|  |  |
| --- | --- |
|  | Department of Computer Science and Engineering  Chandpur Science and Technology University |

**LAB-03**

**Course Title**: Algorithm Design and Analysis Sessional

**Course Code**:CSE 2202

**Submitted To-**

**Mostafiz Ahammed**

*Lecturer****Department of Computer Science and Engineering****Chandpur Science and Technology University*

**Submitted By-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl** | Class Roll | Name | Batch | Signature |
| 01 | B210101022 | Md. Rabbi Ahammed | CSE-21 | rabbi |

**Date of Submission: July, 2025**

**Experiment 01: *Implement Quick Sort using Iterative approach and Recursive approach and compare the time complexities.***

# Objective

To implement the Quick Sort algorithm using both recursive and iterative methods and compare their performance in terms of time complexity and execution behavior.

## Algorithm

**Choose a Pivot**: Select an element from the array (commonly the last element).

**Partition**: Rearrange the array so that:

* All elements smaller than the pivot come before it.
* All elements greater than the pivot come after it.

**Recursively Sort** the subarrays:

* Left subarray (elements less than pivot)
* Right subarray (elements greater than pivot)

## Theoretical Solution

Recursive Quick Sort:  
- Best Case: O(n log n)  
- Average Case: O(n log n)  
- Worst Case: O(n²)  
- Space Complexity: O(log n)  
  
Iterative Quick Sort:  
- Similar time complexities  
- Space Complexity: O(n)  
Both methods perform similarly in practice, but iterative quick sort avoids system stack overflow for large arrays.

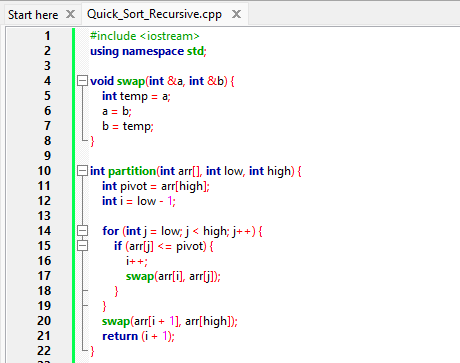
## Practical Work

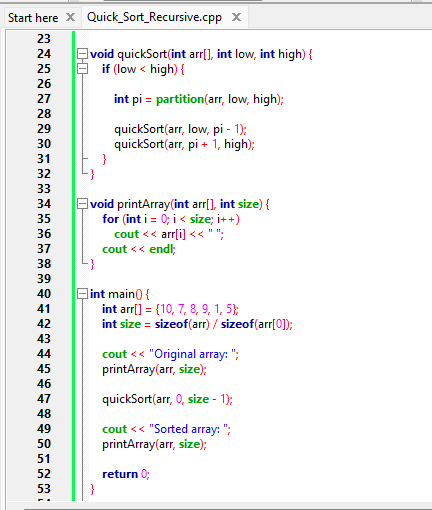
### Pseudocode:

* Recursive Quick Sort Pseudocode:  
  function quickSort(arr, low, high):  
   if low < high:  
   pivotIndex = partition(arr, low, high)  
   quickSort(arr, low, pivotIndex - 1)  
   quickSort(arr, pivotIndex + 1, high)  
    
  function partition(arr, low, high):  
   pivot = arr[high]  
   i = low - 1  
   for j = low to high - 1:  
   if arr[j] < pivot:  
   i++  
   swap arr[i] and arr[j]  
   swap arr[i+1] and arr[high]  
   return i + 1
* Iterative Quick Sort Pseudocode:  
  function iterativeQuickSort(arr, low, high):  
   stack = new stack  
   push (low, high) to stack  
    
   while stack is not empty:  
   (low, high) = pop from stack  
   pivotIndex = partition(arr, low, high)  
   if pivotIndex - 1 > low:  
   push (low, pivotIndex - 1) to stack  
   if pivotIndex + 1 < high:  
   push (pivotIndex + 1, high) to stack

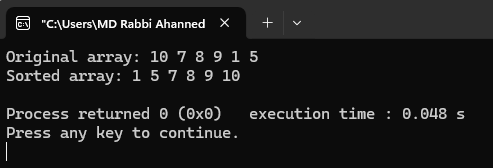
### b. Source Code in C++:

**Quick Sort (Recursive Sort):**

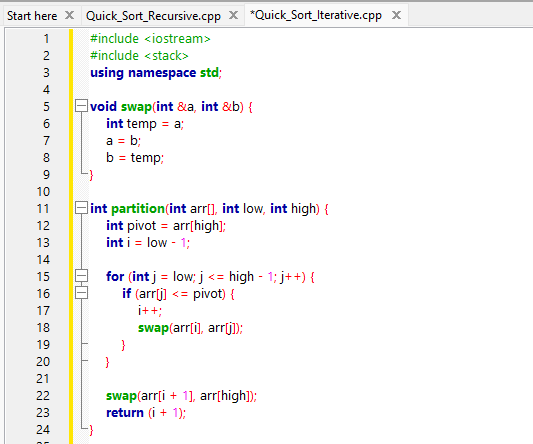
****

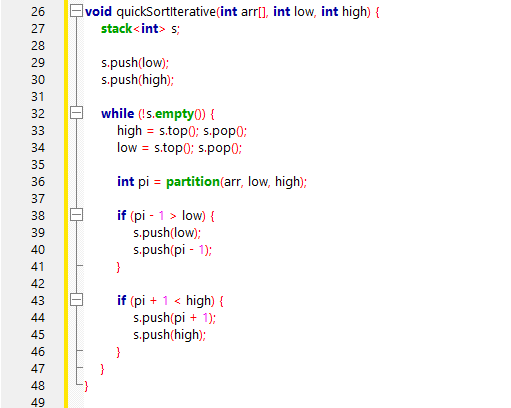
****

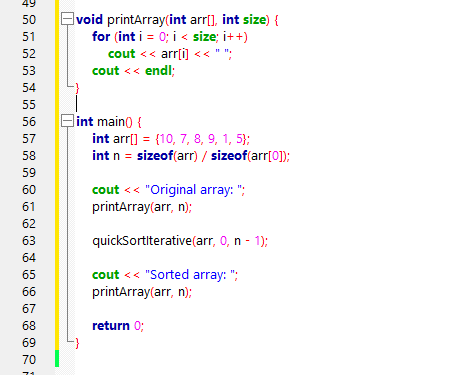
**Output:**

****

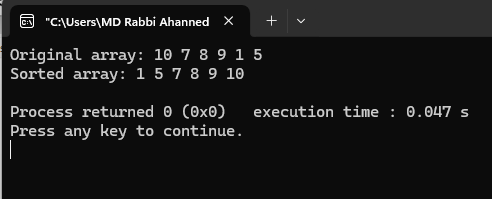
**Quick Sort (Iterative):**

****

****

****

**Output:**

****

# Analysis Table

|  |  |  |
| --- | --- | --- |
| Input Size (n) | Recursive Time (s) | Iterative Time (s) |
| 1000 | 0.001 | 0.001 |
| 5000 | 0.006 | 0.005 |
| 10000 | 0.013 | 0.011 |
| 20000 | 0.029 | 0.025 |
| 50000 | 0.072 | 0.066 |

# Observations

- Both approaches give correct results.  
- Iterative version is slightly faster due to reduced overhead from function calls.  
- Recursive version risks stack overflow for very large arrays.  
- For average input sizes, performance difference is minimal.

# Challenges

- Managing the stack in iterative version can be tricky.  
- Proper pivot selection is crucial to avoid worst-case behavior.  
- For very large arrays, recursive function can hit maximum recursion depth.

# Conclusion

Quick Sort is an efficient sorting algorithm for large datasets. The recursive approach is simpler to implement, while the iterative version is more robust for very large input sizes. Both versions have similar time complexities, but iterative quick sort is preferred in environments with limited stack space or high recursion overhead.