MCA 1st Semester Examination, 2022 MCA T 11: Design and Analysis of Algorithms. Full Marks: 70. Time: 3 hours.

Q1. a. Show that if y(n) ∈ O(z(n)) and x(n) ∈ O(y(n)) then x(n) ∈ O(z(n)).

Proof:

We are given:

1. $y(n) \in O(z(n))$
2. $x(n) \in O(y(n))$

We need to show that $x(n) \in O(z(n))$.

Step 1: Definition of Big-O

Using the definition of Big-O:

- $y(n) \in O(z(n))$ means:

$$\exists c_1 > 0,\ n_1 \text{ such that } y(n) \le c_1 \cdot z(n) \text{ for all } n \ge n_1.$$

- $x(n) \in O(y(n))$ means:

$$\exists c_2 > 0,\ n_2 \text{ such that } x(n) \le c_2 \cdot y(n) \text{ for all } n \ge n_2.$$

Step 2: Combine inequalities

For $n \geq \max(n_1, n_2)$:

- From the first inequality, $y(n) \leq c_1 \cdot z(n)$.
- Substituting $y(n)$ in the second inequality:

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

Thus,

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

Step 3: Conclusion

Let $c = c_1 \cdot c_2$ and $n_0 = \max(n_1, n_2)$.

Then,

$$x(n) \leq c \cdot z(n) \text{ for all } n \geq n_0.$$

By the definition of Big-O, $x(n) \in O(z(n))$.

Hence proved.

---

b. Show that if a(n) Є O(x(n)) and b(n) Є O(y(n)) then a(n)*b(n) Є O(x(n)*y(n)).

Proof:

We are given:

1. $a(n) \in O(x(n))$
2. $b(n) \in O(y(n))$

We need to show that $a(n) \cdot b(n) \in O(x(n) \cdot y(n))$.

Step 1: Definition of Big-O

Using the definition of Big-O:

- $a(n) \in O(x(n))$ means:

$$\exists c_1 > 0,\ n_1 \text{ such that } a(n) \leq c_1 \cdot x(n) \text{ for all } n \geq n_1.$$

- $b(n) \in O(y(n))$ means:

$$\exists c_2 > 0,\ n_2 \text{ such that } b(n) \leq c_2 \cdot y(n) \text{ for all } n \geq n_2.$$

Step 2: Multiply the inequalities

For $n \geq \max(n_1, n_2)$:

- From the first inequality:

$$a(n) \leq c_1 \cdot x(n).$$

- From the second inequality:

$$b(n) \leq c_2 \cdot y(n).$$

Multiplying these two inequalities:

$$a(n) \cdot b(n) \leq (c_1 \cdot x(n)) \cdot (c_2 \cdot y(n)) = (c_1 \cdot c_2) \cdot (x(n) \cdot y(n)).$$

Step 3: Conclusion

Let $c = c_1 \cdot c_2$ and $n_0 = \max(n_1, n_2)$.

Then,

$$a(n) \cdot b(n) \leq c \cdot (x(n) \cdot y(n)) \text{ for all } n \geq n_0.$$

By the definition of Big-O, $a(n) \cdot b(n) \in O(x(n) \cdot y(n))$.

Hence proved.

---

c.

Problem Statement:

We are given the polynomial function:

$$f(x) = a_p x^p + a_{p-1} x^{p-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

where $a_p > 0$ and $x \geq 1$.

We need to show that $f(x) \in O(x^p)$.

Step 1: Definition of Big-O

The definition of $f(x) \in O(x^p)$:

There exist constants $c > 0$ and $x_0 > 0$ such that:

$$f(x) \leq c \cdot x^p \quad \text{for all } x \geq x_0.$$

Step 2: Dominance of the Leading Term

For $x \geq 1$, each term in $f(x)$ can be bounded as follows:

- The leading term $a_p x^p$ remains as it is: $a_p x^p$.
- The second term $a_{p-1} x^{p-1}$ satisfies:

$$a_{p-1} x^{p-1} \leq a_{p-1} x^p, \text{ since } x \geq 1.$$

- Similarly, for all other terms $a_k x^k$ where $k < p$:

$$a_k x^k \leq a_k x^p, \text{ since } x \geq 1.$$

- The constant term a_0:

$$a_0 \leq a_0 \cdot x^p \text{ for } x \geq 1.$$

Step 3: Combine Bounds

Summing up all terms:

$$f(x) = a_p x^p + a_{p-1} x^{p-1} + \cdots + a_1 x + a_0 \leq a_p x^p + a_{p-1} x^p + \cdots + a_1 x^p + a_0 x^p.$$

Factoring out x^p:

$$f(x) \leq (a_p + a_{p-1} + \cdots + a_1 + a_0)\, x^p.$$

Let $c = a_p + a_{p-1} + \cdots + a_1 + a_0$. Since $c > 0$, we have:

$$f(x) \leq c \cdot x^p \quad \text{for all } x \geq 1.$$

Step 4: Conclusion

By the definition of Big-O, $f(x) \in O(x^p)$.

Hence proved.

---

d.

Problem Statement:

We need to show the mutual Big-O relationship between $\log_a n$ and $\log_b n$:

1. $\log_a n \in O(\log_b n)$
2. $\log_b n \in O(\log_a n)$.

Step 1: Change of Base Formula

Using the change of base formula:

$$\log_a n = \frac{\log_b n}{\log_b a}, \quad \text{and} \quad \log_b n = \frac{\log_a n}{\log_a b}.$$

Here, both $\log_b a$ and $\log_a b$ are constants because $a, b > 1$.

Step 2: Prove $\log_a n \in O(\log_b n)$

Using the change of base formula:

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

Since $\log_b a$ is a positive constant, we can write:

$$\log_a n \leq \frac{1}{\log_b a} \cdot \log_b n.$$

Let $c = \frac{1}{\log_b a}$. Then:

$$\log_a n \leq c \cdot \log_b n \quad \text{for all } n \geq 1.$$

By the definition of Big-O, $\log_a n \in O(\log_b n)$.

Step 3: Prove $\log_b n \in O(\log_a n)$

Using the change of base formula:

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

Since $\log_a b$ is a positive constant, we can write:

$$\log_b n \leq \frac{1}{\log_a b} \cdot \log_a n.$$

Let $c = \frac{1}{\log_a b}$. Then:

$$\log_b n \leq c \cdot \log_a n \quad \text{for all } n \geq 1.$$

By the definition of Big-O, $\log_b n \in O(\log_a n)$.

Conclusion:

$$\log_a n \in O(\log_b n) \quad \text{and} \quad \log_b n \in O(\log_a n).$$

Hence proved.

---

e.

Problem Statement:

We are given:

$$f(n) = n^2 \quad \text{and} \quad g(n) = n \log_2 n.$$

We need to show that $f(n) \notin O(g(n))$.

Step 1: Definition of Big-O

For $f(n) \in O(g(n))$, there must exist constants $c > 0$ and $n_0 > 0$ such that:

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

In this case,

$$n^2 \leq c \cdot (n \log_2 n).$$

We aim to show that this inequality does **not** hold for large n.

Step 2: Simplify the Inequality

Divide both sides by n (valid for $n > 0$):

$$n \leq c \cdot \log_2 n.$$

Step 3: Asymptotic Growth

- The left-hand side (n) grows **faster** than the right-hand side ($\log_2 n$) as $n \to \infty$, because:

$$\lim_{n \to \infty} \frac{\log_2 n}{n} = 0.$$

  This shows that n dominates $\log_2 n$.

Thus, for any constant $c > 0$, there exists some n large enough such that:

$$n > c \cdot \log_2 n.$$

Therefore, $n^2 > c \cdot n \log_2 n$ for sufficiently large n.

Step 4: Conclusion

The inequality $f(n) \leq c \cdot g(n)$ does **not** hold for large n.

Hence, $f(n) \notin O(g(n))$.

Hence proved.

---

f.

Problem Statement:

We need to show that:

$$f(n) = 2n^2 2^n + n \log n \in \Theta(n^2 2^n).$$

Step 1: Definition of Θ-notation

For $f(n) \in \Theta(g(n))$, there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0.$$

Here, $g(n) = n^2 2^n$.

We will analyze $f(n) = 2n^2 2^n + n \log n$ to determine its bounds relative to $g(n)$.

Step 2: Dominance of the Terms in $f(n)$

1. The first term $2n^2 2^n$:
   - Clearly, this term is proportional to $g(n)$. Specifically:

$$2n^2 2^n = 2 \cdot g(n).$$

2. The second term $n \log n$:
   - Compare $n \log n$ to $g(n) = n^2 2^n$.
   - For large n, $n \log n$ grows much slower than $n^2 2^n$ because 2^n dominates any polynomial or logarithmic growth.
   - Hence, $n \log n$ is asymptotically negligible compared to $n^2 2^n$:

$$\frac{n \log n}{n^2 2^n} \to 0 \quad \text{as } n \to \infty.$$

Thus, $f(n)$ is asymptotically dominated by the first term $2n^2 2^n$.

Step 3: Upper Bound

For large n, $f(n)$ satisfies:

$$f(n) = 2n^2 2^n + n \log n \leq 2n^2 2^n + n^2 2^n \quad (\text{since } n \log n \leq n^2 2^n \text{ for large } n).$$

Factor $n^2 2^n$:

$$f(n) \leq (2+1) \cdot n^2 2^n = 3 \cdot g(n).$$

Thus, $f(n) \leq c_2 \cdot g(n)$ with $c_2 = 3$.

Step 4: Lower Bound

For large n, $f(n)$ satisfies:

$$f(n) = 2n^2 2^n + n \log n \geq 2n^2 2^n.$$

Thus, $f(n) \geq c_1 \cdot g(n)$ with $c_1 = 2$.

Step 5: Conclusion

We have shown that:

$$2 \cdot g(n) \leq f(n) \leq 3 \cdot g(n) \quad \text{for large } n.$$

By the definition of Θ-notation, $f(n) \in \Theta(n^2 2^n)$.

Hence proved.

---

g.

Problem Statement:

We need to show that:

$$f(n) = 33n^3 + 4n^2 \in \Omega(n^3).$$

Step 1: Definition of Ω-notation

For $f(n) \in \Omega(g(n))$, there exists a constant $c > 0$ and $n_0 > 0$ such that:

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

Here, $g(n) = n^3$.

Step 2: Analyze $f(n)$

$$f(n) = 33n^3 + 4n^2.$$

For large n, the term $33n^3$ dominates $4n^2$ because n^3 grows faster than n^2. Thus:

$$f(n) \geq 33n^3 \quad \text{for all } n \geq 1.$$

Step 3: Simplify the Inequality

Divide both sides by n^3:

$$\frac{f(n)}{n^3} = 33 + \frac{4}{n}.$$

As $n \to \infty$, the term $\frac{4}{n} \to 0$. Therefore:

$$\frac{f(n)}{n^3} \to 33.$$

Thus, for sufficiently large n, say $n \geq n_0 = 1$:

$$f(n) \geq 33 \cdot n^3.$$

Step 4: Conclusion

By the definition of Ω-notation, $f(n) \in \Omega(n^3)$.

Hence proved.

---

h.

Problem Statement:

We need to show that:

$$n! \in O(n^n).$$

We are given the approximation:

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

Step 1: Use the Approximation for $n!$

Using Stirling's approximation for $n!$:

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

We want to show that $n!$ grows at most as n^n, i.e., we want to show that:

$$n! \leq c \cdot n^n \quad \text{for some constant } c \text{ and large } n.$$

Step 2: Simplify the Expression

Substitute the approximation into the expression for $n!$:

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

We want to compare this with n^n:

$$n! \approx \sqrt{2\pi n} \cdot \frac{n^n}{e^n}.$$

Now express this as:

$$n! \approx \frac{\sqrt{2\pi n}}{e^n} \cdot n^n.$$

Step 3: Bound the Growth of $n!$

The factor $\frac{\sqrt{2\pi n}}{e^n}$ grows much slower than n^n as $n \to \infty$. Specifically, the exponential factor e^n in the denominator decays rapidly compared to the polynomial term $\sqrt{2\pi n}$, and so the entire factor $\frac{\sqrt{2\pi n}}{e^n}$ becomes negligible for large n.

Therefore, for sufficiently large n, we can conclude that:

$$n! \leq c \cdot n^n \quad \text{for some constant } c.$$

Step 4: Conclusion

Since we have shown that $n!$ is asymptotically bounded above by n^n for large n, we conclude that:

$$n! \in O(n^n).$$

Hence proved.

Q2. What is a heap data structure? Write an algorithm to sort the following list of numbers using

a heap data structure. Also find out the time and space complexity of the method.

4, 5,2,9,8,7,1,3

Question Breakdown:

1. **Define a heap data structure**
2. **Provide an algorithm for heap sort**
3. **Sort the given list: 4, 5, 2, 9, 8, 7, 1, 3**
4. **Analyze the time and space complexity**

:

1. What is a Heap Data Structure?

A **heap** is a specialized binary tree-based data structure that satisfies the **heap property**:

- **Max-Heap**: The key of each node is greater than or equal to the keys of its children.
- **Min-Heap**: The key of each node is less than or equal to the keys of its children.

The heap is commonly implemented as an array. For a node at index i, the child nodes are located at indices:

- Left child: $2i + 1$
- Right child: $2i + 2$
- Parent: $\lfloor (i - 1)/2 \rfloor$

2. Heap Sort Algorithm

Heap sort is a comparison-based sorting algorithm that involves:

1. Building a max-heap from the input array.
2. Repeatedly extracting the maximum element and rebuilding the heap.

Steps:

1. **Build Max-Heap**:
   - Rearrange the array into a max-heap.
2. **Sort**:
   - Swap the root (largest element) with the last element of the heap.
   - Reduce the heap size by 1.
   - Heapify the root to restore the max-heap property.
   - Repeat until the heap size is 1.

Algorithm:

Heapify Function (Maintains Max-Heap Property)

```
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    left = 2 * i + 1  # Left child
    right = 2 * i + 2  # Right child

    # If left child exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # If right child exists and is greater than the largest so far
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        # Recursively heapify the affected subtree
        heapify(arr, n, largest)
```

Heap Sort Function

```
def heap_sort(arr):
    n = len(arr)

    # Build a max-heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from the heap one by one
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]  # Swap
        heapify(arr, i, 0)
```

3. Sorting the List

Input: $[4, 5, 2, 9, 8, 7, 1, 3]$

Step 1: Build Max-Heap

After heapifying, the max-heap looks like:

$$[9, 8, 7, 5, 4, 2, 1, 3]$$

Step 2: Perform Heap Sort

1. Swap 9 (root) with the last element 3:
   $[3, 8, 7, 5, 4, 2, 1, 9]$.
   Heapify the reduced heap:
   $[8, 5, 7, 3, 4, 2, 1, 9]$.
2. Swap 8 with 1:
   $[1, 5, 7, 3, 4, 2, 8, 9]$.
   Heapify:
   $[7, 5, 2, 3, 4, 1, 8, 9]$.
3. Swap 7 with 1:
   $[1, 5, 2, 3, 4, 7, 8, 9]$.
   Heapify:
   $[5, 4, 2, 3, 1, 7, 8, 9]$.
4. Swap 5 with 1:
   $[1, 4, 2, 3, 5, 7, 8, 9]$.
   Heapify:
   $[4, 3, 2, 1, 5, 7, 8, 9]$.
5. Continue swapping and heapifying until sorted:
   $[1, 2, 3, 4, 5, 7, 8, 9]$.

4. Time and Space Complexity

Time Complexity:

1. **Building the Max-Heap**: $O(n)$.
2. **Heapify Operations**:
    - Each extraction involves $O(\log n)$ heapify operations.
    - For n elements, this takes $O(n \log n)$.

Total time complexity:

$$O(n \log n)$$

Space Complexity:

Heap sort is **in-place**, so the space complexity is:

$$O(1)$$

Final Answer:

Sorted List:

$$[1, 2, 3, 4, 5, 7, 8, 9]$$

Time Complexity: $O(n \log n)$

Space Complexity: $O(1)$

Q3. What are disjoint set data structures? Write algorithms for the following operations on the disjoint set data structure:
a. Find (x): Returns the root of the tree containing the element x.
b. Union (x, y): Replace two sets containing elements x and y respectively by their union.
How the performance of the above operations are improved by using appropriate heuristics.

Let {1}, {2}, {3}, {4}, {5}, {6}, {7} and {8} be singleton sets, each represented by a tree with exactly one node. Use the union find algorithm with union by rank and path compression to find the tree representation of the set resulting from each of the following unions and finds: union(1,2),union(3,4),union(5,6), union(7,8), union(1,3), union(5,7), find(1), union(1,5), find(1).

1. What are Disjoint Set Data Structures?

A **disjoint-set data structure** (also known as **union-find** or **mergetree**) is used to manage a collection of non-overlapping sets. It supports two main operations:

1. **Find (x)**: Determines the set to which element x belongs. This operation returns the representative or root of the set.
2. **Union (x, y)**: Merges the sets containing elements x and y into a single set.

These operations are particularly useful in problems involving connectivity or equivalence relations, such as Kruskal's algorithm for finding a Minimum Spanning Tree or determining connected components in a graph.

2. Key Operations and Algorithms

a. Find (x)

The **Find** operation returns the root of the tree containing the element x.
In a naive approach, we simply follow the parent pointers until we reach a node that points to itself, indicating the root.

However, using **path compression**, we improve the performance by making the trees flatter. During the find operation, we make each node along the path point directly to the root.

Find Algorithm with Path Compression:

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

b. Union (x, y)

The **Union** operation merges the sets containing elements x and y.
In a naive approach, we could simply attach one tree under the other, but to keep the trees balanced and reduce the depth of the tree, we use **union by rank**. The rank is an estimate of the tree's height, and we always attach the tree with the smaller rank to the tree with the larger rank.

Union Algorithm with Union by Rank:

```
def union(x, y, parent, rank):
    rootX = find(x, parent)
    rootY = find(y, parent)
```

```
if rootX != rootY:
    if rank[rootX] > rank[rootY]:
        parent[rootY] = rootX  # rootX becomes the parent
    elif rank[rootX] < rank[rootY]:
        parent[rootX] = rootY  # rootY becomes the parent
    else:
        parent[rootY] = rootX  # Arbitrarily make rootX the parent
        rank[rootX] += 1  # Increment the rank of rootX
```

3. Performance Improvements Using Heuristics

1. **Union by Rank**:
   By always attaching the smaller tree (based on rank) under the larger tree, we keep the height of the resulting trees small, reducing the time required for future operations.
2. **Path Compression**:
   During the Find operation, path compression ensures that each node on the path directly points to the root. This reduces the depth of the tree, leading to faster future queries.

These two heuristics combined make the **amortized time complexity** of both Find and Union operations nearly constant, specifically $O(\alpha(n))$, where $\alpha(n)$ is the **inverse Ackermann function**, which grows very slowly and is considered practically constant for all practical input sizes.

4. Union-Find Operations on the Given Example

We are given the following sets:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}$$

We'll apply the union-find operations one by one and track the trees formed after each operation.

1. **union(1, 2)**:
   - Union sets containing 1 and 2.
   - Since both are singleton sets, we attach one to the other (say, 1 becomes the parent of 2).
   - Resulting trees:

$$\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}$$

2. **union(3, 4)**:
   - Union sets containing 3 and 4.
   - Resulting trees:

$$\{1,2\},\{3,4\},\{5\},\{6\},\{7\},\{8\}$$

3. **union(5, 6)**:
   - Union sets containing 5 and 6.
   - Resulting trees:

$$\{1,2\},\{3,4\},\{5,6\},\{7\},\{8\}$$

4. **union(7, 8)**:
   - Union sets containing 7 and 8.
   - Resulting trees:

$$\{1,2\},\{3,4\},\{5,6\},\{7,8\}$$

5. **union(1, 3)**:
   - Union sets containing 1 and 3.
   - Union by rank: $\{1,2\}$ has rank 1 and $\{3,4\}$ has rank 1, so we can arbitrarily make 1 the root of 3.
   - Resulting trees:

$$\{1,2,3,4\},\{5,6\},\{7,8\}$$

6. **union(5, 7)**:
   - Union sets containing 5 and 7.
   - Resulting trees:

$$\{1,2,3,4\},\{5,6,7,8\}$$

7. **find(1)**:
    - Find the root of the set containing 1.
    - The root is 1, and the tree looks like:

$$\{1, 2, 3, 4\}, \{5, 6, 7, 8\}$$

8. **union(1, 5)**:
    - Union sets containing 1 and 5.
    - Union by rank: Both sets have rank 2, so we can make 1 the root of 5.
    - Resulting trees:

$$\{1, 2, 3, 4, 5, 6, 7, 8\}$$

9. **find(1)**:
    - Find the root of the set containing 1.
    - The root is 1, and the tree is fully merged into a single set.

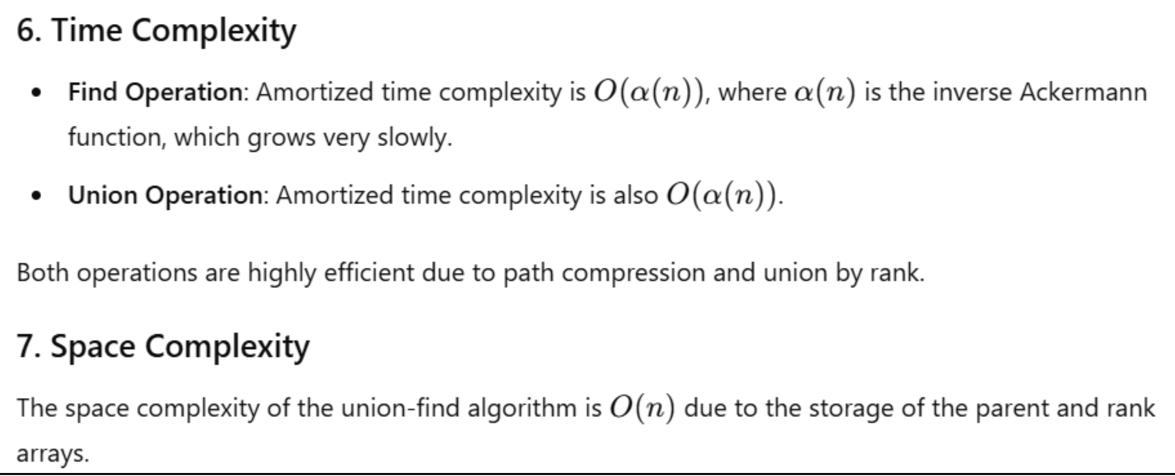
$$\{1, 2, 3, 4, 5, 6, 7, 8\}$$

5. Final Tree Representation

After performing all the operations, the final tree representation of the set is:

$$\{1, 2, 3, 4, 5, 6, 7, 8\}$$

This means all elements are in a single set, with 1 as the root of the tree. Path compression ensures that all elements point directly to the root.



6. Time Complexity

- **Find Operation**: Amortized time complexity is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly.
- **Union Operation**: Amortized time complexity is also $O(\alpha(n))$.

Both operations are highly efficient due to path compression and union by rank.

7. Space Complexity

The space complexity of the union-find algorithm is $O(n)$ due to the storage of the parent and rank arrays.

Q4. Write short notes on: a. Amortized Analysis. b. Binary Search.

a. Amortized Analysis

Amortized analysis is a technique used to evaluate the average time complexity of an operation over a sequence of operations, rather than on a single operation. In cases where a sequence of operations has varying time costs, amortized analysis provides a way to understand the overall performance by averaging the time complexity over all the operations.

There are three main types of amortized analysis:

1. **Aggregate Analysis**:
   This approach calculates the total cost of a sequence of operations and then divides it by the number of operations to find the average cost per operation.

   Example:
   Suppose a sequence of operations consists of 10 operations, with a total cost of 100 units. The amortized cost per operation is 100/10=10 units.

2. **Accounting (or Bank) Method**:
   In this method, we assign "credits" to each operation in advance to pay for the expensive operations. When an expensive operation occurs, the credit is used. If an operation is cheap, the remaining credit is carried forward. This method is useful for situations where operations have varying costs.

   Example:
   If an operation has a worst-case cost of 20, but most operations have a cost of 1, we might allocate 10 units of credit for each cheap operation, saving up enough credit to pay for the more expensive operations.

3. **Potential Method**:
   This method uses a potential function to capture the state of the data structure at any point in time. The potential is used to account for future costs, and the amortized cost is the actual cost of the operation plus the change in potential.

Example:
In a stack with push and pop operations, the potential function can be defined based on the number of elements in the stack. The amortized cost of an operation is the actual cost of the operation plus the change in potential.

Amortized analysis is particularly useful for analyzing data structures or algorithms where operations have varying costs (e.g., dynamic arrays, binary heap operations). It provides a more accurate and insightful analysis compared to worst-case analysis, where only the single most expensive operation is considered.

b. Binary Search

Binary Search is an efficient searching algorithm that works on **sorted arrays or lists.** It repeatedly divides the search interval in half, allowing it to search for an element in $O(\log n)$ time, which is much faster than linear search (which has a time complexity of $O(n)$).

Steps of Binary Search:

1. **Initialization:**

   Start with two pointers, `low` (start of the array) and `high` (end of the array).

2. **Middle Element:**

   Calculate the middle index:
   $\text{mid} = \text{low} + \frac{\text{high} - \text{low}}{2}$

3. **Comparison:**

   - If the middle element equals the target, the search is successful, and the index is returned.
   - If the middle element is less than the target, the search continues on the right half of the array (i.e., update `low` to `mid + 1`).
   - If the middle element is greater than the target, the search continues on the left half of the array (i.e., update `high` to `mid - 1`).

4. **Repeat:**

   Continue halving the array until the target is found or the search interval becomes empty (i.e., `low > high`).

Binary Search Algorithm (Pseudocode):

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = low + (high - low) // 2

        if arr[mid] == target:
            return mid  # Element found, return its index
        elif arr[mid] < target:
            low = mid + 1  # Search in the right half
        else:
            high = mid - 1  # Search in the left half

    return -1  # Element not found
```

Time Complexity:

- **Best Case:** $O(1)$ (when the target is at the middle of the array).
- **Worst Case:** $O(\log n)$ (when the target is not found, or the array is reduced to one element).
- **Average Case:** $O(\log n)$.

Space Complexity:

- **O(1)** for the iterative approach (no additional space required except for variables).
- For the recursive approach, it would be $O(\log n)$ due to the call stack.

Advantages of Binary Search:

1. **Efficiency**: It is much faster than linear search for large datasets, especially when dealing with large sorted arrays.
2. **Logarithmic Time Complexity:** Binary search reduces the problem size by half in each step, making it highly efficient with $O(\log n)$ complexity.

Limitations:

- Binary search can only be applied to **sorted** arrays or lists. If the data is not sorted, the array must first be sorted (which may take $O(n \log n)$ time), making binary search less efficient in those cases.
