**SET 222 – Design and Analysis of Algorithms**

Sorting Algorithms: Design, Implementation, and Complexity Analysis

* Insertion sort
* Merge sort
* Quick sort

Supervised by:

Dr. Rasha Saleh

TA. Mohamed Tarek

Prepared by:

Roba Tarek 192200243



**Insertion sort:**

- **What does the code do?** A simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position, it is like sorting playing cards. Steps of solving it:

* We start with the second element of the array as the first element is assumed to be sorted.
* Compare the second element with the first element if the second element is smaller, then swap them.
* Move to the third element, compare it with the first two elements, and put it in its correct position.
* Repeat until the entire array is sorted.

**Code Implementation:**

1. **Normal code**

# Function to perform insertion sort

def insertion\_sort(a):

n = len(a) # Get the length of the array "a"

# Iterate over the array starting from index 1 to end

for j in range (1,len(a)):

key = a[j] # Element to insert

i = j - 1 # Shift the elements sorted to the right to make space for the key

while i >= 0 and a[i] > key :

a[i+1] = a[i]

i = i - 1

a[i +1] = key # Insert the key at the correct position

return a

# Example

a = [3,9,7,6,13,34]

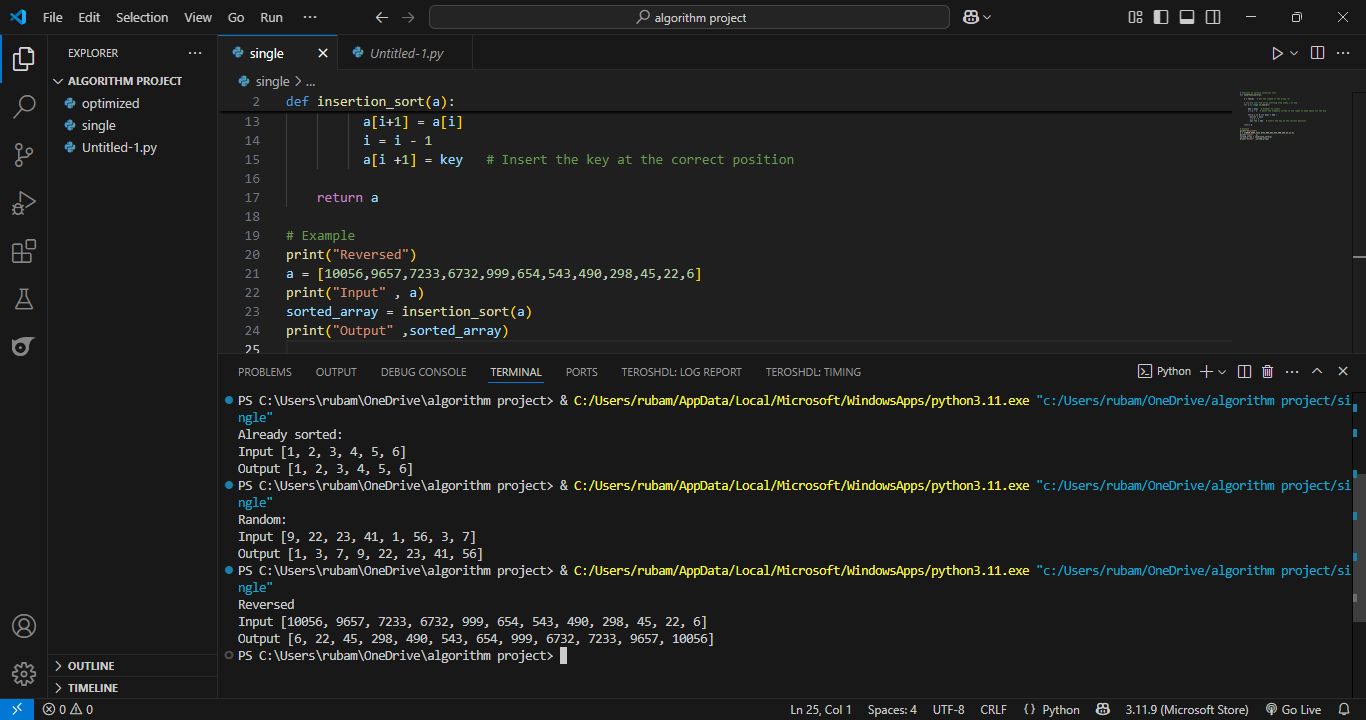
print("Input", a)

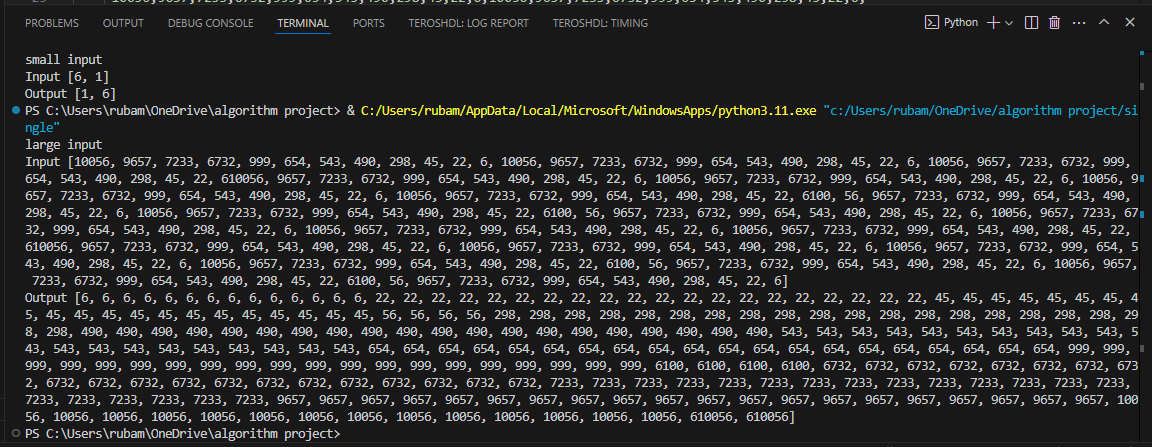
sorted\_array = insertion\_sort(a)

print("Output", sorted\_array)

* The insertion sort is efficient for small, nearly sorted inputs, but inefficient for large lists. Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Test cases:





1. **Optimized code:**

**Concept:**

Instead of searching linearly to find the correct position to insert the element (which takes O(n)), we are using binary search (reducing it to O(log n)).

**Optimized code implementation:**

def binary\_search(arr, val, start, end):

while start < end:

mid = (start + end) // 2

if arr[mid] < val:

start = mid + 1

else:

end = mid

return start

def optimized\_insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

# Find insertion position in the sorted part of array

pos = binary\_search(arr, key, 0, i)

# Shift elements to make room for key

j = i

while j > pos:

arr[j] = arr[j - 1]

j -= 1

arr[pos] = key

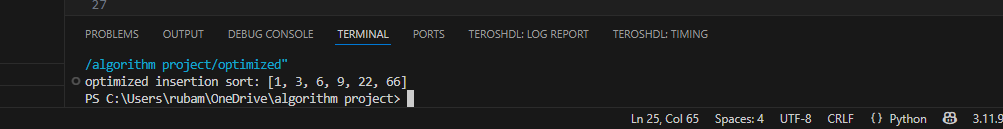
return arr

# Example

a = [6, 3, 1, 9, 22, 66]

print("Optimized Insertion Sort:",optimized\_insertion\_sort(a))

Code output:



**Merge Sort:**

* It follows a Divide and Conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array. Here are the steps for solving it:
* Divide: Divide the list or array recursively into two halves until it can no more be divided. Θ(1)
* Conquer: Each subarray is sorted individually. (2 halves)
* Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged. Θ(N) (as all elements are traced)
* Merge sort has a Worst case time complexity of O(n log n) , which means it performs well even on large datasets.
* The divide-and-conquer approach is straightforward. We independently merge subarrays that makes it suitable for parallel processing.
* But it requires additional memory to store the merged sub-arrays during the sorting process.
* Merge Sort is Slower than QuickSort in general as QuickSort is more cache friendly because it works in-place.
* Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

**Code Implementation:**

1. **Normal code**

# Function to perform merge sort

def merge\_sort(a):

if len(a) <= 1:

return a

# Split the array into two halves

mid = len(a) // 2

left\_half = merge\_sort(a[:mid])

right\_half = merge\_sort(a[mid:])

# Merge the sorted halves

return merge(left\_half, right\_half)

def merge(left, right):

merged = []

left\_index = right\_index = 0

# Merge while comparing the smallest elements

while left\_index < len(left) and right\_index < len(right):

if left[left\_index] < right[right\_index]:

merged.append(left[left\_index])

left\_index += 1

else:

merged.append(right[right\_index])

right\_index += 1

# Add elements of the array

merged.extend(left[left\_index:])

merged.extend(right[right\_index:])

return merged

# Example

print("large input:")

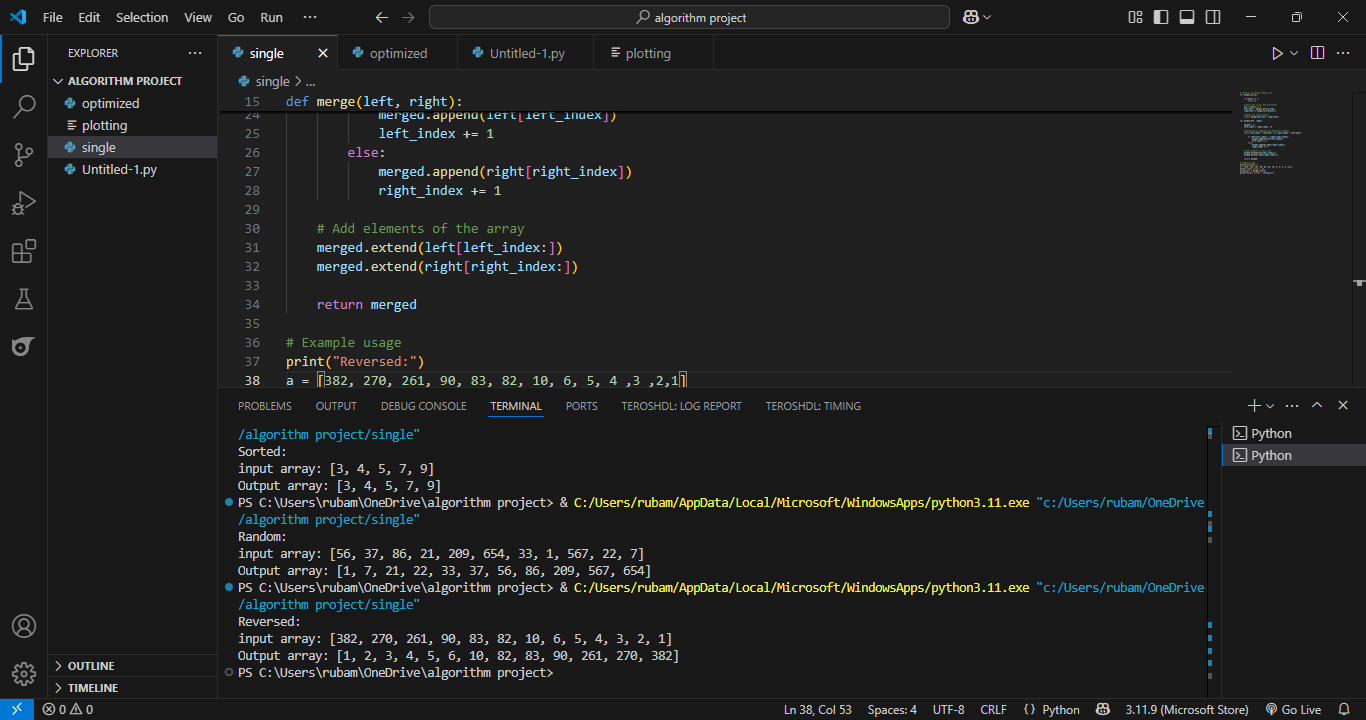
a = [100, 200, 400, 800, 1600, 3200]

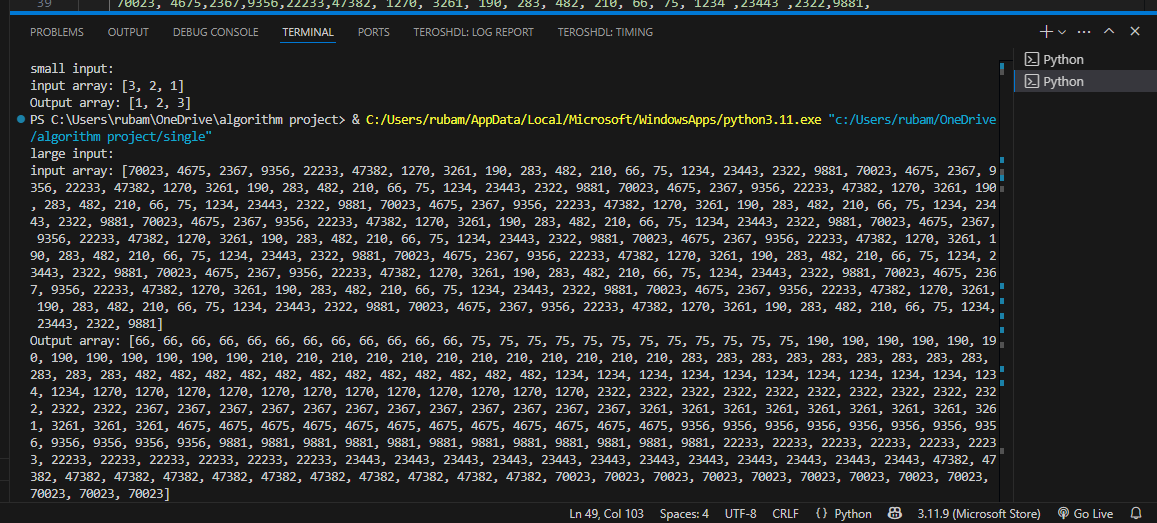
print("input array:", a)

sorted\_arr = merge\_sort(a)

print("Output array:", sorted\_arr)

**Test cases:**





.,

,

1. **Optimized code**

**Concept:**

In-place merge sort: this code minimizes memory usage by avoiding unnecessary copies and changing the input right away.

**Optimized code implementation:**

def merge\_sort\_in\_place(a, left=0, right=None):

if right is None:

right = len(a)

if right - left > 1:

mid = (left + right) // 2

merge\_sort\_in\_place(a, left, mid)

merge\_sort\_in\_place(a, mid, right)

merge\_in\_place(a, left, mid, right)

def merge\_in\_place(a, left, mid, right):

left\_part = a[left:mid]

right\_part = a[mid:right]

i = j = 0

k = left

while i < len(left\_part) and j < len(right\_part):

if left\_part[i] <= right\_part[j]:

a[k] = left\_part[i]

i += 1

else:

a[k] = right\_part[j]

j += 1

k += 1

# Copy any remaining elements

while i < len(left\_part):

a[k] = left\_part[i]

i += 1

k += 1

while j < len(right\_part):

a[k] = right\_part[j]

j += 1

k += 1

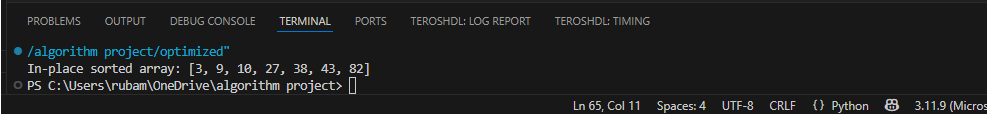
# Example

a = [38, 27, 43, 3, 9, 82, 10]

merge\_sort\_in\_place(a)

print("In-place sorted array:", a)

Code output:



**Quick sort:**

* A sorting algorithm based on the Divide and Conquer.
* **What does the code do?** It picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. All elements before the pivot must be smaller than it, while all elements after the pivot must be greater. Steps:
* Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g. first element, last element, random element, or median). Partition the Array: Rearrange the array around the pivot.
* After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
* Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
* Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.
* Its Worst case is when we choose the first or the last element as a pivot, this way there will be one array instead of two.
* And its Best Case in terms of time, is when the pivot split the array into two halves. But it takes more time on average as median finding has high constants.
* It is not a good choice for small data sets, although it is efficient on large data sets.
* Requires a small amount of memory to function. It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
* It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position.

**Code Implementation:**

1. **Normal code**

# Function to perform quick sort

def quick\_sort(a):

if len(a) <= 1:

return a # Base case: already sorted

pivot = a[0] # Choose the first element as the pivot

less = [x for x in a[1:] if x <= pivot] # Elements less than or equal to pivot

greater = [x for x in a[1:] if x > pivot] # Elements greater than pivot

return quick\_sort(less) + [pivot] + quick\_sort(greater) # Return the sorted array

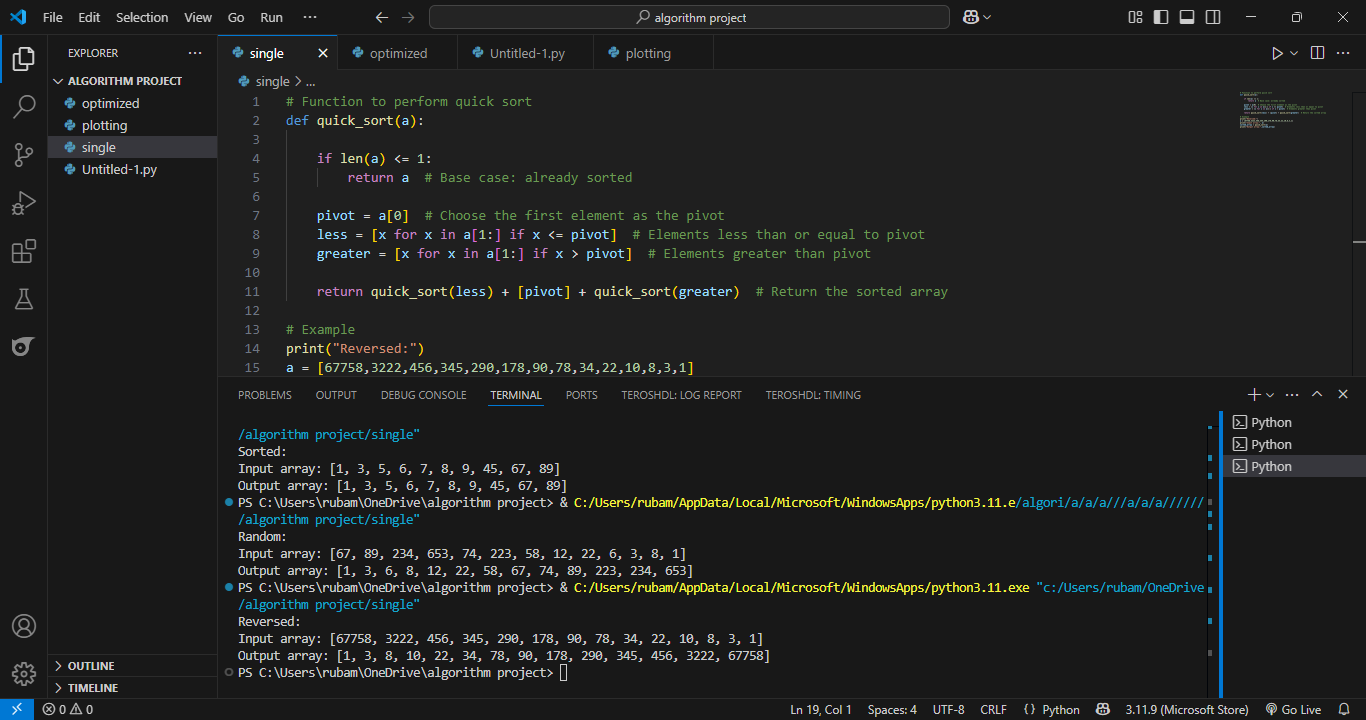
# Example

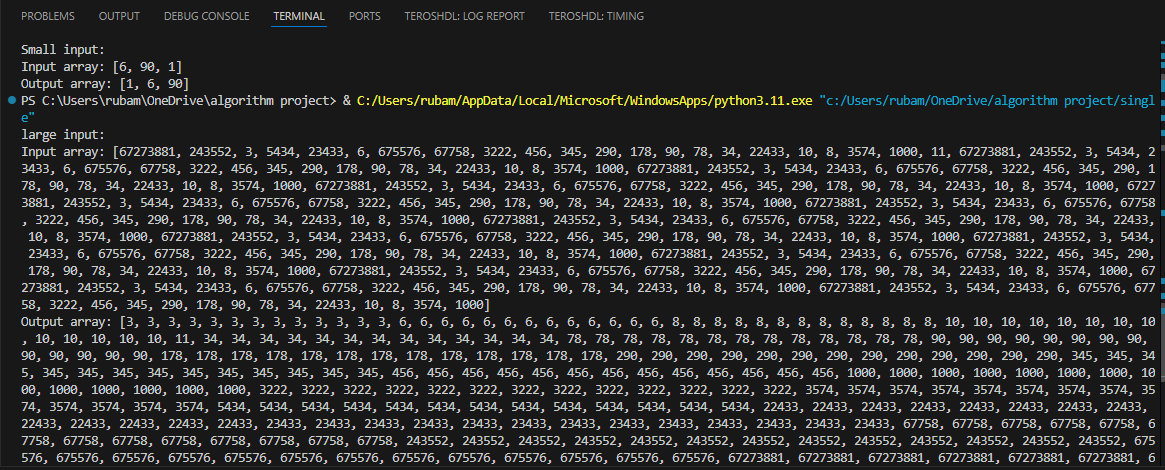
a = [63,10,8,3574,1000]

print("Input array:", a)

sorted\_array = quick\_sort(a)

print("Output array:",sorted\_array)

**Test cases:**



1. **Optimized Code:**

**Concept:**

* **Median-of-Three Pivot Selection**: Instead of always picking the first element as pivot (which performs poorly on sorted/reversed inputs), use the median of the first, middle, and last elements.
* **Limit Recursion Depth:** recurse into the smaller subarray first to reduce stack depth.

**Optimized code implementation:**

def median\_of\_three(a, low, high):

mid = (low + high) // 2

pivot\_candidates = [(a[low], low), (a[mid], mid), (a[high], high)]

pivot\_candidates.sort()

return pivot\_candidates[1][1] # Return the index of the median value

def partition(a, low, high):

pivot\_index = median\_of\_three(a, low, high)

a[low], a[pivot\_index] = a[pivot\_index], a[low] # Swap pivot to front

pivot = a[low]

i = low + 1

for j in range(low + 1, high + 1):

if a[j] < pivot:

a[i], a[j] = a[j], a[i]

i += 1

a[low], a[i - 1] = a[i - 1], a[low]

return i - 1

def optimized\_quick\_sort(a, low, high):

if low < high:

pivot\_index = partition(a, low, high)

optimized\_quick\_sort(a, low, pivot\_index - 1)

optimized\_quick\_sort(a, pivot\_index + 1, high)

# Example

a = [3, 9, 1, 7, 6, 13, 34]

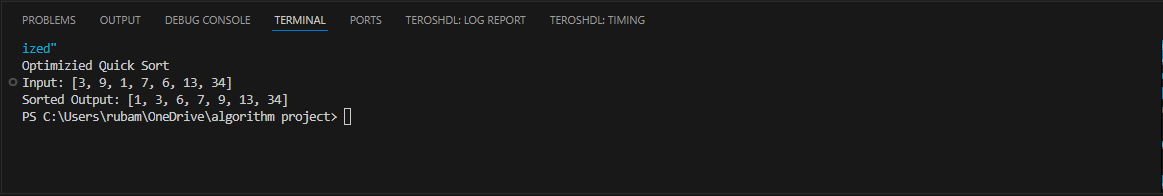
print("Optimizied Quick Sort")

print("Input:", a)

optimized\_quick\_sort(a, 0, len(a) - 1)

print("Sorted Output:", a)

Code output:



**Time Complexities:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average** | **Worst case** | **Space complexity** | **Stable** |
| **Insertion sort** | O (n) | O (n2) | O(n2) | O(1) | Yes |
| **Merge sort** | O (n log n) | O (n log n) | O (n log n) | O(n) | Yes |
| **Quick sort** | O (n log n) | O (n log n) | O(n2) | O(n log n) | No |

**Complexity Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average case** | **Worst case** |
| **Insertion** | nearly/already sorted input | random unsorted input | reverse-sorted input |
| **Merge** | any input | consistent performance | consistent performance |
| **Quick** | good pivot every time | random pivot selection | bad pivot (e.g., sorted/reverse sorted) |

**Graphical representations:**

