**Day 16**

Task 01

Write an algorithm / steps for selection sort.

1. **Start** from the full array of n elements.
2. Let i go from n-1 down to 1 (right to left).
3. Initialize max\_index ← 0 (assume the first element is the largest).
4. For j = 1 to i, do:
   * If array[j] > array[max\_index], update max\_index ← j.
5. After the inner loop, **swap** array[i] and array[max\_index].
6. Now the largest element is at position i, and that part is considered **sorted**.
7. **Repeat** until all elements are sorted.
8. **End.**

**Task 02**

Write a pseudo code for the selection sort

Ans : for i from n-1 down to 1 do

max\_index ← 0

for j from 1 to i do

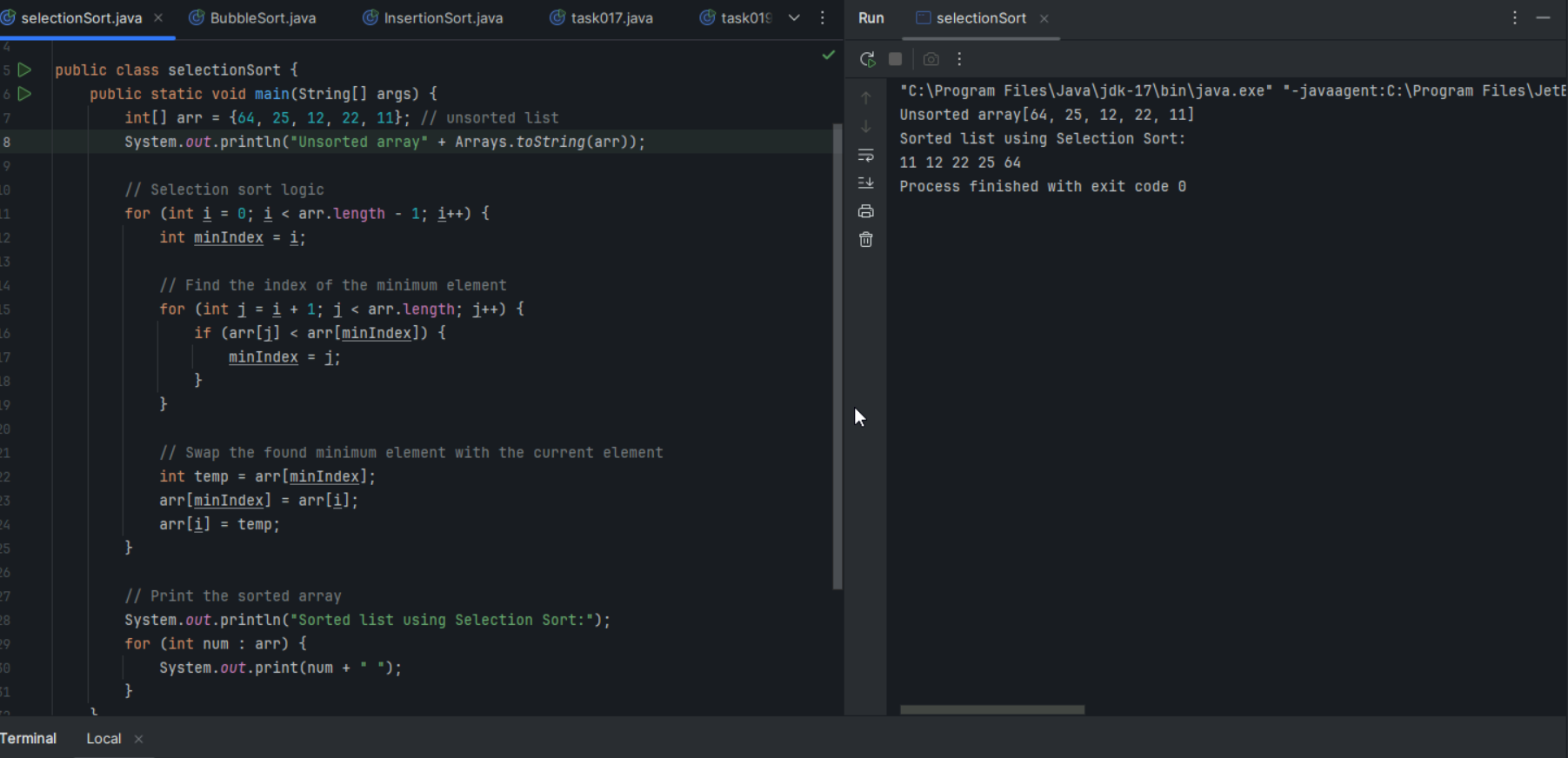
if array[j] > array[max\_index] then

max\_index ← j

swap array[i] and array[max\_index]

**Task 03**

Wap to make sure your list is sorted using selection sort.



Task 04

Write an algorithm / steps for selection sort.

1. **Start** from the first element of the array.
2. **Compare** the current element with the next one.
3. If the current element is **greater**, **swap** them.
4. Move to the next pair and **repeat step 3** for the rest of the array.
5. After the first pass, the **largest element is at the end**.
6. **Repeat steps 1 to 5** for the remaining elements (excluding the last sorted ones).
7. Continue this process until **no more swaps are needed**, i.e., the array is sorted.
8. **End.**

**Task 05**

**for i from 0 to n - 2 do**

**swapped ← false**

**for j from 0 to n - i - 2 do**

**if array[j] > array[j + 1] then**

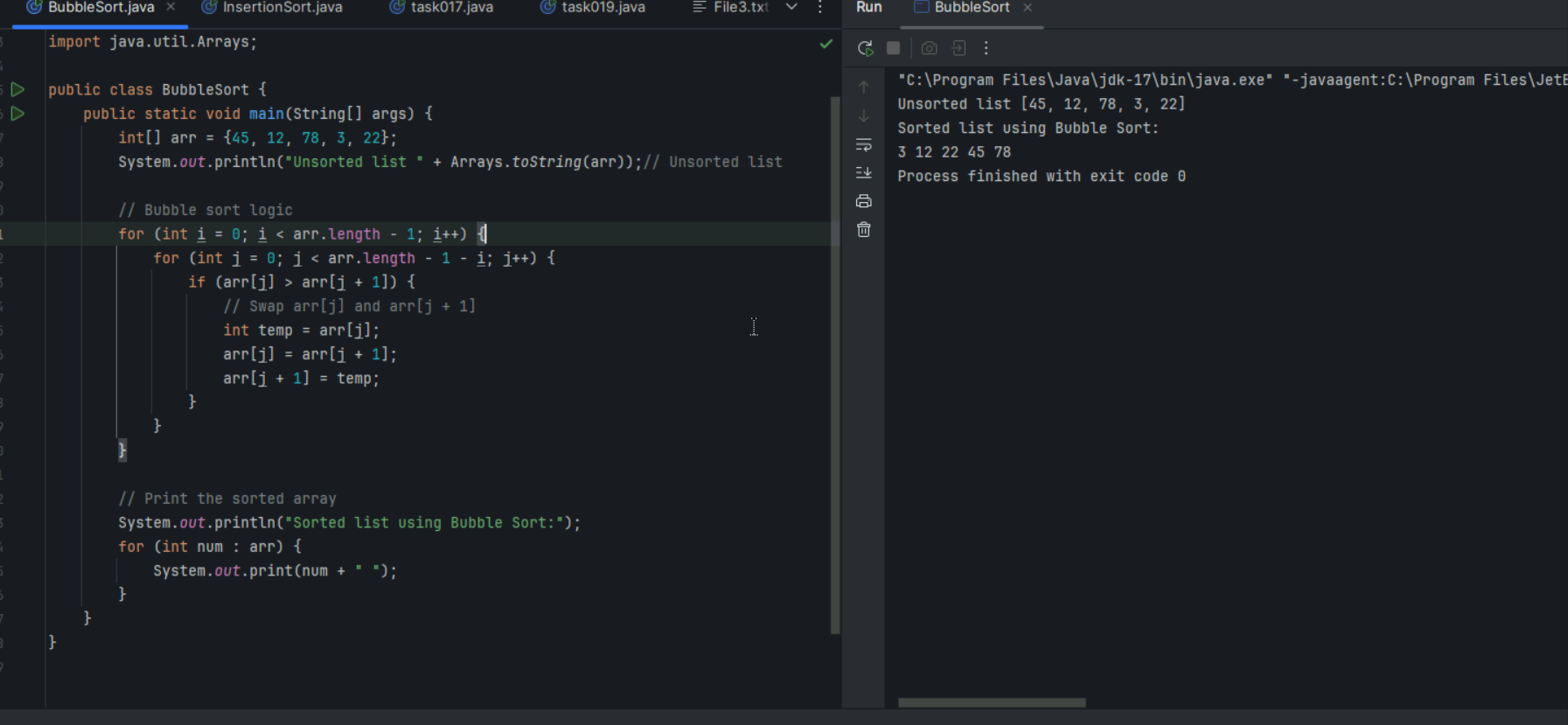
**swap array[j], array[j + 1]**

**swapped ← true**

**if swapped = false then**

**break**

**Task 06**

****

Task 07

Write an algorithm for the Insertion sort.

1. Start from the **second element** (index 1); assume the first element is already sorted.
2. Store the current element in a **temporary variable** (let's call it key).
3. Compare key with the elements in the sorted portion (from right to left).
4. **Shift** elements greater than key one position to the right.
5. Insert key in its correct position.
6. Repeat for all elements until the array is sorted.
7. **End.**

Task 08

Pseudo code

for i from 1 to n-1 do

key ← array[i]

j ← i - 1

while j ≥ 0 and array[j] > key do

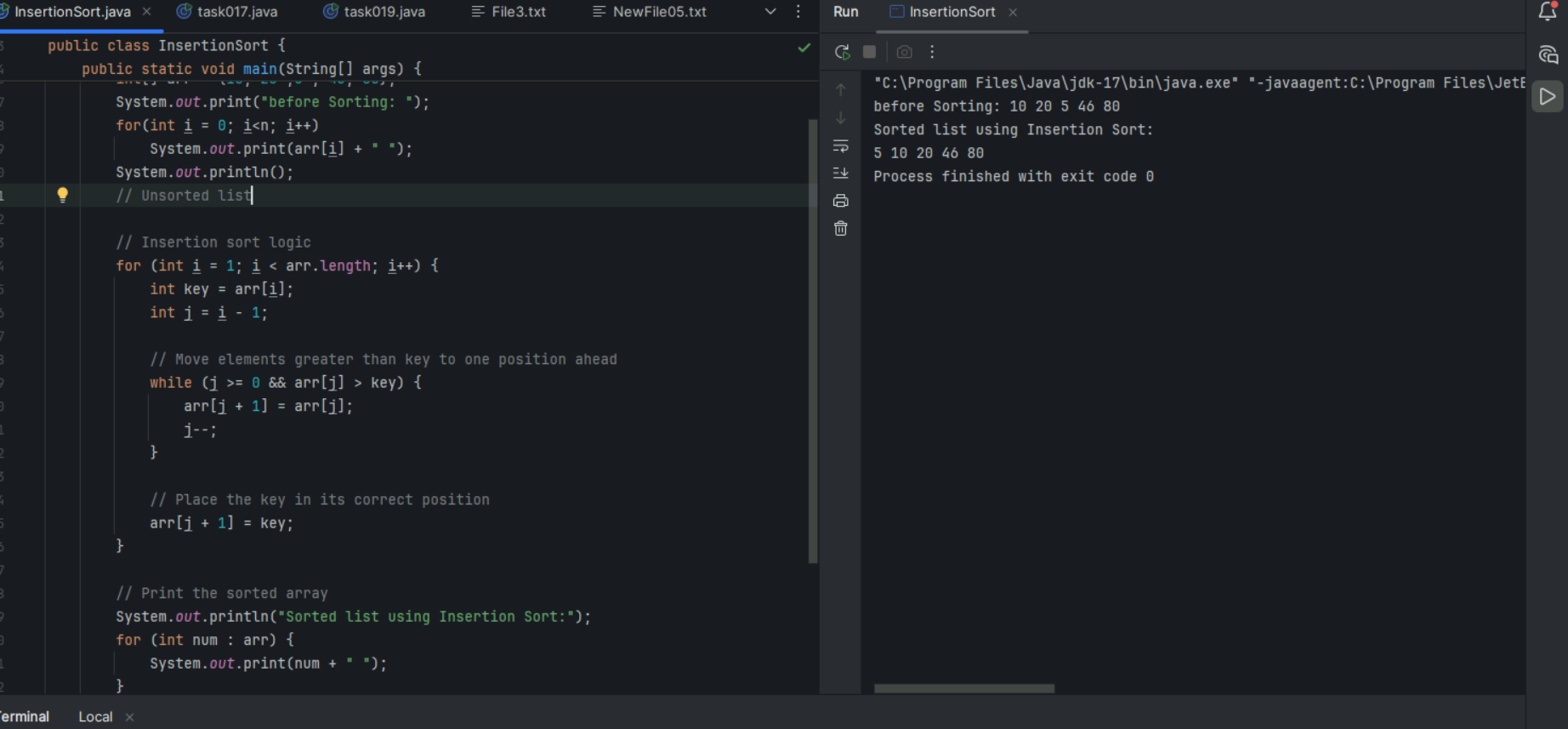
array[j + 1] ← array[j]

j ← j - 1

array[j + 1] ← key

TASK 09

Program using insertion sort



Task 10

What are the advantages and disadvantages of Bubble sort Algo?

List them

Advantages :

1. **Simple to understand and implement**
   * Logic is easy to grasp for beginners.
2. **Requires no extra space (In-place sort)**
   * Only swaps elements within the array — space complexity is **O(1)**.
3. **Stable sort**
   * It maintains the relative order of records with equal keys (important for sorting objects).
4. **Best case performance is O(n)**
   * If the array is already sorted, only one pass is needed (with an optimized version using a swap flag)

Disadvantages :

1. **Poor performance on large lists**
   1. Time complexity is **O(n²)** in the average and worst cases.
2. **Unnecessary comparisons**
   1. Even if the array is sorted early, unoptimized versions still continue comparing all elements.
3. **Too many swaps**
   1. Repeated swapping makes it **slower than insertion sort**, especially on large data sets.
4. **Not suitable for real-world applications**
   1. Inefficient compared to more advanced sorting algorithms like Merge Sort, Quick Sort, or even Insertion Sort.
5. **Wastes time for partially sorted arrays**
   * 1. Without optimization, it doesn’t take advantage of already sorted parts.

Task 12

Merge sort

**1. Divide:**

* If the array has more than one element:
  + Find the middle index.
  + Split the array into two halves: left and right.

**2. Conquer (Recursively sort):**

* Recursively apply merge sort to both halves.

**3. Combine (Merge):**

* Merge the two sorted halves into a single sorted array.

Pseudo code :

function mergeSort(array):

if length of array ≤ 1:

return array

mid ← length(array) / 2

leftHalf ← mergeSort(array[0 to mid-1])

rightHalf ← mergeSort(array[mid to end])

return merge(leftHalf, rightHalf)

function merge(left, right):

result ← empty array

while left and right are not empty:

if left[0] ≤ right[0]:

append left[0] to result

remove left[0] from left

else:

append right[0] to result

remove right[0] from right

append any remaining elements from left to result

append any remaining elements from right to result

return result

Algorithm for the Quick sort

#### 1. ****Choose a Pivot:****

* Select any element as the pivot (e.g., first, last, middle, or random element).

#### 2. ****Partition:****

* Rearrange the array so that:
  + Elements **less than pivot** go to the **left**.
  + Elements **greater than pivot** go to the **right**.
  + Pivot is in its **correct sorted position**.

#### 3. ****Recursive Sort:****

* Recursively apply quick sort to the **left** and **right** subarrays (excluding the pivot).

#### 4. ****Base Case:****

* If the array has 0 or 1 element, it's already sorted.

Pseudo code ;

function quickSort(array, low, high):

if low < high then

pivotIndex ← partition(array, low, high)

quickSort(array, low, pivotIndex - 1)

quickSort(array, pivotIndex + 1, high)

function partition(array, low, high):

pivot ← array[high] // Choose last element as pivot

i ← low - 1 // Index of smaller element

for j ← low to high - 1 do

if array[j] ≤ pivot then

i ← i + 1

swap array[i], array[j]

swap array[i + 1], array[high]

return i + 1 // Pivot index