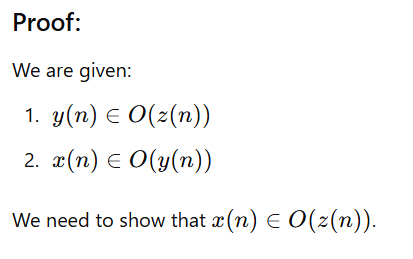
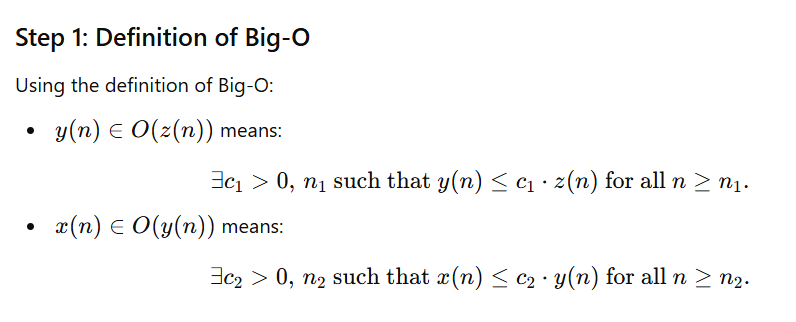
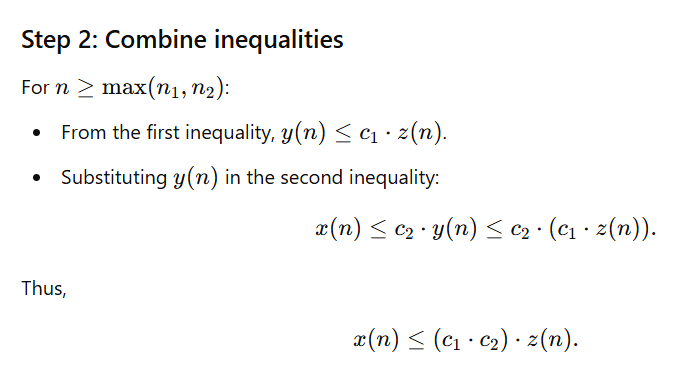
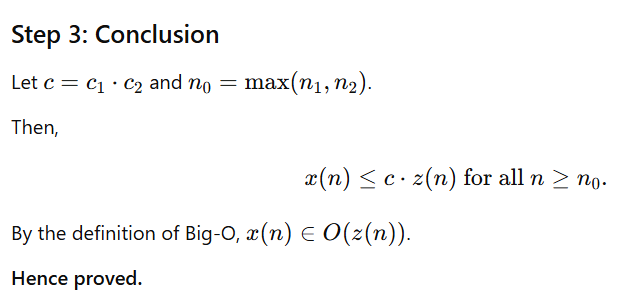
**MCA 1st Semester Examination, 2022 MCA T 11: Design and Analysis of Algorithms. Full Marks: 70. Time: 3 hours.**

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

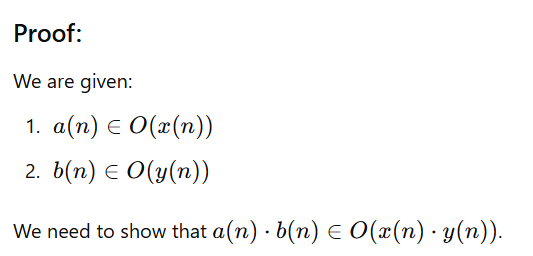


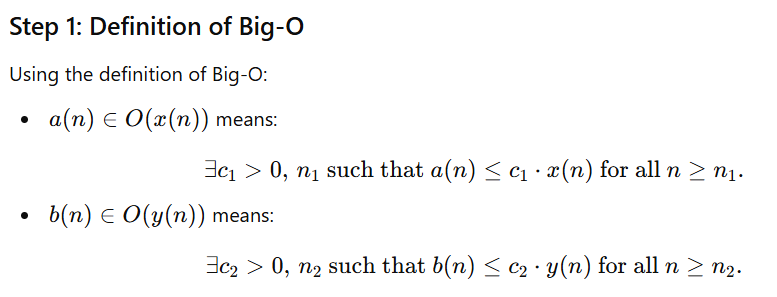


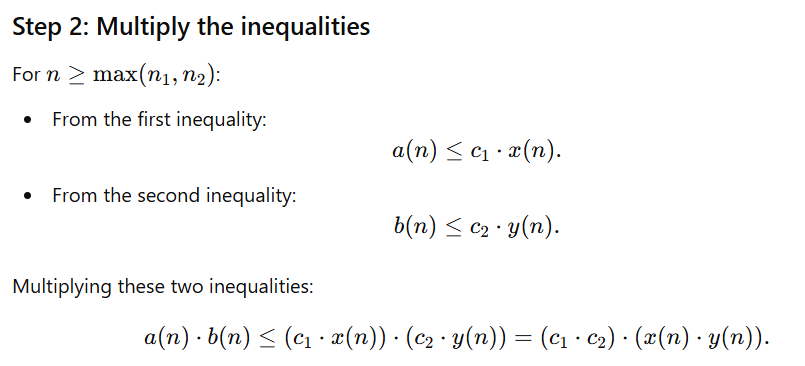


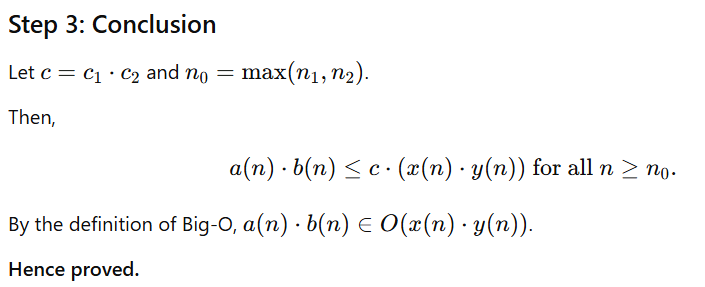


**b. Show that if a(n) Є O(x(n)) and b(n) Є O(y(n)) then a(n)\*b(n) Є O(x(n)\*y(n)).**

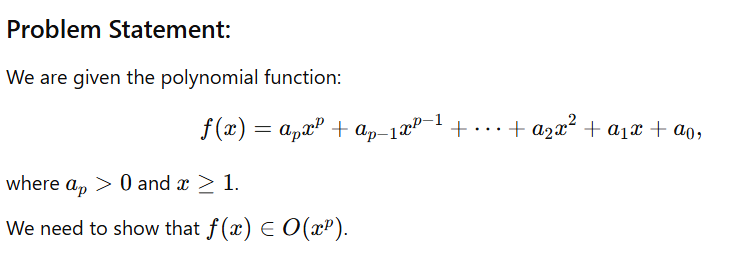


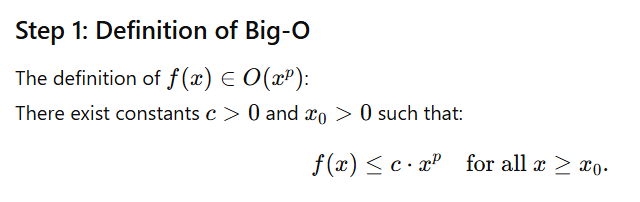


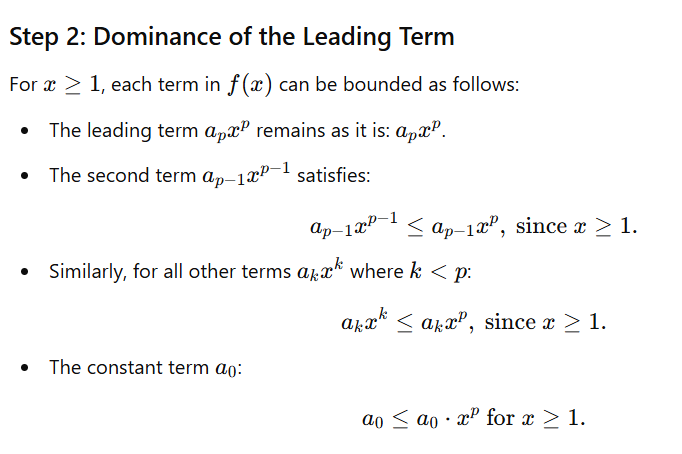


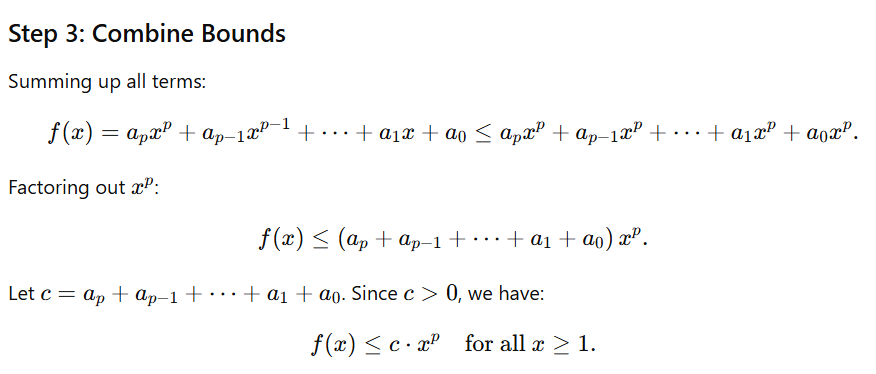


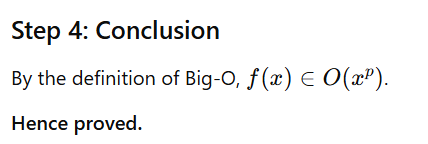
**c.**



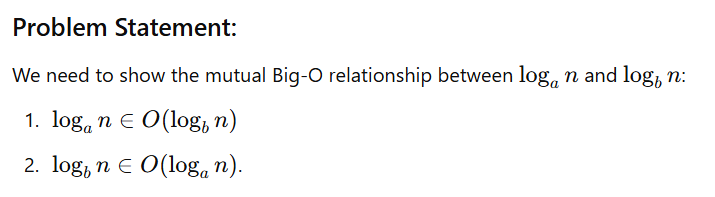


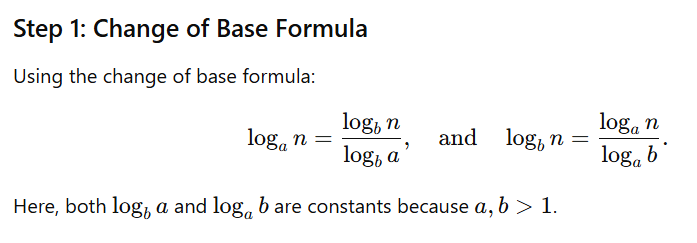


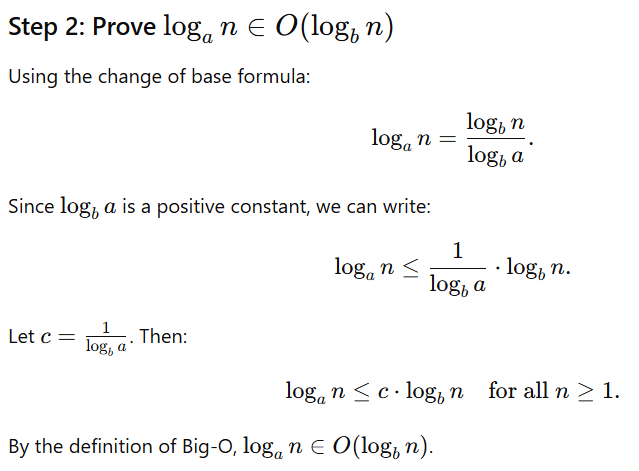


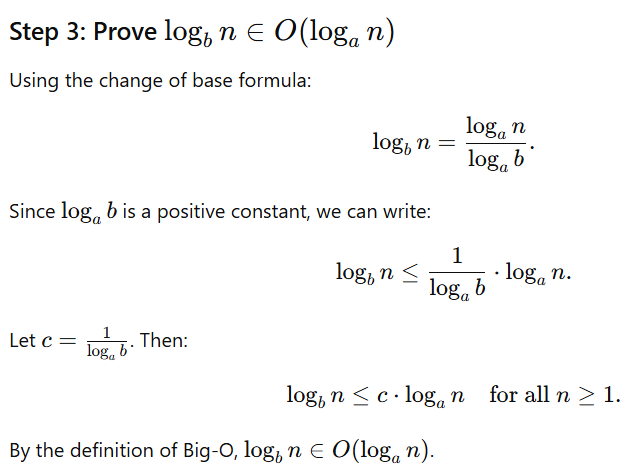


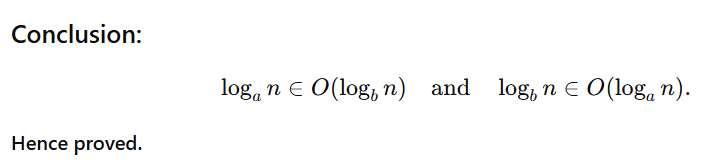
**d.**



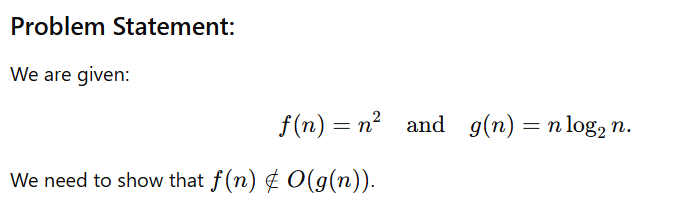


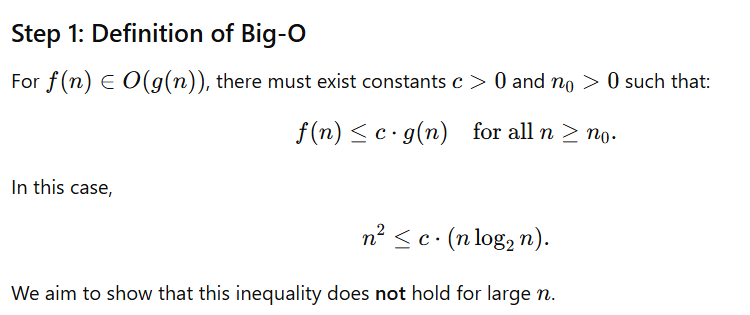


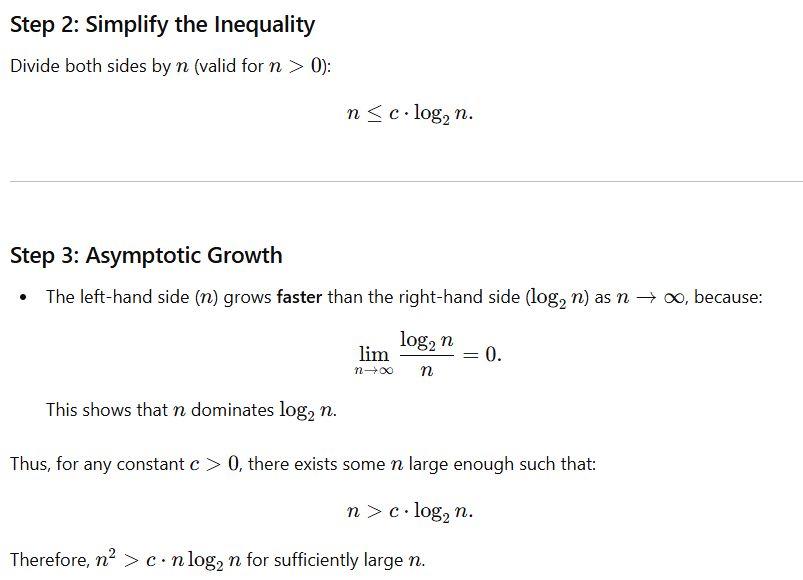


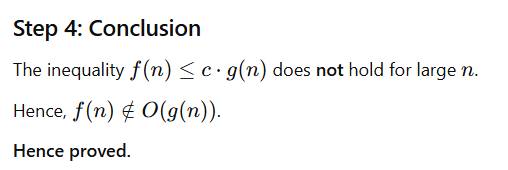


**e.**

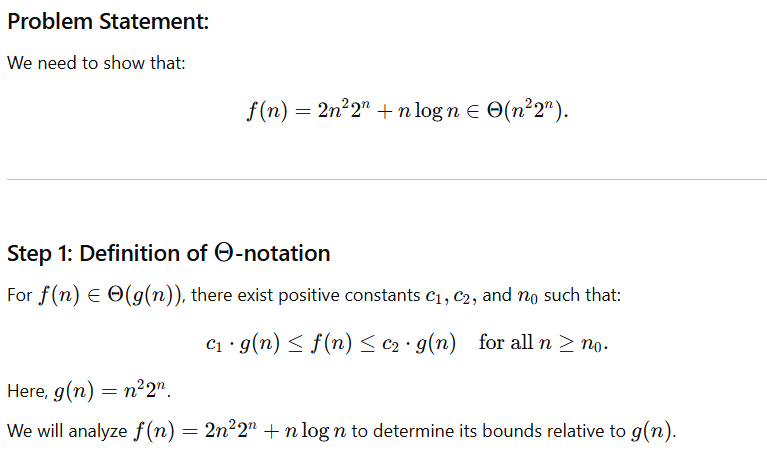


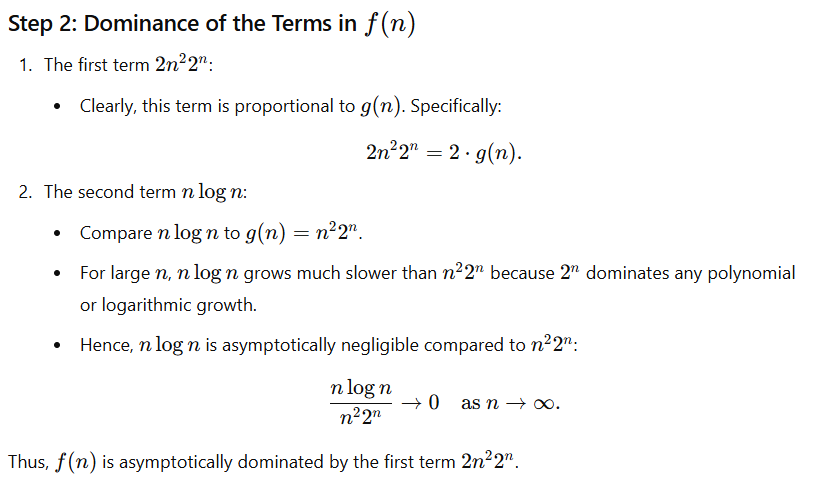


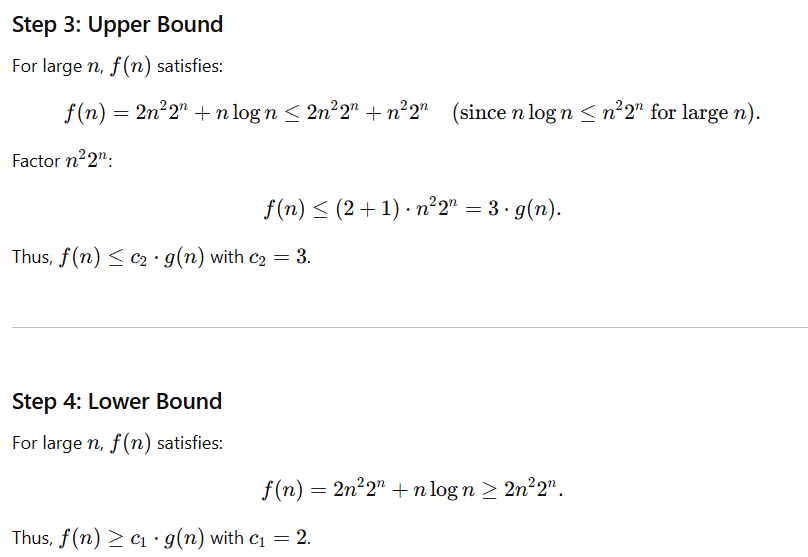


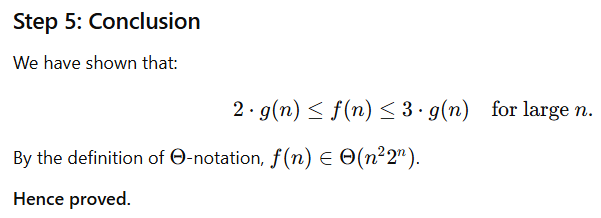


**f.**

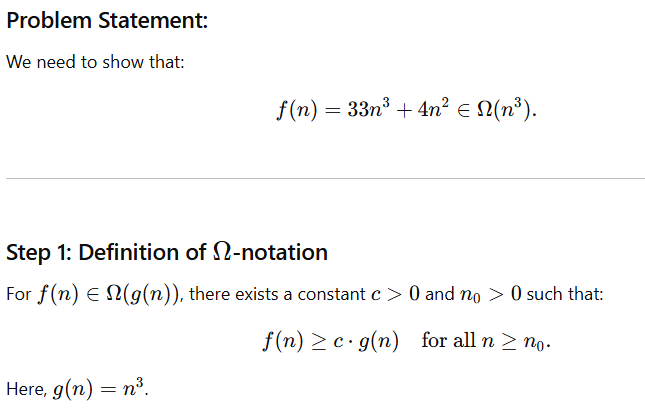


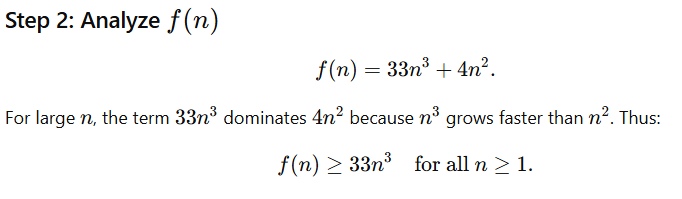


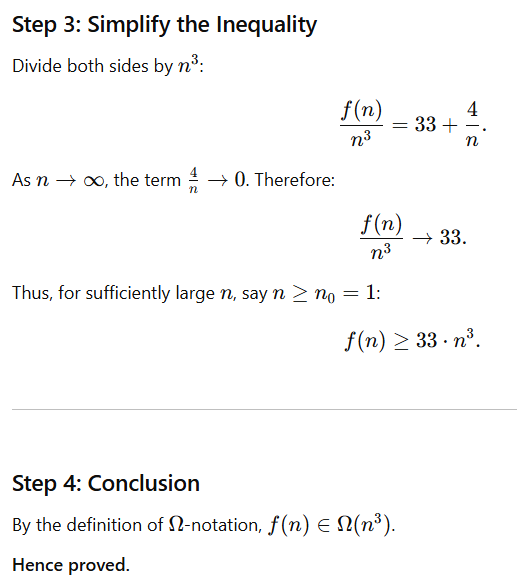




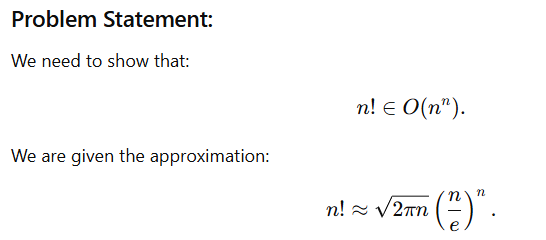
**g.**

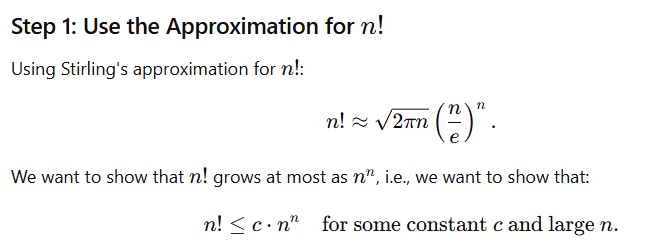


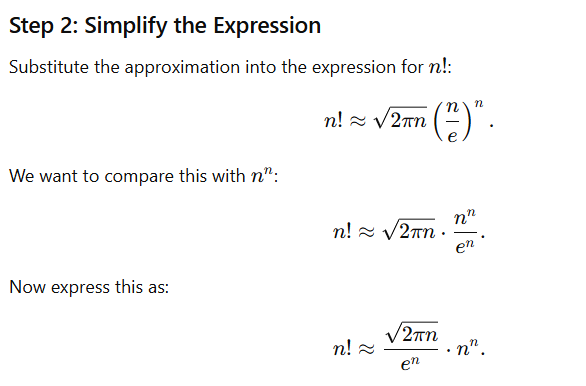


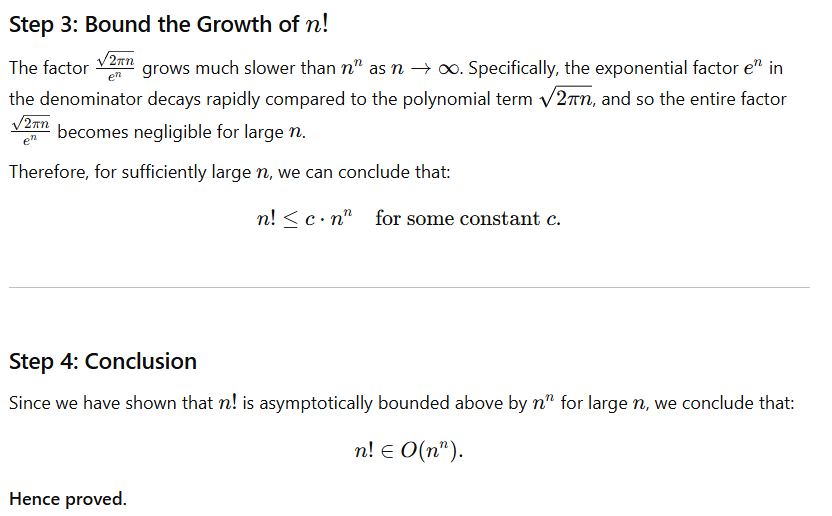


**h.**









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

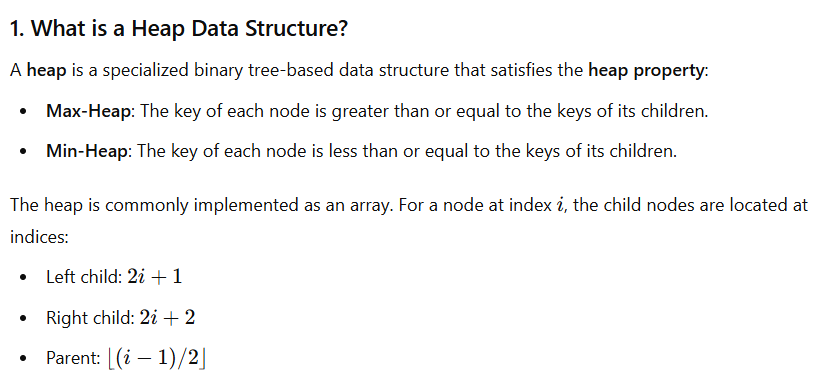
**a heap data structure. Also find out the time and space complexity of the method.**

**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**

**:**



**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):

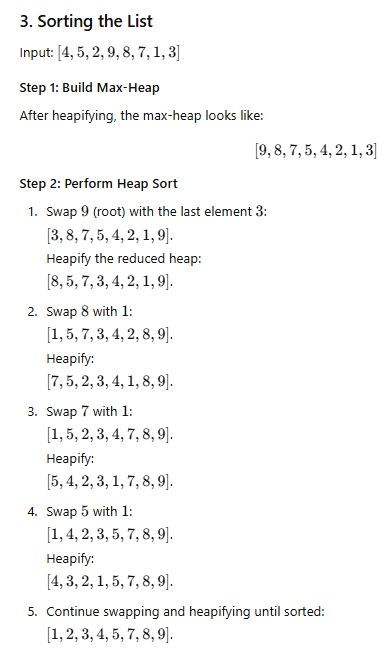
heapify(arr, n, i)

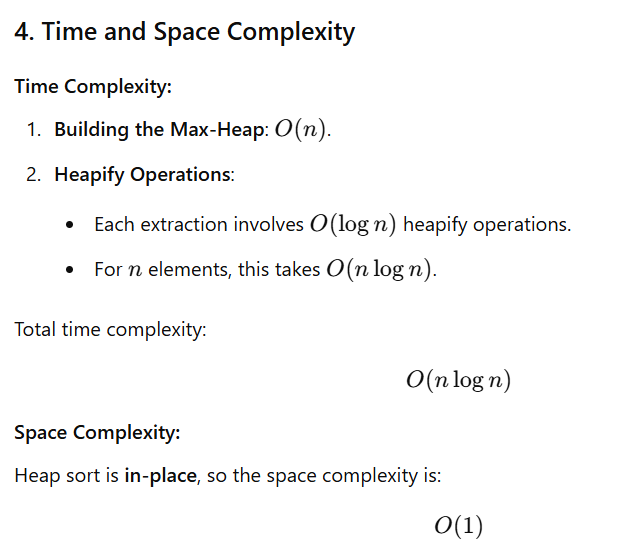
# Extract elements from the heap one by one

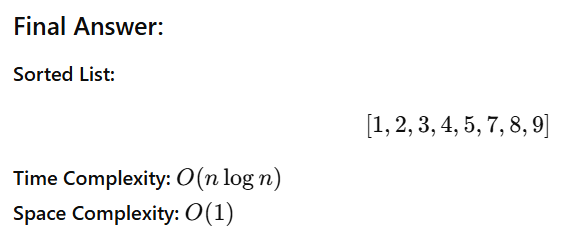
for i in range(n - 1, 0, -1):

arr[0], arr[i] = arr[i], arr[0] # Swap

heapify(arr, i, 0)





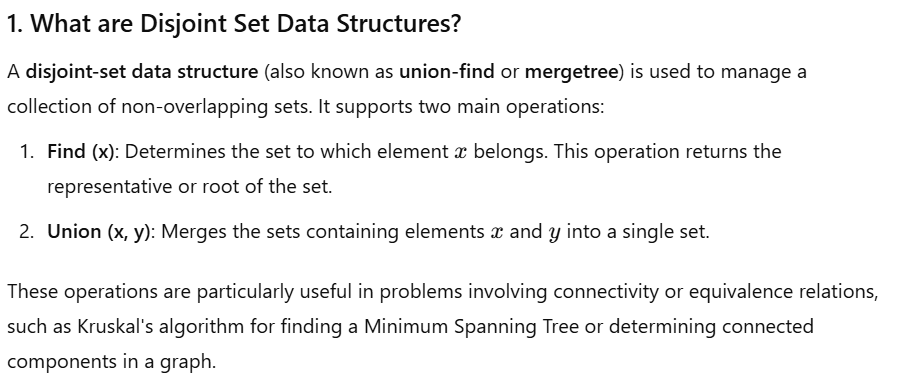


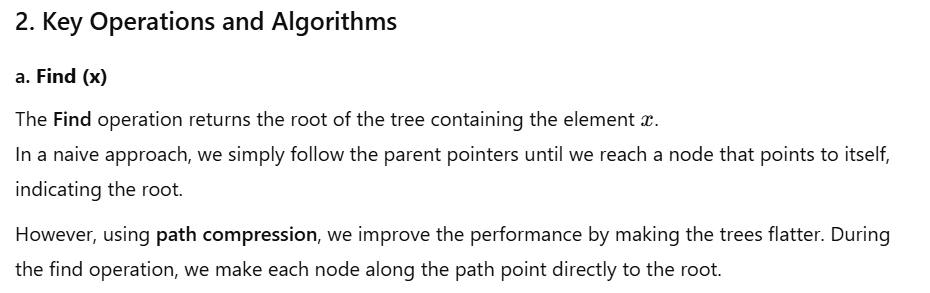
**Q3. What are disjoint set data structures? Write algorithms for the following operations on the disjoint set data structure:**

**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).**





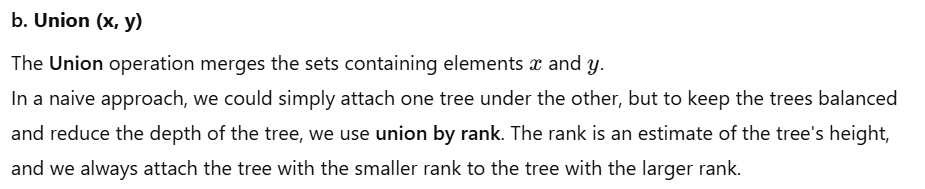
**Find Algorithm with Path Compression:**

def find(x, parent):

if parent[x] != x:

parent[x] = find(parent[x], parent) # Path compression

return parent[x]



**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

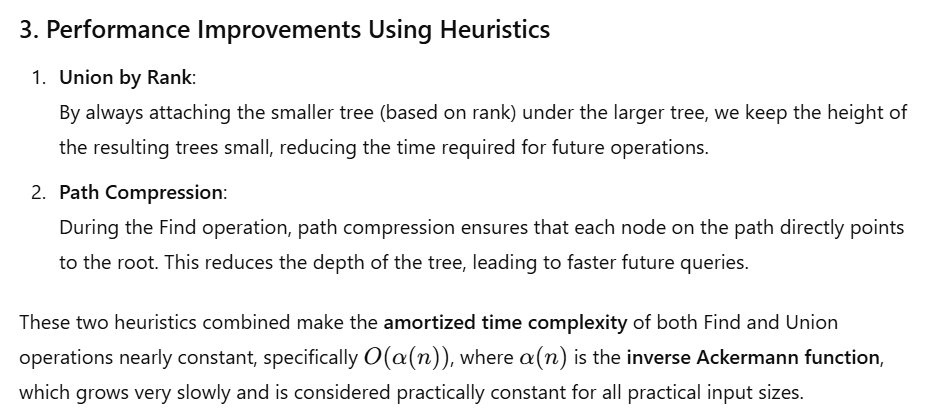
elif rank[rootX] < rank[rootY]:

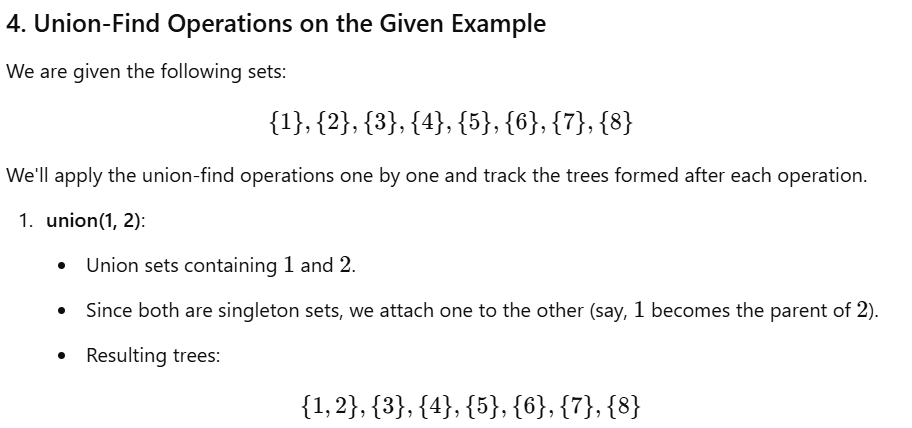
parent[rootX] = rootY # rootY becomes the parent

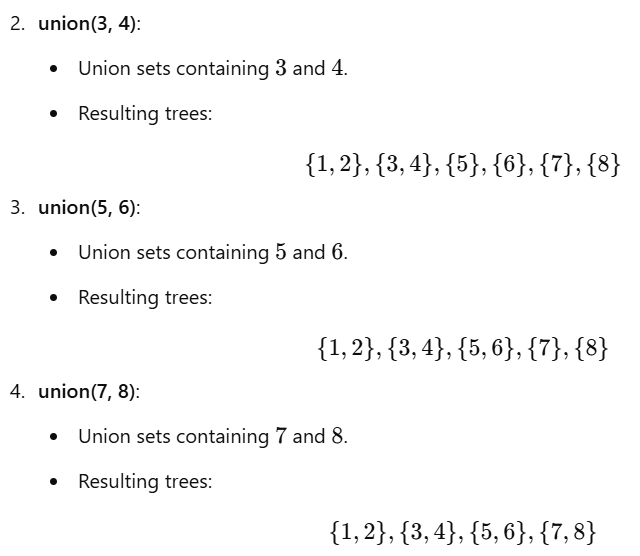
else:

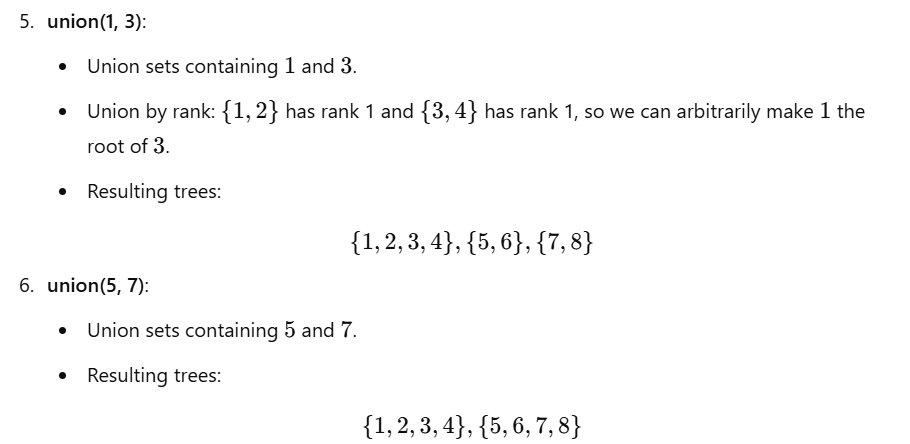
parent[rootY] = rootX # Arbitrarily make rootX the parent

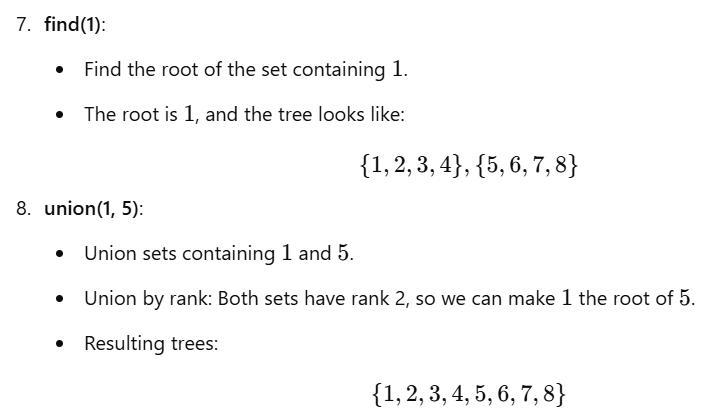
rank[rootX] += 1 # Increment the rank of rootX

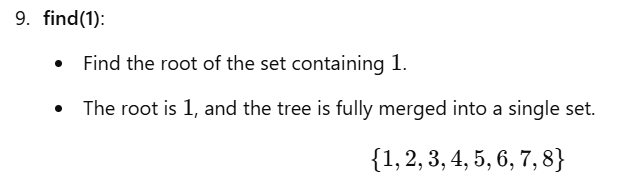


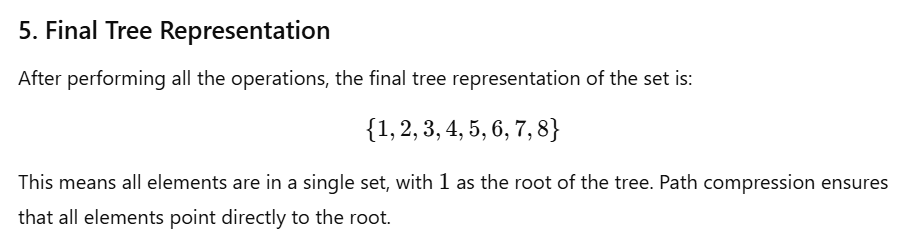


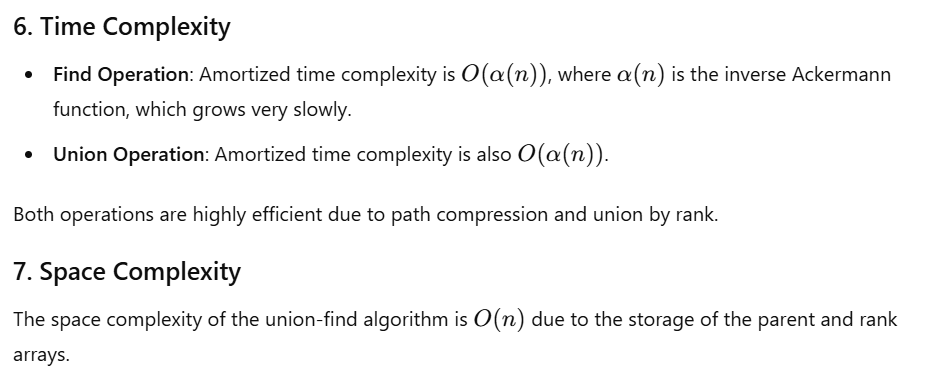












**Q4. Write short notes on: a. Amortized Analysis. b. Binary Search.**

#### ****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.

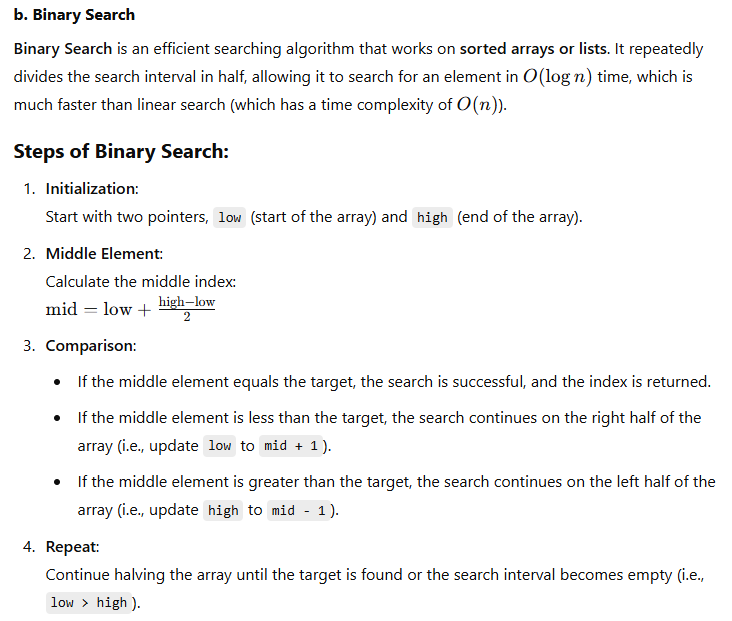
1. **Accounting (or Bank) Method**:  
   In this method, we assign "credits" to each operation in advance to pay for the expensive operations. When an expensive operation occurs, the credit is used. If an operation is cheap, the remaining credit is carried forward. This method is useful for situations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

1. **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.



### ****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

