# ***DATA STRUCTURE***

# ***ASSIGNMENT NO 01***

**

## ***Fall 2023***

# ***Submitted by:***

# ***Eman Khan***

# ***Roll No:***

# ***14644***

# ***Submitted To:***

# ***Sir Jamal Ahad***

# ***Due Date:***

# ***28-OCT-2024***

***DEPARTMENT OF COMPUTER SCIENCE,***

***ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY (AUST)***

**CHAPTER NO 01:-**

**ALGORITHMS :-**

**EXERCISE NO 01:-**

**1.1-1:-** Describe your own real word example that require sorting. Describe that one requires finding the shortest distance between two points ?

# **Sorting Example:**

* LET’s organizing a delivery service for a food delivery app. Suppose we have a list of orders that need to be delivered, and we need to sort the list by delivery time (earliest to latest) to make sure the food is still fresh when it arrives.
* We can also sort orders by distance from the restaurant so that the nearest deliveries are completed first. This makes the process more efficient, as it ensures nearby deliveries are made first, saving time and fuel.

# **Shortest Distance Example:**

* For each delivery, the app also needs to find the **shortest route** from the restaurant to the customer’s address.
* By calculating the shortest distance, it minimizes the delivery time and fuel usage, which is crucial for both cost savings and ensuring that the food arrives hot and on time.

**1.1-2:-** Other than speed, what other measures of efûciency might you need to consider in a real-world setting?

* **Cost Efficiency**: This measures the financial resources used relative to the output produced. It can be assessed by comparing the cost of inputs to the value of outputs.
* **Resource Utilization**: This metric looks at how effectively resources (such as labor, materials, and equipment) are used. High resource utilization indicates that resources are being employed efficiently.
* **Energy Efficiency**: In many industries, especially manufacturing, the amount of energy consumed relative to the output produced is crucial. Lower energy consumption for the same output indicates higher efficiency.
* **Quality of Output**: Efficiency is not just about quantity; the quality of products or services is also critical. High-quality outputs with minimal defects or rework contribute to overall efficiency.
* **Throughput**: This refers to the amount of product or service produced in a given time frame. It helps assess how well a system converts inputs into outputs.
* **Downtime**: This metric tracks the time that a system or process is not operational. Reducing downtime can significantly improve overall efficiency.
* **Cycle Time**: This measures the total time from the beginning to the end of a process. Shorter cycle times can indicate improved efficiency.
* **Customer Satisfaction**: Efficiency can also be gauged by how well a product or service meets customer needs. High customer satisfaction often correlates with efficient processes.
* **Flexibility**: The ability to adapt to changes in demand or processes can also be a measure of efficiency. A flexible system can respond to new requirements without significant delays or costs.
* **Return on Investment (ROI)**: This financial metric evaluates the gain or loss generated relative to the investment made, helping to assess the efficiency of resource allocation.

**1.1-3:-** Select a data structure that you have seen, and discuss its strengths and limitations.

* **Strengths of Linked List**

1. Simpler addition and removal of elements, O(1)O(1) time complexity
2. Does not need contiguous memory space
3. New element can be easily inserted in any location

* **Limitations of Linked List**

1. Accessing an element by index or by value means traversing the list, O(n)O(n) time complexity
2. Additional memory is required for storing the address (pointer) of the next/previous element.

* **Strengths of Array**

1. Accessing any element by index is simple, O(1)O(1) time complexity
2. No additional memory required to store address

* **Limitations of Array**

1. Addition or removal of elements from any index but the last means re-arranging the whole list, O(n)O(n) time complexity
2. Accessing an element by value means traversing the list, O(n)O(n) time complexity
3. Needs contiguous memory

**1.1-4:-** How are the shortest-path and travelling-salesman problems given above similar? How are they different?

* The Shortest Path Problem and the Travelling Salesman Problem (TSP):

# **Similarities:**

1. **Graph-Based**:

Both problems use graphs where nodes represent locations and edges represent paths.

1. **Minimization Goal**:

Both aim to minimize travel costs, whether it’s distance or time.

### Differences:

**Objective**:

* 1. **Shortest Path Problem**: Finds the shortest route from one specific starting point to a specific destination.
  2. **Travelling Salesman Problem**: Requires visiting multiple locations exactly once and returning to the starting point.

**Complexity**:

* 1. **Shortest Path Problem**: Can be solved efficiently with algorithms like Dijkstra’s.
  2. **Travelling Salesman Problem**: NP-hard, meaning it’s much harder to solve, especially as the number of locations increases.

**Output**:

* 1. **Shortest Path Problem**: Provides a single shortest path between two points.
  2. **Travelling Salesman Problem**: Provides a complete tour visiting all points exactly once and returning to the start.

**1.1-5:-** Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is “approximately” the best is good enough.

# **Problem Requiring the Best Solution**

**Example: Heart Surgery**

In heart surgery, doctors must follow the best procedures and techniques to ensure the patient’s safety and recovery. Any mistakes can have serious consequences, so the best solution is crucial.

# **Problem Allowing an Approximately Best Solution**

**Example: Pizza Delivery**

When delivering pizzas, it's important to get them to customers quickly, but it's okay if the route isn't perfect. Using a route that’s “close enough” can still ensure the pizza arrives hot and on time, even if it’s not the absolute shortest path.

**1.1-6:-** Describe a real world problem in which sometimes the entire input is available before you need to solve the problem but others times the input is not entirely available in advance and arrives over times.

# **Real-World Problem: Event Planning**

## **Entire Input Available:**

* **Scenario**: When planning a wedding, the couple decides on all the details (venue, guest list, menu, and decorations) well in advance. The event planner can prepare everything because they have all the necessary information upfront.

## **Input Arriving Over Time:**

* **Scenario**: For a corporate event, guests may RSVP at different times. As responses come in, the planner must adjust seating arrangements, catering orders, and other logistics based on the number of attendees and their preferences. The information arrives gradually, and the planner has to adapt as new data comes in.

This example illustrates how event planning can involve both scenarios: having all details known upfront for a wedding versus receiving guest responses over time for a corporate event.

**EXERCISE 2:-**

**1.2-1:-** Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved

If you think about it, such examples are everywhere these days, much more than when CLRS was first published (1990). Here are few such examples…

# ***File Explorer***

Applies sorting algorithm whenever the user wants to sort the files according to the filenames or file type or date modified.

# **Netflix or Any streaming app**

Applies a handful of [algorithms](https://en.wikipedia.org/wiki/Video_codec" \t "https://atekihcan.github.io/CLRS/01/E01.02-01/_blank) to achieve video decoding and some for [recommending new content](https://help.netflix.com/en/node/100639" \t "https://atekihcan.github.io/CLRS/01/E01.02-01/_blank).

# ***Any Game***

Applies [clipping algorithm](https://en.wikipedia.org/wiki/Clipping_(computer_graphics)" \t "https://atekihcan.github.io/CLRS/01/E01.02-01/_blank) to discard objects that are outside the viewport.

**1.2-2:-**Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in 8n^2 steps, while merge sort runs in 64n lg n steps. For which values of n does insertion sort beat merge sort?

To find the values of n for which insertion sort beats merge sort, we need to determine when the time complexity of insertion sort is less than that of merge sort. Given:

**Insertion Sort:** 8n2

**Merge Sort:** 64nlgn

We are looking for values of nnn such that:

**8n2<64nlgn**

Dividing both sides by 8n (assuming n>0), we get:

**n<8lgn**

Now we need to solve this inequality, n<8lgn, to find the values of n for which it holds.

# ***Approach***

We can approximate the values of n for which n<8lgn by testing different values of n and checking the inequality. However, let's also approach it theoretically to get a rough sense of where the inequality might hold:

* **For small values of n** (like 1 through 10), we can check explicitly.
* **For larger values of n**, since nnn grows faster than lgn, there will be a point where n exceeds 8lgn, so we’ll search for that threshold.

Let's calculate and verify a few values of nnn to pinpoint this threshold.

The inequality n<8lgn holds for values of n up to 43. Therefore, insertion sort outperforms merge sort for input sizes n≤43 on this machine. For n>43, merge sort becomes more efficient

# ***Python Code***

# 

# **1.2-3:-**What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2^n on the same machine?

To find the smallest value of \( n \) such that an algorithm with running time 100n^2 runs faster than an algorithm with running time 2^n , we need to find the smallest integer n for which:

**100n2<2n**

Let's break it down into simple steps:

# ***Step 1: Set Up the Inequality***

We are given:

100n^2 < 2^n

# ***Step 2: Check Small Values of ( n )***

Since 2^n grows exponentially, it will eventually surpass 100n^2 , which grows quadratically. To find when this happens, we can substitute small integer values for n and calculate each side of the inequality until we find the smallest n that satisfies it.

**1. For n = 10 :**

- 100n^2 = 100 \times 10^2 = 10000

- 2^n = 2^{10} = 1024

- Result:10000 > 1024(does not satisfy the inequality)

**2. For n = 15 :**

- 100n^2 = 100 \times 15^2 = 22500

- 2^n = 2^{15} = 32768

- Result: 22500 < 32768 (satisfies the inequality)

**3.For n = 14:**

- 100n^2 = 100 \times 14^2 = 19600

- 2^n = 2^{14} = 16384

- Result: 19600 > 16384 (does not satisfy the inequality)

## ***Step 3: Conclusion***

The smallest value of n that satisfies 100n^2 < 2^n is:

n = 15

So, \*\*\( n = 15 \)\*\* is the answer.

# ***Python Code***

# 

**CHAPTER NO 02 :-**

**ANALYZING ALGORITHMS**

**EXERCISE NO 01 :-**

**2.1-1:-** Using Figure 2.2 as a model, illustrate the operation of Insertion-Sort Insertion-Sort  on the array A=⟨31,41,59,26,41,58⟩A=⟨31,41,59,26,41,58⟩.

To illustrate the operation of Insertion Sort on the array A=⟨31,41,59,26,41,58, let's go through the process step-by-step, focusing on how each element is inserted into its correct position.

The Insertion Sort algorithm works by iterating through the array from left to right, inserting each element into its appropriate position in the already sorted portion of the array.

# ***Step-by-Step Insertion Sort***

**Initial Array**: A=⟨31,41,59,26,41,58

**Iteration 1**: (Key = 41)

* 1. **Array**: ⟨31,41,59,26,41,58⟩
  2. 41 is already greater than 31, so it remains in place.

**Iteration 2**: (Key = 59)

* 1. **Array**: ⟨31,41,59,26,41,58⟩⟩
  2. 59 is greater than 41, so it remains in place.

**Iteration 3**: (Key = 26)

* 1. **Array before insertion**: ⟨31,41,59,26,41,58⟩
  2. 26 is less than 59, so we shift 59 to the right.
  3. 26 is also less than 41, so we shift 41 to the right.
  4. 26 is also less than 31, so we shift 31 to the right.
  5. **Array after insertion**: ⟨26,31,41,59,41,58⟩

**Iteration 4**: (Key = 41)

* 1. **Array before insertion**: ⟨26,31,41,59,41,58⟩
  2. 41 is less than 59, so we shift 59 to the right.
  3. **Array after insertion**: ⟨26,31,41,41,59,58⟩

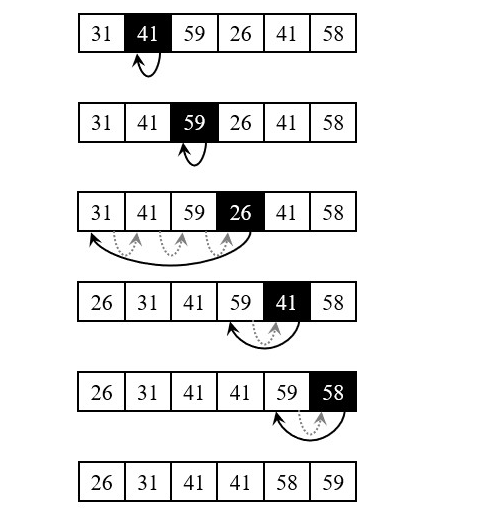
**Iteration 5**: (Key = 58)

* 1. **Array before insertion**: ⟨26,31,41,41,59,58⟩
  2. 58 is less than 59, so we shift 59 to the right.
  3. **Array after insertion**: ⟨26,31,41,41,58,59⟩

# ***Final Sorted Array***

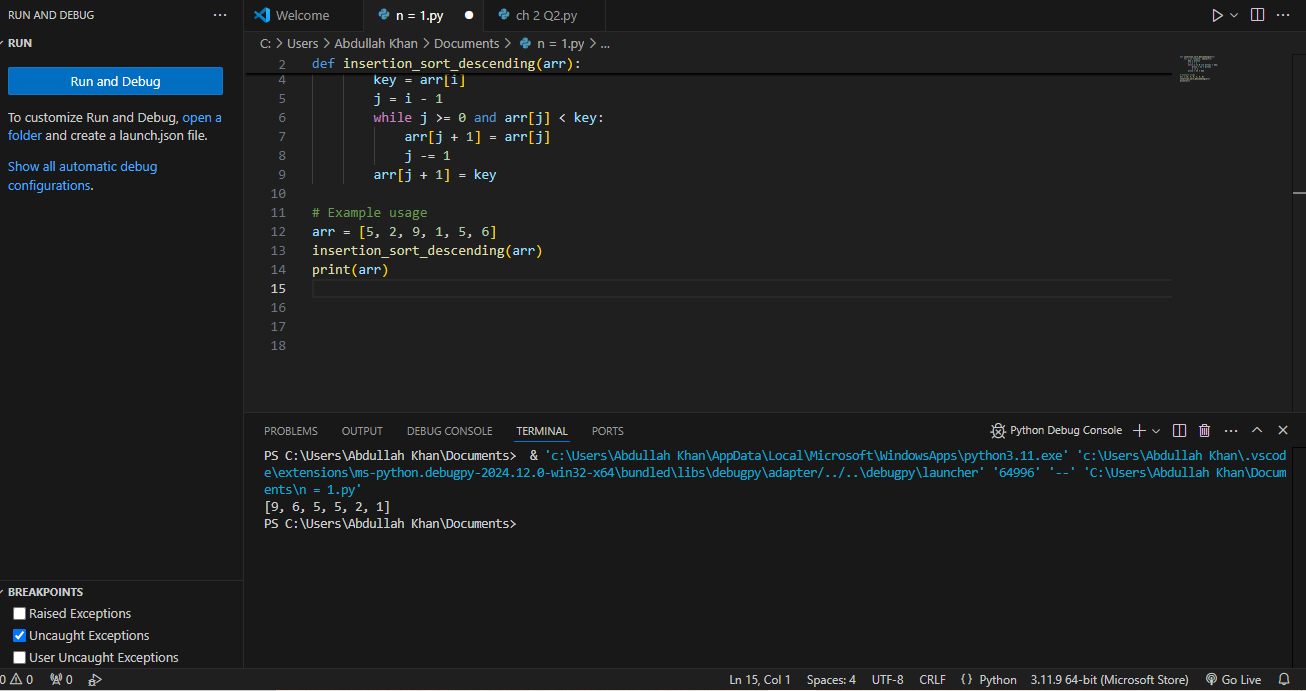
After all iterations, the array is sorted:  
A=⟨26,31,41,41,58,59⟩

Each step above illustrates the Insertion Sort operation as each element finds its place in the sorted portion of the array.



**2.1-2:-** Rewrite the Insertion-SortInsertion-Sort procedure to sort into non-increasing instead of non-decreasing order.

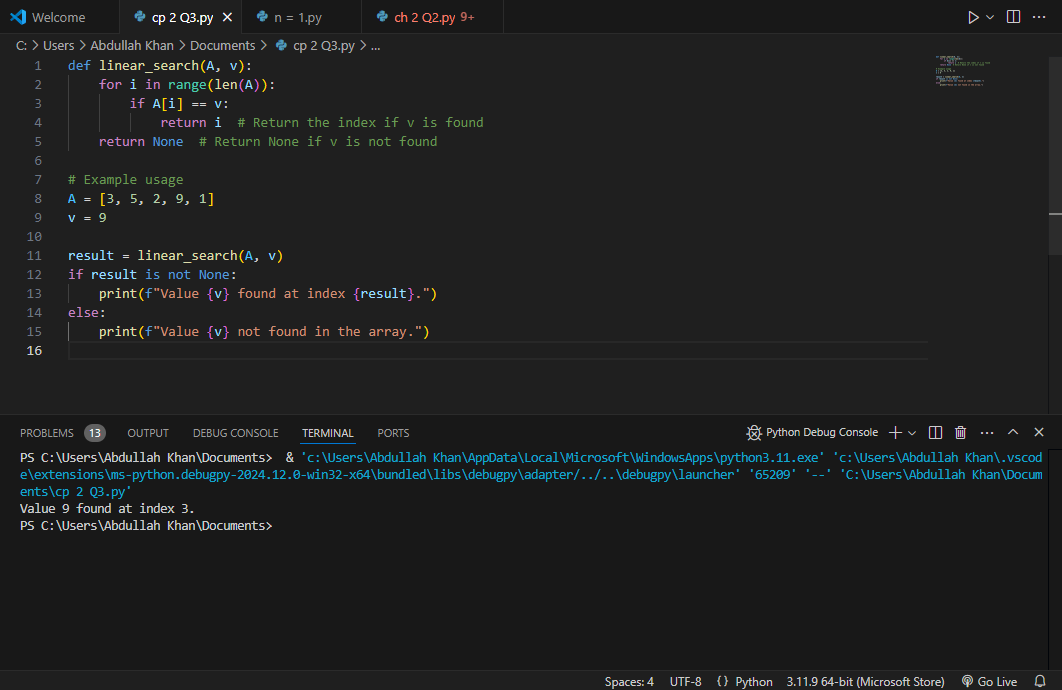
We just need to reverse the comparison of A[i]and key in line 5 as follows



**2.1-3:-** Consider the searching problem:

* Input: A sequence of n numbers A=⟨a1,a2,…,an and a value v.
* Output: An index i such that v=A[i] or the special value NIL if v does not appear in A.

Write pseudo-code for linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.



# ***Loop Invariant***

At the start of the each iteration of the for loop of lines 1-3, the subarray A[1..i−1] does not contain the value v.

And here is how the three necessary properties hold for the loop invariant:

***Initialization:*** Initially the subarray is empty. So, none of its’ elements are equal to v.

***Maintenance:*** In ii-th iteration, we check whether A[i] is equal to v or not. If yes, we terminate the loop or we continue the iteration. So, if the subarray A[1..i−1] did not contain v before the I -the iteration, the subarray A[1..i] will not contain v before the next iteration (unless i theiteration terminates the loop).

***Termination:*** The loop terminates in either of the following cases,

* We have reached index ii such that vv = A[i], or
* We reached the end of the array, i.e. we did not find v in the array A. So, we return NIL.

In either case, our algorithm does exactly what was required, which means the algorithm is correct.

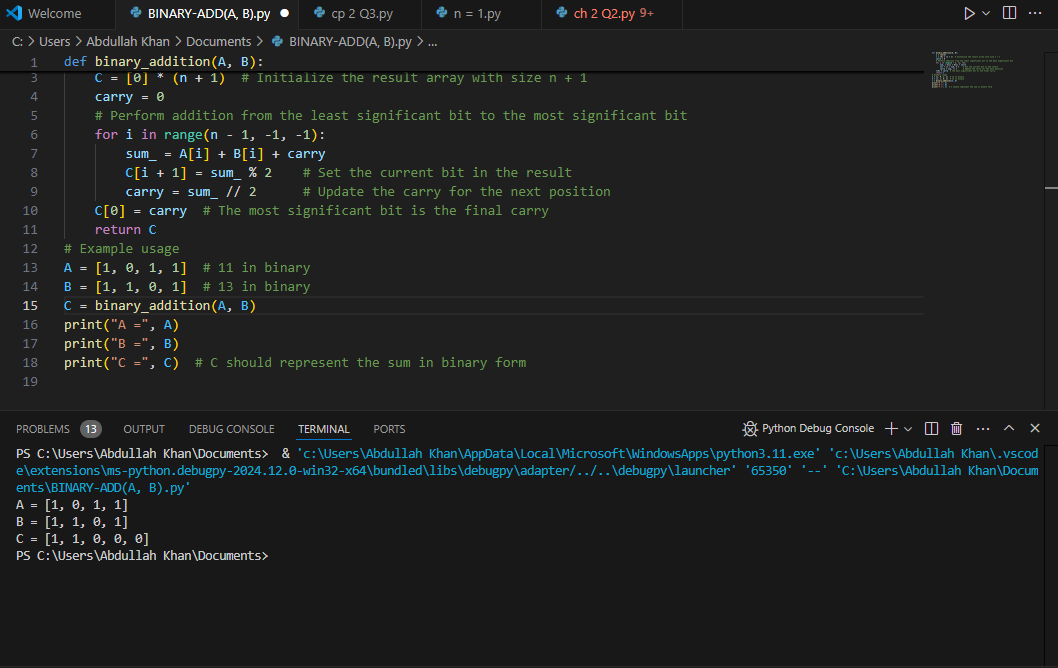
**2.1-4:-**Consider the problem of adding two n-bit binary integers, stored in two nn element arrays AA and BB. The sum of the two integers should be stored in binary form in an (n+1) element array C. State the problem formally and write pseudocode for adding the two integers.

### Problem Statement

Given:

* Two n-bit binary integers represented as arrays A and B, each of length n. The most significant bit is at index 0, and the least significant bit is at index n−1 .
* The objective is to calculate their sum as a binary integer and store it in an (n+1) element array C, which will contain the resulting sum in binary form.

**Formally:**

* **Input**: Arrays A=[a0,a1,…,an−1]and B=[b0,b1,…,bn−1]representing two n-bit binary integers.
* **Output**: Array C=[c0,c1,…,cn]representing the binary sum of A and B, with c0c\_0c0​ as the most significant bit.
* 

· Initialize an array C of size n+1 to store the sum.

· Start from the least significant bit (rightmost bit) and move left.

· For each position i:

* Calculate the sum of A[i], B[i], and the carry from the previous position.
* Store the binary result of this sum in C[i+1] (the i+1 position in C).
* Update the carry based on the integer division of the sum by 2.

· After the loop, the carry (if any) is placed at the most significant bit, C[0].

· The resulting array C represents the binary sum.

**EXERCISE NO 2:-**

**2.2-1:-**Express the function n3/1000−100n2−100n+3 in terms of Θ notation.

# ***Function***

**f(n)=1000n3​−100n2−100n+3**

in terms of Θ notation, we need to identify the term with the highest growth rate as n→∞n , since Θ notation describes the asymptotic behavior.

***-Identify the Dominant Term:*** The terms in f(n) are:

* **n^3/1000**
* **−100n^2**
* **−100n-**
* **+3**

Among n^3/1000 these is the term with the highest degree n^3, so it will dominate the function's growth rate as n becomes large.

***Express in Θ Notation:*** Since n^3/1000​ is the leading term and constant factors are ignored in Θ notation, we have:

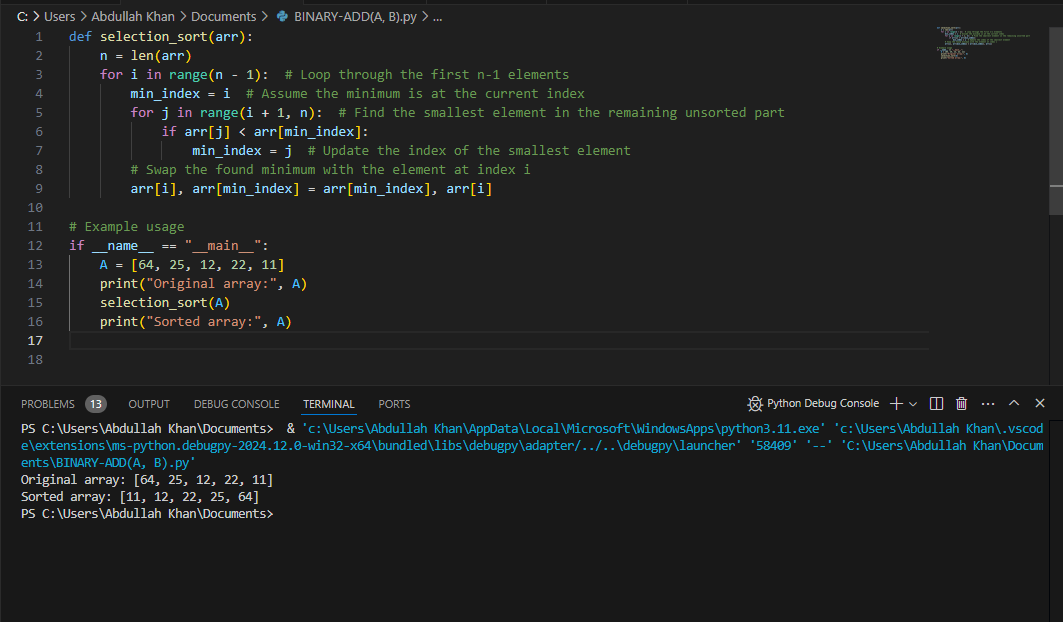
**f(n)=Θ(n^3)**

## ***Final Answer:***

**f(n)=Θ(n^3)**

This means f(n) grows asymptotically at the same rate as n^3 as n→infinity

**2.2-2:-**Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A and exchange it with A[2]. 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 best-case and worst-case running times of selection sort in Θ-notation.



# ***Loop Invariant***

**What is it?**  
At the start of each iteration (for index i):

* The subarray A[0…i−1] is sorted and contains the smallest iii elements of the original array.

### Why Only n−1 Elements?

* The algorithm only needs to run for the first n−1elements because once the first n−1 smallest elements are in place, the last element is automatically in its correct position (it's the only one left).

### Running Times

* **Best Case**: Θ(n^2)
* **Worst Case**: Θ(n^2)

### Explanation of Running Times

* Selection Sort always checks every element in the unsorted part of the array, regardless of the initial order. Therefore, it takes about n^2/2 comparisons:
  + For the first element, it checks n−1 elements.
  + For the second, it checks n−2, and so on.

So both the best-case and worst-case running times are Θ(n^2)

# ***Conclusion***

* **Selection Sort** finds the smallest element and places it correctly.
* **Loop Invariant** ensures sorted elements at each step.
* **Only** n−1 iterations are needed.
* **Both best and worst-case** running times are Θ(n^2)\).

**2.2-3:-**Consider linear search again (see [Exercise 2.1-3](https://atekihcan.github.io/CLRS/02/E02.01-03/)). How many elements of the input sequence 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? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

# ***Linear Search Overview***

In linear search, we look for an element v in an array A by checking each element one by one from the start to the end of the array.

## ***Worst Case***

* **Definition**: The worst case occurs when:
  + The element v is not in the array, or
  + The element v is the last element in the array.
* **Elements Checked**: In both scenarios, you need to check all n elements.
* **Running Time**: **Θ(n)**

## ***Average Case***

**Definition**: The average case assumes that v is equally likely to be at any position in the array.

**Elements Checked**:

* If v is found, on average, you'll check about half of the elements, which is n/2.

**Running Time:Θ(n)**

Even though you check about half of the elements on average, the time complexity is still linear, as it grows proportionally with the size of the array.

# ***Conclusion***

* **Worst-case**: Θ(n) (check all elements)
* **Average-case**: Θ(n) (check about half, but still linear)

Both the worst-case and average-case running times for linear search are linear because they grow with the size of the input array.

**2.2-4:-**How can we modify almost any algorithm to have a good best-case running time?

# ***Short-Circuit Evaluation***

For algorithms that involve conditional checks (like searching or sorting), modify the code to return early when a favorable condition is met. For example, in a search algorithm, if you find the target at the first position, you can immediately return the result without checking the rest.

# ***2. Use of Flags***

In algorithms where you are processing elements based on conditions, use a flag to indicate when a particular condition has been met. If the flag is set, subsequent checks can be skipped.

# ***3. Optimized Data Structures***

Utilize data structures that allow for faster best-case operations:

* **Hash Tables**: For look ups, insertions, or deletions, hash tables can provide O(1) average-case complexity.
* **Balanced Trees**: For dynamic datasets where you need quick insertions and look-ups, using balanced trees can help maintain efficient performance.

# ***4. Preprocessing***

Sometimes prepossessing the input can lead to a significant improvement in best-case performance:

* **Sorting**: If you sort an array beforehand, some algorithms (like binary search) can achieve O(1) best-case time for look ups if the target is the first element.
* **Memorization**: Storing results of expensive function calls can allow for O(1) retrieval for previously computed results.

# ***5. Algorithm Design***

When designing the algorithm:

* **Early Exit Conditions**: Structure your algorithms to exit early if a simple case is detected. For example, in sorting, if the array is already sorted, you can immediately return.
* **Use of Best-Case Scenarios**: Some algorithms inherently have better best-case times based on specific input configurations. Adjusting the algorithm's design to handle these cases effectively can help.

# ***6. Adaptive Algorithms***

Use adaptive algorithms that can take advantage of the input's properties. For example, insertion sort has a best-case running time of O(n)) when the input is already sorted.

## ***Example: Linear Search***

For a linear search algorithm:

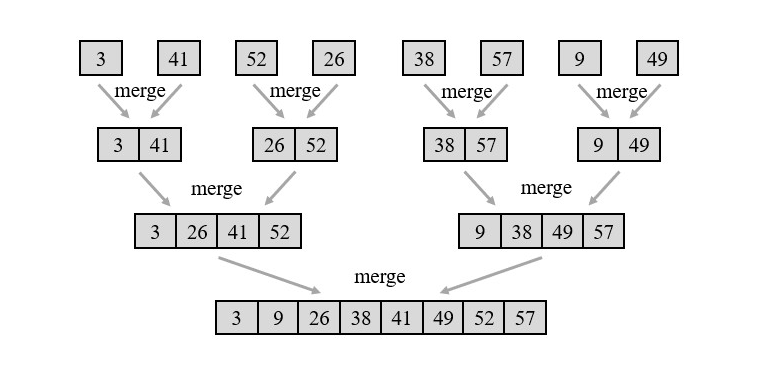
* **Best Case**: You can achieve a best-case time of O(1)if the target element is at the first position. This is done by checking the first element before entering any loops.

# ***Conclusion***

Improving the best-case running time often involves optimizing for specific scenarios or leveraging efficient data structures and algorithms. By implementing early exits, prepossessing, and using appropriate data structures, you can significantly enhance the best-case performance of many algorithms.

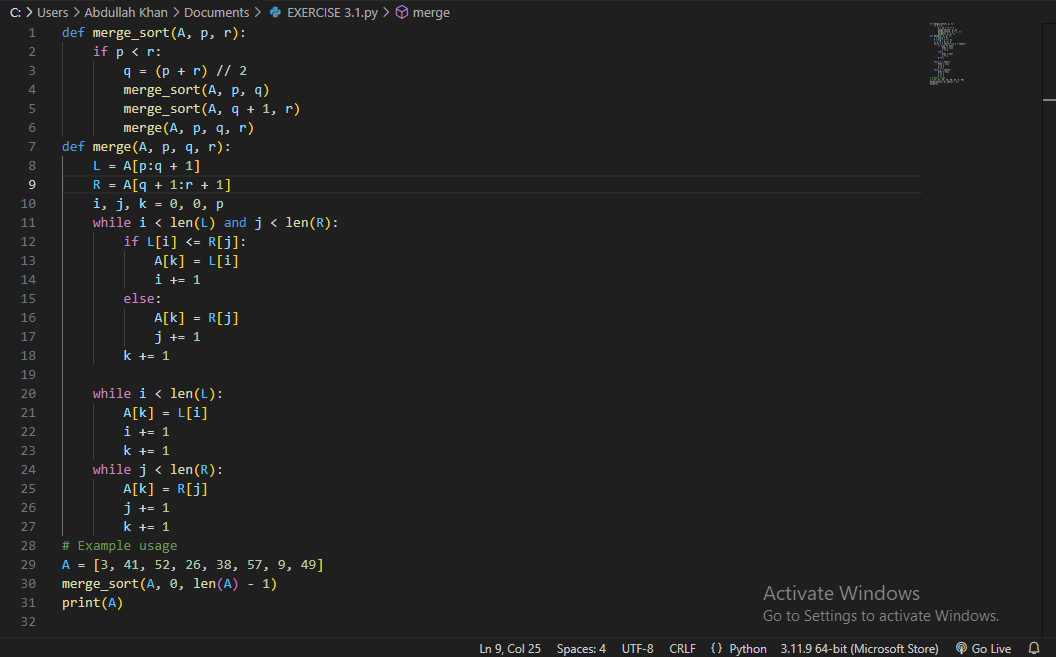
**EXERCISE NO 3:-**

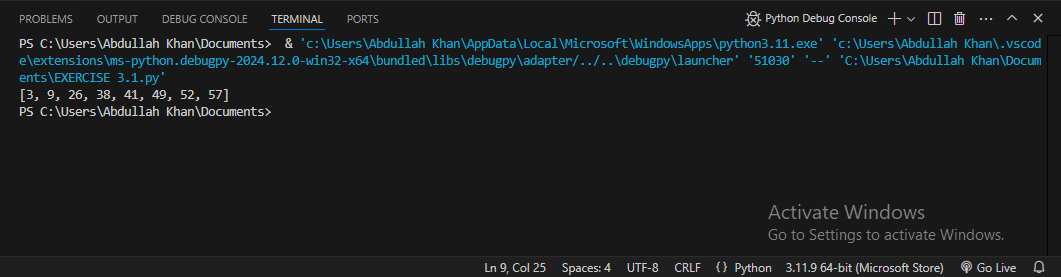
**2.3-1:-**Using Figure 2.4 as a model, illustrate the operation of merge sort on the array A=⟨3,41,52,26,38,57,9,49⟩



The figure is slightly different than the one in the book. The figure in the book was showing sorting progress from bottom to up. Here it is shown from top to bottom, which I feel will be more intuitive to understand.

**2.3-2-**Rewrite the Merge procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.





**2.3-3:** Referring back to the searching problem (see [Exercise 2.1-3](https://atekihcan.github.io/CLRS/02/E02.01-03/)), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The **binary search** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is Θ(lg*n*).

# ***Iterative Binary Search Pseudo-code***

# 

# ***Recursive Binary Search Pseudocode***

# 

# ***Worst-Case Time Complexity Analysis***

**Reduction by Half**: Each step of binary search reduces the problem size (i.e., the number of elements to consider) by half.

**Number of Steps**: Starting with nnn elements, binary search performs a maximum of log^2(n) comparisons to reduce the problem size to 1 (or 0 in case of failure). This is because the size reduces as follows:

**n,n/2,n/4,…,1**

**Θ(lg n) Complexity**: Thus, in the worst case, binary search requires Θ(log^n) comparisons to complete.

**2.3-4:-** Observe that the ****while**** loop of lines 5–7 of the Insertion-SortInsertion-Sort procedure in Section 2.1 uses a linear search to scan (backward) through the sorted sub-array A[1..j−1]]. Can we use a binary search (see [Exercise 2.3-5](https://atekihcan.github.io/CLRS/02/E02.03-05/)) instead to improve the overall worst-case running time of insertion sort to)Θ(*n*lg*n*)?

NO, using binary search in insertion sort does not improve the overall worst-case running time toVΘ(nlogn). Here’s why:

# ***How Binary Search Affects Insertion Sort***

**Binary Search for Positioning**: By using binary search, we can indeed find the correct position for the current element in log j time, where j is the current position in the array. This would reduce the search time within the sorted sub-array to Θ(loG j) rather than Θ(j) in the worst case.

***Insertion Step Remains Costly:*** However, insertion sort not only requires finding the correct position but also involves shifting elements to make space for the new element. Even after finding the insertion point, we must still shift elements to insert the element at the right position. In the worst case, this shifting requires Θ(j) time per element, regardless of how quickly we find the insertion position.

# ***Total Running Time with Binary Search***

****Search Cost**:** Using binary search reduces the search time in each step J to Θ(log J).

**Insertion Cost (Shifting Elements)**: Despite the faster search, the shifting operation still takes Θ(j)time in the worst case.

**Overall Time Complexity**: Summing across all insertions, we still get a total time complexity of:

**∑n​Θ(j)=Θ(2^n(n−1)​)=Θ(n^2)**

Thus, even with binary search, the insertion sort still requires Θ(n^2) time in the worst case due to the shifting of elements. Consequently, binary search does not reduce insertion sort’s worst-case complexity to Θ(nlogn).

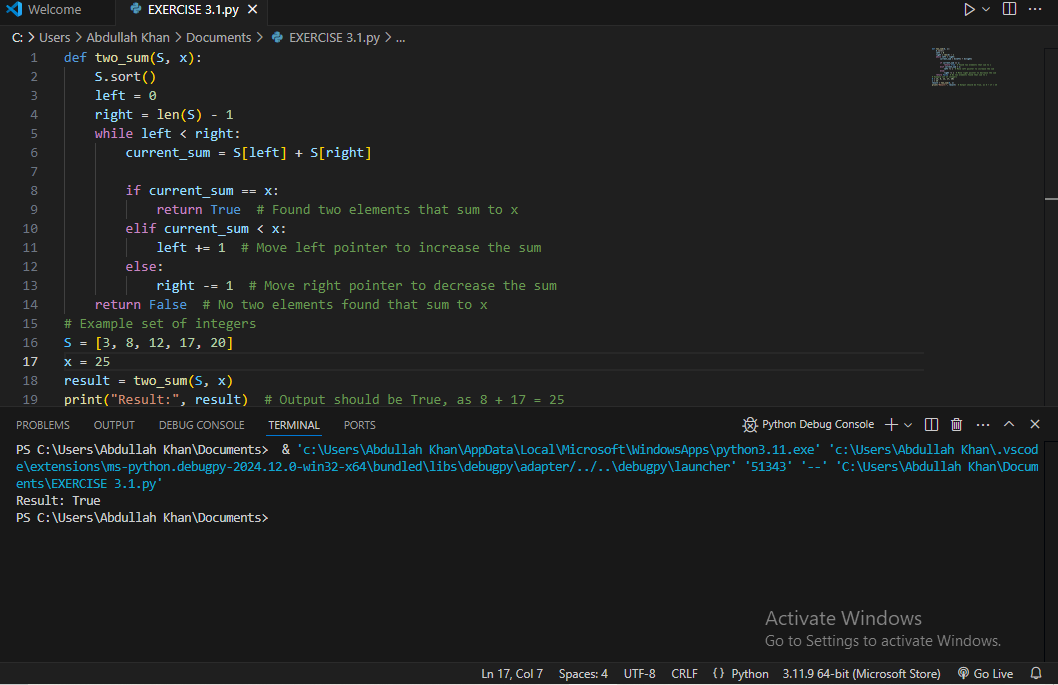
**2.3-5:-**Describe a Θ(n log n)-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

# ***Algorithm***

* ***Sort the Array:*** Start by sorting the array S in Θ(nlog n time.
* ***Initialize Pointers:*** Use two pointers, one (left) starting at the beginning of the sorted array and the other (right) at the end of the sorted array.

## ***Two-Pointer Search:***

* 1. While left < right:
  2. Calculate the sum of the elements at the two pointers, S[left] + S[right].
  3. If this sum is equal to x, return True, as you’ve found two elements that sum to x.
  4. If the sum is less than x, increment the left pointer (to increase the sum).
  5. If the sum is greater than x, decrement the right pointer (to decrease the sum).
  6. If the loop exits without finding such a pair, return False.



**CHAPTER NO 03:-**

**STANDARD NOTATION AND COMMON FUNCTONS**

**EXERCISE NO 01:-**

**3.1-1:-**Let f(n) and g(n) be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that max (f(n),g(n)).

To prove that max(f(n),g(n))=Θ(f(n)+g(n))), we will use the basic definition of Θ notation.

# ***Recall the Definition of Θ notation***

A function h(n)=Θ(p(n))h(n)) if there exist positive constants c1,c2 and n0​ such that:

**c1​⋅p(n) < h(n) < c2​⋅p(n)**

**for all n≥n0.**

## ***Goal***

We want to show:

**max(f(n),g(n))=Θ(f(n)+g(n)).**

This means we need to find constants c1, c2, and n0 such that:

**c1​⋅(f(n)+g(n))≤max(f(n),g(n))≤c2​⋅(f(n)+g(n))**

for n≥n0

## ***Step 1: Upper Bound***

Since max(f(n), g(n)) is the larger of the two functions f(n) and )g(n), it is always true that:

**max(f(n),g(n))≤f(n)+g(n).**

Therefore, we can choose c2=1, which gives us:

**max(f(n),g(n))≤1⋅(f(n)+g(n))**.

## ***Step 2: Lower Bound***

Now, we need to find a constant c1c\_1c1​ such that:

**c1​⋅(f(n)+g(n))≤max(f(n),g(n)).**

Observe that:

**f(n)+g(n)≤2⋅max(f(n),g(n)).**

This inequality holds because either f(n)≤g(n) or g(n)≤f(n))so the sum f(n)+g(n) is at most twice the maximum of the two functions.

Thus, we can choose c1=1/2, giving us:

**1/2⋅(f(n)+g(n))≤max(f(n),g(n)).**

# ***Conclusion***

Combining the upper and lower bounds, we have:

**1/2 ⋅(f(n)+g(n))≤max(f(n),g(n))≤1⋅(f(n)+g(n))**

for all n≥n0,

Therefore,

**max(f(n),g(n))=Θ(f(n)+g(n)),**

which completes the proof.

**3.1-2:-**Show that for any real constants a and b, where b>0,(*n*+*a*)*b*=Θ(*nb*)?

Any real constants a andbb, where b>0, the expression (n+a)b=Θ(n ), we will use the definition of Θ-notation.

# ***Goal***

We want to show:

**(n+a) ^b=Θ(n^b)**.

This means we need to find constants c1​, c2​, and n0 such that:

# **c1​⋅n^b ≤ (n+a)^b≤c2​⋅n^b**

for n≥n0.

# ***Step 1: Upper Bound***

We can expand (n+a)^ b using the binomial theorem:

**(n+a)^ b = n^b+(b/1​)n^b−1 a+(b/2)n^b−2 a^2+⋯+a^b**

For large values of n, the term n^b dominates, as each subsequent term is of lower order in n.

Therefore, for large n, we have:

**(n+a^)b≤2⋅n^b**

hus, we can choose c^2 = 2, giving us:

**(n+a)^b≤2⋅n^b**

## ***Step 2: Lower Bound***

For the lower bound, note that since n is large, n+a is close to nnn. Therefore, we have:

**n≤n+a≤2n**

for n≥n0 (where n0 is sufficiently large to make aaa relatively small compared to n).

Raising each part to the power b, we get:

**n^b≤(n+a)^b≤(2n)^b=2^b⋅n^b**

This inequality implies that:

**(n+a)^b≥n^b.**

Thus, we can choose c1=1 giving us:

**n^b≤(n+a)^b.**

# ***Conclusion***

Combining the upper and lower bounds, we have:

**n^b ≤ (n+a)^b ≤ 2^b⋅n^b,**

for all n≥n0,

Therefore,

**(n+a)^b=Θ(n^b),**

which completes the proof

**3.1-3:-**Explain why the statement, “The running time of algorithm A is at least O(n^2),” is meaningless.

The statement, “The running time of algorithm A is at least O(n^2)),” is indeed meaningless due to a fundamental misunderstanding of Big-O notation. Here’s why:

# ***Understanding Big-O Notation***

Big-O notation, O(f(n)), provides an upper bound on the growth rate of a function. It describes the worst-case scenario for an algorithm's running time, up to a constant factor. Specifically, saying that an algorithm is O(n^2) means that its running time grows at most as fast as n^2, but it could be faster.

# ***Why “At Least O(n^2)” is Meaningless***

When we say an algorithm's running time is "at least" some function, we are generally looking for a lower bound—something that expresses the minimum growth rate of the running time. Big-O notation does not provide a lower bound; it only provides an upper bound.

The correct notation for a lower bound would be **Omega notation** (n^2). Saying “the running time of algorithm A is Ω(n^2)” would mean that the running time grows at least as fast as n^2, which makes logical sense.

# ***Corrected Statement***

If we want to express that the algorithm's running time is at least proportional ton^2, the statement should be:

“The running time of algorithmA is at least Ω(n^2).”

This properly communicates that the algorithm requires at least n^2 time in the asymptotic sense.

**3.1-4:-**Is 2^n+1=O(2^n)? Is 2^2n=O(2^n) ?

# ***1. Is 2^n + 1 = O(2^n)?***

Yes,2^n + 1 = O(2^n)

## ***Explanation:***

Big-O notation provides an upper bound, meaning that for a function f(n)=O(g(n))f(n)grows at most as fast as g(n) for large n.

In this case, as n becomes large, the term 2^n dominates 2^n + 1 because +1becomes insignificant compared to2^n.Formally:

**2^n + 1 ≤2⋅2n**

for all n≥0.

Thus, by choosing a constant c=2, we have:

**2^n + 1≤c⋅2n**

for large n, so 2^n + 1 = O(2^n).

# ***2. Is 2^{2n} = O(2^n)?***

No, 2^2n≠O(2^n).

#### Explanation:

Consider the expression 2^{2n} in terms of\2^n:

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

This means that 2^{2n}grows exponentially faster than 2^n because it is the square of 2^n, not bounded by a constant multiple of 2^n.

For large n, 2^{2n} far exceeds any constant multiple of 2^n, so there are no constants c and n\_0 such that:

**2^{2n} ≤c.2n**

Therefore, 2^2n≠O(2^n).

**3.1.5:-**Prove Theorem 3.1.

Theorem 3.1 says:

For any two functions f(n and g(n), we have f(n)=Θ(g(n)) if and only if f(n)=O(g(n))and f(n)=Ω(g(n)).

To prove this theorem, we need to show the logic holds both ways, i.e.

**f(n)=Θ(g(n))⟹f(n)=O(g(n))andf(n)=Ω(g(n)) eq(1)**

and

**f(n)=O(g(n))andf(n)=Ω(g(n))⟹f(n)=Θ(g(n)) eq(2)**

# ***Part 1***

If f(n)=Θ(g(n)), then for n≥n\_0,

**0≤c1g(n)≤f(n)≤c2g(n)**

As**0≤f(n)≤c2g(n) for n≥n\_0*=*​, f(n)=O(g(n))).**

As **0≤c1g(n)≤f(n) for n≥n\_0, f(n)=Ω(g(n)).**

# ***Part 2***

If f(n)=Ω(g(n)), then for n≥n\_1,

**0≤c1g(n)≤f(n)**

If f(n)=O(g(n)), then for n≥n\_2,

**0≤f(n)≤c2g(n)**

Combining the above two inequalities, we can say for n≥max(n1,n2),

**0≤c\_1g(n)≤f(n)≤c\_2g(n)**

In other words,**f(n)=Θ(g(n)).**

**3.1.6:-**Prove that the running time of an algorithm is Θ(g(n)) if and only if its worst-case running time is O(g(n)) and its best-case running time is Ω(g(n)).

To prove that the running time of an algorithm is Θ(g(n)) if and only if its worst-case running time is O(g(n) and its best-case running time is Ω(g(n))), we will need to show two directions:

1. **If** the running time of the algorithm is Θ(g(n), then its worst-case running time is O(g(n)) and its best-case running time is Ω(g(n)).
2. **If** the worst-case running time of the algorithm is O(g(n)) and the best-case running time is Ω(g(n)), then the running time of the algorithm is Θ(g(n)).

# ***Direction 1: If the Running Time is Θ(g(n)), then the Worst-case is O(g(n))and the Best-case is Ω(g(n))***

Assume that the running time of the algorithm is Θ(g(n)). By definition, this means there exist positive constants c\_1, c\_2​, and n\_0 such that:

**c1⋅g(n)≤T(n)≤c2⋅g(n)**

for all n≥n\_0, where T(nrepresents the running time of the algorithm.

**Worst-case Condition**: Since T(n)≤c2⋅g(n) for all n≥n0, this implies that the worst-case running time T\_worst(n) satisfies:

**T\_worst​(n)≤c2​⋅g(n),**

so T\_worst(n)=O(g(n)).

**Best-case Condition**: Similarly, since T(n)≥c1⋅g(n) for all n≥n\_0​, this implies that the best-case running time T\_best ( n) satisfies:

**T\_best(n)≥c1⋅**

so T\_best(n)=Ω(g(n)).

Thus, if the running time of the algorithm is Θ(g(n)), then the worst-case running time is O(g(n)) and the best-case running time is Ω(g(n)).

# ***Direction 2: If the Worst-case is O(g(n)) and the Best-case is Ω(g(n)), then the Running Time is Θ(g(n))***

Now assume that the worst-case running time of the algorithm is O(g(n))and the best-case running time is Ω(g(n)). This means there exist positive constants c\_1, c\_2​, and n\_0 such that:

**T\_best(n)≥c1⋅g(n) and T\_worst​(n)≤c2​⋅g(n)**

for all n≥n\_0.

Since Tbest(n)≤T(n)≤Tworst(n), it follows that:

**c1⋅g(n)≤T(n)≤c2⋅g(n)**

for all n≥n\_0.

This matches the definition of Θ(g(n)).

# ***Conclusion***

We have shown both directions:

* If the running time is Θ(g(n)), then the worst-case running time is O(g(n)) and the best-case running time is Ω(g(n)).
* If the worst-case running time is O(g(n)) and the best-case running time is Ω(g(n))), then the running time is Θ(g(n)).

Therefore, the running time of an algorithm is Θ(g(n)) **if and only if** its worst-case running time is O(g(n))and its best-case running time is Ω(g(n)).

**3.1-7:-**Prove that o(g(n))∩ω(g(n)) is the empty set.

To prove that o(g(n))∩ω(g(n)) is the empty set, we need to understand what it means for a function to belong to o(g(n))and to ω(g(n)).

# ***Definitions***

**Little-o Notation (**o(g(n))**:** A function f(nis in o(g(n)) if f(n) grows strictly slower than g(n) as n→∞. Formally, f(n)=o(g(n)) if:

**Lim n→∞​f(n)/g(n)​=0.**

**Little-omega Notation (**ω(g(n)**):** A function f(n) is in ω(g(n)) if f(n) grows strictly faster than g(n) as n→∞n . Formally, f(n)=ω(g(n)) if:

**limn→∞​f(n)/g(n)​=∞.**

# ***Goal***

We want to show that no function f(n)can simultaneously be in both o(g(n) and ω(g(n)). In other words, we want to show that:

o(g(n))∩ω(g(n))=∅

# ***Proof***

Assume, for the sake of contradiction, that there exists a function f(n) such that f(n)∈o(g(n)) and f(n)∈ω(g(n))

Since f(n)∈o(g(n)), by definition, we have:

**Lim n→∞​f(n)/g(n)​=0.**

Since f(n)∈ω(g(n), by definition, we also have:

**limn→∞​f(n)/g(n)​=∞.**

This leads to a contradiction because f(n)g(n) cannot approach both 0 and ∞ as n→∞n . A function cannot grow strictly slower than g(n (approaching 0 relative tog(n)) and strictly faster than g(n) (approaching ∞ relative to g(n)) at the same time.

# ***Conclusion***

Since no function f(n) can satisfy both f(n)=o(g(n)) and f(n)=ω(g(n)) simultaneously, we conclude that:

**o(g(n))∩ω(g(n))=∅**

This completes the proof.

**3.1-8:-**We can extend our notation to the case of two parameters n and m that can go to infinity 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) : there exist positive constants c,n0, and m0 such that 0≤f(n,m)≤cg(n,m) for all n≥n0 or m≥m0}

Give corresponding definitions for Ω(g(n,m))and Θ(g(n,m))

* Given the two-parameter function g(n,m), we can define Ω(g(n,m)) and Θ(g(n,m)) similarly to O(g(n,m)), which provides bounds on functions of two variables. Here are the definitions:

# **Definition of Ω(g(n,m))**

The set Ω(g(n,m)) is defined as:

Ω(g(n,m))={f(n,m):there exist positive constants c,n0, and m0 such that 0≤c⋅g(n,m)≤f(n,m) for all n≥n0 or m≥m0

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

# ***Definition of Θ(g(n,m))***

The set Θ(g(n,m)) is defined as:

Θ(g(n,m))={f(n,m):there exist positive constants c1,c2,n0, and m0 such that 0≤c1⋅g(n,m)≤f(n,m)≤c2⋅g(n,m) for all n≥n0 or m≥m0}

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

# ***CPNCLUSION***

* O(g(n,m)) provides an upper bound on f(n,m).
* Ω(g(n,m)) provides a lower bound on f(n,m).
* Θ(g(n,m)) provides an asymptotically tight bound on f(n,m) meaning f(n,m) grows at the same rate as g(n,m) within constant factors for large n or m.