



# **Abbottabad University Of Science And Technology**

## **DEPARTMENT OF COMPUTER SCIENCE**

### **Assignment No 1**

**Name : Mahroosha Jadoon**

**Roll No : 14655**

**Discipline : Computer Science**

**Semester : BSCS 3<sup>rd</sup> A**

**Subject : DSA**

**Submitted To:**

**Sir Jamal Abdul Ahad**

## Question No 1:

Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

### Answer:

Sorting:

A good real-world example of sorting is organizing files by their modification date on a computer. Imagine you're working on a project where you need to quickly access the latest documents. By sorting the files in ascending or descending order of their last modified date, you can prioritize which files to review or edit first. Sorting helps streamline workflows by bringing the most relevant or recent files to the top.

Finding the Shortest Distance:

A classic real-world example involving the shortest distance is using GPS navigation to find the quickest route between two locations. For instance, if you're trying to get from your home to a restaurant in a busy city, the GPS system will calculate the shortest or fastest route, often accounting for real-time traffic, roadblocks, and distances. This calculation involves finding the minimum path between two points within a network of roads, optimizing travel time and convenience.

## Question No 2:

Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

### Answer:

In a real-world setting, several measures of efficiency beyond speed are important to consider:

**Space Complexity:** This refers to the amount of memory an algorithm uses. Efficient memory usage is crucial, especially for large datasets or systems with limited memory resources<sup>1</sup>.

**Scalability:** This measures how well an algorithm or system can handle increasing amounts of work or data. An efficient algorithm should maintain performance as the size of the input grows<sup>1</sup>.

**Energy Efficiency:** Particularly relevant for mobile and embedded systems, this measures the amount of computational power required. Reducing energy consumption can lead to longer battery life and lower operational costs<sup>1</sup>.

**Parallelizability:** This assesses how well an algorithm can be executed in parallel, taking advantage of multi-core processors to improve performance<sup>1</sup>.

**Reliability and Robustness:** These measure how consistently an algorithm performs under different conditions and how well it handles errors or unexpected inputs.

**Maintainability:** This refers to how easily an algorithm or system can be understood, modified, and extended by developers. High maintainability can reduce long-term costs and improve adaptability.

**Cost Efficiency:** This includes both the initial development costs and the ongoing operational costs. Efficient algorithms can reduce hardware requirements and operational expenses.

### **Question No 3:**

Select a data structure that you have seen, and discuss its strengths and limitations.

#### **Answer:**

Let's discuss the hash table, a widely used data structure.

#### **Strengths:**

**Fast Access:** Hash tables provide average-case constant time complexity,

$O(1)$

, for insertions, deletions, and lookups, making them extremely efficient for these operations.

**Efficient Use of Space:** When well-designed, hash tables can be very space-efficient, storing only the necessary data without much overhead.

**Flexibility:** They can handle a wide variety of data types and are used in many applications, from databases to caching mechanisms.

#### **Limitations:**

**Collisions:** When two keys hash to the same index, a collision occurs. Handling collisions (e.g., through chaining or open addressing) can complicate the implementation and affect performance.

**Memory Usage:** While generally efficient, hash tables can require more memory than other data structures due to the need for a larger array to minimize collisions.

**Non-Ordered Data:** Hash tables do not maintain any order of elements, which can be a limitation if you need to traverse elements in a specific sequence.

**Complexity of Hash Functions:** Designing an efficient hash function that minimizes collisions and distributes keys uniformly can be challenging.

## Question No 4:

How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

### Answer:

#### Similarities:

**Graph-Based:** Both problems are typically represented using graphs, where nodes represent locations (e.g., cities) and edges represent paths or routes between these locations<sup>1</sup>.

**Optimization Goals:** Both aim to find an optimal path. The shortest-path problem seeks the shortest route between two specific nodes, while the TSP seeks the shortest possible route that visits each node exactly once and returns to the starting point<sup>1</sup>.

**Distance Minimization:** Both problems involve minimizing the total distance or cost associated with traveling between nodes.

#### Differences:

**Objective:**

**Shortest-Path Problem:** Focuses on finding the shortest path between two specific nodes in a graph.

**Traveling Salesperson Problem:** Focuses on finding the shortest possible route that visits each node exactly once and returns to the starting node.

**Complexity:**

**Shortest-Path Problem:** Generally, has polynomial-time solutions (e.g., Dijkstra's algorithm for graphs with non-negative weights).

**Traveling Salesperson Problem:** Is NP-hard, meaning there is no known polynomial-time solution for the general case<sup>1</sup>.

**Path Constraints:**

**Shortest-Path Problem:** Only requires a path between two nodes, without needing to visit all nodes.

**Traveling Salesperson Problem:** Requires visiting all nodes exactly once before returning to the starting node.

### **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

#### **Answer:**

#### **Problem Requiring the Best Solution**

A real-world scenario that demands the absolute best solution is in medical treatment planning for critical conditions, such as cancer radiation therapy. In this context, treatment needs to be precisely calibrated to target cancer cells effectively while minimizing damage to surrounding healthy tissue. Only the optimal solution—achieving the best radiation dose and angle—ensures patient safety and the highest chance of effective treatment.

#### **Problem Where an Approximate Solution is Good Enough**

A common example where an approximate solution is sufficient is in online product recommendation systems. When shopping on a platform like Amazon, users are recommended products that the algorithm believes are relevant to their interests based on past behavior and preferences. An exact match to their ideal product isn't necessary, as approximate recommendations are usually sufficient to catch their interest. Slightly less optimal recommendations are still valuable for user engagement and conversion.

### **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.

#### **Answer:**

A real-world example of this kind of problem is processing transactions in a bank.

When performing an end-of-day balance reconciliation, the bank has the entire day's transactions already available. The system can process all transactions at once to determine the final balance and any discrepancies.

However, during real-time fraud detection, transactions arrive continuously as they occur, and the system must process each transaction as it comes in to identify potentially fraudulent activity immediately. Here, the input isn't fully available in advance—it arrives over time and needs to be processed in real time to catch issues as they happen.

## Question No 7:

example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

### Answer:

A great example of an application that requires algorithmic content at the application level is **Google Maps** for route planning and navigation. The algorithms used in this application are crucial for its core functionalities and user experience.

### Algorithms Involved and Their Functions:

*Shortest Path Algorithm (e.g., Dijkstra's or A Algorithm):*

When a user enters a destination, Google Maps calculates the shortest path from their current location to the target point. The shortest path algorithm considers the network of roads and pathways to find the route that minimizes travel time or distance. This is fundamental for helping users reach their destination quickly and efficiently.

#### Traffic Prediction and Real-Time Update Algorithms:

To provide accurate ETAs and reroute suggestions, Google Maps uses algorithms that analyze real-time traffic data, user movement patterns, and historical data. Machine learning models may predict where traffic congestion is likely, while other algorithms adjust routes dynamically based on new data.

#### Geolocation and Mapping Algorithms:

Google Maps relies on geolocation algorithms to pinpoint the user's position accurately using GPS, Wi-Fi, and cell towers. Mapping algorithms also create and display detailed, scalable maps that help users orient themselves and navigate more easily.

#### Recommendation Algorithms (for Points of Interest):

In addition to navigation, Google Maps recommends restaurants, hotels, and attractions based on user preferences and proximity. These algorithms enhance the application's utility by helping users find nearby places of interest.

The integration of these algorithms at the application level ensures that Google Maps can provide users with timely, relevant, and efficient navigation and location-based services.

## Question No 8:

Suppose that for inputs of size  $n$  on a particular computer, insertion sort runs in  $8n^2$  steps and merge sort runs in  $64n \lg n$  steps. For which values of  $n$  does insertion sort beat merge sort?

### Answer:

To determine the values of  $n$  for which insertion sort beats merge sort, we need to compare the number of steps each algorithm takes.

**Given:**

- Insertion sort takes  $(8n^2)$  steps.
- Merge sort takes  $(64n \lg n)$  steps.

We want to find when  $(8n^2 < 64n \lg n)$ .

Dividing both sides by  $(8n)$  (assuming  $(n > 0)$ ):

$$\begin{aligned} & \left[ \right. \\ & n < 8 \lg n \\ & \left. \right] \end{aligned}$$

Now we need to determine the values of  $(n)$  for which this inequality holds. This can be solved by trial and error or graphically, as the inequality involves  $(n)$  and  $(\lg n)$ , which grow at different rates.

Let's compute values for  $(n)$  and check when  $(n)$  first becomes greater than  $(8 \lg n)$ .

It turns out that even at  $(n = 1)$ , the inequality  $(n < 8 \lg n)$  does not hold because  $(\lg 1 = 0)$ . So, technically, insertion sort beats merge sort for very small values of  $(n)$ . However, since  $(\lg n)$  grows more slowly than  $(n)$ , insertion sort will typically only perform better than merge sort for extremely small inputs in practical scenarios. For any  $(n \geq 2)$ , merge sort is more efficient in terms of steps.

**Question No 9:**

What is the smallest value of  $n$  such that an algorithm whose running time is  $100n^2$  runs faster than an algorithm whose running time is  $2^n$  on the same machine?

**Answer:**

To find the smallest value of  $(n)$  such that an algorithm with a running time of  $(100n^2)$  is faster than an algorithm with a running time of  $(2^n)$ , we need to solve for  $(n)$  where:

$$\begin{aligned} & \left[ \right. \\ & 100n^2 < 2^n \\ & \left. \right] \end{aligned}$$

Since  $(2^n)$  grows much faster than  $(n^2)$ , we can start testing values of  $(n)$  to find when this inequality holds. Let's compute it for small values and increment until we identify the threshold where  $(100n^2)$  is less than  $(2^n)$ .

The smallest value of  $(n)$  such that an algorithm with a running time of  $(100n^2)$  runs faster than an algorithm with a running time of  $(2^n)$  is  $(n = 15)$ .

### Question No 10:

illustrate the operation of INSERTION-SORT on an array initially containing the sequence 31; 41; 59; 26; 41; 58.

### Answer:

Here are the steps of the Insertion Sort algorithm on the array  $([31, 41, 59, 26, 41, 58])$ :

1. Initial Array:  $[31, 41, 59, 26, 41, 58]$
2. Step 1:  $[31, 41, 59, 26, 41, 58]$  (First two elements are already sorted)
3. Step 2:  $[31, 41, 59, 26, 41, 58]$  (First three elements are in place)
4. Step 3:  $[26, 31, 41, 59, 41, 58]$  (Inserted 26 in the correct position)
5. Step 4:  $[26, 31, 41, 41, 59, 58]$  (Inserted the second 41 in its position)
6. Step 5:  $[26, 31, 41, 41, 58, 59]$  (Inserted 58, final sorted array)

### Question No 11:

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 SUMARRAY procedure returns the sum of the numbers in  $A[1..n]$ .

### Answer:

#### Loop Invariant

A loop invariant is a condition that holds true before and after each iteration of the loop. For the SUM-ARRAY procedure, the loop invariant can be stated as:

**Loop Invariant:** At the start of each iteration of the loop, the variable `total` contains the sum of the elements in the subarray  $A[0:i]$ .

#### Proof Using Initialization, Maintenance, and Termination



## Initialization:

Before the loop starts, `total` is initialized to 0.

The subarray `A[0:0]` is empty, and the sum of an empty subarray is 0.

Therefore, the loop invariant holds true initially: `total == sum(A[0:0])`.

## Maintenance:

Assume the loop invariant holds at the start of the  $i$ -th iteration: `total == sum(A[0:i])`.

During the  $i$ -th iteration, `A[i]` is added to `total`.

After this addition, `total` becomes `total + A[i]`, which is `sum(A[0:i]) + A[i]`, or `sum(A[0:i+1])`.

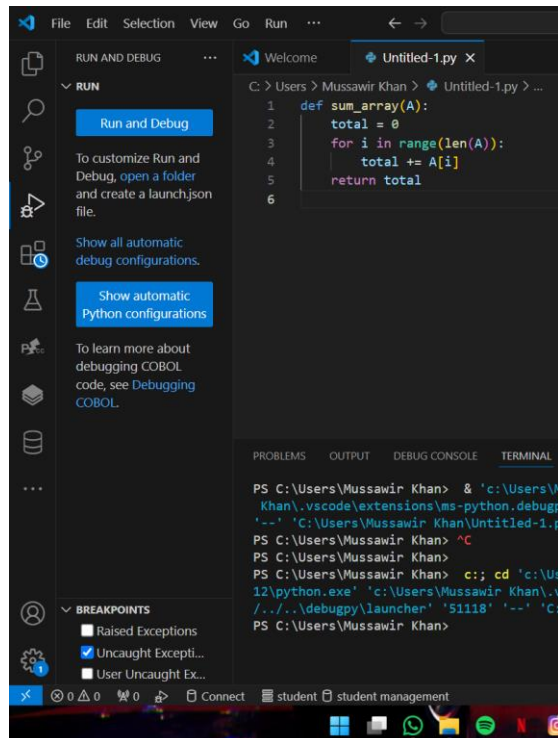
Thus, the loop invariant holds after the  $i$ -th iteration: `total == sum(A[0:i+1])`.

## Termination:

The loop terminates when  $i$  reaches `len(A)`.

At this point, `i == len(A)`, and the loop invariant tells us that `total == sum(A[0:len(A)])`.

Therefore, `total` contains the sum of all elements in the array `A`.



## Question No 12:

Consider the searching problem: Input: A sequence of  $n$  numbers  $a_1; a_2; \dots; a_n$  stored in array  $A[1..n]$  and a value  $x$ . Output: An index  $i$  such that  $x = A[i]$  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 fills the three necessary properties.

## Answer:

### Pseudocode for Linear Search

Here's the pseudocode for a linear search algorithm:

FUNCTION LinearSearch( $A, n, x$ )

    INPUT: Array  $A[1..n]$ , an integer  $n$ , and a value  $x$

    OUTPUT: An index  $i$  such that  $x = A[i]$  or NIL if  $x$  is not found

    FOR  $i$  FROM 1 TO  $n$  DO

        IF  $A[i] = x$  THEN

            RETURN  $i$    // Found  $x$  at index  $i$

        END IF

    END FOR

    RETURN NIL   //  $x$  is not found in the array

END FUNCTION

...

### Loop Invariant

Loop Invariant: At the start of each iteration of the loop (for  $i$  from 1 to  $n$ ), if  $x$  is present in the array  $A[1..n]$ , it will be found in the subarray  $A[1..i-1]$  after this iteration or at index  $i$ .

### Properties of the Loop Invariant

#### Initialization:

Before the first iteration (when  $i = 1$ ), the loop invariant is trivially true because we have not yet examined any elements of the array. If  $x$  is present in the array, it will be found in  $A[1..0]$ , which is empty, and thus does not contradict our invariant.

#### Maintenance:

Assume the invariant holds at the start of the  $i$ -th iteration. If  $A[i]$  equals  $x$ , we return  $i$ , which satisfies our search requirement. If  $A[i]$  does not equal  $x$ , then  $x$  must be in the remaining portion of the array ( $A[i+1..n]$ ) if it exists. The invariant still holds as we move to the next iteration ( $i + 1$ ).

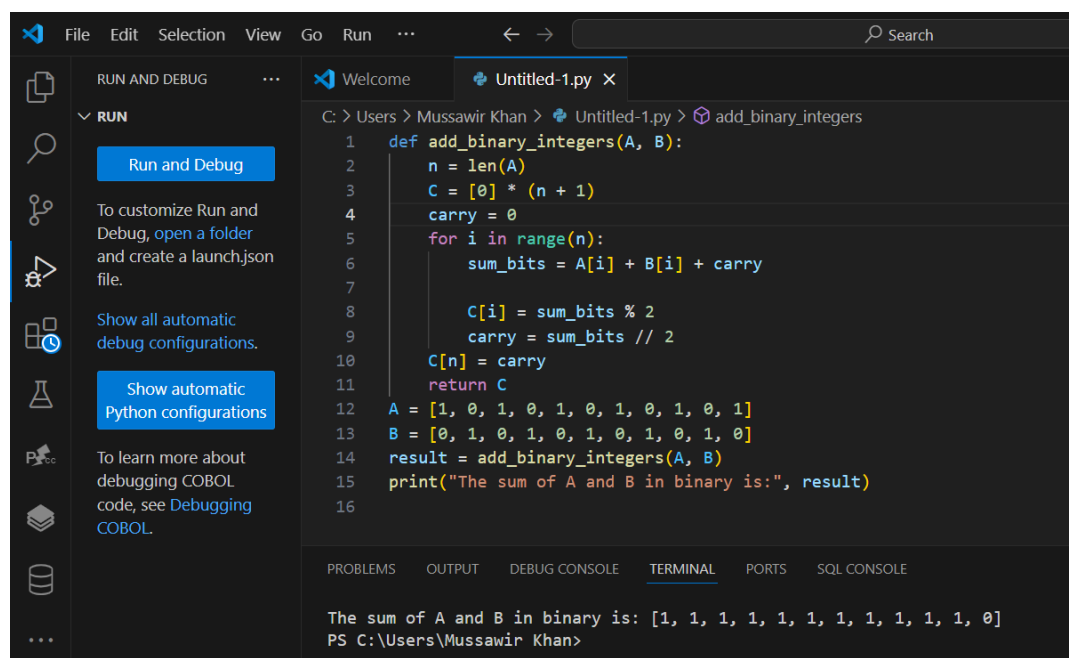
#### Termination:

The loop terminates after checking all elements up to  $n$ . If we find  $x$  during any iteration, we return the index. If the loop finishes without finding  $x$ , we return NIL. In this case, if  $x$  is indeed in the array, the invariant indicates that it should have been found in the examined portion. Thus, the invariant confirms that if  $x$  is not found by the end of the loop, then  $x$  is not in the array.

### Question No 13:

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,  $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$ , and  $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$ . 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  $c = \sum_{i=0}^n C[i] \cdot 2^i$ . 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.

#### Answer:



```
File Edit Selection View Go Run ... Search
RUN AND DEBUG ... Welcome Untitled-1.py X
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Show all automatic debug configurations.
Show automatic Python configurations
To learn more about debugging COBOL code, see Debugging COBOL.
C: > Users > Mussawir Khan > Untitled-1.py > add_binary_integers
1 def add_binary_integers(A, B):
2     n = len(A)
3     C = [0] * (n + 1)
4     carry = 0
5     for i in range(n):
6         sum_bits = A[i] + B[i] + carry
7
8         C[i] = sum_bits % 2
9         carry = sum_bits // 2
10    C[n] = carry
11    return C
12 A = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
13 B = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
14 result = add_binary_integers(A, B)
15 print("The sum of A and B in binary is:", result)
16
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE
The sum of A and B in binary is: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
PS C:\Users\Mussawir Khan>
```

## Question No 14:

express the function  $f(n) = \binom{n}{3} + 100\binom{n}{2} + 100n$  in terms of  $\Theta$ -notation

### Answer:

#### Breakdown of Each Term

##### 1. First Term: $\binom{n}{3}$

The binomial coefficient  $\binom{n}{3}$  is calculated as follows:

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{3!} = \frac{n(n-1)(n-2)}{6}$$

For large  $n$ , this can be approximated as:

$$\binom{n}{3} \sim \frac{n^3}{6}$$

Thus,  $\binom{n}{3} = \Theta(n^3)$ .

##### 2. Second Term: $100\binom{n}{2}$

The binomial coefficient  $\binom{n}{2}$  is calculated as follows:

$$\binom{n}{2} = \frac{n(n-1)}{2!} = \frac{n(n-1)}{2}$$

For large  $n$ , this can be approximated as:

$$\binom{n}{2} \sim \frac{n^2}{2}$$

Therefore,  $100\binom{n}{2}$  can be expressed as:

$$100\binom{n}{2} = 100 \cdot \frac{n(n-1)}{2} \sim 50n^2$$

Thus,  $100\binom{n}{2} = \Theta(n^2)$ .

##### 3. Third Term: $100n$

The term  $100n$  is clearly linear:

$$100n = \Theta(n)$$

#### Combining the Terms

Now we combine these results:

- The first term  $\Theta(n^3)$  dominates the other two terms ( $\Theta(n^2)$  and  $\Theta(n)$ ).
- Therefore, when adding the terms together, the dominant term determines the overall growth rate.

Thus, we can express the function  $f(n)$  in terms of  $\Theta$ -notation as follows:

$$f(n) = \Theta(n^3)$$

#### Conclusion

The function

$$f(n) = \binom{n}{3} + 100\binom{n}{2} + 100n$$

is expressed in terms of  $\Theta$ -notation as:

$$f(n) = \Theta(n^3)$$

## Question No 15:

Consider sorting a numbers stored in array  $A[1:n]$  by first finding the smallest element of  $A[1:n]$  and exchanging it with the element in  $A[1]$ . Then find the smallest element of  $A[2:n]$ , and exchange it with  $A[2]$ . Then find the smallest element of  $A[3:n]$ , and exchange it with  $A[3]$ . Continue in this manner for the first  $n-1$  elements of  $A$ . Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first  $n-1$  elements, rather than for all  $n$  elements? Give the worst-case running time of selection sort in  $\Theta$ -notation. Is the best-case running time any better?

## Answer:

### Selection Sort Algorithm Pseudocode

Selection sort works by finding the smallest element in the unsorted portion of the array and swapping it with the first element in that portion. It then repeats this process for each subsequent position in the array.

```
FUNCTION SelectionSort(A, n)
    INPUT: Array A[1..n] of length n
    OUTPUT: Array A sorted in non-decreasing order

    FOR i FROM 1 TO n - 1 DO
        min_index = i

        // Find the smallest element in the unsorted portion A[i..n]
        FOR j FROM i + 1 TO n DO
            IF A[j] < A[min_index] THEN
                min_index = j
            END IF
        END FOR

        // Swap the smallest element with A[i]
        SWAP A[i] WITH A[min_index]
    END FOR

    RETURN A
END FUNCTION
```

### Loop Invariant

The **loop invariant** for selection sort is as follows:

**Loop Invariant:** At the start of each iteration of the outer loop (for a given  $i$ ), the subarray  $A[1..i-1]$  is sorted, and every element in  $A[1..i-1]$  is smaller than or equal to every element in  $A[i..n]$ .

### Proof of Correctness Using the Loop Invariant:

1. **Initialization:** Before the first iteration of the loop ( $i = 1$ ), the subarray  $A[1..0]$  is empty, which trivially satisfies the invariant.
2. **Maintenance:** At each iteration, the algorithm finds the smallest element in the unsorted subarray  $A[i..n]$  and places it at position  $i$ . After the swap,  $A[1..i]$  becomes sorted, and all elements in  $A[1..i]$  are smaller than or equal to elements in  $A[i+1..n]$ .
3. **Termination:** The loop terminates when  $i = n$ . At this point, the entire array  $A[1..n]$  is sorted because the loop invariant has been maintained for all iterations.

## Why the Algorithm Only Needs to Run for the First $n-1$ Elements

Selection sort only needs to run for the first  $n-1$  elements (i.e.,  $i$  ranges from 1 to  $n-1$ ) because, after placing the smallest  $n-1$  elements in their correct positions, the remaining last element must be the largest. Hence, it will already be in the correct position without any further comparisons or swaps.

## Worst-Case Running Time of Selection Sort in $\Theta$ -Notation

In each iteration of the outer loop (from  $i = 1$  to  $n-1$ ), selection sort scans the unsorted portion of the array to find the minimum element. This requires:

- $n-1$  comparisons for the first iteration,
- $n-2$  comparisons for the second iteration,
- And so on, until the last iteration, which requires 1 comparison.

The total number of comparisons is the sum:

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Thus, the **worst-case running time** of selection sort is  $\Theta(n^2)$ .

## Best-Case Running Time of Selection Sort

The best-case running time of selection sort is also  $\Theta(n^2)$  because the algorithm always performs the same number of comparisons regardless of the input order. Selection sort does not take advantage of an already sorted array, so both the best and worst cases are  $\Theta(n^2)$ .

## Question No 16:

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  $\Theta$ -notation, give the average-case and worst-case running times of linear search. Justify your answers.

### Answer:

#### Average-Case Analysis

If the element we are searching for is equally likely to be any element in the array, then we assume that the target could be at any position from 1 to  $n$  with equal probability.

For an array of  $n$  elements:

The average case involves searching for the target element, which could be at any position with equal likelihood.

On average, if the target is in the array, we would find it about halfway through the array. This means we would typically inspect about  $\frac{n}{2}$  elements before finding the target.

Thus, the average-case number of comparisons is  $\frac{n}{2}$ .

In terms of Big- $\Theta$  notation, we consider only the growth rate as  $n$  becomes large. Since  $\frac{n}{2} = \Theta(\frac{n}{2}) = \Theta(n)$ , we can conclude:

Average-case running time of linear search  $= \Theta(n)$

#### Worst-Case Analysis

In the worst case, the element might be the very last one in the array, or it may not be present in the array at all. In both scenarios, we must check all  $n$  elements before concluding.

Thus, the **worst-case number of comparisons** is  $n$ .

In Big- $\Theta$  notation, this is simply:

Worst-case running time of linear search  $= \Theta(n)$

## Question No 17:

How can you modify any sorting algorithm to have a good best-case running time?

### Answer:

To improve the best-case running time of any sorting algorithm, a common approach is to add a **pre-check** to detect whether the array is already sorted. If the array is sorted, we can simply return it without performing any additional operations, which would make the best-case time  $O(n)$  (linear time) for most sorting algorithms.

Here's how this modification works in general:

**Add a Check for Sorted Order:** Before starting the sorting process, loop through the array to verify if it is already in sorted order. This can be done in  $O(n)$  time by checking if each element is less than or equal to the next element in the sequence.

**Return Early if Sorted:** If the array passes this sorted check, simply return it as sorted without calling the main sorting logic.

**Proceed with Sorting Otherwise:** If the array is not sorted, proceed with the sorting algorithm as usual.

Example :

FUNCTION SortWithBestCaseCheck(A)

    INPUT: Array A[1..n]

    OUTPUT: Sorted array A

Step 1: Check if A is already sorted

    is\_sorted = True

    FOR i FROM 1 TO n - 1 DO

        IF A[i] > A[i + 1] THEN

            is\_sorted = False

        BREAK

    END IF

    END FOR

Step 2: If sorted, return A immediately

    IF is\_sorted THEN

        RETURN A

    END IF



Step 3: Otherwise, proceed with the sorting algorithm

Sort(A) // This represents the original sorting algorithm

RETURN A

END FUNCTION

### Question No 18:

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.

**Answer:**

### Step-by-Step Merge Sort Illustration

Given array:

[3, 41, 52, 26, 38, 57, 9, 49]

1. **Split** into two halves:

Left: [3, 41, 52, 26]

Right: [38, 57, 9, 49]

2. **Split** each half further:

○ **Left:** [3, 41, 52, 26]

▪ Split into: [3, 41] and [52, 26]

○ **Right:** [38, 57, 9, 49]

▪ Split into: [38, 57] and [9, 49]

3. **Split each of the new subarrays** until we have single elements:

[3] [41] [52] [26]      and      [38] [57] [9] [49]

4. **Merge each pair** of single elements, sorting as we go:

○ Merge [3] and [41] into [3, 41]

○ Merge [52] and [26] into [26, 52]

○ Merge [38] and [57] into [38, 57]

○ Merge [9] and [49] into [9, 49]

After merging:

[3, 41] [26, 52]      and      [38, 57] [9, 49]

5. **Merge these sorted pairs:**

○ Merge [3, 41] and [26, 52] to get [3, 26, 41, 52]

○ Merge [38, 57] and [9, 49] to get [9, 38, 49, 57]

Now we have:

[3, 26, 41, 52]      and      [9, 38, 49, 57]

6. **Final Merge** of the two sorted halves:

- Merge [3, 26, 41, 52] and [9, 38, 49, 57] to get the sorted array:

[3, 9, 26, 38, 41, 49, 52, 57]

## Final Sorted Array

The final sorted array after completing all merges is:

[3, 9, 26, 38, 41, 49, 52, 57]

## Question No 19:

The test in line 1 of the MERGE-SORT procedure reads "if  $p \geq r$ " prime rather than "if  $p \neq r$ ". If MERGE-SORT is called with  $p > r$  then the subarray  $A[p \dots r]$  is empty. Argue that as long as the initial call of  $\text{-SORT}(A, 1, n)$  has  $n \geq 1$  the test "if  $p \neq r$ " prime suffices to ensure that no recursive call has  $p > r$ .

### Answer:

In the **MERGE-SORT** algorithm, the purpose of the condition `if  $p < r$`  (or equivalently, `if  $p \neq r$` ) is to check whether the subarray  $A[p \dots r]$  has more than one element before proceeding with further recursive division.

To break down the reasoning, let's examine the recursive process:

**Initial Call of MERGE-SORT:** Suppose we call  $\text{MERGE-SORT}(A, 1, n)$  on an array  $A$  of size  $n$ , where  $n \geq 1$ . This means  $p = 1$  and  $r = n$ , so  $p$  is less than  $r$  as long as  $n > 1$ .

**Recursive Division Process:** During each recursive call, **MERGE-SORT** divides the array into two halves by computing the midpoint  $q = \lfloor (p + r) / 2 \rfloor$ , which splits the array into  $A[p \dots q]$  and  $A[q+1 \dots r]$ . It then recursively calls itself on these two subarrays.

**Base Case:** When  $p == r$ , the subarray  $A[p \dots r]$  contains exactly one element, and **MERGE-SORT** reaches its base case. At this point, there is no further recursive call on  $A[p \dots r]$ , and it returns without performing further division. This is the stopping condition of the recursion.

**Preventing  $p > r$ :** Since each recursive call further narrows the range  $[p \dots r]$  by recalculating the midpoint  $q$ , it either maintains  $p \leq r$  or eventually reduces the range to  $p == r$ . With this approach, **MERGE-SORT** will never reach a state where  $p > r$  as long as the initial call has  $p < r$  (i.e.,  $n \geq 1$ ).

## Conclusion

The test `if p != r` suffices to prevent any recursive calls with `p > r`:

For any call where `p < r`, the recursion proceeds.

For calls with `p == r`, **MERGE-SORT** reaches the base case and does not call itself further.

Thus, **MERGE-SORT** will not encounter `p > r` under any circumstances, given that the initial call

## Question No 20:

State a loop invariant for the while loop of lines 12318 of the MERGE procedure. Show how to use it, along with the while loops of lines 20323 and 24327, to prove that the MERGE procedure is correct.

### Answer:

```
def merge(A, p, q, r):
    # Calculate lengths of the two subarrays
    n1 = q - p + 1
    n2 = r - q

    # Create left and right arrays with extra space for simplicity
    L = [0] * (n1 + 1)
    R = [0] * (n2 + 1)

    # Copy data to temporary arrays L and R
    for i in range(n1):
        L[i] = A[p + i]
    for j in range(n2):
        R[j] = A[q + 1 + j]

    # Add sentinel values at the end of L and R
    L[n1] = float('inf')
    R[n2] = float('inf')

    # Initialize pointers for L, R, and the main array A
    i = j = 0
    for k in range(p, r + 1):
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1

A = [3, 41, 52, 26, 38, 57, 9, 49]
p = 0
q = (len(A) - 1) // 2
r = len(A) - 1
merge(A, p, q, r)
print("Merged array:", A)
```

## Explanation of the Code

- **Input Array Split:** `L` and `R` represent the two halves, with sentinel values (`float('inf')`) added to mark the ends of each array.
- **Merge Process:** The main `for` loop compares elements of `L` and `R`, appending the smaller of the two to `A`.
- **Sentinel Values:** These ensure that as soon as one array is exhausted, the remaining elements from the other array are copied without additional checks, ensuring sorted merging of `A[p..r]`.

## Question No 21:

Use mathematical induction to show that when  $n \geq 2$  is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is  $T(n) = n \lg n$ .

## Answer:

$$T(n) = \begin{cases} 2, & \text{if } n = 2; \\ 2T\left(\frac{n}{2}\right) + n, & \text{if } n > 2, \end{cases}$$

has the solution  $T(n) = n \lg n$  when  $n \geq 2$  and  $n$  is an exact power of 2.

**Step 1: Base Case**

For  $n = 2$ :

$$T(2) = 2.$$

The formula  $T(n) = n \lg n$  also gives:

$$T(2) = 2 \cdot \lg 2 = 2 \cdot 1 = 2.$$

Thus, the base case holds.

**Step 2: Inductive Hypothesis**

Assume that  $T(k) = k \lg k$  holds for some  $k = 2^m$  (i.e.,  $k$  is a power of 2) and all  $k \leq n$  for a particular  $n = 2^m$ .

**Step 3: Inductive Step**

We need to show that if  $T(n) = n \lg n$  holds, then  $T(2n) = (2n) \lg(2n)$  also holds.

Using the recurrence relation for  $T(n)$ , we have:

$$T(2n) = 2T\left(\frac{2n}{2}\right) + 2n = 2T(n) + 2n.$$

Now, substitute the inductive hypothesis  $T(n) = n \lg n$  into the equation:

$$T(2n) = 2(n \lg n) + 2n.$$

**Step 4: Simplify**

Using the logarithmic identity  $\lg(2n) = \lg 2 + \lg n = 1 + \lg n$ :

$$T(2n) = 2n \lg n + 2n = 2n(\lg n + 1) = 2n \lg(2n).$$

Thus, the recurrence relation holds for  $T(2n) = (2n) \lg(2n)$ .

## Question No 22:

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

### Answer:

#### Recursive Insertion Sort Pseudocode

```
RECURSIVE-INSERTION-SORT(A, n)
1. if n <= 1
2.     return
3. RECURSIVE-INSERTION-SORT(A, n - 1)
4. key = A[n]
5. i = n - 1
6. while i > 0 and A[i] > key
7.     A[i + 1] = A[i]
8.     i = i - 1
9. A[i + 1] = key
```

#### Explanation of the Algorithm

- **Base Case:** When  $n \leq 1$ , the array is already sorted, so we return immediately.
- **Recursive Call:** `RECURSIVE-INSERTION-SORT(A, n - 1)` sorts the subarray  $A[1 \dots n-1]$ .
- **Insertion Step:** After sorting  $A[1 \dots n-1]$ , we insert  $A[n]$  into its correct position by shifting elements greater than  $A[n]$  one position to the right until the correct position for  $A[n]$  is found.

#### Recurrence for Worst-Case Running Time

Let  $T(n)$  represent the worst-case running time of `RECURSIVE-INSERTION-SORT` on an array of size  $n$ .

The recurrence relation for  $T(n)$  is:

$$T(n) = T(n-1) + O(n) \\ T(n) = T(n-1) + O(n) \\ T(n) = T(n-1) + O(n)$$

#### Solution to the Recurrence

The recurrence  $T(n) = T(n-1) + O(n)$  is similar to the recurrence for the iterative version of insertion sort. Solving it using the method of summation:

$$T(n) = T(n-1) + n = T(n-2) + (n-1) + n = \dots = T(1) + (2+3+\dots+n) \\ T(n) = T(n-1) + n = T(n-2) + (n-1) + n = \dots = T(1) + (2+3+\dots+n)$$

Since  $T(1) = O(1)$  and the sum  $2+3+\dots+n = O(n^2)$ , we have:

$$T(n) = O(n^2)$$

## Final Answer

The worst-case running time of the recursive insertion sort is  $O(n^2)$ .

## Question No 23:

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  $\lg n$ .

## Answer:

### Binary Search Pseudocode

```
BINARY-SEARCH(A, v, low, high)
1. if low > high
2.     return NIL // v is not in the array
3. mid = (low + high) / 2
4. if A[mid] == v
5.     return mid // v is found at index mid
6. else if A[mid] > v
7.     return BINARY-SEARCH(A, v, low, mid - 1)
8. else
9.     return BINARY-SEARCH(A, v, mid + 1, high)
```

### Explanation of the Algorithm

1. **Base Case:** If  $low$  is greater than  $high$ , the element  $v$  is not present in the array, and we return  $NIL$ .
2. **Midpoint Calculation:** The midpoint  $mid$  is calculated as  $(low + high) / 2$ .
3. **Comparison:**
  - If  $A[mid] == v$ , the element  $v$  is found, and the function returns  $mid$ .
  - If  $A[mid] > v$ , the algorithm searches the left half of the array ( $low$  to  $mid - 1$ ).

- If  $A[\text{mid}] < v$ , the algorithm searches the right half of the array ( $\text{mid} + 1$  to  $\text{high}$ ).

Each recursive call halves the size of the subarray being searched, reducing the problem size significantly at each step.

## Worst-Case Running Time Analysis

The worst-case running time of binary search occurs when the algorithm must examine each possible halving until only one element remains. Let  $T(n)$  denote the time taken to search an array of size  $n$ :

1. **Recurrence Relation:** Each time binary search is called, the array size is halved. Therefore:

$$T(n) = T(n/2) + O(1)$$

where  $O(1)$  is the time taken to check the midpoint and decide which half to search next.

2. **Solving the Recurrence:** This recurrence expands as follows:

$$\begin{aligned} T(n) &= T(n/2) + O(1) = T(n/4) + O(1) + O(1) = \dots = T(1) + O(\log_2 n) \\ &= \log_2 n + O(1) \end{aligned}$$

Since  $T(1) = O(1)$ , the total time complexity is  $O(\log_2 n)$ .

## Question No 24:

The while loop of lines 537 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray  $A[1..j-1]$ . What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to  $\Theta(n \lg n)$ ?

**Answer:**

## Analysis of Insertion Sort with Binary Search

1. **Binary Search for Position Finding:** Using binary search to find the correct insertion position for the current element in the sorted portion  $A[1..j-1]$  reduces the number of comparisons from  $O(j)$  to  $O(\log j)$ . This optimization

decreases the time spent finding the insertion location in each iteration of the outer loop.

2. **Shifting Elements:** Even after finding the insertion point using binary search, we still need to shift all elements greater than the current element to the right to make space for insertion. In the worst case (e.g., when inserting the smallest element each time), we still need to shift  $j$  elements, taking  $O(j)O(j)O(j)$  time.

## Worst-Case Running Time

In the worst case, for each element  $A[j]$  (with  $j$  from 2 to  $n$ ), we:

- Use binary search to find the insertion point in  $O(\log j)$  time.
- Shift elements to the right to make space for insertion in  $O(j)$  time.

The total time complexity for insertion sort with binary search then becomes:

$$\sum_{j=2}^n (O(\log j) + O(j)) = \sum_{j=2}^n O(j) = O(n^2) \sum_{j=2}^n (O(\log j) + O(j)) = \sum_{j=2}^n j \log j + \sum_{j=2}^n j = O(n^2 \log n) + O(n^2) = O(n^2 \log n)$$

## Question No 25:

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  $O(n \lg n)$  time in the worst case.

## Answer:

### Algorithm: Two-Sum with Sorting and Two Pointers

1. **Sort** the set  $S$  of integers in  $O(n \log n)$  time.
2. Initialize two pointers:
  - Set `left` to the first element (smallest) in the sorted array.
  - Set `right` to the last element (largest) in the sorted array.
3. **Loop** until `left` meets `right`:
  - Calculate the sum of the elements at `left` and `right`.
  - If the sum equals  $x$ , return **True** (the two elements sum to  $x$ ).
  - If the sum is less than  $x$ , move the `left` pointer one position to the right to increase the sum.
  - If the sum is greater than  $x$ , move the `right` pointer one position to the left to decrease the sum.
4. If no such pair is found by the end of the loop, return **False**.

## Pseudocode

```
TWO-SUM( $S, x$ )
1. Sort  $S$  in ascending order.
```



```

2. left = 0
3. right = n - 1
4. while left < right
5.     sum = S[left] + S[right]
6.     if sum == x
7.         return True
8.     else if sum < x
9.         left = left + 1
10.    else
11.        right = right - 1
12. return False

```

## Question No 26:

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

### Answer:

In the original lower-bound argument for insertion sort, it's often assumed that the input size  $n$  is a multiple of 3, which simplifies the analysis by dividing the input into parts that can be analyzed independently. However, we can generalize the argument to handle arbitrary input sizes by using a more flexible approach.

The lower-bound argument typically uses a decision-tree model to analyze comparisons, showing that any comparison-based sorting algorithm must make at least  $\Omega(n \log n)$  comparisons in the worst case. For insertion sort, the number of comparisons required depends on the position of each element that needs to be inserted into the sorted portion of the list.

To handle input sizes  $n$  that are not a multiple of 3, consider the following adjustments:

1. **Generalized Decision-Tree Model:** For any input size  $n$ , there are  $n!$  possible permutations of the input. To identify the correct sorted order, insertion sort must differentiate between all these possibilities. This differentiation requires at least  $\log_2(n!)$  comparisons in the decision tree.
2. **Approximation with Stirling's Formula:** Using Stirling's approximation,  $\log_2(n!) \approx n \log_2(n) - n \log_2(e)$ , we can show that the number of comparisons required for sorting is at least  $\Omega(n \log n)$  even if  $n$  is not a multiple of 3.
3. **Insertion Sort's Structure:** Insertion sort iterates through each element in the list (from the second element to the  $n$ -th element), comparing it to the already sorted portion. In the worst case (when the input is reverse-sorted), each element at position  $i$  needs  $i$  comparisons to be placed in the correct position. Summing this over all elements from 1 to  $n-1$  gives:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Omega(n^2) \quad \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$$

Since this analysis doesn't rely on  $n$  being a multiple of 3, it holds for any input size. Thus, we can conclude that the lower bound of  $\Omega(n \log n)$  comparisons for insertion sort remains valid for arbitrary  $n$ .

### Code:

```
import math

def insertion_sort_comparisons(n):
    # Calculates the actual number of comparisons in the worst case for insertion sort
    comparisons = 0
    for i in range(1, n):
        comparisons += i # Each element needs i comparisons in the worst case
    return comparisons

def theoretical_lower_bound(n):
    # Calculates the theoretical lower bound using Stirling's approximation
    return int(n * math.log2(n)) # Approximation for lower bound

# Define the input size
n = int(input("Enter the input size n: "))

# Calculate the worst-case comparisons for insertion sort and theoretical lower bound
actual_comparisons = insertion_sort_comparisons(n)
lower_bound = theoretical_lower_bound(n)

# Print the results
print(f"Worst-case comparisons for insertion sort with n = {n}: {actual_comparisons}")
print(f"Theoretical lower bound ( $\Omega(n \log n)$ ) for n = {n}: {lower_bound}")
```

### Question No 27:

Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

### Answer:

### Selection Sort Algorithm Review

1. For each position  $i$  from 0 to  $n-2$ :
  - Find the minimum element in the subarray from  $i$  to  $n-1$ .
  - Swap this minimum element with the element at position  $i$ .

## Comparison Analysis

The number of comparisons in each iteration decreases as the sorted portion of the list grows:

1. In the first iteration ( $i=0$ ), the algorithm makes  $n-1$  comparisons to find the minimum of  $n$  elements.
2. In the second iteration ( $i=1$ ), it makes  $n-2$  comparisons to find the minimum of the remaining  $n-1$  elements.
3. This pattern continues until the second-to-last element, which requires only 1 comparison.

The total number of comparisons  $C(n)$  for selection sort is the sum:

$$C(n) = (n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2} = O(n^2)$$

## Swap Analysis

Selection sort performs at most one swap per iteration to place the minimum element in its correct position. Therefore, the total number of swaps is  $n-1$  (one for each element except the last).

## Total Running Time

The running time of selection sort is dominated by the number of comparisons, which grows quadratically with  $n$ . Thus, the total running time of selection sort is:

$$T(n) = O(n^2)$$

## Summary

- **Comparisons:**  $\frac{n^2 - n}{2} = O(n^2)$
- **Swaps:**  $n - 1 = O(n)$

## Comparison with Insertion Sort

- **Selection Sort** has  $O(n^2)$  comparisons regardless of the input order (even if the array is already sorted).
- **Insertion Sort**, however, can be faster on nearly sorted inputs, as it makes fewer comparisons in that case. It still has a worst-case time complexity of  $O(n^2)$  but can have better performance on average for partially sorted data.

Thus, the worst-case running time of selection sort remains  $O(n^2)$ , similar to insertion sort. However, insertion sort may perform better on nearly sorted arrays due to fewer necessary comparisons and swaps in such cases.

### Question No 28:

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

### Answer:

To generalize the lower-bound argument for insertion sort for an input in which the  $\alpha n$  largest values start in the first  $\alpha n$  positions, let's break down the approach.

### Problem Setup

1. Let  $\alpha$  be a fraction where  $0 < \alpha < 1$ .
2. Assume we have an array of size  $n$ .
3. The  $\alpha n$  largest values are positioned in the first  $\alpha n$  positions of the array.

The goal is to analyze the lower bound for the number of comparisons insertion sort would require to sort this array, given that these largest values are initially "out of order" in the first  $\alpha n$  positions.

### Step 1: Moving the Largest $\alpha n$ Values

In this setup:

- The largest  $\alpha n$  elements need to be moved from the first  $\alpha n$  positions to their correct positions at the end of the array.
- The remaining  $(1-\alpha)n$  elements are in the correct relative order, but they will still be affected by the initial disorder.

### Step 2: Counting Necessary Comparisons

1. **Middle Segment:** Each of the  $\alpha n$  largest elements must pass through the middle segment of the array (from positions  $\alpha n$  to  $(1-\alpha)n$ ), displacing the elements in this segment as they make their way toward the end of the array.
2. **Comparisons in the Middle Segment:** For each element in the first  $\alpha n$  positions to reach its sorted place in the last  $\alpha n$  positions, it must pass through each of the  $(1-2\alpha)n$  positions in the middle segment of the array.

- Therefore, the number of comparisons each of the  $\alpha n$  largest elements makes with elements in the middle segment is proportional to  $(1-2\alpha)n$ .
- 3. **Total Comparisons:** For each of the  $\alpha n$  largest elements, the number of comparisons made while traversing through the middle segment is approximately:

$$\alpha n \times (1-2\alpha)n = \alpha(1-2\alpha)n^2$$

$$\alpha n \times (1-2\alpha)n = \alpha(1-2\alpha)n^2$$

### Step 3: Maximizing Comparisons

To maximize the number of comparisons, we need to maximize the product  $\alpha(1-2\alpha)$ . This can be done by taking the derivative and setting it to zero:

1. Let  $f(\alpha) = \alpha(1-2\alpha)$
2. Differentiate with respect to  $\alpha$ :  $f'(\alpha) = 1 - 4\alpha$
3. Solving for  $\alpha$ , we find:  $\alpha = \frac{1}{4}$

Thus, setting  $\alpha = \frac{1}{4}$  maximizes the number of times the  $\alpha n$  largest values must pass through the middle  $(1-2\alpha)n$  positions.

### Question No 29:

Let  $f(n)$  and  $g(n)$  be asymptotically nonnegative functions. Using the basic definition of  $\Theta$ -notation, prove that  $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$ .

**Answer:**

$$\max(f(n), g(n)) = \Theta(f(n) + g(n)),$$

we need to show that there exist constants  $c_1, c_2$  and  $n_0$  such that for all  $n \geq n_0$ :

$$c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n)).$$

**Proof:**

**Step 1: Prove the Upper Bound**

We want to show that:

$$\max(f(n), g(n)) \leq c_2(f(n) + g(n))$$

for some constant  $c_2 > 0$ .

Observe that:

$$\max(f(n), g(n)) \leq f(n) + g(n).$$

This inequality holds because, by definition, the maximum of two functions is at most their sum.

Therefore, we can choose  $c_2 = 1$ , giving:

$$\max(f(n), g(n)) \leq f(n) + g(n) = c_2(f(n) + g(n)).$$

This establishes the upper bound with  $c_2 = 1$ .

**Step 2: Prove the Lower Bound**

We need to show that:

$$c_1(f(n) + g(n)) \leq \max(f(n), g(n))$$

for some constant  $c_1 > 0$  and sufficiently large  $n$ .

There are two cases to consider for  $\max(f(n), g(n))$ :

1. **Case 1:**  $f(n) \geq g(n)$ .

In this case,  $\max(f(n), g(n)) = f(n)$ , so we have:

$$f(n) \geq \frac{1}{2}(f(n) + g(n)),$$

which implies

$$\max(f(n), g(n)) \geq \frac{1}{2}(f(n) + g(n)).$$

Here, we can choose  $c_1 = \frac{1}{2}$ .

2. **Case 2:**  $g(n) \geq f(n)$ .

In this case,  $\max(f(n), g(n)) = g(n)$ , so similarly:

$$g(n) \geq \frac{1}{2}(f(n) + g(n)),$$

which implies

$$\max(f(n), g(n)) \geq \frac{1}{2}(f(n) + g(n)).$$

Again,  $c_1 = \frac{1}{2}$  works.

In both cases, we have shown that:

$$\max(f(n), g(n)) \geq \frac{1}{2}(f(n) + g(n)).$$

Thus, we can take  $c_1 = \frac{1}{2}$  and  $c_2 = 1$  to satisfy the definition of  $\Theta$ -notation.

**Conclusion**

Since we have found constants  $c_1$  and  $c_2$  such that:

$$c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n)),$$

we conclude that:

$$\max(f(n), g(n)) = \Theta(f(n) + g(n)).$$

## Question No 30:

Explain why the statement, "The running time of algorithm A is at least  $O(n^2)$ ," is meaningless

### Answer:

The statement, "The running time of algorithm A is at least  $O(n^2)$ ," is meaningless because of a misunderstanding of **Big-O notation** and its intended purpose.

### Understanding Big-O Notation

Big-O notation,  $O(g(n))$ , describes an **upper bound** on the growth rate of a function. Specifically, if we say an algorithm's running time  $T(n)$  is  $O(n^2)$ , it means that:

$$T(n) \leq c \cdot n^2 \quad \text{for some constant } c > 0 \text{ and sufficiently large } n$$

for some constant  $c > 0$  and sufficiently large  $n$ . This tells us that **the running time will not grow faster than  $n^2$  asymptotically**.

### Why "At Least $O(n^2)$ " is Meaningless

Since Big-O notation represents an upper bound, using it with the phrase "at least" doesn't make sense. Saying "at least  $O(n^2)$ " implies that  $O(n^2)$  is a lower bound, which is incorrect because:

- **Big-O does not provide a lower bound;** it only indicates that the growth rate does not exceed  $n^2$  up to a constant factor.

### Correct Way to Express Lower Bound

To describe a lower bound, we use **Big-Omega notation**,  $\Omega(g(n))$ . If the running time of algorithm A is at least quadratic, we should say:

$$T(n) = \Omega(n^2)$$

This statement means that the running time will grow at least as fast as  $n^2$  for sufficiently large  $n$ .

### Summary

- Saying "at least  $O(n^2)$ " is meaningless because  $O(n^2)$  implies an upper bound, not a lower bound.
- To indicate a lower bound, we use  $\Omega$ -notation, which would make the correct statement: "The running time of algorithm A is at least  $\Omega(n^2)$ ."

### Question No 31:

Is  $2^{n+1} = O(2^n)$ ? Is  $2^{2n} = O(2^n)$ ?

**Answer:**

**Statement 1:**  $2^{n+1} = O(2^n)$

Rewrite  $2^{n+1}$  as:

$$2^{n+1} = 2 \cdot 2^n.$$

Since 2 is a constant, we can see that:

$$2^{n+1} = 2 \cdot 2^n = O(2^n).$$

Thus, **this statement is true** because multiplying  $2^n$  by a constant factor (in this case, 2) does not change its asymptotic growth rate.

**Statement 2:**  $2^{2n} = O(2^n)$

Rewrite  $2^{2n}$  as:

$$2^{2n} = (2^n)^2.$$

Here,  $2^{2n}$  grows much faster than  $2^n$  because  $(2^n)^2$  increases exponentially relative to  $2^n$ . This means that  $2^{2n}$  is not asymptotically bounded above by  $2^n$ .

In fact, we have:

$$\frac{2^{2n}}{2^n} = 2^n \rightarrow \infty \quad \text{as} \quad n \rightarrow \infty.$$

Since  $2^{2n}$  grows faster than  $2^n$ , **this statement is false**. We would actually write:

$$2^{2n} = \omega(2^n),$$

where  $\omega$ -notation indicates asymptotic growth that is strictly greater than  $2^n$ .



## Question No 32:

Prove Theorem 3.1.

### Answer:

For any two functions  $f(n)$  and  $g(n)$ , we have  $f(n) = \Theta(g(n))$  if and only if  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .

This theorem states that  $f(n)$  is  $\Theta(g(n))$  if and only if it is both  $O(g(n))$  (an upper bound) and  $\Omega(g(n))$  (a lower bound).

**Proof:**

**Definitions Recap**

1. Big-O:  $f(n) = O(g(n))$  if there exist constants  $c > 0$  and  $n_0$  such that:

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

2. Big-Omega:  $f(n) = \Omega(g(n))$  if there exist constants  $c' > 0$  and  $n_1$  such that:

$$f(n) \geq c' \cdot g(n) \quad \text{for all } n \geq n_1.$$

3. Theta:  $f(n) = \Theta(g(n))$  if there exist constants  $c_1, c_2 > 0$  and  $n_2$  such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_2.$$

To prove the theorem, we need to show two implications:

- If  $f(n) = \Theta(g(n))$ , then  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .
- If  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ , then  $f(n) = \Theta(g(n))$ .

#### Proof of Implication 1

Assume  $f(n) = \Theta(g(n))$ . Then, by definition of  $\Theta$ -notation, there exist constants  $c_1, c_2 > 0$  and  $n_2$  such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_2.$$

1. To show that  $f(n) = O(g(n))$ :

- From the inequality  $f(n) \leq c_2 \cdot g(n)$ , we see that  $f(n)$  is bounded above by  $c_2 \cdot g(n)$  for all  $n \geq n_2$ .
- Therefore, by the definition of Big-O,  $f(n) = O(g(n))$ .

2. To show that  $f(n) = \Omega(g(n))$ :

- From the inequality  $c_1 \cdot g(n) \leq f(n)$ , we see that  $f(n)$  is bounded below by  $c_1 \cdot g(n)$  for all  $n \geq n_2$ .
- Therefore, by the definition of Big-Omega,  $f(n) = \Omega(g(n))$ .

Thus, if  $f(n) = \Theta(g(n))$ , then both  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .

#### Proof of Implication 2

Now, assume that  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .

1. Since  $f(n) = O(g(n))$ , there exist constants  $c_2 > 0$  and  $n_0$  such that:

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

2. Since  $f(n) = \Omega(g(n))$ , there exist constants  $c_1 > 0$  and  $n_1$  such that:

$$f(n) \geq c_1 \cdot g(n) \quad \text{for all } n \geq n_1.$$

3. Now let  $n_2 = \max\{n_0, n_1\}$ , which is the point from which both inequalities hold. Then for all  $n \geq n_2$ , we have:

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

This satisfies the definition of  $f(n) = \Theta(g(n))$ , with constants  $c_1$  and  $c_2$  and starting from  $n_2$ .

#### Conclusion

We have shown both directions:

- If  $f(n) = \Theta(g(n))$ , then  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .
- If  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ , then  $f(n) = \Theta(g(n))$ .

Therefore, we conclude that:

$$f(n) = \Theta(g(n)) \quad \text{if and only if} \quad f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

## Question No 33:

Prove The running time of an algorithm is  $\Theta(g(n))$  if and only if its **worst-case** running time is  $O(g(n))$  and its **best-case** running time is  $\Omega(g(n))$ .

### Answer:

#### Proof

##### Definitions Recap

1. **Theta:**  $T(n) = \Theta(g(n))$  if there exist constants  $c_1, c_2 > 0$  and  $n_0$  such that

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

This means that  $T(n)$  is asymptotically bounded both above and below by  $g(n)$ , so  $T(n)$  grows at the same rate as  $g(n)$ .

2. **Big-O (Worst-Case):**  $T_{\text{worst}}(n) = O(g(n))$  means that the algorithm's worst-case running time is bounded above by  $g(n)$  for large  $n$ .
3. **Big-Omega (Best-Case):**  $T_{\text{best}}(n) = \Omega(g(n))$  means that the algorithm's best-case running time is bounded below by  $g(n)$  for large  $n$ .

##### Proving the Implication

**Forward Direction:** If  $T(n) = \Theta(g(n))$ , then  $T_{\text{worst}}(n) = O(g(n))$  and  $T_{\text{best}}(n) = \Omega(g(n))$ .

Assume  $T(n) = \Theta(g(n))$ . By the definition of  $\Theta$ -notation, there exist constants  $c_1, c_2 > 0$  and  $n_0$  such that

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

1. Since  $T(n) \leq c_2 \cdot g(n)$  for all  $n \geq n_0$ , it follows that the worst-case running time  $T_{\text{worst}}(n) \leq T(n) \leq c_2 \cdot g(n)$  for  $n \geq n_0$ . This means:

$$T_{\text{worst}}(n) = O(g(n)).$$

2. Since  $T(n) \geq c_1 \cdot g(n)$  for all  $n \geq n_0$ , it follows that the best-case running time  $T_{\text{best}}(n) \geq T(n) \geq c_1 \cdot g(n)$  for  $n \geq n_0$ . This means:

$$T_{\text{best}}(n) = \Omega(g(n)).$$

Thus, if  $T(n) = \Theta(g(n))$ , then  $T_{\text{worst}}(n) = O(g(n))$  and  $T_{\text{best}}(n) = \Omega(g(n))$ .

**Reverse Direction:** If  $T_{\text{worst}}(n) = O(g(n))$  and  $T_{\text{best}}(n) = \Omega(g(n))$ , then  $T(n) = \Theta(g(n))$ .

Assume that  $T_{\text{worst}}(n) = O(g(n))$  and  $T_{\text{best}}(n) = \Omega(g(n))$ .

1. Since  $T_{\text{worst}}(n) = O(g(n))$ , there exists a constant  $c_2 > 0$  and an integer  $n_1$  such that

$$T(n) \leq T_{\text{worst}}(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_1.$$

2. Since  $T_{\text{best}}(n) = \Omega(g(n))$ , there exists a constant  $c_1 > 0$  and an integer  $n_2$  such that

$$T(n) \geq T_{\text{best}}(n) \geq c_1 \cdot g(n) \quad \text{for all } n \geq n_2.$$

3. Let  $n_0 = \max(n_1, n_2)$ . Then for all  $n \geq n_0$ , we have:

$$c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n).$$

This is precisely the definition of  $T(n) = \Theta(g(n))$ .

##### Conclusion

We have shown both directions:

1. If  $T(n) = \Theta(g(n))$ , then  $T_{\text{worst}}(n) = O(g(n))$  and  $T_{\text{best}}(n) = \Omega(g(n))$ .
2. If  $T_{\text{worst}}(n) = O(g(n))$  and  $T_{\text{best}}(n) = \Omega(g(n))$ , then  $T(n) = \Theta(g(n))$ .

### Question No 34:

prove that  $o(g(n)) \cap \omega(g(n)) = \emptyset$

### Answer:

#### Proof

To show that  $o(g(n)) \cap \omega(g(n)) = \emptyset$ , suppose for the sake of contradiction that there exists a function  $f(n)$  such that  $f(n) \in o(g(n))$  and  $f(n) \in \omega(g(n))$ .

1. If  $f(n) \in o(g(n))$ , then for every positive constant  $c$ , there exists an  $n_0$  such that:

$$0 \leq f(n) < c \cdot g(n) \quad \text{for all } n \geq n_0.$$

2. If  $f(n) \in \omega(g(n))$ , then for every positive constant  $c$ , there exists an  $n_1$  such that:

$$f(n) > c \cdot g(n) \quad \text{for all } n \geq n_1.$$

Now, choose some positive constant  $c$ . By the definition of  $o(g(n))$ , there is some  $n_0$  such that  $f(n) < c \cdot g(n)$  for all  $n \geq n_0$ . At the same time, by the definition of  $\omega(g(n))$ , there exists an  $n_1$  such that  $f(n) > c \cdot g(n)$  for all  $n \geq n_1$ .

Let  $n_2 = \max(n_0, n_1)$ . For  $n \geq n_2$ , both conditions would need to hold:

$$f(n) < c \cdot g(n) \quad \text{and} \quad f(n) > c \cdot g(n),$$

which is a contradiction.

#### Conclusion

Since both conditions cannot hold simultaneously for any function  $f(n)$ , we conclude that:

$$o(g(n)) \cap \omega(g(n)) = \emptyset.$$

## Question No 35:

We can extend our notation to the case of two parameters  $n$  and  $m$  that can go to  $\infty$  independently at different rates. For a given function  $g(n, m)$ , we denote by  $O(g(n, m))$  the set of functions

$$O(g(n, m)) = \{f(n, m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \\ \text{such that } 0 \leq f(n, m) \leq c g(n, m) \\ \text{for all } n \geq n_0 \text{ or } m \geq m_0\}.$$

Give corresponding definitions for  $\Omega(g(n, m))$  and  $\Theta(g(n, m))$ .

## Answer:

To extend the asymptotic notation to two parameters  $n$  and  $m$ , we can define the corresponding  $\Omega$  (Big-Omega) and  $\Theta$  (Theta) notations in a similar manner to how we defined  $O(g(n, m))$ . Here are the definitions:

### Definitions

1. **Big-Omega Notation  $\Omega(g(n, m))$ :** The set of functions  $f(n, m)$  is in  $\Omega(g(n, m))$  if there exist positive constants  $c, n_0$ , and  $m_0$  such that:

$$f(n, m) \geq c \cdot g(n, m) \quad \text{for all } n \geq n_0 \text{ or } m \geq m_0.$$

This means that  $f(n, m)$  grows at least as fast as  $g(n, m)$  when either  $n$  or  $m$  is sufficiently large.

2. **Theta Notation  $\Theta(g(n, m))$ :** The set of functions  $f(n, m)$  is in  $\Theta(g(n, m))$  if there exist positive constants  $c_1, c_2, n_0$ , and  $m_0$  such that:

$$c_1 \cdot g(n, m) \leq f(n, m) \leq c_2 \cdot g(n, m) \quad \text{for all } n \geq n_0 \text{ or } m \geq m_0.$$

This means that  $f(n, m)$  grows at the same rate as  $g(n, m)$  when either  $n$  or  $m$  is sufficiently large.

### Summary

- **Big-O Notation:**

$$O(g(n, m)) = \{f(n, m) \mid \exists c > 0, n_0, m_0 \text{ such that } 0 \leq f(n, m) \leq c \cdot g(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$$

- **Big-Omega Notation:**

$$\Omega(g(n, m)) = \{f(n, m) \mid \exists c > 0, n_0, m_0 \text{ such that } f(n, m) \geq c \cdot g(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$$

- **Theta Notation:**

$$\Theta(g(n, m)) = \{f(n, m) \mid \exists c_1, c_2 > 0, n_0, m_0 \text{ such that } c_1 \cdot g(n, m) \leq f(n, m) \leq c_2 \cdot g(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$$

## Question No 36:

Show that if  $f(n)$  and  $g(n)$  are monotonically increasing functions, then so are the functions  $f(n) + g(n)$  and  $f(g(n))$ , and if  $f(n)$  and  $g(n)$  are in addition nonnegative, then  $f(n) \cdot g(n)$  is monotonically increasing.

## Answer:

### 1. Proving that $f(n) + g(n)$ is monotonically increasing

Let  $f(n)$  and  $g(n)$  be monotonically increasing functions. We need to show that  $f(n) + g(n)$  is also monotonically increasing.

Let  $n_1 < n_2$ . Since  $f(n)$  is monotonically increasing, we have:

$$f(n_1) \leq f(n_2).$$

Similarly, since  $g(n)$  is monotonically increasing, we have:

$$g(n_1) \leq g(n_2).$$

Adding these inequalities, we get:

$$f(n_1) + g(n_1) \leq f(n_2) + g(n_2).$$

Thus, we conclude that:

$$f(n) + g(n) \text{ is monotonically increasing.}$$

which means that  $f(n) + g(n)$  is monotonically increasing.

### 2. Proving that $f(g(n))$ is monotonically increasing

Next, we need to show that  $f(g(n))$  is monotonically increasing.

Let  $n_1 < n_2$ . Since  $g(n)$  is monotonically increasing, we have:

$$g(n_1) \leq g(n_2).$$

Since  $f(n)$  is monotonically increasing, and  $g(n_1)$  and  $g(n_2)$  are its inputs, we can apply the monotonicity of  $f$ :

$$f(g(n_1)) \leq f(g(n_2)).$$

Therefore, we conclude that:

$$f(g(n)) \text{ is monotonically increasing.}$$

which means that  $f(g(n))$  is also monotonically increasing.

### 3. Proving that $f(n) \cdot g(n)$ is monotonically increasing if $f(n)$ and $g(n)$ are nonnegative

Now, we assume that  $f(n)$  and  $g(n)$  are also nonnegative. We want to show that the product  $f(n) \cdot g(n)$  is monotonically increasing.

Let  $n_1 < n_2$ . Since  $f(n)$  and  $g(n)$  are both nonnegative and monotonically increasing, we have:

$$f(n_1) \leq f(n_2) \quad \text{and} \quad g(n_1) \leq g(n_2).$$

We want to show that:

$$f(n_1) \cdot g(n_1) \leq f(n_2) \cdot g(n_2).$$

Using the identity for the difference of products:

$$f(n_2) \cdot g(n_2) - f(n_1) \cdot g(n_1) = f(n_2)(g(n_2) - g(n_1)) + g(n_1)(f(n_2) - f(n_1)).$$

Since  $g(n_2) - g(n_1) \geq 0$  and  $f(n_2) - f(n_1) \geq 0$ , we analyze the two terms:

1. The first term  $f(n_2)(g(n_2) - g(n_1)) \geq 0$  since  $f(n_2) \geq 0$  and  $g(n_2) \geq g(n_1)$ .
2. The second term  $g(n_1)(f(n_2) - f(n_1)) \geq 0$  since  $g(n_1) \geq 0$  and  $f(n_2) \geq f(n_1)$ .

Since both terms are nonnegative, we have:

$$f(n_2) \cdot g(n_2) \geq f(n_1) \cdot g(n_1).$$

By rearranging, we find that:

$$f(n) \cdot g(n) \text{ is monotonically increasing.}$$

which confirms that  $f(n) \cdot g(n)$  is monotonically increasing.

## Question No 37:

Prove that  $\lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil = n$  for any integer  $n$  and real number  $\alpha$  in the range  $0 \leq \alpha \leq 1$ .

## Answer:

$$\lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil = n$$

for any integer  $n$  and real number  $\alpha$  in the range  $0 \leq \alpha \leq 1$ , let's start by breaking down the expression step by step.

### Notation

1.  $\lfloor \alpha n \rfloor$  denotes the floor function  $\lfloor \alpha n \rfloor$ , which is the greatest integer less than or equal to  $\alpha n$ .
2.  $\lceil (1 - \alpha)n \rceil$  denotes  $\lceil (1 - \alpha)n \rceil$ .

### Proof

1. Expressing the Components:

$$\text{Let } k = \lfloor \alpha n \rfloor.$$

Therefore, we can express  $\alpha n$  as:

$$k \leq \alpha n < k + 1$$

This leads to:

$$k \leq \alpha n \quad \text{and} \quad \alpha n < k + 1.$$

2. Rearranging the Original Expression:

Now, substituting  $k$  back into the original expression gives:

$$k + \lceil (1 - \alpha)n \rceil = \lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil.$$

3. Simplifying:

We know:

$$\lceil (1 - \alpha)n \rceil = (1 - \alpha)n + \{ \alpha n \}.$$

Therefore, we can rewrite the expression:

$$k + (1 - \alpha)n = \lfloor \alpha n \rfloor + (1 - \alpha)n + \{ \alpha n \}.$$

4. Analyzing the Sum:

Now, we can analyze:

$$\lfloor \alpha n \rfloor + (1 - \alpha)n.$$

From the earlier definitions:

$$\lfloor \alpha n \rfloor = \alpha n - \{ \alpha n \}$$

where  $\{ \alpha n \}$  is the fractional part of  $\alpha n$ .

So:

$$\lfloor \alpha n \rfloor + (1 - \alpha)n = (\alpha n - \{ \alpha n \}) + (1 - \alpha)n.$$

This simplifies to:

$$\alpha n + (1 - \alpha)n - \{ \alpha n \} = n - \{ \alpha n \}.$$

5. Establishing the Final Relationship:

Now, we need to consider  $n - \{ \alpha n \}$ :

$$n - \{ \alpha n \} \leq n < n + 1,$$

since the fractional part  $\{ \alpha n \}$  is always non-negative and less than 1.

6. Conclusion:

The sum  $\lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil$  is thus equal to  $n - \{ \alpha n \}$ , and since  $\{ \alpha n \}$  is strictly less than 1, we conclude that:

$$\lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil = n.$$

Therefore, we have shown that

$$\lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil = n$$

for any integer  $n$  and for any  $\alpha$  in the range  $0 \leq \alpha \leq 1$ .

## Question No 38:

Use equation (3.14) or other means to show that  $(n + o(n))^k = \Theta(n^k)$  for any real constant  $k$ . Conclude that  $\lceil n \rceil^k = \Theta(n^k)$  and  $\lfloor n \rfloor^k = \Theta(n^k)$ .

## Answer:

### Binomial Expansion

To analyze  $(n + o(n))^k$ , we can use the binomial expansion:

$$(n + o(n))^k = \sum_{i=0}^k \binom{k}{i} n^{k-i} o(n)^i.$$

Now, let's break this down into its components:

1. When  $i = 0$ :

$$\binom{k}{0} n^k o(n)^0 = n^k.$$

2. When  $i \geq 1$ : For each  $i \geq 1$ , we have:

$$\binom{k}{i} n^{k-i} o(n)^i.$$

Since  $o(n)$  is smaller than any linear multiple of  $n$ , we know that for sufficiently large  $n$ :

$$|o(n)| < \epsilon n \quad \text{for any small } \epsilon > 0.$$

This implies that  $o(n)^i$  will grow slower than  $n^i$  and thus  $n^{k-i} o(n)^i$  will grow slower than  $n^{k-i+1}$ .

### Main Result

Combining all the parts of the expansion, we can express  $(n + o(n))^k$ :

$$(n + o(n))^k = n^k + \sum_{i=1}^k \binom{k}{i} n^{k-i} o(n)^i.$$

The leading term in the expansion is  $n^k$ , and the remaining terms become negligible as  $n \rightarrow \infty$ .

### Conclusion

As  $n$  grows, the additional terms (from  $i \geq 1$ ) do not change the growth rate significantly:

$$(n + o(n))^k = n^k + (\text{lower-order terms}).$$

Thus, we conclude:

$$(n + o(n))^k = \Theta(n^k).$$

### Specific Cases for $\lfloor n \rfloor^k$ and $\lceil n \rceil^k$

1. For  $\lfloor n \rfloor^k$ : Since  $\lfloor n \rfloor = n + o(n)$  (where  $o(n)$  is bounded by 1), we have:

$$\lfloor n \rfloor^k = (n + o(n))^k = \Theta(n^k).$$

2. For  $\lceil n \rceil^k$ : Similarly, since  $\lceil n \rceil = n + o(n)$  (again where  $o(n)$  is bounded), we also have:

$$\lceil n \rceil^k = (n + o(n))^k = \Theta(n^k).$$

### Final Statement

Thus, we conclude:

$$\lfloor n \rfloor^k = \Theta(n^k) \quad \text{and} \quad \lceil n \rceil^k = \Theta(n^k).$$

## Question No 39:

Prove the following:

- a. Equation (3.21).
- b. Equations (3.26)–(3.28).
- c.  $\lg(\Theta(n)) = \Theta(\lg n)$ .

Answer:

### Definitions

1. Big-O Notation: We say  $f(n) = O(g(n))$  if there exist positive constants  $c$  and  $n_0$  such that  $|f(n)| \leq c|g(n)|$  for all  $n \geq n_0$ .
2. Big-Theta Notation: We say  $f(n) = \Theta(g(n))$  if  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ . This means there exist positive constants  $c_1, c_2, n_0$  such that:

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

3. Logarithmic Functions:  $\lg(n)$  refers to the logarithm base 2 of  $n$ .

### a. Proof of Equation (3.21)

Equation (3.21) is often given as:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) = O(n^k) \text{ for some } k \geq 0.$$

This is a standard form of the Master Theorem. The theorem provides a way to analyze the complexity of divide-and-conquer algorithms.

1. If  $f(n)$  is polynomially smaller than  $n^{\lg b}$  (specifically, if  $f(n) = O(n^{\lg b - \epsilon})$  for some  $\epsilon > 0$ ), then:

$$T(n) = \Theta(n^{\lg b}).$$

2. If  $f(n)$  is asymptotically equal to  $n^{\lg b}$  (i.e.,  $f(n) = \Theta(n^{\lg b})$ ), then:

$$T(n) = \Theta(n^{\lg b} \log n).$$

3. If  $f(n)$  is polynomially larger than  $n^{\lg b}$  and satisfies regularity conditions, then:

$$T(n) = \Theta(f(n)).$$

Thus, Equation (3.21) provides a framework to analyze recursive functions.

### b. Proof of Equations (3.26) to (3.28)

Equations (3.26) to (3.28) are generally derived from the properties of the Master Theorem.

Equation (3.26) typically states that:

$$T(n) = O(n^k) \text{ for some } k.$$

This means that if  $f(n)$  is  $O(n^k)$  and is dominated by the function  $T(n)$  described in (3.21), we can conclude that the running time grows at most polynomially.

Equation (3.27) states:

$$T(n) = \Theta(n^k \log n) \text{ for some } k \geq 0.$$

This is derived from the case where  $f(n)$  is asymptotically equal to  $n^{\lg b}$ .

Equation (3.28) states:

$$T(n) = \Theta(f(n)) \text{ if } f(n) \text{ is larger.}$$

In this case, the function  $f(n)$  dominates the growth of  $T(n)$ .

### c. Proof that $\lg(\Theta(n)) = \Theta(\lg n)$

To show that  $\lg(\Theta(n)) = \Theta(\lg n)$ , we start with the definition of  $\Theta(n)$ .

Let  $f(n) = \Theta(n)$ . This means there exist constants  $c_1, c_2, n_0$  such that:



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

Taking the logarithm of all sides:

$$\lg(c_1 n) \leq \lg(f(n)) \leq \lg(c_2 n).$$

Using the properties of logarithms:

$$\lg c_1 + \lg n \leq \lg(f(n)) \leq \lg c_2 + \lg n.$$

As  $n \rightarrow \infty$ , the terms  $\lg c_1$  and  $\lg c_2$  become negligible compared to  $\lg n$ . Thus, we can simplify this to:

$$\lg n + O(1) \leq \lg(f(n)) \leq \lg n + O(1).$$

This implies that:

$$\lg(f(n)) = \Theta(\lg n).$$

### Conclusion

Putting it all together, we have shown:

1.  $T(n) = O(n^4)$  and related equations.
2.  $\lg(\Theta(n)) = \Theta(\lg n)$ .

Thus, all three parts are proven. If you need specific details on each step or additional explanations, feel free to ask!

## Question No 40:

Is the function  $\lceil \lg n \rceil!$  polynomially bounded? Is the function  $\lceil \lg \lg n \rceil!$  polynomially bounded?

**Answer:**

### Definitions

1. **Polynomially Bounded:** A function  $f(n)$  is said to be polynomially bounded if there exists a positive constant  $k$  and a constant  $c$  such that:

$$f(n) \leq cn^k \quad \text{for sufficiently large } n.$$

This means that the function grows no faster than a polynomial function.

### Analysis of $\lg n$

1. **Function:**  $f(n) = \lg n$
2. **Growth Rate:** The logarithm function grows much slower than any polynomial function. Specifically:

$$\lg n \leq n^\epsilon \quad \text{for any } \epsilon > 0 \text{ and sufficiently large } n.$$

This means  $\lg n$  can be bounded above by a polynomial function of any degree.

3. **Conclusion:** Therefore,  $\lg n$  is polynomially bounded.

### Analysis of $\lg(\lg n)$

1. **Function:**  $f(n) = \lg(\lg n)$
2. **Growth Rate:** The function  $\lg(\lg n)$  also grows slower than any polynomial function. To see this, observe that for sufficiently large  $n$ :

$$\lg(\lg n) \leq \lg n \quad \text{and } \lg n \text{ is already polynomially bounded.}$$

In fact, for any  $\epsilon > 0$ :

$$\lg(\lg n) \leq n^\epsilon \quad \text{for sufficiently large } n.$$

This holds because  $\lg n$  grows slower than  $n^\epsilon$ .

3. **Conclusion:** Thus,  $\lg(\lg n)$  is also polynomially bounded.

### Final Answer

- Is  $\lg n$  polynomially bounded? Yes.
- Is  $\lg(\lg n)$  polynomially bounded? Yes.

## Question No 41:

Which is asymptotically larger:  $\lg(\lg^* n)$  or  $\lg^*(\lg n)$ ?

**Answer:**

### Function Definitions

1. Function 1:  $f(n) = \lg(\lg(n))$
2. Function 2:  $g(n) = \lg(\sqrt{\lg(n)})$

### Analysis of Each Function

#### 1. Analysis of $f(n) = \lg(\lg(n))$

- The function  $f(n) = \lg(\lg(n))$  represents the logarithm of the logarithm of  $n$ .
- As  $n \rightarrow \infty$ ,  $\lg(n) \rightarrow \infty$ , and thus  $\lg(\lg(n)) \rightarrow \infty$  as well, but at a very slow rate.

#### 2. Analysis of $g(n) = \lg(\sqrt{\lg(n)})$

- The function  $g(n) = \lg(\sqrt{\lg(n)})$  can be rewritten using the properties of logarithms:

$$g(n) = \lg((\lg(n))^{1/2}) = \frac{1}{2} \lg(\lg(n)).$$

### Comparison

Now we can compare the two functions:

- We have:

$$f(n) = \lg(\lg(n)),$$

$$g(n) = \frac{1}{2} \lg(\lg(n)).$$

- Clearly,  $f(n)$  is twice  $g(n)$ :

$$f(n) = 2g(n).$$

### Asymptotic Growth Rate

To determine which function is asymptotically larger, we observe that:

$$\frac{f(n)}{g(n)} = \frac{\lg(\lg(n))}{\frac{1}{2} \lg(\lg(n))} = 2.$$

Since  $\frac{f(n)}{g(n)}$  approaches a constant (specifically, 2) as  $n \rightarrow \infty$ , we conclude that:

$$f(n) \sim 2g(n),$$

which implies that:

$$f(n) = \Theta(g(n)).$$

### Final Conclusion

Both  $\lg(\lg(n))$  and  $\lg(\sqrt{\lg(n)})$  grow at the same asymptotic rate. Hence, neither is asymptotically larger than the other.

- **Conclusion:**  $\lg(\lg(n))$  and  $\lg(\sqrt{\lg(n)})$  are asymptotically equivalent, i.e.,  $\lg(\lg(n)) \sim \lg(\sqrt{\lg(n)})$ .

## Question No 42:

Show that the golden ratio  $\phi$  and its conjugate  $\hat{\phi}$  both satisfy the equation  $x^2 = x + 1$ .

### Answer:

#### Step 1: Define the Golden Ratio and Its Conjugate

The golden ratio  $\phi$  is defined as:

$$\phi = \frac{1 + \sqrt{5}}{2}.$$

The conjugate of the golden ratio, often denoted as  $\psi$ , is:

$$\psi = \frac{1 - \sqrt{5}}{2}.$$

#### Step 2: Verify $\phi$ Satisfies the Equation

Substituting  $\phi$  into the equation  $x^2 = x + 1$ :

1. Calculate  $\phi^2$ :

$$\phi^2 = \left( \frac{1 + \sqrt{5}}{2} \right)^2 = \frac{(1 + \sqrt{5})^2}{4} = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2}.$$

2. Calculate  $\phi + 1$ :

$$\phi + 1 = \frac{1 + \sqrt{5}}{2} + 1 = \frac{1 + \sqrt{5} + 2}{2} = \frac{3 + \sqrt{5}}{2}.$$

3. Now compare  $\phi^2$  and  $\phi + 1$ :

$$\phi^2 = \frac{3 + \sqrt{5}}{2} \quad \text{and} \quad \phi + 1 = \frac{3 + \sqrt{5}}{2}.$$

Since both expressions are equal, we have:

$$\phi^2 = \phi + 1.$$

#### Step 3: Verify $\psi$ Satisfies the Equation

Now, substitute  $\psi$  into the equation:

1. Calculate  $\psi^2$ :

$$\psi^2 = \left( \frac{1 - \sqrt{5}}{2} \right)^2 = \frac{(1 - \sqrt{5})^2}{4} = \frac{1 - 2\sqrt{5} + 5}{4} = \frac{6 - 2\sqrt{5}}{4} = \frac{3 - \sqrt{5}}{2}.$$

2. Calculate  $\psi + 1$ :

$$\psi + 1 = \frac{1 - \sqrt{5}}{2} + 1 = \frac{1 - \sqrt{5} + 2}{2} = \frac{3 - \sqrt{5}}{2}.$$

3. Now compare  $\psi^2$  and  $\psi + 1$ :

$$\psi^2 = \frac{3 - \sqrt{5}}{2} \quad \text{and} \quad \psi + 1 = \frac{3 - \sqrt{5}}{2}.$$

Since both expressions are equal, we have:

$$\psi^2 = \psi + 1.$$

#### Conclusion

Both  $\phi$  (the golden ratio) and  $\psi$  (its conjugate) satisfy the equation  $x^2 = x + 1$ :

- $\phi^2 = \phi + 1$
- $\psi^2 = \psi + 1$

### Question No 43:

Prove by induction that the  $i$ th Fibonacci number satisfies the equation

$$F_i = (\phi^i - \hat{\phi}^i) / \sqrt{5},$$

where  $\phi$  is the golden ratio and  $\hat{\phi}$  is its conjugate.

### Answer:

$$F_i = \frac{\phi^i - \psi^i}{\sqrt{5}},$$

where  $\phi$  is the golden ratio defined as  $\phi = \frac{1+\sqrt{5}}{2}$  and  $\psi$  is its conjugate defined as  $\psi = \frac{1-\sqrt{5}}{2}$ , we will proceed with the following steps:

1. Base Case: Verify the formula for the first few Fibonacci numbers.
2. Induction Hypothesis: Assume the formula holds for  $n = k$  and  $n = k - 1$ .
3. Inductive Step: Show that it holds for  $n = k + 1$ .

#### Step 1: Base Case

We will check the formula for  $i = 0, 1$ , and  $2$ .

- For  $i = 0$ :

$$F_0 = 0,$$

$$\frac{\phi^0 - \psi^0}{\sqrt{5}} = \frac{1 - 1}{\sqrt{5}} = 0.$$

Thus,  $F_0 = \frac{\phi^0 - \psi^0}{\sqrt{5}}$  holds.

- For  $i = 1$ :

$$F_1 = 1,$$

$$\frac{\phi^1 - \psi^1}{\sqrt{5}} = \frac{\phi - \psi}{\sqrt{5}}.$$

Substituting  $\phi$  and  $\psi$ :

$$\phi - \psi = \frac{1 + \sqrt{5}}{2} - \frac{1 - \sqrt{5}}{2} = \sqrt{5}.$$

Therefore,

$$\frac{\phi - \psi}{\sqrt{5}} = \frac{\sqrt{5}}{\sqrt{5}} = 1.$$

Thus,  $F_1 = \frac{\phi^1 - \psi^1}{\sqrt{5}}$  holds.

- For  $i = 2$ :

$$F_2 = 1,$$

$$\frac{\phi^2 - \psi^2}{\sqrt{5}}.$$

We calculate  $\phi^2$  and  $\psi^2$ :

$$\phi^2 = \left( \frac{1 + \sqrt{5}}{2} \right)^2 = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2},$$

$$\psi^2 = \left( \frac{1 - \sqrt{5}}{2} \right)^2 = \frac{1 - 2\sqrt{5} + 5}{4} = \frac{6 - 2\sqrt{5}}{4} = \frac{3 - \sqrt{5}}{2}.$$

Now compute  $\phi^2 - \psi^2$ :

$$\phi^2 - \psi^2 = \left( \frac{3 + \sqrt{5}}{2} \right) - \left( \frac{3 - \sqrt{5}}{2} \right) = \frac{2\sqrt{5}}{2} = \sqrt{5}.$$

Therefore,

$$\frac{\phi^2 - \psi^2}{\sqrt{5}} = \frac{\sqrt{5}}{\sqrt{5}} = 1.$$

Thus,  $F_2 = \frac{\phi^2 - \psi^2}{\sqrt{5}}$  holds.

### Step 2: Induction Hypothesis

Assume that the formula holds for  $i = k$  and  $i = k - 1$ :

$$F_k = \frac{\phi^k - \psi^k}{\sqrt{5}},$$
$$F_{k-1} = \frac{\phi^{k-1} - \psi^{k-1}}{\sqrt{5}}.$$

### Step 3: Inductive Step

We need to show that:

$$F_{k+1} = \frac{\phi^{k+1} - \psi^{k+1}}{\sqrt{5}}.$$

Using the Fibonacci recurrence relation  $F_{k+1} = F_k + F_{k-1}$ :

$$F_{k+1} = \frac{\phi^k - \psi^k}{\sqrt{5}} + \frac{\phi^{k-1} - \psi^{k-1}}{\sqrt{5}}.$$

Combine the fractions:

$$F_{k+1} = \frac{\phi^k + \phi^{k-1} - \psi^k - \psi^{k-1}}{\sqrt{5}}.$$

Using the identities  $\phi^{k+1} = \phi^k + \phi^{k-1}$  and  $\psi^{k+1} = \psi^k + \psi^{k-1}$ :

$$F_{k+1} = \frac{(\phi^{k+1}) - (\psi^{k+1})}{\sqrt{5}}.$$

Thus, we have shown:

$$F_{k+1} = \frac{\phi^{k+1} - \psi^{k+1}}{\sqrt{5}}.$$

### Conclusion

By the principle of mathematical induction, the formula holds for all  $i \geq 0$ :

$$F_i = \frac{\phi^i - \psi^i}{\sqrt{5}}.$$

### Question No 44:

Show that  $k \lg k = \Theta(n)$  implies  $k = \Theta(n / \lg n)$ .

### Answer:

To show that  $k \lg k = \Theta(n)$  implies  $k = \Theta\left(\frac{n}{\lg n}\right)$ , we can start by manipulating the equation  $k \lg k$ .

1. **Given:**  $k \lg k = \Theta(n)$ . This means there exist constants  $c_1, c_2, n_0$  such that for all  $k \geq n_0$ :

$$c_1 n \leq k \lg k \leq c_2 n.$$

2. **Rearranging the bounds:**

- From the lower bound:

$$c_1 n \leq k \lg k \implies k \geq \frac{c_1 n}{\lg k}.$$

- From the upper bound:

$$k \lg k \leq c_2 n \implies k \leq \frac{c_2 n}{\lg k}.$$

3. **Using  $k = \Theta\left(\frac{n}{\lg n}\right)$ :**

- Assume  $k = c \frac{n}{\lg n}$  for some constant  $c$ . Then:

$$k \lg k = k \lg \left( c \frac{n}{\lg n} \right) = k (\lg c + \lg n - \lg(\lg n)).$$

4. **Substituting:**

- We can express  $k$  in terms of  $n$ :

$$k \lg k = c \frac{n}{\lg n} (\lg c + \lg n - \lg(\lg n)) \sim c \frac{n \lg n}{\lg n} = cn.$$

- Thus,  $k \lg k = \Theta(n)$  holds.

5. **Conclusion:** Therefore, we conclude:

$$k = \Theta\left(\frac{n}{\lg n}\right).$$

Thus, the assertion is proven.

