/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* A client that demonstrates sorting algorithms

\*/

public class Client {

public static void main(String[] args) {

Student[] base1 = generateStudents(1000000);

Student[] base2 = generateStudents(100000);

Student[] mergeArr = base1.clone();

Student[] quickArr = base1.clone();

Student[] sBubArr = base2.clone();

Student[] eBubArr = base2.clone();

Student[] insArr = base2.clone();

Student[] selArr = base2.clone();

Comparator compId = new IdComparator();

Comparator compfName = new fNameComparator();

Comparator complName = new lNameComparator();

Comparator compStanding = new StandingComparator();

Comparator compGpa = new GpaComparator();

long start, stop, elapse;

LinkedQueue times = new LinkedQueue();

start = System.currentTimeMillis(); // Starts timer

Sort.mergeSort(mergeArr, compId); // Sorts on id using merge sort

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

LinkedQueue queue = arrayToQueue(quickArr); // Converts array to queue

start = System.currentTimeMillis(); // Starts timer

Sort.quickSort(queue, compGpa); // Sorts on gpa using quick sort

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

start = System.currentTimeMillis(); // Starts timer

Sort.simpleBubbleSort(sBubArr, compId); // Sorts on gpa using quick sort

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

start = System.currentTimeMillis(); // Starts timer

Sort.efficientBubbleSort(eBubArr, compId); // Sorts on gpa using quick sort

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

start = System.currentTimeMillis(); // Starts timer

Sort.insertionSort(insArr, compGpa); // Sorts on gpa using insertion sort

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

start = System.currentTimeMillis(); // Starts timer

Sort.selectionSort(selArr, compStanding); // Sorts on standing using selection sort

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

start = System.currentTimeMillis(); // Starts timer

// Sorts on standing using selection sort

Sort.radixSort(selArr, compStanding, compStanding); // Standing standing

Sort.radixSort(selArr, compStanding, compGpa); // Standing by gpa

Sort.radixSort(selArr, compGpa, complName); // Standing by gpa

Sort.radixSort(selArr, complName, compfName); // Standing by gpa

Sort.radixSort(selArr, compfName, compId); // Standing by gpa

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

times.enqueue(elapse);

printTable(times);

}

/\*\*

\* @param arSize

\* @return

\*/

public static Student[] generateStudents(int arSize)

{

Student[] array;

array = new Student[arSize];

int size = 0;

for(int i=0; i<arSize; i++)

{

boolean valid = false;

if(size == 0)

{

Student student = new Student();

array[i] = student;

size++;

}

else

{

while(!valid)

{

for(int j=0;j<size;j++)

{

Student student = new Student();

if(student.getId() != array[j].getId()) // Valid is true iff there is no duplicate... otherwise will rerun new student until no dupe

{

valid = true;

}

if(valid == true) // only runs if is valid

{

array[i] = student;

size++;

break;

}

}

}

}

}

return array;

}

/\*\*

\* @param array

\* @return

\*/

public static LinkedQueue arrayToQueue(Student[] array)

{

int trueSize = array.length;

LinkedQueue queue = new LinkedQueue();

for(int i=0; i<trueSize; i++)

{

queue.enqueue(array[i]);

}

return queue;

}

/\*\*

\* @param data

\* @param size

\*/

public static void printArray(Student[] data, int size)

{

for ( int i = 0; i < size; i++ )

System.out.println( data[i] );

}

/\*\*

\* @param data

\* @param size

\*/

public static void printQueue(LinkedQueue data, int size)

{

for ( int i = 0; i < size; i++ )

System.out.println( data.dequeue() );

}

/\*\*

\* @param queue

\*/

public static void printTable(LinkedQueue queue)

{

ASCII table = new ASCII();

table.getColumns().add(new ASCII.Column("Sort"));

table.getColumns().add(new ASCII.Column("N"));

table.getColumns().add(new ASCII.Column("Time (msec)"));

String[] sorts = new String[] {"Merge", "Quick", "SimpleBubble", "EfficientBubble", "Insertion", "Selection", "Radix"};

String[] n = new String[] {"1,000,000", "1,000,000", "100,000", "100,000", "100,000", "100,000", "1,000,000"};

int trueSize = queue.size();

for(int i=0; i<trueSize; i++)

{

ASCII.Row row = new ASCII.Row();

long timeLong = ( long ) queue.dequeue();

String time = Long.toString(timeLong);

table.getData().add(row);

row.getValues().add(sorts[i]);

row.getValues().add(n[i]);

row.getValues().add(time);

}

table.calcWidth();

table.printTable();

}

}

import java.util.Arrays;

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* A class that holds different sorting algorithms

\*/

public class Sort {

/\*\*

\*

\* @param <K>

\* @param data

\* @param comp

\*/

public static <K> void simpleBubbleSort( K[] data, Comparator<K> comp )

{

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

{

for ( int j = 0; j < data.length - 1; j++ )

{

if ( comp.compare( data[j], data[j+1] ) <= 0 )

{

K temp = data[j];

data[j] = data[j+1];

data[j+1] = temp;

}

}

}

}

/\*\*

\*

\* @param <K>

\* @param data

\* @param comp

\*/

public static <K> void efficientBubbleSort( K[] data, Comparator<K> comp )

{

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

{

boolean is\_sorted = true;

// skip the already sorted largest elements

for (int j = 0; j < data.length - i; j++)

{

if (comp.compare(data[j], data[j + 1]) > 0)

{ //data[j] > data[j + 1]

K temp = data[j];

data[j] = data[j + 1];

data[j + 1] = temp;

is\_sorted = false;

}

}

if (is\_sorted) return;

}

}

/\*\*

\*

\* @param <K>

\* @param S1

\* @param S2

\* @param S

\* @param comp

\*/

public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {

int i = 0, j = 0;

while (i + j < S.length) {

if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))

S[i+j] = S1[i++]; // copy ith element of S1 and increment i

else

S[i+j] = S2[j++]; // copy jth element of S2 and increment j

}

}

/\*\*

\*

\* @param <K>

\* @param S

\* @param comp

\*/

public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {

int n = S.length;

if (n < 2) return; // array is trivially sorted

// divide

int mid = n/2;

K[ ] S1 = Arrays.copyOfRange(S, 0, mid); // copy of first half

K[ ] S2 = Arrays.copyOfRange(S, mid, n); // copy of second half

// conquer (with recursion)

mergeSort(S1, comp); // sort copy of first half

mergeSort(S2, comp); // sort copy of second half

// merge results

merge(S1, S2, S, comp); // merge sorted halves back into original

}

/\*\*

\*

\* @param <K>

\* @param S

\* @param comp1

\* @param comp2

\*/

public static <K> void radixSort(K[ ] S, Comparator<K> comp1, Comparator<K> comp2)

{

Sort.mergeSort(S, comp1);

Sort.mergeSort(S, comp2);

}

/\*\* Performs a Selection Sort

\* on an integer array

\* @param the array to sort

\*/

public static <K> void selectionSort( K[] array, Comparator<K> comp )

{

K temp; // temporary location for swap

int max; // index of maximum value in subarray

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

{

// find index of largest value in subarray

max = indexOfLargestElement(array, array.length-1, comp);

// swap array[max] and array[array.length-i-1]

temp = array[max];

array[max] = array[array.length-i-1];

array[array.length-i-1] = temp;

}

}

/\*\* Finds index of largest element

\* @param size the size of the subarray

\* @return the index of the largest element in the subarray

\*/

private static <K> int indexOfLargestElement( K[] array, int size, Comparator<K> comp)

{

int index = 0;

for(int i=1; i<size; i++)

{

if(comp.compare(array[i], array[index]) < 0) // if(array[i] > array[index])

index = i;

}

return index;

}

/\*\* Performs and Insertion Sort on an integer array

\* @param array array to sort

\*/

public static <K> void insertionSort( K[] array, Comparator<K> comp )

{

int j;

K temp;

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

{

j = i;

temp = array[i];

while(j != 0 && comp.compare(array[j-1], temp) > 0) // while(j != 0 && array[j-1] > temp)

{

array[j] = array[j-1];

j--;

}

array[j] = temp;

}

}

/\*\*

\*

\* @param <K>

\* @param S

\* @param comp

\*/

public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {

int n = S.size( );

if (n < 2) return; // queue is trivially sorted

// divide

K pivot = S.first( ); // using first as arbitrary pivot

Queue<K> L = new LinkedQueue<>( );

Queue<K> E = new LinkedQueue<>( );

Queue<K> G = new LinkedQueue<>( );

while (!S.isEmpty( )) { // divide original into L, E, and G

K element = S.dequeue( );

int c = comp.compare(element, pivot);

if (c < 0) // element is less than pivot

L.enqueue(element);

else if (c == 0) // element is equal to pivot

E.enqueue(element);

else // element is greater than pivot

G.enqueue(element);

}

// conquer

quickSort(L, comp); // sort elements less than pivot

quickSort(G, comp); // sort elements greater than pivot

// concatenate results

while (!L.isEmpty( ))

S.enqueue(L.dequeue( ));

while (!E.isEmpty( ))

S.enqueue(E.dequeue( ));

while (!G.isEmpty( ))

S.enqueue(G.dequeue( ));

}

}

import java.util.Random;

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* A class that holds information relevant to a student

\*/

public class Student {

private int id;

private String lname = "", fname = "", standing = "";

private double gpa;

/\*

@param id

@param lname

@param fname

@param standing

@param gpa

\*/

Student()

{

this.id = setId();

this.lname = setLname();

this.fname = setFname();

this.standing = standing();

this.gpa = gpa();

}

public int setId()

{

Random rand = new Random();

int randInt = rand.nextInt(10000000);

return randInt;

}

public int getId() { return id; }

public String setFname()

{

Random rand = new Random();

int randInt1 = rand.nextInt(5);

int totalNum = 10 - randInt1;

String fName = "";

for(int i=0;i<totalNum; i++)

{

int randInt2 = rand.nextInt(26);

char ch = (char)(randInt2 + 'a');

if(i==0)

{

char ch1 = Character.toUpperCase(ch);

fName += ch1;

}

else

{

fName += ch;

}

}

return fName;

}

public String getLname() { return lname; }

public String setLname()

{

Random rand = new Random();

String[] lnames = {"Smith", "Johnson", "Willaims", "Brown", "Jones", "Miller", "Davis", "Garcia", "Rodriguez", "Wilson"};

int randInt = rand.nextInt(10);

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

{

if(randInt == i)

{

return lnames[i];

}

}

return null;

}

public String getFname() { return fname; }

public String standing()

{

Random rand = new Random();

int randInt = rand.nextInt(100);

if(randInt < 10)

return "Senior";

if(10 <= randInt && randInt < 30)

return "Junior";

if(30 <= randInt && randInt < 60)

return "Sophmore";

if(60 <= randInt)

return "Freshman";

return null;

}

public String getStanding() { return standing; }

public double gpa()

{

Random rand = new Random();

int randInt = rand.nextInt(100);

if(randInt < 5) // 0-.9

{

double max = 0.99;

double min = 0;

double randD = min + Math.random() \* (max-min);

return randD;

}

if(5 <= randInt && randInt < 25) // 1-1.9

{

double max = 1.99;

double min = 1;

double randD = min + Math.random() \* (max-min);

return randD;

}

if(25 <= randInt && randInt < 75) // 2-2.9

{

double max = 2.99;

double min = 2;

double randD = min + Math.random() \* (max-min);

return randD;

}

if(75 <= randInt && randInt < 95) // 3-3.9

{

double max = 3.99;

double min = 3;

double randD = min + Math.random() \* (max-min);

return randD;

}

if(90 <= randInt) // 4

return 4.00;

return 0;

}

public double getGpa() { return gpa; }

public String toString( )

{

return id + ":" + fname + ":" + lname + ":" + standing + ":" + gpa;

}

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparable interface

\*/

public interface Comparable <K> {

//

//Used for a comparison based on the "natural ordering"

//

// return < 0 if this.a < b

// return 0 if this.a = b

// return > 0 if this.a > b

int compareTo( K b );

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparator interface

\*/

public interface Comparator <K> {

int compare( K a, K b );

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparator for GPA

\*/

public class GpaComparator implements Comparator<Student> {

public int compare(Student a, Student b)

{

if ( a == null || b == null )

{

throw new NullPointerException("Type cannot be null");

}

double aGpa = a.getGpa();

double bGpa = b.getGpa();

return Double.compare(bGpa, aGpa);

}

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparator for ID

\*/

public class IdComparator implements Comparator<Student> {

public int compare(Student a, Student b)

{

if ( a == null || b == null )

{

throw new NullPointerException("Type cannot be null");

}

int aId = a.getId();

int bId = b.getId();

return Double.compare(aId, bId);

}

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparator for standing

\*/

public class StandingComparator implements Comparator<Student> {

public int compare(Student a, Student b)

{

if ( a == null || b == null )

{

throw new NullPointerException("Type cannot be null");

}

int returnValue = 0;

String aStanding = a.getStanding();

String bStanding = b.getStanding();

switch ( aStanding )

{

case "Senior" :

{

switch ( bStanding )

{

case "Senior" : returnValue = 0; break;

case "Junior" : returnValue = 1; break;

case "Sophmore" : returnValue = 1; break;

case "Freshman" : returnValue = 1; break;

}

break;

}

case "Junior" :

{

switch ( bStanding )

{

case "Senior" : returnValue = -1; break;

case "Junior" : returnValue = 0; break;

case "Sophmore" : returnValue = 1; break;

case "Freshman" : returnValue = 1; break;

}

break;

}

case "Sophmore" :

{

switch ( bStanding )

{

case "Senior" : returnValue = -1; break;

case "Junior" : returnValue = -1; break;

case "Sophmore" : returnValue = 0; break;

case "Freshman" : returnValue = 1; break;

}

break;

}

case "Freshman" :

{

switch ( bStanding )

{

case "Senior" : returnValue = -1; break;

case "Junior" : returnValue = -1; break;

case "Sophmore" : returnValue = -1; break;

case "Freshman" : returnValue = 0; break;

}

break;

}

default :

{

//throw invalid medal exception

}

}

return returnValue;

}

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparator for first name

\*/

public class fNameComparator implements Comparator<Student> {

public int compare(Student a, Student b)

{

if ( a == null || b == null )

{

throw new NullPointerException("Type cannot be null");

}

int returnValue = 0;

String aFname = a.getFname();

String bFname = b.getFname();

return bFname.compareTo(aFname);

}

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* Comparator for last name

\*/

public class lNameComparator implements Comparator<Student> {

public int compare(Student a, Student b)

{

if ( a == null || b == null )

{

throw new NullPointerException("Type cannot be null");

}

int returnValue = 0;

String aLname = a.getLname();

String bLname = b.getLname();

return bLname.compareTo(aLname);

}

}

import java.util.ArrayList;

import java.util.List;

/\*\*

\* @author Samuel Swedberg

\* @version 11/30/22

\*

\* The ASCII class holds information related to printing ASCII Tables

\*/

public class ASCII {

private final List<Column> columns = new ArrayList<>();

private final List<Row> data = new ArrayList<>();

private int maxWidth = Integer.MAX\_VALUE;

/\*\*

\* @return columns

\*/

public List<Column> getColumns() {

return columns;

}

/\*\*

\* @return data

\*/

public List<Row> getData() {

return data;

}

/\*\*

\* @return maxWidth

\*/

public int getNaxWidth() {

return maxWidth;

}

/\*\*

\* @param maxWidth

\*/

public void setMaxWidth(int maxWidth) {

this.maxWidth = maxWidth;

}

public static class Column {

private String name;

private int width;

public Column(String name) {

this.name = name;

}

}

public static class Row {

private final List<String> values = new ArrayList<>();

public List<String> getValues() {

return values;

}

}

/\*\*

\* Calculates width

\*/

public void calcWidth() {

for (Column column : columns) {

column.width = column.name.length() + 1;

}

for (Row row : data) {

int colIdx = 0;

for (String value : row.values) {

Column column = columns.get(colIdx);

if (value == null) continue;

column.width = Math.max(column.width, value.length() + 1);

colIdx++;

}

}

for (Column column : columns) {

column.width = Math.min(column.width, maxWidth);

}

}

/\*\*

\* @param columns, sb

\*/

private void writeColumnNames(List<Column> columns, StringBuilder sb) {

sb.append("| ");

for (Column column : columns) {

sb.append(String.format(" %-" + (column.width) + "s", column.name));

sb.append(" |");

}

sb.append("\n");

}

/\*\*

\* @param columns, sb

\*/

private void writeSeparator(List<Column> columns, StringBuilder sb) {

sb.append("+-");

for (Column column : columns) {

sb.append(String.format(" %-" + (column.width + 1) + "s", "").replace(' ', '-'));

sb.append("-+");

}

sb.append("\n");

}

/\*\*

\* @param columns, rows, sb

\*/

private void writeValues(List<Column> columns, List<Row> rows, StringBuilder sb) {

for (Row row : rows) {

int columnIndex = 0;

sb.append("| ");

for (String value : row.values) {

if (value != null && value.length() > maxWidth)

value = value.substring(0, maxWidth - 1);

sb.append(String.format(" %-" + columns.get(columnIndex).width + "s", value));

sb.append(" |");

columnIndex++;

}

sb.append("\n");

}

}

/\*\*

\* prints table

\*/

public void printTable() {

StringBuilder sb = new StringBuilder();

writeSeparator(columns, sb);

writeColumnNames(columns, sb);

writeSeparator(columns, sb);

writeValues(columns, data, sb);

writeSeparator(columns, sb);

System.out.println(sb.toString());

}

}

run:

+--------------------+-------------+---------------+

| Sort | N | Time (msec) |

+--------------------+-------------+---------------+

| Merge | 1,000,000 | 336 |

| Quick | 1,000,000 | 661 |

| SimpleBubble | 100,000 | 58884 |

| EfficientBubble | 100,000 | 42766 |

| Insertion | 100,000 | 15153 |

| Selection | 100,000 | 97052 |

| Radix | 1,000,000 | 163 |

+--------------------+-------------+---------------+

BUILD SUCCESSFUL (total time: 3 minutes 35 seconds)