1.
(a) Derive tight asymptotic complexities for the following algorithms.

Algorithm COUNT 1
Input: n = 2^k, for some positive integer k.
Output: count = number of times Step 4 is executed.
1. count <- 0
2. **while** n ≥ 1
3. **for** j < 1 to n
4. count <- count + 1
5. **end for**
6. n<-n/2
7. **end while**
8. **return** count

Algorithm COUNT 1 Analysis

Algorithm Description:

The algorithm `COUNT 1` takes an input $n = 2^k$ (for some positive integer k) and counts the number of times the inner loop (Step 4) is executed. Here's a breakdown of the algorithm:

1. Initialize `count` to 0.
2. While $n \geq 1$:
   - Iterate from $j = 1$ to n, and for each iteration, increment the `count`.
   - After the inner loop, halve n (i.e., $n = n/2$).
3. Return the final value of `count`.

Step-by-Step Execution:

1. **Initial Setup:**
   - Let $n = 2^k$ where k is a positive integer.
2. **Outer While Loop:**
   - The outer `while` loop continues as long as $n \geq 1$. In each iteration, n is halved. The number of iterations of the `while` loop will be proportional to $\log n$. Specifically, the loop runs $k = \log_2 n$ times, because in each iteration, n is divided by 2.
3. **Inner For Loop:**
   - In each iteration of the `while` loop, the `for` loop runs n times (where n is the current value of n).
   - In the first iteration, when $n = 2^k$, the inner loop runs 2^k times.
   - In the second iteration, when $n = 2^{k-1}$, the inner loop runs 2^{k-1} times.
   - This continues until $n = 1$, when the inner loop runs 1 time.

Total Number of Executions:

The total number of executions of the statement `count = count + 1` is the sum of the number of times the inner loop is executed across all iterations of the `while` loop:

$$T(n) = 2^k + 2^{k-1} + 2^{k-2} + \cdots + 2^1 + 2^0$$

This is a geometric series with the sum:

$$T(n) = 2^{k+1} - 1$$

Since $n = 2^k$, we can rewrite this as:

$$T(n) = 2n - 1$$

Thus, the total number of executions of the inner loop is approximately $2n$.

Time Complexity:

The time complexity of the algorithm is dominated by the total number of times the inner loop executes, which we derived as $T(n) = 2n - 1$. In Big-O notation, this is simplified to:

$$T(n) = O(n)$$

Conclusion:

The tight asymptotic time complexity for the algorithm `COUNT 1` is $O(n)$.

Algorithm COUNT 2
Input: A positive integer n.
Output: *count* = number of times Step 5 is executed.

1. *count* ← 0
2. for $i \leftarrow 1$ **to** n
3. $m \leftarrow$ *floor(n/i)*
4. **for** $j \leftarrow 1$ **to** m
5. *count* ← *count* + 1
6. **end for**
7. **end for**
8. **return** *count*

Algorithm COUNT 2 Analysis

Algorithm Description:

The algorithm `COUNT 2` counts the number of times the statement `count = count + 1` (Step 5) is executed. Here's the breakdown of the algorithm:

1. Initialize `count` to 0.
2. Loop over i from 1 to n.
   - For each value of i, compute $m = \left\lfloor \frac{n}{i} \right\rfloor$.
   - Then, iterate over j from 1 to m, and for each iteration of j, increment `count`.
3. Return the final value of `count`.

Step-by-Step Execution:

1. **Outer For Loop:**
   - The outer loop runs n times, where i goes from 1 to n.
   - For each value of i, the value of $m = \left\lfloor \frac{n}{i} \right\rfloor$ is computed.
2. **Inner For Loop:**
   - The inner loop runs from $j = 1$ to m, where $m = \left\lfloor \frac{n}{i} \right\rfloor$.
   - The number of iterations for each i is approximately $\left\lfloor \frac{n}{i} \right\rfloor$.

3. **Total Count:**
    - To find the total number of times `count = count + 1` is executed, we need to sum up the iterations of the inner loop for all values of i:

$$\text{Total count} = \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor$$

Since $\left\lfloor \frac{n}{i} \right\rfloor$ is approximately $\frac{n}{i}$ for large i, we can approximate the sum as:

$$\text{Total count} \approx \sum_{i=1}^{n} \frac{n}{i}$$

This is a **harmonic sum**, and the sum of the harmonic series up to n is approximately $O(\log n)$. Hence, the total number of executions of the inner loop is:

$$T(n) = n \cdot \sum_{i=1}^{n} \frac{1}{i} \approx n \cdot \log n$$

Time Complexity:

The time complexity of the algorithm is proportional to the total number of executions of the inner loop, which is $O(n \log n)$.

Conclusion:

The tight asymptotic time complexity for the algorithm `COUNT 2` is $O(n \log n)$.

Algorithm COUNT 3
Input: $n = (2^2)^k$, for some positive integer k.
Output: Number of times Step 6 is executed.

```
1. count ← 0
2. for i ← 1 to n
3. j ← 2
4.       while j ≤ n
5.       j ← j^2
6.       count ← count + 1
7.       end while
8. end for
9. return count
```

Step-by-step Analysis:

1. Input Relation:

- Given $n = (2^2)^k = 4^k$, we can express n in terms of k. Specifically, $k = \log_4 n$. This will help us understand the relationship between n and k when analyzing the loops.

2. Outer Loop (Step 2 - Step 3):

- The outer loop runs from $i = 1$ to $i = n$, meaning it will execute **n** times.
- In this case, $n = 4^k$, so the outer loop will run 4^k times.

3. Inner Loop (Step 3 - Step 6):

- The inner loop is a `while` loop with the condition $j \leq n$. Initially, $j = 2$, and the loop executes as long as $j \leq n$.
- The value of j doubles each time (since $j = j^2$, i.e., j is updated as $j = j^2$).
  - Initially, $j = 2$, then $j = 4$, $j = 16$, $j = 256$, and so on.
  - The number of times the inner loop runs is determined by the number of steps it takes for j to exceed n. This follows the relation:

$$j = 2, 4, 16, 256, \ldots$$

  - Each step corresponds to squaring the previous value, so the number of iterations is logarithmic in terms of n. Specifically, the loop runs $\log_2(\log_2 n)$ times (because each iteration squares j).

4. **Count Updates (Step 6):**

- The inner loop executes $\log_2(\log_2 n)$ times per outer loop iteration, and the `count` is incremented each time the inner loop executes.

5. **Total Complexity:**

- The outer loop runs $n = 4^k$ times, and for each iteration of the outer loop, the inner loop runs $\log_2(\log_2 n)$ times.
- Therefore, the total number of times the `count` variable is updated is:

$$\text{Total updates} = n \times \log_2(\log_2 n)$$

  - Since $n = 4^k$, we can express the total time complexity as:

$$O(n \log_2(\log_2 n)) = O(4^k \log_2(\log_2(4^k)))$$

  - Since $\log_2(4^k) = 2k$, this simplifies to:

$$O(4^k \log_2(2k))$$

  - Since $4^k = n$, this results in:

$$O(n \log_2(\log_2 n))$$

Final Result:

The tight asymptotic complexity of the given algorithm is:

$$O(n \log_2(\log_2 n))$$

(b) Show that the number of element comparisons to MERGE two sorted arrays of sizes n_1 and n_2 respectively where $n_1 \leq n_2$ into one sorted array of size $n = n_1 + n_2$ is between n_1 and $(n - 1)$. Also show that if the two array sizes are floor (n/2) and ceil (n/2) then the number of comparisons needed is between floor (n/2) and $(n - 1)$.

Problem:

We are asked to show that the number of element comparisons required to merge two sorted arrays of sizes n_1 and n_2 (where $n_1 \leq n_2$) into a single sorted array of size $n = n_1 + n_2$ is between n_1 and $n - 1$. Additionally, we are required to show that if the array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons needed is between $\lfloor n/2 \rfloor$ and $n - 1$.

Part 1: General Case for Arrays of Sizes n_1 and n_2

Setup:

Let the two sorted arrays be:

- Array A of size n_1
- Array B of size n_2 where $n_1 \leq n_2$

We want to merge these two arrays into a single sorted array of size $n = n_1 + n_2$.

The merging process works as follows:

1. We compare the smallest elements of both arrays (i.e., the first elements of each array).
2. We choose the smaller of the two elements and place it into the merged array.
3. We repeat this process, comparing the next smallest element of either array, until one of the arrays is exhausted.
4. Once one array is exhausted, the remaining elements from the other array are added directly to the merged array (since they are already sorted).

Number of Comparisons:

The number of comparisons made during the merge process is determined by the number of times we need to compare elements between the two arrays.

- In the worst case, we will compare elements until one of the arrays is exhausted. This happens when we compare up to n_1 times (because once we have selected all the elements from the smaller array, no more comparisons are needed).

Thus, the number of comparisons is between n_1 (in the case where all elements from array A are compared with elements in array B) and $n - 1$ (the maximum number of comparisons, which occurs when we compare elements at each step and exhaust one of the arrays).

Therefore, the number of comparisons is between:

$$n_1 \leq \text{comparisons} \leq n - 1$$

Part 2: Special Case for Arrays of Sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$

Now, let's consider the case where the array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, which are approximately half of the total size n.

- Let $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$.
- We want to merge these two arrays into a single array of size n.

Number of Comparisons:

The same reasoning applies here as in the general case. We compare the smallest elements of the two arrays and continue comparing until one of the arrays is exhausted.

- Since the sizes of the two arrays are approximately $n/2$, we will make up to $\lfloor n/2 \rfloor$ comparisons before exhausting one of the arrays (in the worst case). Once one array is exhausted, the remaining elements from the other array are directly copied into the merged array.

Thus, the number of comparisons will be between $\lfloor n/2 \rfloor$ (the minimum number of comparisons) and $n - 1$ (the maximum number of comparisons, which occurs when all but one element from one array are compared).

Therefore, the number of comparisons is between:

$$\lfloor n/2 \rfloor \leq \text{comparisons} \leq n - 1$$

Conclusion:

- In the general case where the two arrays have sizes n_1 and n_2, the number of comparisons needed to merge them is between n_1 and $n - 1$.
- In the special case where the array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons needed is between $\lfloor n/2 \rfloor$ and $n - 1$.

(c) What is amortized analysis? Explain by citing appropriate examples.

Amortized Analysis:

Amortized analysis is a technique used to analyze the average time complexity of operations in a sequence of operations, rather than analyzing each operation individually. It provides a way to ensure that even if some operations take a long time, the average time per operation over a sequence is still efficient. In other words, amortized analysis gives us the *average cost per operation* in the worst case, considering all operations in a sequence.

Amortized analysis is particularly useful when some operations have a high cost, but these operations occur infrequently, while most operations are cheap. By averaging over a series of operations, we get a better understanding of the overall performance of an algorithm.

Key Concepts in Amortized Analysis:

1. **Total Cost vs. Individual Cost:**
    - In amortized analysis, we look at the total cost of performing a sequence of operations rather than the cost of a single operation.
    - For example, even if one operation might be costly, we distribute that cost across the sequence of operations to get an average cost per operation.
2. **Amortized Cost vs. Actual Cost:**
    - **Actual cost** refers to the real time taken by a single operation.
    - **Amortized cost** refers to the average cost per operation over a sequence of operations. The amortized cost is often less than or equal to the actual cost of any individual operation.
3. **Methods of Amortized Analysis:**
    - **Aggregate Method:** This method involves calculating the total cost of n operations and then dividing that by n to find the average cost per operation.
    - **Accounting Method:** This method assigns different "charges" (credits or debits) to different operations to ensure that expensive operations are paid for by cheaper operations.
    - **Potential Method:** This method uses a potential function to track the "stored work" in a data structure, which is then used to analyze the cost of operations.

1. Dynamic Array Resizing:

Consider the case of dynamically resizing an array when it exceeds its capacity. Initially, the array has a fixed size, say 1, and every time we insert an element, we check if the array has reached its capacity. If it has, we double the array size. The insertion itself is typically an $O(1)$ operation, but when the array is resized, it requires copying all elements to the new array, which takes $O(n)$ time, where n is the current size of the array.

- **Worst-Case Cost:** If we only look at the individual insertion that triggers a resize, it takes $O(n)$ time (because we need to copy n elements).
- **Amortized Cost:** However, if we perform multiple insertions, the doubling of the array happens only logarithmically in terms of the number of insertions. The total time spent on resizing over n insertions is proportional to $O(n)$, and the amortized cost per insertion is $O(1)$, because the total cost divided by n is still constant.

Thus, the amortized cost per insertion is $O(1)$, despite occasional expensive resizes.

2. **Incrementing a Binary Counter:**

Consider a binary counter, where each bit starts at 0. When we increment the counter, we add 1 to the least significant bit. If the bit is 1, it becomes 0, and the carry is propagated to the next bit. This continues until a 0 is encountered, which is flipped to 1. The number of bits flipped in each increment depends on how many consecutive 1's appear in the binary representation of the number.

- **Worst-Case Cost:** In the worst case, if the binary number consists of all 1's, the increment will flip all the bits, which takes $O(k)$ time, where k is the number of bits.
- **Amortized Cost:** However, the total number of bits flipped over a sequence of n increments is proportional to $O(n)$, because each bit in the counter is flipped only once. The amortized cost per increment is $O(1)$, because over n increments, the total number of bit flips is $O(n)$, and the cost per increment is constant.

Thus, even though some increments may involve flipping multiple bits, the amortized cost is constant $O(1)$.

3. **Binary Search Tree (BST) Rebalancing:**

In a binary search tree, certain operations (like insertion or deletion) may require rebalancing the tree (e.g., through rotations in an AVL tree or red-black tree). Rebalancing can be an expensive operation, taking $O(\log n)$ time for a single operation. However, in many cases, these rebalancing operations occur infrequently, and their cost can be amortized over a series of insertions and deletions.

- **Worst-Case Cost:** The cost of a single insertion or deletion could involve a rebalancing that takes $O(\log n)$.
- **Amortized Cost:** If rebalancing happens infrequently, say after every few operations, the amortized cost for each operation is still $O(\log n)$ when averaged over a sequence of operations. Even if a rebalancing happens occasionally, the amortized cost of each operation remains logarithmic.

Conclusion:

Amortized analysis helps us understand the average cost of operations over time, especially in cases where some operations may be expensive, but others are cheap. By using amortized analysis, we can ensure that the overall performance of an algorithm or data structure is efficient in the long run, even if there are occasional "expensive" operations. This is particularly useful in dynamic data structures like dynamic arrays, binary search trees, and other structures that involve occasional resizing or rebalancing.

(d) Explain the different techniques by which graphs are represented in computer memory.

Techniques for Representing Graphs in Computer Memory

Graphs are essential data structures in computer science used to model relationships between objects. In computer memory, graphs can be represented in several ways, depending on the nature of the graph (such as whether it's sparse or dense) and the type of operations required

(such as traversal, searching, etc.). The main techniques used for representing graphs in memory are:

1. Adjacency Matrix

An adjacency matrix is a 2D array that represents the edges between vertices in a graph. It is especially useful for dense graphs, where the number of edges is close to the maximum number of edges possible.

- **Representation:**
  - A graph with V vertices is represented by a $V \times V$ matrix, where each element $A[i][j]$ is:
    - **1** if there is an edge from vertex i to vertex j
    - **0** if there is no edge from vertex i to vertex j
  - For **directed graphs**, $A[i][j] = 1$ means there is a directed edge from i to j.
  - For **undirected graphs**, the matrix is symmetric, i.e., $A[i][j] = A[j][i]$.
- **Space Complexity:** $O(V^2)$ where V is the number of vertices.
- **Advantages:**
  - Fast to check if an edge exists between two vertices: $O(1)$ time.
  - Simple to implement.
- **Disadvantages:**
  - Inefficient for sparse graphs (graphs with few edges), as it wastes memory by using $O(V^2)$ space even if there are only a few edges.
  - Inserting/removing edges can be costly if done frequently.

2. **Adjacency List**

An adjacency list is a more space-efficient way to represent a graph, particularly for sparse graphs. It uses a collection of lists or arrays, where each vertex has a list of its adjacent vertices (i.e., vertices connected by edges).

- **Representation:**
  - The graph is represented as an array of V elements (where V is the number of vertices). Each element of the array corresponds to a vertex and points to a list (or linked list) of the adjacent vertices.
  - For **directed graphs**, if there is a directed edge from vertex u to vertex v, then vertex v will appear in the adjacency list of vertex u.
  - For **undirected graphs**, the edge from u to v will appear in both the adjacency list of vertex u and vertex v.
- **Space Complexity:** $O(V + E)$ where E is the number of edges.
- **Advantages:**
  - More space-efficient than the adjacency matrix, especially for sparse graphs.
  - Efficient for storing graphs where the number of edges is much smaller than the square of the number of vertices.
  - Easy to traverse all edges incident to a vertex.
- **Disadvantages:**
  - Checking whether a specific edge exists between two vertices can take $O(V)$ time in the worst case (if the list is long).
  - Less efficient for dense graphs, where the number of edges is close to V^2.

3. **Edge List**

An edge list is a simple way to represent a graph where each edge is stored as a pair (or tuple) of vertices. This representation is particularly useful when dealing with graph algorithms that focus on edges rather than vertices, such as algorithms for finding the minimum spanning tree.

- **Representation:**
  - Each edge in the graph is represented as a pair (u, v), indicating an edge between vertex u and vertex v.
  - For **directed graphs**, the pair (u, v) means a directed edge from u to v.
  - For **undirected graphs**, the edge is typically represented as an unordered pair (u, v), implying an undirected edge.
- **Space Complexity:** $O(E)$, where E is the number of edges.
- **Advantages:**
  - Simple and compact representation of the graph.
  - Efficient for algorithms that operate primarily on edges, such as finding a minimum spanning tree or performing a depth-first search on all edges.
- **Disadvantages:**
  - Not efficient for tasks requiring frequent access to the neighbors of a vertex, since the entire list must be scanned to find neighbors of a vertex.
  - Checking whether an edge exists between two specific vertices is not efficient (requires $O(E)$ time).

4. Incidence Matrix

An incidence matrix is another way to represent a graph, where the rows represent vertices, and the columns represent edges. Each element in the matrix indicates whether a vertex is incident (connected) to a given edge.

- **Representation:**
  - For a graph with V vertices and E edges, an incidence matrix is a $V \times E$ matrix.
  - If vertex i is incident to edge j, the element $M[i][j]$ is marked (usually with a 1 or -1), indicating that vertex i is part of edge j. For undirected graphs, $M[i][j]$ will be 1 for both ends of an edge. For directed graphs, one entry might be $+1$ and the other -1 depending on the direction of the edge.
- **Space Complexity:** $O(V \times E)$
- **Advantages:**
  - Can be used to represent both directed and undirected graphs in a uniform way.
  - Easy to identify the edges incident to a vertex.
- **Disadvantages:**
  - Not as space-efficient as adjacency lists for sparse graphs.
  - Inefficient for dense graphs, since it uses $O(V \times E)$ space.

5. **Compressed Adjacency List (or CSR - Compressed Sparse Row)**

This is an optimization of the adjacency list that saves space by storing the adjacency lists in a compressed form. It is commonly used in sparse graph representations.

- **Representation:**
  - Instead of storing lists for each vertex, we store all the edges in an array and use two additional arrays:
    1. **Vertex Array**: This array stores the index where the adjacency list of each vertex starts.
    2. **Edge Array**: This array stores all the edges of the graph, where each entry corresponds to a vertex connected to a given vertex.
- **Space Complexity:** $O(V + E)$, similar to the adjacency list but with potentially lower space usage due to compression.
- **Advantages:**
  - More space-efficient than the adjacency list for sparse graphs.
  - Suitable for efficient access to edge data, as we only need to refer to the compressed arrays.
- **Disadvantages:**
  - Slightly more complex to implement than simple adjacency lists.

Conclusion:

The choice of graph representation depends on the properties of the graph (dense or sparse), the types of operations needed (e.g., edge lookups, vertex traversals), and the available memory. Here's a quick summary of when each representation is most useful:

- **Adjacency Matrix**: Best for dense graphs and quick edge existence checks.
- **Adjacency List**: Best for sparse graphs and efficient space usage.
- **Edge List**: Best for algorithms focusing on edges or graphs where edges are dynamically manipulated.
- **Incidence Matrix**: Useful for specific applications requiring a direct relationship between vertices and edges.
- **Compressed Adjacency List**: Ideal for sparse graphs with a need for compact storage.

---

2. (a) Choose an appropriate data structure for representation of disjoint sets. Write efficient FIND and
UNION methods over the chosen data structure. Derive their time complexities.

Data Structure for Representing Disjoint Sets: **Disjoint Set (Union-Find) with Path Compression and Union by Rank**

The **Disjoint Set** (or **Union-Find**) is a data structure that efficiently supports two operations:

1. **FIND**: Determine the set (or group) to which an element belongs.
2. **UNION**: Merge two sets into a single set.

A disjoint set is typically used to keep track of a collection of non-overlapping sets, and it is highly useful in applications like network connectivity, Kruskal's algorithm for finding minimum spanning trees, and connected components in a graph.

1. Choosing the Data Structure:

To represent disjoint sets efficiently, we use an array (or list) for each element. Specifically:

- **Parent Array**: This array stores the "parent" of each element. Initially, each element is its own parent (i.e., every element is its own set).
- **Rank Array (or Size Array)**: This array is used to store the rank (or size) of each set. The rank is used to optimize the union operation by ensuring that smaller trees are always attached under larger trees, preventing the tree from becoming too tall.

2. Operations:

FIND Operation (with Path Compression):

The **FIND** operation is used to find the representative (or "root") of the set containing a given element. Path compression is used to flatten the tree structure so that all nodes directly point to the root, improving the efficiency of future operations.

FIND(x):

1. If x is not the root (i.e., $parent[x] \neq x$), recursively call **FIND** on the parent of x and set the parent of x to be the root.
2. If x is the root (i.e., $parent[x] = x$), return x.

Path Compression: This step ensures that all nodes along the path from x to the root directly point to the root after the operation.

UNION Operation (with Union by Rank):

The **UNION** operation merges two sets containing elements x and y. We use the **union by rank** technique to keep the tree shallow by attaching the smaller tree under the root of the larger tree (based on rank).

UNION(x, y):

1. Perform **FIND(x)** and **FIND(y)** to get the roots r_x and r_y of the sets containing x and y.
2. If r_x and r_y are the same, no action is required (they are already in the same set).
3. Otherwise, attach the tree with the smaller rank under the tree with the larger rank:
   - If the rank of r_x is greater than the rank of r_y, set $parent[r_y] = r_x$.
   - If the rank of r_y is greater than the rank of r_x, set $parent[r_x] = r_y$.
   - If the ranks are equal, arbitrarily make one root the parent of the other, and increase the rank of the new root by 1.

3. Time Complexities:

The **FIND** and **UNION** operations with **path compression** and **union by rank** are highly efficient. The time complexity for each operation is nearly constant.

FIND Time Complexity (with Path Compression):

- The time complexity for a single **FIND** operation with path compression is $O(\alpha(n))$, where $\alpha(n)$ is the **inverse Ackermann function**.
- $\alpha(n)$ grows extremely slowly, so for all practical purposes, it is considered to be a very small constant (even for extremely large n, $\alpha(n)$ is at most 4).
- Hence, the **FIND** operation is **almost constant time**, i.e., $O(\alpha(n))$.

UNION Time Complexity (with Union by Rank):

- The time complexity for a single **UNION** operation with union by rank is also $O(\alpha(n))$, because each **UNION** operation involves performing two **FIND** operations, and each **FIND** operation takes $O(\alpha(n))$ time.
- Hence, the **UNION** operation is also **almost constant time**, i.e., $O(\alpha(n))$.

4. Analysis of Combined Operations:

When we perform a sequence of m **UNION** and **FIND** operations, the amortized time complexity for each operation is $O(\alpha(n))$. Thus, for m operations, the total time complexity is $O(m \cdot \alpha(n))$.

In practice, because $\alpha(n)$ grows extremely slowly, the total time for m operations is effectively linear: $O(m)$, making the Union-Find structure highly efficient even for large data sets.

5. Summary of Operations:

- **FIND(x)**: Returns the root of the set containing x, with path compression. Time complexity: $O(\alpha(n))$.
- **UNION(x, y)**: Merges the sets containing x and y, using union by rank. Time complexity: $O(\alpha(n))$.

Example Code (Python):

```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))  # Initialize parent
array
        self.rank = [0] * n           # Initialize rank array

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path
compression
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            # Union by rank
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1
```

Conclusion:

The **Disjoint Set** (Union-Find) data structure with **path compression** and **union by rank** is an efficient way to manage disjoint sets. The **FIND** and **UNION** operations have an amortized time complexity of $O(\alpha(n))$, which is almost constant for practical purposes. This makes the Union-Find data structure highly efficient for applications involving dynamic connectivity and related problems.

(b) Write and explain the ordered sequential search algorithm. Derive the time complexity of the method.

Ordered Sequential Search Algorithm

The **Ordered Sequential Search** (also called **Linear Search**) is a search algorithm used to find the position of a target element in a **sorted** list or array. The idea behind ordered sequential search is to examine each element of the array in order until the target element is found or the end of the array is reached.

In **ordered sequential search**, the key advantage is that the array is sorted, meaning that we can terminate the search early if we encounter an element that is greater than the target (for ascending order).

Algorithm Explanation:

Ordered Sequential Search Algorithm:

Given:

- A sorted array $A[1..n]$ of size n,
- A target element T to search for.

Steps:

1. Start from the first element $A[1]$.
2. Compare $A[i]$ (the current element) with the target T.
   - If $A[i] = T$, then return the index i (the target is found).
   - If $A[i] > T$, stop the search and return "not found" (since the array is sorted, all subsequent elements will be greater than T).
3. If $A[i] \neq T$, continue to the next element $A[i + 1]$.
4. Repeat steps 2 and 3 until either:
   - The target is found (return the index of the element).
   - The end of the array is reached, in which case the target is not in the array, so return "not found".

Pseudocode:

```
def ordered_sequential_search(A, n, T):
    for i in range(n):
        if A[i] == T:
            return i  # Target found at index i
        if A[i] > T:
            break  # No need to search further, since the
array is sorted
    return -1  # Target not found
```

Time Complexity:

Best Case:

- The best case occurs when the first element of the array is equal to the target, $A[1] = T$. In this case, the algorithm finds the target in the first comparison, so the time complexity is $O(1)$.

Worst Case:

- The worst case happens when the target is either:
  - Not in the array, in which case the algorithm must check all n elements.
  - Or the target is the last element, so the algorithm needs to check all n elements to find it.
- In both cases, the time complexity is $O(n)$ because the algorithm may have to examine all elements in the array.

Average Case:

- On average, the target will be found after checking about half of the elements in the array. Hence, the average number of comparisons is approximately $n/2$. The time complexity for the average case is also $O(n)$, since constant factors are ignored in big-O notation.

Conclusion:

- **Best Case Time Complexity:** $O(1)$
- **Worst Case Time Complexity:** $O(n)$
- **Average Case Time Complexity:** $O(n)$

Explanation of Time Complexity:

- **Best case** happens when the target is found at the first element, and the algorithm stops immediately.
- **Worst case** occurs when the target is either absent from the array or at the very end, so the algorithm has to examine every element.
- **Average case** assumes that the target is randomly distributed in the array, and on average, we would expect to find the target after about $n/2$ comparisons. Since constant factors are ignored in big-O notation, the average case is still $O(n)$.

Thus, while **ordered sequential search** can be fast in the best case, its time complexity is linear in the worst and average cases, which makes it inefficient for large datasets compared to more advanced search algorithms (like binary search, which has $O(\log n)$ complexity).

---

3. Write and explain two graph traversal techniques. Derive the time and space complexities of the

traversal methods considering the data structures used.

Graph Traversal Techniques: **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**

Graph traversal refers to the process of visiting all the vertices of a graph in a systematic way. There are two primary techniques used for graph traversal: **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. These methods are fundamental in graph algorithms and are used to explore the structure of graphs.

1. Breadth-First Search (BFS)

BFS is a traversal technique that explores all the vertices of a graph level by level. It starts at a given vertex and explores all its neighbors before moving on to the next level of neighbors.

Algorithm Explanation:

1. **Initialize**:
   - Mark all vertices as unvisited.
   - Choose a starting vertex and mark it as visited.
   - Enqueue the starting vertex into a queue.
2. **Process**:
   - While the queue is not empty, dequeue a vertex u from the front of the queue.
   - For each unvisited neighbor v of vertex u, mark v as visited and enqueue v.
3. **Repeat**: Continue this process until all reachable vertices from the starting vertex have been visited.

Pseudocode for BFS:

```
def BFS(graph, start):
    visited = set()  # Set to track visited nodes
    queue = []  # Queue for BFS
    visited.add(start)  # Mark the start node as visited
    queue.append(start)  # Enqueue the start node

    while queue:
        node = queue.pop(0)  # Dequeue the first node
        print(node, end=" ")  # Visit the node

        # Visit all the unvisited neighbors
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Time and Space Complexity of BFS:

- **Time Complexity:**
  - **O(V + E)** where V is the number of vertices and E is the number of edges.
    - **O(V)**: Every vertex is enqueued and dequeued exactly once.
    - **O(E)**: Every edge is checked exactly once while exploring neighbors.
- **Space Complexity:**
  - **O(V)** for storing the visited set and the queue. In the worst case, the queue may hold all vertices if they are all connected, and the visited set must store all V vertices.

2. Depth-First Search (DFS)

DFS is a traversal technique that explores a graph by following a path from the starting vertex until it reaches a vertex with no unvisited neighbors. Then it backtracks and explores other unvisited neighbors.

Algorithm Explanation:

1. **Initialize:**
   - Mark all vertices as unvisited.
   - Choose a starting vertex and mark it as visited.
2. **Process:**
   - Recursively visit the unvisited neighbors of the current vertex.
   - Each time a vertex is visited, mark it as visited and continue the traversal for its unvisited neighbors.
   - Backtrack when a vertex has no unvisited neighbors.
3. **Repeat:** Continue this process until all reachable vertices from the starting vertex have been visited.

Pseudocode for DFS:

```
def DFS(graph, node, visited=None):
    if visited is None:
        visited = set()  # Set to track visited nodes

    visited.add(node)  # Mark the current node as visited
    print(node, end=" ")  # Visit the node

    # Recursively visit all unvisited neighbors
    for neighbor in graph[node]:
        if neighbor not in visited:
            DFS(graph, neighbor, visited)
```

Time and Space Complexity of DFS:

- **Time Complexity:**
  - **O(V + E)** where V is the number of vertices and E is the number of edges.
    - **O(V)**: Every vertex is visited exactly once.
    - **O(E)**: Every edge is checked exactly once while exploring neighbors.
- **Space Complexity:**
  - **O(V)** for storing the visited set and the recursive call stack.
    - The recursive call stack can go as deep as the number of vertices in the graph (in case of a deep graph with no branching), so the space complexity in the worst case is $O(V)$.

Comparison Between BFS and DFS:

Aspect	Breadth-First Search (BFS)	Depth-First Search (DFS)
Traversal Method	Level-by-level, explores all neighbors first	Explores deeper along a branch before backtracking
Use Case	Shortest path in unweighted graphs, level-order traversal	Pathfinding, cycle detection, topological sorting
Time Complexity	$O(V + E)$	$O(V + E)$
Space Complexity	$O(V)$	$O(V)$ (for recursion and visited set)
Data Structure Used	Queue	Stack (recursive call stack)

Key Differences:

- **BFS** explores the graph layer by layer, ensuring the shortest path (in terms of edge count) to any vertex is found first. It uses a queue to manage the order of exploration.
- **DFS** explores as deeply as possible along one branch before backtracking. It can be implemented using recursion (call stack) or an explicit stack. DFS is particularly useful in problems like topological sorting and detecting cycles.

Summary:

- **BFS** is ideal for situations where we need to find the shortest path or explore vertices level by level.
- **DFS** is suitable for problems where we want to explore a complete branch before backtracking, and it can be more memory-efficient than BFS in some scenarios (since it may use less space if the graph has a high branching factor).

Both BFS and DFS have time complexities of $O(V + E)$, but their space complexities differ due to the different data structures used (queue for BFS and stack for DFS).

---

4. Explain the divide and conquer algorithm design technique. Write and explain the Quick Sort algorithm.
Derive the worst case, average case and best case time and space complexities of the above method.

Divide and Conquer Algorithm Design Technique

Divide and Conquer is a fundamental algorithm design paradigm that involves breaking down a problem into smaller, more manageable subproblems, solving each of these subproblems independently, and then combining their results to solve the original problem. This technique is often used for problems that exhibit optimal substructure, where the solution to the problem can be constructed from solutions to smaller instances of the same problem.

The general steps of a **Divide and Conquer** algorithm are:

1. **Divide**: Break the problem into two or more smaller subproblems that are similar to the original problem.
2. **Conquer**: Recursively solve the subproblems. If the subproblem is small enough, solve it directly.
3. **Combine**: Combine the solutions of the subproblems to get the solution to the original problem.

Quick Sort Algorithm

Quick Sort is a **Divide and Conquer** algorithm for sorting an array or list. It works by selecting a "pivot" element from the array and partitioning the other elements into two subarrays, according to whether they are smaller or greater than the pivot. The subarrays are then sorted recursively.

Steps of the Quick Sort Algorithm:

1. **Choose a Pivot**: Select an element from the array as the pivot. Different strategies for selecting the pivot can be used (first element, last element, random element, median of the array, etc.). The performance of the algorithm can vary depending on the pivot selection strategy.
2. **Partition the Array**: Rearrange the elements in the array such that:
   - Elements less than or equal to the pivot are on the left side of the pivot.
   - Elements greater than the pivot are on the right side of the pivot.
3. **Recursively Apply the Above Steps**: Apply the same algorithm to the left and right subarrays (excluding the pivot) until the subarrays have only one element or are empty.
4. **Combine**: No combining step is needed since the elements are sorted in place.

Pseudocode for Quick Sort:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]  # Pivot can be chosen in many
ways
```

```
        left = [x for x in arr if x < pivot]  # Elements less than pivot
        middle = [x for x in arr if x == pivot]  # Elements equal to pivot
        right = [x for x in arr if x > pivot]  # Elements greater than pivot
        return quick_sort(left) + middle + quick_sort(right)
```

Alternatively, Quick Sort can be done in-place by modifying the array directly using a partition function.

Time Complexity of Quick Sort:

- **Best Case:**
  - Occurs when the pivot divides the array into two roughly equal halves.
  - In this case, the array is split in half at each step, and the total work is proportional to the number of elements. The recurrence relation is:
    $$T(n) = 2T(n/2) + O(n)$$
    Solving this using the master theorem or recursion tree gives:
    $$T(n) = O(n \log n)$$
    Therefore, the **best case time complexity** is $O(n \log n)$.
- **Average Case:**
  - In most cases, the pivot divides the array in a reasonably balanced way. On average, Quick Sort performs similarly to the best case.
  - The recurrence relation for average case is also:
    $$T(n) = 2T(n/2) + O(n)$$
    which results in:
    $$T(n) = O(n \log n)$$
    So, the **average case time complexity** is $O(n \log n)$.

- **Worst Case**:
  - The worst case occurs when the pivot selection is poor, such as when the pivot is the smallest or largest element in the array, leading to unbalanced partitions. In this case, one subarray will have $n - 1$ elements and the other will have 0 elements.
  - The recurrence relation for this case is:
    $$T(n) = T(n-1) + O(n)$$
    Solving this gives:
    $$T(n) = O(n^2)$$
    Therefore, the **worst case time complexity** is $O(n^2)$.

Space Complexity of Quick Sort:

Quick Sort's space complexity is determined by the depth of the recursion tree and the space required for the partitioning process.

- **In-place partitioning** (if done in-place, using pointers to swap elements) means no extra space is needed to store subarrays.
- The recursion stack, however, can go as deep as the height of the tree formed by the recursive calls.
- **Worst Case Space Complexity**:
  - In the worst case (when the array is highly unbalanced), the recursion depth could be $O(n)$, leading to a space complexity of $O(n)$.
- **Best/Average Case Space Complexity**:
  - In the best and average cases, the recursion depth is $O(\log n)$, so the space complexity is $O(\log n)$.

Thus, the **space complexity** is:

- **Best/Average Case Space Complexity:** $O(\log n)$
- **Worst Case Space Complexity:** $O(n)$

Summary of Time and Space Complexities:

Case	Time Complexity	Space Complexity
Best Case	$O(n \log n)$	$O(\log n)$
Average Case	$O(n \log n)$	$O(\log n)$
Worst Case	$O(n^2)$	$O(n)$

Key Points:

1. **Quick Sort** is a **divide and conquer** algorithm, and it is efficient on average for large datasets due to its $O(n \log n)$ average time complexity.
2. The worst case $O(n^2)$ time complexity can be avoided by using better pivot selection methods (such as random pivoting or choosing the median of three).
3. The **in-place partitioning** allows Quick Sort to have a low space complexity compared to algorithms like Merge Sort, which requires additional space for merging.
4. Despite the worst-case time complexity of $O(n^2)$, Quick Sort is often used in practice because of its fast average-case performance and low space overhead.

---

5. (a) Define and explain the significance of Big-Oh (O), Big-Theta (Θ) and Big Omega (Ω) asymptotic notations in connection to the time complexity analysis of algorithms.

Asymptotic Notations: Big-Oh (O), Big-Theta (Θ), and Big-Omega (Ω)

Asymptotic notations are mathematical tools used to describe the behavior of algorithms in terms of time and space complexity, particularly when the input size grows large. These notations help us understand the efficiency and performance of algorithms in different scenarios, such as best, worst, and average cases.

The most commonly used asymptotic notations are **Big-Oh (O)**, **Big-Theta (Θ)**, and **Big-Omega (Ω)**. Each of these notations provides a different view of the algorithm's time complexity.

1. Big-Oh Notation (O)

Big-Oh (O) notation is used to describe the **upper bound** of an algorithm's time complexity. It represents the worst-case scenario, which means it gives an upper limit on the running time of an algorithm for large inputs.

- **Definition:** $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$.
- In simpler terms, Big-Oh notation provides an upper bound, meaning the function $f(n)$ will not grow faster than $g(n)$ beyond a certain point.
- **Significance:** Big-Oh is useful for understanding the **worst-case performance** of an algorithm. For example, if an algorithm's time complexity is $O(n^2)$, it means that the algorithm will take at most n^2 time for large inputs, even in the worst case.
- **Example:**
  - An algorithm with a time complexity of $O(n^2)$ will take at most proportional to the square of the input size in the worst case.

2. Big-Theta Notation (Θ)

Big-Theta (Θ) notation is used to describe the **tight bound** of an algorithm's time complexity. It represents both the upper and lower bounds of the algorithm's running time, meaning it gives an exact asymptotic behavior for large inputs.

- **Definition:** $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that for all $n > n_0$,

  $$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

  In simpler terms, Big-Theta gives both an upper and lower bound, meaning $f(n)$ grows asymptotically as $g(n)$ for large n.
- **Significance:** Big-Theta notation provides a more **precise** understanding of an algorithm's performance. It tells us that for sufficiently large input sizes, the algorithm's running time will always grow at the same rate as the function $g(n)$.
- **Example:**
  - If an algorithm has a time complexity of $\Theta(n \log n)$, it means that the algorithm's running time grows exactly at a rate proportional to $n \log n$, both in the best and worst cases.

3. Big-Omega Notation (Ω)

Big-Omega (Ω) notation is used to describe the **lower bound** of an algorithm's time complexity. It represents the best-case scenario, which means it gives the minimum time the algorithm will take for large inputs.

- **Definition**: $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that for all $n > n_0$,

  $$f(n) \geq c \cdot g(n)$$

  In simpler terms, Big-Omega provides a lower bound, meaning that the algorithm will take at least $g(n)$ time for sufficiently large n.

- **Significance**: Big-Omega is useful for understanding the **best-case performance** of an algorithm. It tells us the minimum time the algorithm will take in the most favorable scenario, or in the best case.

- **Example**:
  - If an algorithm has a time complexity of $\Omega(n)$, it means that the algorithm will take at least linear time in the best case.

Summary of Big-Oh (O), Big-Theta (Θ), and Big-Omega (Ω) Notations:

Notation	Definition	Significance	Example
Big-Oh (O)	Upper bound (worst-case). $f(n) \leq c \cdot g(n)$ for large n	Describes the worst-case time complexity	$O(n^2)$
Big-Theta (Θ)	Tight bound (exact asymptotic behavior). $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for large n	Describes both upper and lower bounds, providing exact asymptotic behavior	$\Theta(n \log n)$
Big-Omega (Ω)	Lower bound (best-case). $f(n) \geq c \cdot g(n)$ for large n	Describes the best-case time complexity	$\Omega(n)$

Significance in Algorithm Analysis:

- **Big-Oh** notation helps in analyzing the **upper bound**, or the worst-case scenario, which is often most critical for understanding algorithm efficiency under heavy load or worst conditions.
- **Big-Theta** notation gives a **tight bound**, which is more useful for understanding the exact time complexity, especially in well-behaved algorithms.
- **Big-Omega** notation is important for knowing the **minimum time complexity**, particularly in the best case where we might want to measure the algorithm's performance under optimal conditions.

By using these asymptotic notations, we can more accurately compare the efficiency of different algorithms and choose the most appropriate one based on the problem's constraints.

(b) Show that:

(i) If $y(n) \in O(z(n))$ and $x(n) \in O(y(n))$ then $x(n) \in O(z(n))$

(ii) $(2n^2 2^n + n \log n) \in \Theta(n^2 2^n)$

(iii) $(33n^3 + 4n^2) \in \Omega(n^3)$

(iv) $n! \in O(n^n)$ Hint: use $n! \approx \sqrt{(2\Pi n)}(n/e)^n$.

(i) If $y(n) \in O(z(n))$ and $x(n) \in O(y(n))$, then $x(n) \in O(z(n))$

To prove this, let's recall the definitions of **Big-Oh** notation:

- $y(n) \in O(z(n))$ means there exist positive constants c_1 and n_1 such that for all $n > n_1$,

$$y(n) \leq c_1 \cdot z(n)$$

- $x(n) \in O(y(n))$ means there exist positive constants c_2 and n_2 such that for all $n > n_2$,

$$x(n) \leq c_2 \cdot y(n)$$

We want to show that $x(n) \in O(z(n))$. For this, we need to find constants c and n_3 such that for all $n > n_3$,

$$x(n) \leq c \cdot z(n)$$

Proof:

From $x(n) \in O(y(n))$, we know:

$$x(n) \leq c_2 \cdot y(n)$$

From $y(n) \in O(z(n))$, we know:

$$y(n) \leq c_1 \cdot z(n)$$

Substitute the second inequality into the first:

$$x(n) \leq c_2 \cdot y(n) \leq c_2 \cdot c_1 \cdot z(n)$$

Thus, we have:

$$x(n) \leq (c_2 \cdot c_1) \cdot z(n)$$

Therefore, $x(n) \in O(z(n))$ with $c = c_2 \cdot c_1$ and $n_3 = \max(n_1, n_2)$.

This completes the proof.

(ii) Show that $(2n^2 2^n + n \log n) \in \Theta(n^2 2^n)$

We are asked to show that the function $2n^2 2^n + n \log n$ is **asymptotically tight bound** with $n^2 2^n$, meaning it grows at the same rate as $n^2 2^n$ for large values of n.

Proof:

To prove that $2n^2 2^n + n \log n \in \Theta(n^2 2^n)$, we need to show that:

$$c_1 \cdot n^2 2^n \leq 2n^2 2^n + n \log n \leq c_2 \cdot n^2 2^n$$

for some positive constants c_1 and c_2 and for sufficiently large n.

Step 1: Upper Bound

First, we observe that $2n^2 2^n$ dominates $n \log n$ for large n, because 2^n grows exponentially while $\log n$ grows logarithmically. Specifically:

- $2n^2 2^n \in O(n^2 2^n)$, which means that for large n, $2n^2 2^n$ is bounded by a constant multiple of $n^2 2^n$.
- $n \log n$ grows slower than $n^2 2^n$, so we can also conclude that $n \log n \in O(n^2 2^n)$.

Thus, we can write:

$$2n^2 2^n + n \log n \leq 2n^2 2^n + n^2 2^n = (2 + \epsilon) n^2 2^n$$

for large enough n, where ϵ is a very small constant, and thus the upper bound is $c_2 \cdot n^2 2^n$ for some constant c_2.

Step 2: Lower Bound

For the lower bound, note that $2n^2 2^n$ is the dominant term. Hence for sufficiently large n, the term $2n^2 2^n$ is always larger than $n \log n$, and we have:

$$2n^2 2^n + n \log n \geq 2n^2 2^n$$

This shows that the expression is bounded below by $c_1 \cdot n^2 2^n$ for some constant c_1.

Conclusion:

Since both the upper and lower bounds are of the order $n^2 2^n$, we can conclude that:

$$2n^2 2^n + n \log n \in \Theta(n^2 2^n)$$

(iii) Show that $(33n^3 + 4n^2) \in \Omega(n^3)$

We are asked to show that the function $33n^3 + 4n^2$ is **lower-bounded** by n^3, meaning $33n^3 + 4n^2$ grows at least as fast as n^3 for sufficiently large n.

Proof:

To show that $33n^3 + 4n^2 \in \Omega(n^3)$, we need to find constants c and n_0 such that for all $n > n_0$,

$$33n^3 + 4n^2 \geq c \cdot n^3$$

Step 1: Analyzing the Terms

We have two terms in the expression: $33n^3$ and $4n^2$.

- The first term, $33n^3$, clearly grows at the same rate as n^3, so it's trivially $\Omega(n^3)$.
- The second term, $4n^2$, grows slower than n^3, but we can show that $33n^3 + 4n^2$ is still lower-bounded by a constant multiple of n^3 for sufficiently large n.

Step 2: Finding the Constant

For sufficiently large n, the $4n^2$ term becomes negligible compared to the $33n^3$ term. Specifically, we have:

$$33n^3 + 4n^2 \geq 33n^3$$

for large enough n. Therefore, we can take $c = 33$, and we get:

$$33n^3 + 4n^2 \geq 33n^3$$

which satisfies the definition of Big-Omega notation.

Conclusion:

Thus, we have shown that $33n^3 + 4n^2 \in \Omega(n^3)$.

(iv) Show that $n! \in O(n^n)$

We are given the hint that $n!$ is approximately $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ using Stirling's approximation. We need to show that $n! \in O(n^n)$.

Proof:

From Stirling's approximation:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

This expression simplifies to:

$$n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$

For large n, the term $\sqrt{2\pi n}$ grows much slower than $\left(\frac{n}{e}\right)^n$, and we can focus on the dominant term:

$$n! \approx \left(\frac{n}{e}\right)^n$$

Now compare this to n^n. Since $\frac{n}{e}$ is less than n, it follows that:

$$n! \approx \left(\frac{n}{e}\right)^n \in O(n^n)$$

Thus, $n! \in O(n^n)$ because the growth rate of $n!$ is bounded above by n^n for large n.

This completes the proof.

(c) Draw the binary decision tree for binary search with n=14 where n is the total number of elements.

Binary Search Overview:

Binary search works by repeatedly dividing the array in half and comparing the target value with the middle element. Based on the comparison, the search continues in the left or right half of the array.

In a decision tree, each node represents a comparison of an element with the target, and each branch represents the outcome (whether the target is greater or smaller than the current element, or if it is found).

Binary Search Decision Tree for $n = 14$:

For $n = 14$, we assume that the array is sorted in increasing order, and binary search will search for the target by repeatedly splitting the array into two halves.

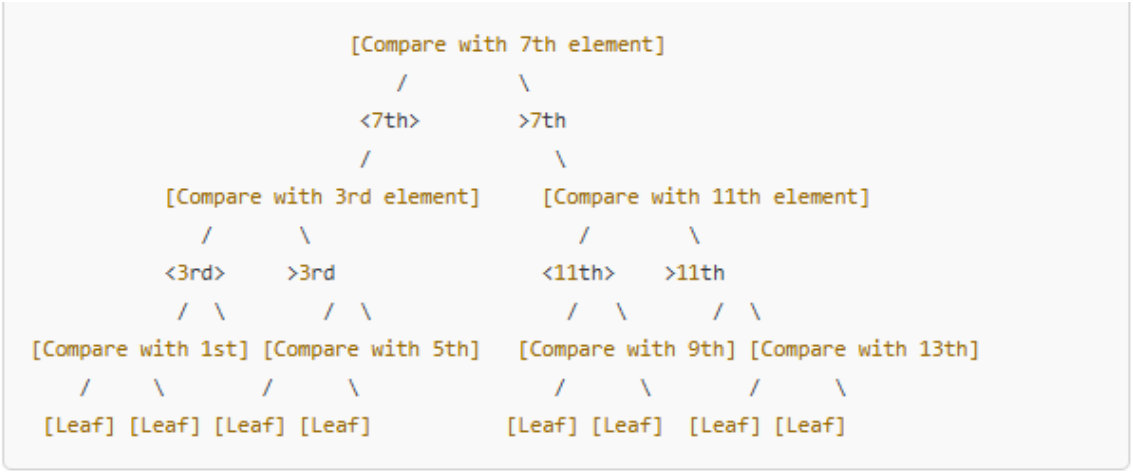
- In the first comparison, we check the middle element.
- If the target is less than the middle element, we search the left half.
- If the target is greater, we search the right half.
- This continues recursively until the target is found or the search space is exhausted.

The depth of the binary decision tree corresponds to the number of comparisons in the worst case. For $n = 14$, the height of the tree is $\log_2(14) \approx 3.8$, which rounds up to 4 levels (because we need to perform at most 4 comparisons to search an array of 14 elements).

Step-by-Step Binary Search for $n = 14$:

We can visualize the decision tree for binary search on a sorted array with 14 elements.

1. **Level 0 (root node)**: The entire array is considered. We check the middle element, which is the 7th element.
2. **Level 1**: Based on whether the target is smaller or larger than the 7th element, we split the array into two halves:
    - Left half: Elements 1 to 6.
    - Right half: Elements 8 to 14.
3. **Level 2**: For each half, we again check the middle element (e.g., the 3rd element for the left half or the 11th element for the right half) and continue splitting based on the comparison.
4. **Level 3**: Repeat the process until we find the target or exhaust the search space.



Explanation:

- At the root, we compare the target with the 7th element.
- If the target is smaller, we compare with the 3rd element from the left half (1-6).
- If the target is larger, we compare with the 11th element from the right half (8-14).
- The tree continues splitting into smaller sub-arrays, represented by the leaf nodes, which show the comparisons of individual elements.

Key Points:

- Each level in the tree represents a decision point where a comparison is made.
- The decision tree has a maximum depth of 4 for $n = 14$.
- The number of leaves (final nodes) is equal to the number of elements in the array, where the target element is located or not found.
