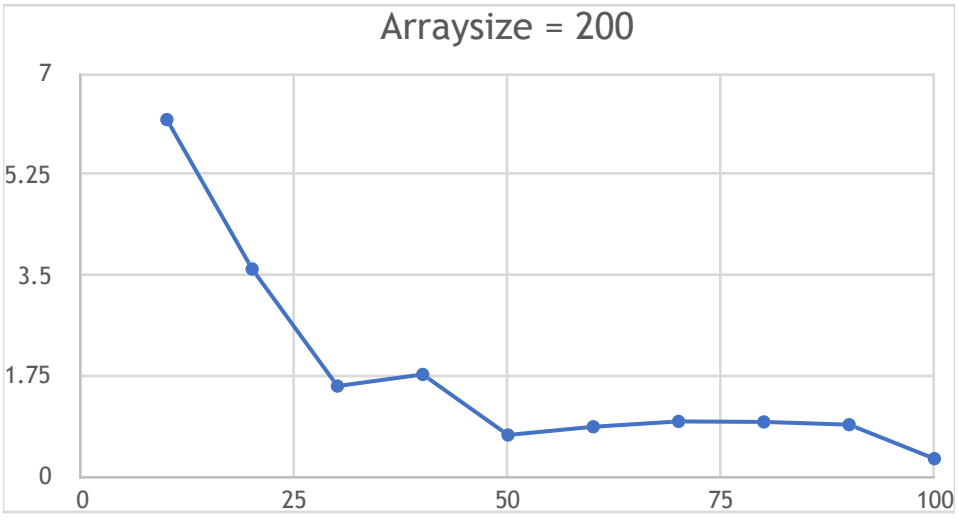
Assignment 4
Changrong Chen 001276880
Tiancheng Lin 001839357

In this assignment, we want to design a parallel sort method and then analyze two different scheme for deciding whether to sort in parallel: the cutoff and maximum recursion depth or number of available threads.

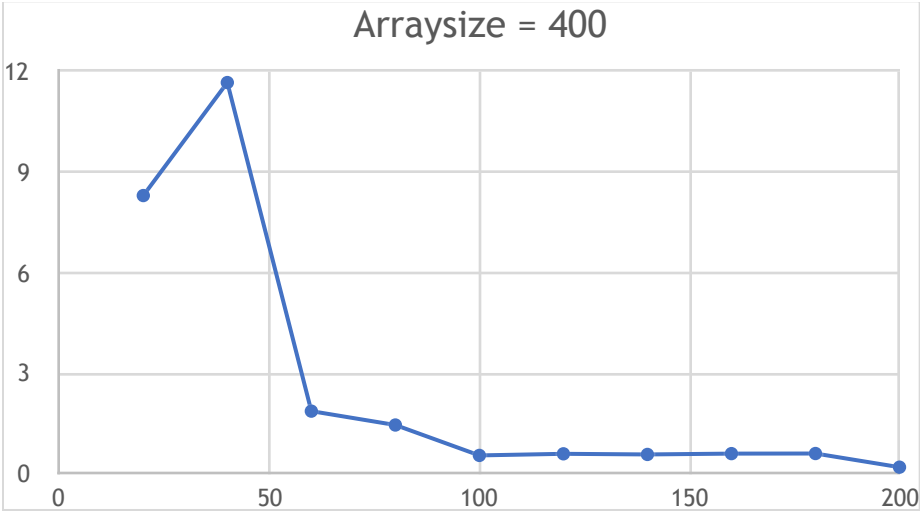
First, when there is one thread, we tried different values of cutoff in different array size. When array size equals 200,We could see that when the value of cutoff is larger than array size/4, the cost is similar, when this value become smaller, the cost will increase dramatically. So, we can have the conclusion that the best value of cutoff is N/4, where N is the size of the array.

```
Array size is: 200Cutoff is: 10, sort time is: 3.328671 ms, the thread is: 1
Array size is: 200Cutoff is: 20, sort time is: 1.974813 ms, the thread is: 1
Array size is: 200Cutoff is: 30, sort time is: 0.804291 ms, the thread is: 1
Array size is: 200Cutoff is: 40, sort time is: 0.849655 ms, the thread is: 1
Array size is: 200Cutoff is: 50, sort time is: 0.27876 ms, the thread is: 1
Array size is: 200Cutoff is: 60, sort time is: 0.336567 ms, the thread is: 1
Array size is: 200Cutoff is: 70, sort time is: 0.255288 ms, the thread is: 1
Array size is: 200Cutoff is: 80, sort time is: 0.262438 ms, the thread is: 1
Array size is: 200Cutoff is: 90, sort time is: 0.389654 ms, the thread is: 1
Array size is: 200Cutoff is: 100, sort time is: 0.17607 ms, the thread is: 1
```

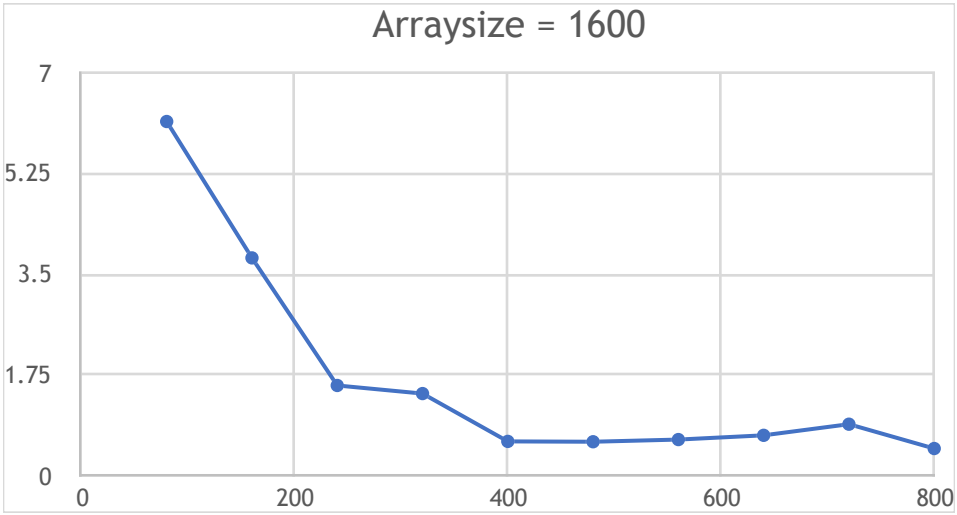
200	10	6.204765
200	20	3.607787
200	30	1.572972
200	40	1.774505
200	50	0.722396
200	60	0.865053
200	70	0.957953
200	80	0.949256
200	90	0.901587
200	100	0.310384



Array size	Cutoff	Time(ms)
400	20	8.307683
400	40	11.675354
400	60	1.883212
400	80	1.469741
400	100	0.563732
400	120	0.611439
400	140	0.592674
400	160	0.61727
400	180	0.620251
400	200	0.213117

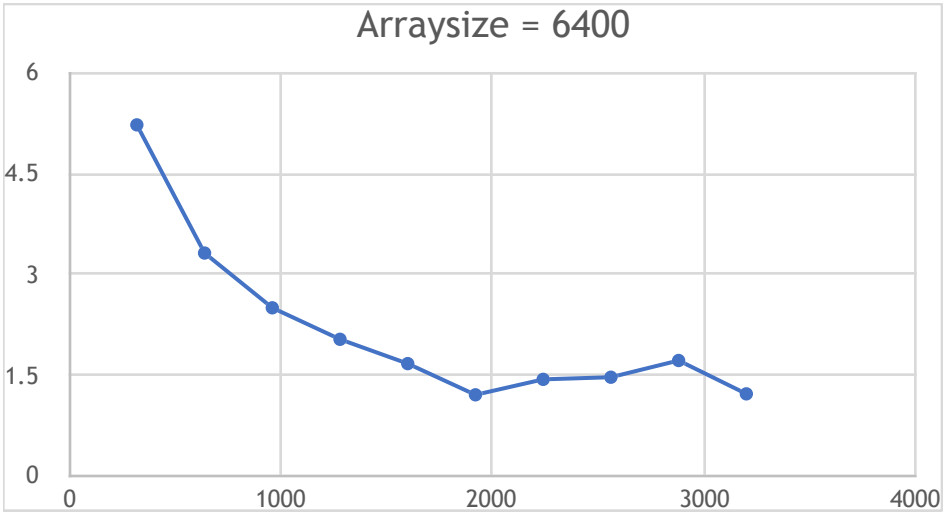


Array size	Cutoff	Time(ms)
1600	80	6.155396
1600	160	3.784481
1600	240	1.566711
1600	320	1.425941
1600	400	0.594542
1600	480	0.588528
1600	560	0.626356
1600	640	0.700164
1600	720	0.893376
1600	800	0.470746

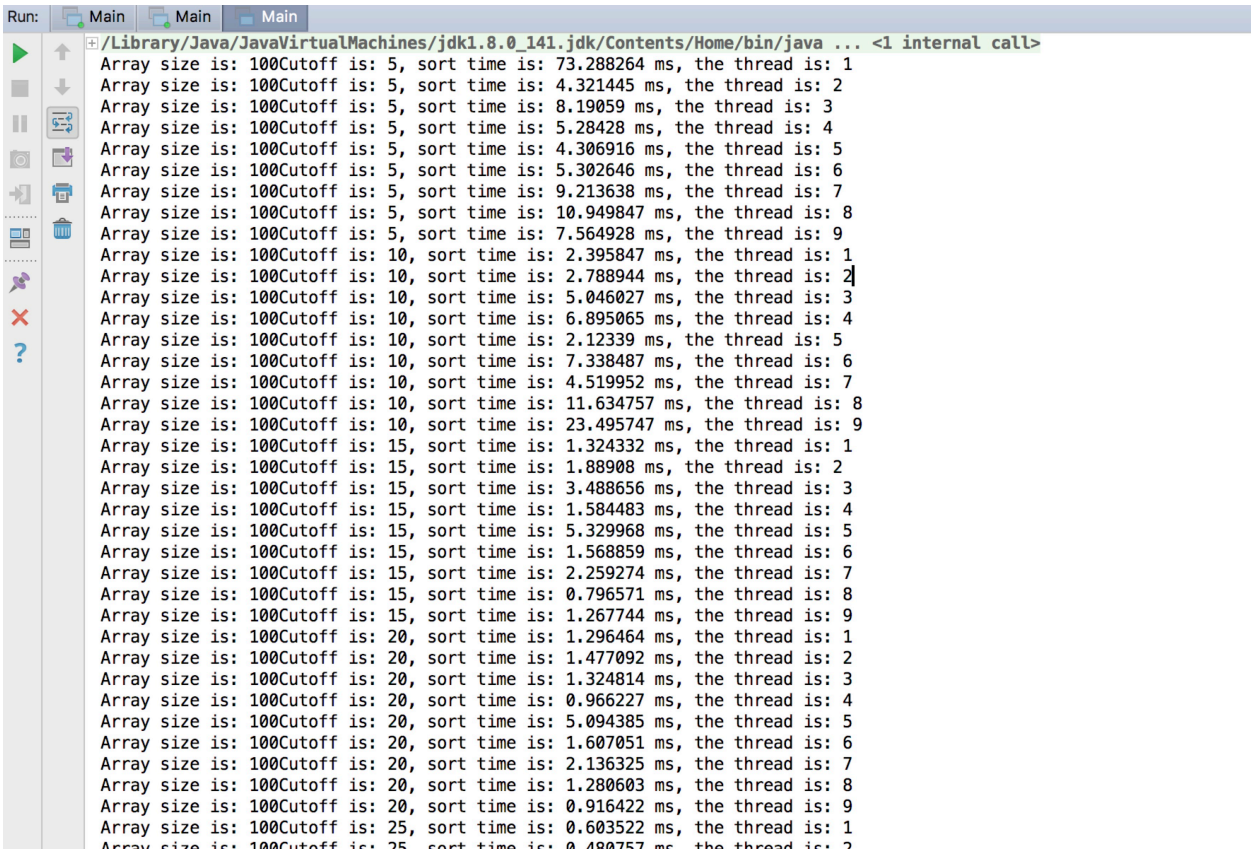


In order to prove this assumption, we change the value of N to 400, 1600 and 6400, we could also see that the time will decrease dramatically when cutoff = N/4.

Array size	Cutoff	Time(ms)
6400	320	5.223046
6400	640	3.31316
6400	960	2.497383
6400	1280	2.027977
6400	1600	1.665518
6400	1920	1.200826
6400	2240	1.431027
6400	2560	1.462851
6400	2880	1.712171
6400	3200	1.21513



For the recursion depth or the number of available threads.



The value of cut off can decide whether the parallel sort method can stop dividing array into two part and sorting then. If the value of cutoff is smaller, the recursion depth will become bigger, and the number of thread will become more. Base on ur assumption, when the value of cutoff is smaller than N/4, where N is the size of the array, which means that the recursion depth exceed log 4 = 2 and the number of thread exceed 4, the cost will increase dramatically. So, the recursion depth should not exceed 2 and the maximum number of thread is 4. And the best value of cutoff is N/4.

Main():

```
public class Main {

    public static void main(String[] args) {


        //if (args.length>0) ParSort.cutoff = Integer.parseInt(args[0]);
        //Scanner scan=new Scanner(System.in);
        //ParSort.cutoff=scan.nextInt();
        Random random = new Random( seed: 0l);



        int[] sortNumber = {100, 200, 400, 800, 1600, 3200, 6400, 12800};
        for (int n = 0; n < sortNumber.length; n++) {
            for (int x = 1; x <= 9; x++) {
            //for (int n = 0; n < sortNumber.length; n++) {
                for (int m = sortNumber[n] / 20; m <= sortNumber[n] / 2; m += sortNumber[n] / 20) {
                    //for (int x = 1; x <= 9; x++) {
                    ParSort.cutoff = m;
                    //ParSort.cutoff = sortNumber[n]/4;

                    ParSort.defineThread(x);
                    int[] array = new int[sortNumber[n]];
                    for (int i = 0; i < array.length; i++) array[i] = random.nextInt( bound: 10000);
                    long time1 = System.nanoTime();
                    ParSort.sort(array, from: 0, to: array.length - 1);
                    long time2 = System.nanoTime();
                    //for (int i : array) System.out.println(i);
                    //if (array[0]==11) System.out.println("Success!");
                    System.out.print("Array size is: " + sortNumber[n] + "Cutoff is: " + ParSort.cutoff +
                            ", sort time is: " + (time2 - time1)/1000000.0 + " ms"
                            + ", the thread is: " + ParSort.executor.getParallelism());
                    System.out.println();

                }
            }
        }
```

Sort:

```
public static void sort(int[] array, int from, int to) {

    int size = to - from + 1;
    if (size <= cutoff) Arrays.sort(array, from,  toIndex: to+1);
    else {
        int mid = from + (to - from) / 2;
        CompletableFuture<int[]> parsort1 = parsort(array, from, mid); // TODO implement me
        CompletableFuture<int[]> parsort2 = parsort(array,  from: mid + 1, to); // TODO implement me
        CompletableFuture<int[]> parsort = parsort1.
                thenCombine(parsort2, (xs1, xs2) -> {
                    int[] result = new int[xs1.length + xs2.length];
                    int a = 0, b = 0;// TODO implement me
                    for (int i = 0; i < xs1.length + xs2.length; i++) {
                        if (a > xs1.length-1) {
                            result[i] = xs2[b++];
                        }
                        else if (b < xs2.length && xs2[b] <= xs1[a]){
                            result[i] = xs2[b++];
                        }
                        else {
                            result[i] = xs1[a++];
                        }
                    }
                    return result;
                }).whenComplete((result, throwable) -> {
                    System.arraycopy(result,  srcPos: 0, array,  destPos: 0, array.length);
        });
          parsort.whenComplete((result, throwable) -> {
              System.arraycopy(result, 0, array, 0, array.length);
          }); //
        parsort.join();
    }

}
```

```
private static CompletableFuture<int[]> parsort(int[] array, int from, int to) {

    return CompletableFuture.supplyAsync(
            () -> {
                int[] result = new int[to - from + 1];
                int l = from;
                int i = 0;
                System.arraycopy(array, l, result, i, result.length);
                sort(result,  from: 0,  to: result.length - 1);// TODO implement me
                return result;
            }, executor);
}
```

Unit test: