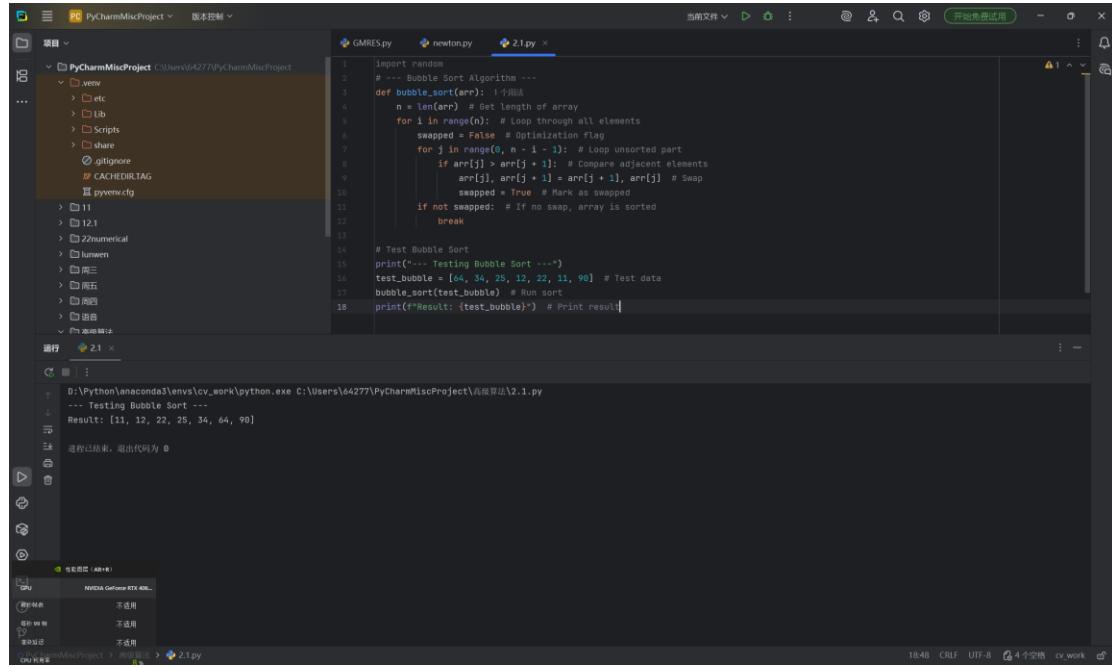


Problem 1 (Sorting Algorithms).

1. write own bad $O(n^2)$ sorting (bubble sorting)



```
import random
# --- Bubble Sort Algorithm ---
def bubble_sort(arr):
    n = len(arr) # Get length of array
    for i in range(n): # Loop through all elements
        swapped = False # Optimization flag
        for j in range(0, n - i - 1): # Loop unsorted part
            if arr[j] > arr[j + 1]: # Compare adjacent elements
                arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap
                swapped = True # Mark as swapped
        if not swapped: # If no swap, array is sorted
            break
    # Test Bubble Sort
    print("--- Testing Bubble Sort ---")
    test_bubble = [64, 34, 25, 12, 22, 11, 90] # Test data
    bubble_sort(test_bubble) # Run sort
    print(f'Result: {test_bubble}') # Print result
```

Code

```
import random
# --- Bubble Sort Algorithm ---
def bubble_sort(arr):
    n = len(arr) # Get length of array
    for i in range(n): # Loop through all elements
        swapped = False # Optimization flag
        for j in range(0, n - i - 1): # Loop unsorted part
            if arr[j] > arr[j + 1]: # Compare adjacent elements
                arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap
                swapped = True # Mark as swapped
        if not swapped: # If no swap, array is sorted
            break
    # Test Bubble Sort
    print("--- Testing Bubble Sort ---")
    test_bubble = [64, 34, 25, 12, 22, 11, 90] # Test data
    bubble_sort(test_bubble) # Run sort
    print(f'Result: {test_bubble}') # Print result
```

2. create few tests of various sizes and control that algo. is working

```

2.2.py
1 import random # Import random library
2 # --- Helper: Partition Function ---
3 def partition(arr, low, high):
4     pivot = arr[high] # Pivot is last element
5     i = low - 1 # Index for smaller element
6     for j in range(low, high): # Traverse array
7         if arr[j] < pivot: # If element < pivot
8             i += 1 # Move index
9             arr[i], arr[j] = arr[j], arr[i] # Swap
10            arr[i + 1], arr[high] = arr[high], arr[i + 1] # Place pivot
11    return i + 1 # Return partition index
12
13 # --- Quick Sort (Random Pivot) ---
14 def quick_sort_random(arr, low, high):
15     if low < high: # Base case
16         # Random Pivot Logic
17         rand_idx = random.randint(low, high) # Pick random index
18         arr[rand_idx], arr[high] = arr[high], arr[rand_idx] # Swap with end
19
20         pi = partition(arr, low, high) # Partition
21         quick_sort_random(arr, low, pi - 1) # Sort left
22         quick_sort_random(arr, pi + 1, high) # Sort right
23
24
25 # --- Test Quick Sort (Random) ---
26 print("\n--- Testing Quick Sort (Random Pivot) ---")
27 test_qr = [10, 7, 8, 9, 1, 5] # Test data
28 quick_sort_random(test_qr, 0, len(test_qr) - 1) # Run sort
29 print(f'Result: {test_qr}') # Print result
30

```

Code

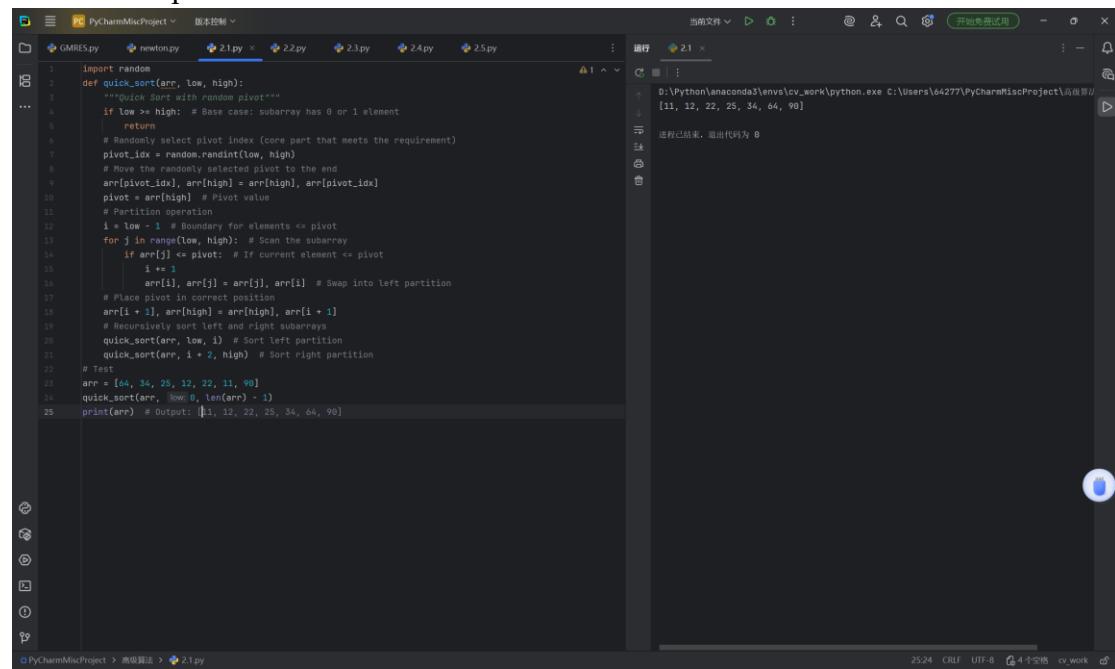
```

import random # Import random library
# --- Helper: Partition Function ---
def partition(arr, low, high):
    pivot = arr[high] # Pivot is last element
    i = low - 1 # Index for smaller element
    for j in range(low, high): # Traverse array
        if arr[j] < pivot: # If element < pivot
            i += 1 # Move index
            arr[i], arr[j] = arr[j], arr[i] # Swap
    arr[i + 1], arr[high] = arr[high], arr[i + 1] # Place pivot
    return i + 1 # Return partition index
# --- Quick Sort (Random Pivot) ---
def quick_sort_random(arr, low, high):
    if low < high: # Base case
        # Random Pivot Logic
        rand_idx = random.randint(low, high) # Pick random index
        arr[rand_idx], arr[high] = arr[high], arr[rand_idx] # Swap with end
        pi = partition(arr, low, high) # Partition
        quick_sort_random(arr, low, pi - 1) # Sort left
        quick_sort_random(arr, pi + 1, high) # Sort right
# --- Test Quick Sort (Random) ---
print("\n--- Testing Quick Sort (Random Pivot) ---")
test_qr = [10, 7, 8, 9, 1, 5] # Test data
quick_sort_random(test_qr, 0, len(test_qr) - 1) # Run sort
print(f'Result: {test_qr}') # Print result

```

3. write own Quick Sort

add random pivot



The screenshot shows the PyCharm IDE interface with the code for quick sort. The code is in a file named 2.1.py. The code implements a quick sort algorithm with a random pivot. It includes a test section at the bottom with an array [64, 34, 25, 12, 22, 11, 90] and its sorted output [11, 12, 22, 25, 34, 64, 90]. The PyCharm interface shows the code in the editor, the run configuration in the top bar, and the terminal output in the bottom right.

```
import random
def quick_sort(arr, low, high):
    """Quick Sort with random pivot"""
    if low >= high: # Base case: subarray has 0 or 1 element
        return
    # Randomly select pivot index (core part that meets the requirement)
    pivot_idx = random.randint(low, high)
    # Move the randomly selected pivot to the end
    arr[pivot_idx], arr[high] = arr[high], arr[pivot_idx]
    pivot = arr[high] # Pivot value
    # Partition operation
    i = low - 1 # Boundary for elements <= pivot
    for j in range(low, high): # Scan the subarray
        if arr[j] <= pivot: # If current element <= pivot
            i += 1
            arr[i], arr[j] = arr[j], arr[i] # Swap into left partition
    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    # Recursively sort left and right subarrays
    quick_sort(arr, low, i) # Sort left partition
    quick_sort(arr, i + 2, high) # Sort right partition
# Test
arr = [64, 34, 25, 12, 22, 11, 90]
quick_sort(arr, 0, len(arr) - 1)
print(arr) # Output: [11, 12, 22, 25, 34, 64, 90]
```

Code

```
import random
def quick_sort(arr, low, high):
    if low >= high: # Base case: subarray has 0 or 1 element
        return
    # Randomly select pivot index (core part that meets the requirement)
    pivot_idx = random.randint(low, high)
    # Move the randomly selected pivot to the end
    arr[pivot_idx], arr[high] = arr[high], arr[pivot_idx]
    pivot = arr[high] # Pivot value
    # Partition operation
    i = low - 1 # Boundary for elements <= pivot
    for j in range(low, high): # Scan the subarray
        if arr[j] <= pivot: # If current element <= pivot
            i += 1
            arr[i], arr[j] = arr[j], arr[i] # Swap into left partition
    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    # Recursively sort left and right subarrays
    quick_sort(arr, low, i) # Sort left partition
    quick_sort(arr, i + 2, high) # Sort right partition
# Test
arr = [64, 34, 25, 12, 22, 11, 90]
quick_sort(arr, 0, len(arr) - 1)
print(arr) # Output: [11, 12, 22, 25, 34, 64, 90]
```

add average pivot (down + middle + up)

The screenshot shows the PyCharm IDE interface. The left pane displays a project structure for 'PyCharmMiscProject' containing several files like GMRES.py, newton.py, 2.1.py, etc. The right pane shows the code editor with the file '2.3.py' open. The code implements a quick sort algorithm using the median-of-three pivot selection method. It includes functions for quick sort, median of three, and a test function. The status bar at the bottom indicates the file is 351B, uses CRLF line endings, and is in UTF-8 encoding.

```
def quick_sort_median_pivot(arr, low, high):
    """Quick Sort using median of three as pivot"""
    if low >= high: # Base case: subarray has 0 or 1 element
        return
    # Get indices for three positions: down, middle, up
    mid = (low + high) // 2
    # Find median of three and use it as pivot index
    pivot_idx = median_of_three(arr, low, mid, high)
    # Move the selected pivot to the end
    arr[pivot_idx], arr[high] = arr[high], arr[pivot_idx]
    pivot = arr[high] # Pivot value
    # Partition operation
    i = low - 1 # Boundary for elements <= pivot
    for j in range(low, high): # Scan the subarray
        if arr[j] <= pivot: # If current element <= pivot
            i += 1
            arr[i], arr[j] = arr[j], arr[i] # Swap into left partition
    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    # Recursively sort left and right subarrays
    quick_sort_median_pivot(arr, low, i) # Sort left partition
    quick_sort_median_pivot(arr, i + 2, high) # Sort right partition

def median_of_three(arr, low, mid, high):
    """Find index of median value among three elements"""
    a, b, c = arr[low], arr[mid], arr[high]
    # Compare three values to find median
    if (a < b < c) or (c < a < b):
        return mid # b is median
    elif (b < a < c) or (c < b < a):
        return low # a is median
    else:
        return high # c is median

# Test
arr = [64, 34, 25, 12, 22, 11, 90]
quick_sort_median_pivot(arr, low=0, len(arr) - 1)
print(arr) # Output: [11, 12, 22, 25, 34, 64, 90]
```

Code

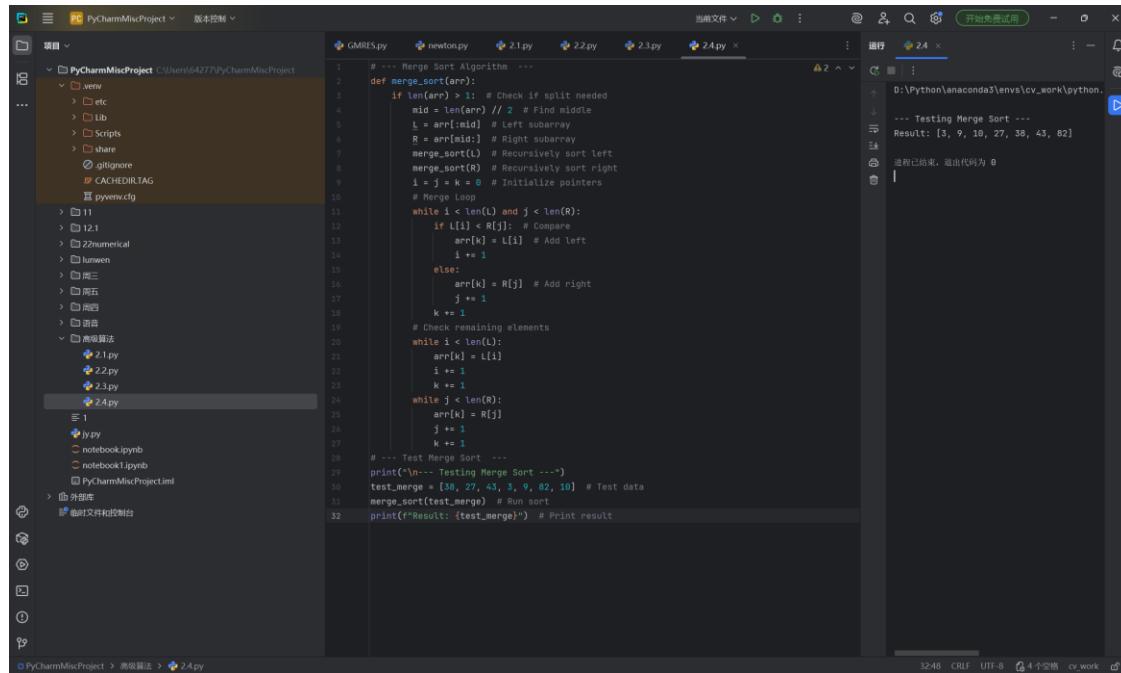
```
def quick_sort_median_pivot(arr, low, high):
    if low >= high: # Base case: subarray has 0 or 1 element
        return
    # Get indices for three positions: down, middle, up
    mid = (low + high) // 2
    # Find median of three and use it as pivot index
    pivot_idx = median_of_three(arr, low, mid, high)
    # Move the selected pivot to the end
    arr[pivot_idx], arr[high] = arr[high], arr[pivot_idx]
    pivot = arr[high] # Pivot value
    # Partition operation
    i = low - 1 # Boundary for elements <= pivot
    for j in range(low, high): # Scan the subarray
        if arr[j] <= pivot: # If current element <= pivot
            i += 1
            arr[i], arr[j] = arr[j], arr[i] # Swap into left partition
    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    # Recursively sort left and right subarrays
    quick_sort_median_pivot(arr, low, i) # Sort left partition
    quick_sort_median_pivot(arr, i + 2, high) # Sort right partition

def median_of_three(arr, low, mid, high):
    """Find index of median value among three elements"""
    a, b, c = arr[low], arr[mid], arr[high]
    # Compare three values to find median
```

```
if (a <= b <= c) or (c <= b <= a):
    return mid  # b is median
elif (b <= a <= c) or (c <= a <= b):
    return low  # a is median
else:
    return high  # c is median

# Test
arr = [64, 34, 25, 12, 22, 11, 90]
quick_sort_median_pivot(arr, 0, len(arr) - 1)
print(arr)  # Output: [11, 12, 22, 25, 34, 64, 90]
```

4. write own Merge Sort



The screenshot shows the PyCharm IDE interface. On the left is the project structure for 'PyCharmMiscProject' containing files like GMRES.py, newton.py, 2.1.py, 2.2.py, 2.3.py, and 2.4.py. The 2.4.py file is open in the center editor window, displaying the Merge Sort algorithm. To the right is the terminal window showing the output of running the script with test data [30, 27, 43, 3, 9, 82, 10]. The status bar at the bottom indicates the file has 324B, CRLF, UFF-8 encoding, and 4个文件 (4 files) are open.

```
1 # --- Merge Sort Algorithm ---  
2  
3 def merge_sort(arr):  
4     if len(arr) > 1: # Check if split needed  
5         mid = len(arr) // 2 # Find middle  
6         L = arr[:mid] # Left subarray  
7         R = arr[mid:] # Right subarray  
8         merge_sort(L) # Recursively sort left  
9         merge_sort(R) # Recursively sort right  
10        i = j = k = 0 # Initialize pointers  
11        # Merge Loop  
12        while i < len(L) and j < len(R):  
13            if L[i] < R[j]: # Compare  
14                arr[k] = L[i] # Add left  
15                i += 1  
16            else:  
17                arr[k] = R[j] # Add right  
18                j += 1  
19        # Check remaining elements  
20        while i < len(L):  
21            arr[k] = L[i]  
22            i += 1  
23            k += 1  
24        while j < len(R):  
25            arr[k] = R[j]  
26            j += 1  
27            k += 1  
28        # --- Test Merge Sort ---  
29        print("\n--- Testing Merge Sort ---")  
30        test_merge = [30, 27, 43, 3, 9, 82, 10] # Test data  
31        merge_sort(test_merge) # Run sort  
32        print(f"Result: {test_merge}") # Print result
```

Code

```
# --- Merge Sort Algorithm ---  
def merge_sort(arr):  
    if len(arr) > 1: # Check if split needed  
        mid = len(arr) // 2 # Find middle  
        L = arr[:mid] # Left subarray  
        R = arr[mid:] # Right subarray  
        merge_sort(L) # Recursively sort left  
        merge_sort(R) # Recursively sort right  
        i = j = k = 0 # Initialize pointers  
        # Merge Loop  
        while i < len(L) and j < len(R):  
            if L[i] < R[j]: # Compare  
                arr[k] = L[i] # Add left  
                i += 1  
            else:  
                arr[k] = R[j] # Add right  
                j += 1  
            k += 1  
        # Check remaining elements  
        while i < len(L):  
            arr[k] = L[i]  
            i += 1  
            k += 1  
        while j < len(R):  
            arr[k] = R[j]  
            j += 1  
            k += 1
```

```
arr[k] = R[j]
j += 1
k += 1
print("\n--- Testing Merge Sort ---")# --- Test Merge Sort ---  
test_merge = [38, 27, 43, 3, 9, 82, 10] # Test data  
merge_sort(test_merge) # Run sort  
print(f"Result: {test_merge}") # Print result
```

5. write own Heap Sort

```
# --- Heapify Function ---
def heapify(arr, n, i):
    largest = i # Initialize root as largest
    left = 2 * i + 1 # Left child
    right = 2 * i + 2 # Right child

    # Check left child
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check right child
    if right < n and arr[right] > arr[largest]:
        largest = right

    # Swap if root is not largest
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest) # Recursive heapify

# --- Heap Sort Algorithm ---
def heap_sort(arr):
    n = len(arr) # Length
    # Build maxheap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    # Extract elements
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # Swap root to end
        heapify(arr, i, 0) # Heapify root

# --- Test Heap Sort ---
print("---- Testing Heap Sort ----")
test_heap = [64, 34, 25, 12, 22, 11, 98] # Test data
heap.sort(test_heap) # Run sort
print(f"Result: {test_heap}") # Print result
```

Code

```
# --- Heapify Function ---
def heapify(arr, n, i):
    largest = i # Initialize root as largest
    left = 2 * i + 1 # Left child
    right = 2 * i + 2 # Right child

    # Check left child
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check right child
    if right < n and arr[right] > arr[largest]:
        largest = right

    # Swap if root is not largest
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest) # Recursive heapify

# --- Heap Sort Algorithm ---
def heap_sort(arr):
    n = len(arr) # Length
    # Build maxheap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    # Extract elements
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # Swap root to end
        heapify(arr, i, 0) # Heapify root
```

```
# --- Test Heap Sort ---
print("\n--- Testing Heap Sort ---")
test_heap = [64, 34, 25, 12, 22, 11, 90]    # Test data
heap_sort(test_heap)    # Run sort
print(f"Result: {test_heap}")    # Print result
```

Problem 2 (Analyse Sorting Algorithms). Analyse succinctly, for all sorting Algorithms above, time and space complexities using the master theorem where applicable

Problem 2. Analyse Sorting Algorithms.

1 Master Theorem

For a recurrence of the form $T(n) = aT(\frac{n}{b}) + O(nd)$, (a - number of subproblems)

Case 1: If $d < \log_b(a)$, then $T(n) = O(n^{\log_b a})$

Case 2: If $d = \log_b a$, then $T(n) = O(nd \log n)$

Case 3: If $d > \log_b a$, then $T(n) = O(nd)$

1. Bubble Sort

Not applicable Master Theorem Applicability. Bubble Sort is not a divide-and-conquer algorithm; it uses nested loops. There is no recurrence relation of the form $T(n) = aT(\frac{n}{b}) + O(nd)$

Time Complexity

Best Case: $O(n)$. When the array is already sorted with early termination optimization.

Average / Worst Case: $O(n^2)$. Two nested loops each iterate up to n times.

Space Complexity: $O(1)$. In-place sorting, only uses a constant number of extra variables.

2. Quick Sort Random Pivot

Can use master theorem (only average)

Best / Average Case: The pivot splits the array roughly in half

Recurrence: $T(n) = 2T(\frac{n}{2}) + O(n)$

Here $a=2$, $b=2$, $d=1$, Since $\log_2 2 = 1 = d$ (use case 2)
 $T(n) = O(n \log n)$

Worst Case : The pivot is always the smallest or largest element, giving maximally unbalanced partitions.

Recurrence : $T(n) = T(n-1) + O(n)$ Not in the Master Theorem form

By telescoping : $T(n) = O(n) + O(n-1) + \dots + O(1) = O(n^2)$

Space Complexity : $O(\log n)$ average \rightarrow recursive call stack.
 $O(n)$ worst case.

3. Quick Sort Median-of Three Pivot

The median-of-three strategy improves pivot selection, making balanced partitions more likely. Can use Master Theorem. (only averages)

Best / Average Case : Same recurrence as random pivot :

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$a=2, b=2, d=\log_2 2 = 1 \quad \text{Case 2: } T(n) = O(n \log n)$$

Worst Case : Still $O(n^2)$, much rarer than basic Quick Sort. The Master Theorem does not apply to the worst case recurrence $T(n) = T(n-1) + O(n)$

space Complexity : $O(\log n)$ average, $O(n)$ worst case.

4. Merge Sort.

Can use master theorem (all cases)

Merge Sort always divides the array into exactly two halves and merges in linear time.

$$\text{Recurrence : } T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$a=2, b=2, d=1. \text{ Since } \log_2 2 = 1. \quad \underline{\text{Case 2}} \quad T(n) = O(n \log n)$$

for all cases $T(n) = O(n \log n)$.

Space Complexity: $O(n)$. requires a temporary array for the merge step.

5. Heap Sort

Can use Master Theorem. (heapsify)

can be applied to the heapsify (sift-down) subroutine:

In heapsify, the node is compared with its children and potentially recursed into one subtree of size at most $\frac{2n}{3}$

$$\text{Recurrence : } T(n) = T\left(\frac{2n}{3}\right) + O(1),$$

$$a=1, b=\frac{3}{2}, d=0. \text{ Since } \log_{\frac{3}{2}} 1 = 0 = d.$$

$$\text{Cas 2 } \Rightarrow T(n) = O(n \log n)$$

Overall Time Complexity.

Build Heap: $O(n)$ (summing heapsify costs across all levels)

Extract n elements: $n \times O(n \log n) = O(n^2 \log n)$

Total: $O(n^2 \log n)$ for all cases.

Space Complexity: $O(1) \rightarrow$ in-place sorting, the heap is built within the original array.