a. Heapsort Algorithm

Heapsort is a comparison-based sorting algorithm that works by first building a **max-heap** (for ascending order) or **min-heap** (for descending order) from the input array and then repeatedly extracting the maximum element (for max-heap) or minimum element (for min-heap) and placing it at the end of the array.

Heapsort involves two main operations:

1. **Building a max-heap.**
2. **Extracting elements from the heap and placing them into the sorted array.**

Required Algorithms for Heapsort

1. Heapify Algorithm (Max-Heapify)

The Heapify algorithm ensures that a subtree rooted at index iii in the array follows the max-heap property, which means the parent node is greater than or equal to both of its children.

python

Copy code

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

    # Check if left child exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child exists and is greater than largest so far
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root, swap it with the largest child
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
```

```
        max_heapify(arr, n, largest)  # Recursively heapify the affected subtree
```

2. Building the Max-Heap

To build a max-heap from an unsorted array, we start from the last non-leaf node (which is at index n2−1\frac{n}{2} - 12n−1) and call max_heapify on each node up to the root.

python

Copy code

```
def build_max_heap(arr):
    n = len(arr)
    # Start from the last non-leaf node and heapify each subtree
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)
```

3. Heapsort Algorithm

After building the max-heap, we repeatedly swap the root (maximum element) with the last element in the heap, then reduce the heap size and call max_heapify to restore the heap property.

python

Copy code

```
def heapsort(arr):
    n = len(arr)

    # Build a max-heap
    build_max_heap(arr)

    # One by one extract elements from the heap
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]  # Swap current root with last element
        max_heapify(arr, i, 0)  # Heapify the reduced heap
```

b. Detailed Analysis of Heapsort Algorithm

Time Complexity Analysis:

1. **Building the Max-Heap (build_max_heap):**

    - The build_max_heap function calls max_heapify on each internal node, starting from the last non-leaf node.
    - Each call to max_heapify takes O(log⁡n)O(\log n)O(logn) time in the worst case.
    - The total time complexity for building the heap is: O(n)O(n)O(n) This is because not every node in the heap requires O(log⁡n)O(\log n)O(logn) operations—leaf nodes don't require heapification, and as we move up the tree, fewer nodes are involved.

2. **Heapsort Execution:**
    - After building the max-heap, we perform n−1n-1n−1 extractions. Each extraction involves:
        - Swapping the root (max element) with the last element in the heap.
        - Calling max_heapify on the root to restore the heap property, which takes O(log⁡n)O(\log n)O(logn) time.
    - Therefore, the total time complexity for the heap extraction part is: O(nlog⁡n)O(n \log n)O(nlogn)

3. **Overall Time Complexity:**
    - Building the heap takes O(n)O(n)O(n) time, and the sorting phase takes O(nlog⁡n)O(n \log n)O(nlogn) time.
    - Hence, the overall time complexity of Heapsort is: O(nlog⁡n)O(n \log n)O(nlogn)
    - Heapsort is **in-place** (does not require additional storage other than the input array) and has a **space complexity of O(1)O(1)O(1)**.

Stability:

- Heapsort is **not stable** because equal elements might get swapped in a way that their relative order is not preserved.