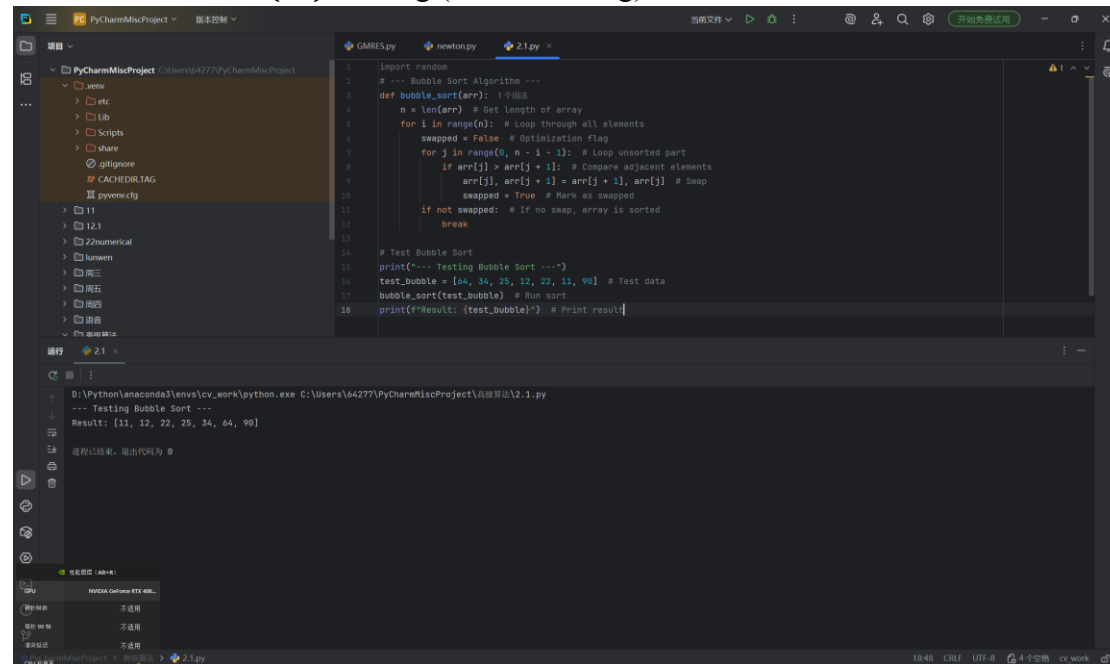


Problem 1 (Sorting Algorithms).

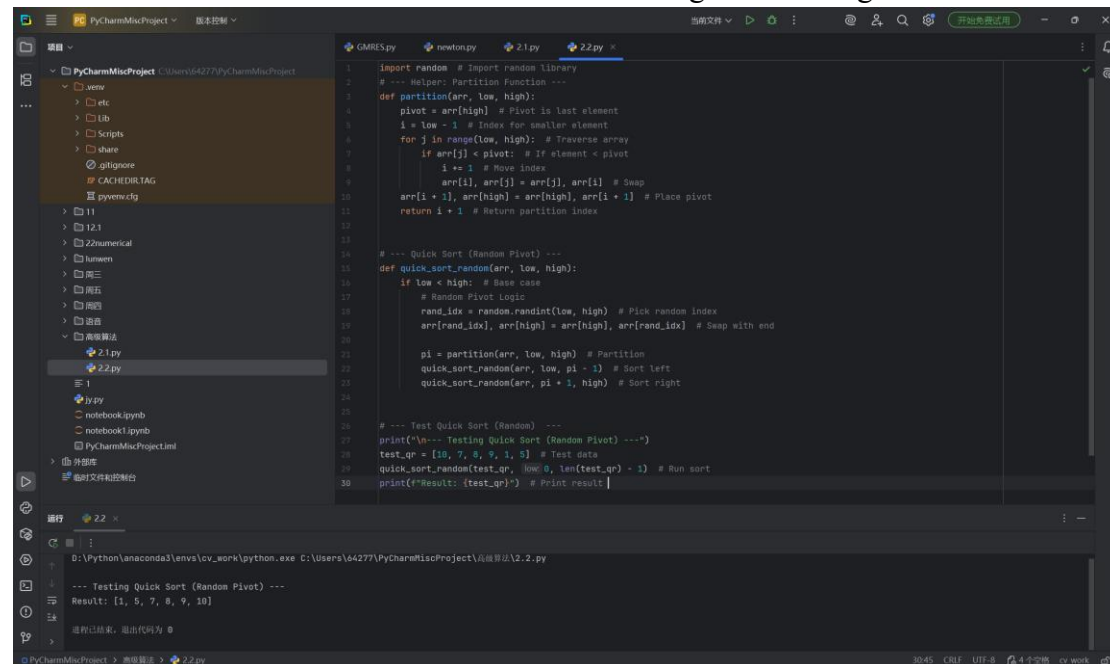
1. write own bad $O(n^2)$ sorting (bubble sorting)



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



The screenshot shows the PyCharm IDE with a project named 'PyCharmMacProject'. The file explorer on the left shows a directory structure with files like '2.1.py' and '2.2.py'. The main editor displays the code for '2.2.py', which implements a Quick Sort algorithm. The code includes a partition function and a recursive quick_sort_random function. The execution console at the bottom shows the output of the test: 'Result: [1, 5, 7, 8, 9, 10]'.

```
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         pi = partition(arr, low, high) # Partition
20         quick_sort_random(arr, low, pi - 1) # Sort left
21         quick_sort_random(arr, pi + 1, high) # Sort right
22
23 # --- Test Quick Sort (Random) ---
24 print("\n--- Testing Quick Sort (Random Pivot) ---")
25 test_qr = [10, 7, 8, 9, 1, 5] # Test data
26 quick_sort_random(test_qr, 0, len(test_qr) - 1) # Run sort
27 print(f"Result: {test_qr}") # Print result
```

Running: 2.2.py
D:\Python\anaconda3\envs\cv_work\python.exe C:\Users\64277\PyCharmMacProject\src\2.2.py
--- Testing Quick Sort (Random Pivot) ---
Result: [1, 5, 7, 8, 9, 10]
进程已结束，退出代码为 0

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
```

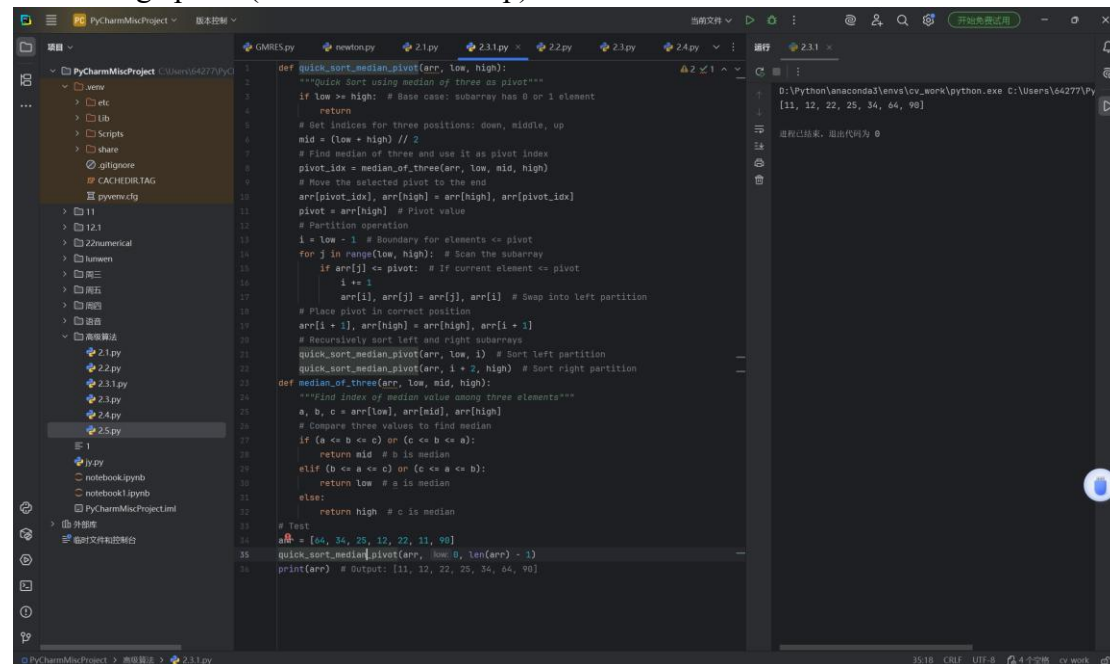
3. write own Quick Sort add random pivot

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



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

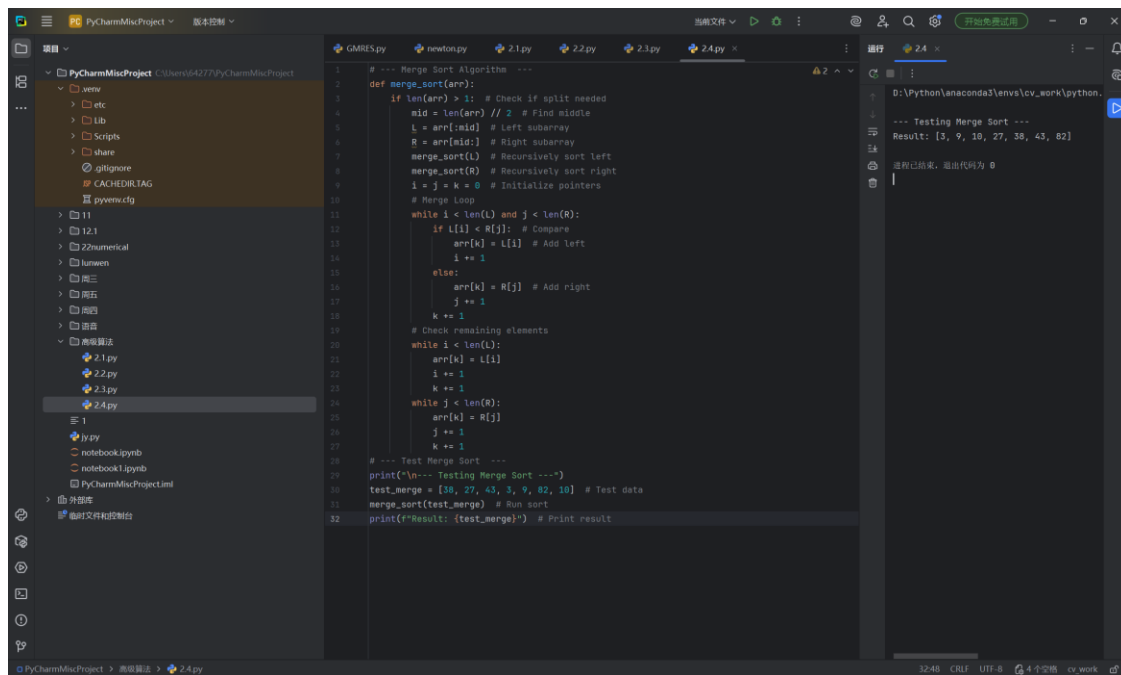
def median_of_three(arr, low, mid, high):
    """Find index of median value among three elements"""
    a, b, c = arr[low], arr[mid], arr[high]
    # Compare three values to find median
    if (a <= b <= c) or (c <= b <= a):
        return mid # b is median
    elif (b <= a <= c) or (c <= a <= b):
        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, low=0, len(arr)-1)
print(arr) # Output: [11, 12, 22, 25, 34, 64, 90]
```

```
    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



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

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

```
1 # --- Heapify Function ---
2 def heapify(arr, n, i):
3     largest = i # Initialize root as largest
4     left = 2 * i + 1 # Left child
5     right = 2 * i + 2 # Right child
6
7     # Check left child
8     if left < n and arr[left] > arr[largest]:
9         largest = left
10
11    # Check right child
12    if right < n and arr[right] > arr[largest]:
13        largest = right
14
15    # Swap if root is not largest
16    if largest != i:
17        arr[i], arr[largest] = arr[largest], arr[i]
18        heapify(arr, n, largest) # Recursive heapify
19
20 # --- Heap Sort Algorithm ---
21 def heap_sort(arr):
22     n = len(arr) # Length
23     # Build maxheap
24     for i in range(n // 2 - 1, -1, -1):
25         heapify(arr, n, i)
26     # Extract elements
27     for i in range(n - 1, 0, -1):
28         arr[i], arr[0] = arr[0], arr[i] # Swap root to end
29         heapify(arr, i, 0) # Heapify root
30
31 # --- Test Heap Sort ---
32 if __name__ == '__main__':
33     test_heap = [64, 34, 25, 12, 22, 11, 98] # Test data
34     heap_sort(test_heap) # Run sort
35     print("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.

Δ 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 imbalanced 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 average)

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.

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

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

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. (heapify)

Can be applied to the heapify (sift-down) subroutine:

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

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

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

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

Overall Time Complexity.

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

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

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

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