Southern Cross University

COMP6008 Data Structures and Algorithms

Assessment 2

Submitted By

Ramon Henrique de Paiva Pacheco

Student ID

24575333

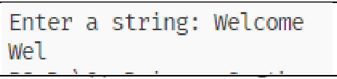
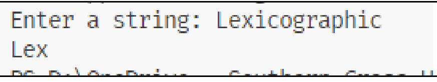
Module Four Tasks

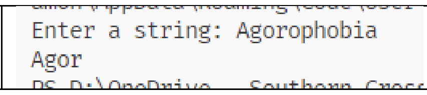
Task 1: Consecutive increasingly ordered substring

1. Solution code

```java
import java.util.*;

public class Substring {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String str = scanner.nextLine();
        scanner.close();

        // String str = new String("abcdefa");
        String currentStr = new String("");
        String longestStr = new String("");
        char prev = str.charAt(0);
        for (int i = 0; i < str.length() - 1; i++) {
            if (str.charAt(i) > prev) {
                currentStr += str.charAt(i);
            } else {
                currentStr = str.substring(i, i + 1);
            }
            prev = str.charAt(i);
            longestStr = currentStr.length() > longestStr.length() ? currentStr
: longestStr;
        }
        System.err.println(longestStr);
    }
}
```

2. Testing inputs and outputs

Input	Output
Welcome	 `Enter a string: Welcome` `Wel`
Lexicographic	 `Enter a string: Lexicographic` `Lex`

Agorophobia	 Enter a string: Agorophobia Agor PS D:\OneDrive - Southern Cross

Analysis

Since there is only one for loop on the code, which is dependent on the array size, the complexity is O(n). Within the main loop some string methods are used, such as charAt() and substring(), which have O(1) complexity.

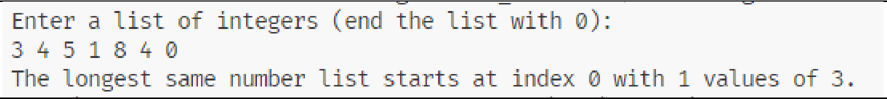
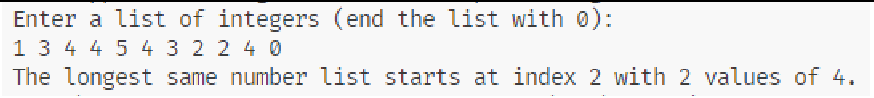
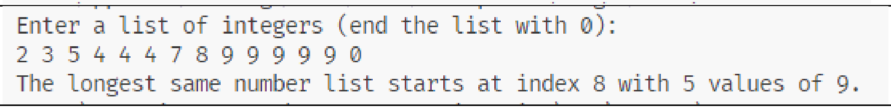
Task 2: Longest subsequence with the same number

1. Solution code

```
import java.util.*;

public class LongestNumberRep {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ArrayList<Integer> list = new ArrayList<>();

        System.out.println("Enter a list of integers (end the list with 0):");

        while (true) {
            int num = scanner.nextInt();
            if (num == 0) {
                scanner.close();
                break;
            }
            list.add(num);
        }

        int index = 0;
        int maxIndex = 0;
        int rep = 1;
        int maxRep = 1;
        int value = list.get(0);
        int maxValue = list.get(0);

        for (int i = 1; i < list.size(); i++) {
            if (list.get(i) == value) {
```

```java
                rep++;
            } else {
                // if number repetition had ended, check and possibly update
max variables
                if (rep > maxRep) {
                    maxRep = rep;
                    maxIndex = index;
                    maxValue = value;
                }
                // reset
                value = list.get(i);
                index = i;
                rep = 1;
            }
        }
        if (rep > maxRep) {
            maxRep = rep;
            maxIndex = index;
            maxValue = value;
        }

        System.out.println("The longest same number list starts at index " +
maxIndex + " with " + maxRep + " values of " + maxValue + ".");

    }
}
```

2. Testing inputs and outputs

Input	Output
3 4 5 1 8 4 0	 Enter a list of integers (end the list with 0): 3 4 5 1 8 4 0 The longest same number list starts at index 0 with 1 values of 3.
1 3 4 4 5 4 3 2 2 4 0	 Enter a list of integers (end the list with 0): 1 3 4 4 5 4 3 2 2 4 0 The longest same number list starts at index 2 with 2 values of 4.
2 3 5 4 4 4 7 8 9 9 9 9 9 0	 Enter a list of integers (end the list with 0): 2 3 5 4 4 4 7 8 9 9 9 9 9 0 The longest same number list starts at index 8 with 5 values of 9.

Analysis: There are no nested loops or expensive operations inside the main loop, keeping the overall time complexity linear at O(n). The loop tracks the current number's repetition count and updates maximum values when needed, all in constant time operations.

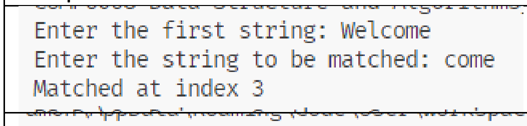
__Task 3: Substring matching__

1. Solution code

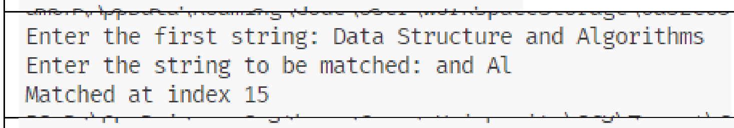
```
import java.util.Scanner;

public class MatchIndex {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first string: ");
        String str = scanner.nextLine();
        System.out.print("Enter the string to be matched: ");
        String sub = scanner.nextLine();
        scanner.close();
        int pointerSub = 0;

        if (sub.length() > str.length()) {
            System.err.println("The second string does not match the first due
to its larger size");
            return;
        }

        for (int pointerStr = 0; pointerStr < str.length(); pointerStr++) {
            if (str.charAt(pointerStr) == sub.charAt(pointerSub)) {
                if (pointerSub == sub.length() - 1) {
                    // if subpointer gets to the end of the substring, the
match has been
                    // verified
                    System.out.println("Matched at index " + (pointerStr -
pointerSub));
                    return;
                }
                // if letter at pointer position are the same, increase pointer
to check next
                // character
                pointerSub++;
            } else if (str.charAt(pointerStr) != sub.charAt(pointerSub)) {
                // if letters are at pointer position are different, substring
pointer resets
                pointerSub = 0;
            }
        }
        System.out.println("Strings do not match.");
    }
}
```

2. Testing inputs and outputs

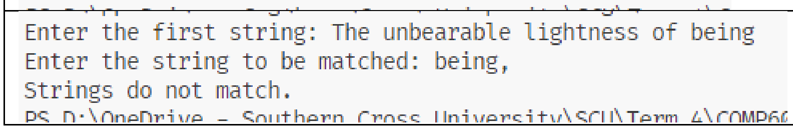
Output



```
Enter the first string: Welcome
Enter the string to be matched: come
Matched at index 3
```



```
Enter the first string: Data Structure and Algorithms
Enter the string to be matched: and Al
Matched at index 15
```



```
Enter the first string: The unbearable lightness of being
Enter the string to be matched: being,
Strings do not match.
PS D:\OneDrive - Southern Cross University\SCU\Term 4\COMP6(
```

Analysis

The program performs a single initial length comparison O(1), then uses a single main loop that iterates through each character of the main string O(n).

Module Five Tasks

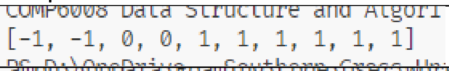
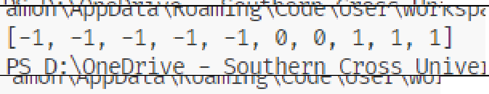
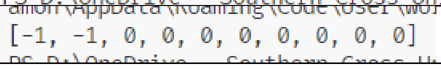
Task 4: O(n) Array sorting

1. Solution code

```
import java.util.*;

public class SortEfficiency {
    public static int[] generateRandomArray(int n) {
        Random rand = new Random();
        int[] arr = new int[n];
```

```java
        for (int i = 0; i < n; i++)
            arr[i] = rand.nextInt(3) - 1;
        return arr;
    }
    public static void main(String[] args) {
        int[] array = generateRandomArray(10);
        List<Integer> array2 = new ArrayList<>();
        int zeroIndex = 0;
        for (int i = 0; i < array.length; i++) {
            if (array[i] < 0) {
                array2.add(0, array[i]);
                zeroIndex++;
            } else if (array[i]==0){
                array2.add(zeroIndex,array[i]);
            }
            else array2.add(array[i]);
        }
        System.err.println(array2);
    }
}
```

2. Testing inputs and outputs

Output
COMP6008 Data Structure and Algori [-1, -1, 0, 0, 1, 1, 1, 1, 1, 1]
amon\AppData\Roaming\Code\User\workspa [-1, -1, -1, -1, -1, 0, 0, 1, 1, 1] PS D:\OneDrive - Southern Cross Univer
amon\AppData\Roaming\Code\User\wor [-1, -1, 0, 0, 0, 0, 0, 0, 0, 0]

4. Analyse why the time complexity of your sorting program is O(n), considering that efficient sorting algorithms are at best O(n log n), while inefficient algorithms can be O(n2).

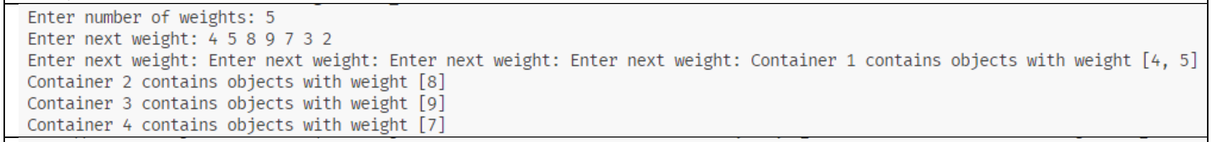
The O(n) time complexity of this code is due to fact that the loop iterate through the array only once, either for creating the random array or to reposition the elements. Each element is examined and added to the array2 list in a constant number of steps, regardless of the array's size.

__Task 5: Bin packing__

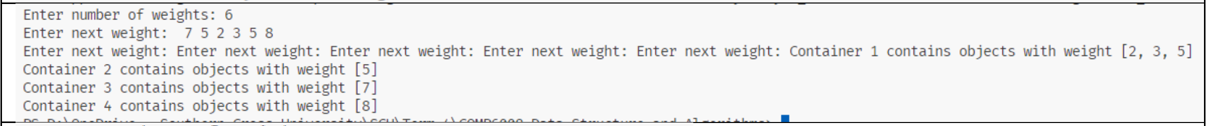
1. Solution code

```
import java.util.*;

public class Containers {
    public static void main(String[] args) {
        List<Integer> weights = new ArrayList<>();

        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter number of weights: ");
        int n = scanner.nextInt();

        for (int i = 0; i < n; i++) {
        int weight = 0;
        while (weight <= 0 || weight > 10) {
        System.out.print("Enter next weight: ");
        weight = scanner.nextInt();
        }
        weights.add(weight);
        }
        scanner.close();

        Collections.sort(weights);

        List<Integer> containerItems = new ArrayList<>();
        int containerNumber = 1;
        int containerWeight = 0;

        for (int e : weights) {
            if (containerWeight + e <= 10) {
                containerItems.add(e);
                containerWeight += e;
            } else {
                System.out.println("Container " + containerNumber++ + "
contains objects with weight " + containerItems);
                containerItems.clear();
                containerItems.add(e);
                containerWeight = e;
            }
        }
        System.err.println("Container " + containerNumber + " contains objects
with weight " + containerItems);
    }
}
```

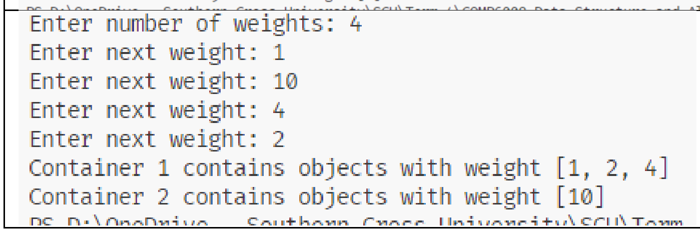
Output



```
Enter number of weights: 5
Enter next weight: 4 5 8 9 7 3 2
Enter next weight: Enter next weight: Enter next weight: Enter next weight: Container 1 contains objects with weight [4, 5]
Container 2 contains objects with weight [8]
Container 3 contains objects with weight [9]
Container 4 contains objects with weight [7]
```



```
Enter number of weights: 6
Enter next weight:  7 5 2 3 5 8
Enter next weight: Enter next weight: Enter next weight: Enter next weight: Enter next weight: Container 1 contains objects with weight [2, 3, 5]
Container 2 contains objects with weight [5]
Container 3 contains objects with weight [7]
Container 4 contains objects with weight [8]
PS D:\OneDrive - Southern Cross University\SCU\Term 4\COMP6008 Data Structures and Algorithms>
```



```
Enter number of weights: 4
Enter next weight: 1
Enter next weight: 10
Enter next weight: 4
Enter next weight: 2
Container 1 contains objects with weight [1, 2, 4]
Container 2 contains objects with weight [10]
PS D:\OneDrive - Southern Cross University\SCU\Term
```

5. Analysis of experiment results (if required)

Analyse the time and space complexity of your program. Discuss whether this program produces an optimal solution, that is, finding the minimum number of containers to pack the objects, and suggest a way to improve the solution

Even though the time complexity of the Collections.sort() has O(n log n) as the worst-case scenario, for random arrays is usually has O(n) complexity in average. The space complexity of the code follows the same complexity - two ArrayLists are created: weights and containerItems. Even though there are nested loops for the creation of the weights array list, the nested while loop is not dependent on the array size. Both arrays store n elements in the worst case. Other variables like containerNumber and containerWeight use constant space. (O(1)).

However, this approach is not the optimal solution, as it starts by sorting the array in an ascending order, and inserts the smallest element first, in a way that the larger elements are left to fill the gaps as the containers are created. An optimal solution would be to sort the array in a descending order, which would be able to start with larger elements and use smaller ones to fill the gaps in a more efficient manner.

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

- What resource?
  ChatGPT
- How did you use it?
  Establishing the time complexity of the Collections.sort().

Module Six Tasks

Task 7: Hybrid sort

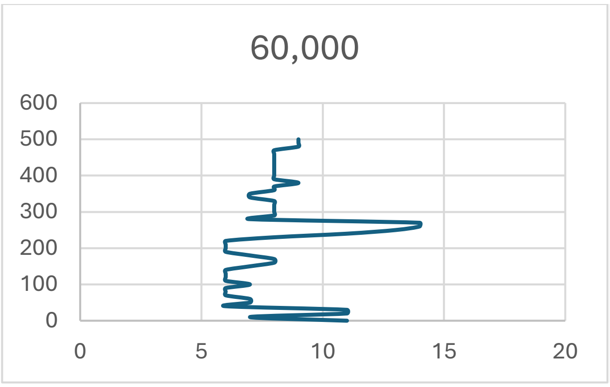
1. Solution code

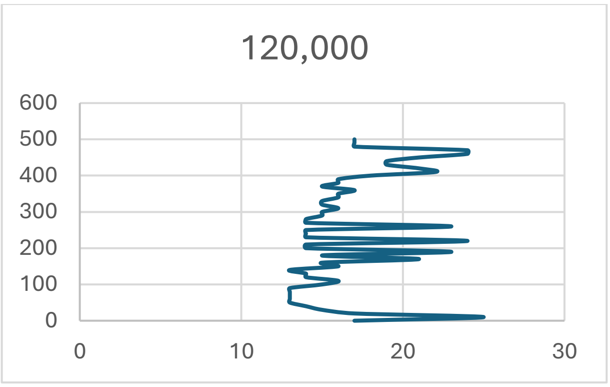
```
import java.io.*;

public class HybridSort {

    static void HybridSort(int[] list, int startIndex, int endIndex, int
threshold) {
        if (endIndex - startIndex < threshold) {
            InsertionSort(list, startIndex, endIndex);

        } else if (startIndex < endIndex) {
            int pivotIndex = partition(list, startIndex, endIndex);
            HybridSort(list, startIndex, pivotIndex - 1, threshold);
            HybridSort(list, pivotIndex + 1, endIndex, threshold);
        }
    }

    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;
    }
```

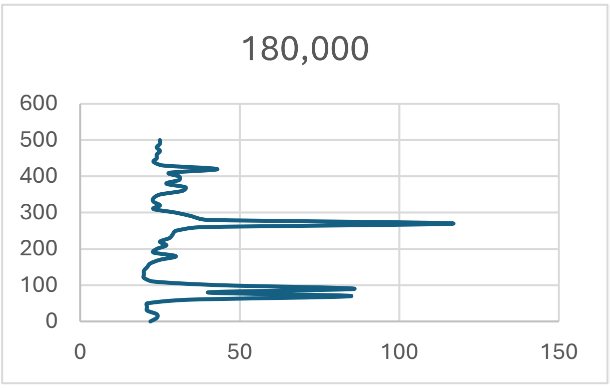
```java
    static void swap(int[] list, int i, int j) {
        int temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }

    static void InsertionSort(int[] list, int startIndex, int endIndex) {
        for (int i = startIndex; i < endIndex; i++) {
            int insertIndex = i;
            insert(list, insertIndex);
        }
    }

    static void insert(int[] list, int insertIndex) {
        int insertKey = list[insertIndex];
        while (insertIndex > 0 && list[insertIndex - 1] > insertKey) {
            list[insertIndex] = list[insertIndex - 1];
            insertIndex -= 1;
        }
        list[insertIndex] = insertKey;
    }

    public static void main(String[] args) {
        int[] sizes = { 60000, 120000, 180000, 240000, 300000, 360000, 720000,
1000000 };

        for (int e : sizes) {
            File file = new File(
                    "D:///OneDrive - Southern Cross University///SCU///Term
4///COMP6008 Data Structure and Algorithms///Module 6///test"
                            + e + ".csv");
            try (BufferedWriter writer = new BufferedWriter(new
FileWriter(file))) {
                writer.write("sep=,");
                writer.newLine();
                writer.write("threshold,avg_time");
                writer.newLine();
                int iterations = 10;

                for (int j = 0; j <= 500; j += 10) {
                    long startTime = System.currentTimeMillis();
                    for (int i = 0; i < iterations; i++) {
                        int[] testList = ArrayUtils.generateRandomArray(e, 6);
                        HybridSort(testList, 0, testList.length - 1, j);
                    }

                    long endTime = System.currentTimeMillis();

                    long delta = (endTime - startTime) / iterations;

                    writer.write(j + ", " + delta);
                    System.err.println("Average Quick Sort execution time: " +
delta + " ms for " + j + " threshold.");
                    writer.newLine();
                }
                System.out.println("CSV file written successfully to " + file);
            } catch (IOException err) {
                err.printStackTrace();
            }
        }
    }
}
```

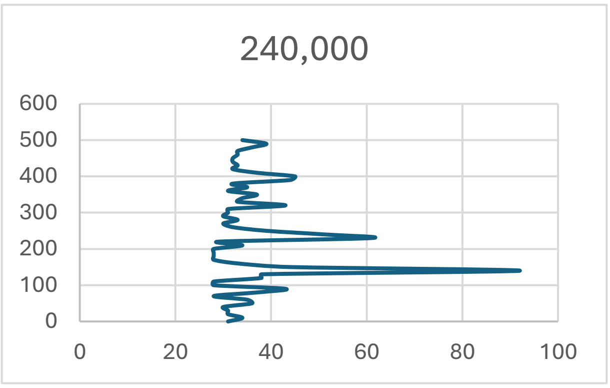
5. Analysis of experiment results (if required)

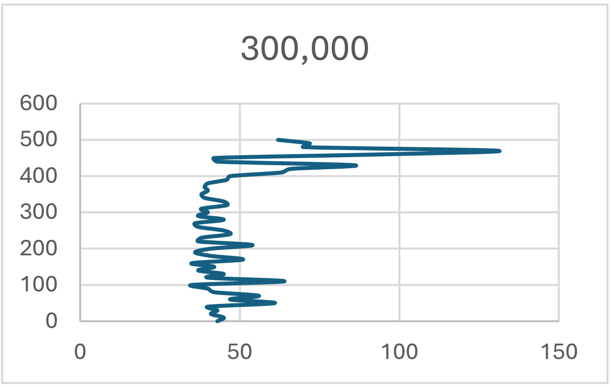
Size	Optimal Partition Limit
60,000	40
120,000	80
180,000	130
240,000	145
300,000	170
360,000	195
720,000	300
1,000,000	400

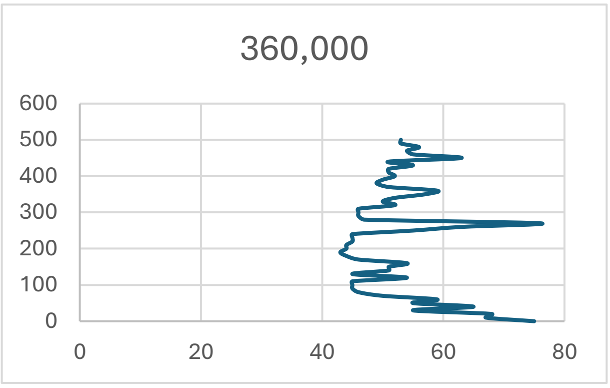


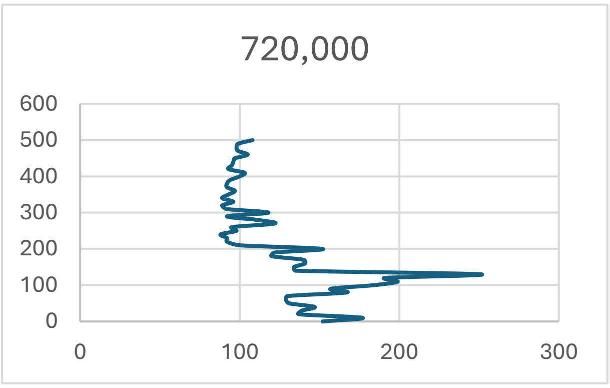


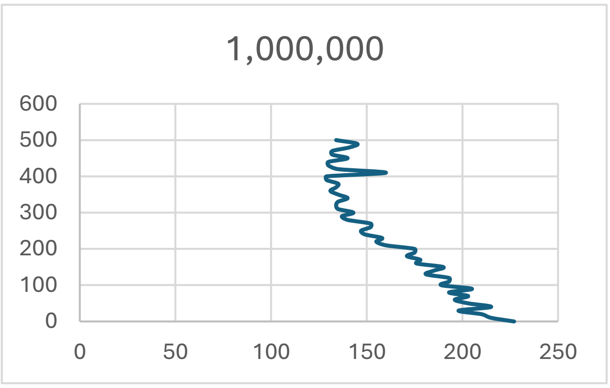












The program is capable of looping through different array sizes and thresholds multiple times and calculate the execution time. The results were then recorded into CSV files, which were analysed and plotted into graphs. The goal of the analysis is to determine the threshold value for each array size that yields the lowest execution time, which would allow the algorithm to slit the execution of Insertion and Quick Sort algorithms the best way possible.

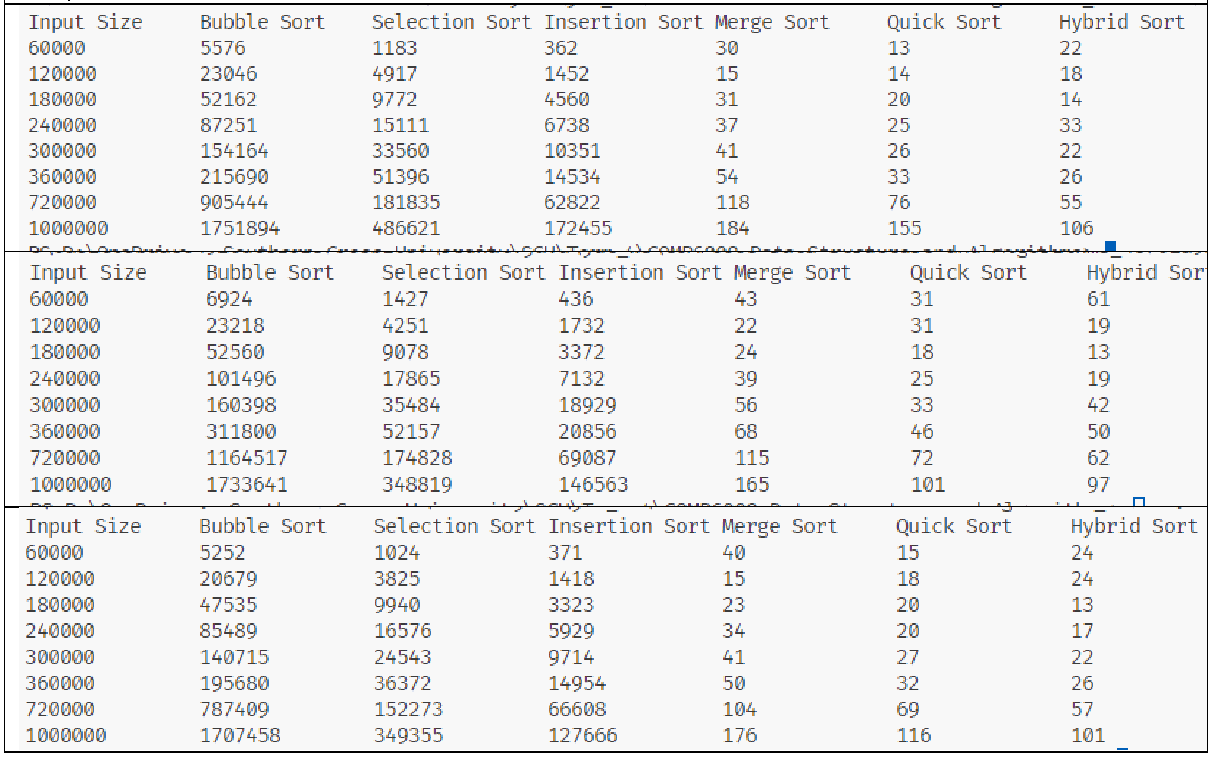
Task 8: Sorting a million 32-bit integers

1. Solution code

```java
import java.util.Arrays;
import java.util.Random;

public class Benchmark {
    static void BubbleSort(int[] list, int len) {
        int topIndex = len - 1;
        while (topIndex > 1) {
            bubbleUp(list, topIndex);
            topIndex--;
        }
    }

    static void bubbleUp(int[] list, int top) {
        for (int j = 0; j < top; j++)
            if (list[j] > list[j + 1])
                swap(list, j, j + 1);
    }

    static void swap(int[] list, int i, int j) {
        int temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }

    static void SelectionSort(int[] list, int len) {
        for (int startIndex = 0; startIndex < len - 1; startIndex++) {
            int minIndex = findMin(list, startIndex, len);
            swap(list, minIndex, startIndex);
        }
    }

    static int findMin(int[] list, int startIndex, int len) {
        int minIndex = startIndex;
        for (int i = startIndex + 1; i < len; i++)
            if (list[i] < list[minIndex])
                minIndex = i;
        return minIndex;
    }

    static void InsertionSort(int[] list, int len) {
        for (int i = 1; i < len; i++) {
            int insertIndex = i;
            insert(list, insertIndex);
        }
    }

    static void insert(int[] list, int insertIndex) {
        int insertKey = list[insertIndex];
        while (insertIndex > 0 && list[insertIndex - 1] > insertKey) {
            list[insertIndex] = list[insertIndex - 1];
            insertIndex -= 1;
        }
        list[insertIndex] = insertKey;
    }

    static void MergeSort(int[] list, int[] auxList, int startIndex, int
endIndex) {
        if (startIndex < endIndex) {
            int midIndex = (startIndex + endIndex) / 2 + 1;
            MergeSort(list, auxList, startIndex, midIndex - 1);
            MergeSort(list, auxList, midIndex, endIndex);
            merge(list, auxList, startIndex, midIndex, endIndex);
        }
    }

    static void merge(int[] list, int[] auxList, int startIndex, int midIndex,
int endIndex) {
        int leftIndex = startIndex;
        int leftEnd = midIndex - 1;
        int rightIndex = midIndex;
        int rightEnd = endIndex;
        int auxIndex = startIndex;

        while (leftIndex <= leftEnd && rightIndex <= rightEnd) {
                if (list[leftIndex] <= list[rightIndex]) {
                    auxList[auxIndex] = list[leftIndex];
                    leftIndex += 1;
                } else {
                    auxList[auxIndex] = list[rightIndex];
                    rightIndex += 1;
                }
                auxIndex += 1;
            }

            while (leftIndex <= leftEnd) {
                auxList[auxIndex] = list[leftIndex];
                leftIndex += 1;
                auxIndex += 1;
            }
            while (rightIndex <= rightEnd) {
                auxList[auxIndex] = list[rightIndex];
                rightIndex += 1;
                auxIndex += 1;
            }

            for (int i = startIndex; i <= endIndex; i++)
                list[i] = auxList[i];
        }

        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;
    }

    private static final Random random = new Random();

    public static void main(String[] args) {
        int[] sizes = { 60000, 120000, 180000, 240000, 300000, 360000, 720000,
        1000000 };
        int runs = 1;

        long bubbleTime = 0, selectionTime = 0, insertionTime = 0, mergeTime =
0, quickTime = 0, hybridTime = 0;

        System.out.printf("%-15s%-15s%-15s%-15s%-15s%-15s%-15s%n", "Input
Size", "Bubble Sort", "Selection Sort",
                "Insertion Sort", "Merge Sort", "Quick Sort", "Hybrid Sort");

        for (int size : sizes) {

            int[] array = new int[size];
            for (int i = 0; i < size; i++) {
                array[i] = random.nextInt();
            }
            // int[] array = ArrayUtils.generateRandomArray(size, 7);
            for (int i = 0; i < runs; i++) {
                int[] testList = Arrays.copyOfRange(array, 0, array.length);

                long startTime = System.currentTimeMillis();
            BubbleSort(testList, testList.length);
            long endTime = System.currentTimeMillis();
            bubbleTime += endTime - startTime;

            testList = Arrays.copyOfRange(array, 0, array.length);
            startTime = System.currentTimeMillis();
            SelectionSort(testList, testList.length);
            endTime = System.currentTimeMillis();
            selectionTime += endTime - startTime;

            testList = Arrays.copyOfRange(array, 0, array.length);
            startTime = System.currentTimeMillis();
            InsertionSort(testList, testList.length);
            endTime = System.currentTimeMillis();
            insertionTime += endTime - startTime;

            testList = Arrays.copyOfRange(array, 0, array.length);
            startTime = System.currentTimeMillis();
            int[] auxList = Arrays.copyOfRange(array, 0, array.length);
            MergeSort(testList, auxList, 0, testList.length - 1);
            endTime = System.currentTimeMillis();
            mergeTime += endTime - startTime;

            testList = Arrays.copyOfRange(array, 0, array.length);
            startTime = System.currentTimeMillis();
            QuickSort(testList, 0, testList.length - 1);
            endTime = System.currentTimeMillis();
            quickTime += endTime - startTime;

            testList = Arrays.copyOfRange(array, 0, array.length);
            startTime = System.currentTimeMillis();
            HybridSort.HybridSort(testList, 0, testList.length - 1, 250);
            endTime = System.currentTimeMillis();
            hybridTime += endTime - startTime;
        }

        System.out.printf("%-15d%-15d%-15d%-15d%-15d%-15d%-15d%n",
                size,
                    bubbleTime,
                    selectionTime,
                    insertionTime,
                    mergeTime,
                    quickTime,
                    hybridTime);
            bubbleTime = 0;
            selectionTime = 0;
            insertionTime = 0;
            mergeTime = 0;
            quickTime = 0;
            hybridTime = 0;

        }

    }
}
```

2. Testing inputs and outputs

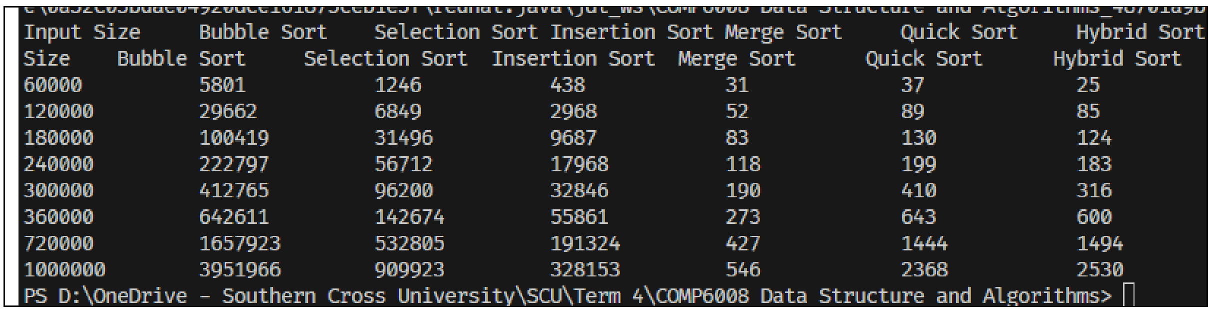
Output



```
Input Size    Bubble Sort   Selection Sort Insertion Sort Merge Sort    Quick Sort    Hybrid Sort
60000         5576          1183          362           30            13            22
120000        23046         4917          1452          15            14            18
180000        52162         9772          4560          31            20            14
240000        87251         15111         6738          37            25            33
300000        154164        33560         10351         41            26            22
360000        215690        51396         14534         54            33            26
720000        905444        181835        62822         118           76            55
1000000       1751894       486621        172455        184           155           106
```

```
Input Size    Bubble Sort   Selection Sort Insertion Sort Merge Sort    Quick Sort    Hybrid Sor
60000         6924          1427          436           43            31            61
120000        23218         4251          1732          22            31            19
180000        52560         9078          3372          24            18            13
240000        101496        17865         7132          39            25            19
300000        160398        35484         18929         56            33            42
360000        311800        52157         20856         68            46            50
720000        1164517       174828        69087         115           72            62
1000000       1733641       348819        146563        165           101           97
```

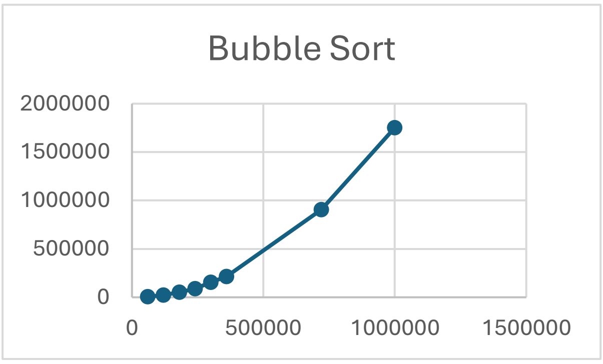
```
Input Size    Bubble Sort   Selection Sort Insertion Sort Merge Sort    Quick Sort    Hybrid Sort
60000         5252          1024          371           40            15            24
120000        20679         3825          1418          15            18            24
180000        47535         9940          3323          23            20            13
240000        85489         16576         5929          34            20            17
300000        140715        24543         9714          41            27            22
360000        195680        36372         14954         50            32            26
720000        787409        152273        66608         104           69            57
1000000       1707458       349355        127666        176           116           101
```



```
Input Size      Bubble Sort     Selection Sort Insertion Sort Merge Sort     Quick Sort      Hybrid Sort
Size     Bubble Sort     Selection Sort  Insertion Sort  Merge Sort      Quick Sort      Hybrid Sort
60000           5801            1246            438             31              37              25
120000          29662           6849            2968            52              89              85
180000          100419          31496           9687            83              130             124
240000          222797          56712           17968           118             199             183
300000          412765          96200           32846           190             410             316
360000          642611          142674          55861           273             643             600
720000          1657923         532805          191324          427             1444            1494
1000000         3951966         909923          328153          546             2368            2530
PS D:\OneDrive - Southern Cross University\SCU\Term 4\COMP6008 Data Structure and Algorithms>
```

Analysis

Although the tests were devised and developed with a 'runs' variable to adjust how many times the program would test each array size, the variable was set to 1 due to performance issues. Even when running just once, the algorithm would take nearly an hour and a half (1.44h) to finish. Even though repeatability might have been compromised, the single run results are depictive enough to demonstrate performance difference between sorting algorithms. Also, it should be noted that the last output results differ from the first - initially I was not resetting the timer between array sizes, so the screenshot on the black background were taken before the reset feature was implemented as shown in the first output screenshot.



As expected, Bubble Sort performs poorly for larger input sizes due to its O(n²) time complexity. The execution time increases dramatically, reaching nearly 20 minutes for an input size of 1,000,000 elements. It's the slowest algorithm in almost all cases, especially for large inputs. Selection Sort is consistently faster than Bubble Sort but still grows quickly as the input size increases, making it impractical for larger datasets.

Insertion Sort shows better performance than both Bubble Sort and Selection Sort, particularly for small to medium input sizes.

Merge Sort, on the other hand, is where we start to notice more perceptive differences compared to the previous algorithms with its O(n log n) performance.

Quick Sort is highly efficient with a time complexity of O(n log n) on average. It remains one of the fastest algorithms, especially for larger input sizes, even with the increase space complexity. At the maximum array size, Quick Sort performs slightly worse than Merge Sort, taking 155ms compared to Merge Sort’s 184ms. However, this could be due to the input data or implementation specifics (pivot choice, etc.). The more pre-sorted the array, the worst is its performance, approaching the worst-case scenario.

The Hybrid Sort performs the best, combining the shorter array capabilities from the Insertion Sort and the efficiency of Quick Sort for larger arrays.

For the largest input size (1,000,000), it takes 106ms, making it the fastest algorithm in this test. This suggests the hybrid approach is highly efficient for large datasets.

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

- What resource?
  ChatGPT
- How did you use it?

  Converting the screenshot to a CSV file to be imported on Excel.