**COMP2080**

**Assignment 1 (10%)**

**Due: 6th March 2022 11:30PM**

**Given: 13th February 2022**

**Submission Instructions:**

1. **Fill in your full name and student number in the spaces provided further down this page.**
2. **Fill in all results into the tables provided.**
3. **Paste the code for each class and main program into this document after the question.**
4. **Upload this document to blackboard .**

* You **must** have your name and student ID number commented in all code submitted. **Failure to do this will result in a loss of marks**.
* All submissions should at least compile.
* **Non-compiling assignments will NOT be marked and will be given a grade of 0**.
* All your submissions should be suitably documented.

Name: Marco Stevanella

Student ID: 101307949

**Description:**

The purpose of this question is to allow the student to independently compare the run time of various **sorting and searching** 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 **descending** order.

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

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

**Each sort and search 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 two million (2,000,000).
2. For each comparative sorting 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 (1,000) 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 measured using System.nanoTime() for consistency. 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 |
| 1,000 | 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 example if the test data size is fifty (50), the time taken for all the sorting algorithms to sort the data must be shown one after another.

Fill in the tables below with the results of your tests.

Sorting Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Number of Items** | | | | |
| Selection Sort | **50** | **1,000** | **10,000** | **100,000** | **1,000,000** |
| Run 1 | 105300 | 7186500 | 74873200 | 6529806900 | 736669 |
| Run 2 | 43100 | 8468200 | 90839900 | 6686191400 | 983151 |
| Run 3 | 46800 | 6294500 | 72239700 | 6628925800 | 843044 |
| Run 4 | 131600 | 9611800 | 73054300 | 6713199800 | 672302 |
| Run 5 | 112100 | 9451100 | 81743400 | 6682053300 | 706515 |
| **Average** | **87,780 ns** | **8,202,420 ns** | **78,550,100 ns** | **6,639,530,975 ns** | **788,336 ms** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Number of Items** | | | | |
| Insertion Sort | **50** | **1,000** | **10,000** | **100,000** | **1,000,000** |
| Run 1 | 68900 | 4292300 | 16504200 | 1163882900 | 141823 |
| Run 2 | 24000 | 2866000 | 23095900 | 1137029400 | 251402 |
| Run 3 | 45200 | 3542700 | 18430800 | 1148386000 | 122380 |
| Run 4 | 42400 | 3432100 | 16159100 | 1075032700 | 120515 |
| Run 5 | 107000 | 3032600 | 22391200 | 1233487500 | 124261 |
| **Average** | **57,500 ns** | **3,433,140 ns** | **19,316,240 ns** | **1,151,563,700 ns** | **152,076 ms** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Number of Items** | | | | |
| Merge Sort | **50** | **1,000** | **10,000** | **100,000** | **1,000,000** |
| Run 1 | 93100 | 1118900 | 3751900 | 34303800 | 259 |
| Run 2 | 98900 | 1086700 | 3980500 | 32118600 | 378 |
| Run 3 | 74300 | 1184600 | 3958200 | 31229800 | 228 |
| Run 4 | 107100 | 1238200 | 5034600 | 28686600 | 235 |
| Run 5 | 128300 | 1154600 | 7641200 | 30261500 | 234 |
| **Average** | **100,340 ns** | **1,170,566 ns** | **4,873,280 ns** | **31,320,060 ns** | **267 ms** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Number of Items** | | | | |
| Quick Sort | **50** | **1,000** | **10,000** | **100,000** | **1,000,000** |
| Run 1 | 63600 | 634800 | 2694600 | 21569400 | 127 |
| Run 2 | 22400 | 599200 | 2723500 | 21454500 | 169 |
| Run 3 | 35800 | 1000500 | 2093500 | 22986500 | 109 |
| Run 4 | 31700 | 738100 | 3141700 | 17716700 | 123 |
| Run 5 | 68900 | 621700 | 3765800 | 20016700 | 116 |
| **Average** | **44,480 ns** | **718,860 ns** | **2,883,820 ns** | **20,748,760 ns** | **128 ms** |

**Searching 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 two million (2,000,000). **This array must then be sorted in descending order**. (You may use the sorted array from the first part).
2. For each comparative search test, you must search for the number 2,500,000 (two million five hundred thousand). This number does not exist in the array and represents the worst case for both searches. You must repeat the tests 5 times with binary search and linear search, timing how long it takes to complete for each dataset size. Place the results in the table below.

All recorded times **must** be measured using System.nanoTime() for consistency.

Fill in the tables below with the results of your tests.

Searching Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Number of Items** | | | | |
| Binary Search | **50** | **1,000** | **10,000** | **100,000** | **1,000,000** |
| Run 1 | 7800 | 9400 | 14300 | 10800 | 11800 |
| Run 2 | 3500 | 9000 | 7900 | 33300 | 15500 |
| Run 3 | 7200 | 10000 | 11400 | 14600 | 16100 |
| Run 4 | 8700 | 13700 | 8400 | 16000 | 9600 |
| Run 5 | 8400 | 8100 | 11600 | 31300 | 8000 |
| **Average** | **7,120 ns** | **10,040 ns** | **10,720 ns** | **21,200 ns** | **12,200 ms** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Number of Items** | | | | |
| Linear Search | **50** | **1,000** | **10,000** | **100,000** | **1,000,000** |
| Run 1 | 11900 | 85000 | 690900 | 2053500 | 7222300 |
| Run 2 | 6400 | 81700 | 340400 | 3516800 | 8635400 |
| Run 3 | 7300 | 69800 | 437400 | 2200900 | 6135900 |
| Run 4 | 11700 | 173800 | 501500 | 2006700 | 7197300 |
| Run 5 | 8500 | 72800 | 525400 | 3488100 | 6762900 |
| **Average** | **9,160 ns** | **96,620 ns** | **499,120 ns** | **2,653,200 ns** | **7,190,760 ms** |

PASTE YOUR CLASSES FOR THE ASSIGNMENT AND MAIN HERE:

Your code must have your name commented in each new class.

MAIN 🡪 Renamed as App.Java

*/\*\*  
 \* Marco Stevanella - 101307949  
 \* COMP 2080 - ASGMT\_1 (10%)  
 \* Prof. Andrew Rudder  
 \*  
 \* Notes: HI prof Rudder, this assignment was very fun to do. I created a menu that lets the  
 \* user (you) choose the size of the array you want to test. I hope you will like the program.  
 \*/*public class App {  
  
 public static void main(String[] args) {  
  
 // Create CoreData obj  
 CoreData coreData = new CoreData();  
 // fill the CoreData  
 coreData.fillArray();  
 // Create the CoreData Objects by the wanted size  
 CoreData[] sizedCoreData = Tester.*arraySizeBuilder*(coreData);  
 // copy CoreData objects for testing  
 CoreData[][] testCoreData = Tester.*arrayCopierBySize*(sizedCoreData);  
 // executes the sorts and searches tests  
 Tester.*testCoreData*(testCoreData);  
  
 }  
}

Tester.java

/\*  
 \* Marco Stevanella - 101307949  
 \* COMP 2080 - ASGMT\_1 (10%)  
 \* Prof. Andrew Rudder  
 \*/  
  
import java.util.Scanner;  
public class Tester {  
  
 // Arrays Sizes  
 static final int *fifty*= 50;  
 static final int *oneThousand*= 1000;  
 static final int *tenThousand*= 10000;  
 static final int *oneHundredThousand*= 100000;  
 static final int *oneMillion*= 1000000;  
  
 // Number To Search  
 static final int *numberToSearch* = 2500000;  
 static int *indexOfElement*;  
  
 */\*\*  
 \* it creates an array containing CoreData objects by the size of 50, 1000, 10000, 100000 and 1000000.  
 \** ***@param*** *coreData is the original CoreData obj to be trimmed.  
 \** ***@return*** *an array of CoreData objects.  
 \*/* public static CoreData[] arraySizeBuilder(CoreData coreData) {  
 CoreData trimmedCoreData\_50 = coreData.trimBySize(coreData, *fifty*);  
 CoreData trimmedCoreData\_1000 = coreData.trimBySize(coreData, *oneThousand*);  
 CoreData trimmedCoreData\_10000 = coreData.trimBySize(coreData, *tenThousand*);  
 CoreData trimmedCoreData\_100000 = coreData.trimBySize(coreData, *oneHundredThousand*);  
 CoreData trimmedCoreData\_1000000 = coreData.trimBySize(coreData, *oneMillion*);  
 return new CoreData[]{trimmedCoreData\_50, trimmedCoreData\_1000, trimmedCoreData\_10000, trimmedCoreData\_100000, trimmedCoreData\_1000000};  
 }  
  
 */\*\*  
 \* Creates a 2 Dimensional Array containing 4 copies of each size to be tested.  
 \* 4 of size 50, 4 of size 1,000, 4 of size 10,000, 4 of size 100,000, 4 of size 1,000,000  
 \** ***@param*** *coreData is an array containing the CoreData Objects to be copied.  
 \** ***@return*** *a 2 Dimensional Array that have 4 CoreData Objects, for each size, that will be tested by each sorting algo.  
 \*/* public static CoreData [] [] arrayCopierBySize(CoreData [] coreData){  
 String [] sortAlsoType = new String[] {"SelectSort","InsertionSort","MergeSort","QuickSort"};  
 CoreData[][] testCoreData = new CoreData[coreData.length][4];  
 for (int i = 0 ; i < coreData.length ; i++){  
 for (int j = 0 ; j < 4 ; j++){  
 if(coreData[i] != null){  
 int size = coreData[i].getNumElements();  
 switch (size){  
 case *fifty*:  
 CoreData arraySize\_50 = coreData[i];  
 int [] numbers\_50 = new int[arraySize\_50.getMaxElements()];  
 System.*arraycopy*(coreData[i].getCoreData(),0,numbers\_50,0,numbers\_50.length);  
 testCoreData[i][j] = new CoreData(arraySize\_50, sortAlsoType[j],numbers\_50);  
 break;  
 case *oneThousand*:  
 CoreData arraySize\_1000 = coreData[i];  
 int [] numbers\_1000 = new int[arraySize\_1000.getMaxElements()];  
 System.*arraycopy*(coreData[i].getCoreData(),0,numbers\_1000,0,numbers\_1000.length);  
 testCoreData[i][j] = new CoreData(arraySize\_1000, sortAlsoType[j], numbers\_1000);  
 break;  
 case *tenThousand*:  
 CoreData arraySize\_10000 = coreData[i];  
 int [] numbers\_10000 = new int[arraySize\_10000.getMaxElements()];  
 System.*arraycopy*(coreData[i].getCoreData(),0,numbers\_10000,0,numbers\_10000.length);  
 testCoreData[i][j] = new CoreData(arraySize\_10000, sortAlsoType[j], numbers\_10000);  
 break;  
 case *oneHundredThousand*:  
 CoreData arraySize\_100000 = coreData[i];  
 int [] numbers\_100000 = new int[arraySize\_100000.getMaxElements()];  
 System.*arraycopy*(coreData[i].getCoreData(),0,numbers\_100000,0,numbers\_100000.length);  
 testCoreData[i][j] = new CoreData(arraySize\_100000, sortAlsoType[j], numbers\_100000);  
 break;  
 case *oneMillion*:  
 CoreData arraySize\_1000000 = coreData[i];  
 int [] numbers\_1000000 = new int[arraySize\_1000000.getMaxElements()];  
 System.*arraycopy*(coreData[i].getCoreData(),0,numbers\_1000000,0,numbers\_1000000.length);  
 testCoreData[i][j] = new CoreData(arraySize\_1000000, sortAlsoType[j], numbers\_1000000);  
 break;  
 default:  
 System.*out*.println("No arrays found");  
 }  
 }  
 }  
 }  
 return testCoreData;  
 }  
  
 */\*\*  
 \* It Tests the CoreData by timing the execution of each Algorithm, based on the chosen size .  
 \** ***@param*** *testCoreData is a second dimensional array that contains the CoreData Objects to test.  
 \*/* public static void testCoreData (CoreData[][] testCoreData) {  
 int size = *askSizeArrayToTest*(*fifty*, *oneThousand*, *tenThousand*, *oneHundredThousand*, *oneMillion*);  
 long start,  
 end,  
 timeTaken;  
 System.*out*.println("-=-=-=-=-=-=-=-=-=- TESTING SIZE " + size + "-=-=-=-=-=-=-=-=-=-");  
 for (int x = 0; x < testCoreData.length; x++) {  
 for(int j= 0 ; j < testCoreData[x].length ; j++){  
 if (testCoreData[x][j].getNumElements() == size) {  
 if(testCoreData[x][j].getSortingAlgoType().equals("SelectSort")){  
 start = System.*nanoTime*();  
 testCoreData[x][j].selectionSortDesc();  
 end = System.*nanoTime*();  
 timeTaken = end - start;  
 if (size == *oneMillion*){  
 System.*out*.println("Selection Sort: " + timeTaken / 1000000 + " millisecond");  
 }  
 else {  
 System.*out*.println("Selection Sort: " + timeTaken + " nanoseconds");  
 }  
 }  
 if(testCoreData[x][j].getSortingAlgoType().equals("InsertionSort")){  
 start = System.*nanoTime*();  
 testCoreData[x][j].insertionSortDesc();  
 end = System.*nanoTime*();  
 timeTaken = end - start;  
 if (size == *oneMillion*){  
 System.*out*.println("Insertion Sort: " + timeTaken / 1000000 + " millisecond");  
 }  
 else {  
 System.*out*.println("Insertion Sort: " + timeTaken + " nanoseconds");  
 }  
 }  
 if(testCoreData[x][j].getSortingAlgoType().equals("MergeSort")){  
 start = System.*nanoTime*();  
 testCoreData[x][j].mergeSort();  
 end = System.*nanoTime*();  
 timeTaken = end - start;  
 if (size == *oneMillion*){  
 System.*out*.println("Merge Sort: " + timeTaken / 1000000 + " millisecond");  
 }  
 else {  
 System.*out*.println("Merge Sort: " + timeTaken + " nanoseconds");  
 }  
 }  
 if (testCoreData[x][j].getSortingAlgoType().equals("QuickSort")){  
 start = System.*nanoTime*();  
 testCoreData[x][j].quickSortDesc();  
 end = System.*nanoTime*();  
 timeTaken = end - start;  
 if (size == *oneMillion*){  
 System.*out*.println("Quick Sort: " + timeTaken / 1000000 + " millisecond");  
 }  
 else {  
 System.*out*.println("Quick Sort: " + timeTaken + " nanoseconds");  
 }  
 // Execute Binary Search  
 start = System.*nanoTime*();  
 *indexOfElement* = testCoreData[x][j].binarySearch(*numberToSearch*);  
 end = System.*nanoTime*();  
 timeTaken = end - start;  
 System.*out*.println("In the worst case, Binary Search takes " + timeTaken + " nanoseconds to find the number " + *numberToSearch* + " at index "+ *indexOfElement*);  
 // Execute Linear Search  
 start = System.*nanoTime*();  
 *indexOfElement* = testCoreData[x][j].linearSearch(*numberToSearch*);  
 end = System.*nanoTime*();  
 timeTaken = end - start;  
 System.*out*.println("In the worst case, Linear Search takes " + timeTaken + " nanoseconds to find the number " + *numberToSearch* + " at index "+ *indexOfElement*);  
 }  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Displays a Menu to the user that allow to select the size of the CoreData to be tested.  
 \** ***@param*** *fifty CoreData of 50 integer elements.  
 \** ***@param*** *oneThousand CoreData of 1000 integer elements.  
 \** ***@param*** *tenThousand CoreData of 10000 integer elements.  
 \** ***@param*** *oneHundredThousand CoreData of 100000 integer elements.  
 \** ***@param*** *oneMillion CoreData of 1000000 integer elements.  
 \** ***@return*** *an integer representing the size.  
 \*/* public static int askSizeArrayToTest(int fifty, int oneThousand, int tenThousand, int oneHundredThousand, int oneMillion){  
 Scanner input = new Scanner(System.*in*);  
 System.*out*.println("Please Select the size of the array you want to test with Sorting Algorithms: Select a number 1 to 5");  
 System.*out*.println("1- Size 50");  
 System.*out*.println("2- Size 1,000");  
 System.*out*.println("3- Size 10,000");  
 System.*out*.println("4- Size 100,000");  
 System.*out*.println("5- Size 1,000,000");  
 System.*out*.println(" ");  
 System.*out*.println("SIZE? ");  
  
 boolean isUserSelecting = true;  
 int size = 0;  
 while (isUserSelecting){  
 String selection = input.next();  
 switch (selection){  
 case "1":  
 size = fifty;  
 isUserSelecting = false;  
 break;  
 case "2":  
 size = oneThousand;  
 isUserSelecting = false;  
 break;  
 case "3":  
 size = tenThousand;  
 isUserSelecting = false;  
 break;  
 case "4":  
 size = oneHundredThousand;  
 isUserSelecting = false;  
 break;  
 case"5":  
 size = oneMillion;  
 isUserSelecting = false;  
 break;  
 default:  
 System.*out*.println("I think you made a mistake, select a number from 1 to 5");  
 System.*out*.print("SIZE? ");  
 }  
 }  
 return size;  
 }  
  
}

CoreData.java

/\*  
 \* Marco Stevanella - 101307949  
 \* COMP 2080 - ASGMT\_1 (10%)  
 \* Prof. Andrew Rudder  
 \*/  
import java.util.Random;  
  
public class CoreData {  
  
 private int [] coreData;  
 private int numElements;  
 private int maxElements;  
 private String sortingAlgoType;  
  
 public CoreData(){  
 maxElements = 1000000;  
 numElements = 0;  
 coreData = new int[maxElements];  
 }  
 public CoreData(int max){  
 maxElements = max;  
 numElements = 0;  
 coreData = new int[max];  
 }  
  
 public CoreData(CoreData coreData, String algo, int [] numbers){  
 this.maxElements= coreData.maxElements;  
 this.numElements= coreData.numElements;  
 this.coreData = numbers;  
 this.sortingAlgoType = algo;  
 }  
  
 // GETTERS  
 public int getMaxElements() {  
 return maxElements;  
 }  
  
 public int getNumElements() {  
 return numElements;  
 }  
  
 public int[] getCoreData() {  
 return coreData;  
 }  
  
 public String getSortingAlgoType(){  
 return sortingAlgoType;  
 }  
 // Sorting Algorithms  
 // Selection Sort  
  
 public void selectionSortDesc(){  
 for(int start= 0 ; start < numElements ; start ++){  
 // find the index of the biggest id in each iteration of the loop  
 int locBiggest = start;  
 // loop array again to compare Index of Biggest to the rest of the elements in array  
 for(int candidateBigger = start +1 ; candidateBigger < numElements ; candidateBigger++){  
 if(coreData[locBiggest] < coreData[candidateBigger]){  
 locBiggest = candidateBigger;  
 }  
 }  
 // now that we found the biggest element of the iteration, SWAP THE VALUES  
 int temp = coreData[start];  
 coreData[start] = coreData[locBiggest];  
 coreData[locBiggest] = start;  
 }  
 }  
  
 // Insertion Sort  
 public void insertionSortDesc(){  
 for(int start= 1; start < numElements ; start++){  
 // store start element in a temp var for further comparison  
 int temp = coreData[start];  
 // need to set the current pos before start  
 int currentPos = start -1;  
 // Keep the sorting in the boundary of the array, and check if the element is bigger.  
 while(currentPos >= 0 && coreData[currentPos] < temp){  
 // shift the item up to make space, and swap.  
 coreData[currentPos +1] = coreData[currentPos];  
 // decrement the current pos. Mind that if currPos is at index 0 it will go to -1  
 currentPos --;  
 }  
 // item smaller? keep it there.  
 coreData[currentPos +1] = temp;  
 }  
 }  
 // Merge Sort  
 private void merge (int [] arr, int left, int mid, int right){  
 // new array with copied left values  
 int [] leftArray = new int[mid - left+1];  
 // new array with copied right values  
 int [] rightArray = new int[right - mid];  
  
 //Copy left elements into new array 'leftArray'  
 for(int x= 0 ; x < leftArray.length ; x++){  
 leftArray[x] = arr[left +x];  
 }  
 //Copy right elements into new array 'rightArray'  
 for(int x= 0 ; x < rightArray.length ; x++){  
 rightArray[x] = arr[mid+1 +x];  
 }  
  
 // Merge technically starts from here  
 int pointerA = 0;  
 int pointerB = 0;  
 int pointerC =left;  
  
 while(pointerA < leftArray.length && pointerB < rightArray.length){  
 if(leftArray[pointerA] > rightArray[pointerB]){  
 arr[pointerC] = leftArray[pointerA];  
 pointerA++;  
 }  
 else{  
 arr[pointerC] = rightArray[pointerB];  
 pointerB++;  
 }  
 pointerC++;  
 }  
 // just copy the rest of the element in case there is left over  
 while(pointerA < leftArray.length){  
 arr[pointerC] = leftArray[pointerA];  
 pointerA++;  
 pointerC++;  
 }  
 while(pointerB < rightArray.length){  
 arr[pointerC] = rightArray[pointerB];  
 pointerB++;  
 pointerC++;  
 }  
  
 }  
  
 private void mergeSortRecursive(int [] arr, int left, int right){  
 if(left < right){  
 int mid = (left + right) /2;  
 // split on left  
 mergeSortRecursive(arr, left, mid);  
 // split on right  
 mergeSortRecursive(arr, mid+1, right);  
 // now merge left and right together  
 merge(arr, left, mid, right);  
 }  
 }  
  
 public void mergeSort(){  
 mergeSortRecursive(coreData, 0, numElements-1);  
 }  
  
 // Quick Sort  
 public void quickSortDesc(){  
 // here the call the recursive function  
 quickSortRecursive(0, numElements -1);  
 }  
 public void quickSortRecursive(int low, int high){  
 if(low < high){  
 int pivot = partition(low,high);  
 quickSortRecursive(low, pivot - 1);  
 quickSortRecursive(pivot + 1, high);  
 }  
 }  
  
 private int partition (int low, int high){  
 int pivot = coreData[high];  
 int marker = low -1, temp;  
 for (int pres = low; pres < high ; pres++){  
 if(coreData[pres] > pivot){  
 marker++;  
 temp = coreData[marker];  
 coreData[marker] = coreData[pres];  
 coreData[pres] = temp;  
 }  
 }  
 temp = coreData[marker +1 ];  
 coreData[marker+1] = coreData[high];  
 coreData[high] = temp;  
 return marker+1;  
 }  
 // Searching Algorithms  
  
 // Binary Search  
 public int binarySearch(int num){  
 int low = 0;  
 int hi = getNumElements() -1;  
 while(low <= hi){  
 int mid = (low +hi) / 2;  
 if(coreData[mid] == num)  
 return mid;  
 else if (coreData[mid] > num)  
 hi = mid -1;  
 else  
 low = mid +1;  
 }  
 return -1;  
 }  
  
 // Linear Search  
 public int linearSearch(int num){  
 for (int i = 0 ; i < getNumElements() ; i++){  
 if(coreData[i] == num)  
 return i;  
 }  
 return -1;  
 }  
  
 */\*\*  
 \* HELPER FUNCTIONS  
 \*/* public void listItems(){  
 for (int i = 0 ; i < numElements ; i ++){  
 System.*out*.print(coreData[i] + " - ");  
 }  
 System.*out*.println("");  
 }  
  
 public void listItems(int amt){  
 if(amt <= numElements){  
 for (int i = 0 ; i < numElements ; i ++){  
 System.*out*.print(coreData[i] + " - ");  
 }  
 System.*out*.println("");  
 }  
 }  
  
 public void fillArray(){  
 Random rand = new Random(2000000);  
 for (int index = 0 ; index < maxElements ; index ++){  
 int number = rand.nextInt(2000000);  
 coreData [index] = number;  
 numElements++;  
 }  
 }  
  
 public CoreData trimBySize(CoreData array, int size){  
 CoreData trimmedArray = new CoreData(size);  
 for(int i = 0 ; i < size ; i++ ){  
 trimmedArray.coreData[i] = array.coreData[i];  
 trimmedArray.numElements++;  
 }  
 return trimmedArray;  
 }  
  
}

Example Output:

Text

Description automatically generated