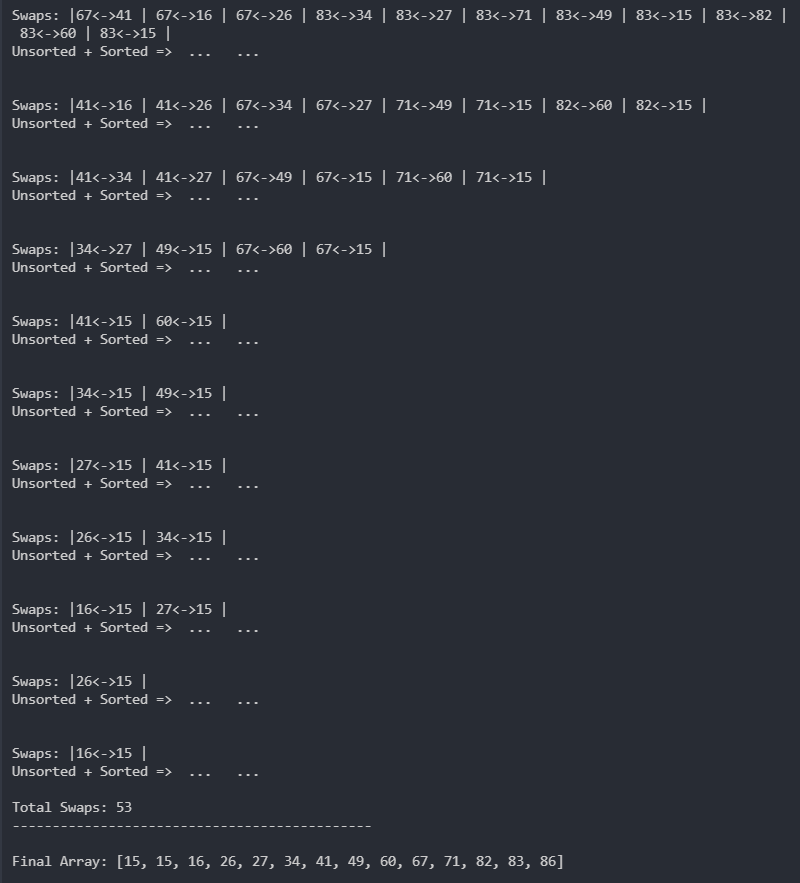
|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Pratik Pujari** | | |
| **UID no.** | **2020300054** | **Class:** | **Comps C Batch** |
| **Experiment No.** | 3 | | |

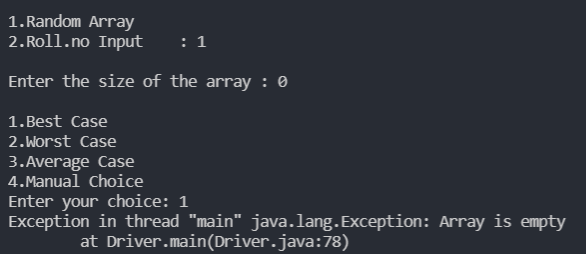
|  |  |
| --- | --- |
| **AIM:** | To implement the Bubble sort algorithm using Recurrence Relations concepts. |
| **THEORY:** | **What is Recursion?**  The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.  **What is base condition in recursion?**  In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.   int fact(int n)  {  if (n < = 1) // base case  return 1;  else  return n\*fact(n-1);  }  In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.  **What is the difference between direct and indirect recursion?**  A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in Table 1.  **// An example of direct recursion**  void directRecFun()  {  // Some code....  directRecFun();  // Some code...  }  **// An example of indirect recursion**  void indirectRecFun1()  {  // Some code...  indirectRecFun2();  // Some code...  }  void indirectRecFun2()  {  // Some code...  indirectRecFun1();  // Some code...  }  **Recurrence Relation**  A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.  **For Example**, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.  T (n) = θ (1) if n=1  2TDAA Recurrence Relation + θ (n) if n>1  **What is Bubble Sort?**  Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.  How Bubble Sort Works?  We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.  Bubble Sort  Bubble sort starts with very first two elements, comparing them to check which one is greater.  Bubble Sort  In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.  Bubble Sort  We find that 27 is smaller than 33 and these two values must be swapped.  Bubble Sort  The new array should look like this −  Bubble Sort  Next we compare 33 and 35. We find that both are in already sorted positions.  Bubble Sort  Then we move to the next two values, 35 and 10.  Bubble Sort  We know then that 10 is smaller 35. Hence they are not sorted.  Bubble Sort  We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −  Bubble Sort  To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −  Bubble Sort  Notice that after each iteration, at least one value moves at the end.  Bubble Sort  And when there's no swap required, bubble sorts learns that an array is completely sorted.  Bubble Sort  Now we should look into some practical aspects of bubble sort. |
| **PSEUDOCODE:** | begin BubbleSort(int arr[],int n):  for i=0 to i< (n-1)  if arr[i] is greater than arr[i+1]  swap arr[i] and arr[i+1]  if n-1 is greater than 1 return BubbleSort(arr,n-1) |
| **EXPERIMENT 1** | |
| **CODE:** | Bubble Sort code:  import java.util.Arrays;  public class BubbleSort {      int counter = 0;      int swapNum = 0;      static int maxPos = 0;      String swaps = "Swaps: |";  *// swap function*      public static void swap(int[] arr, int i, int j) {          int temp = arr[i];          arr[i] = arr[j];          arr[j] = temp;      }  *// bubble sort function*      public void bubbleSortIteration(int arr[]) {          int n = arr.length;          System.out.printf("Iteration\tSwap\t\tArray\n");          for (int i = 0; i < n - 1; i++) {              for (int j = 0; j < n - i - 1; j++) {                  ++counter;                  if (arr[j] > arr[j + 1]) {  *// swap arr[j+1] and arr[j]*                      System.out.printf("%d\t\t%d<->%d\t\t%s\n", counter, arr[j], arr[j + 1], printArr(arr));  *// swappging the elements*                      int temp = arr[j];                      arr[j] = arr[j + 1];                      arr[j + 1] = temp;                  } else {                      System.out.printf("%d\t\t----\t\t%s\n", counter, printArr(arr));                  }              }          }      }      public void bubbleSortRec(int[] arr, int n) {          for (int i = 0; i < n - 1; i++) {              if (arr[i] > arr[i + 1]) {                  ++swapNum;  *// swap arr[i+1] and arr[i]*                  swaps += arr[i] + "<->" + arr[i + 1] + " | ";  *// swappging the elements*                  swap(arr, i, i + 1);              }          }          if (n - 1 > 1) {              System.out.print("\n\n" + swaps);              System.out.print(                      "\nUnsorted + Sorted => " + printArr(arr, 0, n - 1) + " " + printArr(arr, n - 1, arr.length)                              + "\n");              swaps = "Swaps: |";              bubbleSortRec(arr, n - 1);          }      }      public int getSwapCount() {          return swapNum;      }      public int[] bubbleSort(int i, int[] arr) {          if (arr[i] > arr[i + 1] && (i + 1) < arr.length - 1) {              swap(arr, i, i + 1);              maxPos++;          }          if (i < arr.length - 1) {              System.out.print("\n-----------------------------------------------------");              System.out.println("\nMax Places " + arr[i] + " is shifted " + maxPos + " times");              System.out.println("\nArray: " + printArr(arr));              maxPos = 0;              bubbleSort(i + 1, arr);          }          return arr;      }      public String printArr(int arr[]) {  *// printing the array*          if (arr.length > 10)              return " ... ";          else              return Arrays.toString(arr);      }      public String printArr(int[] arr, int start, int end) {  *// printing the array*          if (arr.length > 10)              return " ... ";          else              return Arrays.toString(Arrays.copyOfRange(arr, start, end));      }  *// They need the array size case*  }  Driver Code:  import java.util.ArrayList;  import java.util.Arrays;  import java.util.Collections;  import java.util.Scanner;  public class Driver {      public static void main(String[] args) throws Exception {  *// Arraylist of Integers*          ArrayList<Integer> list = new ArrayList<Integer>();  *// array for sorting*          int[] array;  *// User input*          Scanner input = new Scanner(System.in);          System.out.print("\n1.Random Array\n2.Roll.no Input    : ");          int choice = input.nextInt();          if (choice == 1) {              System.out.print("\nEnter the size of the array : ");              int size = input.nextInt();              for (int i = 0; i < size; i++) {                  list.add((int) (i + (Math.random() \* 100)));              }              Collections.sort(list);          } else {              System.out.print("\nEnter the roll no: ");              int roll = input.nextInt();              for (int i = 0; i < 10; i++) {                  list.add(roll + (roll + 1) \* i);              }          }          System.out.print("\n1.Best Case\n2.Worst Case\n3.Average Case\n4.Manual Choice\nEnter your choice: ");          int newChoice = input.nextInt();          int listSize = list.size();          array = new int[listSize];          switch (newChoice) {              case 1:                  for (int i = 0; i < listSize; i++) {                      array[i] = list.get(i);                  }                  break;              case 2:                  for (int i = listSize - 1; i >= 0; i--) {                      array[(listSize - 1) - i] = list.get(i);                  }                  break;              case 3:                  Collections.shuffle(list);                  for (int i = 0; i < listSize; i++) {                      array[i] = list.get(i);                  }                  break;              case 4:                  if (choice == 1) {                      System.out.print("\nAre u sure u want to enter the array manually?(y/n): ");                      String choice1 = input.next().toLowerCase();                      if (choice1.equals("y")) {                          System.out.print("Enter the size of the array: ");                          int size = input.nextInt();                          list.clear();                          System.out.print("Enter the elements of the array(with space): ");                          for (int i = 0; i < size; i++) {                              list.add(input.nextInt());                          }                      }                  }                  break;              default:                  System.out.println("Invalid choice");                  break;          }          int arrLen = array.length;  *// Array segregation*          switch (arrLen) {              case 0:                  input.close();                  throw new Exception("Array is empty");              case 1:                  input.close();                  throw new Exception("Array has only one element");              default:                  System.out.print("\nArray: " + Arrays.toString(array));                  break;          }          System.out                  .print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");          System.out.print("\nBubble Sort\n");  *// Bubble Sort*          System.out.print("\n1.Iteration Buuble Sort\n2.Recursive Buuble Sort\n3.Bubble Sort (without for loops)   : ");          int choice2 = input.nextInt();  *// Create bubble sort object*          BubbleSort bs = new BubbleSort();  *// Call bubble sort function*          System.out.print("\nBefore sorting: " + Arrays.toString(array));          System.out.print("\n----------------------  -----------------------\n");          if (choice2 == 1) {              bs.bubbleSortIteration(array);          } else {              bs.bubbleSortRec(array, listSize);              System.out.print("\nTotal Swaps: " + bs.getSwapCount());          }          System.out.print("\n----------------------  -----------------------\n");          System.out.print("\nFinal Array: " + Arrays.toString(array) + " \n");  *// input closing method*          input.close();      }  } |
| **OUTPUT:** | Best Case:      Average Case:          Worst Case:      **Time Complexity of Bubble Sort:**  **Worst Case Time Complexity**  Θ(N^2) is the Worst Case Time Complexity of Bubble Sort.  This is the case when the array is reversely sort  The number of swaps of two elements is equal to the number of comparisons in this case as every element is out of place.  T(N)=C(N)=S(N)=N∗(N−1)2T(N)=C(N)=S(N)=N∗(N−1)2  Therefore, in the worst case:   * Number of Comparisons: O(N^2) time * Number of swaps: O(N^2) time   **Best Case Time Complexity**  Θ(N) is the Best Case Time Complexity of Bubble Sort.  This case occurs when the given array is already sorted.  T(N)=C(N)=NT(N)=C(N)=N S(N)=0S(N)=0  Therefore, in the best case:   * Number of Comparisons: N = O(N) time * Number of swaps: 0 = O(1) time   **Average Case Time Complexity**  **Θ(N^2)** is the Average Case Time Complexity of Bubble Sort.  The number of comparisons is constant in Bubble Sort so in average case, there is O(N^2) comparisons. This is because irrespective of the arrangement of elements, the number of comparisons C(N) is same.  For the number of swaps, consider the following points:   * If an element is in index I1 but it should be in index I2, then it will take a minimum of I2-I1 swaps to bring the element to the correct position. * An element E will be at a distance of I3 from its position in sorted array * The sum of maximum difference in position across all elements will be:   (N-1) + (N-3) + (N-5) ... + 0 + ... + (N-3) + (N-1) = N x N - 2 x (1 + 3 + 5 + ... + N/2) = N^2 - 2 x N^2 / 4 = N^2 - N^2 / 2 = N^2 / 2  Therefore, in average, the number of swaps = O(N^2).  Therefore, in the average case time complexity of Bubble sort:   * Number of Comparisons: O(N^2) time * Number of swaps: O(N^2) time |
| **TIME COMPLEXITY:** |  |
| **RESULT:** Things learnt during procedural programming during solving of the problem   * Learnt how to implement bubble sort using iteration and recursion * Learnt how to use exception in order to show that the array is of invalid length * Learnt different time complexity cases of bubble sort and how it can be used. * Learnt bubble sort required more time and space as it go through the array N^2 times | |

Extra Outputs:

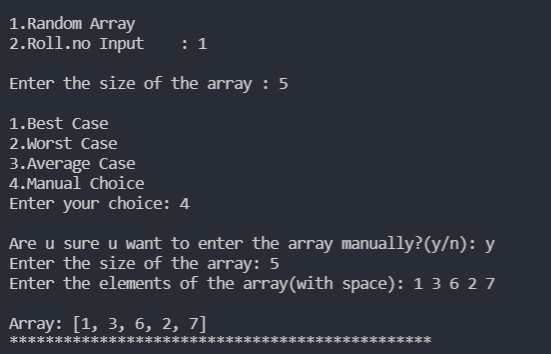


(If the array is too big it shows the ‘….’)

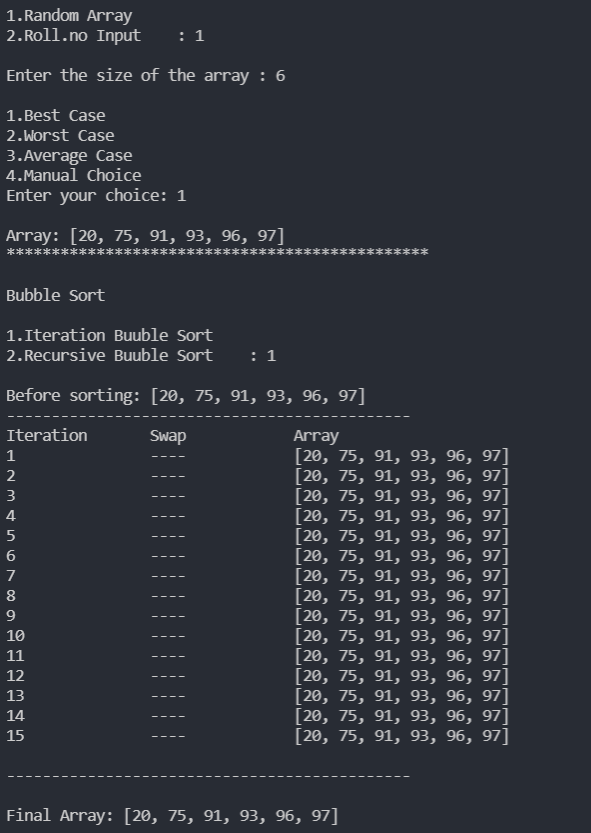
(When the array size is 0/1 it shows error)



(User gets a chance to renter array)



(Iterative Bubble sort)



(Roll no input)

