## **Lab 5 Report**

## **CS303L-L3 Algorithms and Data Structures**

## **Sam Lazrak**

## **Spring Semester 2018**

**Objectives:**

* Implement basic quick sort algorithm
* Implement quick sort using median of 3 partitioning
* Compare the performance of insertion sort, merge sort, heap sort, and quick sort.

**In-class Assignment:**

1. Implement a method to sort a given array using the basic Quicksort algorithm. Use the algorithm from the textbook.
2. Write a driver program to test the Quicksort algorithm for the file uploaded in the canvas.
3. Compare the performance of the Quicksort algorithm with three cases of input files: sorted, reversed sorted, and random. These files are provided in Canvas in the Quicksort Input Files folder.

**Source code for In-class Assignment:**

package lab5;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Arrays;

import java.util.Scanner;

public class lab5 {

public static void main(String[] args) throws FileNotFoundException {

String[] filename = {

"Input\_Random.txt",

"Input\_ReversedSorted.txt",

"Input\_Sorted.txt"

};

File[] files = new File[3];

files[0] = new File(filename[0]);

files[1] = new File(filename[1]);

files[2] = new File(filename[2]);

for (int i = 0; i <= 2; i++) {

int[] a = new int[1024];

Scanner scan = new Scanner(files[i]);

int j = 0;

while (scan.hasNextInt()) {

a[j] = scan.nextInt();

j++;

}

System.out.println("Unsorted Array: " + Arrays.toString(a));

long sortTime = System.nanoTime();

quickSort(a);

sortTime = System.nanoTime() - sortTime;

System.out.println("Sorted Array: " + Arrays.toString(a));

System.out.println("It took " + sortTime + " nanoseconds to sort file " + filename[i] + "\n");

}

}

public static void quickSort(int[] a) {

int p = 0;

int r = a.length - 1;

quickSort(a, p, r);

}

public static void quickSort(int[] a, int p, int r) {

if (p <= r) {

int q = partition(a, p, r);

quickSort(a, p, q - 1);

quickSort(a, q + 1, r);

}

}

private static int partition(int[] a, int p, int r) {

int x = a[r];

int i = p - 1;

for (int j = p; j <= r - 1; j++) {

if (a[j] <= x) {

i = i + 1;

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

int temp = a[i + 1];

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

a[r] = temp;

return i + 1;

}

}

**Output for In-class Assignment:**

The output was too large to put here. Included screenshot: [Just in case that image doesn't load properly here’s a link as well.](https://i.imgur.com/ePhzLZy.png)



**Homework Assignment:**

1. Compare the execution time of quick sort with the other sorting techniques we have already seen. Make sure you use the same array to compare the performance. Use a table or plot to summarize the results and document your observations and analysis in the report.
2. Use the Median of 3 partitioning algorithm (see below) to implement Quicksort. This algorithm chooses the pivot element as the median of the three elements namely: A[p], A[r], and A[(p+r)/2].
3. Test the program for the same array sizes and values. Compare the performance with the original merge sort implementation, plot the execution times, and document the analysis in your lab report.
4. Compare the performance of the Quicksort using median of 3 partitioning with the basic Quicksort algorithm using the input files located on Canvas in Quicksort Input Files folder.
5. Compare the execution time of Quicksort with the execution time of Insertion Sort, Merge Sort, and Heapsort. Make sure you use the same array to compare the performance. Use a table or plot to summarize the results and document your observations and analysis in the report. You can use the input files ranging from 100 to 1,000,000 or the input files ranging from 16 to 8192.

**Source code for Homework Assignment:**

package lab5;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Arrays;

import java.util.Scanner;

import lab5.InsertionSort;

import lab5.MergeSort;

import lab5.HeapSort;

public class QuickSort {

public static void main(String[] args) throws FileNotFoundException {

File[] files = new File[7];

files[0]= new File("input\_100.txt");

files[1]= new File("input\_1000.txt");

files[2]= new File("input\_5000.txt");

files[3]= new File("input\_10000.txt");

files[4]= new File("input\_50000.txt");

files[5]= new File("input\_100000.txt");

files[6]= new File("input\_500000.txt");

int[] size = {100, 1000, 5000, 10000, 50000, 100000};

for(int i = 0; i <= 6; i++) {

int length = size[i];

int[] array = new int[length];

Scanner scanner = new Scanner(files[i]);

int j = 0;

while(scanner.hasNextInt()) {

array[j] = scanner.nextInt();

j++;

}

int[] QuickMerge = Arrays.copyOf(array, array.length);

long QMsortTime = System.nanoTime();

medianQuickSort(QuickMerge);

QMsortTime = System.nanoTime() - QMsortTime;

int[] Quick = Arrays.copyOf(array, array.length);

long QsortTime = System.nanoTime();

quickSort(Quick, 0 , Quick.length -1);

QsortTime = System.nanoTime() - QsortTime;

int[] Heap = Arrays.copyOf(array, array.length);

long HsortTime = System.nanoTime();

HeapSort.heapSort(Heap, Heap.length -1);

HsortTime = System.nanoTime() - HsortTime;

int[] Insertion = Arrays.copyOf(array, array.length);

long IsortTime = System.nanoTime();

InsertionSort.insertionSort(Insertion);

IsortTime = System.nanoTime() - IsortTime;

int[] Merge = Arrays.copyOf(array, array.length);

long MsortTime = System.nanoTime();

MergeSort.mergeSort(Merge);

MsortTime = System.nanoTime() - MsortTime;

int[] InsertionMerge = Arrays.copyOf(array, array.length);

long IMsortTime = System.nanoTime();

MergeSort.mergeInsertionSort(InsertionMerge, size[i]/10);

IMsortTime = System.nanoTime() - IMsortTime;

System.out.println("Array Size: " + size[i] + "\n" + "Insertion Sort Time: " + IsortTime + "\n" + "Heap Sort Time: " + HsortTime + "\n" + "Merge Sort Time: " + MsortTime + "\n" +

"Merge Sort with Insertion Time: " + IMsortTime + "\n" + "Quick Sort Time: " + QsortTime + "\n" + "Quick Sort with Median Time: " + QMsortTime + "\n" );

}

}

public static void quickSort(int[] a, int p, int r){

if(p < r) {

int q = partition(a, p ,r);

quickSort(a, p, q-1);

quickSort(a, q+1, r);

} }

public static void medianQuickSort(int[] a){

int p = 0;

int r = a.length -1;

medianQuickSort(a, p, r);

}

public static void medianQuickSort(int[] a, int p, int r) {

int n = (r - p) + 1;

if(n <= 3) {

InsertionSort.insertionSort(a);

}

else {

int m = Median3(a, p, p + n/2, r);

int temp = a[m];

a[m] = a[p];

a[p] = temp;

int q = partition(a, p, r);

medianQuickSort(a, p, q-1);

medianQuickSort(a, q+1, r);

}

}

private static int Median3(int[] a, int p, int m, int r) {

if(a[p] > a[m]) {

int temp = a[p];

a[p] = a[m];

a[m] = temp;

}

if(a[p] > a[r]){

int temp = a[p];

a[p] = a[r];

a[r] = temp;

}

if(a[m] > a[r]) {

int temp = a[m];

a[m] = a[r];

a[r] = temp;

}

int temp = a[m];

a[m] = a[r-1];

a[r-1] = temp;

return a[r-1];

}

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

int pivot = a[r];

int i = p-1;

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

if(a[j] <= pivot){

i = i +1;

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

int temp = a[i+1];

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

a[r]= temp;

return i+1;

}

}

**Output for Homework Assignment:**

Array Size: 100

Insertion Sort Time: 67532

Heap Sort Time: 996220

Merge Sort Time: 811595

Merge Sort with Insertion Time: 35223

Quick Sort Time: 30156

Quick Sort with Median Time: 1123606

Array Size: 1000

Insertion Sort Time: 1837230

Heap Sort Time: 285756

Merge Sort Time: 639198

Merge Sort with Insertion Time: 50835

Quick Sort Time: 211781

Quick Sort with Median Time: 12500700

Array Size: 5000

Insertion Sort Time: 20536982

Heap Sort Time: 1527945

Merge Sort Time: 1031570

Merge Sort with Insertion Time: 84684

Quick Sort Time: 565429

Quick Sort with Median Time: 39157766

Array Size: 10000

Insertion Sort Time: 29779893

Heap Sort Time: 1235633

Merge Sort Time: 1194596

Merge Sort with Insertion Time: 134011

Quick Sort Time: 663477

Quick Sort with Median Time: 128329866

Array Size: 50000

Insertion Sort Time: 395198027

Heap Sort Time: 7102882

Merge Sort Time: 5739311

Merge Sort with Insertion Time: 510270

Quick Sort Time: 3824821

Quick Sort with Median Time: 2122074685

Array Size: 100000

Insertion Sort Time: 1562679501

Heap Sort Time: 12398587

Merge Sort Time: 11841510

Merge Sort with Insertion Time: 940538

Quick Sort Time: 10142725

Quick Sort with Median Time: 8641611362

**Analysis:**

Insertion sort has a running time of Θ(n2). Merge sort has a running time of Θ(n lg n ). Heap sort has a running time of O(n lg n ). From the data we can see that insertion sort takes the longest time and heap sort takes a bit longer than merge sort. Quick sort has a running time of O(n lg n ) and it is faster than heap and merge sort. Merge sort and heap sort achieve this upper bound in the worst case while quicksort achieves it on average. Quick sort using median of 3 partitioning takes longer than quicksort because choosing the median of three values is not necessarily a good pivot choice. Overall it seems as though quick sort is the one to use.