

***NAME: Abdullah liaqat***

***ROLL NO: 14687***

***SECTION: A***

***SUBJECT: Data structure and Algorithm***

***Assignment :1***

***SUBMITTED TO: Sir Jamal Abdul Ahad***

***Git hub link:***

<https://github.com/Abdullah-9696/assignment-DSA-.git>

***Assignment :1***

***Chapter 1***

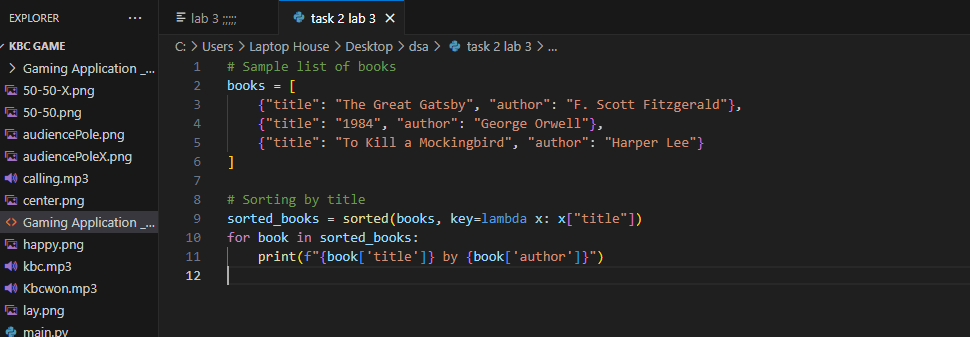
***The role of algorithm in computing***

**QUESTION NO** 1 Describe your own real-world example that requires sorting. Describe one that requires ûnding the shortest distance between two points.

**Real-World Examples of Sorting and Shortest Distance**

**Sorting Example:** Imagine a library that needs to organize its books on shelves. The librarian sorts books by genre, author, or title to make it easier for patrons to find what they’re looking for. This requires a sorting algorithm to efficiently arrange the books.

**Shortest Distance Example:** Consider a delivery service that needs to determine the most efficient route for its drivers to deliver packages across a city. The service uses algorithms like Dijkstra’s to find the shortest distance between the starting point and multiple delivery locations, ensuring timely deliveries.

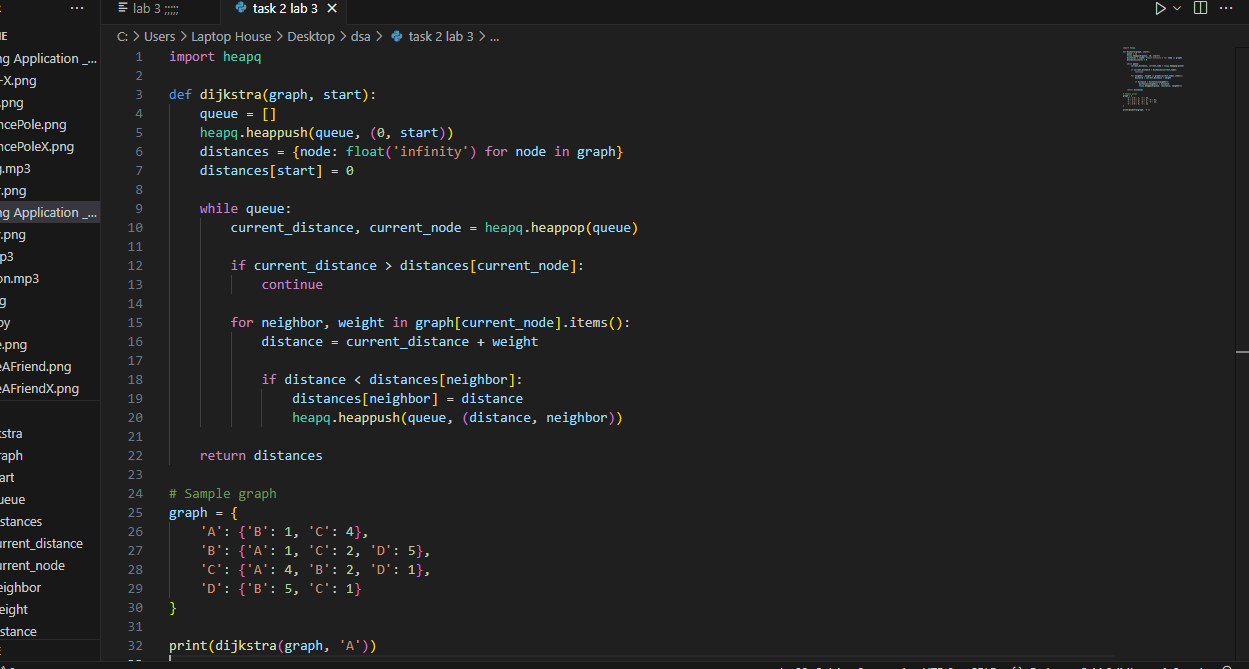


**QUESTION NO 2** Other than speed, what other measures of efûciency might you need to consider in a real-world setting?.

**Other Measures of Efficiency**

Aside from speed, other measures of efficiency to consider in real-world settings might include:

* **Memory Usage:** How much memory does the process consume?
* **Scalability:** Can the solution handle increased workload or larger datasets?
* **Maintainability:** How easy is it to update or modify the solution?
* **Reliability:** How consistently does the solution perform without failure?
* **Cost-effectiveness:** Does the solution provide a good return on investment in terms of resources spent versus benefits gained?



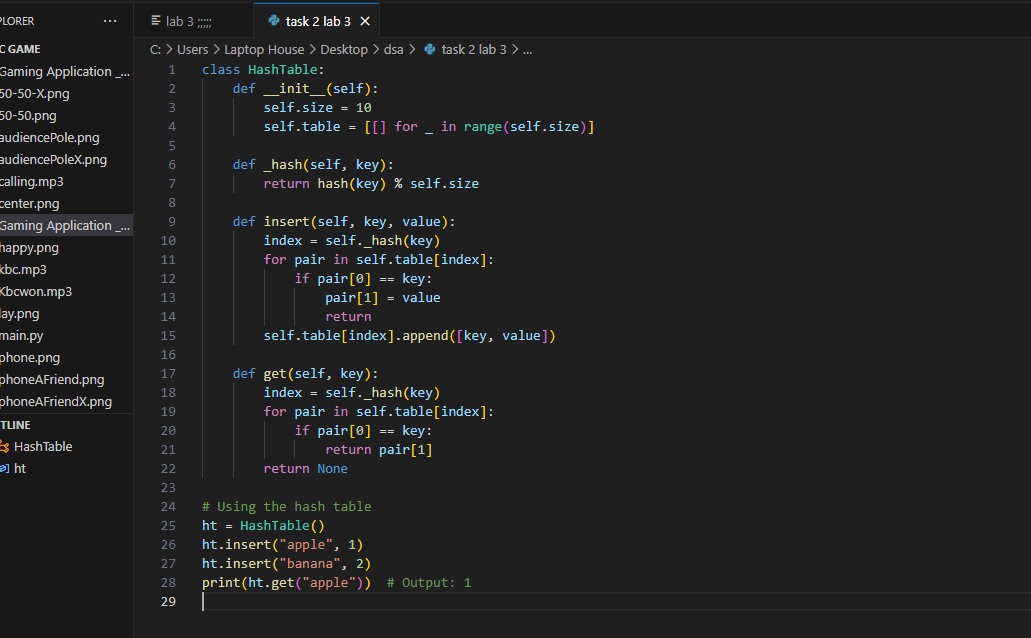
**QUESTION NO 3** give answer for dsa and give me without copy write Select a data structure that you have seen, and discuss its strengths and limitations.

**Data Structure Discussion**

**Data Structure:** Hash Table  
**Strengths:**

* **Fast Access:** Provides average-case constant time complexity (O(1)) for insertions, deletions, and lookups, making it very efficient for accessing data.
* **Flexible Key Types:** Can use various types of keys (strings, numbers) for storing values.

**Limitations:**

* **Collisions:** When two keys hash to the same index, it can lead to collisions, requiring additional handling like chaining or open addressing.
* **Memory Consumption:** Can be memory-intensive, especially if the load factor is low, leading to wasted space.
* **Order:** Does not maintain any order of elements, which can be a disadvantage when order matters. 

**QUESTION NO 4** How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

**Similarities and Differences in Shortest-Path and Traveling-Salesperson Problems**

**Similarities:**

* Both involve finding optimal paths in a graph.
* Each can be represented using nodes (points) and edges (connections).

**Differences:**

* **Shortest-Path Problem:** Focuses on finding the minimum distance between a single starting point and a specific destination.
* **Traveling-Salesperson Problem (TSP):** Involves finding the shortest possible route that visits each location exactly once and returns to the origin, making it more complex and computationally challenging.

**QUESTION NO 5** Suggest a real-world problem in which only the best solution will do. Then come up with one in which <approximately= the best solution is good enough.

**Real-World Problems: Best vs. Approximate Solutions**

**Best Solution Required:** In healthcare, determining the optimal treatment plan for a patient can be crucial. The best treatment can significantly impact patient outcomes, so there’s little room for error.

**Approximate Solution Good Enough:** In web search engines, while they strive to return the most relevant results, an approximate match can often be acceptable. Users may not always require the absolute best result, as long as the information is relevant.

**QUESTION NO 6** Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

**Real-World Problem with Varying Input Availability**

A common example is traffic management in a smart city. Sometimes, all traffic data (e.g., current traffic flow, accidents, road closures) is available before making decisions about traffic signal timings or route adjustments. Other times, data arrives in real-time, such as during an unexpected event or an accident, requiring quick adjustments to optimize traffic flow dynamically.

**QUESTION NO 7** Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

**Example Application: Online Recommendation Systems Overview:**

Online recommendation systems are widely used in platforms like Netflix, Amazon, and Spotify to suggest products, movies, or music to users based on their preferences and behavior. These systems require algorithmic content at the application level to analyze data and generate recommendations.

**Algorithmic Components:**

1. **Collaborative Filtering**:

 This technique analyzes user behavior and preferences to recommend items. For example, if User A and User B have similar tastes, the system can recommend items that User B liked to User A.

**Algorithmic Content:**

Algorithms like Matrix Factorization (e.g., Singular Value Decomposition) are used to decompose user-item interaction matrices to identify patterns and similarities.

1. **Content-Based Filtering:**

This approach recommends items based on the attributes of items that a user has liked in the past. For example, if a user liked action movies, the system will recommend other action movies.

**Algorithmic Content**:

Algorithms that utilize techniques such as TF-IDF (Term Frequency-Inverse Document Frequency) for text analysis or cosine similarity to measure item similarity based on features.

1. **Hybrid Methods:**

Many systems combine both collaborative and content-based filtering to improve accuracy and address the limitations of each approach.

**Algorithmic Content:**

Algorithms that integrate results from both methods and may use machine learning techniques to optimize recommendations based on user feedback.

**Importance:**

**User Experience:**

By providing personalized recommendations, these systems enhance user satisfaction and engagement.

**Business Impact:**

Effective recommendations can lead to increased sales, higher user retention, and improved customer loyalty.

**Example in Practice:**

**Netflix:**

Uses complex algorithms to analyze user viewing history, ratings, and behaviors to suggest new shows and movies. For instance, if you often watch thrillers, Netflix will recommend other thrillers that similar users have enjoyed.

**Conclusion:**

Online recommendation systems exemplify applications that heavily rely on algorithmic content to process large datasets, derive insights, and deliver personalized experiences. The effectiveness of these systems hinges on sophisticated algorithms that continuously learn and adapt to user preferences. If you have more questions or need further examples, feel free to ask.

**QUESTION NO 8** Suppose that for inputs of size n on a particular computer, insertion sort runs in 8𝑛2 steps and merge sort runs in 64 n log n steps. For which values of n does insertion sort beat merge sort?

To determine for which values of n insertion sort beats merge sort, we need to compare their running times:

1. **Insertion Sort**:8𝑛2
2. **Merge Sort**: 64n𝑙𝑜𝑔2 n

We want to find the values of n for which:

8𝑛2<64n𝑙𝑜𝑔2

**Step 1: Simplify the Inequality** Divide both sides by 8n (assuming n>0):

n<8𝑙𝑜𝑔2𝑛

**Step 2: Analyze the Inequality**

To solve the inequality n<8𝑙𝑜𝑔2𝑛

𝑛

<n

𝑙𝑜𝑔2𝑛

**Step 3: Find Values of n**

𝑛

This inequality indicates that we need to check values of n to see where is less than 8.

𝑙𝑜𝑔2𝑛

1. **Try small values of n**:

**For n=1:**

1

𝑙𝑜𝑔21 is undefined(since 𝑙𝑜𝑔2 1 = 0

**For n=2:**

2 2

= = 2 < 8

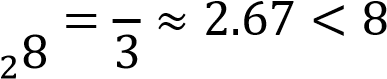
𝑙𝑜𝑔22 1

**For n=4:**

4 4

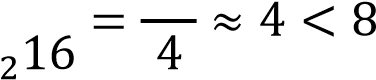
= = 2 < 8

𝑙𝑜𝑔24 2 **For n=8:**

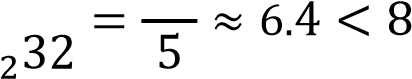
𝑙𝑜𝑔

**For n=16:**

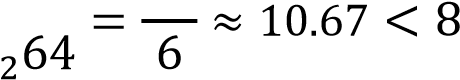
𝑙𝑜𝑔

**For n=32:**

𝑙𝑜𝑔

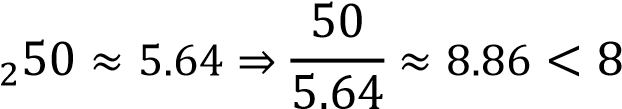
**For n=64:**

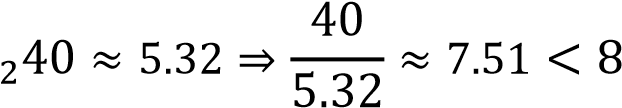
𝑙𝑜𝑔

1. **Continue Checking Larger Values**:

**For n=50:**

𝑙𝑜𝑔

**For n=40:**

𝑙𝑜𝑔

***THE END OF CHAPTER 1***

***Chapter 2***

**Analyzing Algorithms**

**QUESTION NO 1**illustrate the operation of I NSERTION-SORT on an array initially containing the sequence {31, 41,59, 26,41, 58}. def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

## j = i -

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

### j -= 1

arr[j + 1] = key array = [31, 41, 59, 26, 41, 58] insertion\_sort(array) print("Sorted array:", array)

**QUESTION NO 2** Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array A[1:n]. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in A[1:n]. SUM-ARRAY(A, n):

sum = 0 for i from 1 to n do sum = sum + A[i] return sum

**Loop Invariant:**

**At the start of each iteration of the loop, the variable sum contains the sum of the elements in the subarray A[1…i−1].**

**Initialization:**

* Before the loop begins, sum is initialized to 0 (line 1). At this point, for i=1, the subarray A[1…0] (an empty subarray) has a sum of 0. Thus, the invariant holds true before the first iteration.

**Maintenance:**

* Assume that the invariant holds at the start of iteration iii (i.e., sum contains the sum of the elements in A[1…i−1]).
* During the 𝑖𝑡ℎiteration (line 3), we execute sum = sum + A[i].
* After this operation, sum now contains the sum of the elements in A[1…i], because it adds the 𝑖𝑡ℎelement to the sum of the previous elements.
* Thus, the invariant continues to hold for the next iteration.

**Termination:**

* The loop terminates when iii exceeds n. At this point, the loop has iterated for i=1 to n.  By the loop invariant, when the loop exits, sum contains the sum of the elements in A[1…n].
* Therefore, upon termination, sum correctly represents the total sum of the array A[1…n].

**QUESTION NO 3** Rewrite the I NSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing o def insertion\_sort\_decreasing(arr): for i in range(1, len(arr)):

key = arr[i] j = i - 1 while j >= 0 and arr[j] < key:

arr[j + 1] = arr[j] j -= 1

arr[j + 1] = key

array = [31, 41, 59, 26, 41, 58] insertion\_sort\_decreasing(array) print("Sorted array (decreasing order):", array)

**QUESTION NO 4** Consider the searching problem:

**Input:**

A sequence of n numbers {𝑎1,𝑎2,……𝑎𝑛} stored in array A[1 : n]and a value x.

**Output:**

An index i such that x equals A[i] or the special value NIL if x does not appear in A.

**Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.**

LINEAR-SEARCH(A, n, x):

for i = 1 to n do if A[i] == x then return i return NIL

**Loop Invariant**

At the start of each iteration of the loop, if **x** is in the array **A[1…n],** then it is located in **A[1…i−1]** or the search has not yet checked **A[i].**

**Properties of the Loop Invariant Initialization**:

Before the first iteration (when i=1), the invariant holds because the algorithm has not yet checked any elements, so it correctly states that if x is in the array, it hasn't been found in A[1…0] (which is empty). Thus, the invariant holds at initialization.

**Maintenance:**

* Assume the invariant holds at the beginning of the 𝑖𝑡ℎiteration (i.e., if x is in the array, it is in A[1…i−1] or has not yet been checked).
* In this iteration, we check if A[i]==x

If true, we return iii, and the search is successful.

If false, the invariant still holds because we have now confirmed that xxx is not in A[i] and we proceed to check the next element A[i+1].

* Thus, after this iteration, if xxx is in the array, it must be in A[1…i] or the search continues to A[i+1] **Termination**:
* The loop terminates when iii exceeds n. At this point, we have checked all elements in the array.
* If the loop exits without returning an index, it means x was not found in any of the elements A[1…n], so we return NIL.
* By the invariant, we can conclude that if x exists in the array, it will have been found; otherwise, we correctly return NIL.

**QUESTION NO 5** Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays

A[0:n-1] and B[0:n-1], where each element is either 0 or 1, a = 𝑖, and b

𝑖. The sum c = a + b of the two integers should be stored in binary form in an (n+1)element array C [0:n], where c =  2 i . Write a procedure ADD-BINARY-

INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

ADD-BINARY-INTEGERS(A, B, n):

#Initialize the result array C with size n + 1 (for possible carry) C = array of size (n + 1 carry = 0 #Initialize carry to 0

#Iterate through each bit from least significant to most significant for i from 0 to n - 1 do

#Calculate the sum of the current bits and the carry sum = A[i] + B[i] + carry

#C[i] will be the least significant bit of the sum C[i] = sum mod 2 carry = sum C[n] = carry return C

**Explanation:**

**Initialization**:

* Create an array C of size n+1to store the result.
* Initialize a variable carry to 0, which will hold the carry from the addition of bits.

**Loop through bits:**

Loop from 0 to n−1 to process each bit of arrays A and B:

* Compute the sum of the corresponding bits A[i] and B[i], along with any carry from the previous addition.
* Store the least significant bit of the result in C[i].
* Update the carry for the next iteration using integer division by 2.

**Final carry:**

* After the loop, check if there is a carry left and store it in C[n].

**Return result:**

* Finally, return the array C, which now contains the binary sum of A and B.

**Example Usage:**

If you want to add the binary numbers A=[1,0,1] (which represents the number 5 in decimal) and B=[1,1,0] (which represents the number 6 in decimal), you can call the function as follows:

A = [1, 0, 1] // 5 in binary B = [1, 1, 0] // 6 in binary n = 3

C = ADD-BINARY-INTEGERS(A, B, n)

// C will contain the result: [0, 0, 1, 1] which represents 11 in binary

**QUESTION NO 6** Express the function 𝑛3/1000 C+100𝑛2 - 100n + 3 in terms of ‚ Θ-notation. To express the function f(n)= 𝑛3 +100𝑛2−100n+3in term of Θ-notation ,we need to identify the

1000 dominant term as n grow larger:

**Analyzing the Function:**

1. **Identify the Growth of Each Term**:
   * The term 𝑛3 grows as O(𝑛3 ). 1000
   * The term 100𝑛2 grows as O(𝑛2).
   * The term −100ngrows as O(n).
   * The constant 3 is O(1).
2. **Dominant Term**:

As n becomes very large, 𝑛3 will dominate the growth of the function since it is a

1000 cubic term, which grows faster than the quadratic or linear terms.

**Conclusion:**

Thus, the function f(n) can be expressed in Θ-notation as:

f(n)=Θ(𝑛3)

This notation indicates that f(n) grows at the same rate as 𝑛3for large n.

**QUESTION NO 7** Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? Using ‚-notation, give the average-case and worst-case running times of linear search. Justify your answers

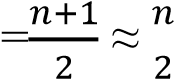
**Linear Search Analysis**

Linear search is a simple algorithm that checks each element in an array sequentially until the desired element is found or the end of the array is reached.

**Average Case**

1. **Average Number of Comparisons**:

* If the array has n elements, the desired element could be in any position with equal likelihood.
* On average, the element will be found after checking about half of the elements.
* Therefore, the average number of comparisons is:

Average comparisons

**Average-Case Running Time:**

* Using Big O notation, the average-case running time is: O(n)

𝑛

* Justification: Even though it’s comparisons on average, we express it in terms of Big O

2 as O(n) because we focus on the highest-order term as n grows large.

**Worst Case**

**Worst Number of Comparisons**:

* The worst-case scenario occurs when the desired element is not present in the array or is the last element.
* In both cases, all n elements must be checked.
* Therefore, the worst number of comparisons is:

Worst comparisons = n **Worst-Case Running Time**:

* The worst-case running time is: O(n)
* Justification: Since the search must check every element in the worst case, the time complexity is linear with respect to the number of elements.

**QUESTION NO 8** How can you modify any sorting algorithm to have a good best-case running time?

**Insertion Sort Optimization Best Case:**

Insertion Sort already has a best-case time complexity of O(n) when the input array is already sorted or nearly sorted. You can improve it further by adding a check at the beginning to see if the array is already sorted.

**Implementation:**

Before proceeding with the sort, iterate through the array once to check if each element is less than or equal to the next element. If this condition holds for the entire array, you can conclude it’s sorted and return early.

**Merge Sort Modification Best Case:**

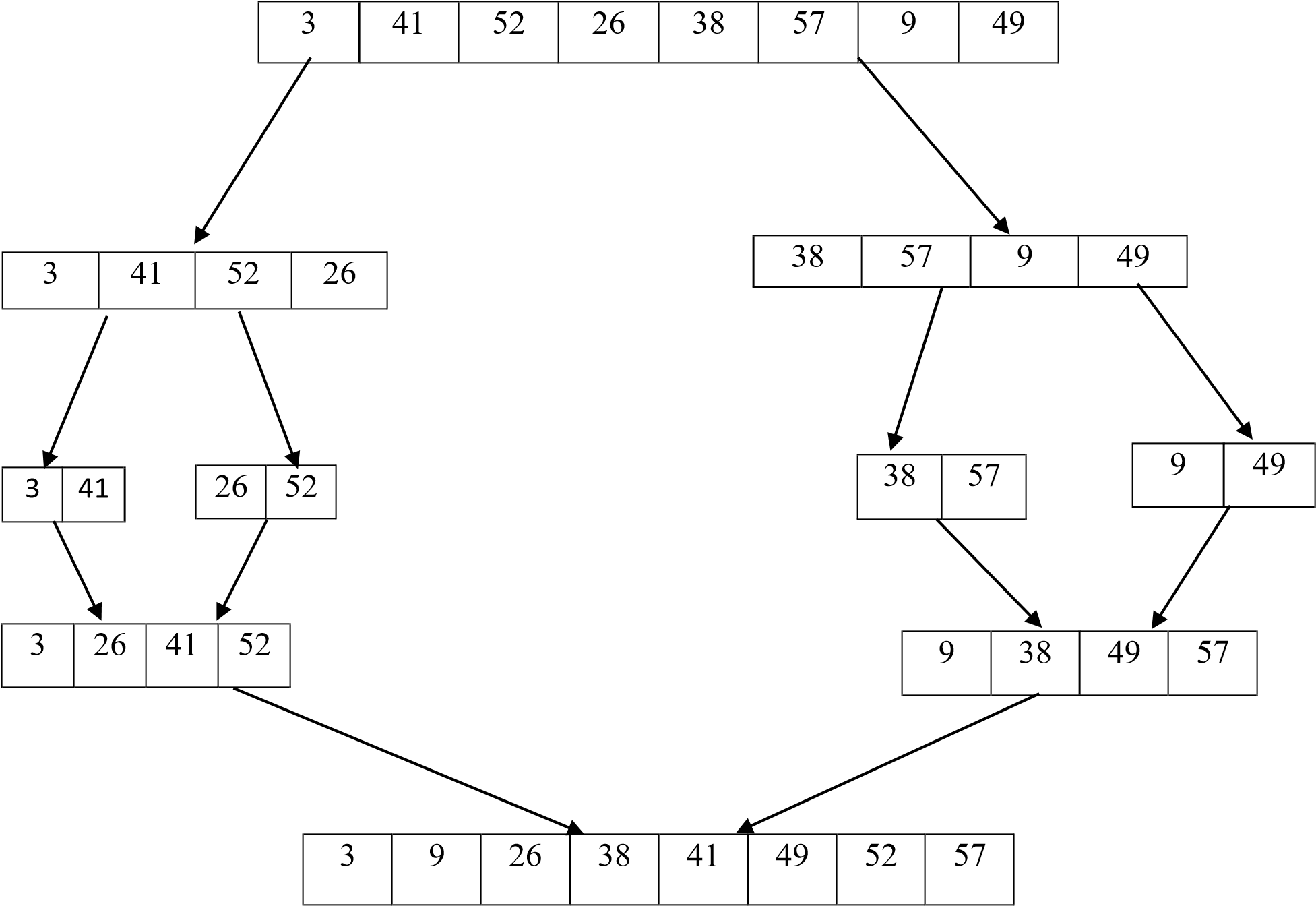
Merge Sort typically has a time complexity of O(nlogn), but you can improve its best-case performance by recognizing if subarrays are already sorted.

**Implementation**:

Before merging two sorted subarrays, check if the last element of the first subarray is less than or equal to the first element of the second subarray. If true, the entire array is already sorted, and no further action is necessary.

**QUESTION NO 9** Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence {3, 41,52, 26, 38,57,9,49}.

**Merge sorting**



**QUESTION NO 10** The test in line 1 of the MERGE-SORT procedure reads r, if P >= r rather then p≠ r if merge sort is called with p > r then subarray A[p:r] is empty is empty. Argue that as long as the initial call of MERGE-SORT.(A, 1,n) has n ≥ 1, the test p≠r suffices to ensure that no recursive call has p > r.

**Initial Call:**

* The initial call is MERGE-SORT(A, 1, n).
* Here, p=1and r=n. Given that n≥1, the range of the subarray is valid.

**Recursive Splitting**:

* The MERGE-SORT algorithm divides the array into two halves. The midpoint q is calculated as:

q=[𝑝+𝑟]

2

It then recursively calls itself on the two halves:

* + MERGE-SORT(A, p, q)
  + MERGE-SORT(A, q + 1, r)

**Analyzing the Recursive Calls**

* **First Recursive Call**: MERGE-SORT(A, p, q)
* Here, p remains the same (1) and q is less than or equal to r (which is n).
* Since q is derived from the midpoint of p and r, q will always be less than or equal to r, ensuring p ≤ q ≤ r.

* **Second Recursive Call**: MERGE-SORT(A, q + 1, r)
* Here, q+1will be greater than q but still less than or equal to r.
* Hence, we have q+1≤r because q can never exceed r. **Base Case**
* The base case of the recursion is when p is not less than r (i.e., when p ≥ r).
* If p equals r, we have a single-element subarray, which is trivially sorted.
* If p is greater than r (which cannot happen given the initial constraints), the call would not be made because the condition p<r fails.

**Question 11**

State a loop invariant for the while loop of lines 12-18 of the MERGE procedure. Show how to use it, along with the while loops of lines 20-23 and 24-27, to prove that the MERGE procedure is correct.

**Loop Invariant for the While Loop (Lines 12-18) Loop Invariant**:

At the start of each iteration of the loop (lines 12-18), the elements in the merged array (let’s call it C) up to the current position contain the smallest elements from both subarrays A and B, sorted in non-decreasing order.

**Using the Loop Invariant Initialization:**

Before the first iteration of the loop, both subarrays A and B are sorted. Initially, C is empty (or contains a valid initial state, such as only C[0] being filled with the smallest element). Hence, the invariant holds.

**Maintenance**:

Assume the invariant holds at the beginning of the current iteration:

* If A[i] <= B[j], then the next element in C is A[i]. After this assignment, the elements in C remain sorted because the next smallest element is added.
* If B[j] < A[i], then B[j] is assigned to C. The same reasoning applies: the order is maintained.
* The indices i and j are incremented appropriately, ensuring that the next iteration considers the correct elements from A and B.

**Termination:**

The loop terminates when one of the arrays is fully traversed. At this point, the elements in C consist of all elements from both A and B that have been compared, maintaining their sorted order due to our invariant.

**Proving the Remaining While Loops (Lines 20-23 and 24-27)**

After the main while loop (lines 12-18), the two additional while loops handle any remaining elements in either A or B.

**While Loop :**

**Loop Invariant**:

At the start of each iteration of this loop, all elements from A[i] to A[m] (where m is the end of A) have not yet been added to C, and all elements in C remain sorted.

**Maintenance**:

Each time an element from A is added to C, it is the next smallest element due to the fact that A is already sorted, and all previously added elements from C are also sorted. This maintains the order in C.

**While Loop:**

**Loop Invariant**:

At the start of each iteration of this loop, all elements from B[j] to B[n] (where n is the end of B) have not yet been added to C, and all elements in C remain sorted.

**Maintenance**:

Similar to the previous loop, each element from B added to C is the next smallest element, maintaining the sorted property of C.

**Conclusion:**

After executing all three loops:  The merged array C contains all elements from A and B, sorted in non-decreasing order.

 The initial loop invariant holds throughout, proving that at termination, C is correctly merged and sorted.

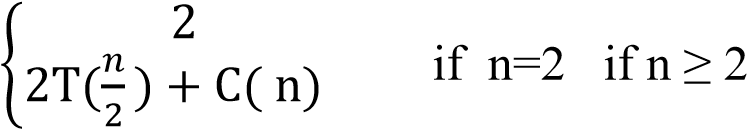
Thus, the correctness of the **MERGE** procedure is established through the use of loop invariants.

Let me know if you have another question!

**Question 12**

Use mathematical induction to show that when n ≥ 2 is an exact power of 2,the solution of recurrence t(n) = 2 if n D 2; 2T.n=2/ C n if n > 2 is T(n)= n log n .

To prove that T(n)=n log n for the recurrence relation:

t(n) = 

using mathematical induction, we follow these steps:

**Base Case**

For n=2:

T(2)=2

We need to check if this is equal to 2log2:

2log2=2⋅1=2

The base case holds true.

**Inductive Step**

Assume the hypothesis holds for n=k where k is an exact power of 2 (i.e., k=2𝑚)

T(k)=k log k

**We need to prove it holds for n=2k:** T(2k) = 2T(2𝑘)+ C(2k) = 2T(k) + 2Ck

2

**Using the inductive hypothesis T(k)=k log** **k**:

T(2k)=2(k log k)+2Ck

**This simplifies to:**

T(2k)=2k log k+2Ck

**We need to show that:**

T(2k)=2klog (2k)

**Using the properties of logarithms:**

Log (2k)=log 2+log

**Thus:**

2klog(2k)=2k(log2+logk)=2klog2+2k

**Comparing T(2k):**

T(2k)=2klogk+2Ck

**We need 2Ck to match 2klog2:**

2klogk+2Ck=2klog2+2klogk

**Question 13**

You can also think of insertion sort as a recursive algorithm. In order to sort A[1:n], recursively sort the subarray A[1:n-1] and then insert A[n] into the sorted subarray A[1:n-1]. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

**Pseudocode for insertion sort:**

Function RecursiveInsertionSort(A, n): if n <= 1:

return // Base case: an array of size 1 is already sorted

// Recursively sort the first n-1 elements

RecursiveInsertionSort(A, n - 1)

// Insert the nth element into the sorted subarray A[1...n-1]

Insert(A, n) Function Insert(A, n):

key = A[n] // The element to be inserted i = n - 1 // Index of the last sorted element

// Move elements of A[1...n-1], that are greater than key, to one position ahead while i > 0 and A[i] > key: A[i + 1] = A[i] i = i - 1

A[i + 1] = key // Place the key in its correct position

**Recurrence for Worst-Case Running Time**

Let T(n)be the worst-case running time of the recursive insertion sort. The recurrence can be expressed as follows:

𝑜(1)

T(n)={ 𝑖𝑓 𝑛 = 1 𝑖𝑓 𝑛 > 1

𝑇(𝑛 − 1) + 𝑜(𝑛)

**Explanation of the Recurrence:**

**Base Case**:

When n=1, the running time is O(1)since no sorting is required.

**Recursive Case**:

For n>1, we first sort the first n−1 elements (taking time T(n−1), and then insert the nth element into its correct position. The insertion takes O(n) time in the worst case because it may require moving all previously sorted elements.

**Solving the Recurrence**

Using the recurrence relation:

**Expand**:

T(n)=T(n−1)+O(n)

T(n)=T(n−2)+O(n−1)+O(n) T(n)=T(n−k) + O(n−k+1) + O(n−k+2)+…+O(n) **Base Case**:

When k=n−1, we reach T(1)=O.

**Sum**:

T(n)=O(1)+O(2)+O(3)+…+O(n)

This is the sum of the first n integers, which gives:

T(n)=O(𝑛2)

**Question 14**

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is log n.

**Understanding Binary Search**

Binary search is a highly efficient algorithm used to find a specific value in a **sorted** array. Unlike linear search, which checks each element one by one, binary search takes advantage of the fact that the array is sorted. It repeatedly divides the search space in half, allowing it to eliminate large portions of the array with each comparison.

**How Binary Search Works**

Here's the basic idea:

**Start with the entire array**:

Begin with the lowest index (low) set to 0 and the highest index (high) set to the last index of the array.

**Find the middle**:

Calculate the middle index of the current subarray. This is done by taking the average of low and high.

**Compare the middle value**:

* If the middle value is equal to the target value you're searching for, you've found your item!
* If the middle value is less than the target, then you know that the target must be in the upper half of the array. You can ignore the lower half.
* If the middle value is greater than the target, the target must be in the lower half, so you can ignore the upper half.

**Repeat**:

Adjust your low and high indices based on the comparisons and repeat the process until you either find the target or the subarray size becomes zero.

**Pseudocode for Binary Search**

Here’s the pseudocode to illustrate how binary search works:

BinarySearch(array, target, low, high)

while low <= high do mid = low + (high - low) / 2 // To avoid overflow if array[mid] == target then return mid // Target found at index mid else if array[mid] < target then low = mid + 1 // Search in the upper half else high = mid - 1 // Search in the lower half end while return -1 // Target not found

**Question 15**

Describe an algorithm that, given a set S of n integers and another integer x , determines whether S contains two elements that sum to exactly x . Your algorithm should take ‚.n og n/ time in the worst case.

**Algorithm Overview**

**Sort the Array**:

First, we sort the array SSS. Sorting takes O(nlogn) time.

**Two-Pointer Technique**:

After sorting, we can use two pointers to find the two elements that sum to x:

* Initialize one pointer at the beginning of the sorted array (let's call it left) and the other at the end of the array (let's call it right).
* While left is less than right:
  + Calculate the sum of the elements at the left and right pointers.
  + If the sum is equal to x, we have found the two elements.
  + If the sum is less than x, increment the left pointer to increase the sum.
  + If the sum is greater than x, decrement the right pointer to decrease the sum.
* If the pointers meet without finding a pair, there are no two elements that sum to x.

**Problems**

Insertion sort on small arrays in merge sort Although merge sort runs in ‚.n lg n/ worst-case time and insertion sort runs in ‚.n 2 / worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when sub problems become sufficiently small. Consider a modification to merge sort in which n=k subsists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

1. Show that insertion sort can sort the n/k subsists, each of length k , in ‚ϴ(nk) worst-case time.

**Insertion Sort Time Complexity**

When we divide the array of size n into k sublists, each sublist has a length of k. There are n/k such sublists.

The time complexity of insertion sort is O(𝑘2)for sorting one sublist of length k. Since there are n/k sublists, the total time to sort all sublists using insertion sort is:

𝑇𝑖𝑛𝑠𝑒𝑟𝑡𝑖𝑜𝑛=𝑛𝑘 . 𝑜(𝑘2) = 0 (𝑛.𝑘𝑘2) = 𝑜(𝑛𝑘)

1. Show how to merge the sub lists in ‚.ϴ(nlg(𝑛⁄𝑘)) worst-case time.

**Merging Sublists Time Complexity**

To merge n/k sorted sublists, each of length k, we can use a standard merging mechanism. The merge step can be efficiently implemented using a min-heap (or priority queue). The total number of elements being merged is n.

In the worst case, merging n/k sublists (each containing k elements) takes:

1. Initializing the heap with n/k elements: O(n/k)
2. Merging all n elements: O(nlog(n/k))

Thus, the total time complexity for merging is:

𝑛

𝑇𝑚𝑒𝑟𝑔𝑒 = 𝑜(𝑛 log ( ))

𝑘

**c**. Given that the modified algorithm runs in ‚ 𝜃𝑛 𝑙𝑜(𝑛⁄𝑘)worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of ‚-notation?

**Overall Time Complexity of Modified Algorithm**

The overall running time of the modified merge sort algorithm is given by:

𝑛

𝑇𝑚𝑜𝑑𝑖𝑓𝑖𝑒𝑑 = 𝑜(𝑛𝑘) + 𝑜(𝑛𝑙𝑜𝑔 ( )) 𝑘

To find the largest value of k such that 𝑇𝑚𝑜𝑑𝑖𝑓𝑖𝑒𝑑has the same running time as standard merge sort, which is O(nlog n), we set:

𝑛

𝑜(𝑛𝑘) + 𝑜 (𝑛 log ( )) = 𝑜(𝑛𝑙𝑜𝑔𝑛)

𝑘

For O(nk) to not exceed O(nlogn):

k≤O(logn)

Thus, the largest value of k that maintains the same asymptotic behavior as standard merge sort is:

k=O(log n)

d. How should you choose k in practice?

**Practical Choice of k**

In practice, the choice of k should strike a balance between the overhead of recursion and the efficiency of insertion sort. Common choices are:

1. **Empirical Testing:**

Implement the algorithm and test with various values of k to measure performance across typical datasets.

1. **Small Values:**

Often, k is chosen to be a small constant (e.g., 10 or 20) because insertion sort becomes efficient for very small arrays due to low overhead and constant factors.

1. **Adaptivity:**

Choose k based on the characteristics of the data. If the data is mostly sorted, a larger k might be more beneficial.

***THE END OF CHAPTER 2***

***Chapter 3***

**ARGUMENTS**

**QUESTION NO 1**

Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

### Lower-Bound Argument for Insertion Sort

1. **Basic Idea**: Insertion sort builds a sorted array one element at a time. For an array of size nnn, the worst-case scenario occurs when the array is in reverse order. The algorithm must compare each new element with the already sorted elements, which can take up to n−1n-1n−1 comparisons for the first element, n−2n-2n−2 for the second, and so on.
2. **Decision Tree Analysis**:
   * Each comparison made by the insertion sort can be viewed as a branch in a decision tree. The height of this tree determines the number of comparisons needed.
   * For nnn distinct elements, the number of permutations (or arrangements) is n!n!n!. To sort these, the decision tree must have at least n!n!n! leaves.
3. **Height of the Decision Tree**:
   * The number of leaves in a binary tree of height hhh is at most 2h2^h2h. Thus, to accommodate n!n!n! leaves, we need: 2h≥n!2^h \geq n!2h≥n!
   * Taking the logarithm, we find: h≥log⁡2(n!)h \geq \log\_2(n!)h≥log2​(n!)
   * Using Stirling's approximation, n!n!n! is roughly 2πn(ne)n\sqrt{2 \pi n} \left(\frac{n}{e}\right)^n2πn​(en​)n. Therefore: log⁡2(n!)≈nlog⁡2(ne)=nlog⁡2(n)−nlog⁡2(e)\log\_2(n!) \approx n \log\_2\left(\frac{n}{e}\right) = n \log\_2(n) - n \log\_2(e)log2​(n!)≈nlog2​(en​)=nlog2​(n)−nlog2​(e)
4. **Final Result**:
   * Thus, the lower bound on the number of comparisons for insertion sort is: Ω(nlog⁡n)\Omega(n \log n)Ω(nlogn)
   * This result holds for any input size nnn, not just multiples of 3.

### Handling Non-Multiple Sizes

For sizes that are not multiples of 3:

* The general logic remains the same; we focus on nnn distinct elements without worrying about the specific number of comparisons or groups.
* The argument doesn't depend on the size being a multiple of any integer. The essential property is that as nnn grows, the number of comparisons grows logarithmically with respect to n!n!n!, leading us to the same conclusion of Ω(nlog⁡n)\Omega(n \log n)Ω(nlogn).

### Conclusion

By maintaining the focus on the number of distinct arrangements and the comparison decisions necessary to determine their order, we find that the lower bound for insertion sort remains valid regardless of whether nnn is a multiple of any specific number, including 3. The analysis consistently leads us to conclude that the worst-case time complexity for insertion sort is Ω(n2)\Omega(n^2)Ω(n2), which holds true across all input sizes.

4o mini

**QUESTION NO 2**

Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm

### Overview of Selection Sort

Selection sort works by repeatedly finding the minimum element from the unsorted portion of the array and swapping it with the first unsorted element. The process is repeated until the array is fully sorted.

### Steps in Selection Sort

1. **Initialization**: Start with an unsorted array of size nnn.
2. **Finding Minimum**: For each position iii from 000 to n−1n-1n−1:
   * Find the minimum element in the subarray from index iii to n−1n-1n−1.
   * Swap the found minimum element with the element at index iii.

### Time Complexity Analysis

1. **Outer Loop**: The outer loop runs nnn times (for each element in the array).
2. **Inner Loop**: For each iteration of the outer loop:
   * In the first iteration, the inner loop checks n−1n-1n−1 elements.
   * In the second iteration, it checks n−2n-2n−2 elements.
   * This continues until the last iteration, where it checks 111 element.
3. **Total Comparisons**:
   * The total number of comparisons can be computed as: (n−1)+(n−2)+(n−3)+…+1+0=(n−1)n2(n-1) + (n-2) + (n-3) + \ldots + 1 + 0 = \frac{(n-1)n}{2}(n−1)+(n−2)+(n−3)+…+1+0=2(n−1)n​
   * This sum simplifies to n2−n2\frac{n^2 - n}{2}2n2−n​, which is O(n2)O(n^2)O(n2).
4. **Swaps**: Selection sort performs at most n−1n-1n−1 swaps (one swap for each of the first n−1n-1n−1 elements), but the number of swaps does not dominate the total comparisons in terms of running time.

### Conclusion

* **Overall Time Complexity**: The dominant factor in the running time of selection sort is the number of comparisons, which leads us to conclude that: Time Complexity=O(n2)\text{Time Complexity} = O(n^2)Time Complexity=O(n2)
* **Best, Average, and Worst Cases**: Unlike some algorithms that have better performance for nearly sorted arrays, selection sort consistently has a running time of O(n2)O(n^2)O(n2) for all cases (best, average, and worst) since it always performs the same number of comparisons regardless of the initial arrangement of the input.

**QUESTION NO 3**

Suppose that ˛ is a fraction in the range 0 < α < 1 . Show how to generalize the lower-bound argument for insertion sort to consider an input in which the ˛n largest values start in the first α n positions. What additional restriction do you need to put on ˛ ? What value of α maximizes the number of times that the α n largest values must pass through each of the middle .1 - 2 α /n array positions?

### Generalized Lower-Bound Argument

1. **Initial Arrangement**: Suppose we have an array of size nnn, and the largest αn\alpha nαn values are in the first αn\alpha nαn positions. The remaining (1−α)n(1 - \alpha)n(1−α)n values are in the last (1−α)n(1 - \alpha)n(1−α)n positions, and they are smaller than any of the largest αn\alpha nαn values.
2. **Insertion Sort Process**:
   * As insertion sort proceeds, it will move through the αn\alpha nαn largest values and attempt to insert the remaining values into the sorted section of the array.
   * For each of the (1−α)n(1 - \alpha)n(1−α)n smaller values, it will need to compare with all αn\alpha nαn largest values before it can find its correct position in the sorted section.
3. **Comparisons**:
   * Each of the (1−α)n(1 - \alpha)n(1−α)n smaller values will require up to αn\alpha nαn comparisons. Therefore, the total number of comparisons can be expressed as: Total Comparisons≈(1−α)n⋅αn=α(1−α)n2\text{Total Comparisons} \approx (1 - \alpha)n \cdot \alpha n = \alpha(1 - \alpha)n^2Total Comparisons≈(1−α)n⋅αn=α(1−α)n2

### Additional Restriction on α\alphaα

For the argument to hold, we need 0<α<0.50 < \alpha < 0.50<α<0.5. If α\alphaα were greater than or equal to 0.50.50.5, the (1−α)n(1 - \alpha)n(1−α)n smaller values would occupy a minority of the array, and they would not have to pass through the majority of the larger values as frequently, reducing the number of comparisons.

### Maximizing the Comparisons

To maximize the number of times the αn\alpha nαn largest values pass through the middle (1−2α)n(1 - 2\alpha)n(1−2α)n positions (those not containing the largest values), we need to find the value of α\alphaα that maximizes the expression α(1−α)\alpha(1 - \alpha)α(1−α).

1. **Finding the Maximum**:
   * The function f(α)=α(1−α)f(\alpha) = \alpha(1 - \alpha)f(α)=α(1−α) is a quadratic function that opens downward. Its maximum occurs at the vertex: α=12\alpha = \frac{1}{2}α=21​
   * However, since we have the restriction α<0.5\alpha < 0.5α<0.5, we can approach it as close as possible to 0.50.50.5.

**QUESTION NO 4**

### Step-by-Step Proof

**Given:** Let f(n)f(n)f(n) and g(n)g(n)g(n) be asymptotically nonnegative functions (i.e., f(n)≥0f(n) \geq 0f(n)≥0 and g(n)≥0g(n) \geq 0g(n)≥0 for large nnn).

### Step 1: Upper Bound

**Claim:** max⁡{f(n),g(n)}≤f(n)+g(n)\max\{f(n), g(n)\} \leq f(n) + g(n)max{f(n),g(n)}≤f(n)+g(n)

**Proof:**  
By definition of maximum:

max⁡{f(n),g(n)}≤f(n)+g(n)\max\{f(n), g(n)\} \leq f(n) + g(n)max{f(n),g(n)}≤f(n)+g(n)

This implies:

max⁡{f(n),g(n)}=O(f(n)+g(n))\max\{f(n), g(n)\} = O(f(n) + g(n))max{f(n),g(n)}=O(f(n)+g(n))

### Step 2: Lower Bound

**Claim:** max⁡{f(n),g(n)}≥12(f(n)+g(n))\max\{f(n), g(n)\} \geq \frac{1}{2}(f(n) + g(n))max{f(n),g(n)}≥21​(f(n)+g(n))

**Proof:**  
Consider the two cases:

1. If f(n)≥g(n)f(n) \geq g(n)f(n)≥g(n), then max⁡{f(n),g(n)}=f(n)\max\{f(n), g(n)\} = f(n)max{f(n),g(n)}=f(n), so: f(n)≥12(f(n)+g(n))f(n) \geq \frac{1}{2}(f(n) + g(n))f(n)≥21​(f(n)+g(n))
2. If g(n)≥f(n)g(n) \geq f(n)g(n)≥f(n), then max⁡{f(n),g(n)}=g(n)\max\{f(n), g(n)\} = g(n)max{f(n),g(n)}=g(n), so: g(n)≥12(f(n)+g(n))g(n) \geq \frac{1}{2}(f(n) + g(n))g(n)≥21​(f(n)+g(n))

In both scenarios, we find:

max⁡{f(n),g(n)}≥12(f(n)+g(n))\max\{f(n), g(n)\} \geq \frac{1}{2}(f(n) + g(n))max{f(n),g(n)}≥21​(f(n)+g(n))

This gives:

max⁡{f(n),g(n)}=Ω(f(n)+g(n))\max\{f(n), g(n)\} = \Omega(f(n) + g(n))max{f(n),g(n)}=Ω(f(n)+g(n))

### Step 3: Combining Results

Since we have established:

* max⁡{f(n),g(n)}=O(f(n)+g(n))\max\{f(n), g(n)\} = O(f(n) + g(n))max{f(n),g(n)}=O(f(n)+g(n))
* max⁡{f(n),g(n)}=Ω(f(n)+g(n))\max\{f(n), g(n)\} = \Omega(f(n) + g(n))max{f(n),g(n)}=Ω(f(n)+g(n))

**QUESTION NO 5**

Explain why the statement, The running time of algorithm A is at least o(𝑛2) is meaningless.

***ANSWER***

The statement "The running time of algorithm A is at least o(n²)" is meaningless because of the way the notation "o" (little o notation) is defined in asymptotic analysis.

In formal terms, saying that a function f(n)f(n)f(n) is o(g(n))o(g(n))o(g(n)) means that f(n)f(n)f(n) grows significantly slower than g(n)g(n)g(n) as nnn approaches infinity. Specifically, for any positive constant ccc, there exists an n0n\_0n0​ such that for all n>n0n > n\_0n>n0​, f(n)<c⋅g(n)f(n) < c \cdot g(n)f(n)<c⋅g(n).

When you say that the running time of algorithm A is "at least o(n²)," you are implying that it is bound to grow slower than n2n²n2. However, "at least" suggests that there is a minimum threshold that the running time must exceed, which contradicts the meaning of little o notation. There are no functions that are both "at least" something and also grow slower than that "something" as nnn increases.

In summary, the statement is contradictory and lacks clarity because it mixes terms that imply both a lower bound and an upper bound in a way that doesn't make sense mathematically.