

UNIT 4

1. Decrease-and-Conquer:
 - a. Introduction,
 - b. Insertion Sort,
 - c. Topological Sorting
2. Divide-and-Conquer:
 - a. Introduction,
 - b. Merge Sort,
 - c. Quick Sort,
 - d. Binary Search, Binary Tree traversals and related properties

Decrease & Conquer

- Decrease-and-conquer is an algorithm design paradigm in which a problem is transformed into a smaller subproblem and then that subproblem is solved first.
- Unlike the case of [Divide and conquer](#) algorithms, the reduction process here generates a single subproblem.
- The transformation is applied either iteratively or recursively until the sub-problem becomes simple enough to be solved directly as a base case.
- Finally, the solutions of all sub-problems are extended to get a solution to the original problem

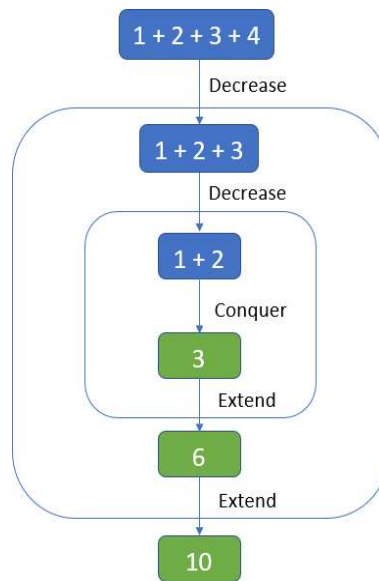
The decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion). There are three major variations of decrease-and-conquer:

1. decrease by a constant
2. decrease by a constant factor
3. variable size decrease

The steps of the decrease-and-conquer algorithm

Generally, the decrease-and-conquer approach implies the following three steps:

- **Decrease:** reduce a problem to a smaller instance of the same problem.
- **Conquer:** iteratively or recursively solve the sub-problem.
- **Extend:** apply the sub-problem solution to the next sub-problem to get a solution to the original problem.



Insertion Sort

Insertion sort is one of the simplest sorting algorithms for the reason that it sorts a single element at a particular instance. It is not the best sorting algorithm in terms of performance, but it's slightly more efficient than [selection sort](#) and [bubble sort](#) in practical scenarios. It is an intuitive sorting technique.

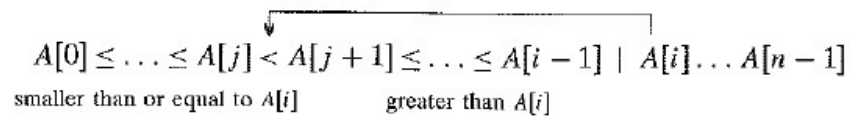


FIGURE 5.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

ALGORITHM *InsertionSort*($A[0..n - 1]$)

//Sorts a given array by insertion sort

//Input: An array $A[0..n - 1]$ of n orderable elements//Output: Array $A[0..n - 1]$ sorted in nondecreasing order**for** $i \leftarrow 1$ **to** $n - 1$ **do** $v \leftarrow A[i]$ $j \leftarrow i - 1$ **while** $j \geq 0$ **and** $A[j] > v$ **do** $A[j + 1] \leftarrow A[j]$ $j \leftarrow j - 1$ $A[j + 1] \leftarrow v$

Example

89		45	68	90	29	34	17
45		89		68	90	29	34
45		68		89		90	29
45		68		89		90	
29		45		68		89	
29		34		45		68	
17		29		34		45	

Working of Insertion Sort algorithm

Consider an example: $arr[]: \{12, 11, 13, 5, 6\}$

12	11	13	5	6
----	----	----	---	---

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6
----	----	----	---	---

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6
----	----	----	---	---

Second Pass:

- Now, move to the next two elements and compare them

11	12	13	5	6
----	----	----	---	---

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are **11 and 12**
- Moving forward to the next two elements which are 13 and 5

11	12	13	5	6
----	----	----	---	---

- Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6
----	----	---	----	---

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6
----	---	----	----	---

- Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6
---	----	----	----	---

- Here, 5 is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are **5, 11 and 12**
- Moving to the next two elements **13 and 6**

5	11	12	13	6
---	----	----	-----------	----------

- Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13
---	----	----	----------	-----------

- Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13
---	----	----------	-----------	----

- Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13
---	----------	-----------	----	----

- Finally, the array is completely sorted.

Time Complexity of Insertion Sort

- The **worst-case** time complexity of the Insertion sort is **$O(N^2)$**
- The **average case** time complexity of the Insertion sort is **$O(N^2)$**
- The time complexity of the **best case** is **$O(N)$** .

Characteristics of Insertion Sort

- This algorithm is one of the simplest algorithms with a simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.

TOPOLOGICAL SORTING

- Topological sorting is a sorting method to list the vertices of the graph in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.
- NOTE: There is no solution for topological sorting if there is a cycle in the digraph . [MUST be a DAG]
- Topological sorting problem can be solved by using
 - DFS method
 - Source removal method

DFS METHOD

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A **directed cycle** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, **a, b, a** is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a **dag**, an acronym for **directed acyclic graph**.

Edge directions lead to new questions about digraphs that are either meaningless or trivial for undirected graphs. In this section, we discuss one such question. As a motivating example, consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (Figure 4.6). In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called **topological sorting**. It can be posed for an

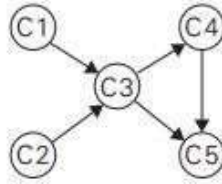


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

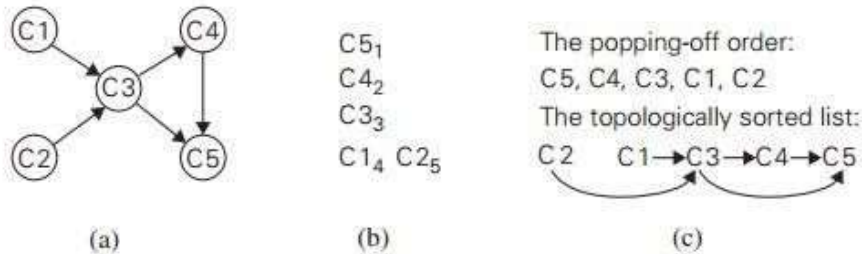


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

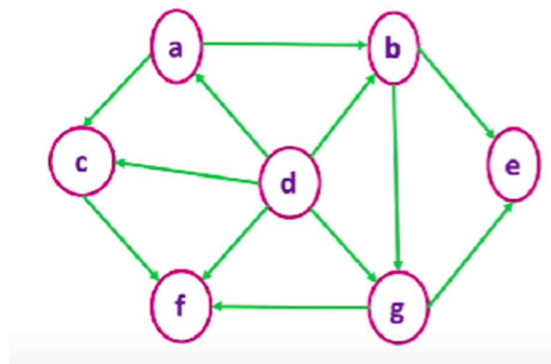
arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v . (Otherwise, (u, v) would have been a back edge.) Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

Figure 4.7 illustrates an application of this algorithm to the digraph in Figure 4.6. Note that in Figure 4.7c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.

Another way to Solve using DFS Method



Step	Stack	Adjacent Vertex	Node Visited	Stack Pop
initial	a	-	a	-
1.	a	b	a, b	-
2.	a, b	e	a, b, e	-
3	a, b, e	-	a, b, e	e
4	a, b	g	a, b, e, g	-
5	a, b, g	f	a, b, e, g, f	-
6	a, b, g, f	-	a, b, e, g, f	f
7	a, b, g	-	a, b, e, g, f	g
8	a, b	-	a, b, e, g, f	b
9.	a	c	a, b, e, g, f, c	-
10	a, c	-	a, b, e, g, f, c	c
11	a	-	a, b, e, g, f, c	a
12	d	-	a, b, e, g, f, c, d	-

SOURCE REMOVAL

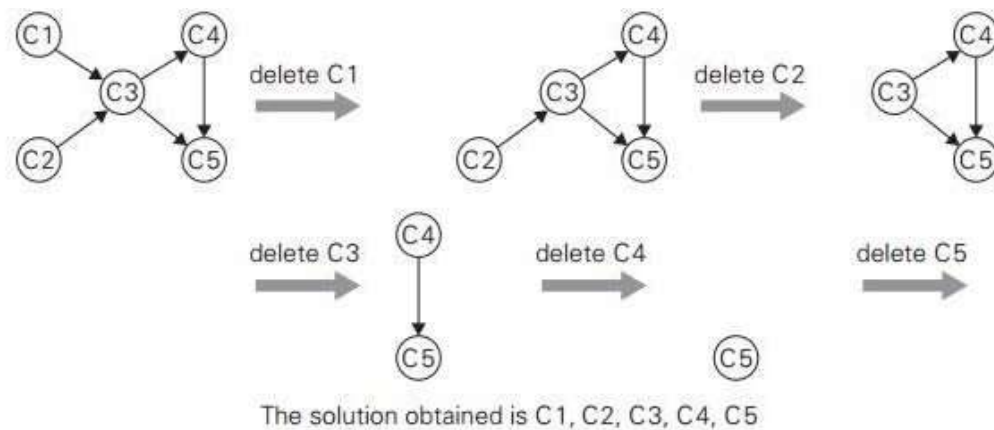


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a **source**, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved—see Problem 6a in this section's exercises.) The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8.

Algorithm *TopoSource*($G(V, E)$)

make empty Container of vertices, C

make empty List of vertices, S

for each vertex v in V **do**

$\text{incounter}(v) \leftarrow \text{in-deg}(v)$

if $\text{incounter}(v) == 0$ **then** $C.\text{add}(v)$

while C is not empty **do**

$u \leftarrow C.\text{remove}()$

$S.\text{add}(u)$

for each edge (u, w) **do** // edge is out from u to w

$\text{incounter}(w) \leftarrow \text{incounter}(w) - 1$

if $\text{incounter}(w) == 0$ **then** $C.\text{add}(w)$

if $S.\text{size}() == V.\text{size}()$ **then return** S

else return " G has a cycle"

Divide-and-Conquer

Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

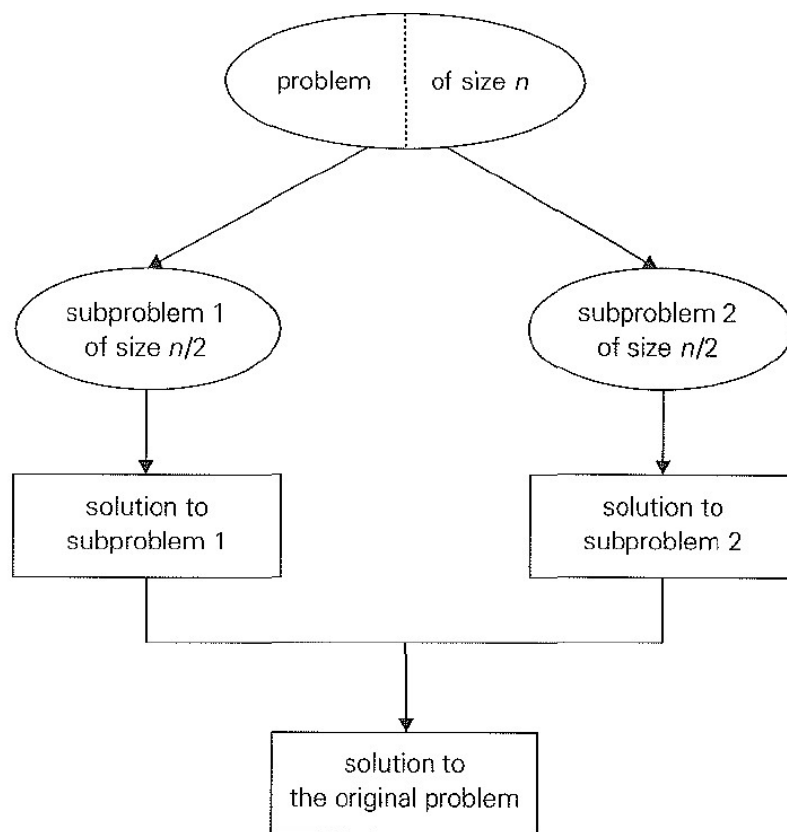


FIGURE 4.1 Divide-and-conquer technique (typical case)

Examples of Divide and Conquer Approach

The following computer algorithms are based on divide-and-conquer programming approach –

1. Max-Min Problem
2. Merge Sort Algorithm
3. Quick Sort Algorithm
4. Binary Search Algorithm

Mergesort

- Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array $A[0 \dots n - 1]$ by dividing it into two halves
- $A[0 \dots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \dots n - 1]$, sorting each of them recursively, and then **merging the two smaller sorted arrays into a single sorted one.**

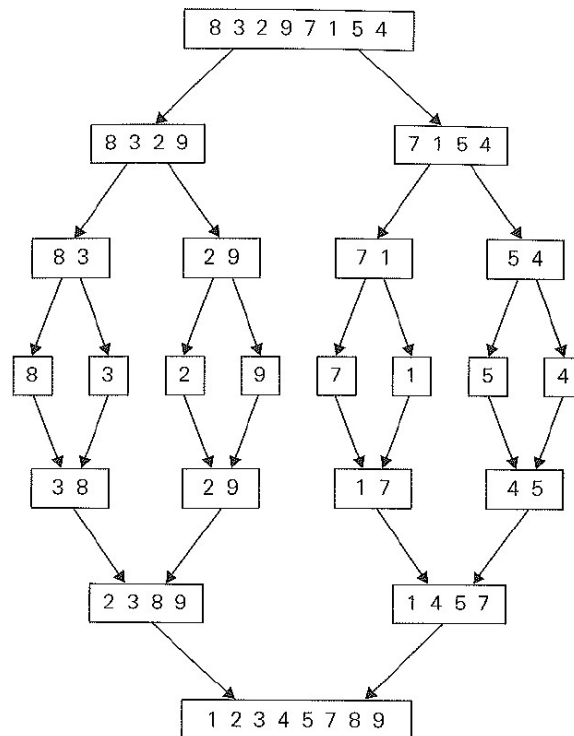


FIGURE 4.2 Example of mergesort operation

ALGORITHM *Mergesort*($A[0..n - 1]$)

//Sorts array $A[0..n - 1]$ by recursive mergesort

//Input: An array $A[0..n - 1]$ of orderable elements

//Output: Array $A[0..n - 1]$ sorted in nondecreasing order

if $n > 1$

 copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$

 copy $A[\lfloor n/2 \rfloor..n - 1]$ to $C[0..\lfloor n/2 \rfloor - 1]$

Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)

Mergesort($C[0..\lfloor n/2 \rfloor - 1]$)

Merge(B, C, A)

ALGORITHM *Merge*($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)

//Merges two sorted arrays into one sorted array

//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted

//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

if $B[i] \leq C[j]$

$A[k] \leftarrow B[i]$; $i \leftarrow i + 1$

else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$

$k \leftarrow k + 1$

if $i = p$

 copy $C[j..q-1]$ to $A[k..p+q-1]$

else copy $B[i..p-1]$ to $A[k..p+q-1]$

Quick Sort

Quick sort is another sorting algorithm that follows the Divide and Conquer approach. It works by selecting a "pivot" element from the array and partitioning the other elements into two groups – those less than the pivot and those greater than the pivot. The pivot element is then placed in its correct position, and the algorithm is recursively applied to the two groups. Quick sort has an average-case time complexity of $O(n \log n)$, but its worst-case time complexity is $O(n^2)$, making it less optimal than merge sort for certain datasets.

How Quick Sort Works

Here's a high-level overview of the quick sort algorithm:

- If the array has less than two elements, it is already sorted. Return the array.
- Choose a pivot element from the array.
- Partition the array into two groups – elements less than the pivot and elements greater than the pivot.
- Recursively sort the two groups.
- Combine the sorted groups and the pivot element to create the final sorted array.

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array $A[0..n-1]$ to achieve its **partition**, a situation where all the elements before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

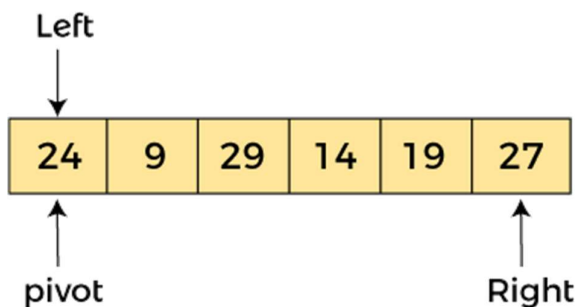
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

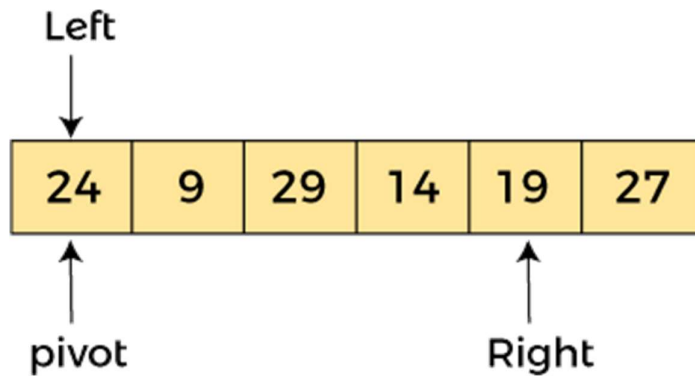
24	9	29	14	19	27
----	---	----	----	----	----

In the given array, we consider the leftmost element as pivot. So, in this case, $a[\text{left}] = 24$, $a[\text{right}] = 27$ and $a[\text{pivot}] = 24$.

Since, pivot is at left, so algorithm starts from right and move towards left.

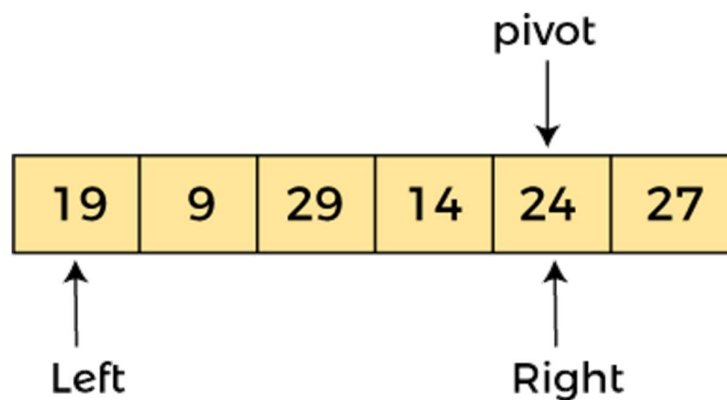


Now, $a[\text{pivot}] < a[\text{right}]$, so algorithm moves forward one position towards left, i.e. -



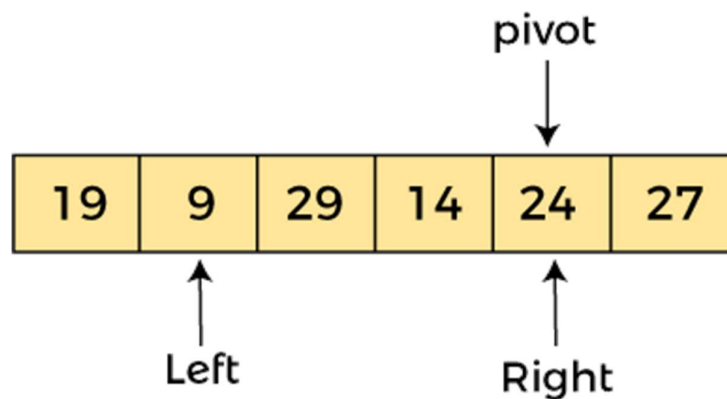
Now, $a[\text{left}] = 24$, $a[\text{right}] = 19$, and $a[\text{pivot}] = 24$.

Because, $a[\text{pivot}] > a[\text{right}]$, so, algorithm will swap $a[\text{pivot}]$ with $a[\text{right}]$, and pivot moves to right, as -

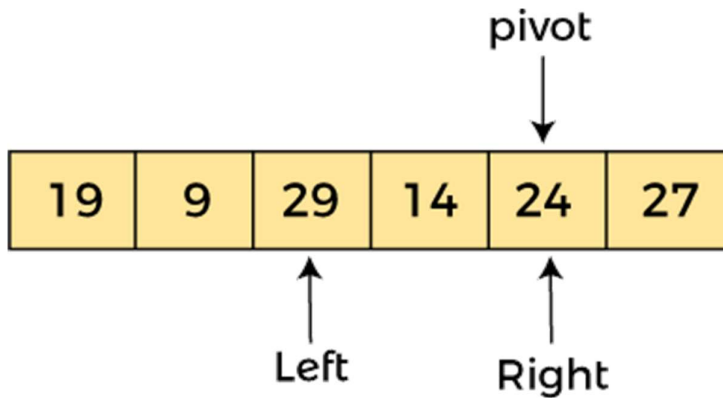


Now, $a[\text{left}] = 19$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. Since, pivot is at right, so algorithm starts from left and moves to right.

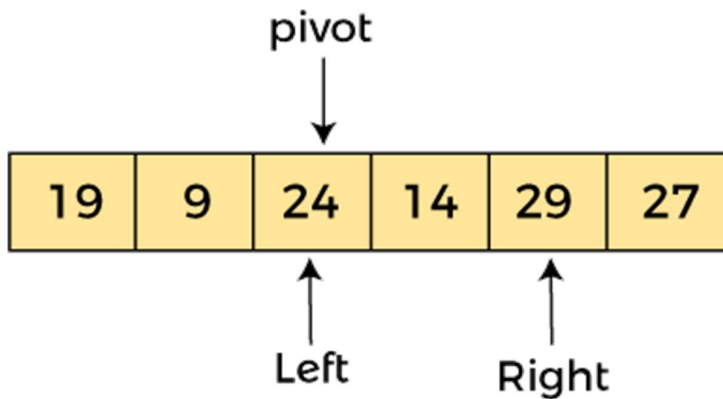
As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as -



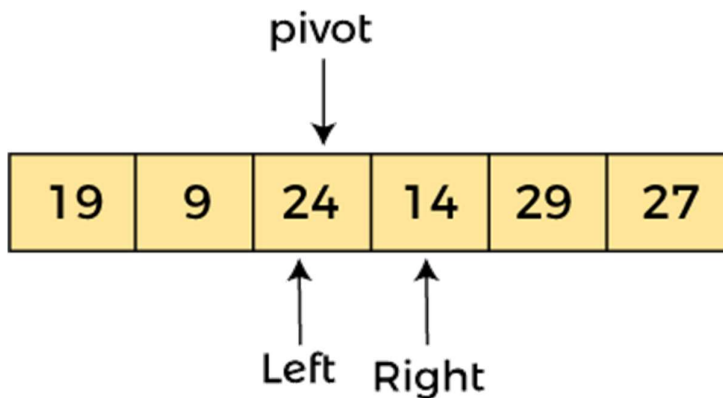
Now, $a[\text{left}] = 9$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as -



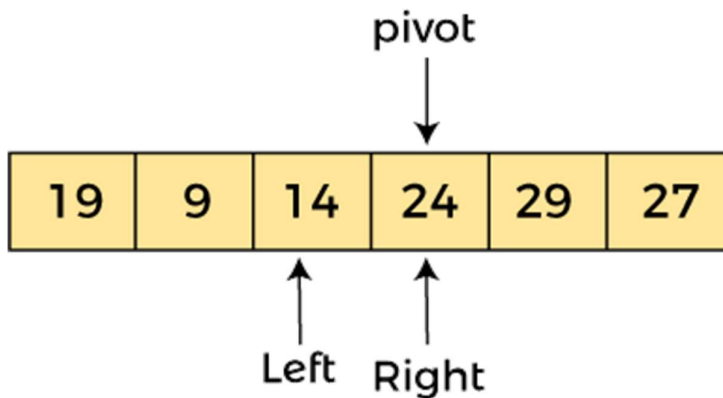
Now, $a[\text{left}] = 29$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{left}]$, so, swap $a[\text{pivot}]$ and $a[\text{left}]$, now pivot is at left, i.e. -



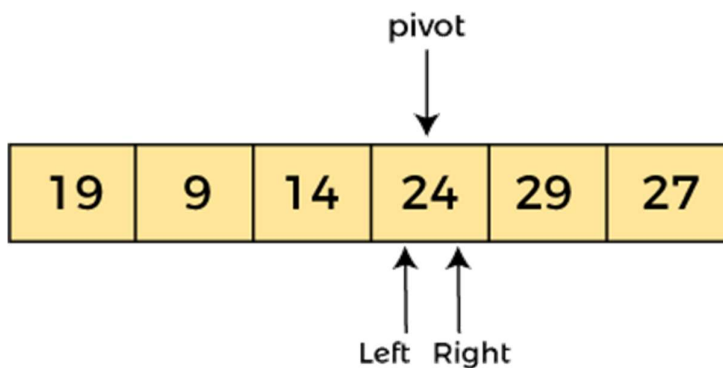
Since, pivot is at left, so algorithm starts from right, and move to left. Now, $a[\text{left}] = 24$, $a[\text{right}] = 29$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{right}]$, so algorithm moves one position to left, as -



Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 14$. As $a[\text{pivot}] > a[\text{right}]$, so, swap $a[\text{pivot}]$ and $a[\text{right}]$, now pivot is at right, i.e. -



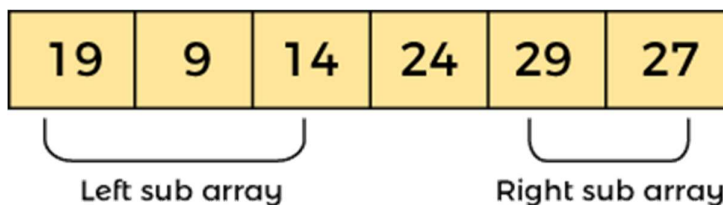
Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 14$, and $a[\text{right}] = 24$. Pivot is at right, so the algorithm starts from left and move to right.



Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 24$. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

9	14	19	24	27	29
---	----	----	----	----	----

ALGORITHM *Quicksort*($A[l..r]$)



//Sorts a subarray by quicksort
 //Input: A subarray $A[l..r]$ of $A[0..n - 1]$, defined by its left and right indices
 // l and r
 //Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
 $s \leftarrow \text{Partition}(A[l..r])$ // s is a split position
 Quicksort($A[l..s - 1]$)
 Quicksort($A[s + 1..r]$)

ALGORITHM *Partition*($A[l..r]$)

//Partitions a subarray by using its first element as a pivot
 //Input: A subarray $A[l..r]$ of $A[0..n - 1]$, defined by its left and right
 // indices l and r ($l < r$)
 //Output: A partition of $A[l..r]$, with the split position returned as
 // this function's value
 $p \leftarrow A[l]$
 $i \leftarrow l; \quad j \leftarrow r + 1$
repeat
 repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
 repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
 swap($A[i], A[j]$)
until $i \geq j$
 swap($A[i], A[j]$) //undo last swap when $i \geq j$
 swap($A[l], A[j]$)
return j

Binary Search

- Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element. Otherwise, the search is called unsuccessful.
- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match

BINARY SEARCH			 Array  Divide and Conquer
Best	Average	Worst	
$O(1)$	$O(\log n)$	$O(\log n)$	

search (A, t)
1. low = 0
2. high = n - 1
3. **while** (low ≤ high) **do**
4. ix = (low + high)/2
5. **if** (t = A[ix]) **then**
6. **return true**
7. **else if** (t < A[ix]) **then**
8. high = ix - 1
9. **else** low = ix + 1
10. **return false**
end

search (A, 11)

low

ix

high

first pass

1 4 8 9 11 15 17

low ix high

second pass

1 4 8 9 11 15 17

low ix high

third pass

1 4 8 9 11 15 17

low ix high

explored elements

Binary Search



	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	9
23 > 16 take 2 nd half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 < 56 take 1 st half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

Binary Tree Traversals and Related Properties

A **binary tree** T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root. We usually think of a binary tree as a special case of an ordered tree (Figure 5.4).

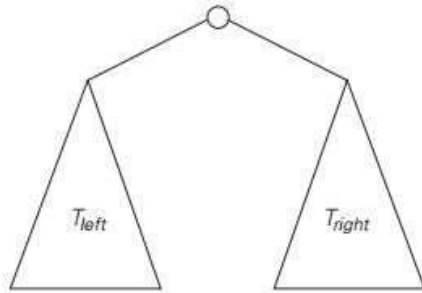


FIGURE 5.4 Standard representation of a binary tree.

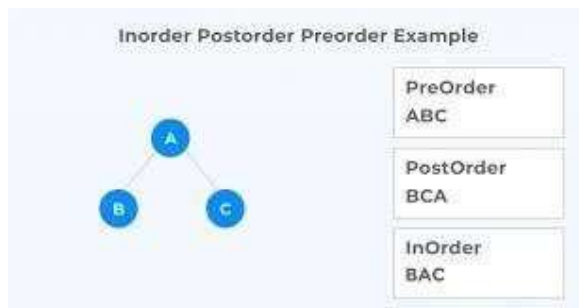
The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder.

All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ only by the timing of the root's visit:

In the **preorder traversal**, the root is visited before the left and right subtrees are visited (in that order).

In the **inorder traversal**, the root is visited after visiting its left subtree but before visiting the right subtree.

In the **postorder traversal**, the root is visited after visiting the left and right subtrees (in that order).



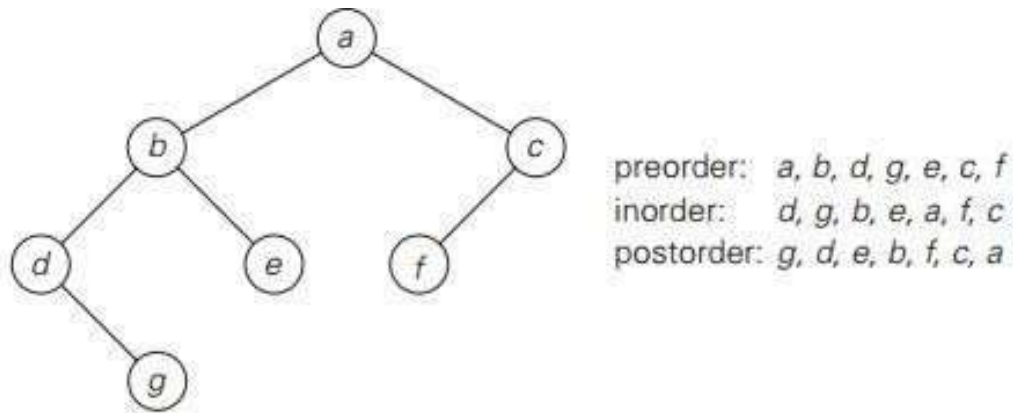


FIGURE 5.6 Binary tree and its traversals.

Preorder traversal

- This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

Until all nodes of the tree are not visited

Step 1 - Visit the root node

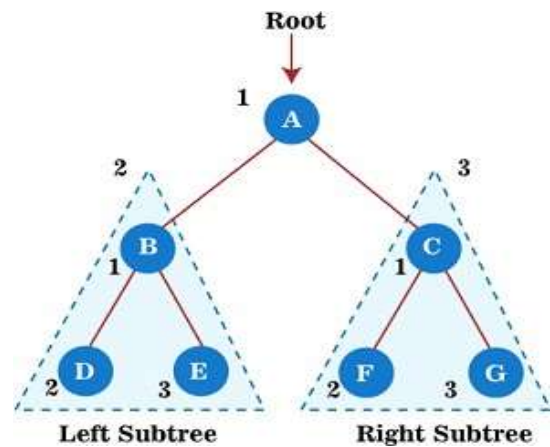
Step 2 - Traverse the left subtree recursively.

Step 3 - Traverse the right subtree recursively.

Algorithm PreorderTraversal(node)

```

{
  if node ≠ null:
    process(node)      // Visit the root node
    PreorderTraversal(node.left) // Recurse on left subtree
    PreorderTraversal(node.right) // Recurse on right subtree
}
  
```



Postorder traversal

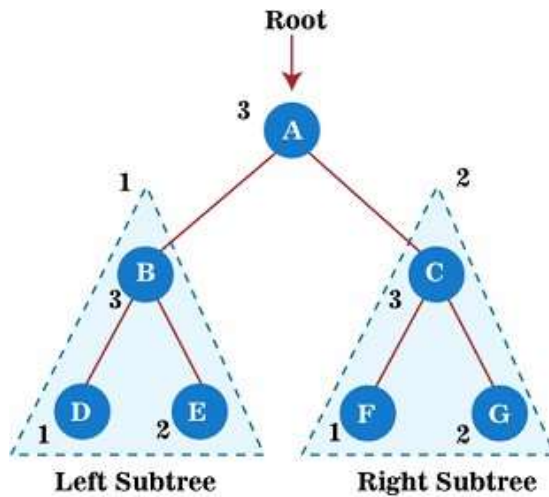
- This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally,

the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

Step 1 - Traverse the left subtree recursively.

Step 2 - Traverse the right subtree recursively.

Step 3 - Visit the root node.



Inorder traversal

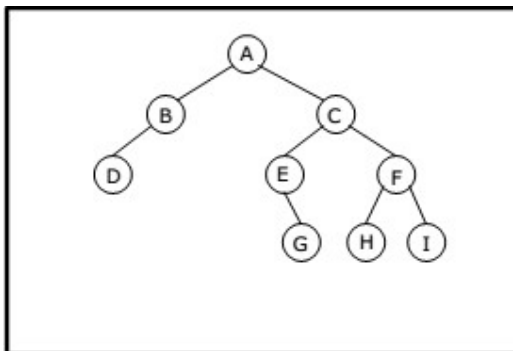
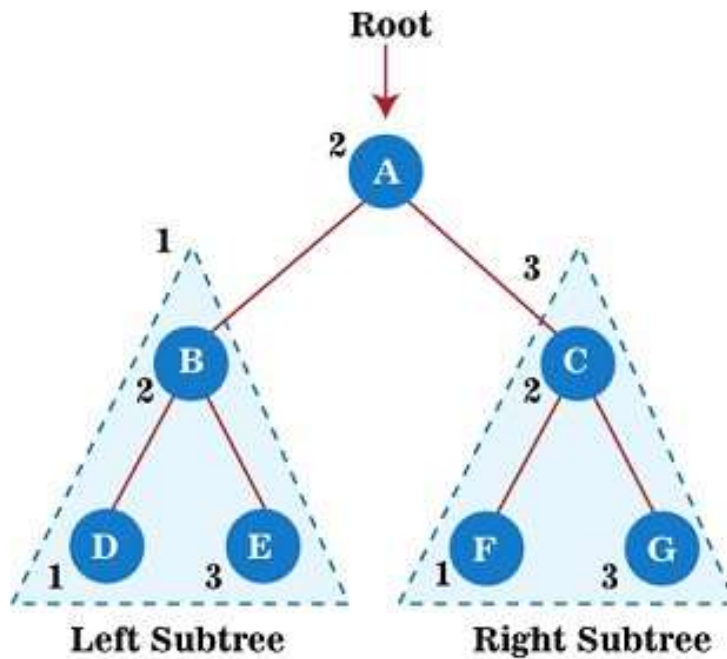
- This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

Step 1 - Traverse the left subtree recursively.

Step 2 - Visit the root node.

Step 3 - Traverse the right subtree recursively.

```
Algorithm InorderTraversal(node):
{
  if node ≠ null:
    InorderTraversal(node.left) // Recurse on left subtree
    process(node)              // Visit the root node
    InorderTraversal(node.right) // Recurse on right subtree
}
```



Binary Tree

- Preorder traversal yields:
A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
D, B, A, E, G, C, H, F, I
- Level order traversal yields:
A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

```

Algorithm PostorderTraversal(node):
{
    if node ≠ null:
        PostorderTraversal(node.left) // Recurse on left subtree
        PostorderTraversal(node.right) // Recurse on right subtree
        process(node)                // Visit the root node
}

```