1 Inorder Binary Tree Traversal

1.1 Traversal

There are several types of traversals. Depth-first traversals are traversals which traverse the complete depth of a node, and then move to the next node, as opposed to breadth-first traversals which traverse completely on the same level before moving on to the next level. Depth first traversals are of 3 types -

1. Inorder
2. Preorder
3. Postorder

1.2 Inorder traversal

In inorder tree traversal, we recursively call the function on the left subtree first, then we print the current node value, and then we call the function recursively on the right subtree. This method of traversal makes the value on the leftmost node to be printed first and the rightmost node to be printed at the last, and all the nodes in the same order as in the tree, and hence, it is called inorder traversal. [**?**]

In case of a binary tree, this traversal will print in **sorted order** (or reverse sorted, if the tree is maintained in a reverse way.)

1.3 Analysis

The algorithm runs in $O(n)$ time complexity and $O(1)$ space complexity. The analysis can be done as follows-

1. Time taken to print a node is of order $O(1)$.
2. $T(n) = T(k) + T(n-k) + O(1)$

Each node is only accessed once, hence, we can easily say that the time compelexity is of order $O(n)$.

2 Shortest path problem

2.1 Introduction

In a graph, the 'shortest path problem' is the problem of finding the minimum distance (represented by weights of edges) between two nodes of the graph. The problem of finding the shortest path between two intersections on a road map (the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment) may be modelled by a special case of the shortest path problem in graphs.

2.2 Definition

The shortest path problem can be defined for graphs whether undirected, directed, or mixed. It is defined here for undirected graphs; for directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.[**?**]

2.3 Algorithms

There are many algorithms to solve the Shortest Path problem. Some of them are-

1. **Dijkstra's algorithm** solves the single-source shortest path problem.
2. **BellmanFord algorithm** solves the single-source problem if edge weights may be negative.
3. **A* search algorithm** solves for single pair shortest path using heuristics to try to speed up the search.
4. **FloydWarshall algorithm** solves all pairs shortest paths.
5. **Johnson's algorithm** solves all pairs shortest paths, and may be faster than FloydWarshall on sparse graphs.
6. **Viterbi algorithm** solves the shortest stochastic path problem with an additional probabilistic weight on each node.

3 Quicksort

3.1 Introduction

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order.[**?**]

3.2 Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are -

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Base case: When there are 0 or 1 elements in the array, which is trivially sorted. We use the Hoare partitioning scheme to determine the partition.

3.3 Pseudocode

3.3.1 Algorithm

```
1: function QUICKSORT(A, begin, end)
2:     if begin < end then
3:         p ← PARTITION(A, begin, end)
4:         QUICKSORT(A, begin, p − 1)
5:         QUICKSORT(A, p + 1, end)
6:     end if
7: end function
```

3.3.2 Partitioning Scheme

```
1: function PARTITION(A, begin, end)
2:     pivot ← A[end]
3:     i ← begin
4:     for j ← begin to end − 1 do
5:         if A[j] ≤ pivot then
6:             swap A[i] with A[j]
7:             i ← i + 1
8:         end if
9:     end for
10:    swap A[i] with A[end]
11:    return i
12: end function
```