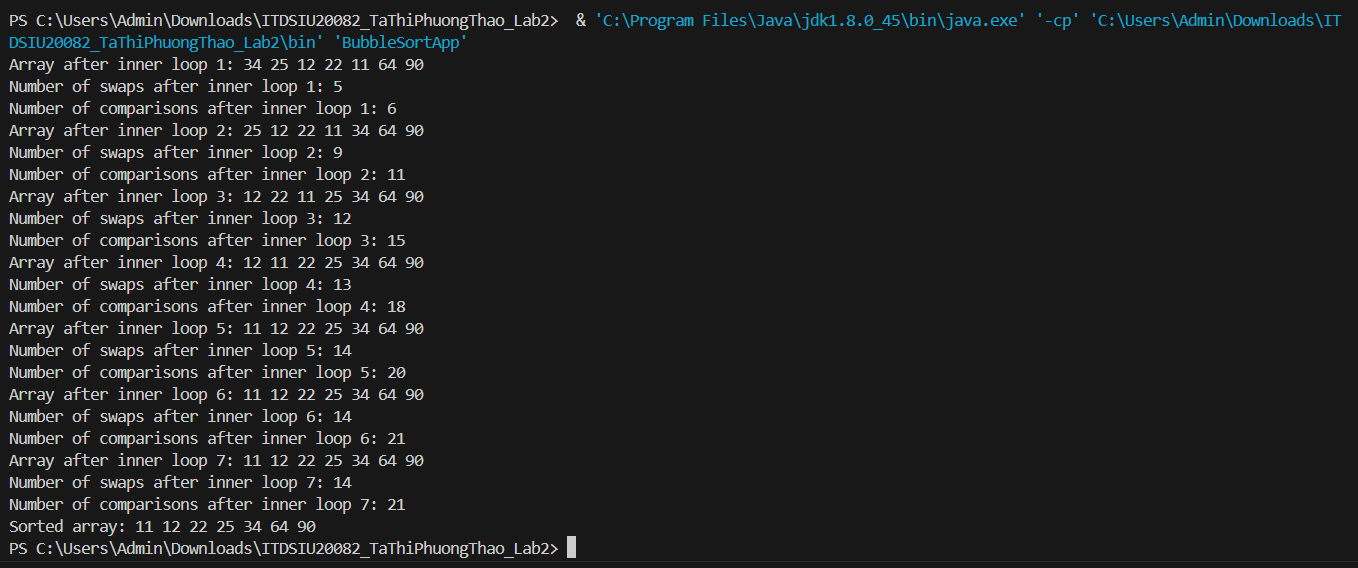
**Problem 1:**

**A screen shot of a computer program

Description automatically generated**

Output:

****

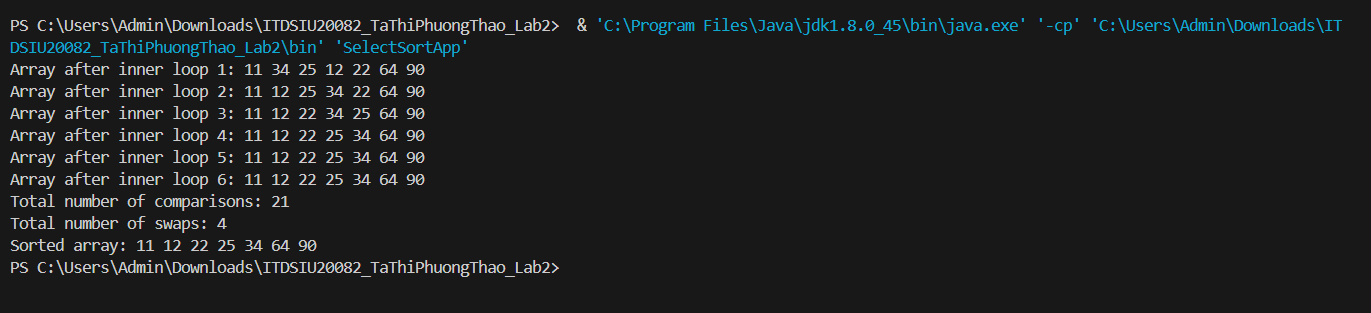
BubbleSort iterates through the array, comparing adjacent elements and swapping them if necessary to gradually move larger elements towards the end of the array. The number of swaps and comparisons made during each iteration provides insight into the algorithm's performance and efficiency. The sorted array at the end indicates that the algorithm successfully sorts the array in ascending order.

**Problem 2:**

**A screen shot of a computer program

Description automatically generated**

Output:

****

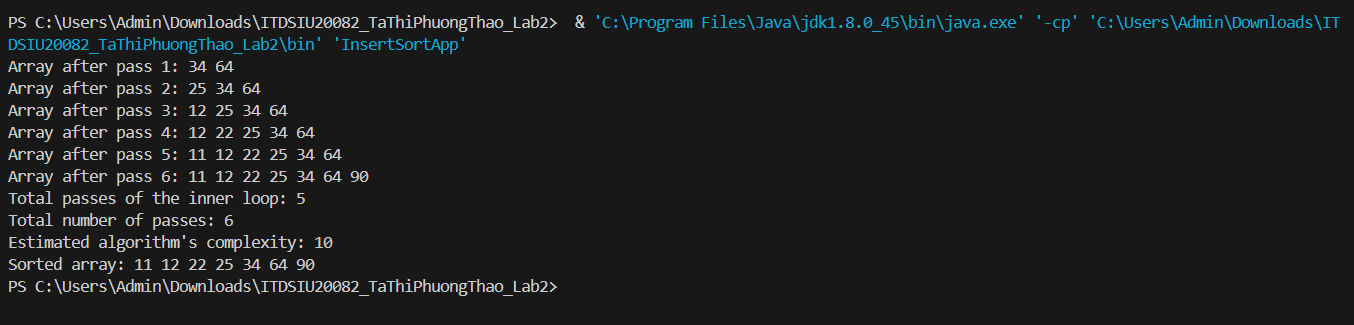
SelectSort works by iteratively selecting the smallest (or largest) element from the unsorted portion of the array and moving it to its correct position in the sorted portion. In each inner loop iteration, the smallest element is identified and swapped with the element at the current position. This process repeats until the array is sorted. The number of comparisons is reduced because it only swaps elements when necessary.

**Problem 3:**

**A screen shot of a computer program

Description automatically generated**

Output:

****

InsertSort algorithm iteratively builds a sorted array by inserting each element into its correct position in the sorted subarray. With each pass, it ensures that the elements to the left are sorted, gradually expanding the sorted region until the entire array is sorted. The complexity of this variation seems to be around O(n^2), considering the number of total passes and the total number of passes of the inner loop. Despite its simplicity, it performs adequately for small datasets but may be inefficient for larger ones."

**Problem 4:**

**A screen shot of a computer screen

Description automatically generated**

|  |  |  |  |
| --- | --- | --- | --- |
| **COPIES/ COMPARISONS/ SWAPS** | | | |
|  | **Bubble Sort** | **Selection Sort** | **Insertion Sor** |
| 10000 | 118171800 ns | 35800800 ns | 53160400 ns |
| 15000 | 264172600 ns | 55516600 ns | 98092200 ns |
| 20000 | 453302600 ns | 155936800 ns | 49720000 ns |
| 25000 | 724291900 ns | 143750500 ns | 77324400 ns |
| 30000 | 1084437500 ns | 207642700 ns | 110700700 ns |
| 35000 | 1421313900 ns | 274658400 ns | 147561200 ns |
| 40000 | 1908798900 ns | 379388300 ns | 219627200 ns |
| 45000 | 2648322700 ns | 464681900 ns | 247007800 ns |
| 50000 | 3099665000 ns | 636223100 ns | 295248200 ns |

The analysis indicates that for larger array sizes, all three sorting methods exhibit poor performance due to their quadratic time complexity. While Insertion Sort may outperform Bubble Sort and Selection Sort for smaller datasets, it is generally less efficient than more advanced algorithms like Merge Sort or Quick Sort for larger datasets. Consideration of the specific characteristics of the input data and the desired trade-offs between time complexity, space complexity, and stability is crucial in selecting the most suitable sorting algorithm for a given scenario.

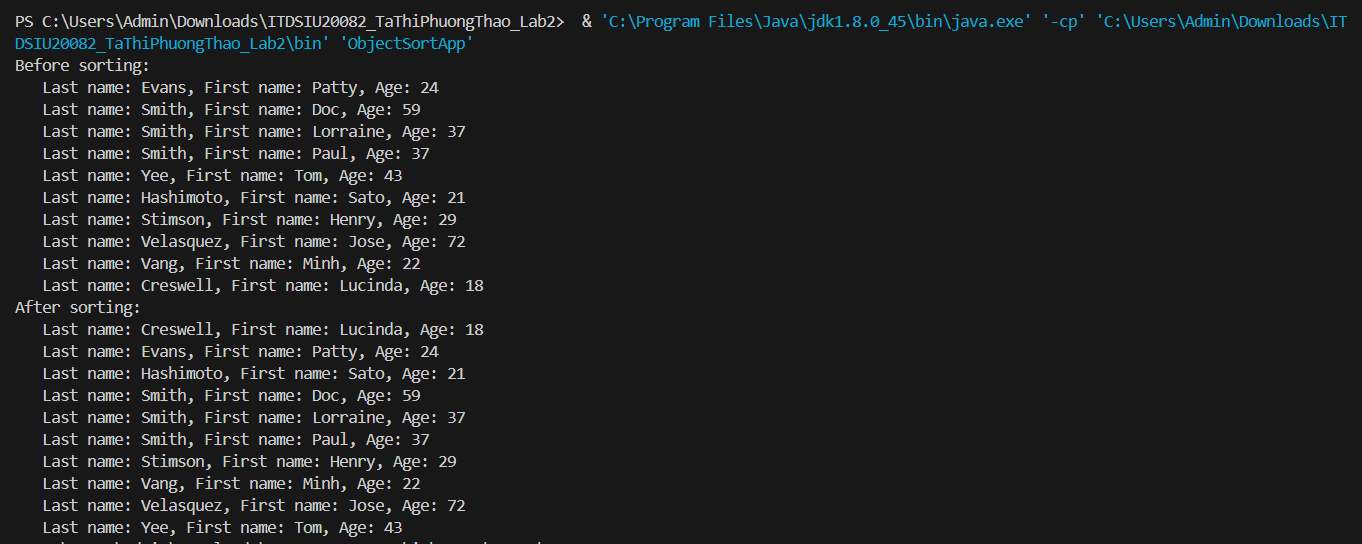
**Problem 5:**

**A computer screen shot of a program code

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**



The objects are sorted primarily by last name, and in cases where the last names are the same, by first name. Age appears to serve as a secondary sorting criterion. The sorting algorithm employed likely follows a stable sorting method, ensuring that elements with the same key values retain their relative order. Overall, the output demonstrates a successful sorting process, organizing the objects in ascending order according to the specified criteria.

**Problem 6**

**A computer screen shot of code

Description automatically generated**

Flight Class:

* flightId: A unique identifier for each flight
* time: The scheduled time for the flight in 24-hour format
* priority: An integer that determines the importance of the flight (lower numbers represent higher priority)
* getTime(): Returns the scheduled time of the flight
* getPriority(): Returns the priority of the flight
* toString(): Overrides the toString() method to return a readable representation of the flight in the form (flightId, time)

A screen shot of a computer screen

Description automatically generated

RunwaySchedule Class:

scheduleFlights():

+Input: Takes in a list of Flight objects and an integer representing the number of available runways.

+Sorting:

* The flights are first sorted based on priority. If two flights have the same priority, they are sorted based on time (earlier times are prioritized) by using Collections.sort() method.

+Runway Assignment:

* For each flight, the function checks all available runways to see if it can be assigned without a time conflict.
* If a flight can be assigned to a runway without conflict, it is added to that runway's list.
* If a flight cannot be scheduled due to a time conflict on all runways, an error message is displayed for that flight.

+Sorting After Assignment:

* Once assigned, each runway's flights are sorted by time to ensure proper order.

MAIN:

A screen shot of a computer program

Description automatically generated

printRunways()

Output:

A screen shot of a computer program

Description automatically generated

* Data structures: list (ArrayList) to store flight and runways.
* Algorithm: sorting (first is priority, second is time)
* Time complexity: O(nlogn + R \* n)