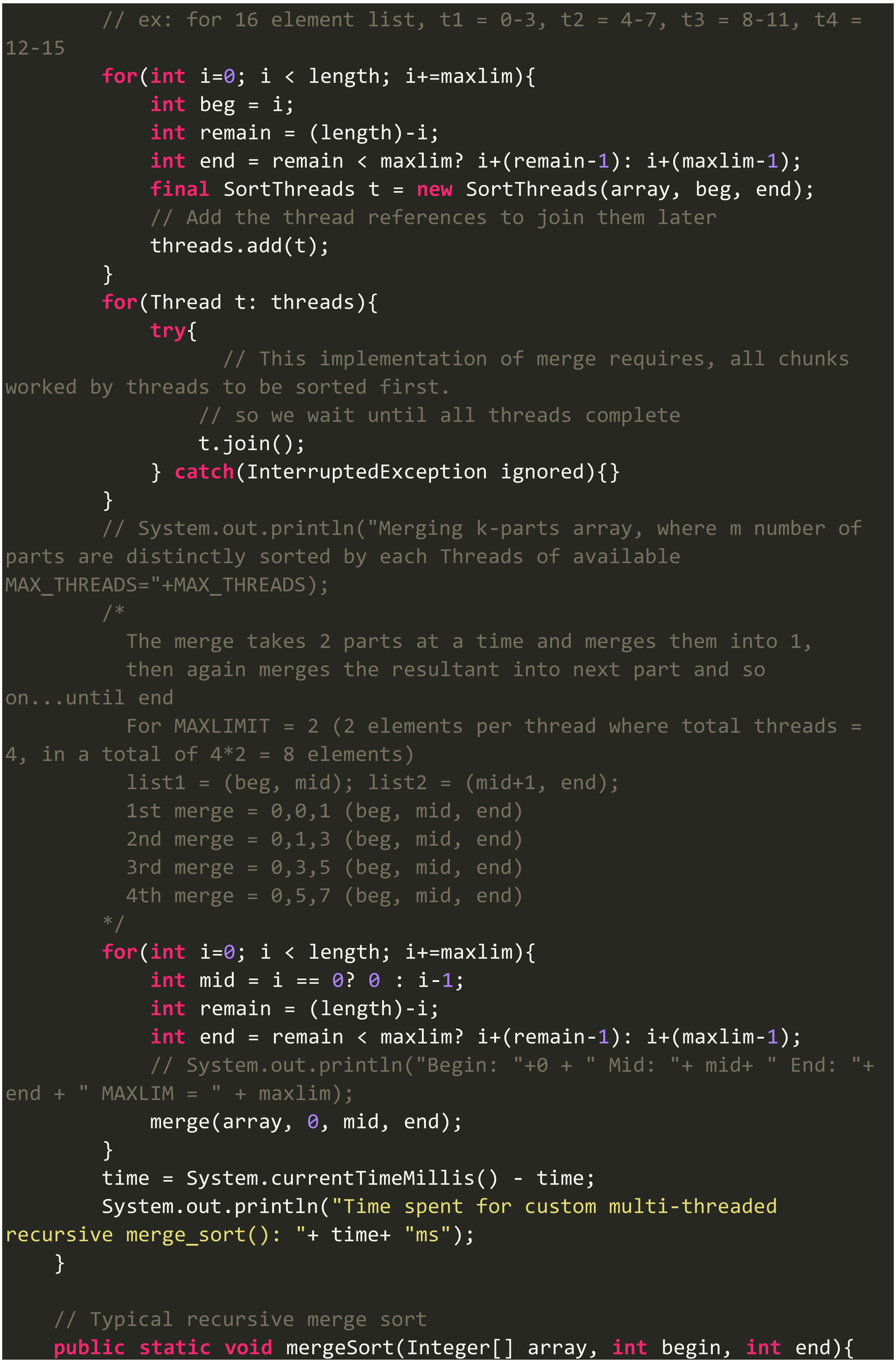
MERGE SORT USING MULTI-THREADING

```
Name:Abhang Rushikesh
Roll.NO:BE-A30

// Java Program to implement merge sort using
// multi-threading
import java.lang.System;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

class MergeSort{

      // Assuming system has 4 logical processors
    private static final int MAX_THREADS = 4;

    // Custom Thread class with constructors
    private static class SortThreads extends Thread{
        SortThreads(Integer[] array, int begin, int end){
            super(()->{
                MergeSort.mergeSort(array, begin, end);
            });
            this.start();
        }
    }

      // Perform Threaded merge sort
    public static void threadedSort(Integer[] array){
          // For performance - get current time in millis before starting
        long time = System.currentTimeMillis();
        final int length = array.length;
        // Workload per thread (chunk_of_data) =
total_elements/core_count
        // if the no of elements exactly go into no of available threads,
        // then divide work equally,
        // else if some remainder is present, then assume we have
(actual_threads-1) available workers
        // and assign the remaining elements to be worked upon by the
remaining 1 actual thread.
        boolean exact = length%MAX_THREADS == 0;
        int maxlim = exact? length/MAX_THREADS: length/(MAX_THREADS-1);
        // if workload is less and no more than 1 thread is required for
work, then assign all to 1 thread
        maxlim = maxlim < MAX_THREADS? MAX_THREADS : maxlim;
        // To keep track of threads
        final ArrayList<SortThreads> threads = new ArrayList<>();
        // Since each thread is independent to work on its assigned
chunk,
        // spawn threads and assign their working index ranges
```



```
        // ex: for 16 element list, t1 = 0-3, t2 = 4-7, t3 = 8-11, t4 =
12-15
        for(int i=0; i < length; i+=maxlim){
            int beg = i;
            int remain = (length)-i;
            int end = remain < maxlim? i+(remain-1): i+(maxlim-1);
            final SortThreads t = new SortThreads(array, beg, end);
            // Add the thread references to join them later
            threads.add(t);
        }
        for(Thread t: threads){
            try{
                  // This implementation of merge requires, all chunks
worked by threads to be sorted first.
                // so we wait until all threads complete
                t.join();
            } catch(InterruptedException ignored){}
        }
        // System.out.println("Merging k-parts array, where m number of
parts are distinctly sorted by each Threads of available
MAX_THREADS="+MAX_THREADS);
        /*
          The merge takes 2 parts at a time and merges them into 1,
          then again merges the resultant into next part and so
on...until end
          For MAXLIMIT = 2 (2 elements per thread where total threads =
4, in a total of 4*2 = 8 elements)
          list1 = (beg, mid); list2 = (mid+1, end);
          1st merge = 0,0,1 (beg, mid, end)
          2nd merge = 0,1,3 (beg, mid, end)
          3rd merge = 0,3,5 (beg, mid, end)
          4th merge = 0,5,7 (beg, mid, end)
        */
        for(int i=0; i < length; i+=maxlim){
            int mid = i == 0? 0 : i-1;
            int remain = (length)-i;
            int end = remain < maxlim? i+(remain-1): i+(maxlim-1);
            // System.out.println("Begin: "+0 + " Mid: "+ mid+ " End: "+
end + " MAXLIM = " + maxlim);
            merge(array, 0, mid, end);
        }
        time = System.currentTimeMillis() - time;
        System.out.println("Time spent for custom multi-threaded
recursive merge_sort(): "+ time+ "ms");
    }

    // Typical recursive merge sort
    public static void mergeSort(Integer[] array, int begin, int end){
        if (begin<end){
            int mid = (begin+end)/2;
            mergeSort(array, begin, mid);
            mergeSort(array, mid+1, end);
            merge(array, begin, mid, end);
        }
    }

    //Typical 2-way merge
    public static void merge(Integer[] array, int begin, int mid, int
end){
        Integer[] temp = new Integer[(end-begin)+1];

        int i = begin, j = mid+1;
        int k = 0;

        // Add elements from first half or second half based on whichever
is lower,
        // do until one of the list is exhausted and no more direct one-
to-one comparison could be made
        while(i<=mid && j<=end){
            if (array[i] <= array[j]){
                temp[k] = array[i];
                i+=1;
            }else{
                temp[k] = array[j];
                j+=1;
            }
            k+=1;
        }

        // Add remaining elements to temp array from first half that are
left over
        while(i<=mid){
            temp[k] = array[i];
            i+=1; k+=1;
        }

        // Add remaining elements to temp array from second half that are
left over
        while(j<=end){
            temp[k] = array[j];
            j+=1; k+=1;
        }

        for(i=begin, k=0; i<=end; i++,k++){
            array[i] = temp[k];
        }
    }
}

class Driver{
    // Array Size
    private static Random random = new Random();
    private static final int size = random.nextInt(100);
    private static final Integer list[] = new Integer[size];
    // Fill the initial array with random elements within range
    static {
      for(int i=0; i<size; i++){
        // add a +ve offset to the generated random number and subtract
same offset
        // from total so that the number shifts towards negative side by
the offset.
        // ex: if random_num = 10, then (10+100)-100 => -10
        list[i] = random.nextInt(size+(size-1))-(size-1);
      }
    }
    // Test the sorting methods performance
    public static void main(String[] args){
      System.out.print("Input = [");
      for (Integer each: list)
        System.out.print(each+", ");
      System.out.print("] \n" +"Input.length = " + list.length + '\n');

      // Test standard Arrays.sort() method
      Integer[] arr1 = Arrays.copyOf(list, list.length);
      long t = System.currentTimeMillis();
      Arrays.sort(arr1, (a,b)->a>b? 1: a==b? 0: -1);
      t = System.currentTimeMillis() - t;
      System.out.println("Time spent for system based Arrays.sort(): " +
t + "ms");

      // Test custom single-threaded merge sort (recursive merge)
implementation
      Integer[] arr2 = Arrays.copyOf(list, list.length);
      t = System.currentTimeMillis();
      MergeSort.mergeSort(arr2, 0, arr2.length-1);
      t = System.currentTimeMillis() - t;
      System.out.println("Time spent for custom single threaded recursive
merge_sort(): " + t + "ms");

      // Test custom (multi-threaded) merge sort (recursive merge)
implementation
      Integer[] arr = Arrays.copyOf(list, list.length);
      MergeSort.threadedSort(arr);
      System.out.print("Output = [");
      for (Integer each: arr)
        System.out.print(each+", ");
      System.out.print("]\n");
    }
}
```

```
Output:
Sorted Array: 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93


Time Taken: 0.001023
```