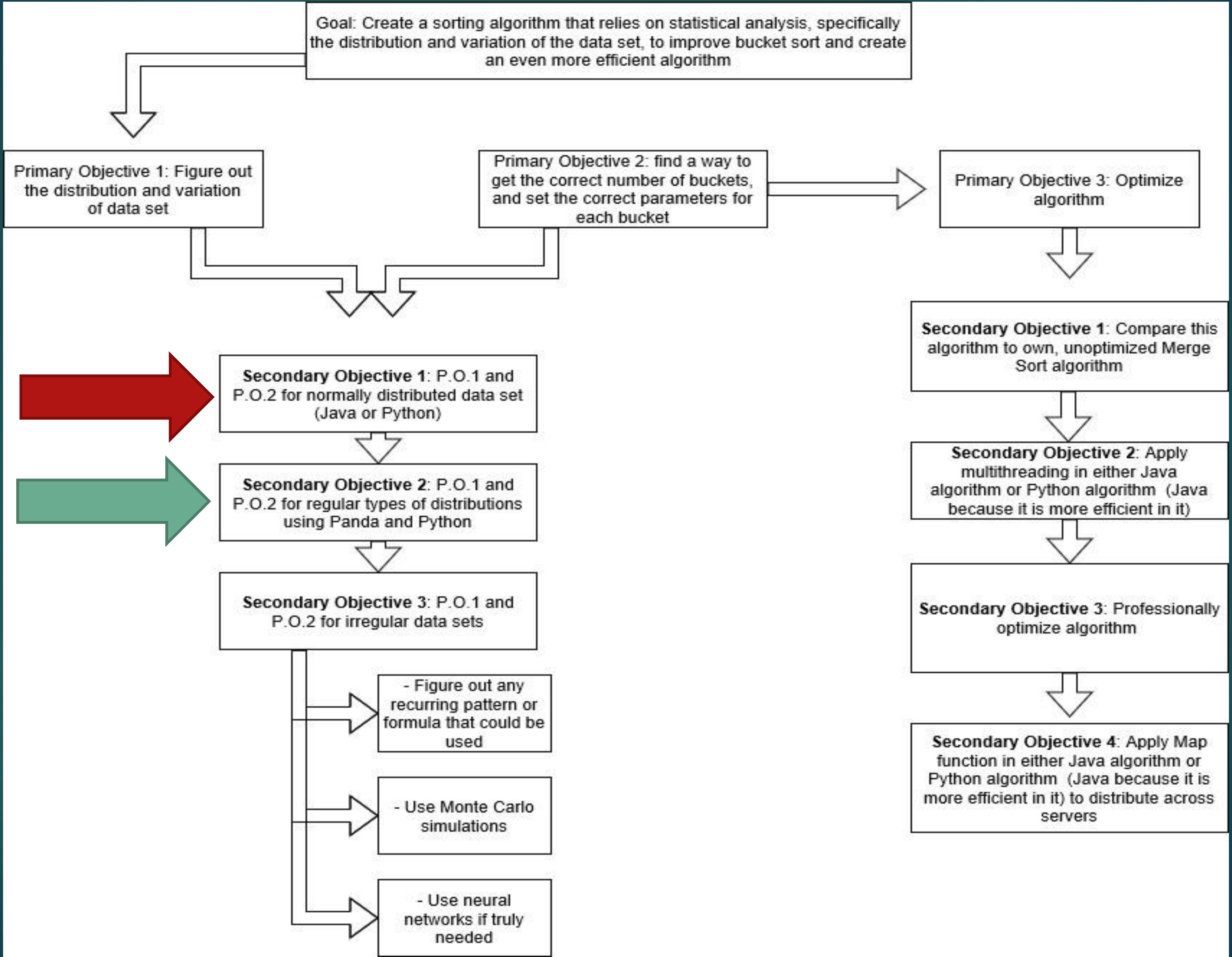
Progress Report 1 – 10/13/2017

GOAL: CREATE A SORTING ALGORITHM THAT RELIES ON THE DISTRIBUTION AND VARIATION OF THE DATA SET TO CREATE THE BUCKET PARAMETERS, IN AN EFFORT TO MAKE SORTING OF LARGE DATA SETS OF VARIOUS DISTRIBUTIONS MORE EFFICIENT


Goal: Create a sorting algorithm that relies on statistical analysis, specifically the distribution and variation of the data set, to improve bucket sort and create an even more efficient algorithm
Primary Objective 1: Figure out the distribution and variation of data set
Primary Objective 2: find a way to get the correct number of buckets, and set the correct parameters for each bucket
Primary Objective 3: Optimize algorithm
Secondary Objective 1: P.O.1 and P.O.2 for normally distributed data set (Java or Python)
Secondary Objective 2: P.O.1 and P.O.2 for regular types of distributions using Panda and Python
Secondary Objective 3: P.O.1 and P.O.2 for irregular data sets
- Figure out any recurring pattern or formula that could be used
- Use Monte Carlo simulations
- Use neural networks if truly needed
Secondary Objective 1: Compare this algorithm to own, unoptimized Merge Sort algorithm
Secondary Objective 2: Apply multithreading in either Java algorithm or Python algorithm (Java because it is more efficient in it)
Secondary Objective 3: Professionally optimize algorithm
Secondary Objective 4: Apply Map function in either Java algorithm or Python algorithm (Java because it is more efficient in it) to distribute across servers

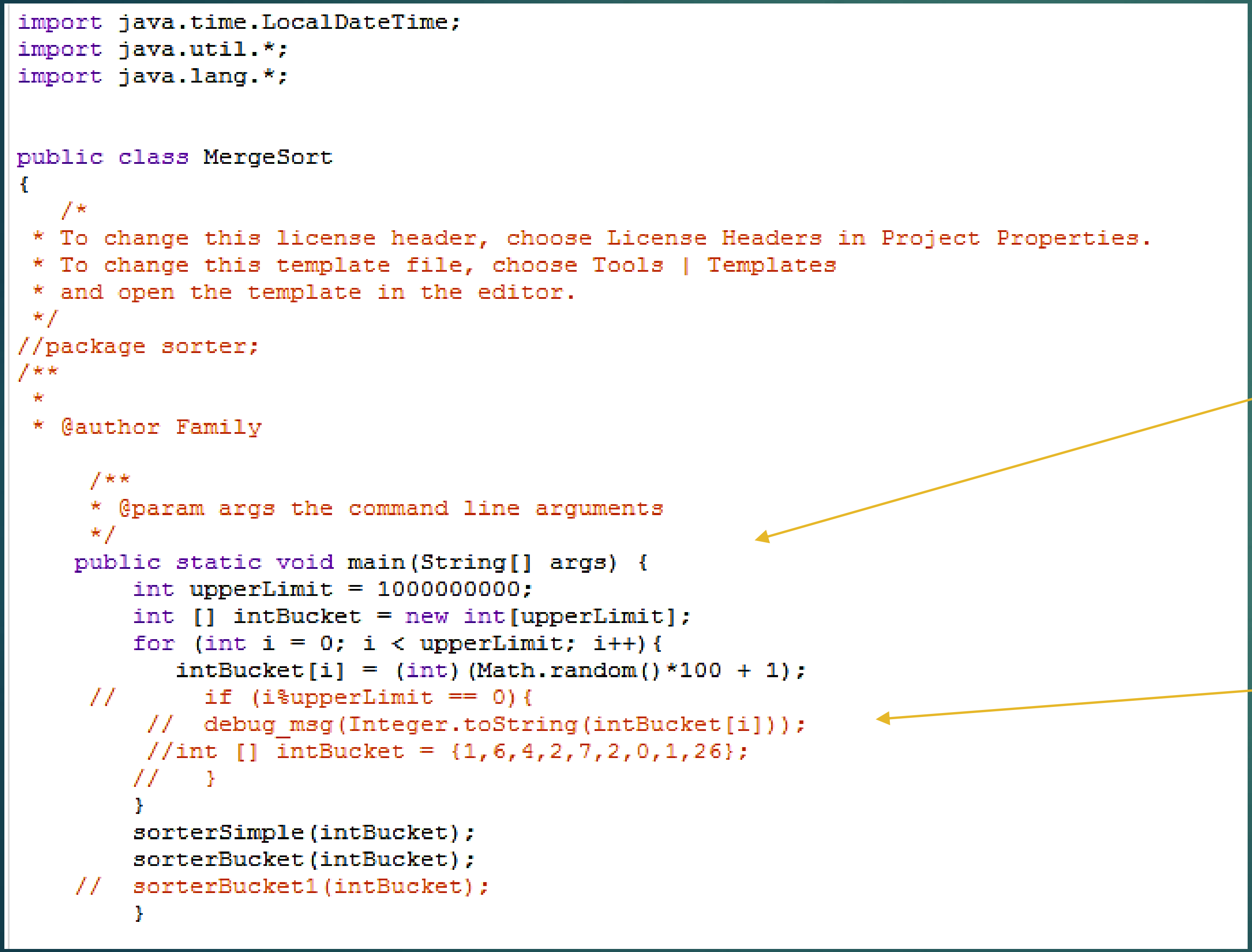
Goal in Depth

- Create a sorting algorithm that
  - Calculates customized parameters of *n* number of buckets based on elements in data set
    - Done by figuring out mean, variance value, and number of elements
  - At most 1000 elements in each bucket (buckets themselves are sorted), and each should have equal number of elements regardless
  - Should be able to sort data sets that follow any type of distribution
  - Incorporate multithreading to speed up algorithm
  - Professionally optimize algorithm

What Has Been Done Thus Far

- Created backbone frame of algorithm in Java
  - Technical errors in preliminary code took time to figure out
    - Still examining why fixing the errors the way I did resolved the issue and still proved to be more efficient
- Collected data when compared against Parallel Sort/Merge Sort
- Transferring code into Python and learning Panda
- Finalized Background Research
- Experimental Design with Stats

Backbone Code in Java – Main

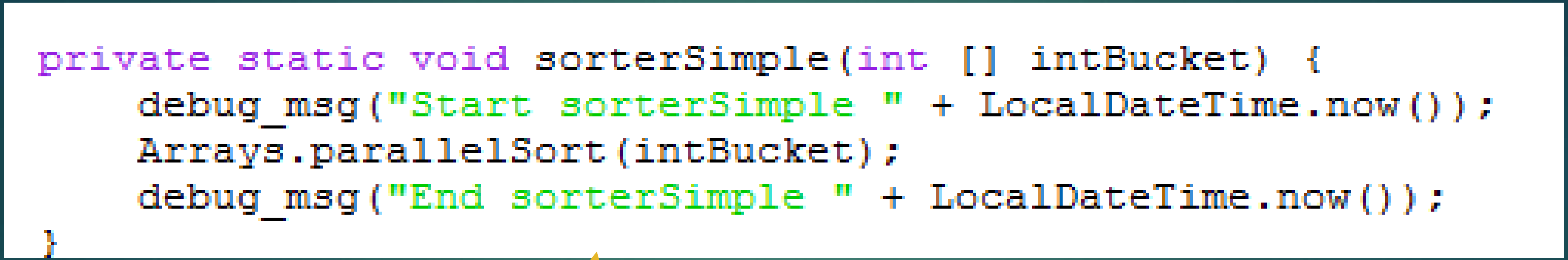


```java
import java.time.LocalDateTime;
import java.util.*;
import java.lang.*;


public class MergeSort
{
   /*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
//package sorter;
/**
 *
 * @author Family

     /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int upperLimit = 1000000000;
        int [] intBucket = new int[upperLimit];
        for (int i = 0; i < upperLimit; i++){
           intBucket[i] = (int)(Math.random()*100 + 1);
     //      if (i%upperLimit == 0){
         //  debug_msg(Integer.toString(intBucket[i]));
         //int [] intBucket = {1,6,4,2,7,2,0,1,26};
        //   }
        }
        sorterSimple(intBucket);
        sorterBucket(intBucket);
    //  sorterBucket1(intBucket);
        }
```

- ❖ Main method that creates the randomized array
- ❖ Outputs parallel sort and new stat sort run times

Checking if algorithm actually sorts numbers – it does.

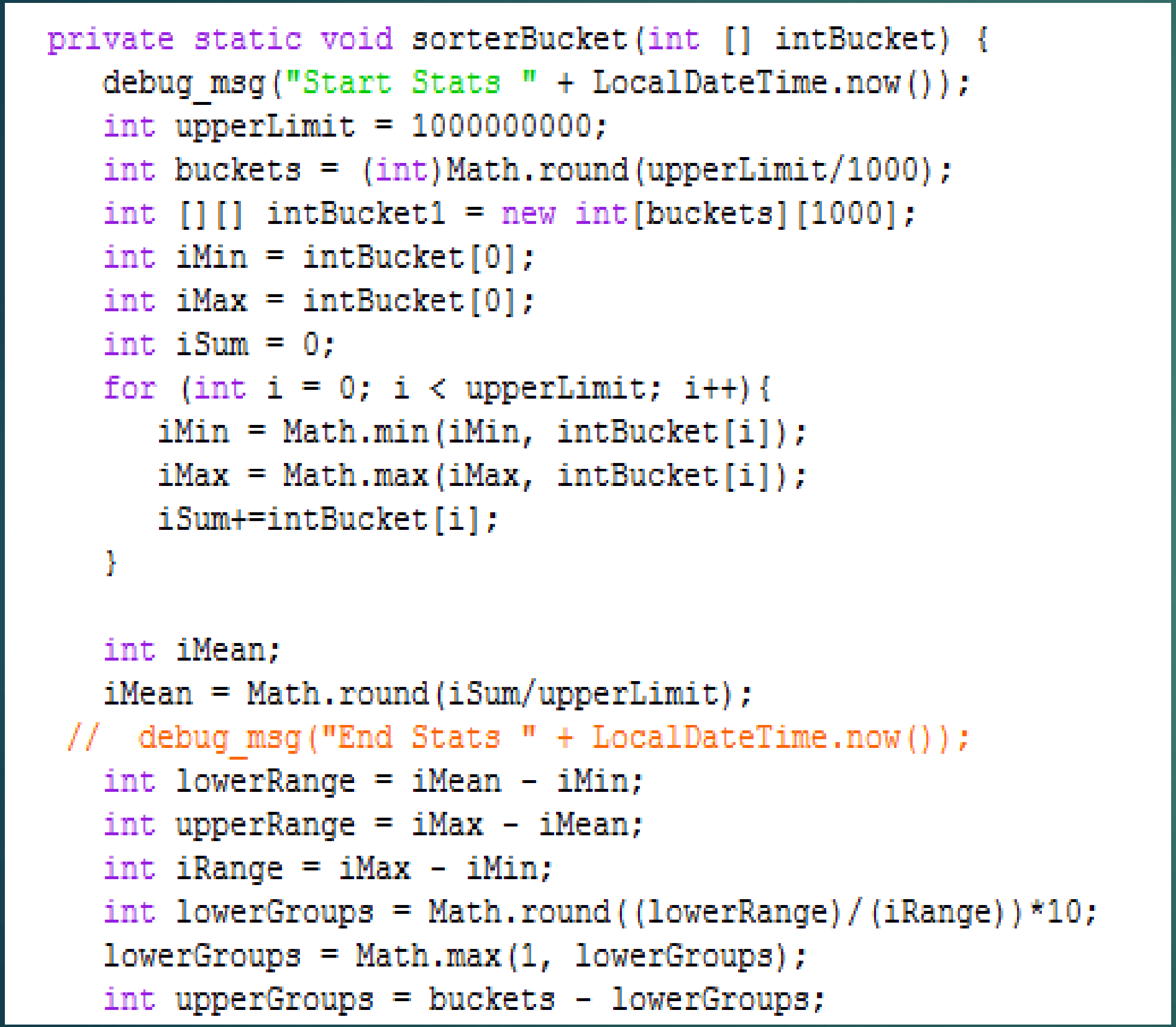
Backbone in Java – Parallel Sort



```
private static void sorterSimple(int [] intBucket) {
    debug_msg("Start sorterSimple " + LocalDateTime.now());
    Arrays.parallelSort(intBucket);
    debug_msg("End sorterSimple " + LocalDateTime.now());
}
```

Getting run time of Parallel Sort – this is a Java merge-sort like sorting algorithm used to sort arrays by threading

Backbone in Java – Stat Sort

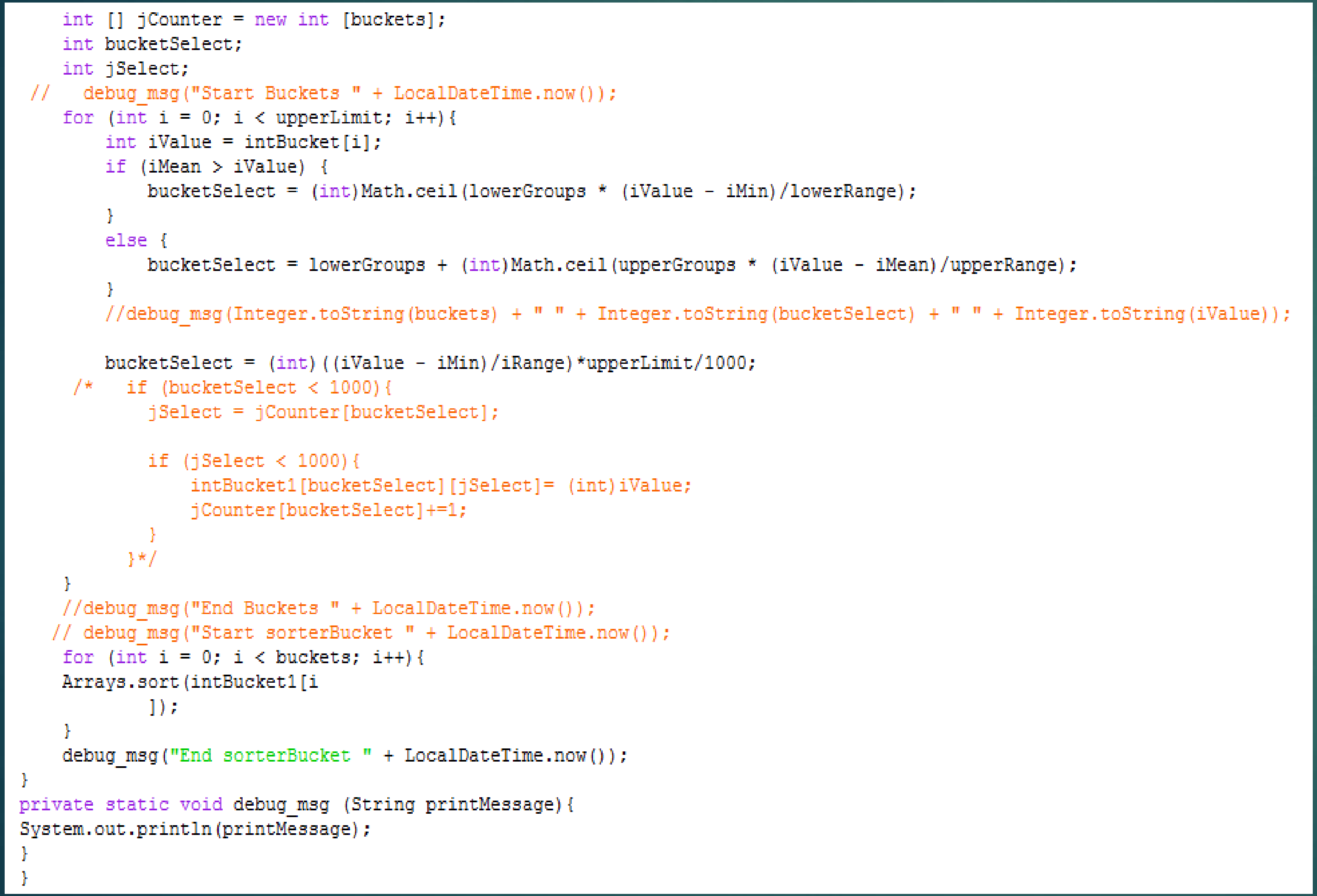


```java
private static void sorterBucket(int [] intBucket) {
   debug_msg("Start Stats " + LocalDateTime.now());
   int upperLimit = 1000000000;
   int buckets = (int)Math.round(upperLimit/1000);
   int [][] intBucket1 = new int[buckets][1000];
   int iMin = intBucket[0];
   int iMax = intBucket[0];
   int iSum = 0;
   for (int i = 0; i < upperLimit; i++){
      iMin = Math.min(iMin, intBucket[i]);
      iMax = Math.max(iMax, intBucket[i]);
      iSum+=intBucket[i];
   }

   int iMean;
   iMean = Math.round(iSum/upperLimit);
 //  debug_msg("End Stats " + LocalDateTime.now());
   int lowerRange = iMean - iMin;
   int upperRange = iMax - iMean;
   int iRange = iMax - iMin;
   int lowerGroups = Math.round((lowerRange)/(iRange))*10;
   lowerGroups = Math.max(1, lowerGroups);
   int upperGroups = buckets - lowerGroups;
```

- ❑ This is the method that performs the “stat” sort
- ❑ This part of the method is calculating the
  - ❑ Max and min
  - ❑ mean
  - ❑ The lower range (min to mean)
  - ❑ The upper range (mean to max)
  - ❑ The number buckets for the two ranges

The comment was added to see how long it took to perform the stats – about half of a millisecond

Backbone in Java – Buckets

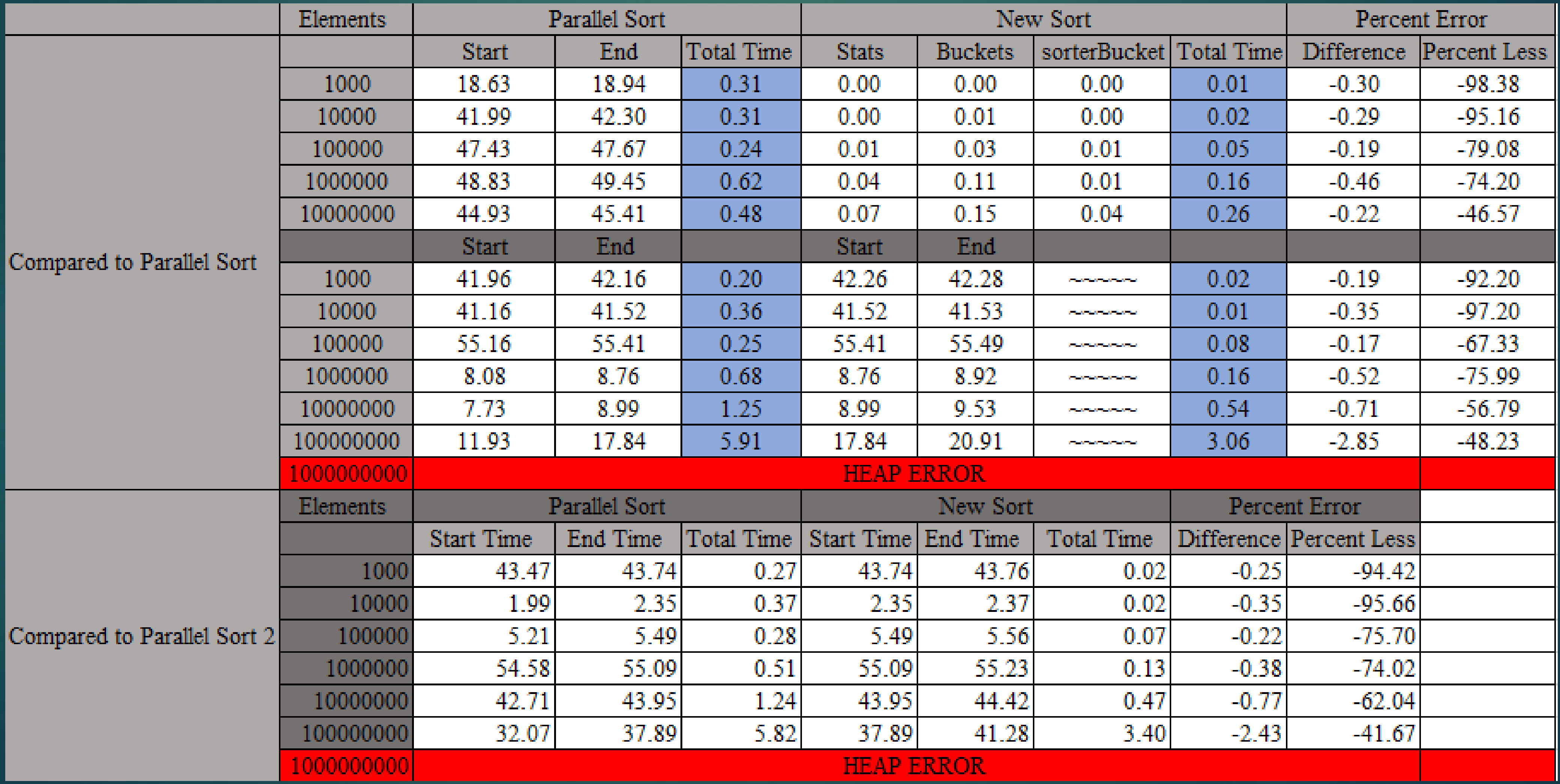


```
    int [] jCounter = new int [buckets];
    int bucketSelect;
    int jSelect;
 //   debug_msg("Start Buckets " + LocalDateTime.now());
    for (int i = 0; i < upperLimit; i++){
        int iValue = intBucket[i];
        if (iMean > iValue) {
            bucketSelect = (int)Math.ceil(lowerGroups * (iValue - iMin)/lowerRange);
        }
        else {
            bucketSelect = lowerGroups + (int)Math.ceil(upperGroups * (iValue - iMean)/upperRange);
        }
        //debug_msg(Integer.toString(buckets) + " " + Integer.toString(bucketSelect) + " " + Integer.toString(iValue));

        bucketSelect = (int)((iValue - iMin)/iRange)*upperLimit/1000;
     /*   if (bucketSelect < 1000){
            jSelect = jCounter[bucketSelect];

            if (jSelect < 1000){
                intBucket1[bucketSelect][jSelect]= (int)iValue;
                jCounter[bucketSelect]+=1;
            }
          }*/
    }
    //debug_msg("End Buckets " + LocalDateTime.now());
   // debug_msg("Start sorterBucket " + LocalDateTime.now());
    for (int i = 0; i < buckets; i++){
    Arrays.sort(intBucket1[i
            ]);
    }
    debug_msg("End sorterBucket " + LocalDateTime.now());
}
private static void debug_msg (String printMessage){
System.out.println(printMessage);
}
}
```

- This is the part of the method in which the elements are being sorted into their respective buckets
- Inputs the end time of the sort which is used to calculated run time

This commented section was what was causing errors. This is supposed to make sure there are only 1000 elements in each bucket. However, the algorithm still sorted the data sets (1 million +) faster than parallel sort

Data When Compared to Parallel Sort



	Elements	Parallel Sort			New Sort				Percent Error	
		Start	End	Total Time	Stats	Buckets	sorterBucket	Total Time	Difference	Percent Less
Compared to Parallel Sort	1000	18.63	18.94	0.31	0.00	0.00	0.00	0.01	-0.30	-98.38
	10000	41.99	42.30	0.31	0.00	0.01	0.00	0.02	-0.29	-95.16
	100000	47.43	47.67	0.24	0.01	0.03	0.01	0.05	-0.19	-79.08
	1000000	48.83	49.45	0.62	0.04	0.11	0.01	0.16	-0.46	-74.20
	10000000	44.93	45.41	0.48	0.07	0.15	0.04	0.26	-0.22	-46.57
		Start	End		Start	End				
	1000	41.96	42.16	0.20	42.26	42.28	~~~~~~	0.02	-0.19	-92.20
	10000	41.16	41.52	0.36	41.52	41.53	~~~~~~	0.01	-0.35	-97.20
	100000	55.16	55.41	0.25	55.41	55.49	~~~~~~	0.08	-0.17	-67.33
	1000000	8.08	8.76	0.68	8.76	8.92	~~~~~~	0.16	-0.52	-75.99
	10000000	7.73	8.99	1.25	8.99	9.53	~~~~~~	0.54	-0.71	-56.79
	100000000	11.93	17.84	5.91	17.84	20.91	~~~~~~	3.06	-2.85	-48.23
	1000000000	HEAP ERROR								

	Elements	Parallel Sort			New Sort			Percent Error	
		Start Time	End Time	Total Time	Start Time	End Time	Total Time	Difference	Percent Less
Compared to Parallel Sort 2	1000	43.47	43.74	0.27	43.74	43.76	0.02	-0.25	-94.42
	10000	1.99	2.35	0.37	2.35	2.37	0.02	-0.35	-95.66
	100000	5.21	5.49	0.28	5.49	5.56	0.07	-0.22	-75.70
	1000000	54.58	55.09	0.51	55.09	55.23	0.13	-0.38	-74.02
	10000000	42.71	43.95	1.24	43.95	44.42	0.47	-0.77	-62.04
	100000000	32.07	37.89	5.82	37.89	41.28	3.40	-2.43	-41.67
	1000000000	HEAP ERROR							

What the Data Tells

- The new algorithm is more efficient than the optimized Parallel Sort/Merge Sort
- However, the efficiency decreases by about 10% each time the element number increases by a factor of 10 after 10,000 elements have been inputted
- This may be due to the high standard error
  - The extra time it takes to print out these statements will affect the run times
- Considering this is compared to an optimized algorithm, my algorithm will most likely be more efficient than this sort when optimized
- This also means that commenting out the 1000 element restriction did not affect the run times by a great amount

Goals Until The Next Progress Report

- Transfer algorithm into Python – next two weeks
- Learn how to implement Panda and how Panda works– next four weeks
- Use Panda in algorithm and check whether it will sort the data sets – whole of November, a bit into December
- Implement Panda-like method into Java algorithm – November and December