Jeffrey Pulinat

CS-323, SECTION: 11 (7912)

Lab#2

4/20/17

Due date: 4/20/17

**Instructions on Lab2.java**

All textfiles with given names "Num8.txt", "Num16.txt", "Num32.txt", etc must be in the same directory as Lab2.java

A textfile named Output.txt will be created and have the text output of the program

Compile Instructions:

Go to correct directory

javac Lab2.java

java Lab2

**Test Case Summary**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Average | Θ(n^2) | Θ(n+k) | Θ(n+k) | Θ(nk) |
| Worst | O(n^2) | O(n^2) | O(n+k) | O(nk) |
|  | Insertion Sort | Bucket Sort | Counting Sort | Radix Sort |
| Num8.txt | 14 | 9 | 8 | 8 |
| Num16.txt | 84 | 87 | 16 | 32 |
| Num32.txt | 249 | 222 | 32 | 64 |
| Num128.txt | 966 | 988 | 64 | 128 |
| Num128.txt | 4195 | 4193 | 128 | 384 |
| Num256.txt | 15372 | 15173 | 256 | 768 |

**Analysis**

**a) Discuss what your results mean regarding the theoretical run-time of the different algorithms.**

The Radix sort, counting sort and bucket sort, are integer based algorithms. Radix sort in theory is one of the fastest sorts. Radix creates a bucket for each digit like bucket that contains a growable list that may admit different keys. Radix Sort was a bit slower than counting sort in practice. Radix average and worst case was (nk). Bucket can be very fast since technically counting sort is its upper bound. My Bucket sort using insertion sort was probably the main factor in slowing it down. Bucket average case Θ(n+k) is the same as counting sort average case. Bucket sort worst case is O(n^2) which is what my data really mirrors. Bucket Sort was using Insertion Sort which also had the average and worst case of O(n^2). For counting sort average and worst was (n+k). This sort turned out to be my fastest sort.

**b) Do the sorts really take O(n) steps to run? c) Explain how you got your answer to this question.**

No the sorts take O(n^2), O(n+k) and O(nk). ). Insertions Sort is very similar too O(n^2). We can see this by taking n/2 inputs and count times for all the text files with n^2. In Counting Sort the runtime we get runtimes very similar to O(n+k) and Radix O(nk). There is some discrepancy when comparing the counts to the O run times because the O run times represent worst case running times.

**d) Which of the sorts takes the most steps? e) Why**?

Insertion Sort and Bucket Sort took the most steps because its average run time is O(n^2). BucketSort had very similar runtimes to Insertion sort but it looked like it was a pulling away when the data was larger. Bucket Sort average runtime is Θ(n+k).

**f) Under what circumstances might you prefer to use a particular one of the three linear sorts? g) Why?**

Theoretically Radix Sort is the quicker sort. However in my testing Counting Sort is shown to have the better runtime. I believe my implementation of Radix Sort is what really messed up the runtime.

**h) In general, which sort seems preferable? i) Why?**

Counting Sort seems preferable because it was easier to implement then Radix. Radix relied on a modified counting sort.

Counting Sort is a much faster than my other sorts.

**j) When might you prefer to use Quicksort instead of one of the linear-time sorts?**

Quick Sort is highly dependable on data randomness and pivot. Its cache performance is higher than other sorting algorithms. This is because of its in-place characteristic. There is no extra memory being used in quicksort.

**k) When might Insertion sort be the best sort of all to use?**

Insertion sort is faster for small n because there is no overhead extra overhead that would usually happen from recursive function calls. Insertion sort is stable and requires less memory.

**Source Code:**

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.\*;

public class Lab2 {

//global variable count

public static int count;

public static String Output = "";

public static boolean flagprint= false; //used for bigger textfiles

public static void main(String[] args) {

// Array of textfile names

String [] FileNames = { "Num8.txt", "Num16.txt", "Num32.txt", "Num64.txt", "Num128.txt","Num256.txt" };

String DataFile = "";

try{

//loops on all textfiles

for(int FileNamesIndex = 0; FileNamesIndex < FileNames.length; FileNamesIndex++){

DataFile = FileNames[FileNamesIndex];

FileReader fileReader = new FileReader(new File(DataFile));

BufferedReader bufferedReader = new BufferedReader(fileReader);

int ArraySize = Integer.parseInt(DataFile.replaceAll("[^0-9]", "")); //gets int from String

//declare all unsorted arrays

Integer []SetArray1 = new Integer[ArraySize];

int []SetArray2 = new int[ArraySize];

int []SetArray3 = new int[ArraySize];

int []SetArray4 = new int[ArraySize];

//fills all unsorted arrays

for(int i=0; i<SetArray1.length; ++i){

String aLine = bufferedReader.readLine();

SetArray1[i] = Integer.parseInt(aLine);

SetArray2[i] = Integer.parseInt(aLine);

SetArray3[i] = Integer.parseInt(aLine);

SetArray4[i] = Integer.parseInt(aLine);

}

fileReader.close();

//special print case when file 128 or greater

if(DataFile.equals("Num128.txt"))flagprint=true;

//Application of All Sorts and String outputs.

OutputMold(SetArray1, SetArray2, SetArray3, SetArray4, DataFile);

}

}catch (IOException e) {

e.printStackTrace();

}

}

//--------------------OutputMold-----------------

//Application of all Sorts. Mold/Stencil for how String should be printed

public static void OutputMold(Integer array1[], int array2[], int array3[], int array4[],String DataFile){

//Start of Files

Output = Output + "\n-------" + DataFile + "------------\n";

//Set count o zero before every sort

//INSERT

count = 0;

InsertionSort(array1);

Output = Output + "InsertionSort: " + count + "\n";

PrintArray(array1);

//BucketSort

count = 0;

int []BucketSorted = BucketSort(array2);

Output = Output + "\nBucketSort: " + count + "\n";

PrintArray(BucketSorted);

//CoutingSort

count = 0;

int x = array3.length;

int []temp = new int[x];

int []CountSorted = CountingSort(array3,temp,x);

Output = Output + "\nCountingSort: " + count + "\n";

PrintArray(CountSorted);

//RadixSort

count = 0;

RadixSort(array4);

Output = Output + "\nRadixSort: " + count + "\n";

PrintArray(array4);

System.out.println(Output);

try{

//Writes all Output to txtfile called output for easier read.

PrintWriter out = new PrintWriter( "Output.txt" );

out.println(Output );

out.close();

} catch (IOException e) {

e.printStackTrace();

}

}

//-----------------PrintArray(int [])----------------------

//Prints out the array for int arrays

public static void PrintArray(int array[]){

if(flagprint==true){

int copy[] = new int [50]; //wanted to use Array to String because its cleaner. Need to create copy array to print specific indexes.

int copyindex=0;

for(int i = 51 ;i <=100; i++){

copy[copyindex] = array[i];

copyindex++;

}

Output = Output + (Arrays.toString(copy));

}else

Output = Output + (Arrays.toString(array));

}

//-----------------PrintArray(Integer [])-----------------

//Prints out the array for Integer arrays

public static void PrintArray(Integer array[]){

if(flagprint==true){

int copy[] = new int [50]; //wanted to use Array to String because its cleaner. Need to create copy array to print specific

int copyindex=0;

for(int i = 51 ;i <=100; i++){

copy[copyindex] = array[i];

copyindex++;

}

Output = Output + (Arrays.toString(copy));

}else

Output = Output + (Arrays.toString(array));

}

//---------------------------CountingSort---------------------

//Parameters: Unsorted Array

//Returns Sorted Array

public static int[] CountingSort (int A[], int B[], int k){

int C[] = new int[k+1];

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

C[i] = 0;

}

for(int j= 1; j<k; j++){

C[A[j]] = C[A[j]] + 1;

// C[j] now contains the # of elements = to i.

}

for(int i = 1; i<=k; i++){

C[i] = C[i] + C[i-1];

//C[i] now contains the # of elements ≤ to i.

}

//Making the sorted array. B. Modified this for loop from psudeo code. Couldn't get psuedo to work.

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

count++; //cost && sort

B[C[A[i]]] = A[i];

C[A[i]] = C[A[i]] -1;

}

return B;

}

//---------------------------BucketSort--------------------

//Parameters: Unsorted Array

//Returns Sorted Array

public static int[] BucketSort(int[] A) {

int n = A.length;

HashMap<Integer, ArrayList> map = new HashMap<>(); //ArrayLists are easier to add too. Could of used arrays

int buckets = A.length - 2; //used 2 buckets less than n.

int divider = (int) Math.ceil(n + 1 / buckets); //my divider case

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

//insert A[i] into hashmap. Think of it like an array of arrays

//j represents bucket indexes.

int j = (int) Math.floor(A[i] / divider);

//no index j exists. create new arraylist and add A[i] to it

if (!map.containsKey(j)) {

ArrayList<Integer> list = new ArrayList<>();

list.add(A[i]);

map.put(j, list);

}

//index j exists. addA[i] to existing arraylist

if (map.containsKey(j)) {

map.get(j).add(A[i]);

}

}

//declaring Sorted Array and index

int[] Sorted = new int[A.length];

int Sindex = 0;

//Sorting Buckets

for (int j = 0; j <= buckets; j++) {

//Makes sure going to j that have arraylists.

if (map.containsKey(j)) {

Integer[] B = new Integer[map.get(j).size()]; //convert bucket arraylist to array

map.get(j).toArray(B);

InsertionSort(B);

//append buckets to Sorted array. No Array out of bound.

if(Sindex<Sorted.length){

for (int i = 0; i < B.length; i++) {

int temp = (B[i]);

Sorted[Sindex] = temp;

Sindex++;

}

}

}

}

return Sorted;

}

//--------------------------RadixSort----------------

//Parameters: Unsorted Array

//Most confusing sort

//Sorting by least significant digit ones. to most

//ex: 24 ,501, 2, 90, 354

// 501, 90, 2, 24, 354 (ones)

// 501, 2, 24, 354, 90 (twos)

// 2, 24, , 90 ,354 , 501 (third)

public static void RadixSort (int A[]){

int copyLength = A.length;

// Get maximum number in the array.

int max = A[0];

for (int i = 1; i < A.length; i++)

if (A[i] > max)

max = A[i];

//getting length of max number

for(int k = 1; max/k>0; k \*=10 ){

int[] B = new int[copyLength]; // Sorted Array. B

// Declare and create Distinct count array

int[] C = new int[10];

for(int j = 0; j<10; ++j) {

C[j] = 0;//start as empty array

}

//Modified COunting Sort

// fill C. Distinct Count of element in A.

//ex. A[1,1,2,2,2]

// C[2,3]

//removing leading zeros

for(int j=0; j<copyLength; j++) {

C[ (A[j]/k)%10 ] += 1;

}

// C[j] now holds the index of element in B

for (int j=1; j<10; j++) {

C[j] = C[j] + C[j - 1];

}

// Fill up Sorted Array. B.

for (int j = (copyLength-1); j>=0; j--){

count++; //cost&sort

B[C[ (A[j]/k)%10 ]-1 ] = A[j];

C[ (A[j]/k)%10 ] -= 1;

}

for(int j=0; j<copyLength; j++){

A[j] = B[j];

}

}

}

//--------------InsertSort. same as Lab1-----------------------

//Parameters. Unsorted Array. Mimics psuedo code

public static void InsertionSort (Integer myArray[]){

for(int index= 1; index<myArray.length; index++){ //start from index 1 compare to index 0

int key = myArray[index];

int i = index - 1;

while(i >= 0 && myArray[i] > key){

count ++; //cost & sort

myArray[i + 1] = myArray[i];

i = i - 1;

}

myArray[i + 1] = key;

}

}

}

Output

-------Num8.txt------------

InsertionSort: 14

[1, 2, 3, 4, 5, 6, 7, 8]

BucketSort: 9

[1, 2, 2, 3, 4, 5, 6, 7]

CountingSort: 8

[0, 1, 3, 4, 5, 6, 7, 8]

RadixSort: 8

[1, 2, 3, 4, 5, 6, 7, 8]

-------Num16.txt------------

InsertionSort: 84

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

BucketSort: 87

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 15]

CountingSort: 16

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16]

RadixSort: 32

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

-------Num32.txt------------

InsertionSort: 249

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

BucketSort: 222

[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

CountingSort: 32

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

RadixSort: 64

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]

-------Num64.txt------------

InsertionSort: 966

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]

BucketSort: 988

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63]

CountingSort: 64

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]

RadixSort: 128

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]

-------Num128.txt------------

InsertionSort: 4195

[51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101]

BucketSort: 4193

[51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

CountingSort: 128

[51, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101]

RadixSort: 384

[51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101]

-------Num256.txt------------

InsertionSort: 15372

[52, 53, 54, 55, 56, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101]

BucketSort: 15173

[51, 52, 53, 54, 55, 56, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

CountingSort: 256

[52, 53, 54, 55, 56, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101]

RadixSort: 768

[52, 53, 54, 55, 56, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101]