**Voter Class**

import java.util.Random;

/\*\*

\* a class for voter in which initialization may be base on randomization

\* or correctly specifying the fields.

\* @author Richelin Metellus

\* @version 04/20/2017

\*/

public class Voter {

int ID;

String name;

String party;

String voted;

Random rand = new Random();

public Voter()

{

ID = rand.nextInt(100000000); // 0 - 99,999,999

name = generateRandomName();

party = generateRandomParty();

voted = hasVoted();

}

public Voter(int id, String name, String party, String Voted )

{

ID = id;

this.name = name;

this.party = party;

voted = Voted;

}

public int getId(){return ID;}

public String getName(){return name;}

public String getParty(){ return party;}

public String getVoted(){return voted;}

/\*\*

\*

\* @return a random stream of character with length between 5 and 10(inclusive)

\*/

private String generateRandomName()

{

int stringLength = 10- rand.nextInt(6); // name has to be btw 5 and 10(inclusive)

StringBuilder name = new StringBuilder(stringLength);

int randChar; // random ASCII value of character btw a and z as an integer.

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

{

//create a number btw 97(a) and 122(z) inclusive

randChar = 97 + rand.nextInt(122-97+1);

name.append((char) randChar);

}

return name.toString();

}

/\*\*

\*

\* @return random political party such that 30% is republican, and on.

\*

\*/

private String generateRandomParty()

{

int randNum = rand.nextInt(100);

if( randNum >= 0 && randNum < 30)

return "democrat";

else if(randNum >= 30 && randNum < 60)

return "republican";

else if (randNum >= 60 && randNum < 80)

return "independent";

else

return "other";

}

/\*\* @return a random voting status such that 58% might be yes.

private String hasVoted()

{

int randNum = rand.nextInt(100);

if( randNum >= 0 && randNum < 58)

return "Yes";

else

return "No";

}

@Override

public String toString()

{

return getClass().getName() + ":" + name + ":" + ID +":" + party + ":" + voted;

}

}

**Sort Class**

import java.util.ArrayList;

import java.util.Arrays;

public class Sort {

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

{

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

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

{

if( comp.compare(data[i], data[j]) < 0) // if data[i](3) > data[j](2) ---(in comp 3 > 2--> -1 < 0 true for ascending order u swap

// data[i] (Ama) < data[j](Aman) for for alphabetical

{

K temp = data[i];

data[i] = data[j];

data[j] = temp;

}

}

}

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

{

K temp; // temporary location for swap

int indexOfMax; // index of the maximum value in subarray

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

{

// find index of largest value in subarray

indexOfMax = indexOfLargestElement(data,data.length-i, comp);

// swap data[indexofMax] and data[data.length-i-1]

temp = data[indexOfMax];

data[indexOfMax] = data[data.length-i-1];

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

}

}

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

{

int j;

K temp;

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

{

j = i;

temp = data[i];

while( j!= 0 && comp.compare(data[j-1], temp) < 0) //if data[j-1].value > temp.value for ascending order

{

data[j] = data[j-1];

j--;

}

data[j] = temp;

}

}

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

{

int n = S.length;

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

// divide

int mid = n/2;

K[] S1 = Arrays.copyOfRange(S, 0, mid);

K[] S2 = Arrays.copyOfRange(S, mid, n);

//conquer with recursion

mergeSort(S1,comp);

mergeSort(S2,comp);

//merge results (rom..After all the child call is

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

}

/\*\*

\*

\* @param <K>

\* @param S

\* @param comp

\* @param a start of the segment

\* @param b end of the segment

\*/

public static<K> void quickSortInPlace(K[] S, Comparator<K> comp, int a, int b) // 1st call Qs(S,0,7) for an array of size 8.

{

if(a >= b) return; // subarray is trivially sorted. Base case

// when we have only 1 element into the subarray. (a=0, b=0) or the segment is invalid

/\*start of partionning logic(where element lesser than the pivot on left of pivot

and element greater than the pivot on the right of pivot.

\*/

int left = a; // the index 1st element in subarray

int right = b-1; // index element before pivot

K pivot = S[b]; // last element in orginal segment as pivot

K temp; // temp object for swapping

while(left <= right)

{

// scan until reaching value equal or larger than pivot(or right marker)

while(left <= right && comp.compare(S[left], pivot) > 0) left++;

// scan until reaching value equal or smaller than pivot(or left marker)

while(left <= right && comp.compare(S[right],pivot) < 0) // (96) > 50 comp: -1 < 0 true

right--;

if(left<= right) // indices did not strictly cross

{

// swap values and shring range

temp = S[left];

S[left] = S[right];

S[right] = temp;

left++; right--;

}

}

//put pivot into its final place(currently marked by left index)

temp = S[left];

S[left] = S[b];

S[b] = temp;

//----- End of partitioning---

//make recursive calls

quickSortInPlace(S, comp, a, left-1);

quickSortInPlace(S,comp, left+1, b);

}

/\*\*

\*

\* @param data sequence of element to be sorted

\* @param compList a bag of the comparators in which the highest comparator key

\* should be put 1st in the bag

\*/

public static<K> void radixSort(K[] data, ArrayBag<Comparator<K>> compList)

{

int lowKeyIndex = compList.getCurrentSize() -1;

mergeSort(data, compList.get(lowKeyIndex)); // name

mergeSort(data, compList.get(lowKeyIndex-1)); // voted

mergeSort(data, compList.get(lowKeyIndex-2)); // party comp

}

//--------------------------Private Utility --------------------------------

/\*\*

\*

\* @param <K>

\* @param array the array for which the comparison is to be done on.

\* @param size the size of the subarray.

\* @param comp comparator on the key of the array

\* @return the index of the largest element key. (index greater int for ID, greater A-Z for alphabetical order)

\*/

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

{

int index = 0;

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

{

if(comp.compare(array[i], array[index]) < 0) // array[i]< array[index] //< 0 ascending order; A-Z

// this is similar if array[i].value(23) > (12) array[index].value, then index = i.

{ // if you're doing the comp on Id number, then 23 >12 3 ret--> -1< 0...true

index = i;

}

}

return index;

}

/\*\*

\* Compare the value at the index of each subarray and

\* copy the smaller of the two.

\* @param <K>

\* @param S1 left subarray

\* @param S2 right subArray for number greater than the pivot

\* @param S Sequence of element to sort. an Array

\* @param comp comparator

\*/

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

{

int i = 0; int j = 0;

while(i+j < S.length)

{

if(j== S2.length || (i < S1.length && comp.compare(S1[i],S2[j]) > 0)) // if element at S1(11) > el @ S2(10)---> -1 > 0 false.

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

else

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

}

}

}

**ID Comparator Class**

/\*\*

\* This comparator is base on how you count things. 1 will have a higher priority

\* than 2.

\* @author Richelin Metellus

\* @version 04/20/2017

\*/

public class IdComparator implements Comparator<Voter>{

public int compare(Voter a, Voter b) throws NullPointerException

{

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

throw new NullPointerException("Invalid object reference");

int aId = a.getId();

int bId = b.getId();

if(aId < bId)

return 1;

else if(aId == bId)

return 0;

else

return -1;

}

}

**Name Comparator**

/\*\*

\* This comparator is based on the Lexicographic representation of a string.

\* @author Richelin Metellus

\* @version 04/21/2017

\*/

public class NameComparator implements Comparator<Voter> {

@Override

public int compare(Voter a, Voter b) throws NullPointerException

{

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

throw new NullPointerException("Invalid object reference");

int comparisonResult = 0;

String aName = a.getName();

int aNameSize = aName.length();

String bName = b.getName();

int bNameSize = bName.length();

// finding lesser string length to lower # of loop iterationn

int lengthDiff = aNameSize - bNameSize;

int iterationLimit = (lengthDiff >= 0) ? bNameSize: aNameSize;

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

{

char aNameCur = aName.charAt(i); // current char testing for nameA

char bNameCur = bName.charAt(i); // current char testing for name of Voter b.

if(aNameCur < bNameCur) // a = 97 < b =98 in ASCII value

{

comparisonResult = 1;

return comparisonResult;

} // i.e aName comes 1st alphabeically

else if( aNameCur > bNameCur)

{

comparisonResult = -1;

return comparisonResult;

}

else if (aNameCur == bNameCur &&aNameSize==bNameSize){

comparisonResult = 0;

}

else if (aNameCur == bNameCur &&aNameSize < bNameSize ) // special case.

comparisonResult = 1; // if comparing Ama with Aman

else if (aNameCur == bNameCur &&aNameSize > bNameSize) // special case.

comparisonResult = -1;

}

return comparisonResult; // if loop fully executes and reaches this point it return 1 or -1,0.. for sepcial case

}

**Political Party Comparator**

/\*\*

\*

\* @author Richelin Metellus

\* @@version 04/21/2017

\*/

public class PartyComparator implements Comparator<Voter> {

@Override

public int compare(Voter a, Voter b) throws NullPointerException, IllegalArgumentException

{

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

throw new NullPointerException("Invalid object reference");

String aParty = a.getParty();

String bParty = b.getParty();

int comparisonResult = 0;

switch(aParty)

{

case "republican" :

{

switch(bParty)

{

case "republican": comparisonResult = 0; break;

case "democrat" : comparisonResult = 1; break;

case "other" : comparisonResult = 1; break;

case "independent": comparisonResult = 1; break;

}

break;

}

case "democrat":

{

switch(bParty)

{

case "republican" : comparisonResult = -1; break;

case "democrat" : comparisonResult = 0; break;

case "other" : comparisonResult = 1 ; break;

case "independent": comparisonResult = 1 ; break;

}

break;

}

case "other":

{

switch(bParty)

{

case "republican" : comparisonResult = -1 ; break;

case "democrat" : comparisonResult = -1 ; break;

case "other" : comparisonResult = 0 ; break;

case "independent": comparisonResult = 1 ; break;

}

break;

}

case "independent":

{

switch(bParty)

{

case "republican": comparisonResult = -1 ; break;

case "democrat" : comparisonResult = -1 ; break;

case "other" : comparisonResult = -1 ; break;

case "independent":comparisonResult = 0 ; break;

}

break;

}

}

return comparisonResult;

}

}

**Voter Decision Comparator**

/\*\*

\* this comparator yes has a higher priority than no

\* @author Richelin Metellus

\* @@version 04/20/2017

\*/

public class DecisionComparator implements Comparator<Voter> {

@Override

public int compare(Voter a, Voter b) throws NullPointerException

{

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

throw new NullPointerException("Invalid object reference");

String aDecision = a.getVoted(); // return a string Yes or No

String bDecision = b.getVoted();

int comparisonResult = 0;

switch(aDecision)

{

case "Yes":

{

switch(bDecision)

{

case "Yes" : comparisonResult = 0; break;

case "No" : comparisonResult = 1; break;

}

break;

}

case "No":

{

switch(bDecision)

{

case "Yes" : comparisonResult = -1 ; break;

case "No" : comparisonResult = 0 ; break;

}

break;

}

}

return comparisonResult;

}

}

**Client**

import java.util.Scanner;

/\*\*

\* This class test the running time of different sorting algorithms.

\* to test for smaller uncomment the section. and change limitMax to be greater than limit.

\* @author Richelin Metellus

\* @version 04/21/2017

\*/

public class SortingVoterClient {

public static void main(String[] args) {

int limit = 100000; // size for slower sorting algorithm

Voter[] voters = new Voter[limit];

Voter[] votersCopy;

int limitMax = 1000000; // size for faster sorting. need to modify for test sets

Voter[] largerVoters = new Voter[limitMax];

Voter[] largerVotersCopy;

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

if( i < limit)

{

Voter temp = new Voter();

voters[i] = temp;

largerVoters[i] = temp;

}

else

largerVoters[i] = new Voter(); // create more voter for larger set.

}

ArrayBag<Comparator<Voter>> compBag = new ArrayBag(4);

compBag.add(new PartyComparator()); // index 0

compBag.add(new DecisionComparator()); // index 1

compBag.add(new NameComparator()); // index 2; lower priority index.

System.out.println("");

Comparator idComp = new IdComparator();

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runtime for mergeSort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Comparator voterNameComp = compBag.get(2);

largerVotersCopy = arrayClone(voters);

long mergStartTime = System.currentTimeMillis();

Sort.mergeSort(largerVotersCopy, voterNameComp);

long mergEndTime = System.currentTimeMillis();

long mergElapsedTime = mergEndTime - mergStartTime;

System.out.printf("Runtime of merge Sort(Name) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limitMax,mergElapsedTime);

// System.out.println("Sorted array by Name using merge sort \n------------------");

// printArray(largerVotersCopy);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runtime for quickSort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Comparator voterPartyComp = compBag.get(0);

largerVotersCopy = arrayClone(voters);

long quickStartTime = System.currentTimeMillis();

Sort.quickSortInPlace(largerVotersCopy, voterPartyComp,0,largerVotersCopy.length-1);

long quickEndTime = System.currentTimeMillis();

long quickElapsedTime = quickEndTime - quickStartTime;

System.out.printf("Runtime of quickSort(Party) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limitMax,quickElapsedTime);

// System.out.println("Sorted array by party using quick sort \n------------------");

// printArray(largerVotersCopy);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runtime for bubbleSort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

votersCopy = arrayClone(voters);

long bubbleStartTime = System.currentTimeMillis();

Sort.simpleBubbleSort(votersCopy, idComp);

long bubbleEndTime = System.currentTimeMillis();

long bubbleElapsedTime = bubbleEndTime - bubbleStartTime;

System.out.printf("Runtime of bubbleSort(ID) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limit,bubbleElapsedTime);

// System.out.println("Sorted array by id using bubble sort \n------------------");

// printArray(votersCopy);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runtime for InsertionSort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Comparator votedComp = compBag.get(1);

votersCopy = arrayClone(voters);

long inserStartTime = System.currentTimeMillis();

Sort.insertionSort(votersCopy, votedComp);

long inserEndTime = System.currentTimeMillis();

long inserElapsedTime = inserEndTime - inserStartTime;

System.out.printf("Runtime of insertionSort (Voted) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limit,inserElapsedTime);

// System.out.println("Sorted array by voted status using insertion sort \n------------------");

// printArray(votersCopy);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runtime for SelectionSort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Comparator partyComp = compBag.get(0);

votersCopy = arrayClone(voters);

long selStartTime = System.currentTimeMillis();

Sort.selectionSort(votersCopy, partyComp);

long selEndTime = System.currentTimeMillis();

long selElapsedTime = selEndTime - selStartTime;

System.out.printf("Runtime of selectionSort(Party) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limit,selElapsedTime);

// System.out.println("Sorted array by party using selectionSort\n----------------------");

// printArray(votersCopy);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runtime for radixSort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

largerVotersCopy = arrayClone(largerVoters);

long radixStartTime = System.currentTimeMillis();

Sort.radixSort(largerVotersCopy, compBag);

long radixEndTime = System.currentTimeMillis();

long radixElapsedTime = radixEndTime - radixStartTime;

System.out.printf("Runtime of radix Sort \t for N\t = %,7d \t time \t= %,10d miliseconds \n\n",limitMax,radixElapsedTime);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Printing run time of each soring algorithm \*\*\*\*\*\*\*\*\*\*\*\*

System.out.printf("Runtime of bubbleSort(ID) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limit,bubbleElapsedTime);

System.out.printf("Runtime of insertionSort (Voted) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limit,inserElapsedTime);

System.out.printf("Runtime of selectionSort(Party) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limit,selElapsedTime);

System.out.printf("Runtime of quickSort(Party) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limitMax,quickElapsedTime);

System.out.printf("Runtime of merge Sort(Name) \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limitMax,mergElapsedTime);

System.out.printf("Runtime of radix Sort \t for N\t = %,7d \t time \t= %,10d miliseconds \n",limitMax,radixElapsedTime);

}

//---------------------------------------Utility methods -----------------------------------------

public static void printArray( Voter[] data )

{

for (Voter legalVoter : data) {

System.out.println(legalVoter);

}

System.out.println("");

}

public static Voter[] arrayClone(Voter[] parent)

{

int parentSize = parent.length;

Voter[] clone = new Voter [parentSize];

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

{

clone[i] = parent[i];

}

return clone;

}

}

**Output**

run:

Item successfully added to the bag

Item successfully added to the bag

Item successfully added to the bag

Runtime of merge Sort(Name) for N = 1,000,000 time = 224 miliseconds

Runtime of quickSort(Party) for N = 1,000,000 time = 127 miliseconds

Runtime of bubbleSort(ID) for N = 100,000 time = 136,858 miliseconds

Runtime of insertionSort (Voted) for N = 100,000 time = 38,942 miliseconds

Runtime of selectionSort(Party) for N = 100,000 time = 221,597 miliseconds

Runtime of radix Sort for N = 1,000,000 time = 3,356 miliseconds

Runtime of bubbleSort(ID) for N = 100,000 time = 136,858 miliseconds

Runtime of insertionSort (Voted) for N = 100,000 time = 38,942 miliseconds

Runtime of selectionSort(Party) for N = 100,000 time = 221,597 miliseconds

Runtime of quickSort(Party) for N = 1,000,000 time = 127 miliseconds

Runtime of merge Sort(Name) for N = 1,000,000 time = 224 miliseconds

Runtime of radix Sort for N = 1,000,000 time = 3,356 miliseconds

BUILD SUCCESSFUL (total time: 6 minutes 44 seconds)

run:

Item successfully added to the bag

Item successfully added to the bag

Item successfully added to the bag

Runtime of merge Sort(Name) for N = 1,000,000 time = 2,984 miliseconds

Runtime of quickSort(Party) for N = 1,000,000 time = 786 miliseconds

Runtime of bubbleSort(ID) for N = 1,000,000 time = 35,079,506 miliseconds

Runtime of insertionSort (Voted) for N = 1,000,000 time = 4,596,509 miliseconds

Runtime of selectionSort(Party) for N = 1,000,000 time = 24,394,289 miliseconds

Runtime of radix Sort for N = 1,000,000 time = 7,633 miliseconds

Runtime of bubbleSort(ID) for N = 1,000,000 time = 35,079,506 miliseconds

Runtime of insertionSort (Voted) for N = 1,000,000 time = 4,596,509 miliseconds

Runtime of selectionSort(Party) for N = 1,000,000 time = 24,394,289 miliseconds

Runtime of quickSort(Party) for N = 1,000,000 time = 786 miliseconds

Runtime of merge Sort(Name) for N = 1,000,000 time = 2,984 miliseconds

Runtime of radix Sort for N = 1,000,000 time = 7,633 miliseconds

BUILD SUCCESSFUL (total time: 1,068 minutes 8 seconds)