Report

Q1a:

Sorted ArrayLists for a week:

```
TERMINAL
PS C:\Important Uni files\uni_coursework\object_orientented_java_programming_coursework\spring part 2> cd "c:\Important Uni files\uni_coursewor
k\object_orientented_java_programming_coursework\spring part 2\" ; if ($?) { javac Q1aQ1b.java } ; if ($?) { java Q1aQ1b }
Printing out the three types of arrayLists:
Sorted ArrayLists:
Day 1 (Size: 1000):
First item: 1087
Last item: 499941


Day 2 (Size: 5000):
First item: 1045
Last item: 499853


Day 3 (Size: 10000):
First item: 1017
Last item: 499965


Day 4 (Size: 50000):
First item: 1000
Last item: 499995


Day 5 (Size: 75000):
First item: 1012
Last item: 500000


Day 6 (Size: 100000):
First item: 1004
Last item: 499987


Day 7 (Size: 500000):
First item: 1000
Last item: 499996
```

Unsorted Random Arraylists for a week:

```
TERMINAL
Unsorted ArrayLists:
Day 1 (Size: 1000):
First item: 358832
Last item: 172153


Day 2 (Size: 5000):
First item: 113970
Last item: 372102


Day 3 (Size: 10000):
First item: 296886
Last item: 82709


Day 4 (Size: 50000):
First item: 255138
Last item: 240792


Day 5 (Size: 75000):
First item: 306854
Last item: 267663


Day 6 (Size: 100000):
First item: 354104
Last item: 277242


Day 7 (Size: 500000):
First item: 272743
Last item: 265521
```

Reverse Sorted ArrayLists for a week:

Report

```
TERMINAL
Reverse Sorted ArrayLists:
Day 1 (Size: 1000):
First item: 499941
Last item: 1087
Last item: 1045


Day 3 (Size: 10000):
First item: 499965
Last item: 1017


Day 4 (Size: 50000):
First item: 499995
Last item: 1000


Day 5 (Size: 75000):
First item: 500000
Last item: 1012


Day 6 (Size: 100000):
First item: 499987
Last item: 1004


Day 7 (Size: 500000):
First item: 499996
Last item: 1000
```

Pseudocode for quicksort algorithm:

```
function quickSort(list, low, high):
  create an empty stack
  push low and high onto stack

  while stack is not empty:
    high = pop from stack
    low = pop from stack

    pivot = partition(list, low, high)

    if pivot - 1 > low:
      push low onto stack
      push (pivot - 1) onto stack

    if pivot + 1 < high:
      push (pivot + 1) onto stack
      push high onto stack
```

Report

function partition(list, low, high):

```
  pivotIndex = (low + high) / 2
  pivot = list[pivotIndex]
  swap list[pivotIndex] with list[high]

  i = low - 1

  for j from low to high - 1:
    if list[j] <= pivot:
      i = i + 1
      swap list[i] with list[j]

  swap list[i + 1] with list[high]
  return (i + 1)
```

Q1b:

Efficiency Testing for Quicksort Algorithm:

```
Passing the three types of ArrayLists through the quickSort Sorting Algorithm:
Sorting Random ArrayLists:
System took 299 ms to run.
Sorting Reverse Sorted ArrayLists:
System took 153 ms to run.
Sorting already sorted ArrayLists:
System took 138 ms to run.
```

Q1c:

Efficiency Testing for Hybrid Algorithm:

```
Passing the three types of ArrayLists through the Hybrid Sorting Algorithm:
Sorting Random ArrayLists:
System took 172 ms to run.
Sorting Reverse Sorted ArrayLists:
System took 132 ms to run.
Sorting already sorted ArrayLists:
System took 116 ms to run.
```

Report

Q1d:

Development of the Hybrid Sorting Algorithm

The hybrid sorting algorithm was created to combine the strengths of QuickSort and Insertion Sort. QuickSort is highly efficient for large datasets, but its overhead makes it less ideal for small subarrays. Insertion Sort, on the other hand, excels at handling small or nearly sorted lists. The challenge was to integrate these two sorting methods effectively, using QuickSort for partitioning while switching to Insertion Sort for small sections to improve efficiency.

Different Approaches Considered

Initially, a recursive QuickSort was implemented with a middle pivot. This worked well for random lists but struggled with sorted and reverse-sorted lists, leading to deep recursion and performance inefficiencies. To counter this, QuickSort was rewritten as an iterative version using an explicit stack (ArrayDeque), which reduced memory usage and eliminated the risk of stack overflow.

The final approach combined QuickSort and Insertion Sort. A threshold-based switching mechanism was introduced: when subarrays contained fewer than 60 elements, the algorithm transitioned to Insertion Sort, preventing unnecessary QuickSort partitioning. Additionally, Insertion Sort was used as a finishing step to optimize performance on nearly sorted lists.

Key Decisions and Their Impact

Pivot Selection

The middle pivot was chosen to maintain balanced partitions and prevent skewed splits when dealing with sorted inputs. While random pivot selection was tested, it led to inconsistent performance, so the middle pivot was retained.

Threshold for Switching to Insertion Sort

Several threshold values (10, 20, 30, 50) were tested, but 60 was found to be the best balance between QuickSort's fast partitioning and Insertion Sort’s efficiency for small lists.

Performance Across Different Input Types

The hybrid sorting algorithm demonstrated O(n log n) complexity for random lists, leveraging QuickSort’s efficiency. For sorted and reverse-sorted lists, worst-case scenarios were mitigated, ensuring a significant performance boost compared to a standard QuickSort implementation.

Efficiency & Runtime Complexity

Report

Approach	**Average runtime (Random ArrayLists)**	**Average runtime (Sorted ArrayLists)**	**Average runtime (Reverse Sorted ArrayLists)**
Iterative QuickSort	275 ms	129 ms	149 ms
Hybrid QuickSort-Insertion Sort	174 ms	120 ms	131 ms

The hybrid approach improves sorting efficiency across all cases, particularly benefiting sorted and nearly sorted lists. By reducing recursion depth and using explicit stack partitioning, memory usage is also significantly lowered.

Conclusion

The development of the hybrid sorting algorithm focused on refining QuickSort's partitioning process while incorporating Insertion Sort for smaller subarrays. Through testing and optimization, a 60-element threshold and middle pivot selection were determined as the best parameters for balanced performance across various input types. This approach strikes an effective balance between speed and memory efficiency, making it a practical solution for real-world sorting applications.