Project 1: Combining Quicksort with Insertionsort

Revanth Malay (ITCS-6114-091)

Student Id: 800916718

Problem Statement:

One way to improve the performance of QuickSort is to switch to InsertionSort when a subfile has <= M elements instead of recursively calling itself. Implement a recursive QuickSort with a cutoff to InsertionSort for subfiles with M or less elements. Empirically determine the value of M for which it performs fewest key comparisons on inputs of 60000 random natural numbers less than K for K = 10, 100, 1000, 10000, 100000, 1000000. Does the optimal value M depend on K? Prepare a project report to present your approach and finding. The source code listing should be documented and attached to your project report.

Introduction:

Quicksort is one of the fastest sorting algorithms and works very well with large number of inputs whereas insertion sort is also one among the fastest algorithms which work very well for small input values. Quicksort algorithm uses divide and conquer concept such that initially it divides the array into two sub files depending on the pivot element chosen. The two subfiles are now recursively called and are implemented in quicksort again. This continues till the array is sorted completely. This creates a run time of O(n logn), the n because we have to do n-1 comparisons on each iteration and log n because we have to divide the list/array logn times. However, the worst case of quicksort is O(n^2).

To improve the performance of quicksort in worst case we are now using insertion sort inside the quicksort algorithm such that we determine a threshold value and call insertion sort algorithm when threshold is <= array elements.

This project is concerned with improving the performance of quicksort because for an array with few elements we are using insertion sort and then passing the sorted array to quick sort for final sorting. The process is further explained below with code, results and observations.

Code:

1. ***Quicksort code:***

The code below is written and compiled in JAVA language using NetBeans IDE 8.0.2 as a compiler which can be obtained from www.netbeans.org.

```
// Quicksort implemented in Java

public static void QuickSort(int A[],int start,int end){

                        if(start<end){

                int pivot=partition(A,start,end); // Calls out for Partition procedure.

                QuickSort(A,start,pivot-1,m); // Recursive call to Quicksort()

                QuickSort(A,pivot+1,end,m); // Recursive call to Quicksort()

                        }

// PARTITION OF QUICKSORT
```

```
	public static int partition(int A[], int p, int r) {
		int x=A[r];
		int temp;
		int i=p-1;
		for(int j=p;j<r;j++){
			if(A[j]<=x){
				hcount++;
				i=i+1;
				temp=A[j];
				A[j]=A[i];
				A[i]=temp;
			}
		}
		temp=A[i+1];
		A[i+1]=A[r];
		A[r]=temp;
		return i+1;
	}
```

The above code successfully sorts when an array elements is passed into the method.

Insertion Sort:

```
//Insertion sort code implemented in Java
public static void InsertionSort(int Array[], int p, int r){
   for(int j=p+1;j<=r;j++){
	int i=j-1;
	int key = Array[j];
	while(i>(p-1) && Array[i]>key){
		hcount++;
		Array[i+1]=Array[i];
		i=i-1;
	}
```

```
          Array[i+1]=key;
    }
}
```

The above code successfully sorts when an array elements is passed into the method.

Combining Quicksort with Insertion Sort (Project Code)

```
// ********************** Code Starts from Here ****************************
// NOTE: Increasing the input size may lead to errors ad system may get crashed This code is
written for input size with 60000 and less.
// NOTE: This code is initialized with input array of 60000; range of random numbers as 10;
threshold value from 1000 to 20000 with 1000 incremented at each time;
package improvequick;
import java.util.Random;  // package with class random for generating random numbers;
class Sorting {
public static int hcount; // variable for total comparisions
public static void InsertionSort(int Array[], int p, int r){  // Insertion Sort Method starts from
here
for(int j=p+1;j<=r;j++){
int i=j-1;
int key = Array[j];
while(i>(p-1) && Array[i]>key){
hcount++;
Array[i+1]=Array[i];
i=i-1;
}
Array[i+1]=key;
}
} // InsertionSort is done.
          public static void QuickSort(int A[],int start,int end,int m){ // Quicksort is starting

if((start<end)&&(end-start)>m && m!=0){  // Condition for checking m value
int pivot=partition(A,start,end); // Calls out for Partition procedure.
QuickSort(A,start,pivot-1,m); // Recursive call to Quicksort()
QuickSort(A,pivot+1,end,m); // Recursive call to Quicksort()

}

else{

InsertionSort(A,start,end); // Calling Insertionsort method

}

}

// Partition Procedure starts

public static int partition(int A[], int p, int r) {

int x=A[r];

int temp;

int i=p-1;

for(int j=p;j<r;j++){

if(A[j]<=x){

hcount++;

i=i+1;

temp=A[j];

A[j]=A[i];

A[i]=temp;

}

}

temp=A[i+1];

A[i+1]=A[r];

A[r]=temp;

return i+1;

}

}


public class ImproveQuick{ // main class ***NOTE: initialize the file name with this
classname as classname.java

public static void main(String[] args){ //main method

Random gen=new Random(); // creating object for random class
```

```
int primary[]=new int[60010]; // input array with size 60000 ** Size can be changed **

int secondary[]=new int[60010]; // another array with size 60000 ** Size can be changed **

for(int i=1; i<=60000; i++){ // for loop to load array with random numbers.

primary[i]=gen.nextInt(10);  // loading random numbers to input array with range 100. ** Range can be changed from 10-10000000

secondary[i]=primary[i]; // loading numbers from primary to secondary

        }

for(int w=1000;w<20000;w=w+1000){ // *** Note : this loop is used to determine the threshold value, initialize the variable w, change the condition if necessary and increment the w value according to the random range. So that Threhold value can be determined.

Sorting.QuickSort(primary,1,60000,w); // calling quicksort method

System.out.println(" Total Comparisions are   "  + Sorting.hcount + "   m value " + w); // diplaying the value of comparisons

Sorting.hcount=0; // Setting count to zero

System.arraycopy(secondary, 1, primary, 1, 59999); // reloading the elements from secondary to primary array

}

        }

}

// ********************** code ends here *******************
```

Description Of Code:

The code is written with 60000 random numbers as input given to the array. The random numbers range can be varied from 10-1000000 i.e., in the code manually which is mentioned in the comments. The input to the array can also be changed manually in the code as indicated in the comments. Increasing the input size might crash the compiler since the quick sort is called recursively which leads to Stackoverflow error. For the best output of code the input array can be reduced or can be made with 60000 inputs. This code determines the threshold value indicated as m. For threshold value > array length we call quick sort else the control calls insertion sort. The value of m has been determined empirically and the results for different range of k values can be seen below. The code uses two arrays (primary, secondary) where one array (primary) is used to store the random numbers initially and passes the array to respective sortings. The secondary array is used to store the random numbers from primary array initially and it is used to reload back the primary array with random numbers after determining the m value at each case. We can’t run the program for each time since the random numbers are not the same when the code is compiled again. The variable “hcount” is used to determine the total comparisons in both insertion sort and quicksort for every m value. This hcount and m value are plotted in a graph which gives the threshold value.

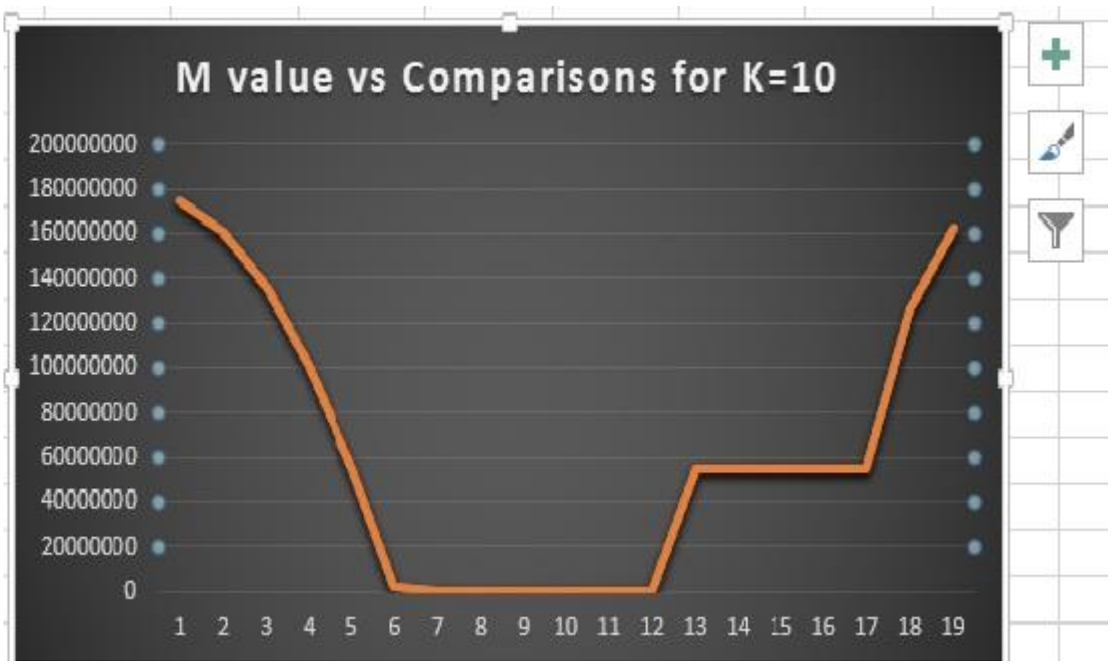
Analysis Of Code:

The code determines m value with total comparisons for given range through which the threshold value can be determined. From the analysis, the threshold value is inversely proportional to range of random numbers. If we increase the range of random numbers the threshold value gets decreases and vice versa. For smaller values of input range the quick sort gets into worst complexity because with less range the quicksort in some cases gets an array with sorted elements which increases the time complexity to O(n logn).

Observations And Results:

Output case1: For k=10, input array size=60000, m value from 1000 to 19000 as shown below

```
Total count for each m value

hcount = 175092690 ; m value = 1000

hcount = 160117537 ; m value = 2000

hcount = 135112537 ; m value = 3000

hcount = 100107537 ; m value = 4000

hcount = 55102537 ; m value = 5000

hcount = 1702125 ; m value = 6000

hcount = 216167 ; m value = 7000

hcount = 216167 ; m value = 8000

hcount = 216167 ; m value = 9000

hcount = 216167 ; m value = 10000

hcount = 216167 ; m value = 11000

hcount = 216167 ; m value = 12000

hcount = 54560584 ; m value = 13000

hcount = 54560584 ; m value = 14000

hcount = 54560584 ; m value = 15000

hcount = 54560584 ; m value =16000

hcount = 54560584  ; m value = 17000

hcount = 126087927 ; m value = 18000

hcount = 162511712 ; m value = 19000
```

Graph: 

X-axis: M value

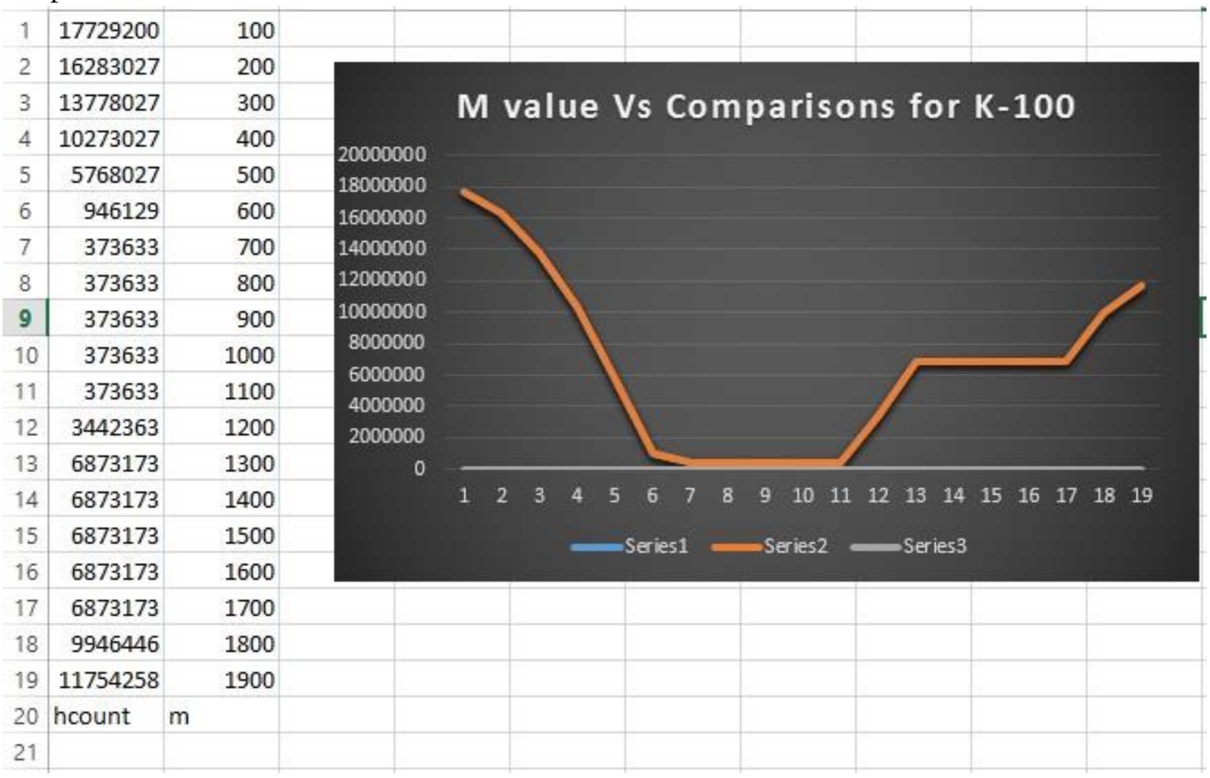
Y-axis: Total Comparisons.

Result: The threshold value of m is 9000 on an avg.

Output Case 2:

For k=100, input array size=60000, m value from 100 to 1900 as shown below

Graph:



1	17729200	100
2	16283027	200
3	13778027	300
4	10273027	400
5	5768027	500
6	946129	600
7	373633	700
8	373633	800
9	373633	900
10	373633	1000
11	373633	1100
12	3442363	1200
13	6873173	1300
14	6873173	1400
15	6873173	1500
16	6873173	1600
17	6873173	1700
18	9946446	1800
19	11754258	1900
20	hcount	m
21		

X-axis: M value

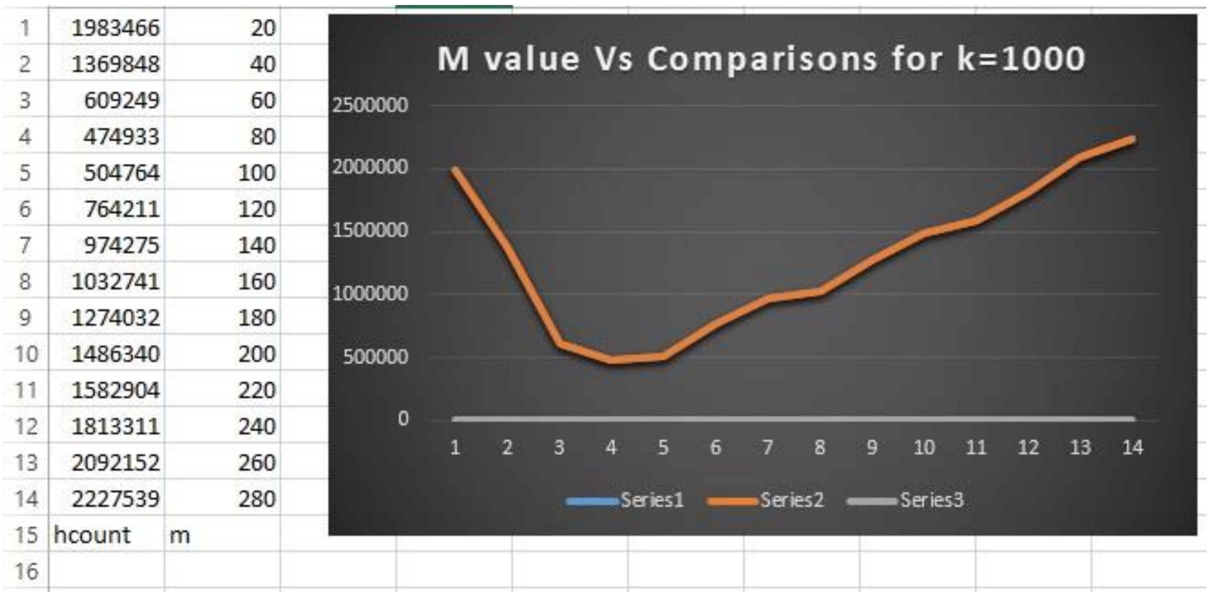
Y-axis: Total Comparisons.

Result: The threshold value of m is 900 on an avg.

Output Case 3:

For k=1000, input array size=60000, m value from 1000 to 20000 as shown below

Graph:



1	1983466	20
2	1369848	40
3	609249	60
4	474933	80
5	504764	100
6	764211	120
7	974275	140
8	1032741	160
9	1274032	180
10	1486340	200
11	1582904	220
12	1813311	240
13	2092152	260
14	2227539	280
15	hcount	m
16		

X-axis: M value

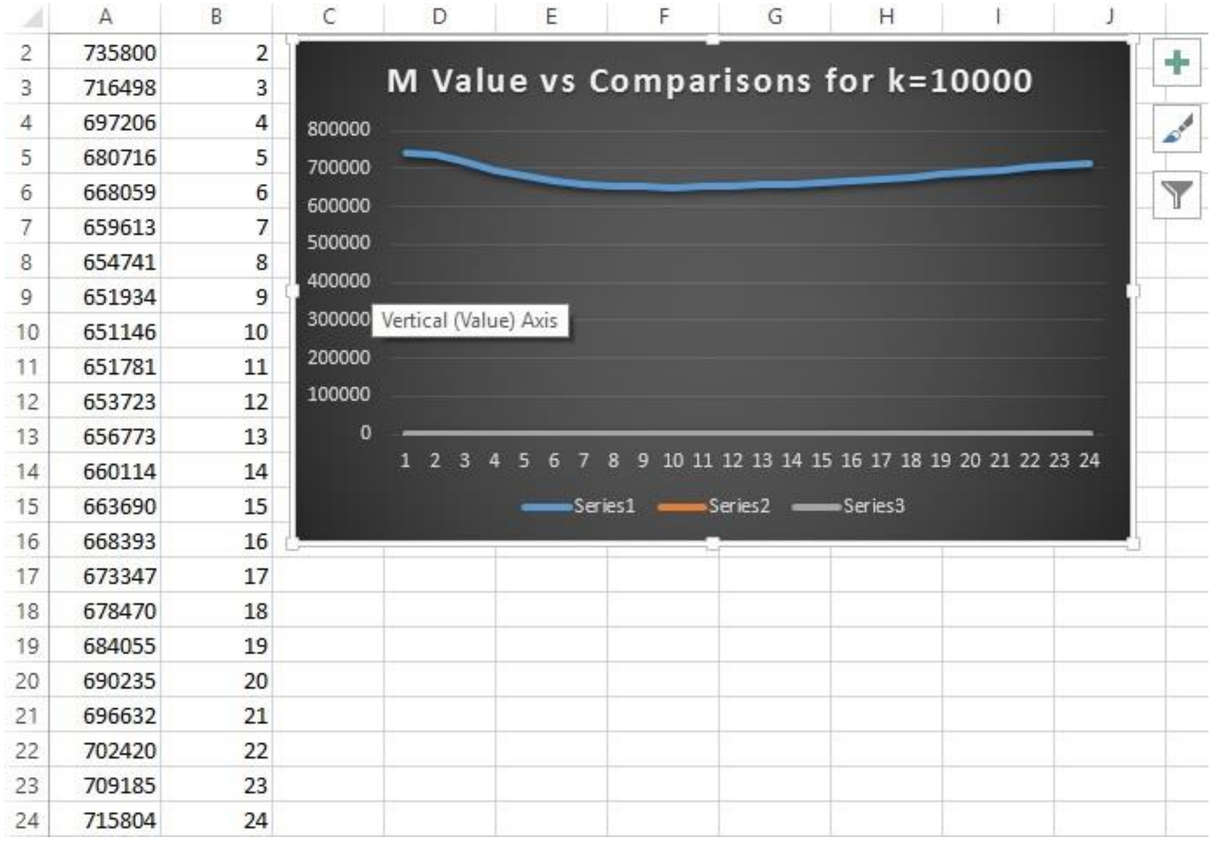
Y-axis: Total Comparisons.

Result: The threshold value of m is 80 on an avg.

Output Case 4:

For k=10000, input array size=60000, m value from 2 to 25 as shown below

Graph



	A	B
2	735800	2
3	716498	3
4	697206	4
5	680716	5
6	668059	6
7	659613	7
8	654741	8
9	651934	9
10	651146	10
11	651781	11
12	653723	12
13	656773	13
14	660114	14
15	663690	15
16	668393	16
17	673347	17
18	678470	18
19	684055	19
20	690235	20
21	696632	21
22	702420	22
23	709185	23
24	715804	24

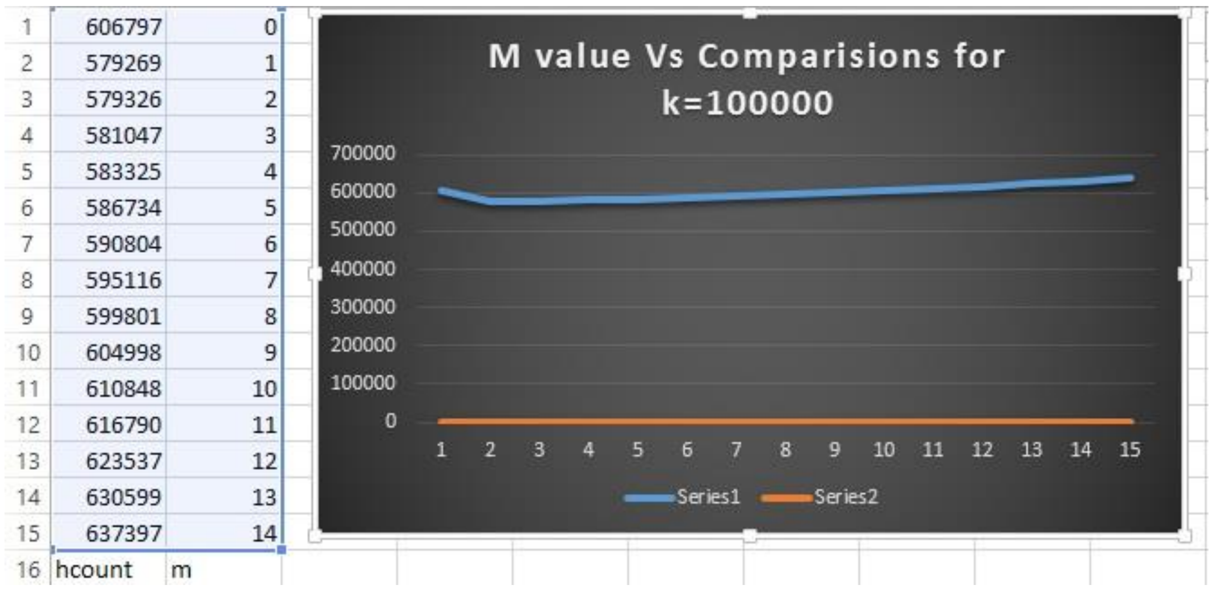
X-axis: M value

Y-axis: Total Comparisons.

Result: The threshold value of m is 10 on an avg.

Output case 5:

For k=100000, input array size=60000, m value from 0 to 14 as shown below



1	606797	0
2	579269	1
3	579326	2
4	581047	3
5	583325	4
6	586734	5
7	590804	6
8	595116	7
9	599801	8
10	604998	9
11	610848	10
12	616790	11
13	623537	12
14	630599	13
15	637397	14
16	hcount	m

X-axis: M value

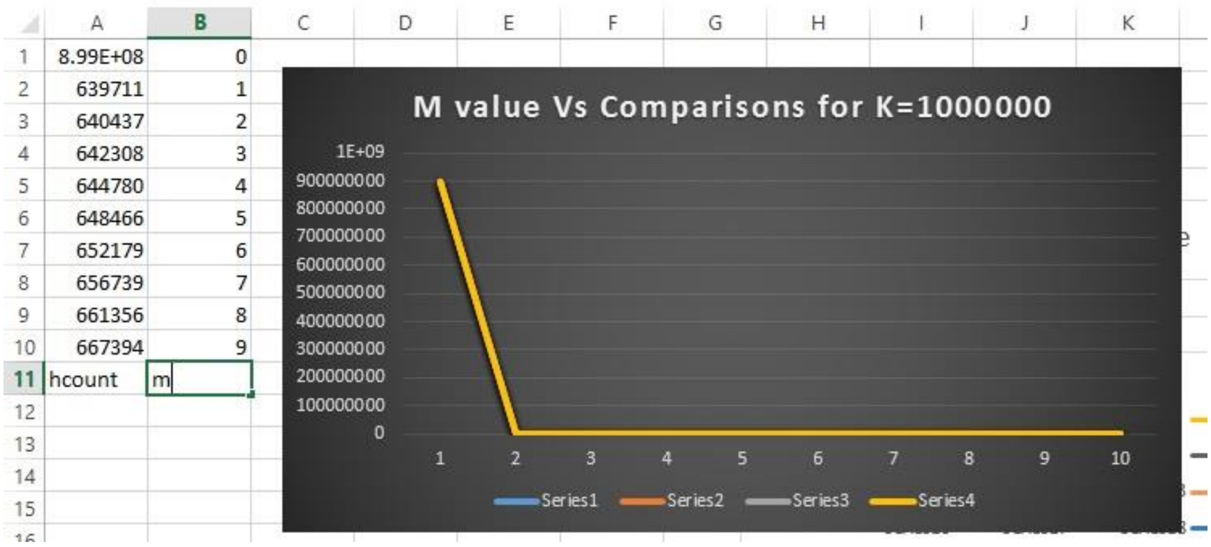
Y-axis: Total Comparisons.

Result: The threshold value of m is 4 on an avg.

Output Case 6:

For k=1000000, input array size=60000, m value from 0 to 14 as shown below

Graph:



	A	B
1	8.99E+08	0
2	639711	1
3	640437	2
4	642308	3
5	644780	4
6	648466	5
7	652179	6
8	656739	7
9	661356	8
10	667394	9
11	hcount	m
12		
13		
14		
15		

X-axis: M value

Y-axis: Total Comparisons.

Result: The threshold value of m is 3 on an avg.

***System Variables*:** Compiled and executed in Windows 10, 64 bit operating system, RAM: 4 GB, Hard Disk : 1 TB, Compiler : NetBeans IDE 8.0.2.

***Conclusion*:** From the above observation and results it can be determined as threshold value is inversely proportional to Range of random numbers. By the above the improvement of quick sort is done by adding insertion sort in it.