**Department of Computing Sciences**

**The College at Brockport, State University of New York**

**CSC 205 (Fundamentals of Data Structures)**

**Partner 1 Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Partner 2 Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Lab Exercise 6**

**Objective: Sorting Algorithms (second set) comprehension**

Here is the enhanced (full) code for the “GenericSortingMethods” class, containing the second set of sorting algorithms (Merge Sort and Quick Sort) we have seen:

**package sortingsearching;**

**/\*\***

**\* Implement the generic versions of sorting algorithms**

**\* Selection Sort, Insertion Sort and Bubblesort**

**\* Using the Comparable Interface**

**\***

**\* @author T.M. Rao**

**\* @version September 2018**

**\***

**\*/**

**public class GenericSortingMethods**

**{**

**//-----------------------------------------------------------------**

**private static int numComps = 0;**

**/\*\***

**\* swap procedure**

**\* @param Comparable[] a -- the array in which the swapping is done**

**\* @param i**

**\* @param j -- the two locations in the array that will be swapped**

**\*/**

**public static void swap(Comparable[] a, int i, int j)**

**{**

**Comparable temp = a[i];**

**a[i] = a[j];**

**a[j] = temp;**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* findSmallestLoc -- Locates the smallest element in the array between**

**\* array positions start and end**

**\* @param int[] a -- the array in which the locating is to be done**

**\* @param start -- starting location**

**\* @param end -- end location**

**\* This is called by the selectionSort method**

**\*/**

**public int select(Comparable[] a, int k)**

**{**

**//find the location of the smallest element in the array**

**//between position k and the end of the array**

**int smallestPos = k;**

**for (int j = k; j < a.length; j++)**

**{**

**numComps++;**

**if (a[j].compareTo(a[smallestPos]) < 0)**

**smallestPos = j;**

**}**

**return smallestPos;**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* The selection sort method: sorts the given unsorted array a**

**\* @param Comparable[] a -- the array to be sorted**

**\* calls: findSmallestLoc and swap methods**

**\* Time Complexity: O(N \*\* 2)**

**\*/**

**public void selectionSort(Comparable[] a)**

**{**

**int pos; //position of the smallest elelemt**

**int temp; //temp variable for swap**

**numComps = 0;**

**for (int k = 0; k < a.length-1; k++)**

**{**

**pos = select(a, k); //find the smallest element**

**swap(a, k, pos);**

**}**

**System.out.println("NumComps = "+numComps);**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* The insertion sort method: Sorts the given array a**

**\***

**\* @param Comparable[] a -- the array to be sorted**

**\* calls: insert and swap methods**

**\* Time Complexity: O(N \*\* 2)**

**\*/**

**public void insertionSort(Comparable a[])**

**{**

**numComps = 0;**

**for (int k = 1; k < a.length; k++)**

**{**

**insert(a, a[k], k-1);**

**}**

**System.out.println("NumComps = "+numComps);**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* The insert method. It is assimed that the part of the array between**

**\* position 0 and position end is sorted. It inserts a new element x into**

**\* the array, by moving other elements down and making space for the new**

**\* element. It maintains the increasing order of the array.**

**\* @param Comparable[] a - the sorted array into which insertion is made**

**\* @param x -- the integer that is to be inserted**

**\* @param end -- array is assumed to be sorted from 0 to end**

**\***

**\* Called by: the insertionSort method**

**\*/**

**public void insert(Comparable a[], Comparable x, int end)**

**{**

**int k = end;**

**boolean done = false;**

**while ((k >= 0) && (! done))**

**{**

**// As long as the element in the array is > x, keep moving**

**// it one position down.**

**numComps++;**

**if (a[k].compareTo(x)>0)**

**{**

**a[k+1] = a[k];**

**k--;**

**}**

**// Stop moving elements down as soon as you find an element >= x**

**else**

**{**

**done = true;**

**}**

**}**

**// Insert the new element at position k+1**

**a[k+1] = x;**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* pass: It makes one "pass" over the data, starting at position 0 up to**

**\* position end. It compares a[0] with a[1], a[1] with a[2] ... and**

**\* a[end-1] with a[end], and makes a swap if necessary. In case it swaps**

**\* it will record that fact by setting didSwap to true. This is what is**

**\* returned by the method.**

**\* @param Comparable[] a -- the array to be sorted**

**\* @param end -- 0 to end is the part of the array to focus on.**

**\* The elements at positions end+1 ... will be sorted.**

**\*/**

**public static boolean pass(Comparable[] a, int end)**

**{**

**boolean didSwap = false;**

**for (int k = 0; k < end; k++)**

**{**

**numComps++;**

**if (a[k].compareTo(a[k+1]) > 0)**

**{**

**swap(a, k, k+1);**

**didSwap = true;**

**}**

**}**

**return didSwap;**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* The bubblesort method. It makes as many "pass"es as needed over the**

**\* array until it is sorted.**

**\* @param Comparable[] a -- the array to be sorted**

**\* Time Complexity: O(N \*\* 2)**

**\*/**

**public static void bubbleSort(Comparable[] a)**

**{**

**numComps = 0;**

**boolean swapped = true;**

**// Keep track of whether we made a swap in this pass**

**// Initially set to true, because we have to make at**

**// least one pass**

**int passCount = 0;**

**// Counts the number of passes. We need no more than**

**// n-1 passes, where n is the length of the array**

**int end = a.length-1;**

**// 0 to end is the part of the array to focus on**

**// Loop as long as passCount < n-1 and we have made a swap**

**while ((passCount < a.length-1) && (swapped))**

**{**

**swapped = pass(a, end);**

**passCount++;**

**end--;**

**}**

**System.out.println("NumComps = "+numComps);**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* This is the main quicksort method. It just sets the parameters**

**\* properly and calls another quicksort method to sort it**

**\*/**

**public static void quickSort(Comparable[] a)**

**{**

**numComps = 0;**

**quickSort(a, 0, a.length-1);**

**System.out.println("NumComps = "+numComps);**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* This is the recursive quicksort method. It first partitions the**

**\* array into two parts such that all elements in the first part**

**\* are less than all elements in the second part. It then recursively**

**\* calls itself on both.**

**\* @param Comparable[] a -- the array to be sorted**

**\* Worst Case Time Complexity: O(N \*\* 2)**

**\* Average case: O(N log N)**

**\*/**

**public static void quickSort(Comparable[] a, int first, int last)**

**{**

**if (first < last)**

**{**

**int pivotIndex = partition(a, first, last);**

**//System.out.println("first = "+first+ " last = "+last**

**//+ " pivot = "+pivotIndex);**

**quickSort(a, first, pivotIndex-1);**

**quickSort(a, pivotIndex+1, last);**

**}**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* This is a private supporting method for quicksort. It just does**

**\* the partitionaing of the array.**

**\* @param first: the start index of the sub-array**

**\* @param last: the end index of the sub-array**

**\*/**

**private static int partition(Comparable[] a, int first, int last)**

**{**

**Comparable pivot = a[first];**

**int up = first;**

**int down = last;**

**do**

**{**

**while ( (up < last) && (pivot.compareTo(a[up]) >= 0))**

**{**

**numComps++;**

**//System.out.println("Numcomps = "+numComps);**

**up++;**

**}**

**while ( pivot.compareTo(a[down]) < 0)**

**{**

**numComps++;**

**//System.out.println("Numcomps = "+numComps);**

**down--;**

**}**

**if (up < down)**

**swap(a, up, down);**

**} while (up < down);**

**swap(a, first, down);**

**return down;**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* The linear search (sequential search method. If the array is not**

**\* sorted, this is the only way you can search.**

**\* @param Comparable[] a -- the array in which we are searching**

**\* @param x -- The value that we are searching for**

**\* Time Complexity: O(N)**

**\*/**

**public int linSearch(Comparable[] a, Comparable x)**

**{**

**numComps = 0;**

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

**{**

**if (a[i].compareTo(x) == 0) return i;**

**numComps++;**

**}**

**return -1;**

**}**

**//-----------------------------------------------------------------**

**/\*\***

**\* binary search method. This will work only if the array is sorted.**

**\* @param Comparable[] a -- the array in which we are searching**

**\* @param x -- The value that we are searching for**

**\* Time complexity O(log N)**

**\*/**

**public int binSearch(Comparable[] a, Comparable x)**

**{**

**int h = a.length-1;**

**int l = 0;**

**int mid = 0;**

**numComps = 0;**

**while (l <= h)**

**{**

**mid = (l+h)/2;**

**numComps++;**

**if (x.compareTo(a[mid]) == 0) return mid;**

**else if (x.compareTo(a[mid]) > 0) l = mid+1;**

**else h = mid-1;**

**}**

**return -1;**

**}**

**/\*\***

**\* It is assumed that:**

**\* Two parts of the array a: a[start1] to a[start2-1] and**

**\* a[start2] to a[end2] are sorted. end2 >= start2 >= start1**

**\* This will merge those two parts to make one long**

**\* sorted part between a[start1] to a[end2]**

**\*/**

**public void merge(Comparable[] a, int start1, int start2, int end2)**

**{**

**int c1 = start1; //Counter for the first half of the sorted array**

**int c2 = start2; //Counter for the second half of the sorted array**

**int sortedCounter = 0; //Counter for the sorted array**

**//Temporary array for the merged elements**

**Object [] b = new Object[end2 - start1 +1];**

**//Scan the sorted array a and keep filling b by elements from a or**

**//b, depending on which is smaller**

**while (c1 < start2 && c2 <= end2)**

**{**

**//If a[c1] < a[c2], move a[c1] into b and increment c1**

**if (a[c1].compareTo(a[c2]) < 0)**

**{**

**b[sortedCounter] = a[c1];**

**sortedCounter++;**

**c1++;**

**numComps++;**

**}**

**//If a[c1] > a[c2], move a[c2] into b and increment c2**

**else if (a[c1].compareTo(a[c2]) > 0)**

**{**

**b[sortedCounter] = a[c2];**

**sortedCounter++;**

**c2++;**

**numComps++;**

**}**

**//If a[c1] == a[c2], move both a[c1] and a[c2] into b and**

**//increment both c1 and c2**

**else**

**{**

**b[sortedCounter] = a[c1];**

**sortedCounter++;**

**b[sortedCounter] = a[c2];**

**sortedCounter++;**

**c1++;**

**c2++;**

**}**

**}**

**//Copy the rest of the elements from first half**

**while (c1 < start2)**

**{**

**b[sortedCounter] = a[c1];**

**sortedCounter++;**

**c1++;**

**}**

**//Copy the rest of the elements from second half**

**while (c2 <= end2)**

**{**

**b[sortedCounter] = a[c2];**

**sortedCounter++;**

**c2++;**

**}**

**//Finally copy all the elements from b to a.**

**for (int k = start1; k <= end2; k++)**

**{**

**a[k] = (Comparable)b[k-start1];**

**}**

**}**

**/\*\***

**\* This is the top-level call to the merge sort algorithm**

**\* @param a: the array to be sorted**

**\*/**

**public void mergeSort(Comparable[] a)**

**{**

**numComps = 0;**

**mergeSort(a, 0, a.length-1);**

**System.out.println("NumComps = "+numComps);**

**}**

**/\*\***

**\* This is the recursive method that actually sorts**

**\*/**

**public void mergeSort(Comparable[] a, int start, int end)**

**{**

**int len = end - start + 1;**

**int half = len/2;**

**if (len > 1)**

**{**

**mergeSort(a, start, start+half-1); //sort first half**

**mergeSort(a, start+half, end);//sort second half**

**merge(a, start, start+half, end);**

**}**

**}**

**public static int getNumComps()**

**{**

**return numComps;**

**}**

**}**

Type (or copy-paste) this code into Eclipse, into the package called “sortingsearching”. Note that the most relevant methods we will be focusing on in this code have the following signatures:

**public static void quickSort(Comparable[] a)**

**public static void mergeSort(Comparable[] a)**

Now, as we practiced in class, we may need to “hand execute” the code in these methods to understand the sorting algorithms better. Complete these templates:

**MERGESORT DEMO**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index 🡪 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Initial Array 🡪 | 42 | 75 | 42 | 28 | 7 | 64 | 56 | 65 |
| Merge cycle 1 |  |  |  |  |  |  |  |  |
| Merge cycle 2 |  |  |  |  |  |  |  |  |
| Merge cycle 3 |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index 🡪 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Initial Array 🡪 | 55 | 15 | 90 | 53 | 87 | 73 | 58 | 14 |
| Merge cycle 1 |  |  |  |  |  |  |  |  |
| Merge cycle 2 |  |  |  |  |  |  |  |  |
| Merge cycle 3 |  |  |  |  |  |  |  |  |

**QUICKSORT DEMO (Just show one execution of the Partition algorithm. It won’t sort the array completely. It just breaks the array into two parts such that all elements to the left of the pivot are less than or equal to the pivot, and all to the right are greater.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index 🡪 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Initial Array 🡪  Pivot = | 42 | 75 | 42 | 28 | 7 | 64 | 56 | 65 |
| lInd = rInd =  lInd < rInd = T or F  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |
| lInd = rInd =  lInd < rInd = T or F  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |
| lInd = rInd =  lInd < rInd = T or F  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |
| Final Swap  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index 🡪 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Initial Array 🡪  Pivot = | 55 | 15 | 90 | 53 | 87 | 73 | 58 | 14 |
| lInd = rInd =  lInd < rInd = T or F  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |
| lInd = rInd =  lInd < rInd = T or F  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |
| lInd = rInd =  lInd < rInd = T or F  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |
| Final Swap  swap a[ ] & a[ ] |  |  |  |  |  |  |  |  |

**Submit these pages containing the blank templates with the answers filled in**

Now, write a class called “SortComparator” which contains the public static void main method. It should do the following:

1. Create an Integer array of size 128. Call it “a”.
2. Fill it up with random numbers between 0 and 100,000. Use

**for (int k = 0; k < 128; k++)**

**a[k] = (int) (Math.random() \* 100000);**

to generate your random numbers.

1. Make one more copy of “a”, call it “b”. Don’t just write b = a. Create brand new array “b” of the same size as that of “a” and copy the elements of “a” into it using for loops. (Note: as a result, both these arrays – a and b finally contain exactly the same elements)
2. Run the Merge sort algorithm on “a” and note the number of comparisons.
3. Run the Quick sort algorithm on “b” and note the number of comparisons.

Repeat 1-6 above for sizes 512, 2048, and 8192. Based on the results you observe from these runs, complete the following tables and submit them:

Fill the following tables with results from your runs:

**Number of Comparisons**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting Method | ArraySize =128 | ArraySize = 512 | ArraySize = 2048 | ArraySize = 8192 |
| Merge Sort | 739 | 3963 | 19904 | 95423 |
| Quick Sort | 919 | 4946 | 26287 | 122659 |
| Which method had the least number of comparisons in each case? | Merge Sort | Merge Sort | Merge Sort | Merge Sort |

Run the same experiment again (Just run the program again. Results may be different because you are generating whole new sets of random arrays) and note down the results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting Method | ArraySize =128 | ArraySize = 512 | ArraySize = 2048 | ArraySize = 8192 |
| Merge Sort | 729 | 3949 | 19903 | 95407 |
| Quick Sort | 979 | 4929 | 29662 | 124184 |
| Which method had the least number of comparisons in each case? | Merge Sort | Merge Sort | Merge Sort | Merge Sort |

**Objective: Comprehension of Complexity**

**How to measure the time complexity of an algorithm?**

1. Identify an important operation in the algorithm that is executed most frequently.
2. Express the number of times it is executed as a function of N.
3. Convert this expression into the Big-O notation.

**A. For each of the three fragments of code, what is its worst-case time complexity, in the form "*O*(…)". (Use the given solution to the first problem as a model)**

**//----------------- This is a sample problem – solved ------**

**Given an array a of size N, what is the complexity?**

**int i=n, x=0;**

**while (i>0) {**

**i--;**

**x = x + a[i];**

**}**

What is the typical operation performed? \_\_\_addition\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Expression showing the number of times it is performed\_\_\_\_ N \_\_\_\_\_\_\_

Time Complexity in Big-O Notation\_\_\_\_ O(N) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//1.--------------------------------------------------------------**

**Given three 2-dimensional arrays A, B, C of size N x N, what is the complexity?**

**for (int i=0; i<N; i++) {**

**for (int k=0; k<N; k++) {**

**int x=0;**

**for (int j=0; j<N; j++) {**

**x = x + A[i][j] \* B[j][k];**

**}**

**C[i][k] = x;**

**}**

**}**

What is the 'operation' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Expression showing the number of times it is performed\_\_\_\_\_\_\_\_\_\_\_

Time Complexity in Big-O Notation\_\_\_ O( ) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//2.--------------------------------------------------------------**

**Given an array of size N, what is the complexity?**

**int m = A[0];**

**for (int i=1; i<N; i++)**

**{ if (A[i] > m) m = A[i]; }**

**int k = 0;**

**for (int i=0; i<N; i++)**

**{ if (A[i] == m) k++; }**

What is the 'operation' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Expression showing the number of times it is performed\_\_\_\_\_\_\_\_\_\_\_

Time Complexity in Big-O Notation O( ) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**//3.--------------------------------------------------------------**

**int mult(int N, int M)**

**{**

**int s = 0;**

**while (N > 0)**

**{**

**s = s + M;**

**N = N-1;**

**}**

**return s;**

**}**

What is the 'operation' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Expression showing the number of times it is performed\_\_\_\_\_\_\_\_\_\_\_

Time Complexity in Big-O Notation O( ) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4**. **The following method computes the value of x raised to the power of n, where n is a positive integer. (For example, power(3, 4) would return the value of 34, i.e. 81. What is the time complexity of this method (as a function of n?)**

**public int power(int m, int n) //m raised to the power n**

**{**

**int res = 1; int p = m;**

**while (n != 0)**

**{**

**if (n%2 == 1)**

**res = res \* p;**

**p = p\*p;**

**n = n/2;**

**}**

**return res;**

**}**

What is the 'operation' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Expression showing the number of times it is performed\_\_\_\_\_\_\_\_\_\_\_

Time Complexity in Big-O Notation O( ) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**B. For each of the following problems, think about an algorithm to solve it. Write a segment of java code showing how you would solve it (i.e. code your algorithm in Java) analyze the algorithm and indicate its time complexity. Assume that a and b are integer arrays of size N.**

**Here is a sample solved problem: An algorithm to decide if a given number is a prime number:**

**public boolean isPrime(int n)**

**{**

**if (n == 2)**

**return true; //Done, we already know n is prime**

**int div = 3; //start the possible divisors at 3**

**while (div < Math.sqrt(n))**

**{**

**if ( n % div == 0) //we found a divisor**

**return false; //so n is not prime**

**div = div + 2; //if not, try the next number**

**}**

**return true; //No divisor was found. So, num is a prime**

**}**

**Analysis of the algorithm (example):**

Typical Operation: % or +

Number of times performed: Goes up to sqrt(n), increments by 2, therefore in the worst case this is done n1/2 / 2 times

Time complexity = n1/2

1. Print the value at the middle of the array (You can assume that the length of the array is an odd number). (Example: if a = {5, 3, 6, 2, 1} it should print 6.)

**public static void printMiddleArray(Comparable[] array) {**

**int length = array.length;**

**if (length % 2 == 1) {**

**System.out.print(array[(length/2) + 1]);**

**}**

**}**

**Analysis of the algorithm:**

Typical Operation: System print

Number of times performed: worst case (complexity of 3)

Time complexity: O(1)

1. Print the smallest element of the array ***a*** (assuming that a is NOT sorted)

**public static void printSmallestInArray (int [] array) {**

**int length = array.length;**

**int smallest = array[length – 1];**

**while (length > 0) {**

**smallest = smallest > array[length - 1]? array[length – 1]: smallest;**

**length--;**

**}**

**System.out.print(smallest);**

**}**

**Analysis of the algorithm:**

Typical Operation: Check each element in the array for the smallest element

Number of times performed: n + 3

Time complexity: O(n)

1. Print the smallest element of the array *a* (assuming a is sorted in ascending order)

**System.out.print(a[0]);**

**Analysis of the algorithm:**

Typical Operation: print first element in the array

Number of times performed: 1

Time complexity: O(1)

1. A "pair" is two consecutive locations in the array having the same value. (Note: {2, 3, 3, 4, 2} has a pair 3, 3. But, {4, 5, 2, 9, 2} doesn't have a pair). Find if the array *a* has a pair in it.

**public static Boolean hasPair(Comparable [] array) {**

**for (int count = 0; count < array.length - 1; count++) {**

**if (array[count].compareto(array.[count + 1]) == 0)**

**return true;**

**}**

**return false;**

**}**

**}**

**Analysis of the algorithm:**

Typical Operation: check if there are any pairs in the array

Number of times performed: worst case , n

Time complexity: O(n)

1. Find if any two values in the array are the same (i.e. *a* has a repeated element). For example {2,3,3,4,2} has two repeated elements but {4,5,2,9,6} doesn't have repeated elements.

**public static boolean hasRepeat(Comparable [] array) {**

**int length = array.length;**

**for (int count = 0; count < length; count++) {**

**for (int num = count + 1; num < length; num++) {**

**if (array[count].compareto(array[num]) == 0)**

**return true;**

**}**

**}**

**return false;**

**}**

**Analysis of the algorithm:**

Typical Operation: check if there are any repeats in the array

Number of times performed: worst case n(n-1/2)

Time complexity: O(n^2)

f. A "Magic Square" is a nxn matrix, containing distinct positive integers, whose rows, columns and diagonals add up to the same number. Write an algorithm to determine whether a matrix is a magic square. Determine the time complexity of your algorithm

For example, the following is a magic square whose rows/columns/ diagonals add up to 34.

16 03 02 13

05 10 11 08

09 06 07 12

04 15 14 01

(Hint: Write separate sub-algorithms such as rowSum(k), colSum(k), mainDiagonalSum(), reverseDiagonalSum(), etc. to compute row-sums, column-sums, and diagonal sums)

**public static boolean isMagicSquare(int [] [] array) {**

**int length = array[0].length;**

**boolean bool = false;**

**int sum = mainDiagonalSum(array);**

**if (sum == reverseDiagonalSum(array) {**

**int [] row = rowSum(array);**

**int [] col = colSum(array);**

**int count = 0;**

**while (count < length) {**

**if ((row[count] == sum) && (col[count] == sum))**

**bool = true;**

**else**

**bool = false;**

**count++;**

**}**

**}**

**return bool;**

**}**

**public static int[] rowSum(int [][] a) {**

**int length = a[0][0].length;**

**int [] row = new int[length];**

**for (int count = 0; count < length; count++) {**

**for(int num = 0; num < a[count].length; num++) {**

**row[count] += a[num][count];**

**}**

**}**

**return row;**

**}**

**public static int[] colSum(int [][] a) {**

**int length = a[0][0].length;**

**int [] col = new int[length];**

**for (int count = 0; count < length; count++) {**

**for(int num = 0; num < a[count].length; num++) {**

**col[count] += a[count][num];**

**}**

**}**

**return col;**

**}**

**public static int[] colSum(int [][] a) {**

**int length = a[0][0].length;**

**int [] col = new int[length];**

**for (int count = 0; count < length; count++) {**

**for(int num = 0; num < a[count].length; num++) {**

**col[count] += a[count][num];**

**}**

**}**

**return col;**

**}**

**public static int mainDiagonalSum(int [][] a) {**

**int length = a[0].length;**

**int diag = 0;**

**for (int count = 0; count < length; count++) {**

**diag += a[count][count];**

**}**

**return diag;**

**}**

**public static int reverseDiagonalSum(int [][] a) {**

**int length = a[0].length;**

**int diag = 0;**

**for (int count = length; count >= 0; count--) {**

**diag += a[count][count];**

**}**

**return diag;**

**}**

**Analysis of the algorithm:**

Typical Operation: (check sum of rows, columns, sum of diagonals so they all equal to sum)

Number of times performed: 2n^2 + 4n

Time complexity: O(n^2)

**SUBMISSION:** Submit the completed tables above. Also, you do not have to present exact working code (demonstrated to be working after being tested with various input data) for the complexity part. But be sure to TYPE UP the algorithms (they should be coded in Java – you can just type up a Java method for each problem a – f above, it is not necessary to put it into a class) and also type up your analysis below each method, in a style that is as close as you can be to the analysis shown for the ‘isPrime()’ method above.