MTech CSE – 1st Semester
Student Name: SIDDHARTH KUMAR
Student ID: A125021

Question 1

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:

Understanding the Heapify Operation

Heapify is a fundamental procedure used to preserve the heap property in a binary heap data structure. In a max-heap, the value stored at a parent node must be greater than or equal to the values stored in its children, whereas the opposite condition holds for a min-heap. Heapify is typically applied after deletion of the root or during heap construction.

The procedure starts at a given node and compares it with its left and right children. If the heap property is violated, the node is swapped with the appropriate child. This process may propagate further down the heap, so Heapify is recursively applied to the affected subtree. Importantly, at every step, recursion follows only one path from the root toward the leaf.

Since a binary heap containing n elements has a height of $O(\log n)$, the maximum number of recursive calls made by Heapify is logarithmic.

At each recursive call, the work performed consists of a constant number of comparisons and at most one swap. This leads naturally to the recurrence relation used for analyzing the time complexity.

Heapify Pseudocode with Cost Annotation

Recursive Heapify for Max-Heap

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

4.  if left < n and A[left] > A[largest]
5.      largest ← left

6.  if right < n and A[right] > A[largest]
7.      largest ← right

8.  if largest  i
9.      swap A[i] and A[largest]
10.     HEAPIFY(A, n, largest)
```

Each statement outside the recursive call executes in constant time.

Deriving the Recurrence Relation

The comparison and swapping operations take $O(1)$ time at each level. The recursive call is made on the largest child subtree, whose size is at most $\frac{2n}{3}$.

Therefore, the time complexity satisfies the recurrence:

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

Solving the Recurrence

Step 1: Expansion

Let the constant work at each level be denoted by c. Expanding the recurrence:

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

Step 2: Recursion Depth

The recursion ends when the input size becomes 1:

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

Solving for k:

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

Step 3: Final Time Complexity

Substituting the recursion depth:

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

Since $T(1) = O(1)$ and $k = O(\log n)$,

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

Conclusion

The recursive Heapify operation follows a single downward path in the heap and performs only constant work at each level. As a result, its running time is proportional to the height of the heap.

$$\boxed{T(n) = O(\log n)}$$

This confirms the efficiency of Heapify and explains its critical role in heap-based algorithms such as Heap Sort and priority queues.