MTech CSE – 1st Semester
Student ID: **A125023**
Student Name: **SRIJITA VERMA**

Question

Prove that the time complexity of the recursive *Heapify* operation is $O(\log n)$ using the recurrence relation:

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

Answer

Heapify Operation: Conceptual Overview

The Heapify operation is a core procedure used to maintain the heap property in a binary heap data structure. In a max-heap, each parent node must contain a value greater than or equal to the values of its children, while in a min-heap, the parent node must contain a value smaller than or equal to its children. Heapify is invoked when this property is violated, such as after the deletion of the root element or during heap construction.

The operation begins at a given node and compares its value with those of its left and right children. If the heap property is violated, the node is swapped with the appropriate child. This adjustment may introduce a violation further down the tree, so Heapify is applied recursively to the affected subtree. Importantly, Heapify proceeds along only one path from the root toward the leaves.

Since a binary heap with n elements has height $O(\log n)$, the recursion depth of Heapify is logarithmic.

At each recursive step, only a constant amount of work is performed, consisting of a small number of comparisons and at most one swap. This behavior motivates the recurrence relation used to analyze its time complexity.

Heapify Pseudocode with Complexity Annotation

Recursive Heapify (Max-Heap)

```
HEAPIFY(A, n, i)
1.  largest ← i                      // O(1)
2.  left ← 2i + 1                    // O(1)
3.  right ← 2i + 2                   // O(1)

4.  if left < n and A[left] > A[largest]
5.      largest ← left               // O(1)
```

```
6.  if right < n and A[right] > A[largest]
7.      largest ← right                 // O(1)

8.  if largest  i
9.      swap A[i] and A[largest]        // O(1)
10.     HEAPIFY(A, n, largest)           // T(2n/3)
```

Complexity Analysis from Pseudocode

Lines 1–9 perform a constant number of comparisons and assignments. Therefore, the total time taken by these steps is $O(1)$.

Line 10 makes at most one recursive call on a subtree whose size is at most $\frac{2n}{3}$.

Heapify proceeds down only one branch of the heap.

This directly gives the recurrence relation:

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

Recurrence Relation for Heapify

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

where:

- $T(n)$ is the time required to Heapify a subtree of size n,
- $\frac{2n}{3}$ represents the size of the largest child subtree,
- $O(1)$ accounts for comparisons and swaps performed at each level.

Solving the Recurrence

Step 1: Expand the Recurrence

Let the constant amount of work done at each level be denoted by c. Expanding the recurrence relation:

$$\begin{aligned}
T(n) &= T\left(\frac{2n}{3}\right) + c \\
&= T\left(\left(\frac{2}{3}\right)^2 n\right) + 2c \\
&\vdots \\
&= T\left(\left(\frac{2}{3}\right)^k n\right) + kc
\end{aligned}$$

Step 2: Determine the Recursion Depth

The recursion terminates when the problem size reduces to 1:

$$\left(\frac{2}{3}\right)^k n = 1$$

Solving for k:

$$\left(\frac{2}{3}\right)^k = \frac{1}{n}$$

Taking logarithms on both sides:

$$k \log\left(\frac{2}{3}\right) = \log\left(\frac{1}{n}\right)$$

$$k = \frac{\log n}{\log\left(\frac{3}{2}\right)}$$

Since $\log\left(\frac{3}{2}\right)$ is a positive constant, we conclude that:

$$k = O(\log n)$$

Step 3: Compute the Total Time Complexity

Substituting the recursion depth into the expanded form:

$$T(n) = T(1) + kc$$

Since:

$$T(1) = O(1), \quad k = O(\log n), \quad c = O(1),$$

we obtain:

$$T(n) = O(\log n)$$

Recursion Tree Explanation (In Words)

The recurrence relation for the Heapify operation is given by:

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

This recurrence can be visualized using a recursion tree, where each node represents a recursive call to Heapify on a subtree of a certain size.

At the root level of the recursion tree, the Heapify operation is applied to a heap of size n. The work done at this level consists of a constant number of comparisons and at most one swap, which together take $O(1)$ time.

From this root, the recursion proceeds to exactly one child node, corresponding to a recursive call on the largest child subtree of size at most $\frac{2n}{3}$. Unlike divide-and-conquer algorithms that branch into multiple subproblems, Heapify generates only a single branch at each level. As a result, the recursion tree is not wide but forms a single downward path.

At the next level, the same process repeats. The Heapify operation works on a subtree of size $\frac{2n}{3}$, again performing $O(1)$ work and making a single recursive call on a subtree of size

$$\left(\frac{2}{3}\right)^2 n.$$

Each subsequent level of the recursion tree represents a further reduction in problem size by a factor of $\frac{2}{3}$.

This pattern continues until the size of the subtree becomes constant, at which point the recursion terminates. The total number of levels in the recursion tree is equal to the number of times n can be multiplied by $\frac{2}{3}$ before it reaches 1. This occurs after $O(\log n)$ levels.

Since each level of the recursion tree contributes only $O(1)$ work, the total cost of all levels combined is proportional to the height of the tree. Therefore, the total running time of Heapify is:

$$O(1) \times O(\log n) = O(\log n).$$

This recurrence also satisfies the Master Theorem for decreasing subproblem size, which further confirms the logarithmic time bound obtained through the recurrence expansion.

Final Conclusion

$$T(n) = O(\log n)$$

Thus, the recursive Heapify operation runs in logarithmic time. This result matches the height of a binary heap and explains the efficiency of heap-based algorithms such as Heap Sort and priority queues, where Heapify is a fundamental operation.