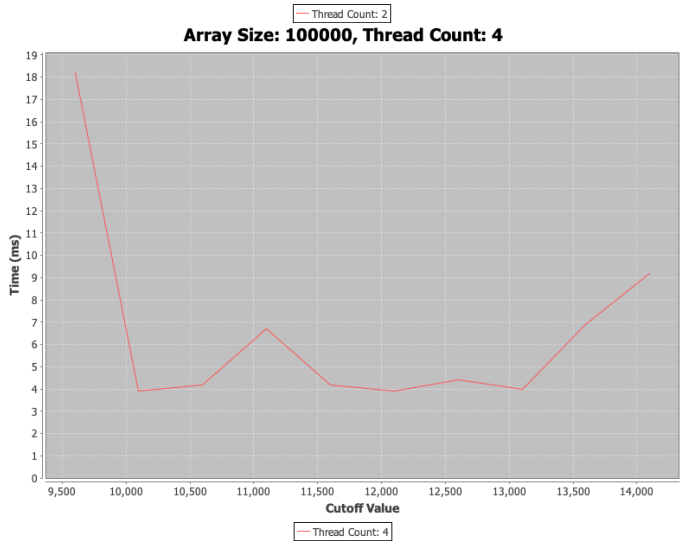


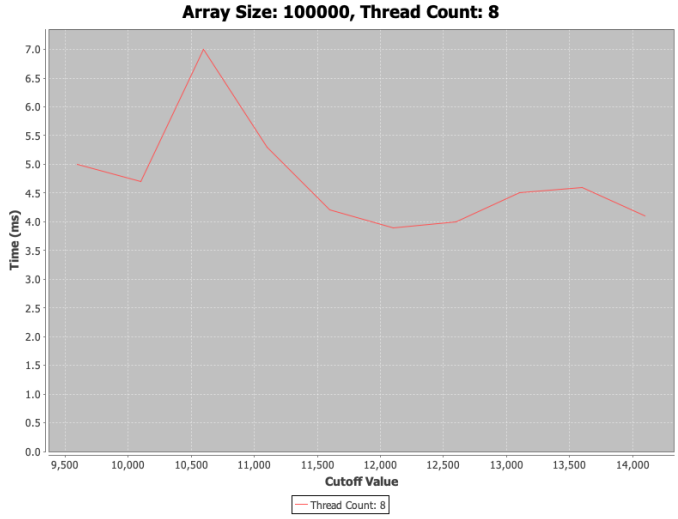


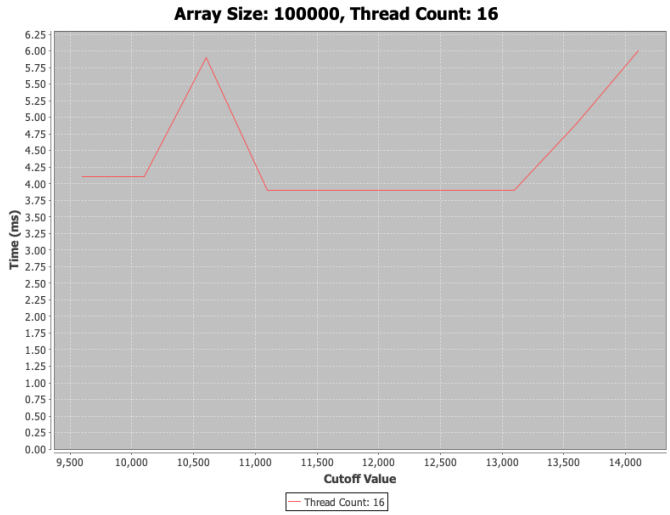

Sort Time vs Cutoff for Array Size 300000
Sort Time (ms)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
5,000
5,500
6,000
6,500
7,000
7,500
8,000
8,500
9,000
9,500
Cutoff
Thread Count: 2
Thread Count: 4
Thread Count: 8
Thread Count: 16
Thread Count: 32
Thread Count: 64

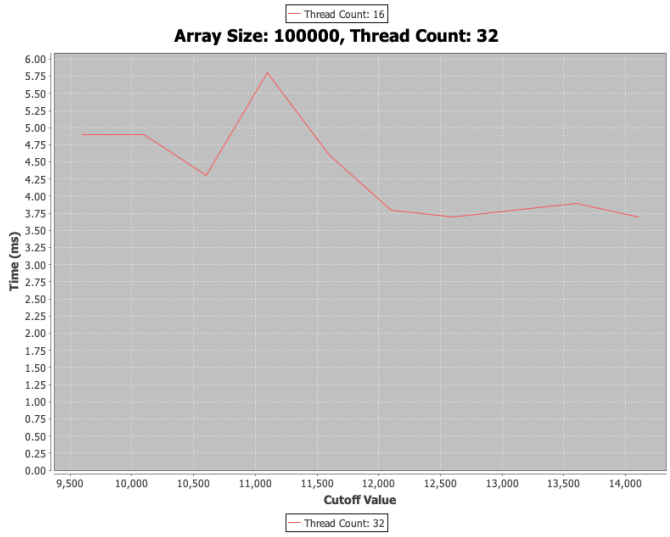
We can also draw a single graphs to show the above results →
For ArraySize – 100000





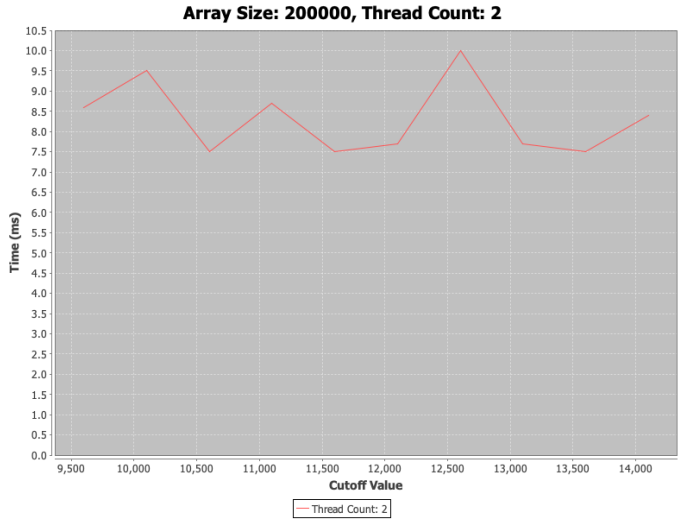


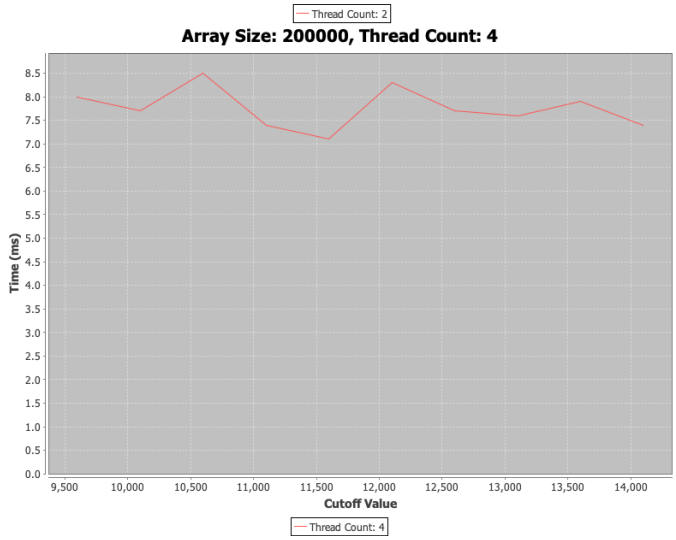


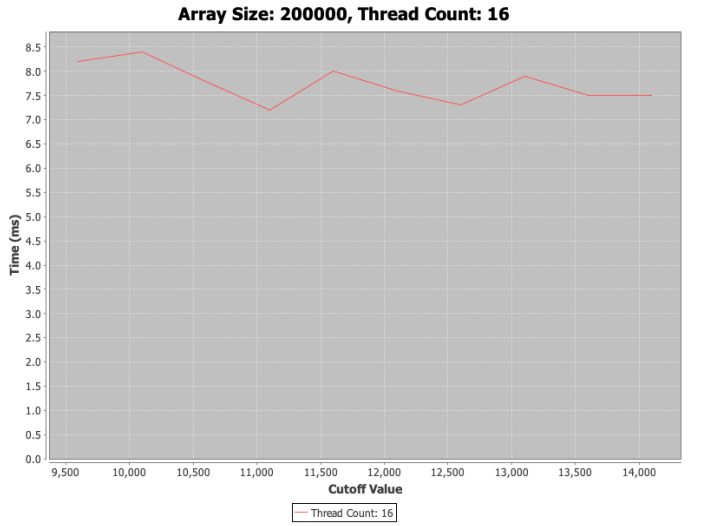


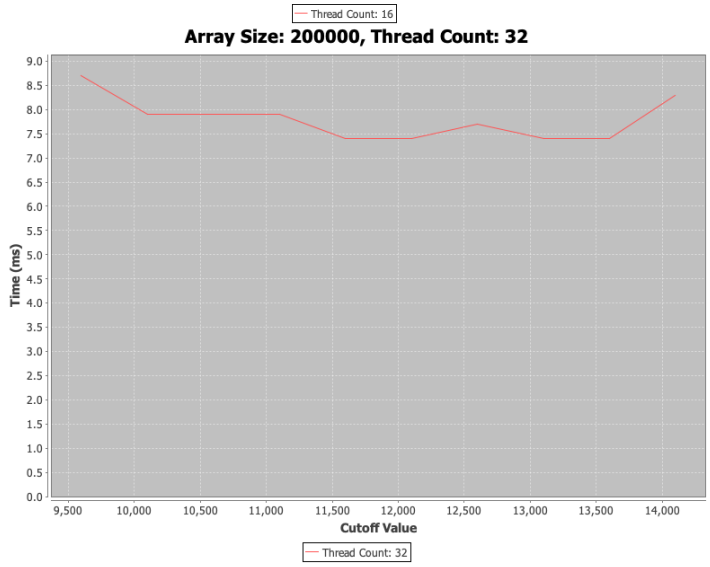
Based on these observations we can see that the number of threads for the above array size would be- 64

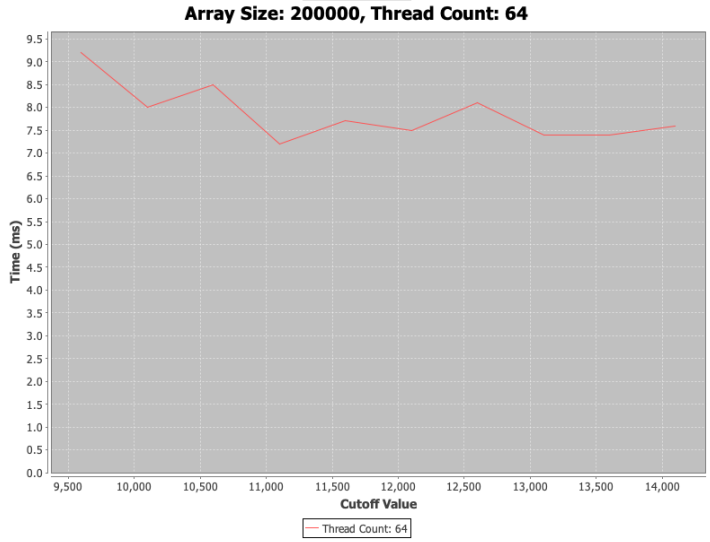
For ArraySize – 200000





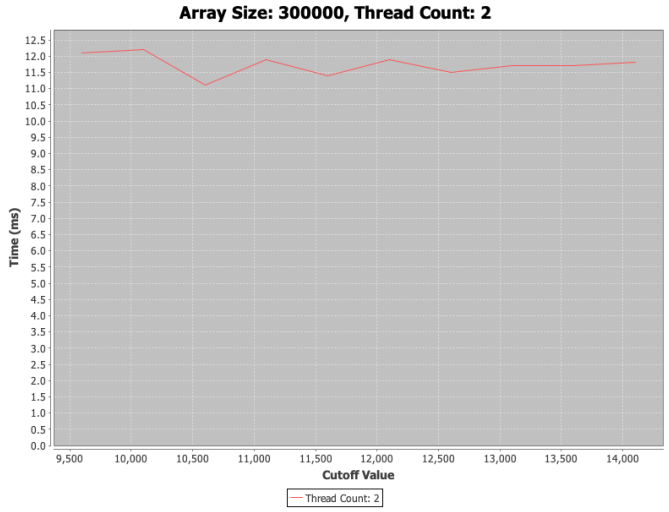


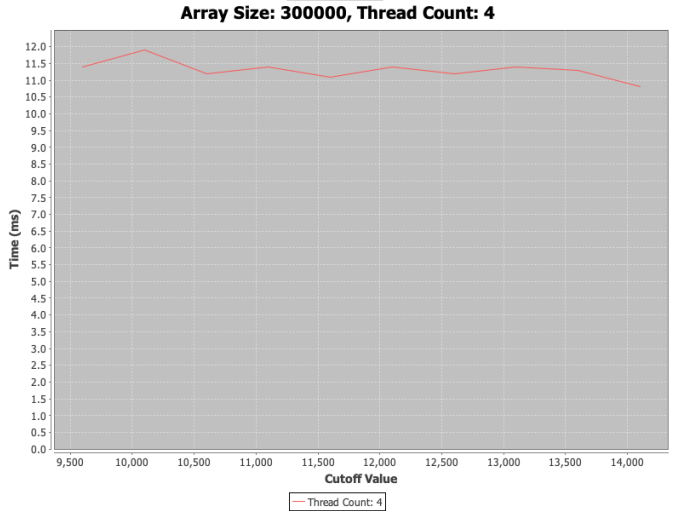


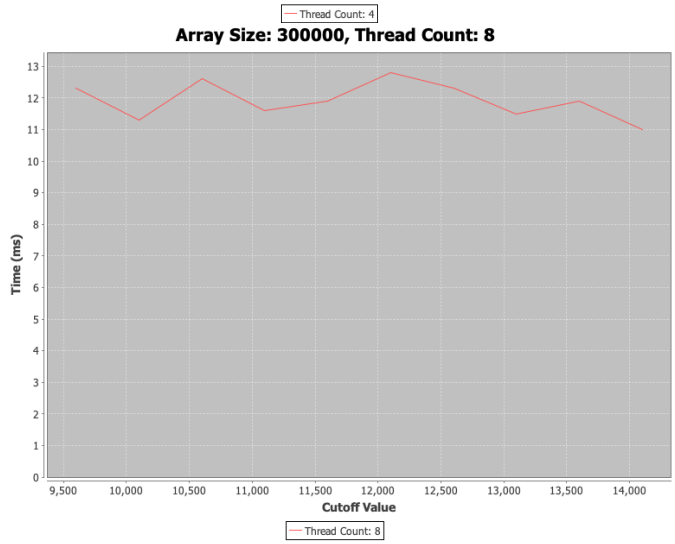


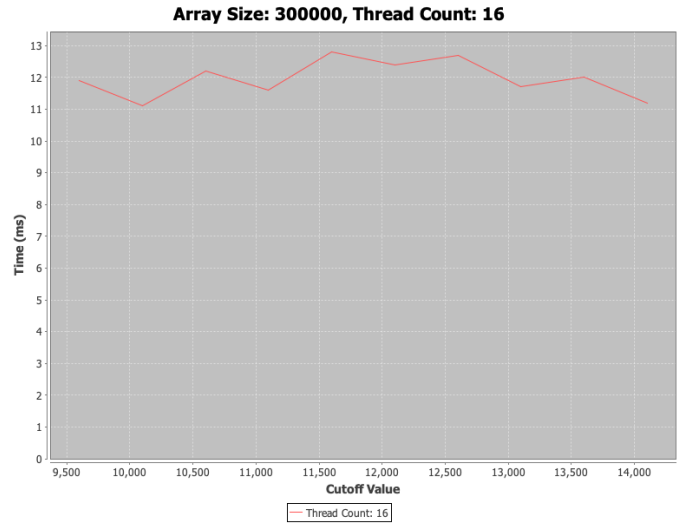
Based on these observations we can see that the number of threads for the above array size would be- 16

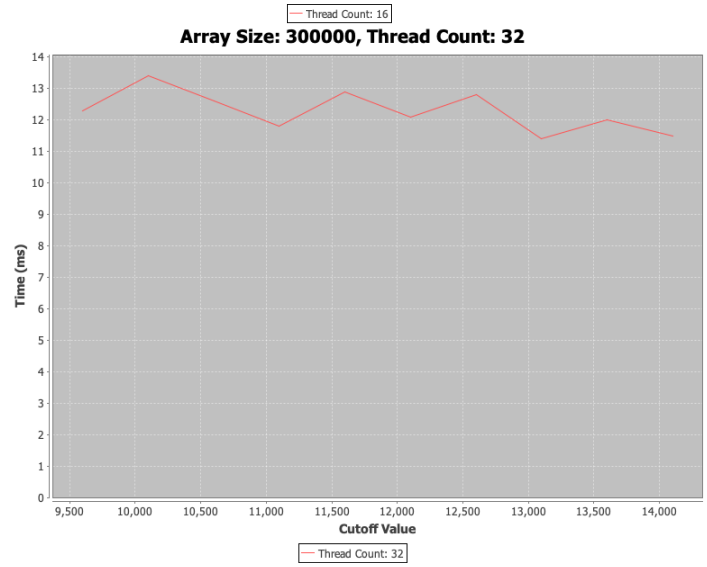
For ArraySize – 300000

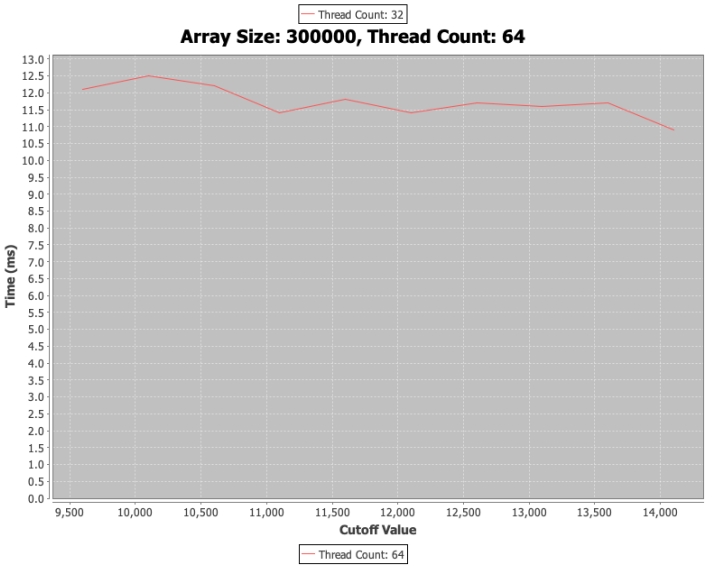












Based on these observations we can see that the number of threads for the above array size would be- 8