Day-16

Name- Rishabh Manchanda  
  
Employee ID: 109883998

Task-1

What is the algorithm of selection sort ?

1. Start with the first element of the array as the minimum.
2. Iterate through the array to find the smallest element in the unsorted part.
3. Swap the found minimum element with the first element of the unsorted part.
4. Move the boundary of the sorted and unsorted parts one element to the right.
5. Repeat steps 1-4 until the entire array is sorted.

Pseudo code

function selectionSort(array):

n = length(array) // Get the number of elements in the array

for i from 0 to n-2: // Iterate over each element in the array

minIndex = i // Assume the minimum is the first element in the unsorted part

for j from i+1 to n-1: // Iterate through the unsorted part

if array[j] < array[minIndex]: // Find the smallest element

minIndex = j // Update the index of the minimum element

// Swap the found minimum element with the first element of the unsorted part

temp = array[minIndex]

array[minIndex] = array[i]

array[i] = temp

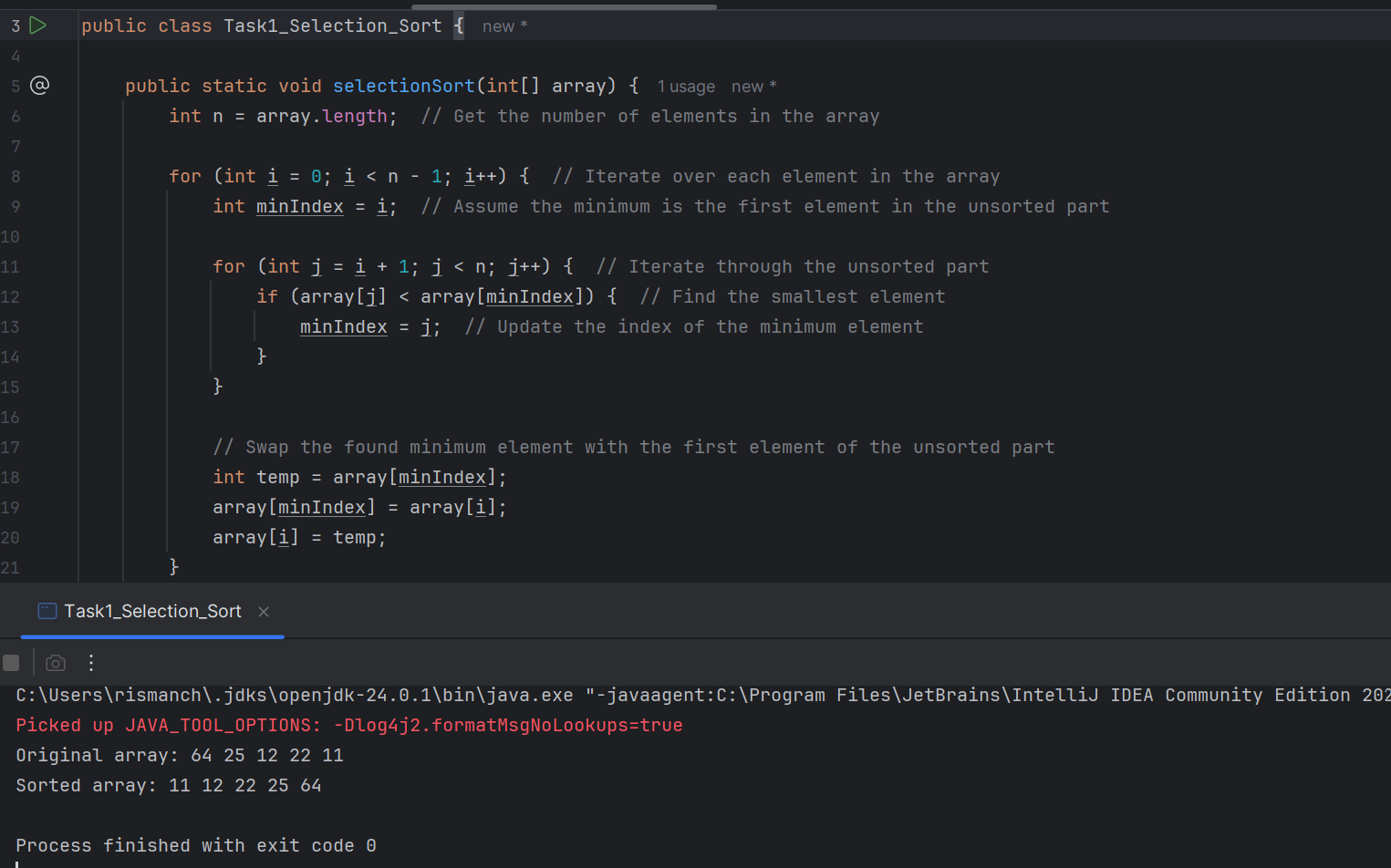
// Main function to execute the program

function main():

array = [64, 25, 12, 22, 11] // Example array

print("Original array:", array) // Print the original array

selectionSort(array) // Call the selection sort function

print("Sorted array:", array) // Print the sorted array  
  
  


Task- 4  
Write algorithm of bubble sort?

1. bubbleSort Method:
   * This method takes an array as input and sorts it using the Bubble Sort algorithm.
   * It uses two nested loops: the outer loop for each pass and the inner loop for comparing adjacent elements.
   * A boolean flag swapped is used to track whether any swaps were made during a pass. If no swaps occur, the array is already sorted, and the algorithm can terminate early.
2. main Method:
   * Initializes an example array {64, 34, 25, 12, 22, 11, 90}.
   * Prints the original array.
   * Calls the bubbleSort method to sort the array.
   * Prints the sorted array.
3. printArray Helper Method:
   * This method prints the elements of the array in a readable format.

Task-5

Pseudo code   
  
function bubbleSort(array):

n = length(array) // Get the number of elements in the array

for i from 0 to n-2: // Outer loop for each pass

swapped = false // Flag to check if a swap occurred

for j from 0 to n-i-2: // Inner loop for comparing adjacent elements

if array[j] > array[j+1]: // Compare adjacent elements

// Swap if they are in the wrong order

temp = array[j]

array[j] = array[j+1]

array[j+1] = temp

swapped = true // Set the flag to true

if not swapped: // If no swaps occurred, the array is already sorted

break

// Main function to execute the program

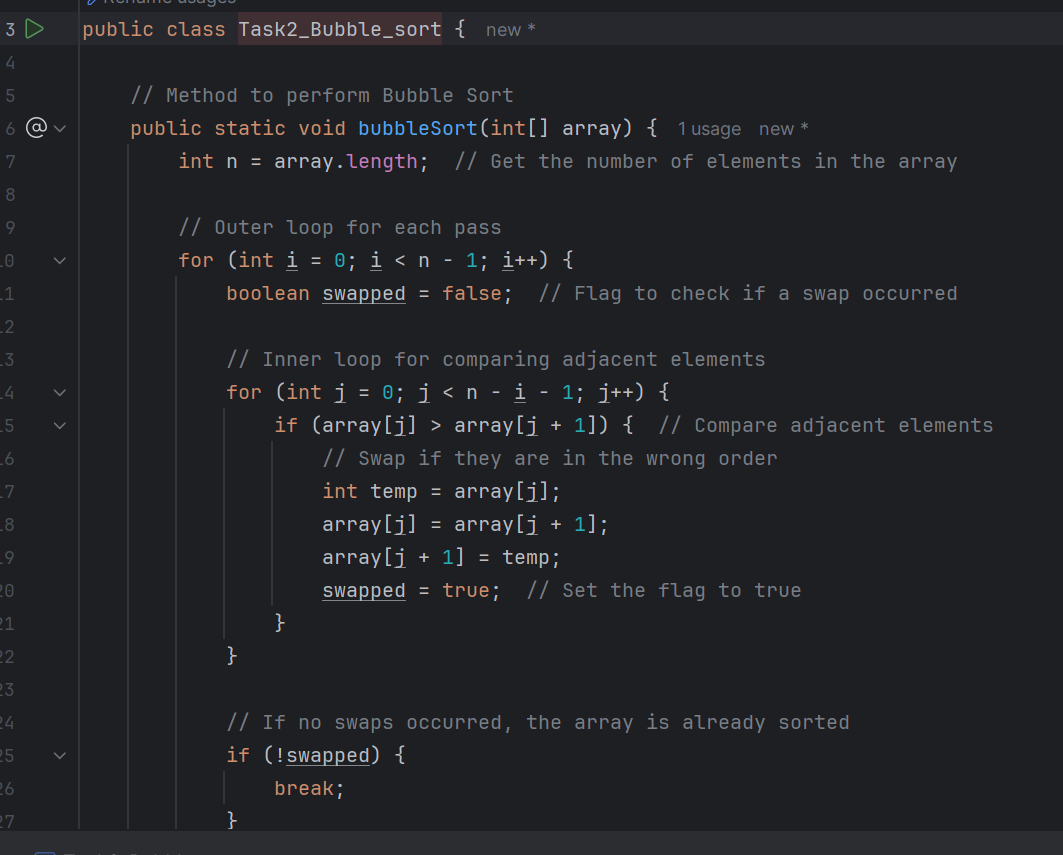
function main():

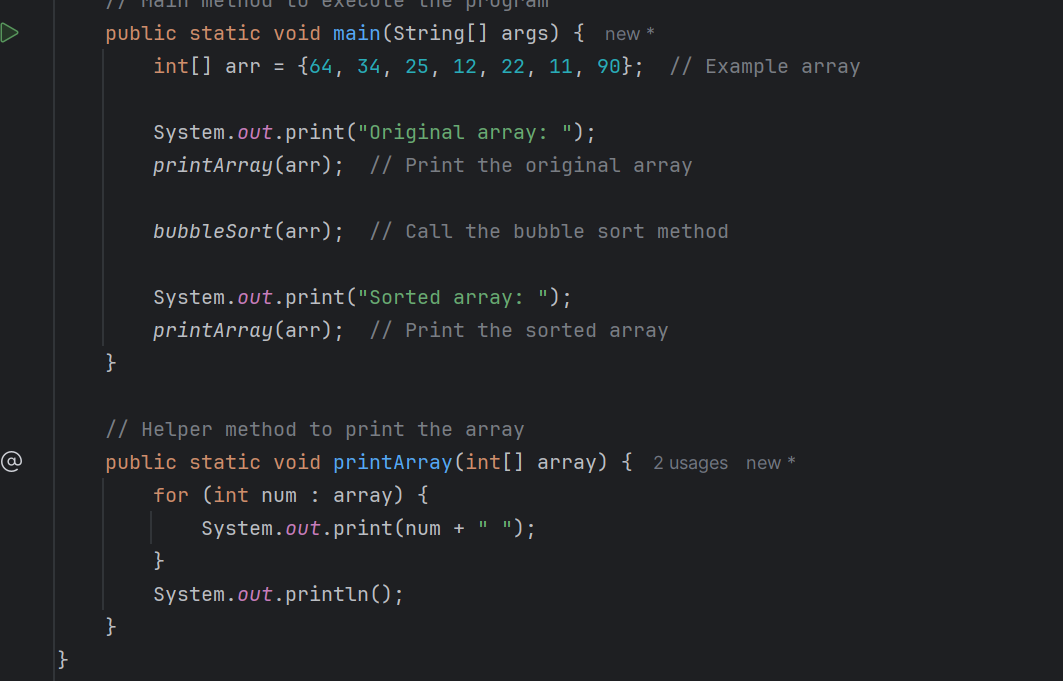
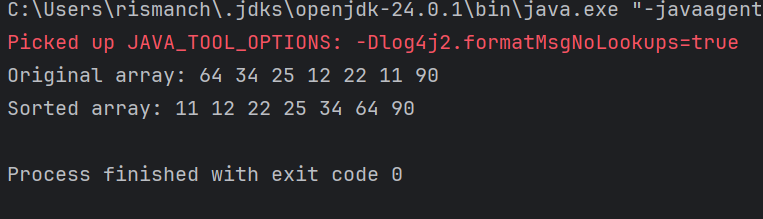
array = [64, 34, 25, 12, 22, 11, 90] // Example array

print("Original array:", array) // Print the original array

bubbleSort(array) // Call the bubble sort function

print("Sorted array:", array) // Print the sorted array

Task-6  


  
  
  
  
  
Task-7  
  
Write an algorithm for the Insertion sort.

1. insertionSort Method:
   * This method takes an array as input and sorts it using the Insertion Sort algorithm.
   * It iterates through the array starting from the second element (index 1).
   * For each element (referred to as key), it compares it with the elements in the sorted portion of the array (to its left) and shifts larger elements to the right to make space for the key.
   * Once the correct position for the key is found, it is inserted into that position.
2. main Method:
   * Initializes an example array {64, 34, 25, 12, 22, 11, 90}.
   * Prints the original array.
   * Calls the insertionSort method to sort the array.
   * Prints the sorted array.
3. printArray Helper Method:
   * This method prints the elements of the array in a readable format.

Task-8  
  
function insertionSort(array):

n = length(array) // Get the number of elements in the array

// Iterate through each element starting from the second element

for i from 1 to n-1:

key = array[i] // The element to be inserted

j = i - 1 // Index of the last sorted element

// Move elements of array[0..i-1], that are greater than key,

// to one position ahead of their current position

while j >= 0 and array[j] > key:

array[j + 1] = array[j] // Shift element to the right

j = j - 1 // Move to the next element

array[j + 1] = key // Insert the key at the correct position

// Main function to execute the program

function main():

array = [64, 34, 25, 12, 22, 11, 90] // Example array

print("Original array:", array) // Print the original array

insertionSort(array) // Call the insertion sort function

print("Sorted array:", array) // Print the sorted array

Task-12   
  
Algo of merge sort  
  
**Divide Phase:**

- Find middle point to divide array into two halves

- Create two temporary arrays for left and right halves

**Conquer Phase (Recursive):**

- Recursively sort first half

- Recursively sort second half

**Merge Phase:**

- Create pointers i, j, k for left, right, and main arrays

- Compare elements from both halves

- Place smaller element in main array

- Move respective pointers forward

- Copy remaining elements if any  
  
  
Task-13  
  
Pseudo code   
  
ALGORITHM MergeSort(arr)

IF length of arr <= 1 THEN

RETURN arr

END IF

mid = length of arr / 2

left = arr[0 to mid-1] // First half

right = arr[mid to end] // Second half

MergeSort(left)

MergeSort(right)

Merge(arr, left, right)

END ALGORITHM

ALGORITHM Merge(arr, left, right)

i = j = k = 0

WHILE i < length(left) AND j < length(right)

IF left[i] <= right[j] THEN

arr[k] = left[i]

i++

ELSE

arr[k] = right[j]

j++

END IF

k++

END WHILE

// Copy remaining elements

ADD remaining elements from left

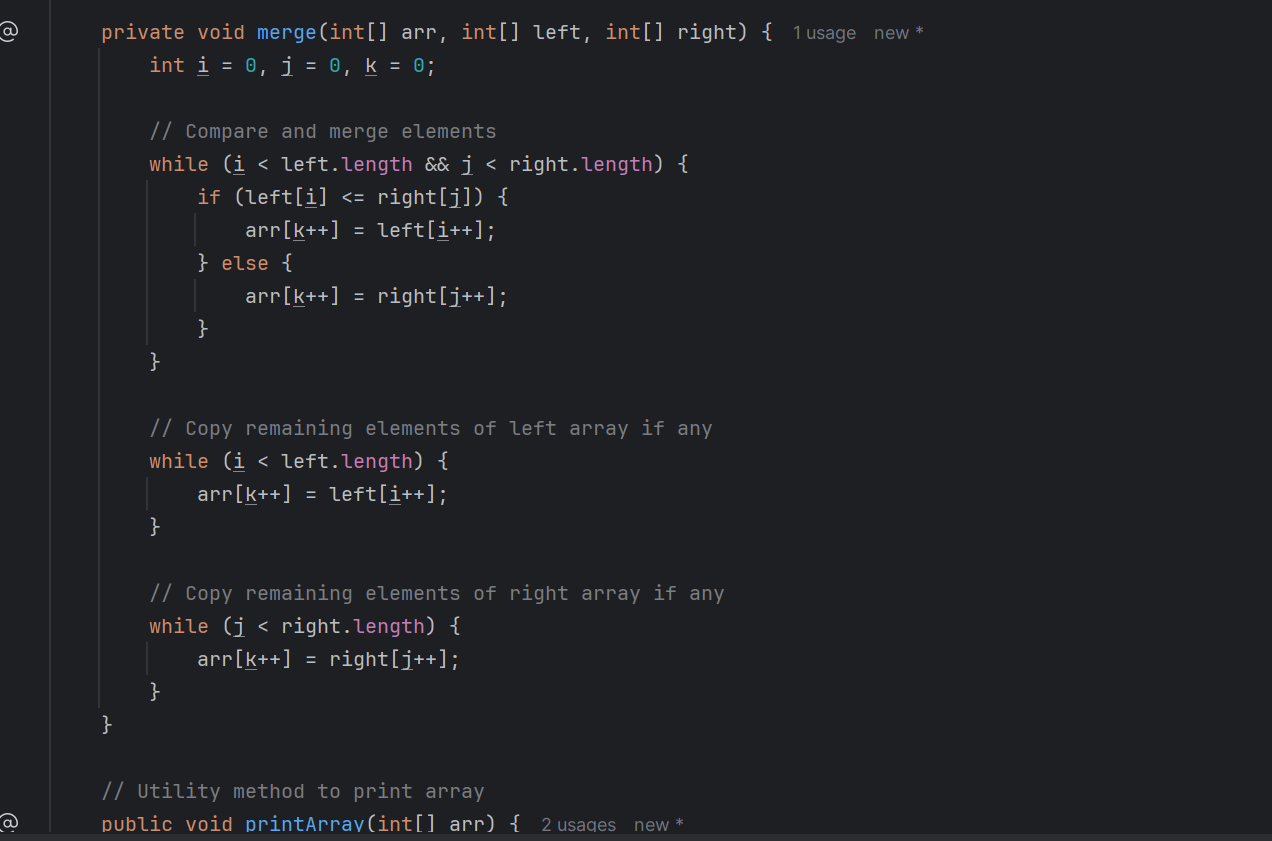
ADD remaining elements from right

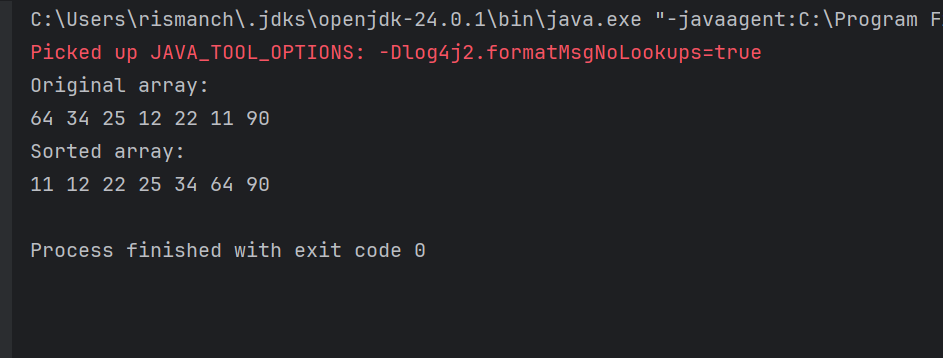
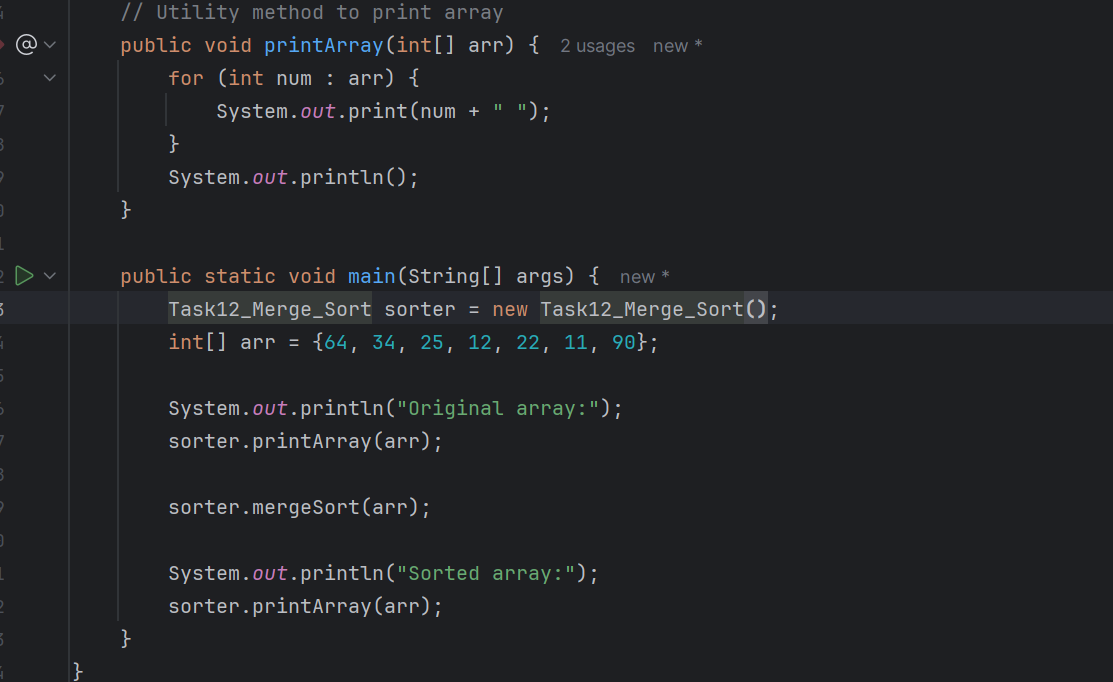
END ALGORITHM

TASK 14

code for Merge sort







Task 15:

Algo for quick sort

The efficiency of QuickSort depends heavily on pivot selection. Common strategies:

* Last element (shown above)
* First element
* Random element
* Median of three

Task-16  
  
Pseudo code   
  
ALGORITHM QuickSort(arr, low, high)

IF low < high THEN

pivot\_index = Partition(arr, low, high)

QuickSort(arr, low, pivot\_index - 1)

QuickSort(arr, pivot\_index + 1, high)

END IF

END ALGORITHM

ALGORITHM Partition(arr, low, high)

pivot = arr[high]

i = low - 1

FOR j = low to high - 1

IF arr[j] < pivot THEN

i++

swap arr[i] with arr[j]

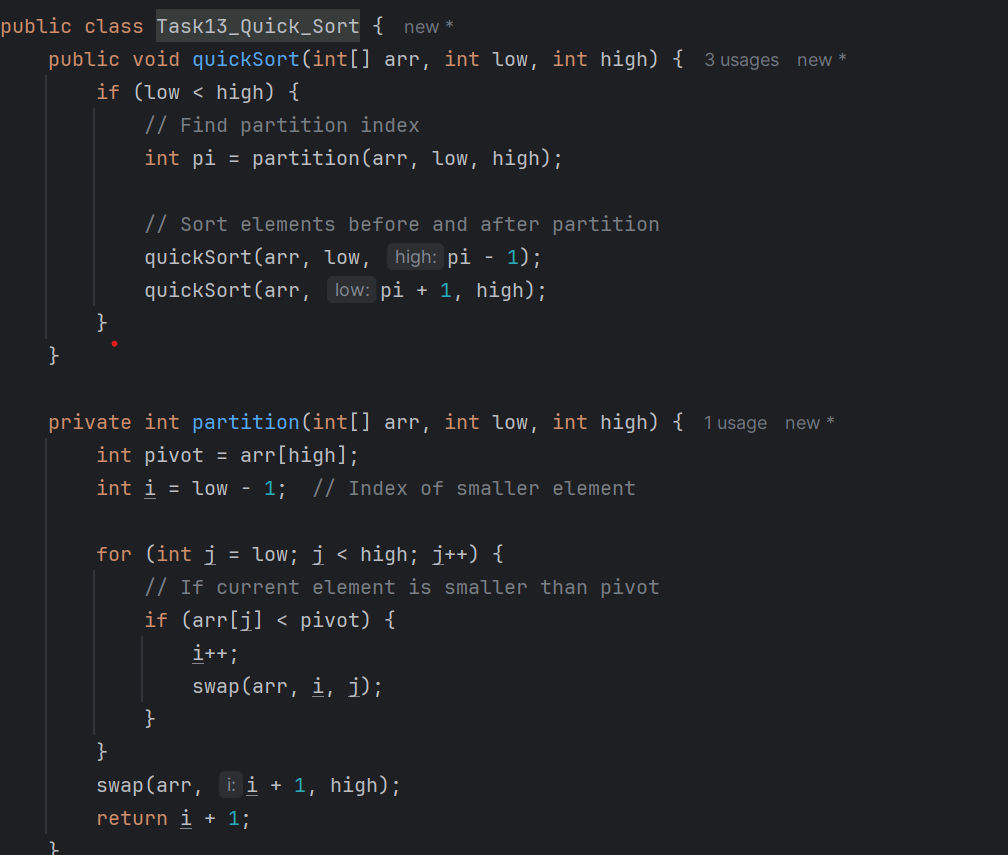
END IF

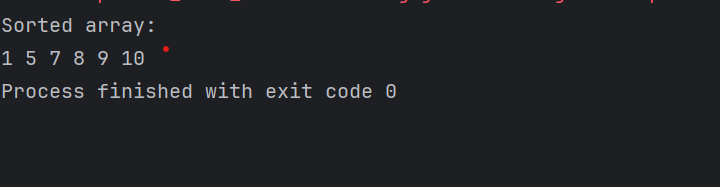
END FOR

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

RETURN i + 1

END ALGORITHM

Task-17  
  


Task 15

Step 1: Take Array, low index and high index and find pivot Index first to sort the elements.

Step 2: create a partition method to find the partition index. Take Array, low index and high index.

Step 3: store pivot as first element.

Step 4: take 2 pointers as low and high to compare the elements. Low = 0, high = n-1;

Step 5: we will move from left to right till elements of the array are small or equal to the pivot.

Step 6: we will move from right to left with j pointer till elements of the array are greater than pivot.

Step 7: remaining elements will be swapped. Arr[i] and Arr[j].

Step 8: Outside this while loop, we can swap the jth element with first element to ensure everything on the left is smaller and everything on the right is greater than jth element.

Step 9: we will create a recursive method while low<high.

Step 10: the array will be divided basis on the low to pIndex -1 and pIndex+1 to high;

Task 16

quicksort(arr, low index, high index){

if(low<high){

Int partitionIndex = partition(arr, low, high)

quicksort(arr, low, partitionIndex-1)

quicksort(arr, partitionIndex+1)

}

}

partition(arr, low, high){

Pivot = arr[low]

I =low;

J = high

while(i<j){

while(arr[i]<=pivot && i<=high-1){

i++;

} while(arr[j]>pivot && j>low+1){

j–;

} if(i<j){

swap(arr[i],arr[j])

}

swap(arr[j], arr[low])

Return j;

}

}

**What is the difference between binary tree and binary search tree (bst)**

**In sorted array why do you think binary search tree is better than linear search?**

Linear Search Starts from the beginning of the array and checks each element one by one and Time Complexity: O(n) — worst case, you may have to look at every element.

While If you create a BST from a sorted array, and if it's balanced, you can Start from the root. At each step, eliminate half of the remaining nodes. Time Complexity: O(log n) (if tree is balanced).

**Difference between static and dynamic arrays**

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | **Fixed at compile time** | **Can grow or shrink during runtime** |
| **Memory Allocation** | **Allocated at compile time (stack/heap)** | **Allocated at runtime (heap)** |
| **Resize Possible?** | **No (size cannot be changed)** | **Yes (size can be changed dynamically)** |
| **Performance** | **Faster (no overhead of resizing)** | **Slightly slower (due to resizing, copying)** |
| **Memory Usage** | **Efficient, but inflexible** | **May use extra memory (over-allocation)** |
| **Ease of Use** | **Simple but limited** | **More flexible and powerful** |
| **Example in Java** | **int[] arr = new int[10];** | **ArrayList<Integer> arr = new ArrayList<>();** |

Task 11  
  
