**COMP2080**

**Assignment 1 (10%)**

**Due: 14th February 2019 11:30PM**

**Given: 20th January 2019**

**Submission Instructions:**

1. **Fill in your full name and student number in the spaces provided further down this page.**
2. **Paste the code for each class and main program into this document after your chosen question.**
3. **Upload this document to blackboard and your complete compressed solution as two separate files.**

* You **must** have your name and student id number commented at the top of all code submitted.
* All submissions should at least compile.
* **Non-compiling assignments will not be marked and be given a grade of 0**.
* All your submissions should be suitably documented.

Name:Edward Entecott

Student ID:101190443

You must choose **ONE** of the questions below to submit

**Question 1 :**

The purpose of this question is to allow the student to independently compare the run time of various **sorting** algorithms. This will allow the student to get a better understanding of time complexity. It also aims to build an appreciation of the effects that the size and organization of data have on the speed of algorithms.

The sorting algorithms that will be examined are:

1. Selection sort
2. Insertion sort
3. Merge sort
4. Quick Sort

All sorting algorithms must sort the arrays in ascending order.

Each sorting and searching algorithm will be comparatively run on arrays of the following sizes:

1. Fifty (50)
2. one thousand (1000)
3. ten thousand (10,000)
4. one hundred Thousand (100,000)
5. one million (1,000,000)

**Each sort is to be tested on the same data set to strive for some consis**tency.

**Sorting Methodology and Requirements:**

1. Create a **core data set** called “***coreData***” which must be a single array of size one million (1,000,000) integers filled with random numbers between one (1) and three million (3,000,000) in size.
2. For each comparative test, four (4) **copies** (one for each sort to be tested) consisting of the same data from “***coreData***“ must be made. For example, if the comparative test is on one thousand (1000) elements, four arrays of size (1000) should be made and filled with a **copy** of the first one thousand (1000) elements of “***coreData***“.

All recorded times **must** be used using System.nanoTime() for best accuracy. An example of how to use it is given below.

long start = System.nanoTime();

//code to be tested

long end = System.nanoTime();

long timeTaken = end – start;

|  |  |
| --- | --- |
| **Test data size** | **Time unit** |
| 50 | nanoSeconds |
| 1000 | nanoSeconds |
| 10,000 | nanoSeconds |
| 100,000 | nanoSeconds |
| 1,000,000 | milliseconds |

The unit for printing the time must be done according to the following table

**Output requirements:**

The **name of each sorting algorithm tested** and **the time it took to sort the data** must be shown grouped by the test data size. For a test data size of fifty (50), the time taken for all the sorting algorithms to sort the data must be shown one after another.

PASTE YOUR CLASSES FOR QUESTION 1 HERE:

package comp2080assignment1;

public class InsertionSortAsc {

public void sort(int array[]){

int n = array.length;

for (int x = 1; x < n; x++) {

int temp = array[x];

int pos = x - 1;

while (pos >= 0 && array[pos] > temp) {

array[pos + 1] = array[pos];

pos--;

}

array[pos + 1] = temp;

package comp2080assignment1;

public class MergeSort {

public void merge(int arr[], int left, int mid, int right)

{

// Finding the sizes of the two subarrays that will be merged

int leftArr = mid - left + 1;

int rightArr = right - mid;

// Creating the temp arrays to hold the two halves of the main array

int tempLeft[] = new int [leftArr];

int tempRight[] = new int [rightArr];

//Copy the data to temp the arrays

for (int x=0; x<leftArr; x++)

tempLeft[x] = arr[left + x];

for (int y=0; y<rightArr; y++)

tempRight[y] = arr[mid + 1 + y];

// Merge the temp arrays

// Initial indexes of first and second subarrays

int i = 0, j = 0;

// Initial index of the merged subarry array

int k = left;

while (i < leftArr && j < rightArr)

{

if (tempLeft[i] <= tempRight[j])

{

arr[k] = tempLeft[i];

i++;

}

else

{

arr[k] = tempRight[j];

j++;

}

k++;

}

// Copy remaining elements of tempLeft[] if any

while (i < leftArr)

{

arr[k] = tempLeft[i];

i++;

k++;

}

// Copy remaining elements of tempRight[] if any

while (j < rightArr)

{

arr[k] = tempRight[j];

j++;

k++;

}

}

// Main function that will sort the arrays using the merge function

public void mSort(int arr[], int left, int right)

{

if (left < right)

{

// Find the middle point

int mid = (left+right)/2;

// Sort first and second halves

mSort(arr, left, mid);

mSort(arr , mid+1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

}

package comp2080assignment1;

public class QuickSort {

public int partition(int arr[], int low, int high)

{

//low will be the first element, high will be the last element

int pivot = arr[high]; //Set the pivot point for the last element

int x = (low - 1); // index of smaller element

for (int y = low; y < high; y++)

{

if (arr[y] < pivot) // If the current array element is smaller than the pivot point

{

x++;

int temp = arr[x];

arr[x] = arr[y];

arr[y] = temp;

}

}

int temp = arr[x + 1];

arr[x + 1] = arr[high];

arr[high] = temp;

return x + 1;

}

//arr[] = The array that will be sorted

//low wil be the starting point

//high will be the end point

public void QSort(int arr[], int low, int high)

{

if (low < high)

{

int pivot = partition(arr, low, high);

QSort(arr, low, pivot-1);

QSort(arr, pivot+1, high);

}

}

}

package comp2080assignment1;

public class SelectionSort {

public void selSort(int arr[])

{

int n = arr.length;

for (int x = 0; x < n-1; x++){

int smallestLoc = x;

for (int y = x+1; y < n; y++){

if (arr[y] < arr[smallestLoc]) //m\_array[y] is less than smallestLoc, sorts in Ascending

smallestLoc = y; // greater than ( > ) will sort in Descending

}

int temp = arr[smallestLoc];

arr[smallestLoc] = arr[x];

arr[x] = temp;

}

package comp2080assignment1;

import static java.lang.Math.random;

import java.util.\*;

public class Comp2080Assignment1 {

public static void printArray(int array[]) {

int n = array.length;

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

System.out.print(array[i] + " ");

}

System.out.println();

}

public static void main(String[] args) {

int max = 1000000; //Size of the main Array

int max50 = 50; //Size for the 50 Array

int maxThou = 1000; //Size for the 1000 Array

int maxTenThou = 10000; //Size for the 10,000 Array

int maxHunThou = 100000; //Size for the 100,000 Array

int maxMill = 1000000; //Size for the 1,000,000 Array

int coreData[] = new int [max];

//Fill the Main Array with random variables

for(int x = 0; x < max; x++){

coreData[x] = (int)(Math.random()\*3000000 + 1);

}

//Fill the arrays of size 50 with variables from the main toSort array

//Quicksort Array of 50

int[] arrayQ50 = new int[max50];

for(int x=0; x<max50; x++) {

arrayQ50[x] = coreData[x];

}

//Insertion Sort Array of 50

int[] arrayIS50 = new int[max50];

for(int x=0; x<max50; x++) {

arrayIS50[x] = coreData[x];

}

//Merge Sort Array of 50

int[] arrayMS50 = new int[max50];

for(int x=0; x<max50; x++) {

arrayMS50[x] = coreData[x];

}

//Selection Sort Array of 50

int[] arraySelS50 = new int[max50];

for(int x=0; x<max50; x++) {

arraySelS50[x] = coreData[x];

}

//Fill the arrays of size one thousand with variables from the main toSort array

//Quicksort Array of 1000

int[] arrayQThou = new int[maxThou];

for(int x = 0; x < maxThou; x++){

arrayQThou[x] = coreData[x];

}

//Insertion Sort Array of 1000

int[] arrayISThou = new int[maxThou];

for(int x = 0; x < maxThou; x++){

arrayISThou[x] = coreData[x];

}

//Merge Sort Array of 1000

int[] arrayMSThou = new int[maxThou];

for(int x=0; x<maxThou; x++) {

arrayMSThou[x] = coreData[x];

}

//Selection Sort Array of 1000

int[] arraySelSThou = new int[maxThou];

for(int x=0; x<maxThou; x++) {

arraySelSThou[x] = coreData[x];

}

//Fill the arrays of size ten thousand with variables from the main toSort array

//Quick Sort Array of 10,000

int[] arrayQTenThou = new int[maxTenThou];

for(int x = 0; x < maxTenThou; x++){

arrayQTenThou[x] = coreData[x];

}

//Insertion Sort of 10,000

int[] arrayISTenThou = new int[maxTenThou];

for(int x = 0; x < maxTenThou; x++){

arrayISTenThou[x] = coreData[x];

}

//Merge Sort of 10,000

int[] arrayMSTenThou = new int[maxTenThou];

for(int x = 0; x < maxTenThou; x++){

arrayMSTenThou[x] = coreData[x];

}

//Selection Sort Array of 10,000

int[] arraySelSTenThou = new int[maxTenThou];

for(int x = 0; x < maxTenThou; x++){

arraySelSTenThou[x] = coreData[x];

}

//Fill the arrays of size one hundred thousand with variables from the main toSort array

//Quicksort Array of 100,000

int[] arrayQHunThou = new int[maxHunThou];

for(int x = 0; x < maxHunThou; x++){

arrayQHunThou[x] = coreData[x];

}

//Insertion Sort Array of 100,000

int[] arrayISHunThou = new int[maxHunThou];

for(int x = 0; x < maxHunThou; x++){

arrayISHunThou[x] = coreData[x];

}

//Merge Sort Array of 100,000

int[] arrayMSHunThou = new int[maxHunThou];

for(int x = 0; x < maxHunThou; x++){

arrayMSHunThou[x] = coreData[x];

}

//Selection Sort Array of 100,000

int[] arraySelSHunThou = new int[maxHunThou];

for(int x = 0; x < maxHunThou; x++){

arraySelSHunThou[x] = coreData[x];

}

//Fill the arrays of size one million with variables from the main toSort array

//Quicksort Array of 1,000,000

int[] arrayQMill = new int[maxMill];

for(int x = 0; x < maxMill; x++){

arrayQMill[x] = coreData[x];

}

//Insertion Sort Array of 1,000,000

int[] arrayISMill = new int[maxMill];

for(int x = 0; x < maxMill; x++){

arrayISMill[x] = coreData[x];

}

//Merge Sort Array of 1,000,000

int[] arrayMSMill = new int[maxMill];

for(int x = 0; x < maxMill; x++){

arrayMSMill[x] = coreData[x];

}

//Selection Sort Array of 1,000,000

int[] arraySelSMill = new int[maxMill];

for(int x = 0; x < maxMill; x++){

arraySelSMill[x] = coreData[x];

}

System.out.println("\t ---- Algorithim Timer ---- \n");

System.out.println("---- Recorded times for Arrays of Size 50 ----");

QuickSort quick50 = new QuickSort(); //Quicksort for Array of 50

InsertionSortAsc insert50 = new InsertionSortAsc(); //Insertion Sort for Array of 50

MergeSort mergeS50 = new MergeSort(); //Merge Sort for Array of 50

SelectionSort sel50 = new SelectionSort(); //Selection Sort for Array of 50

long startQ50 = System.nanoTime(); //Official start time

quick50.QSort(arrayQ50, 0, max50 - 1);

long endQ50 = System.nanoTime();//Offical endtime

long outQ50 = (endQ50 - startQ50); // Get the final time

System.out.println(" Quicksort: " + outQ50 + " ns"); // Print out the final time

long startInsert50 = System.nanoTime(); //Official start time

insert50.sort(arrayIS50);

long endInsert50 = System.nanoTime();//Offical endtime

long outInsert50 = (endInsert50 - startInsert50); // Get the final time

System.out.println(" Insertion Sort: " + outInsert50 + " ns"); // Print out the final time

long startM50 = System.nanoTime(); //Official start time

mergeS50.mSort(arrayMS50, 0, max50 - 1);

long endM50 = System.nanoTime(); // Official end time

long outM50 = (endM50 - startM50); //Get the final time

System.out.println(" Merge Sort: " + outM50 + " ns"); //Print the final time

long startSel50 = System.nanoTime(); // Official start time

sel50.selSort(arraySelS50);

long endSel50 = System.nanoTime(); //Official end time

long outSel50 = (endSel50 - startSel50); //Get the final time

System.out.println(" Selection Sort: " + outSel50 + " ns"); //Print the final time

System.out.println("\n---- Recorded Times for Arrays of Size 1000 ----");

QuickSort quickThou = new QuickSort(); //Quicksort for Array of 1000

InsertionSortAsc insertThou = new InsertionSortAsc(); //Insertion Sort for Array of 1000

MergeSort mergeSThou = new MergeSort(); //Merge Sort for Array of 1000

SelectionSort selThou = new SelectionSort(); //Selection Sort for Array of 1000

long startQThou = System.nanoTime();

quickThou.QSort(arrayQThou, 0, maxThou - 1);

long endQThou = System.nanoTime();

long outQThou = (endQThou - startQThou);

System.out.println(" Quicksort: " + outQThou + " ns");

long startInsertThou = System.nanoTime();

insertThou.sort(arrayISThou);

long endInsertThou = System.nanoTime();

long outInsertThou = (endInsertThou - startInsertThou);

System.out.println(" Insertion Sort: " + outInsertThou + " ns");

long startMThou = System.nanoTime();

mergeSThou.mSort(arrayMSThou, 0, maxThou - 1);

long endMThou = System.nanoTime();

long outMThou = (endMThou - startMThou);

System.out.println(" Merge Sort: " + outMThou + " ns");

long startSelThou = System.nanoTime();

selThou.selSort(arraySelSThou);

long endSelThou = System.nanoTime();

long outSelThou = (endSelThou - startSelThou);

System.out.println(" Selection Sort: " + outSelThou + " ns");

System.out.println("\n---- Recorded Times for Arrays of Size 10,000 ----");

QuickSort quickTenThou = new QuickSort(); //Quicksort for Array of 10,000

InsertionSortAsc insertTenThou = new InsertionSortAsc(); //Insertion Sort for Array of 10,000

MergeSort mergeSTenThou = new MergeSort(); //Merge Sort for Array of 10,000

SelectionSort selTenThou = new SelectionSort(); //Selection Sort for Array of 10,000

long startQTenThou = System.nanoTime();

quickTenThou.QSort(arrayQTenThou, 0, maxTenThou - 1);

long endQTenThou = System.nanoTime();

long outQTenThou = (endQTenThou - startQTenThou);

System.out.println(" Quicksort: " + outQTenThou + " ns");

long startInsertTenThou = System.nanoTime();

insertTenThou.sort(arrayISTenThou);

long endInsertTenThou = System.nanoTime();

long outInsertTenThou = (endInsertTenThou - startInsertTenThou);

System.out.println(" Insertion Sort: " + outInsertTenThou + " ns");

long startMTenThou = System.nanoTime();

mergeSTenThou.mSort(arrayMSTenThou, 0, maxTenThou - 1);

long endMTenThou = System.nanoTime();

long outMTenThou = (endMTenThou - startMTenThou);

System.out.println(" Merge Sort: " + outMTenThou + " ns");

long startSelTenThou = System.nanoTime();

selTenThou.selSort(arraySelSTenThou);

long endSelTenThou = System.nanoTime();

long outSelTenThou = (endSelTenThou - startSelTenThou);

System.out.println(" Selection Sort: " + outSelTenThou + " ns");

System.out.println("\n---- Recorded Times for Arrays of Size 100,000 ");

QuickSort quickHunThou = new QuickSort(); //Quicksort for Array of 100,000

InsertionSortAsc insertHunThou = new InsertionSortAsc(); //Insertion Sort for Array of 100,000

MergeSort mergeSHunThou = new MergeSort(); //Merge Sort for Array of 100,000

SelectionSort selHunThou = new SelectionSort(); //Selection Sort for Array of 100,000

long startQHunThou = System.nanoTime();

quickHunThou.QSort(arrayQHunThou, 0, maxHunThou - 1);

long endQHunThou = System.nanoTime();

long outQHunThou = (endQHunThou - startQHunThou);

System.out.println(" Quicksort: " + outQHunThou + " ns");

long startInsertHunThou = System.nanoTime();

insertHunThou.sort(arrayISHunThou);

long endInsertHunThou = System.nanoTime();

long outInsertHunThou = (endInsertHunThou - startInsertHunThou);

System.out.println(" Insertion Sort: " + outInsertHunThou + " ns");

long startMHunThou = System.nanoTime();

mergeSHunThou.mSort(arrayMSHunThou, 0, maxHunThou - 1);

long endMHunThou = System.nanoTime();

long outMHunThou = (endMHunThou - startMHunThou);

System.out.println(" Merge Sort: " + outMHunThou + " ns");

long startSelHunThou = System.nanoTime();

selHunThou.selSort(arraySelSHunThou);

long endSelHunThou = System.nanoTime();

long outSelHunThou = (endSelHunThou - startSelHunThou);

System.out.println(" Selection Sort: " + outSelHunThou + " ns");

System.out.println("\n---- Recorded Times for Arrays of Size 1,000,000 ---- ");

QuickSort quickMill = new QuickSort(); //Quicksort for Array of 1000000

InsertionSortAsc insertMill = new InsertionSortAsc(); //Insertion Sort for Array of 1000000

MergeSort mergeSMill = new MergeSort(); //Merge Sort for Array of 1000000

SelectionSort selMill = new SelectionSort(); //Selection Sort for Array of 1000000

long startQMill = System.currentTimeMillis();

quickMill.QSort(arrayQMill, 0, maxMill - 1);

long endQMill = System.currentTimeMillis();

long outQMill = (endQMill - startQMill);

System.out.println(" Quicksort: " + outQMill + " ms");

long startInsertMill = System.currentTimeMillis();

insertMill.sort(arrayISMill);

long endInsertMill = System.currentTimeMillis();

long outInsertMill = (endInsertMill - startInsertMill);

System.out.println(" Insertion Sort: " + outInsertMill + " ms");

long startMMill = System.currentTimeMillis();

mergeSMill.mSort(arrayMSMill, 0, maxMill - 1);

long endMMill = System.currentTimeMillis();

long outMMill = (endMMill - startMMill);

System.out.println(" Merge Sort: " + outMMill + " ms");

long startSelMill = System.currentTimeMillis();

selMill.selSort(arraySelSMill);

long endSelMill = System.currentTimeMillis();

long outSelMill = (endSelMill - startSelMill);

System.out.println(" Selection Sort: " + outSelMill + " ms");

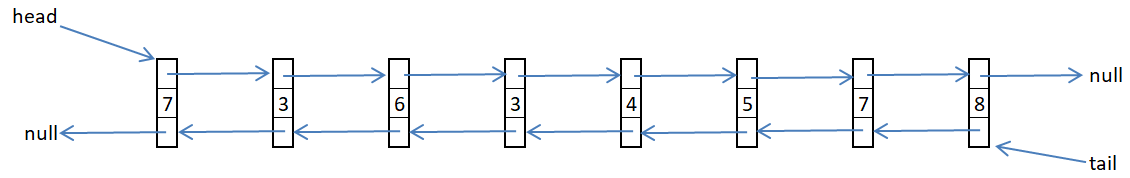
}

}

**Question 2:**

Your task is to create a class called “***HugeInteger***” that stores and allows operations on huge integers. The class ***HugeInteger*** stores the digits of the number in a doubly linked list **in reverse order** (i.e the least significant digit is the first item, most significant digit is the last).

For example the number “87543637” would be stored in reverse order as shown below.



This class must have three (3) state values:

|  |  |
| --- | --- |
| **State variable name** | **Information stored** |
| ***isPositive*** | Stores the sign of the number (positive or negative) as a boolean state value |
| **head** | Stores the first Node of the linked list (It is null if the list is empty). |
| **tail** | Stores or keeps track of the last item in the list (It is null if the list is empty). |
| **length** | Stores the number of digits in the number (excluding the sign) |

|  |  |
| --- | --- |
| **Constructors** |  |
| ***Public HugeInteger ()*** | An empty linked list is created. That is: ***isPositive*** must be set to true by default. **Head** and **tail** are set to null. If this variable is to be displayed, a “0” should be printed (the zero is not stored). **Length** must be set to 0; |
| ***Public HugeInteger (String number)*** | Creates the number from the string with all leading zeros (0) removed. |

Constructors

Additional Behaviour

|  |  |
| --- | --- |
| **Method Prototype** |  |
| ***HugeInteger addPositive(HugeInteger num2)*** | Returns a new ***HugeInteger*** containing the result of adding num2 to the stored number**. You MUST assume *num2* and the number being added to are BOTH positive.** |
| ***int compareTo (HugeInteger num2)*** | Returns -1 if the number stored is less than num2  Returns 0 if the number stored is equal to num2  Returns 1 if the number stored is greater than num2 |
| ***String toString()*** | Returns a string representation of the number |
| ***void concatenateDigit(int digit)*** | Adds a digit to the **end** of the number (at the front of the list). **Note: if the list is empty leading zeros should not be added.** |
| ***void addLast(int digit)*** | Adds a digit to the **front** of the number (at the end of the list). This can be used in the ***addPositive*** method |

You must create a ***HugeInteger*** class based on the specifications outlined above and create a main program that tests it with the following code (on the next page):

public static void main(String[] args) {

HugeInteger hi = new HugeInteger();

System.out.println(hi);

// testing sign

HugeInteger hi1 = new HugeInteger("34545234");

System.out.println(hi1);

HugeInteger hi2 = new HugeInteger("-2455434324344");

System.out.println(hi2);

//testing leading zeros

HugeInteger hi3 = new HugeInteger("000034545234");

System.out.println(hi3);

HugeInteger hi4 = new HugeInteger("-00000002455434324344");

System.out.println(hi4);

// testing concatenate with a single digit

HugeInteger hi5 = new HugeInteger();

System.out.println(hi5);

hi5.concatenateDigit(3);

System.out.println(hi5);

// testing add with two positive numbers

HugeInteger hi6 = new HugeInteger("9");

HugeInteger hi7 = new HugeInteger("6");

HugeInteger hi8 = hi6.addPositive(hi7);

System.out.println(hi6+" + "+hi7+" = "+hi8);

HugeInteger hi9 = new HugeInteger("9996354");

HugeInteger hi10 = new HugeInteger("4656");

HugeInteger hi11 = hi9.addPositive(hi10);

System.out.println(hi9+" + "+hi10+" = "+hi11);

System.out.println(hi5.compareTo(hi4));

System.out.println(hi2.compareTo(hi1));

}

Sample output:

0

34545234

-2455434324344

34545234

-2455434324344

0

3

9 + 6 = 15

9996354 + 4656 = 10001010

1

-1

**PASTE YOUR HUGEINTEGER CLASS HERE:**