COMP6008 Data Structures and Algorithms

Assessment 1

Submitted By

Ramon Pacheco

Student ID

24575333

Module One Programming Tasks

Task 1: Lexicographical order

1. Solution code (compulsory)

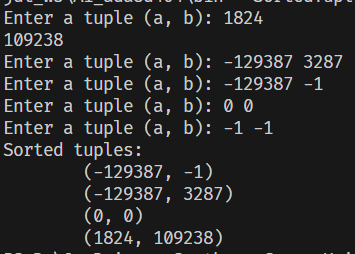
|  |
| --- |
| import java.util.\*;  class SortedTupleList {      public static class Tuple {          int x, y;          public Tuple(int x, int y) {              this.x = x;              this.y = y;          }          boolean equal(Tuple other) {              if (this.x == other.x && this.y == other.y)                  return true;              else                  return false;          }          public boolean greaterThan(Tuple other) {              if (this.x > other.x)                  return true;              if (this.x == other.x && this.y > other.y)                  return true;              return false;          }      }      public static void main(String[] args) {          Scanner scanner = new Scanner(System.in);          List<Tuple> list = new ArrayList<>();          while (true) {              System.out.print("Enter a tuple (a, b): ");              int a = scanner.nextInt();              int b = scanner.nextInt();              if (a == -1 && b == -1)                  break;              Tuple newTuple = new Tuple(a, b);              insert(list, newTuple);          }          System.out.println("Sorted tuples:");          for (Tuple t : list) {              System.out.println("\t(" + t.x + ", " + t.y + ")");          }          scanner.close();      }      public static void insert(List<Tuple> list, Tuple newTuple) {          int position = list.size(); // Start from the end of the list          while (position > 0 && !newTuple.greaterThan(list.get(position - 1))) {              position--;          }          list.add(position, newTuple);      }  } |

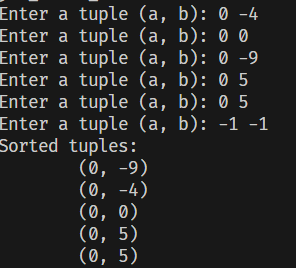
2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
| Enter a tuple (a, b): 1 5  Enter a tuple (a, b): 1 4  Enter a tuple (a, b): 0 0  Enter a tuple (a, b): -1 3  Enter a tuple (a, b): -1 -1 | Sorted tuples:  (-1, 3)  (0, 0)  (1, 4)  (1, 5) |
| Enter a tuple (a, b): 1824  109238  Enter a tuple (a, b): -129387 3287  Enter a tuple (a, b): -129387 -1  Enter a tuple (a, b): 0 0  Enter a tuple (a, b): -1 -1 | Sorted tuples:  (-129387, -1)  (-129387, 3287)  (0, 0)  (1824, 109238) |
| Enter a tuple (a, b): 0 -4  Enter a tuple (a, b): 0 0  Enter a tuple (a, b): 0 -9  Enter a tuple (a, b): 0 5  Enter a tuple (a, b): 0 5  Enter a tuple (a, b): -1 -1 | Sorted tuples:  (0, -9)  (0, -4)  (0, 0)  (0, 5)  (0, 5) |

3. Screenshots of tests (compulsory)

A screen shot of a computer

Description automatically generated 



Task 2: Checking unbalanced opening and closing tags in HTML

1. Solution code (compulsory)

|  |
| --- |
| import java.util.\*;  import java.io.\*;  public class StackTesting {      String path = "";      public StackTesting(String path) {          this.path = path;      }      public void tagCheck() {          Stack<String> symStack = new Stack<>();          try {              String line = "";              BufferedReader br = new BufferedReader(new FileReader(path));              while ((line = br.readLine()) != null) {                  String[] tokens = line.split(" ");                  for (String e : tokens) {                      if (e.charAt(0) == '<' && e.charAt(1) != '/')                          symStack.push(e);                        else if (e.charAt(0) == '<' && e.charAt(1) == '/') {                          if (symStack.empty())                              System.err.println("Tags being closed before they are being opened.");                          else                              symStack.pop();                      }                  }              }              br.close();          } catch (FileNotFoundException e) {              System.err.println("File could not be found.\nPlease make sure the path to the file is correct.");              e.printStackTrace();          } catch (IOException e) {              e.printStackTrace();          }          System.err.println(symStack.isEmpty());      }      public static void main(String[] args) {          String path = "D:\\OneDrive - Southern Cross University\\SCU\\Term 4\\COMP6008 Data Structure and Algorithms\\A1\\Module 1\\html\_file.txt";          StackTesting std = new StackTesting(path);          std.tagCheck();      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
| <html> <head> <title> Example </title> </head> <body> <h1> Hello, world </h1> </body> </html> | True |
| <html> <head> <title> Example <title> </head> <body> <h1> Hello, world </h1> </body> </html> | False |
| <html> <head> <title> Example <title> </head> <body> <h1> Hello, world </h1> </body> <html> | False |

3. Screenshots of tests (compulsory)

4. Analysis of experiment results(if required)

Algorithm can only identify which tags or in which position they are open, it only takes individual closing and opening characters to identify whether or not there is a match for open tags. The program, however, includes a conditional statement to check whether there are tags being close without being opened first ie. removing from empty stack

Task 3: Benchmark two queue implementations

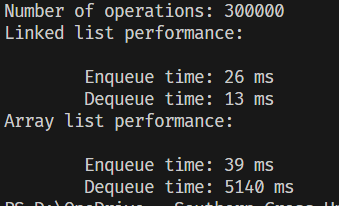
1. Solution code (compulsory)

|  |
| --- |
| import java.util.\*;  public class QueueTest {      private static final int numOperations = 300\_000;      public static void main(String[] args) {          List<Integer> qLinked = new LinkedList<>();          List<Integer> qArray = new ArrayList<>();          System.err.println("Number of operations: "+numOperations);          System.out.println("Linked list performance: \n");          benchmark(qLinked);          System.out.println("Array list performance: \n");          benchmark(qArray);      }      private static void benchmark(List<Integer> queue) {          // Enqueue benchmark          long startTime = System.currentTimeMillis();          for (int i = 0; i < numOperations; i++) {              queue.add(i);          }          long endTime = System.currentTimeMillis();          long enqueueTime = endTime - startTime;          // Dequeue benchmark          startTime = System.currentTimeMillis();          for (int i = 0; i < numOperations; i++) {              queue.remove(0);          }          endTime = System.currentTimeMillis();          long dequeueTime = endTime - startTime;          System.out.printf("\tEnqueue time: %d ms\n", enqueueTime);          System.out.printf("\tDequeue time: %d ms\n", dequeueTime);      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
| Case 1 | Number of operations: 300000  Linked list performance:  Enqueue time: 26 ms  Dequeue time: 13 ms  Array list performance:  Enqueue time: 39 ms  Dequeue time: 5140 ms |
| Case 2 | Number of operations: 300000  Linked list performance:  Enqueue time: 23 ms  Dequeue time: 13 ms  Array list performance:  Enqueue time: 52 ms  Dequeue time: 5112 ms |
| Case 3 | Number of operations: 300000  Linked list performance:  Enqueue time: 27 ms  Dequeue time: 12 ms  Array list performance:  Enqueue time: 50 ms  Dequeue time: 5183 ms |

3. Screenshots of tests (compulsory)

 A screen shot of a computer

Description automatically generated

4. Analysis of experiment results(if required)

The number of queue/dequeue operations performed by the system (300,000) was defined based on the time the array list took to dequeue (5145ms in average), which was by far superior to the performance registered by the other executions. That equals more than 10 times the required time to perform the enqueue with the same data structure.

The linked list took around 25ms to execute the enqueue and half that time to dequeue, presenting itself as a much better alternative to the queue processes.

Module Two Programming Tasks

Task 4: Longest consecutive sequence

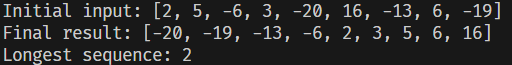
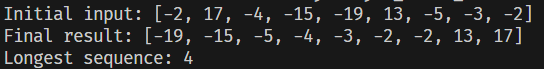
1. Solution code (compulsory)

|  |
| --- |
| import java.util.Arrays;  public class ConsecutiveSequence {      static int longestSeq = 0;      public static void main(String[] args) {          int[] X = ArrayUtils.generateRandomArray(9, 8);          System.out.println("Initial input: " + Arrays.toString(X));          QuickSort(X, 0, X.length-1);          countSeq(X);          System.out.println("Final result: " + Arrays.toString(X));          System.out.println("Longest sequence: " + longestSeq);      }      static void QuickSort(int[] list, int startIndex, int endIndex) {          if (startIndex < endIndex) {              int pivotIndex = partition(list, startIndex, endIndex);              QuickSort(list, startIndex, pivotIndex - 1);              QuickSort(list, pivotIndex + 1, endIndex);          }      }        static int partition(int[] list, int startIndex, int endIndex) {          int pivotVal = list[startIndex];          int leftMark = startIndex + 1;          int rightMark = endIndex;          while (leftMark <= rightMark) {              while (leftMark <= rightMark && list[leftMark] <= pivotVal)                  leftMark++;              while (leftMark <= rightMark && list[rightMark] >= pivotVal)                  rightMark--;              if (leftMark <= rightMark)                  swap(list, leftMark, rightMark);          }          swap(list, startIndex, rightMark);          return rightMark;      }      static void swap(int[] list, int i, int j) {          int temp = list[i];          list[i] = list[j];          list[j] = temp;      }      static void countSeq(int[] list) {          int currentSeq = 1;          for (int i = 0; i < list.length - 1; i++) {              if (list[i] + 1 == list[i + 1]) {                  currentSeq++;              } else {                  currentSeq = 1;              }              longestSeq = currentSeq > longestSeq ? currentSeq : longestSeq;          }      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
| Initial input: [2, 5, -6, 3, -20, 16, -13, 6, -19] | Longest sequence: 2 |
| Initial input: [-2, 17, -4, -15, -19, 13, -5, -3, -2] | Longest sequence: 4 |
| Initial input: [-5, 16, 8, 21, -13, -15, 14, -21, -15] | Longest sequence: 1 |

3. Screenshots of tests (compulsory)

  A number on a black background

Description automatically generated

4. Reflection on the use of online resources including GenAI (if any)

* What resource? (e.g., ChatGPT, Stack Overflow)
  + ChatGPT
* How did you use it and how did it help you?
  + It helped me define and design a utility class called ArrayUtils (included in the Appendix) to generate random arrays used by the algorithms. Also, it showed me how to use a method from another file.

Task 5: Alternative pivot value for quicksort

1. Solution code (compulsory)

|  |
| --- |
| public class PivotPosition {      public static void main(String[] args) {          int[] testList = ArrayUtils.generateRandomArray(200000, 8);          int[] testList2 = new int[200000];          for (int i = 0; i < testList.length; i++) {              testList2[i] = testList[i];          }          long startTime = System.currentTimeMillis();          QuickSort(testList, 0, testList.length - 1);          long endTime = System.currentTimeMillis();          System.out.printf("Quick sort execution time: %d ms.", endTime - startTime);          startTime = System.currentTimeMillis();          QuickSortSwapping(testList2, 0, testList.length - 1);          endTime = System.currentTimeMillis();          System.out.printf("\nQuick sort swapping pivot index execution time: %d ms.", endTime - startTime);      }      static void QuickSort(int[] list, int startIndex, int endIndex) {          if (startIndex < endIndex) {              int pivotIndex = partition(list, startIndex, endIndex);              QuickSort(list, startIndex, pivotIndex - 1);              QuickSort(list, pivotIndex + 1, endIndex);          }      }      static void QuickSortSwapping(int[] list, int startIndex, int endIndex) {          if (startIndex < endIndex) {              swapPivot(list, startIndex, endIndex);              int pivotIndex = partition(list, startIndex, endIndex);              QuickSort(list, startIndex, pivotIndex - 1);              QuickSort(list, pivotIndex + 1, endIndex);          }      }      static void swapPivot(int[] list, int startIndex, int endIndex) {          int midIndex = (startIndex + endIndex) / 2 + 1;          if (list[startIndex] <= list[midIndex] && list[midIndex] <= list[endIndex]) {              swap(list, startIndex, midIndex);          } else if (list[startIndex] <= list[endIndex] && list[midIndex] <= list[endIndex]) {              swap(list, startIndex, endIndex);          }      }      static int partition(int[] list, int startIndex, int endIndex) {          int pivotVal = list[startIndex];          int leftMark = startIndex + 1;          int rightMark = endIndex;          while (leftMark <= rightMark) {              while (leftMark <= rightMark && list[leftMark] <= pivotVal)                  leftMark++;              while (leftMark <= rightMark && list[rightMark] >= pivotVal)                  rightMark--;              if (leftMark <= rightMark)                  swap(list, leftMark, rightMark);          }          swap(list, startIndex, rightMark);          return rightMark;      }      static void swap(int[] list, int i, int j) {          int temp = list[i];          list[i] = list[j];          list[j] = temp;      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
| Case 1 |  |
| Case 2 |  |
| Case 3 |  |
| Case 4 |  |
| Case 5 |  |
| Case 6 |  |
| Case 7 |  |

3. Analysis of experiment results

A randomly generated array of size 20,000 was declared for each iteration of the algorithm. The declared array was then duplicated so that both versions of the quick sort could sort the same array, enhancing the precision for the comparison.

For each case, there was a timer that started just before each method was executed and finished just after the method was executed. The difference between start and finish for each method was analysed. Seven tests were carried out to increase the overall precision of the test, so that the average execution time for each represented more accurately the perfomances.

The average execution time for the quick sort algorithm that always assumes the pivot value as the first element of the array averaged 437.3ms, whereas the quick sort that took the median element averaged 416.4ms. That represents a 5% relative difference between the two.

Task 6: Intersection of sets

1. Solution code (compulsory)

|  |
| --- |
| import java.util.\*;  public class ArrayIntersection {      public static void findIntersection(int[] X, int[] Y) {          long startTime = System.currentTimeMillis();          ArrayList<Integer> intersectionList = new ArrayList<>();          Arrays.sort(X);          Arrays.sort(Y);          int Xpointer = 0;          int Ypointer = 0;          while (Xpointer < X.length && Ypointer < Y.length) {              if (X[Xpointer] == Y[Ypointer] && !intersectionList.contains(X[Xpointer])) {                  intersectionList.add(X[Xpointer]);                  Xpointer++;              } else if (X[Xpointer] < Y[Ypointer]) {                  Xpointer++;              } else {                  Ypointer++;              }          }          // System.out.print("Intersection of X and Y: ");          // System.out.println(intersectionList);          long endTime = System.currentTimeMillis();          long deltaHS = endTime - startTime;          System.out.println("Pointer approach execution time: " + deltaHS + " ms.");      }      public static void bruteForce(int[] X, int[] Y) {          long startTime = System.currentTimeMillis();          ArrayList<Integer> intersectionList = new ArrayList<Integer>();          for (int i : X) {              for (int j : Y) {                  if (i == j && !intersectionList.contains(i)) {                      intersectionList.add(i);                  }              }          }          long endTime = System.currentTimeMillis();          long deltaBF = endTime - startTime;          System.out.println("Brute force execution time: " + deltaBF + " ms.");      }      public static void main(String[] args) {          int[] X = new int[50000];          int[] Y = new int[X.length];          X = ArrayUtils.generateRandomArray(X.length, 7);          Y = ArrayUtils.generateRandomArray(Y.length, 7);          findIntersection(X, Y);          bruteForce(X, Y);      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
| Case 1 |  |
| Case 2 |  |
| Case 3 |  |

4. Analysis of experiment results(if required)

There is a noticeable difference between the pointer and the brute force approaches.

The pointer approach consists of setting a pointer for each pre-sorted array and moving it based on the elements being compared. If the compared elements are the same, the element is then added to a third array list, which stores the intersection elements.

The brute force array, on the other hand, has a loop inside another loop to test every possible combination between the array elements and if it finds any common elements, adds them to the intersection array.

Similarly to the previous example, a timer started and finished immediately before and after the methods were executed, and multiple tests were executed so that the average depicts the methods performances with improved accuracy.

The pointer approach averaged 106ms, being over 65 times faster than the brute force approach.

Another analysis was made by developing another code (listed in the Appendix) and replacing the two arrays by hash sets and using the contains() method to efficiently verify the presence of common elements, resulting in a 30ms average execution time. Despite efficiency, the solution does not use any sorting algorithms required by the task description.

Module Three Programming Tasks

Task 7: First occurrence of binary search

1. Solution code (compulsory)

|  |
| --- |
| public class FirstOccurence {      public static void main(String[] args) {          int[] testList = { 1, 2, 2, 2, 2, 2, 32, 47 };          int target = 2;          searchResult(testList, target);      }      static void searchResult(int[] list, int target) {          int result = binarySearch(list, target, 0, list.length);          if (result < 0)              System.out.println("Index for " + target + " was not found");          else {              while (list[result] == list[result - 1] && result >= 0) {                  result--;              }              System.out.println("Element " + target + " found at index " + result);          }      }      static int binarySearch(int[] list, int target, int startIndex, int endIndex) {          if (startIndex > endIndex)              return -1;          else {              int midIndex = (startIndex + endIndex) / 2;              if (list[midIndex] == target)                  return midIndex;              else if (list[midIndex] > target)                  return binarySearch(list, target, startIndex, midIndex - 1);              else                  return binarySearch(list, target, midIndex + 1, endIndex);          }      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
|  |  |
|  |  |
|  |  |

Task 8: Stable sort insertion

1. Solution code (compulsory)

|  |
| --- |
| import java.util.\*;  public class InsertInOrder {      public static void main(String[] args) {          ArrayList<Integer> testList = new ArrayList<>();          int[] list = { 54, 17, 93, 17, 77, 31, 44, 55, 20 };          Arrays.sort(list);          for (int e : list) {              testList.add(e);          }          insert(testList, -100); // negative value          System.out.println(testList);      }      static void insert(ArrayList<Integer> testList, int target) {          if (target < testList.get(0)) {              testList.add(0, target);              return;          } else if (target > testList.get(testList.size() - 1)) {              testList.add(testList.size(), target);              return;          }          int result = binarySearch(testList, target, 0, testList.size() - 1);          if (result > 0) {              while (result > testList.size() && testList.get(result) == testList.get(result + 1)) {                  result++;              }              testList.add(result, target);          }      }      static int binarySearch(ArrayList<Integer> testList, int target, int startIndex, int endIndex) {          if (startIndex > endIndex) {              testList.add(startIndex, target);              return -1;          } else {              int midIndex = (startIndex + endIndex) / 2;              if (testList.get(midIndex) == target)                  return midIndex;              else if (testList.get(midIndex) > target)                  return binarySearch(testList, target, startIndex, midIndex - 1);              else                  return binarySearch(testList, target, midIndex + 1, endIndex);          }      }  } |

2. Testing inputs and outputs (compulsory)

|  |  |
| --- | --- |
| Input | Output |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Appendix

A utility class was created to generate arrays of integers with varying degrees of magnitude.

Such arrays can be created by using the method ArraysUtils.generateRandomArray(size, order).

import java.util.\*;

import java.lang.Math;

public class ArrayUtils {

    // Method to generate an array of random integers

    public static int[] generateRandomArray(int size, double order) {

        //order will reduce the order/magnitude of the integers generated. eg order 0 will generate integers to the 10^8 power, so to reduce integers to the range 1 - 20, the order should be set to 8

        int[] randomNumbers = new int[size]; // Initialize an array with the specified size

        Random random = new Random();

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

            randomNumbers[i] = (int) (random.nextInt()/Math.pow(10,order)); // Generate a random integer and assign it to the array

        }

        return randomNumbers; // Return the generated array

    }

}

import java.util.\*;

public class ArrayIntersection {

    public static void findIntersection(int[] X, int[] Y) {

        long startTime = System.currentTimeMillis();

        HashSet<Integer> X\_set = new HashSet<>();

        for (int num : X) {

            if (!X\_set.contains(num)) // Remove repetitive items

                X\_set.add(num);

        }

        // Find intersection elements

        HashSet<Integer> intersectionSet = new HashSet<>();

        for (int num : Y) {

            if (X\_set.contains(num) && !intersectionSet.contains(num))

                intersectionSet.add(num);

        }

        System.out.print("Intersection of X and Y: ");

        System.out.println(intersectionSet);

        long endTime = System.currentTimeMillis();

        long deltaHS = endTime - startTime;

        System.out.println("Hash set execution time: " + deltaHS + " ms.");

    }

    public static void bruteForce(int[] X, int[] Y) {

        long startTime = System.currentTimeMillis();

        ArrayList<Integer> intersectionList = new ArrayList<Integer>();

        for (int i : X) {

            for (int j : Y) {

                if (i == j && !intersectionList.contains(i)) {

                    intersectionList.add(i);

                }

            }

        }

        System.out.println(intersectionList.size());

        long endTime = System.currentTimeMillis();

        long deltaBF = endTime - startTime;

        System.out.println("Brute force execution time: " + deltaBF + " ms.");

    }

    public static void main(String[] args) {

        int[] X = new int[50000];

        int[] Y = new int[X.length];

        X = ArrayUtils.generateRandomArray(X.length, 7);

        Y = ArrayUtils.generateRandomArray(Y.length, 7);

        findIntersection(X, Y);

        bruteForce(X, Y);

    }

}