Required Algorithms for Heapsort

The Heapsort algorithm involves two primary steps:

1. *Build a Max Heap*: Transform the input array into a max heap.

2. *Sort the Array*: Repeatedly extract the largest element from the heap and move it to the end of the array, reducing the heap size.

Algorithm 1: Heapify

Heapify(A, n, i)

Input:

A is the array, n is the size of the heap, i is the index to heapify.

Output:

Maintains the max-heap property for the subtree rooted at i.

1. Initialize largest = i.

2. Set left = 2i + 1 (left child index).

3. Set right = 2i + 2 (right child index).

4. If left < n and A[left] > A[largest], set largest = left.

5. If right < n and A[right] > A[largest], set largest = right.

6. If largest != i:

    a. Swap A[i] and A[largest].

    b. Recursively call Heapify(A, n, largest).

Algorithm 2: BuildMaxHeap

BuildMaxHeap(A, n)

Input: A is the array, n is the size of the array.

Output: Constructs a max-heap from the input array.

1. For i = n/2 - 1 down to 0:

    a. Call Heapify(A, n, i).

Algorithm 3: HeapSort

HeapSort(A)

Input: A is the array to sort.

Output: Sorted array A.

1. Call BuildMaxHeap(A, n).

2. For i = n - 1 down to 1:

    a. Swap A[0] and A[i].

    b. Call Heapify(A, i, 0).

----

b. Analysis of Heapsort Algorithms

Time Complexity:

1. Heapify Operation:

   The heapify operation runs in O(log n), as the height of the heap is proportional to (log n).

2. BuildMaxHeap Operation:

Building the max heap involves calling Heapify for each non-leaf node, resulting in O(n) time.

3. Sorting the Heap:

Extracting the max element and re-heapifying for n elements takes O(n log n).

Overall time complexity: O(n log n).

Space Complexity:

Heapsort operates in-place and requires no additional memory, resulting in O(1) space complexity.