**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

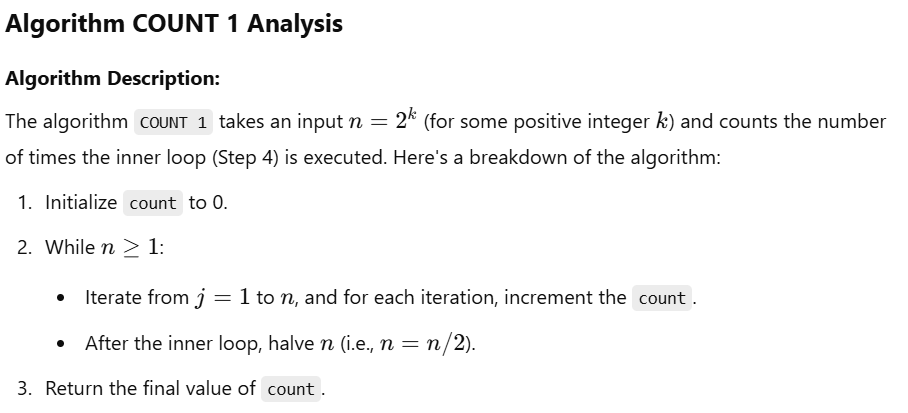
4. count <- count + 1

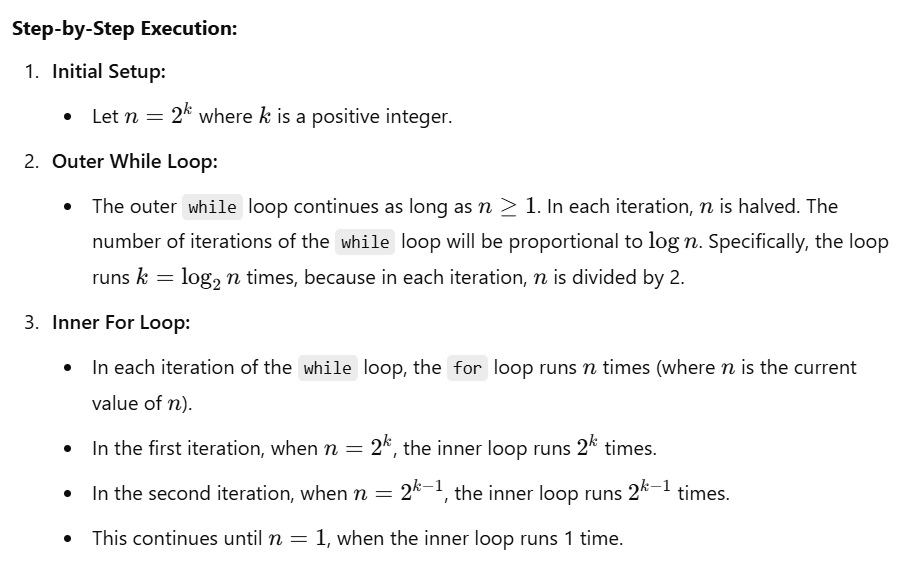
5. **end for**

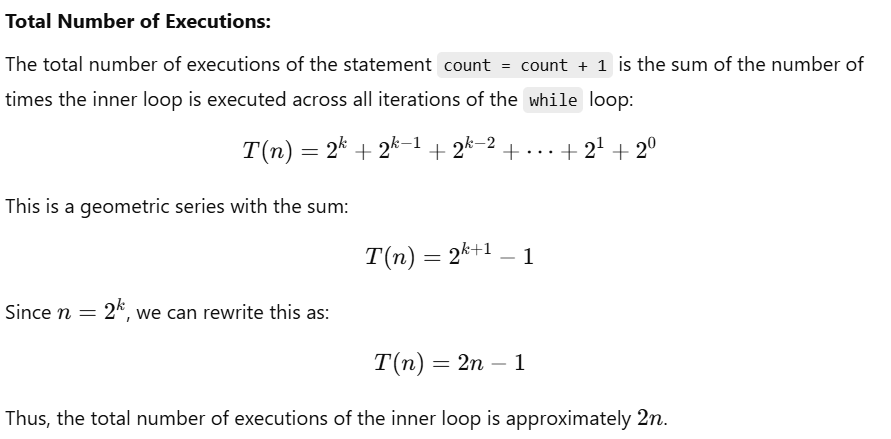
6. n<-n/2

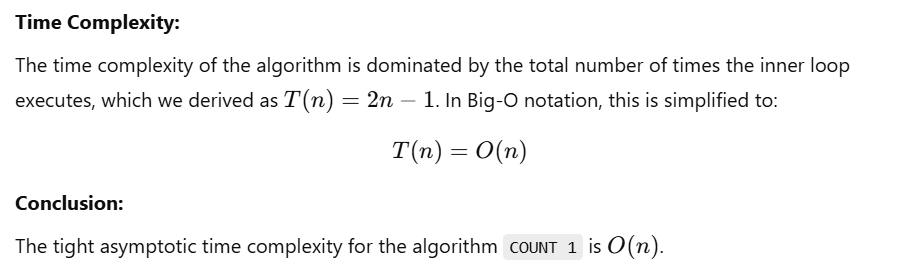
7. **end while**

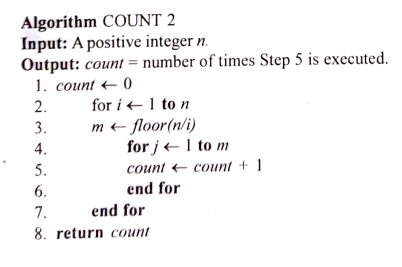
8. **return** count

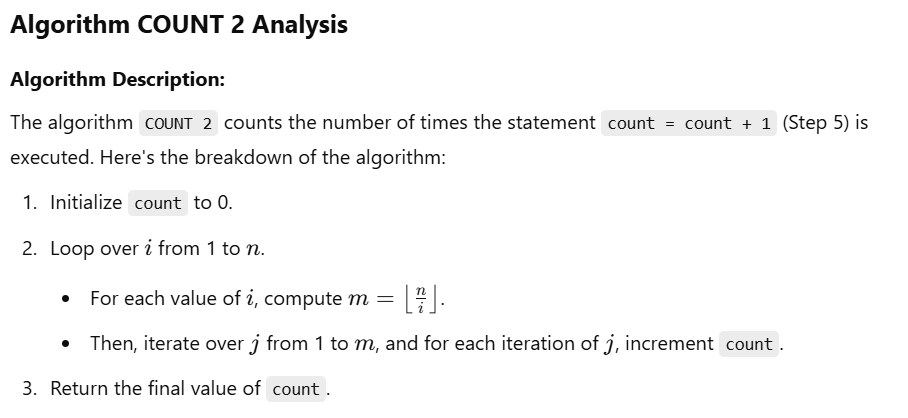


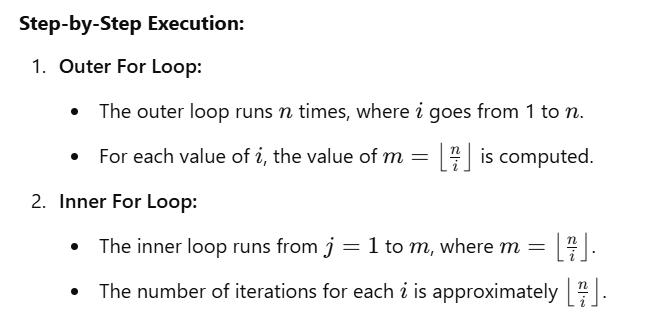


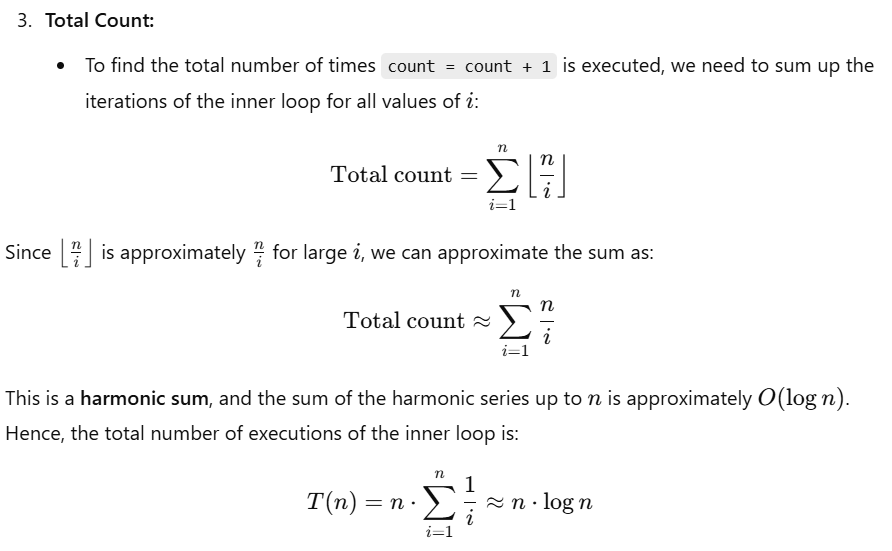


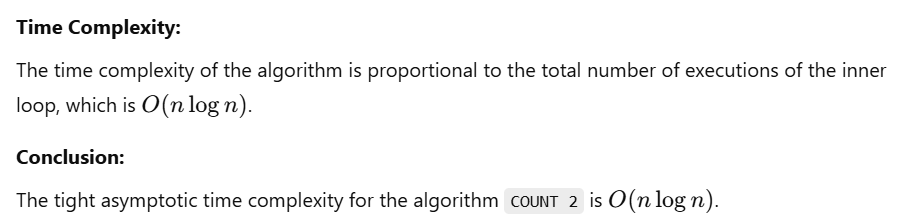


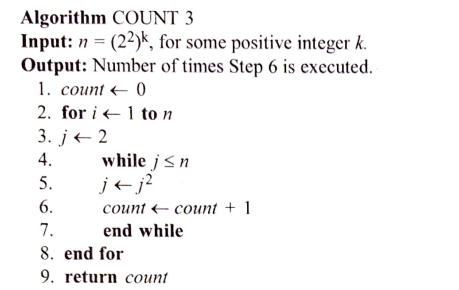


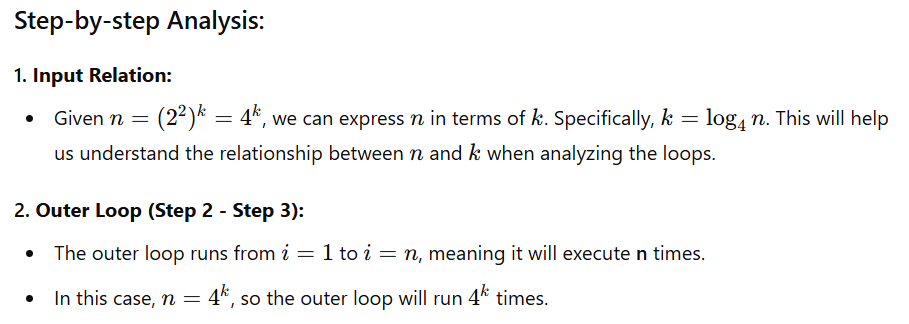


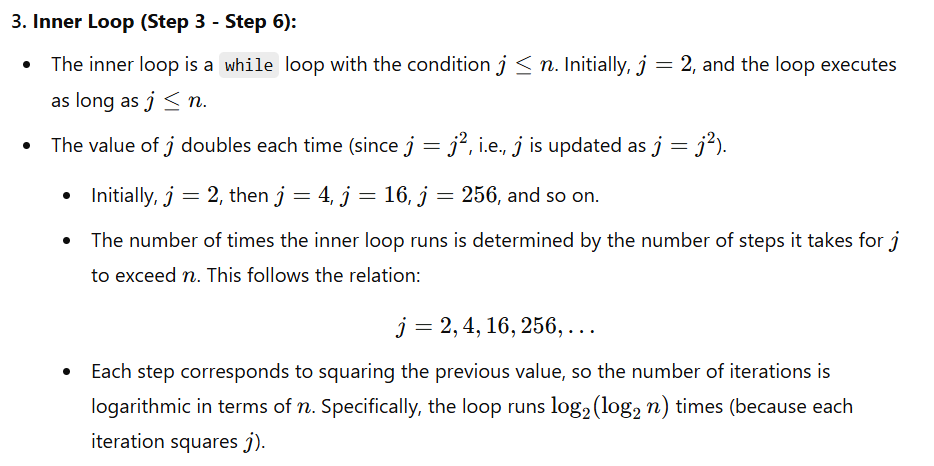


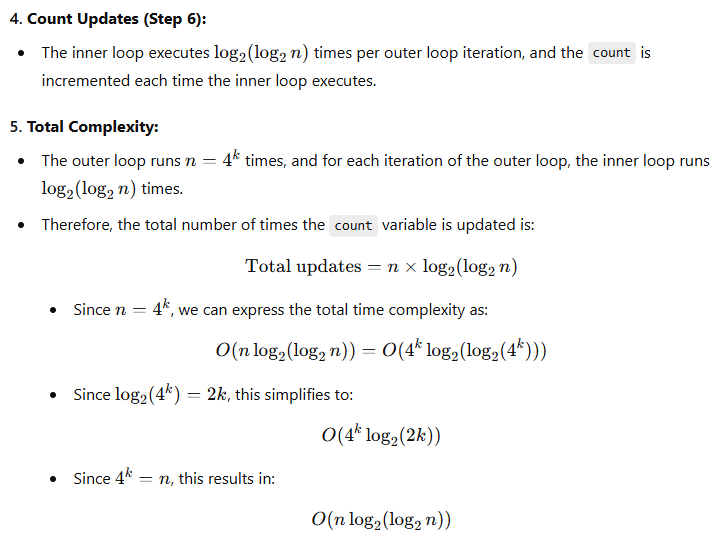


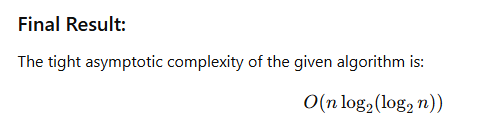


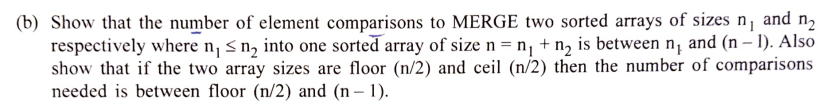


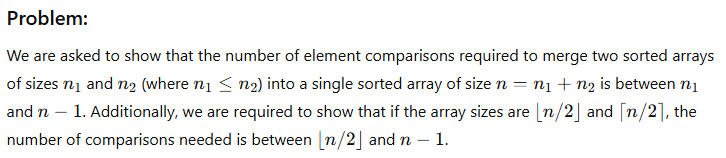


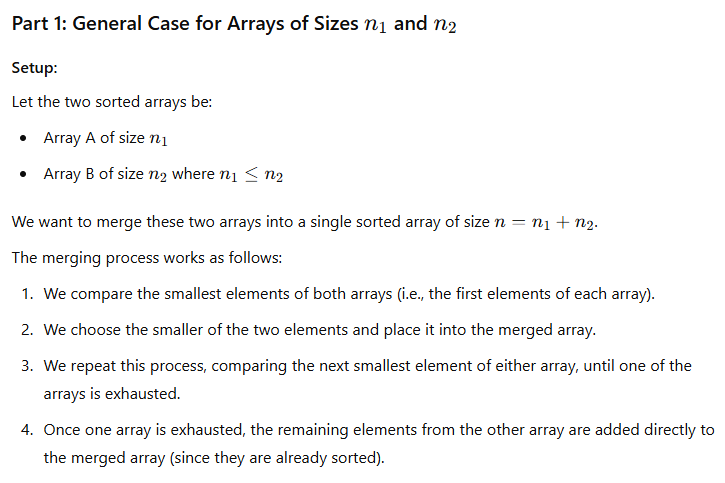


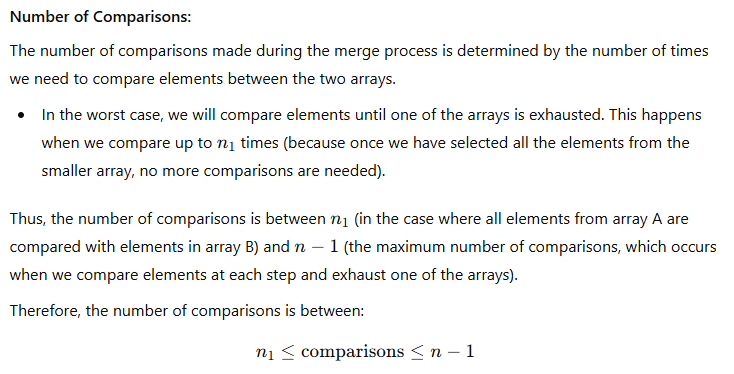


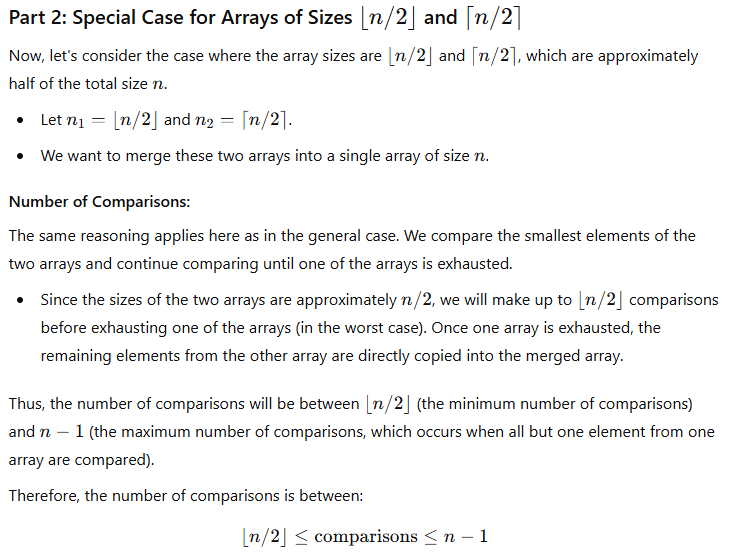


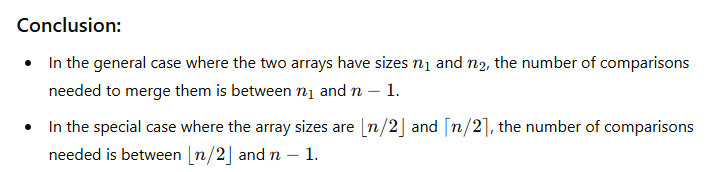












**(c) What is amortized analysis? Explain by citing appropriate examples.**

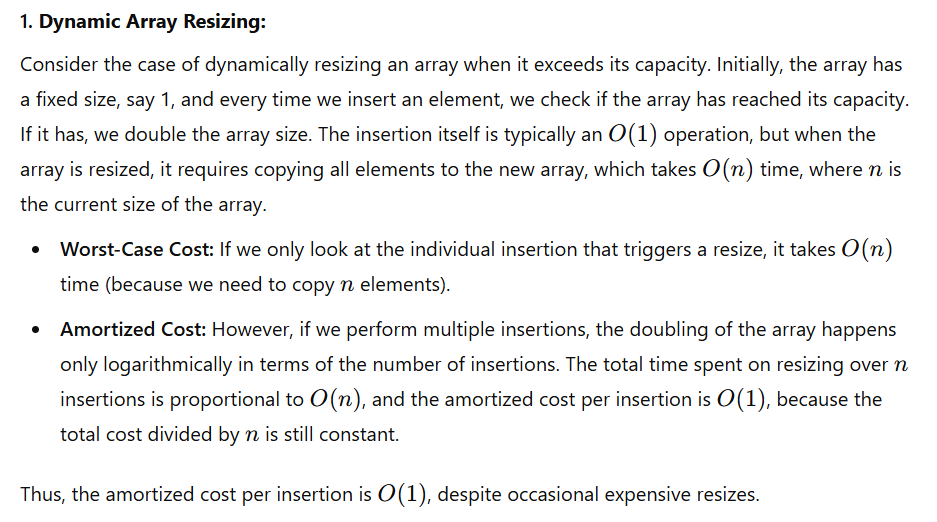
**Amortized Analysis:**

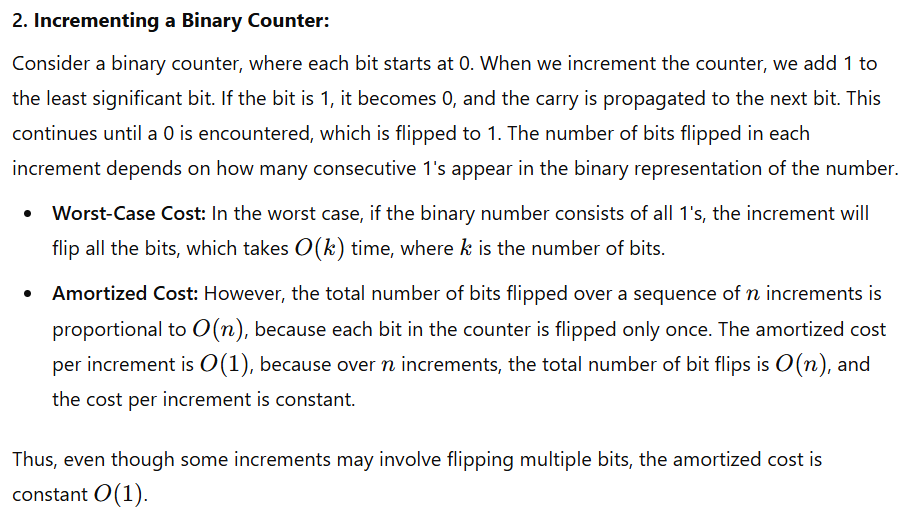
Amortized analysis is a technique used to analyze the average time complexity of operations in a sequence of operations, rather than analyzing each operation individually. It provides a way to ensure that even if some operations take a long time, the average time per operation over a sequence is still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

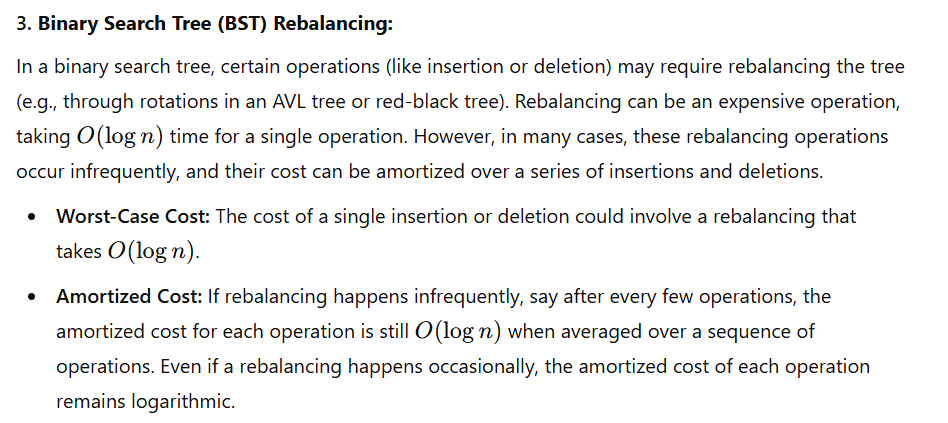
Amortized analysis is particularly useful when some operations have a high cost, but these operations occur infrequently, while most operations are cheap. By averaging over a series of operations, we get a better understanding of the overall performance of an algorithm.

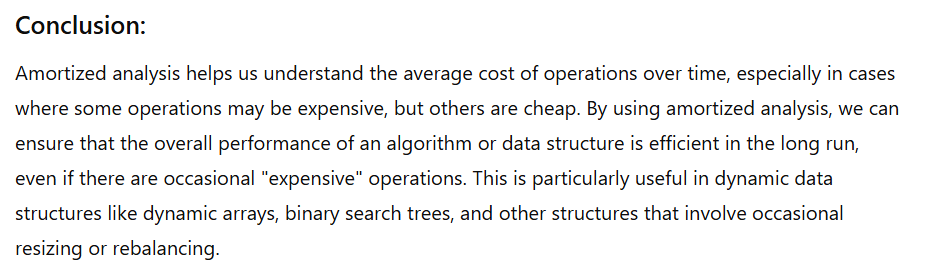
**Key Concepts in Amortized Analysis:**

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.





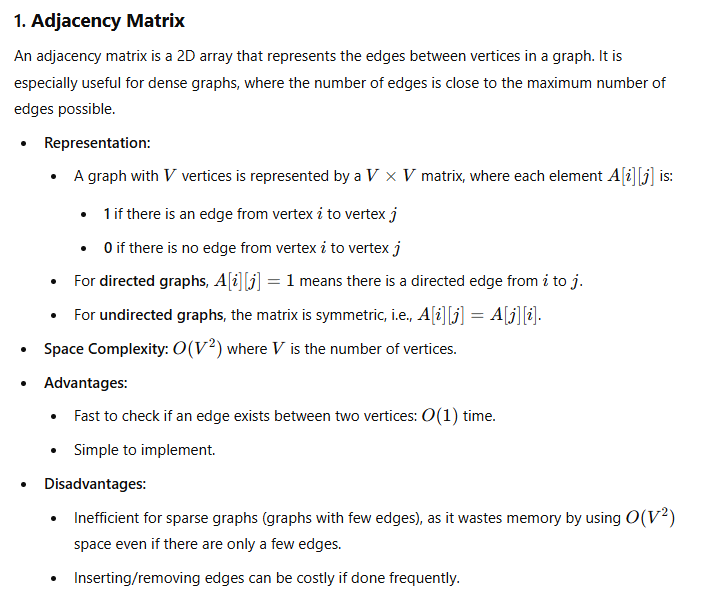


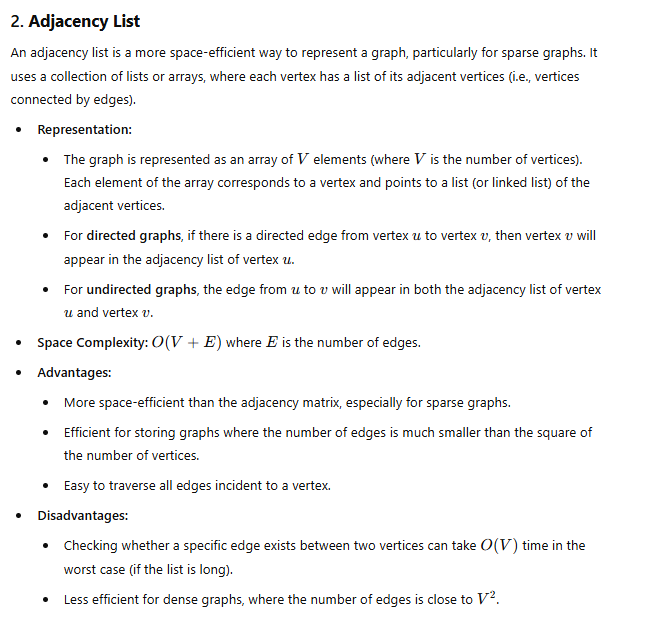


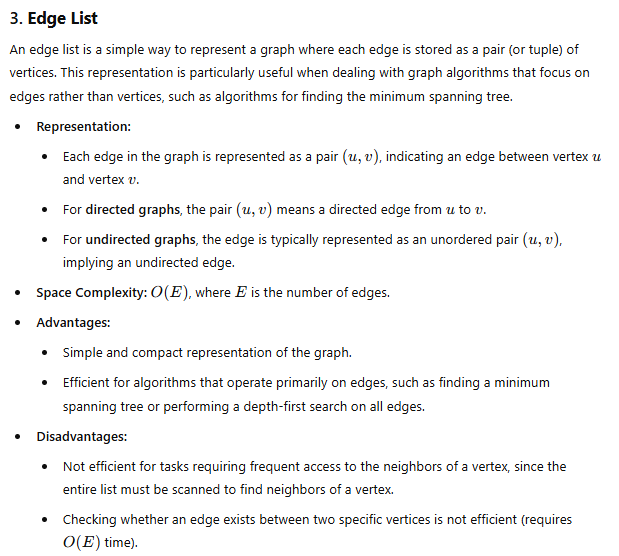
**(d) Explain the different techniques by which graphs are represented in computer memory.**

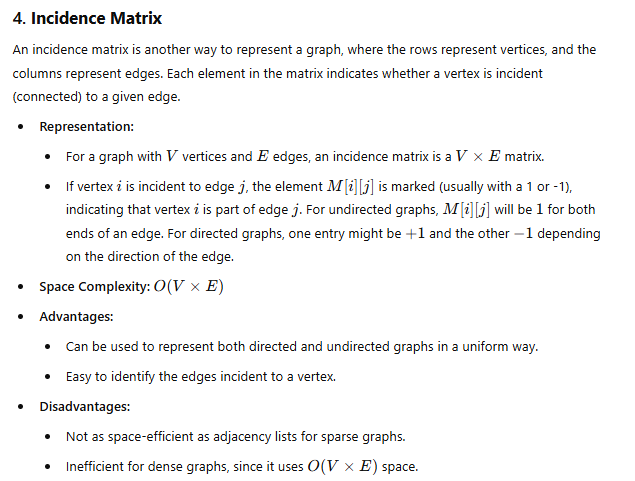
**Techniques for Representing Graphs in Computer Memory**

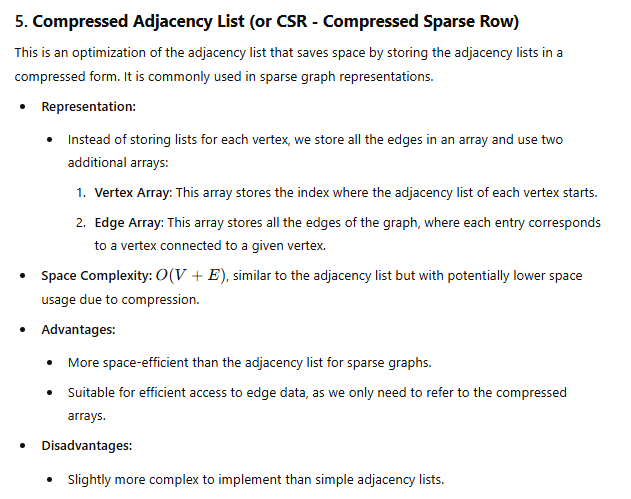
Graphs are essential data structures in computer science used to model relationships between objects. In computer memory, graphs can be represented in several ways, depending on the nature of the graph (such as whether it's sparse or dense) and the type of operations required (such as traversal, searching,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

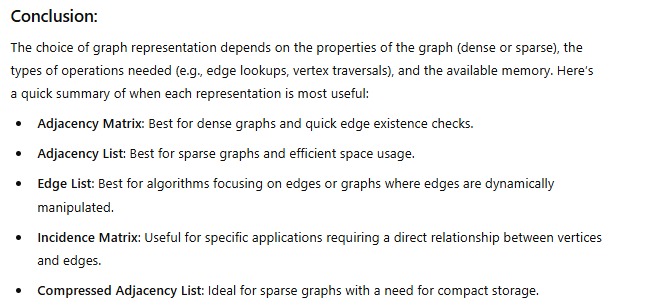






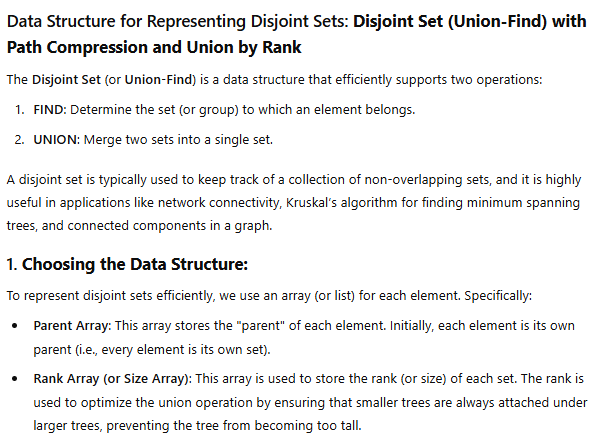


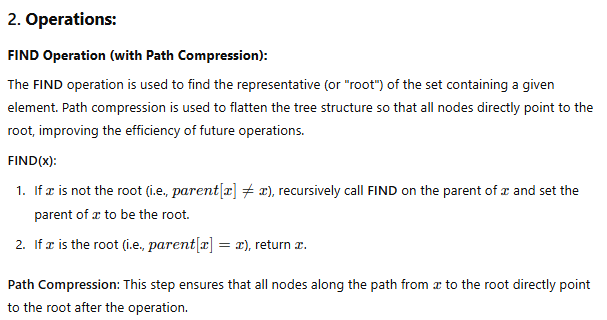


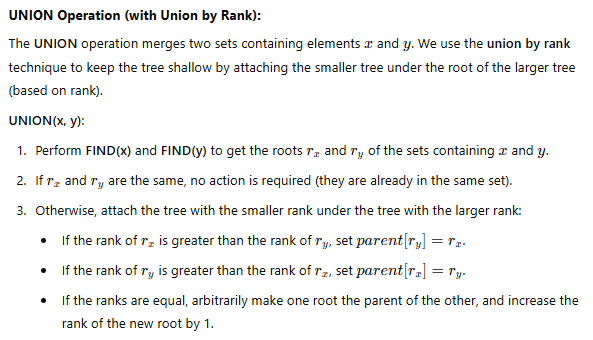


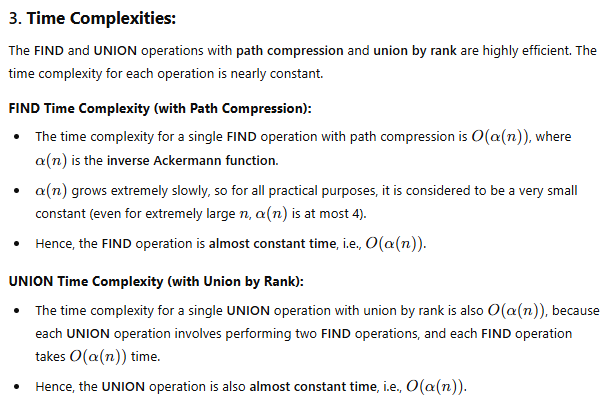
**2. (a) Choose an appropriate data structure for representation of disjoint sets. Write efficient FIND and**

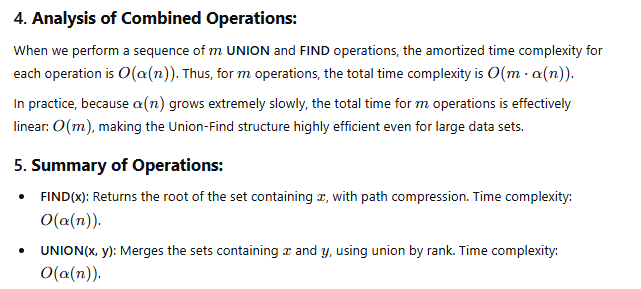
**UNION methods over the chosen data structure. Derive their time complexities.**











**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

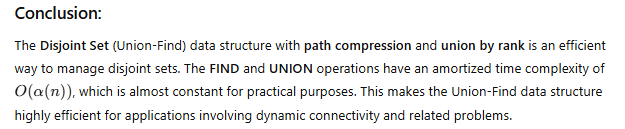
elif self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

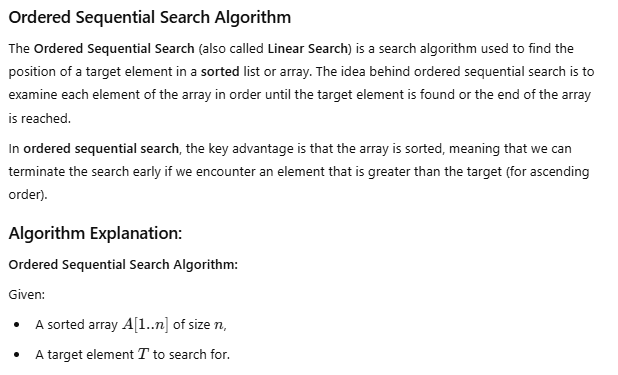
else:

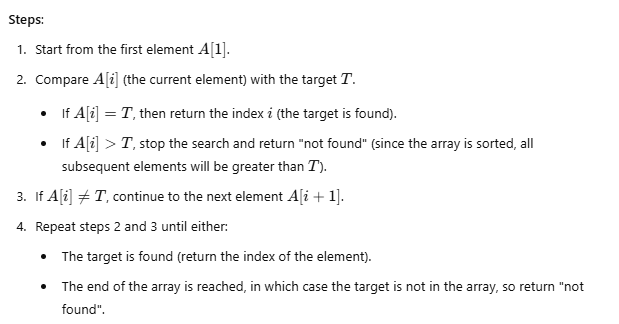
self.parent[rootY] = rootX

self.rank[rootX] += 1



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





**Pseudocode:**

def ordered\_sequential\_search(A, n, T):

for i in range(n):

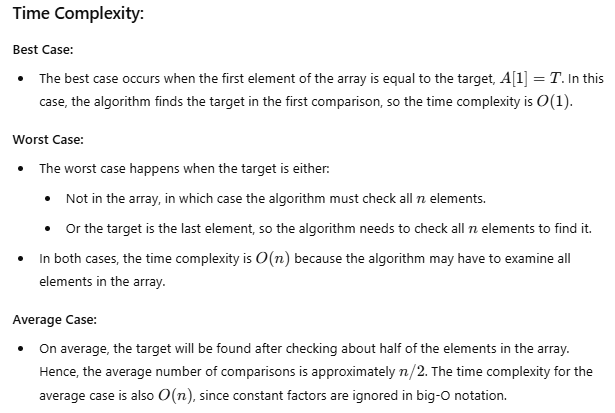
if A[i] == T:

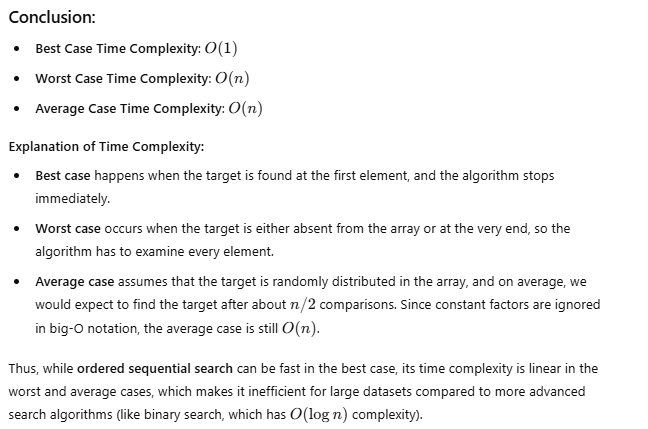
return i # Target found at index i

if A[i] > T:

break # No need to search further, since the array is sorted

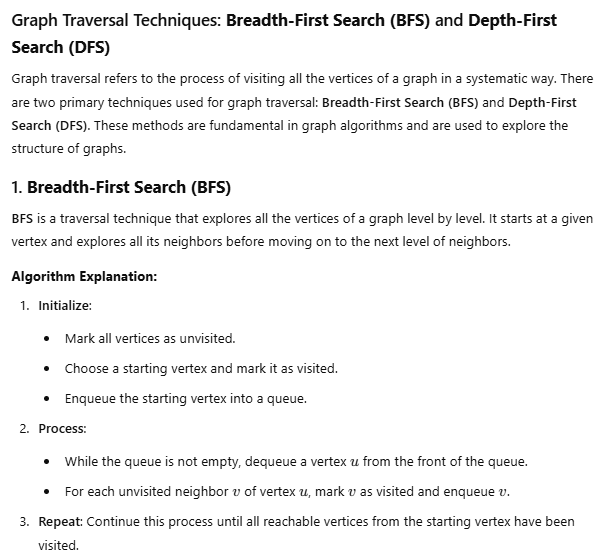
return -1 # Target not found





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

**traversal methods considering the data structures used.**



**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

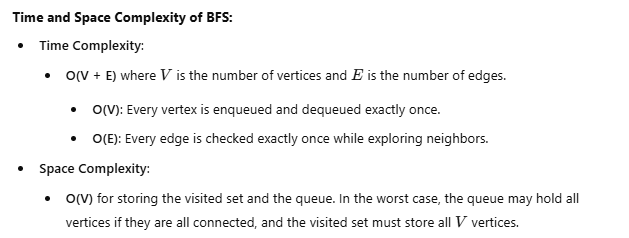
# Visit all the unvisited neighbors

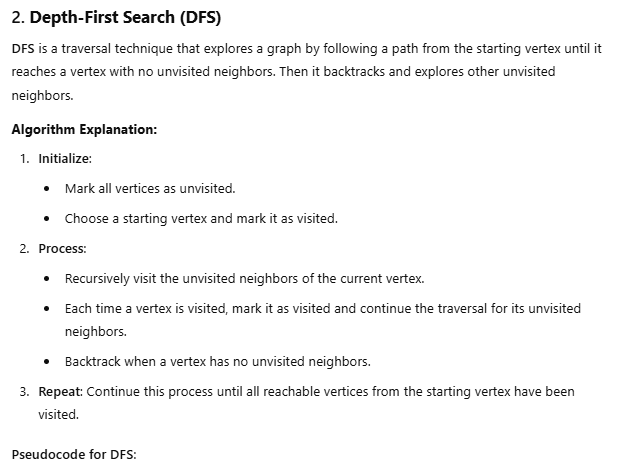
for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)





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

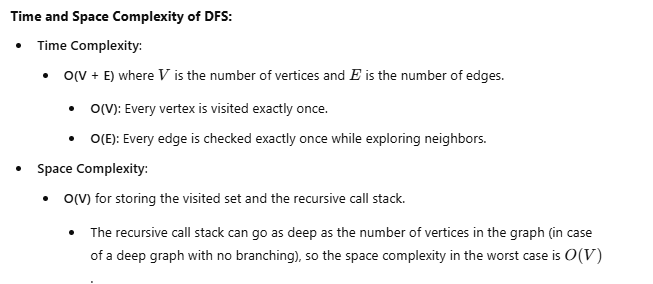
print(node, end=" ") # Visit the node

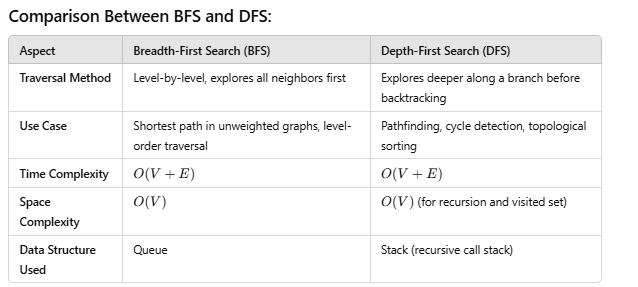
# Recursively visit all unvisited neighbors

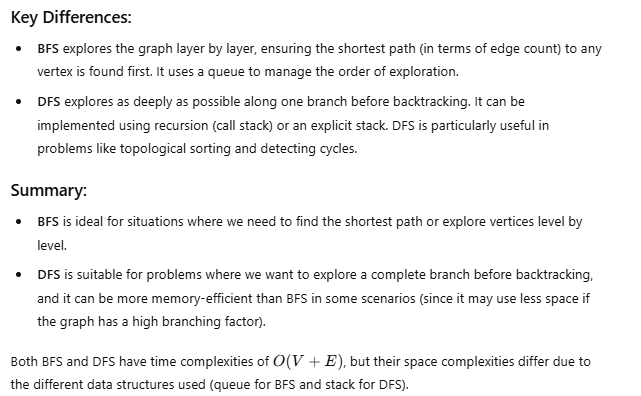
for neighbor in graph[node]:

if neighbor not in visited:

DFS(graph, neighbor, visited)

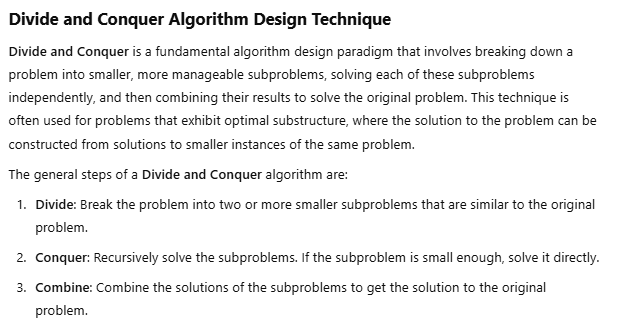


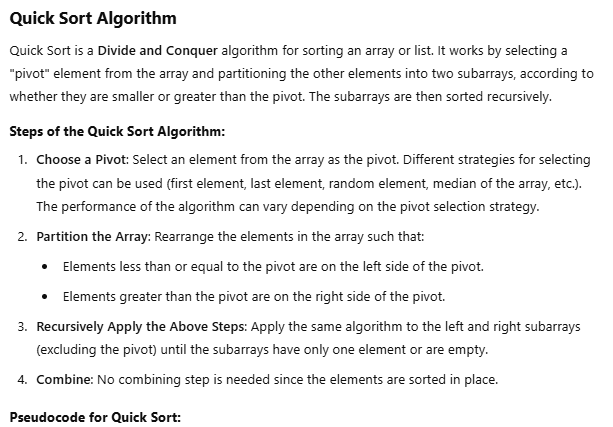




**4. Explain the divide and conquer algorithm design technique. Write and explain the Quick Sort algorithm.**

**Derive the worst case, average case and best case time and space complexities of the above method.**





def quick\_sort(arr):

if len(arr) <= 1:

return arr

else:

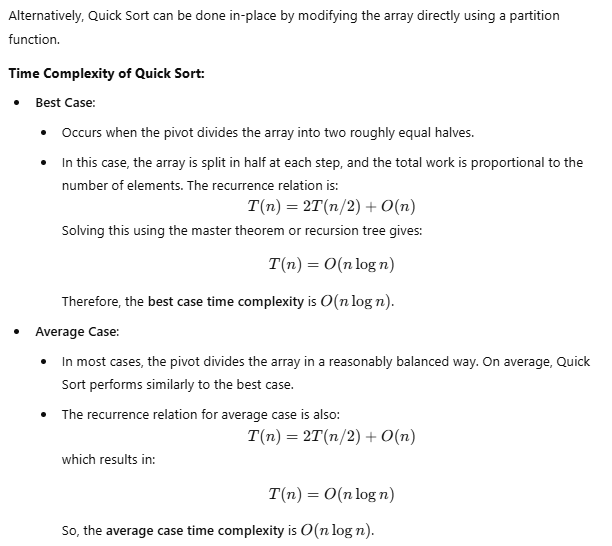
pivot = arr[len(arr) // 2] # Pivot can be chosen in many ways

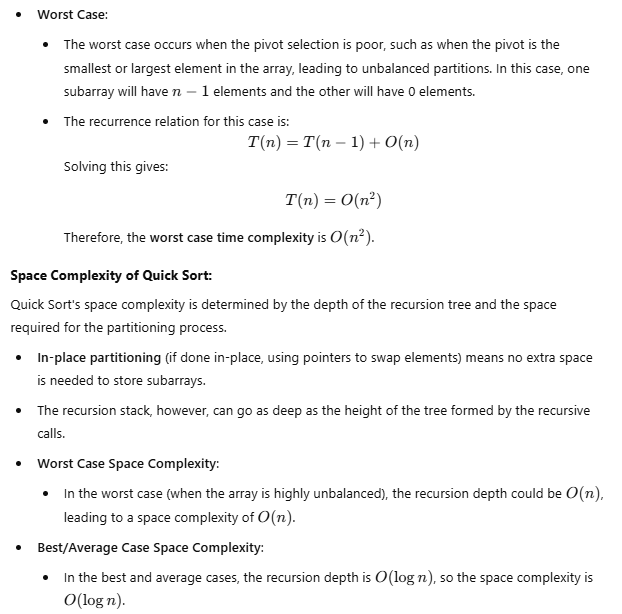
left = [x for x in arr if x < pivot] # Elements less than pivot

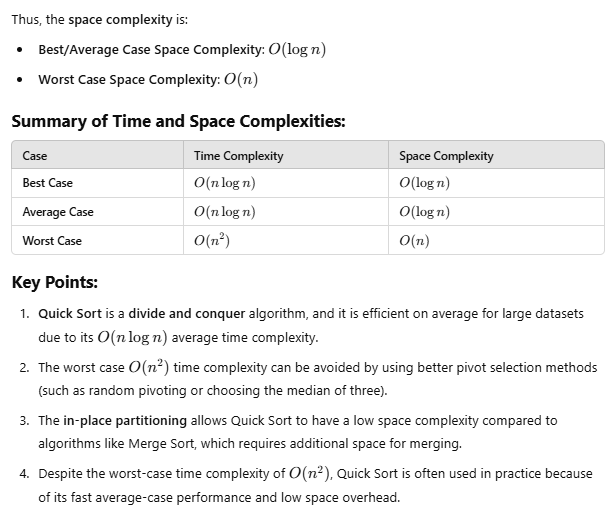
middle = [x for x in arr if x == pivot] # Elements equal to pivot

right = [x for x in arr if x > pivot] # Elements greater than pivot

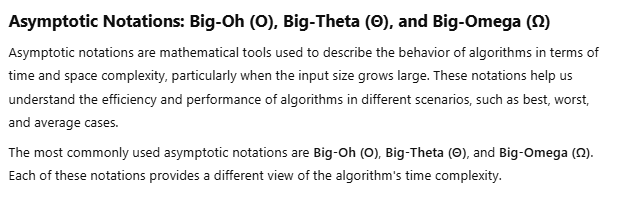
return quick\_sort(left) + middle + quick\_sort(right)

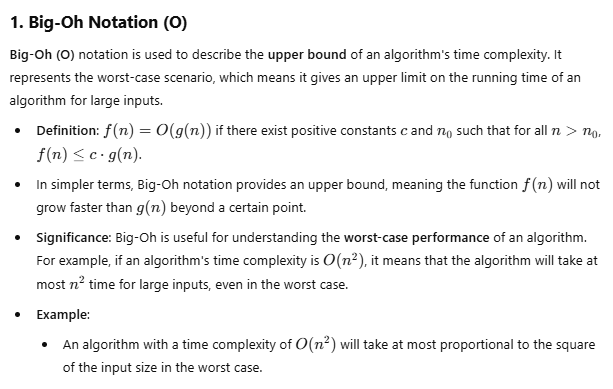


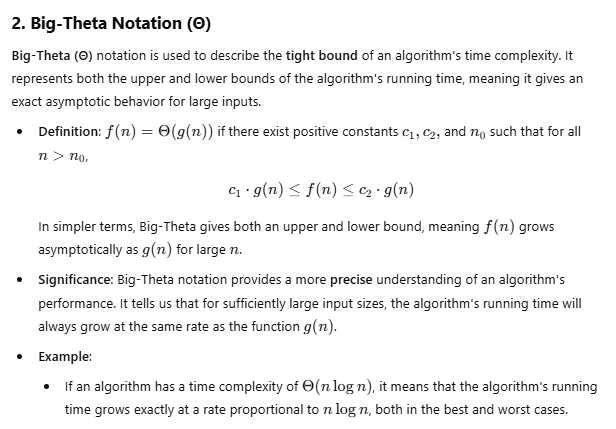


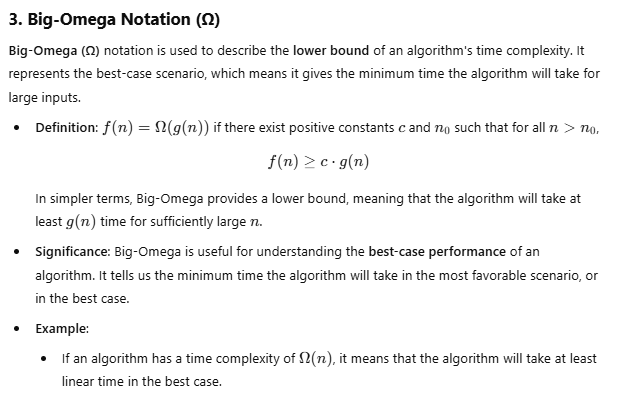


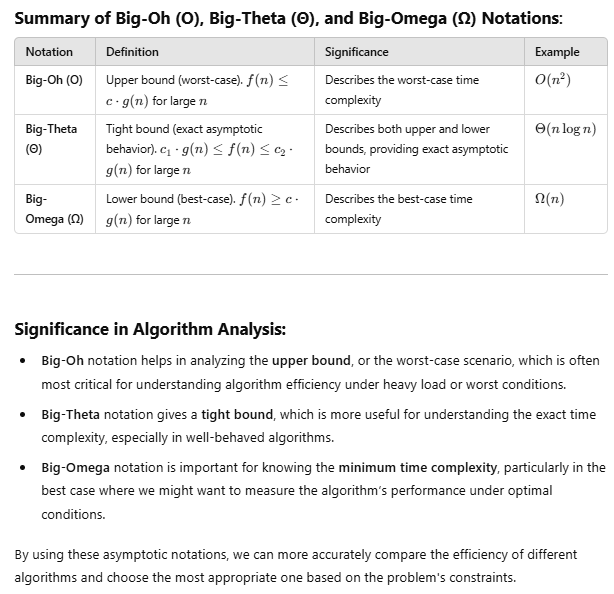


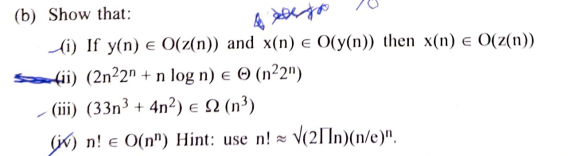


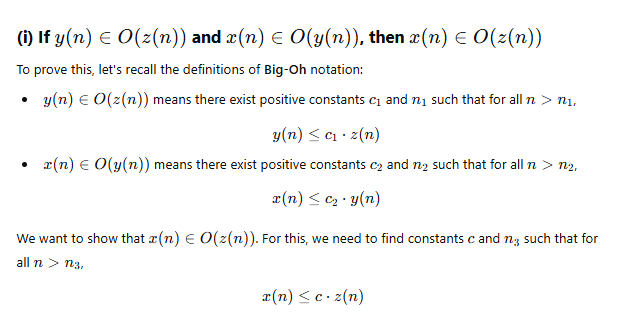


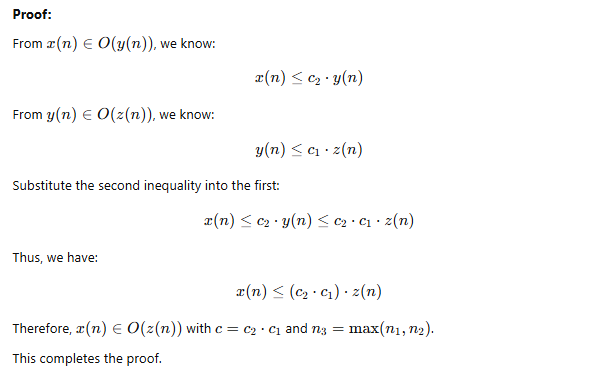


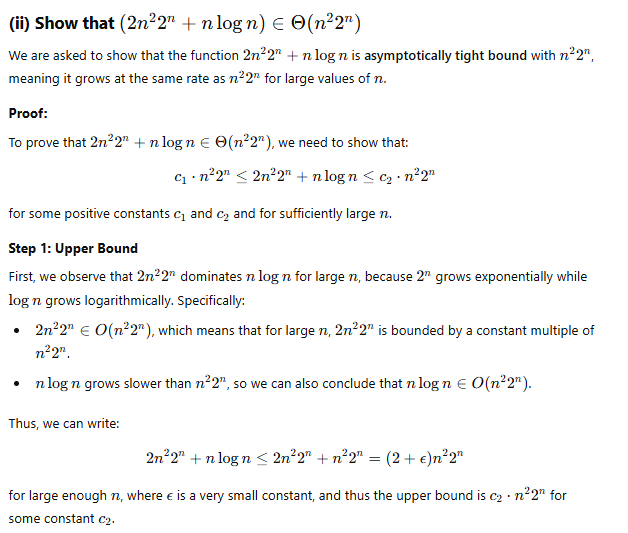


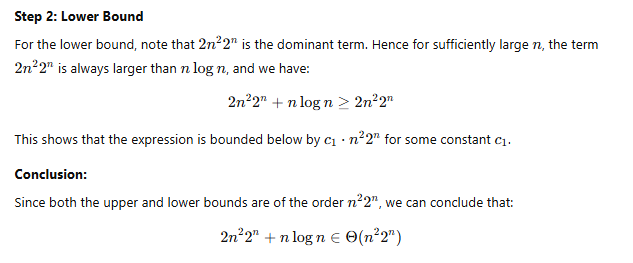


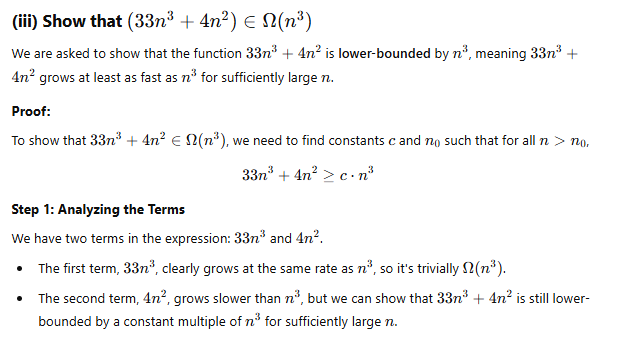


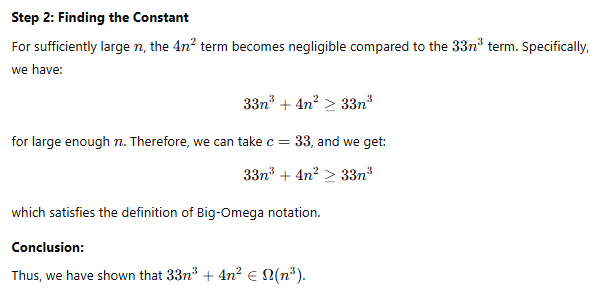


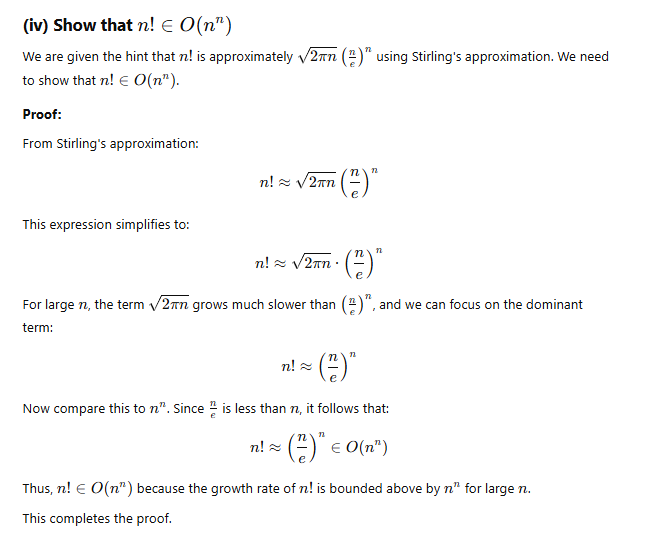












**(c) Draw the binary decision tree for binary search with n=14 where n is the total number of elements.**

