DAY 16

SAIKUMAR PENTAKOTA

TASK1

Selection Sort Algorithm

1: Set minIndex to position 0

2: Search for the smallest element in the unsorted subarray and update minIndex

3: Swap the element at the position minIndex with the first element of the unsorted subarray.

4: Again set minIndex to the first position of the unsorted subarray

5: Repeat steps 2 to 4 until the array gets sorted

6: print the sorted array

TASK 2

arr : array of integers

n : size of the array

for i =1 to n-1

minIndex=i

for j =i+1 to n

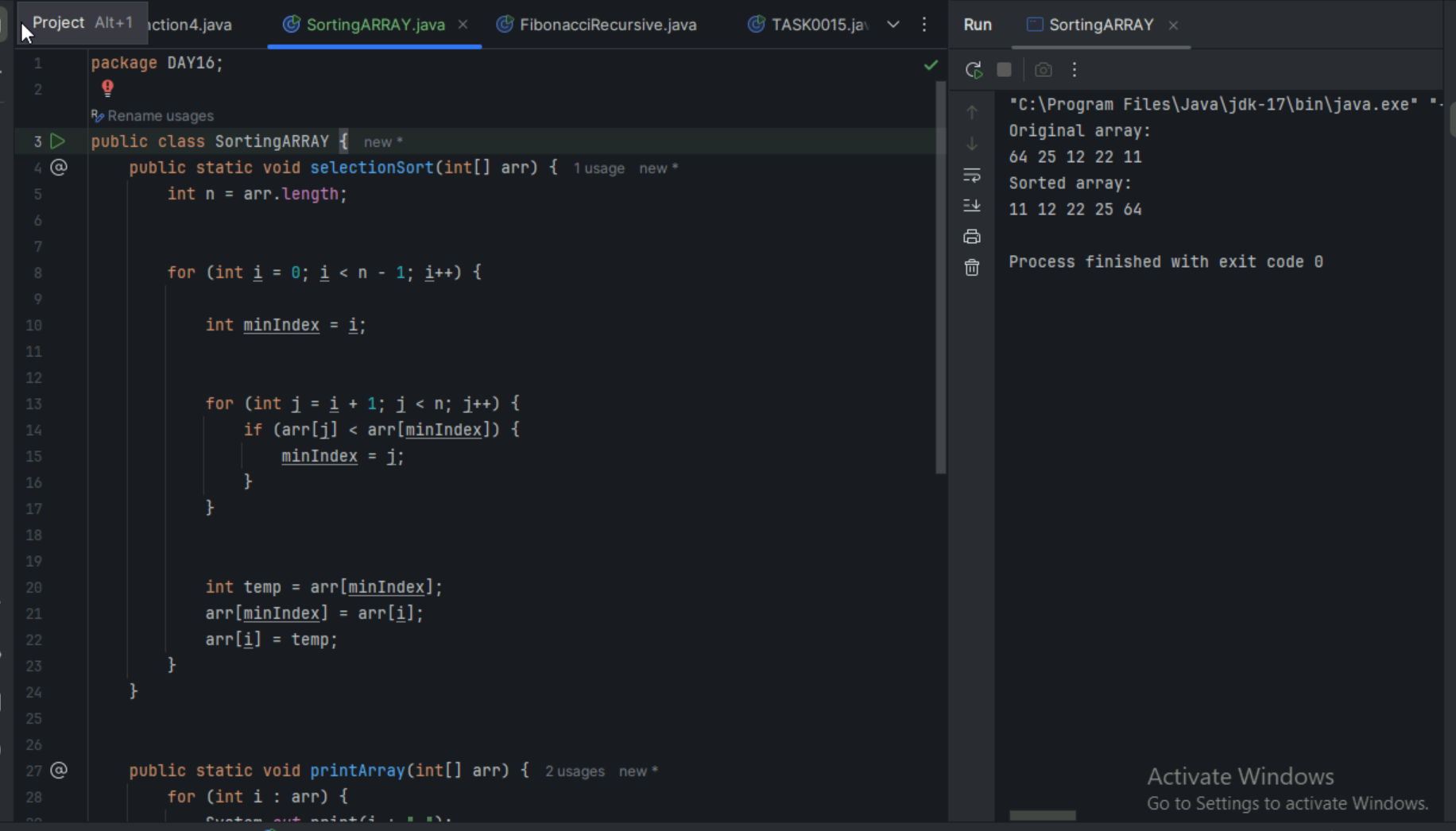
if(arr[j] < arr[minIndex])

minIndex=j

end if

end for

TASK 3



**Task4**

Check if the first element in the input array is greater than the next element in the array.

If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Repeat Step 2 until we reach the end of the array.

Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

The final output achieved is the sorted array.

**TASK5**

FOR i = 0 TO n-1

swapped = false

FOR j = 0 TO n-i-1

IF A[j] > A[j+1]

SWAP A[j] with A[j+1]

swapped = true

END IF

END FOR

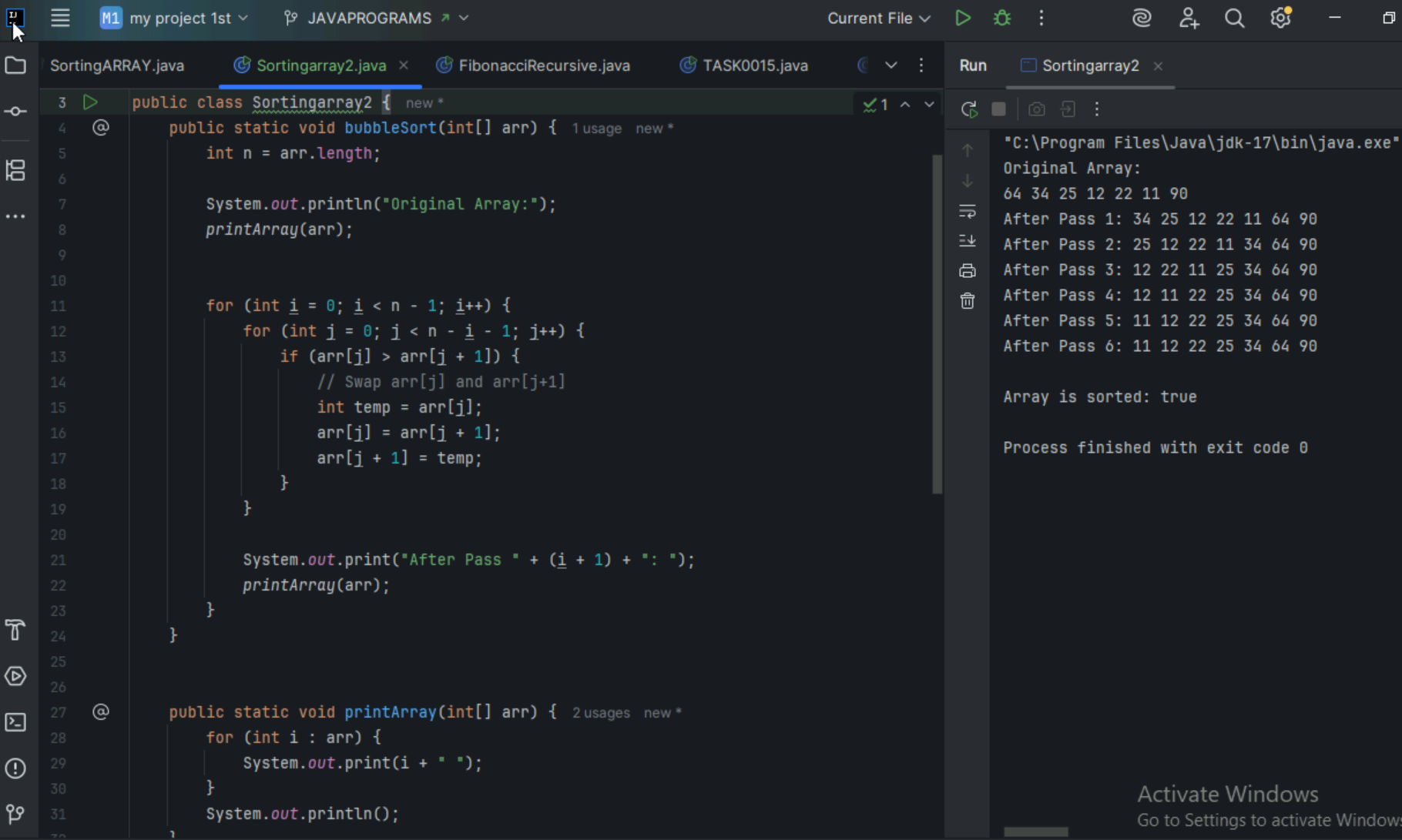
IF swapped = false

BREAK

END IF

END FOR

**Task 6**



TASK 7

Start with the first element as sorted portion

Take next element and insert it into sorted portion at correct position

Repeat step 2 until all elements are sorted

print the sorted array

Task8

FOR i = 1 to n-1

key = A[i]

j = i-1

WHILE j >= 0 and A[j] > key

A[j+1] = A[j]

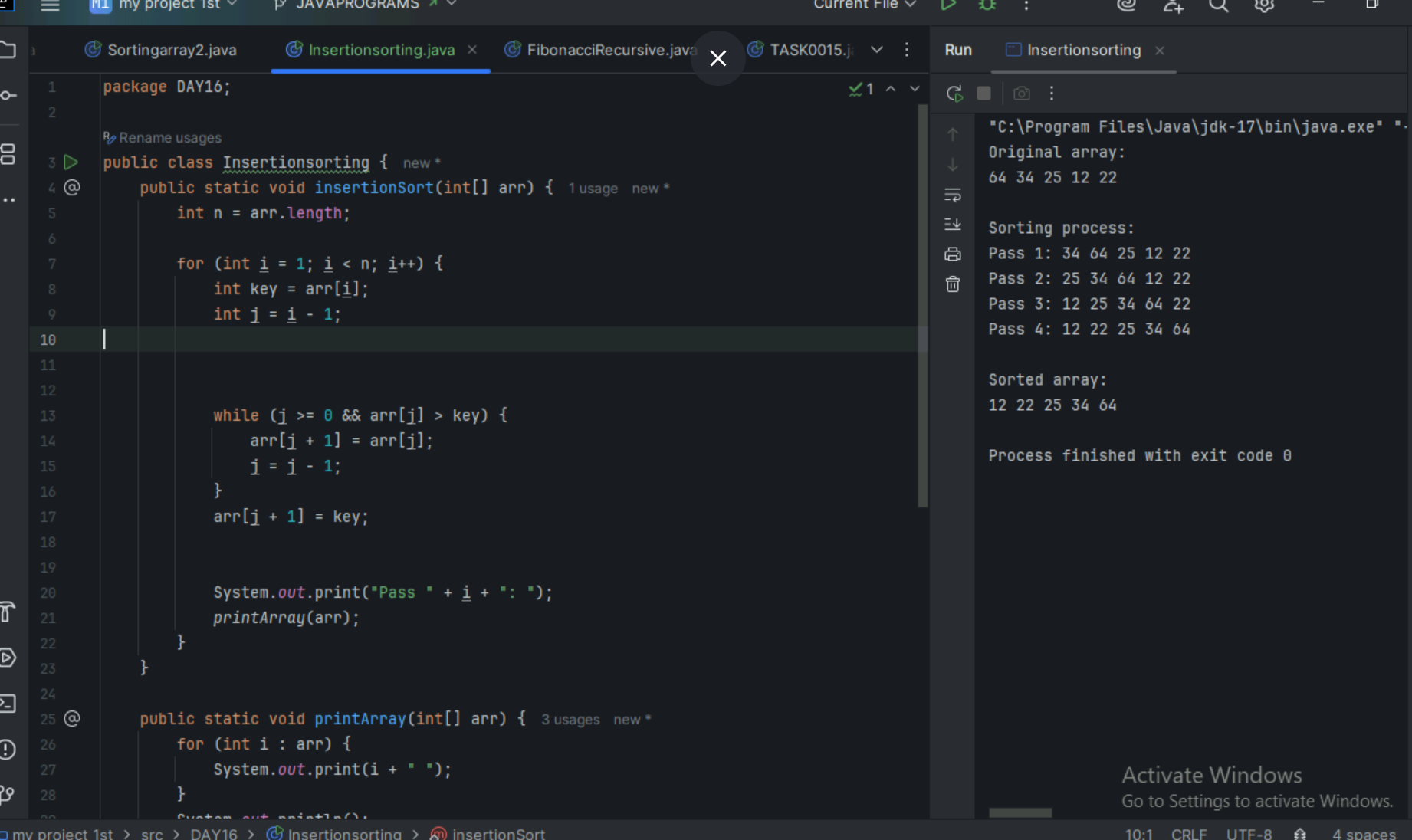
j = j-1

END WHILE

A[j+1] = key

END FOR

TASK9



Task10

Advantages

Simple Implementation

Memory Efficient

Stable Sorting

Adaptive

No Additional Memory Space

Easy to Detect if Array is Sorted

DISADVANTAGES:

Poor Time Complexity

O(n²)

Poor Performance

Not Suitable for Large Datasets

High Number of Comparisons

High Number of Swaps

Not Cache Friendly

No Branch Prediction

Task 11

Divide the array into two halves:

Find the middle index mid = (low + high) / 2

Divide array A[low...high] into:

Left half: A[low...mid]

Right half: A[mid+1...high]

Recursively apply merge sort to both halves:

Sort the left half

Sort the right half

Merge the two sorted halves into one sorted array:

Create two temporary arrays for left and right

Compare elements from each and build the final sorted array

Task12

MERGE\_SORT(A, low, high):

if low < high:

mid = (low + high) / 2

MERGE\_SORT(A, low, mid)

MERGE\_SORT(A, mid + 1, high)

MERGE(A, low, mid, high)

MERGE(A, low, mid, high):

n1 = mid - low + 1

n2 = high - mid

create arrays L[0...n1-1] and R[0...n2-1]

copy A[low...mid] into L

copy A[mid+1...high] into R

i = 0, j = 0, k = low

while i < n1 and j < n2:

if L[i] <= R[j]:

A[k] = L[i]

i = i + 1

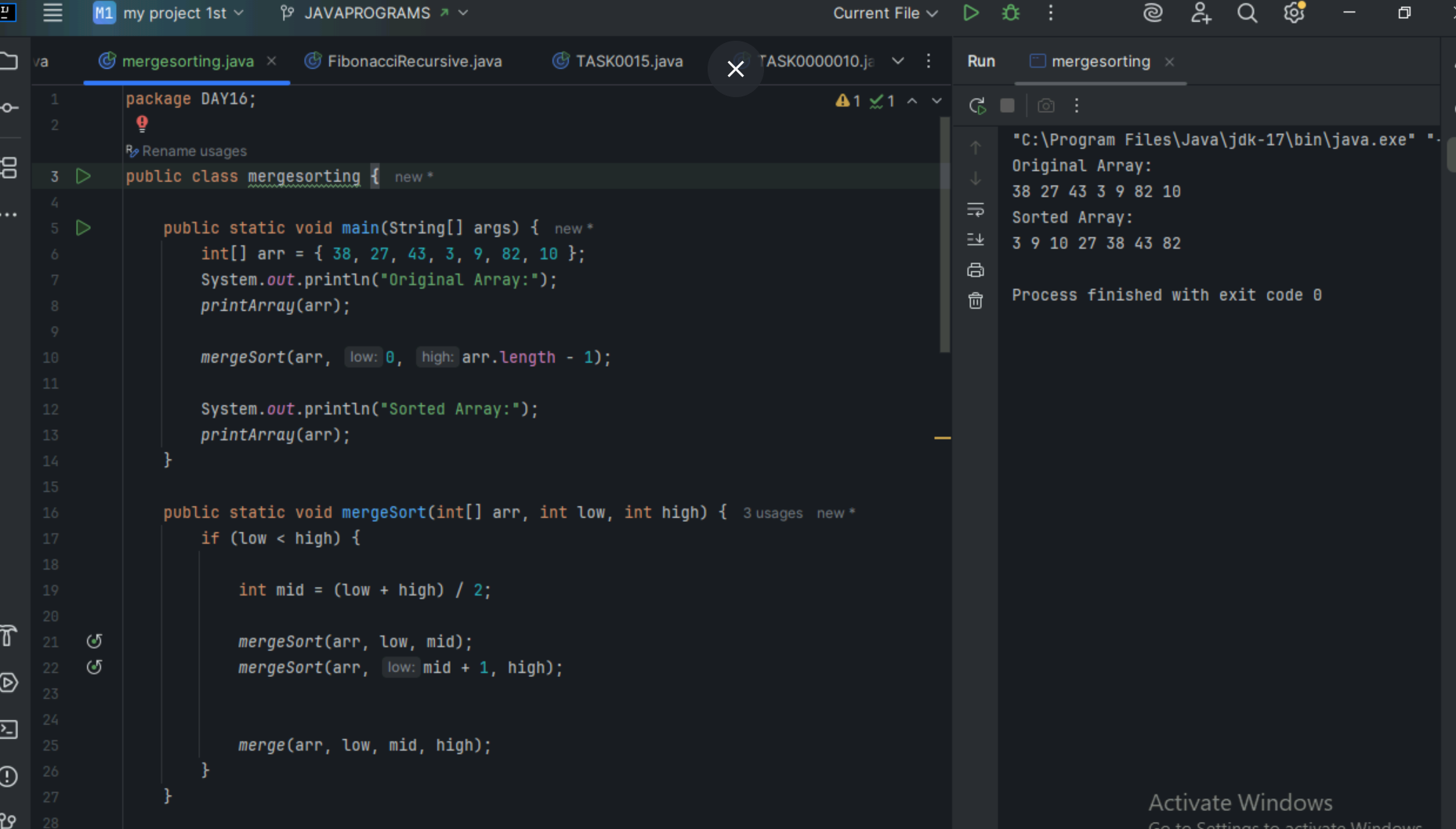
else:

A[k] = R[j]

j = j + 1

k = k + 1

Task 13



Task 15

Choose the highest index value has pivot

Take two variables to point left and right of the list excluding pivot

Left points to the low index

Right points to the high

While value at left is less than pivot move right

While value at right is greater than pivot move left

If both step 5 and step 6 does not match swap left and right

If left ≥ right, the point where they met is new pivot

TASK16

QUICKSORT(arr, low, high):

if low < high:

pivotIndex = PARTITION(arr, low, high)

QUICKSORT(arr, low, pivotIndex - 1)

QUICKSORT(arr, pivotIndex + 1, high)

PARTITION(arr, low, high):

pivot = arr[high]

i = low - 1

for j from low to high - 1:

if arr[j] <= pivot:

i = i + 1

swap arr[i] with arr[j]

swap arr[i + 1] with arr[high]

return i + 1

Task 17

