Lab 2 – Report

**randomArray –** This generates a random array with **size** as the length of the array and with the random are specified using **range** which is from 0 to the given range subtracting 1. I would first create the integer array with the specified size given, from that I would use a random object to generate a random integer. I then create an array to cycle through the array and populate it with random numbers depending on the range. I used nextInt() to get the next random integer available and assign it to the array and lastly return the generated array.

**hasDuplicate** – A given method that checks if the given array has any duplicate integers. It uses the first number and checks itself to see if it has any duplicate values within itself. The issue with this method is it checks every number even when I know those numbers have been checked before. It also does a redundant check to see if the index of the first for loop isn’t the same as the previous and doesn’t return right after finding one duplicate number.

**hasDuplicateImproved** – This is an improved method that checks if there are any duplicate numbers within the given array. The improved part of this method is that it doesn’t have any redundant checks like the first method did. It makes sure it doesn’t check itself with itself or a previous index that has already been checked before. It also now returns immediately after it finds one duplicate number unlike the previous one which just assigned a value to true and return it after check everything.

**selectionSort –** This is a sorting algorithm that sorts the given array by repeatedly finding the smallest element and placing it in the start of the array. I would first move through the unsorted array, assign the index to the **minimum** variable value as a default, then I would start checking to see if one value is bigger than the other and assign the smaller value’s index to minimum variable. I would then swap the minimum element with the first element and repeat this process until the array is sorted.

**quickSort –** This is a sorting algorithm that sorts the array by using a low index and a high index which then uses a dividing a conquering technique to sort the array. I would first choose an element which is the pivot from the array, in this method I choose the middle index element. I would then re-order the list so that all the elements with the values less than the chosen pivot before and all elements bigger come after, which would I then divide the array. I would then exchange the numbers and move the index to the next position of the sides and then I would repeat this step until its sorted.

**hasDuplicateSinglePass –** This method checks again if the sorted array has any duplicate values. It would check the values right next to each other to see if they’re the same then there are duplicate values within the sorted array. If there are any duplicate values the it would immediately return true as it won’t care how many duplicate values but if it has any.

**hasDuplicateBooleanArray** – This method uses a Boolean array to see if there are any duplicate values within a given unsorted array. I would use a create a Boolean array, I would then use a single loop to check if there are any duplicate values. It would first check if the number is already present first the reason for this is to check if the Boolean array already has true for that value and this is the main part of duplicate checking, if it isn’t it would assign the value of the random integer according to the index of the Boolean array to true to see that this number is present. It would do this until the array ends.

**Main** – It would have the range and the number of numbers you would want. I would create an integer array using **randomArray**. Now I check the created array to see if it has duplicates wit the unfixed **hasDuplicate** method that was given in the lab and the improved **hasDuplicateImproved** method. I then create a new array and assign it its value the same as the array created and used within **hasDuplicateImproved**. would then create a new array and assign its value to the original made and pass it to the selection sort method to sort the array and have visual proof that it’s been sorted and use the **hasDuplicateSinglePass** to check if it has any duplicate values by only doing a single pass through the array and print true it has any duplicate values. I would repeat the same steps with **quickSort**. I then would create a new integer array and assign its value to the original create and pass it to **hasDuplicatedBooleanArray** and print out true if it has a duplicate value.

**Experimental Results: N would represent the number of integers used and M would be the range which is N – 1. The numbers are represented in nanoseconds.** For 10 Integers quicksort was the fastest among the methods and sorting algorithms, and the hasDuplicate method was the slowest overall. For 100 Integers quicksort remained the fastest sorting algorithm and method, while hasDuplicate remained the slowest. For 1000 integers both method that a single pass through the array were around the same speed while selection sort and the hasDuplicate method became tied to the slowest method. For 10000 Integers hasDuplicate remained the slowest with selection sort a little faster, while the method that both has single pass loops remained the fastest. For 10000 Integers hasDuplicate method remained the slowest while hasDuplicateSinglePass became the fasts. Overall hasDuplicate method remained the slowest method, selection sort remained the slowest sorting algorithm and the quick sort was faster with integers while the methods who had just one looped became the fastest the more integers were introduced.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | N = 10 | N = 100 | N = 1000 | N = 10000 | N = 10000 |
| **hasDuplicate** | 145140 | 313619 | 3204375 | 68331202 | 6141647108 |
| **hasDuplicateImproved** | 17869 | 20786 | 44855 | 203488 | 960547 |
| **selectionSort** | 3647 | 81687 | 3296273 | 53291022 | 4605069747 |
| **quickSort** | 3282 | 13493 | 214792 | 909857 | 16476989 |
| **hasDuplicateSinglePass** | 47407 | 18234 | 19328 | 41280 | 21515 |
| **hasDuplicateBooleanArray** | 31362 | 42302 | 19693 | 41938 | 67465 |

**Conclusion:** I was able to understand how to properly check for duplicate in many ways while also improving on how to use the selection sort and quick sort. With the stats above I was able to conclude that the quick sort was faster the selection sort.

import java.util.Random;

/\*\*

\* Objective - To determine if an array of integers in a certain range contains duplicate elements.

\* Date Last Modified - 2/12/18

\* Course - Data Structures

\* Lab 2

\* Instructor - Olac Fuentues

\* Teaching Assistants - Zakia Al Kadri

\* @author Isaias Leos Ayala

\*/

public class Lab2

{

/\*\*

\* Array generated with random numbers

\* @param size size of the array

\* @param range the range from 0 to m - 1

\* @return generated array

\*/

public static int[] randomArray(int size, int range)//O(N)

{

int[] generatedArray = new int[size];//generate array

Random random = new Random();//random object to generate random numbers

for(int i = 0; i < generatedArray.length; i++)//populate array with random numbers

{

generatedArray[i] = random.nextInt(range);

}

return generatedArray;

}

/\*\*

\* Checks if there is a duplicate number in the array.

\* @param randomIntArray generated array with random numbers at the range of m - 1

\* @return

\*/

public static boolean hasDuplicates(int[] randomIntArray)//O(N^2)

{

boolean duplicates = false;

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

{

for(int j = 0; j < randomIntArray.length; j++)

{

if(randomIntArray[i] == randomIntArray[j] && i != j)

{

duplicates = true;

}

}

}

return duplicates;

}

/\*\*

\* Improved method that checks if there is a duplicate number in the array.

\* @param randomIntArray generated array with random numbers at the range of m - 1

\* @return

\*/

public static boolean hasDuplicatesImproved(int[] randomIntArray)//O(N^2)

{

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

{

for(int j = i + 1; j < randomIntArray.length; j++)//i + 1 so it doesn't check itself or the previous index

{

if(randomIntArray[i] == randomIntArray[j])//remove && i != j

{

return true;

}

}

}

return false;

}

/\*\*

\* A sorting algorithm that sorts an array by repeatedly finding the minimum element from unsorted part

\* and placing it at the beginning.

\* @param randomIntArray generated array with random numbers at the range of m - 1

\*/

public static void selectionSort(int[] randomIntArray)//O(N^2)

{

for(int i = 0; i < randomIntArray.length; i++)//Move the the unsorted array

{

int minimum = i;//Find the minimum element

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

{

if(randomIntArray[j] < randomIntArray[minimum])

{

minimum = j;

}

}

int temp = randomIntArray[minimum];//Swap the minimum element with the first

randomIntArray[minimum] = randomIntArray[i];

randomIntArray[i] = temp;

}

}

/\*\*

\* QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given

\* array around the picked pivot.

\* @param array generated array with random numbers at the range of m - 1

\* @param lowIndex first element of the array

\* @param highIndex second element of the array

\*/

public static void quickSort(int[] array, int lowIndex, int highIndex)//O(N^2)

{

int lowPivot = lowIndex;

int highPivot = highIndex;

int pivot = array[lowIndex + (highIndex - lowIndex) / 2];//Get the pivot number | Pivot is the middle index

while(lowPivot <= highPivot)// Divide into two arrays

{

while(array[lowPivot] < pivot)

{

lowPivot++;

}

while(array[highPivot] > pivot)

{

highPivot--;

}

if(lowPivot <= highPivot)//If current element is smaller than the pivot

{

int temp = array[lowPivot];

array[lowPivot] = array[highPivot];

array[highPivot] = temp;

//move index to next position on both sides

lowPivot++;

highPivot--;

}

}

if(lowIndex < highPivot)

{

quickSort(array, lowIndex, highPivot);

quickSort(array, lowPivot, highIndex);

}

}

/\*\*

\* Checks if the sorted array has any duplicate values

\* @param A array with randomly generated integers that have been sorted

\* @return

\*/

public static boolean hasDuplicatesSinglePass(int[] A)//O(N)

{

for(int i = 0; i < A.length - 1; i++)//check the array

{

if(A[i] == A[i + 1])//compare current to next variable

{

return true;//if duplicates are present

}

}

return false;//if duplicates aren't present

}

/\*\*

\* Uses a boolean array to check if the provided array has any duplicate numbers.

\* @param A generated array with random numbers at the range of m - 1

\* @param m the range from 0 to m - 1

\* @return

\*/

public static boolean hasDuplicatedBooleanArray(int[] A, int m)

{

boolean[] isNumberPresent = new boolean[m];//keep track of what number are present

for(int i = 0; i < A.length; i++)//single pass

{

if(isNumberPresent[A[i]] == true)//to check if the number is already present

{

return true;//if it is, end and return true.

}

else

{

isNumberPresent[A[i]] = true;//sets the index to true if not a duplicate

}

}

return false;//no duplicates

}

public static void main(String[] args)

{

int range = 100;

int count = 10;

int[] originalArray = randomArray(count, range);//random generated

System.out.println("Unimproved Duplicate Check: " + hasDuplicates(originalArray));//Unimproved Duplicate Pass

System.out.println("Improved Duplicate Check: " + hasDuplicatesImproved(originalArray));//Improved Duplicate Pass

int[] selectionSortArray = originalArray;//new array for selection sort

selectionSort(selectionSortArray);//Selection Sort

System.out.print("Selection Sort: ");

for(int i = 0; i < selectionSortArray.length; i++)//visual proof that its working

{

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

}

System.out.println("");

System.out.println("Single Pass Duplicate Check: " + hasDuplicatesSinglePass(selectionSortArray));//Check the if the array has duplicates

int[] quickSortArray = originalArray;//new array for quicksort

quickSort(quickSortArray, 0, quickSortArray.length - 1);//Quicksort

System.out.print("Quick Sort: ");

for(int i = 0; i < quickSortArray.length; i++)//visual proof that its working

{

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

}

System.out.println("");

System.out.println("Single Pass Duplicate Check: " + hasDuplicatesSinglePass(quickSortArray));//Check the if the array has duplicates

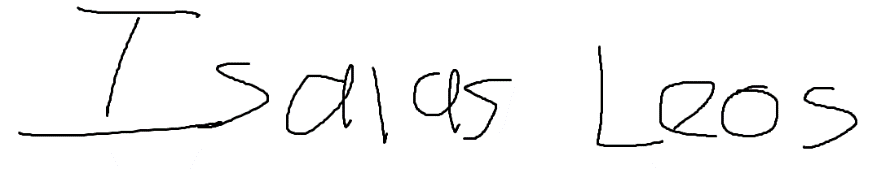
int[] booleanArray = originalArray;//new array

System.out.println("Boolean Array Duplicate Check: " + hasDuplicatedBooleanArray(booleanArray, range));//Check the if the array has duplicates /w boolean array

}

}

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

****

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_