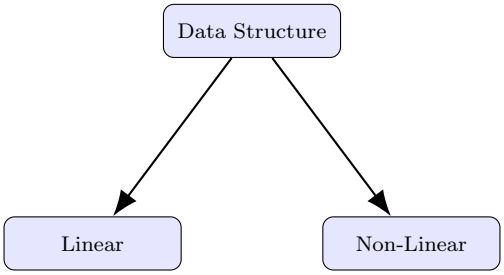


Linear

1. Array
2. Tuple
3. Linked List
4. Stack
5. Queue
6. Dictionary

Non-Linear

1. Tree
2. Graph
3. Heap
4. Trie
5. Hash Table

Algorithms

An algorithm is a sequence of finite steps to solve any particular problem. In computer science, algorithms are essential for performing tasks, processing data, and solving complex problems efficiently. They provide a clear set of instructions that a computer can follow to achieve a specific goal.

Why Algorithms Matter:

Algorithms are the backbone of computer programming and software development. They allow us to automate tasks, process large amounts of data, and solve problems that would be impossible to tackle manually. Understanding algorithms is crucial for anyone interested in computer science, as they form the foundation of efficient and effective programming.

Example of a Simple Algorithm:

Multiplication of Two Numbers

1. **Take two numbers a and b.**
2. **Multiply the two numbers to get the result c.**

$$c = a \times b \tag{1}$$

3. **Return the result c.**

1 Properties of Algorithms

An algorithm must satisfy the following properties:

1. **Output:** It should produce at least one output.
2. **Termination:** It should terminate after some point of time.
3. **Deterministic:** The output should be the same every time it is run. The output should be unambiguous, meaning that at every point of time, similar results should be achievable.
4. **Finite Steps:** It should have a finite number of steps.
5. **Language Independence:** It should be independent of any programming language.

Not an Algorithm

Not an algorithm →

Infinite Loop →

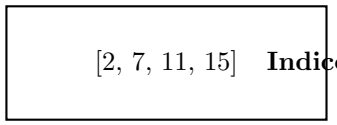
```
While(True):
print("DSA course")
```

Explanation: The above code snippet is not considered an algorithm because it creates an infinite loop. An algorithm must terminate after some point of time to produce an output.

Steps Required to Construct an Algorithm

1. **Problem Definition**

   - Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.
   - You may assume that each input would have exactly one solution, and you may not use the same element twice.
   - You can return the answer in any order.
   - Example 1:
     - **Input:** `nums = [2,7,11,15], target = 9`
     - **Output:** `[0,1]`
     - **Explanation:** Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.
   - Example 2:
     - **Input:** `nums = [3,2,4], target = 6`
     - **Output:** `[1,2]`
   - Example 3:
     - **Input:** `nums = [3,3], target = 6`
     - **Output:** `[0,1]`



2. **Design an Algorithm**

   - Develop the logic to solve the problem.
   - Methods:
     - Divide and Conquer (D&C)
     - Dynamic Programming (DP)
     - Recursion
     - Greedy
     - Many more...
   - See which algo is best for the problem statement
   - Example Logic: Iterate through the list and check pairs of numbers to see if they add up to the target.

3. **Test the Algorithm**

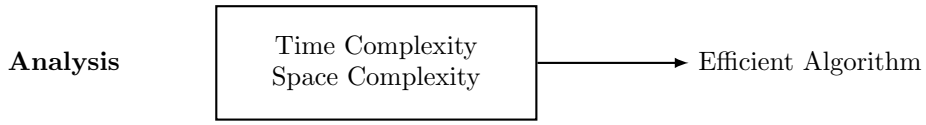
   - Verify the algorithm with available test cases.
   - Ensure that the algorithm works correctly and efficiently.

4. **Implement the Algorithm**

   - First do it in paper-pen and code when its confirmed
   - Write the actual code in the chosen programming language.
   - Make sure the implementation is clean and well-documented.

5. **Analyze the Algorithm**

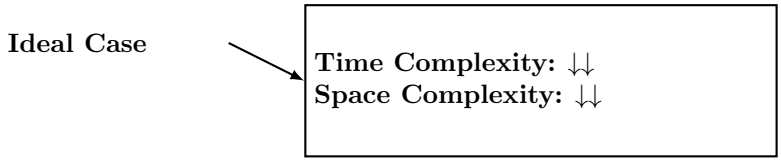
   - Evaluate the time and space complexity.
   - Ensure the algorithm is efficient and optimal.



Analysis of Algorithm Efficiency

1. **Ideal Case**

   - In the ideal case, an algorithm has:
     - **Time Complexity:** Very low (or reduced).
     - **Space Complexity:** Very low (or reduced).
   - This represents an efficient algorithm.



2. **Efficient Case**

   - In real-time projects, achieving the ideal case is often not possible.
   - Instead, a trade-off between time and space complexity is necessary.

Time and Space Trade-off

Question: Why do we often focus more on reducing the time complexity of algorithms rather than the space complexity?

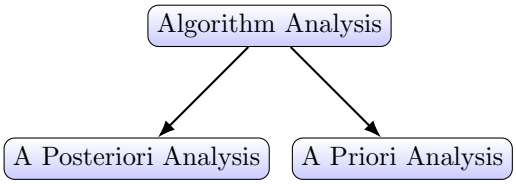
Answer: In algorithm design, there's often a trade-off between time and space complexity. If we have a lack of space, then we can compensate with time by using algorithms that are less space-efficient but run slower. However, in 2024, we have an abundance of storage space. Most users can't wait for longer processing times as they have limited attention spans and expect quick results. Therefore, we primarily focus on reducing the time complexity of a project rather than the space complexity.

Additionally, in real-world applications, the cost of storage has significantly decreased, making it more feasible to use more space to achieve faster processing times. This is particularly important in user-facing applications where responsiveness is critical to user satisfaction and retention. Furthermore, many modern applications operate in environments where computational resources are abundant, but user attention is scarce, reinforcing the emphasis on time efficiency.

Algorithm Analysis

Algorithm analysis is crucial for understanding the efficiency and effectiveness of algorithms. It helps in determining the resources required by an algorithm to solve a problem. There are two main types of algorithm analysis:

- **A Posteriori Analysis:** This type of analysis is performed after the algorithm has been implemented and run. It is dependent on the hardware and provides exact measurements.
- **A Priori Analysis:** This type of analysis is performed before the algorithm is implemented. It is independent of hardware and focuses on the logical structure and quality of the algorithm.



Definitions

A Posteriori Analysis: This analysis is performed after an algorithm is executed on a computer. It measures the actual performance, such as the execution

time and the amount of memory used. It is highly dependent on the specific hardware and software environment in which the algorithm runs.

A Priori Analysis: This analysis is theoretical and is done before running the algorithm. It focuses on the logical reasoning and mathematical proofs to estimate the time and space complexities. It is independent of the hardware and aims to predict the performance in a general sense.

Comparison of A Posteriori and A Priori Analysis

Aspect	**A Posteriori Analysis**	**A Priori Analysis**
Timing	After implementation and execution	Before implementation
Dependency	Dependent on hardware	Independent of hardware
Measurements	Provides exact measurements	Provides theoretical estimates
Focus	Actual performance	Logical structure and quality
Examples	Execution time, memory usage	Time complexity, space complexity

Understanding Time Complexity with Examples

Example 1

Consider the following function:

```
main() {
    x = y + z
}
```

- Here, $y = 3$ and $z = 2$.
- The code ‘x = y + z‘ will run exactly once.
- It is independent of any value of n.
- Therefore, the time complexity is constant, denoted as $O(1)$.
- This represents the worst-case scenario.

Real-life Analogy

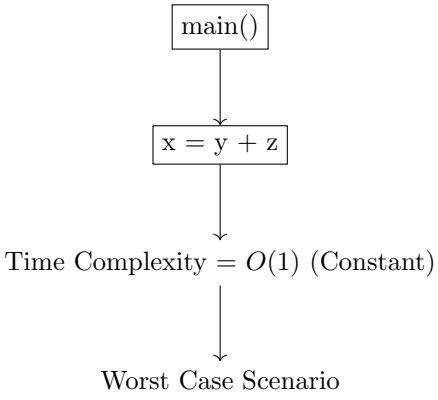
To judge a good friend:

- A good friend stands by you in good times.
- But a true friend is there for you in your worst situations.

Similarly:

- If our code performs well in the worst-case scenario, it is considered efficient.

- For a SaaS platform, the efficiency can't be judged with fewer users.
- However, if it handles a large number of users efficiently, it is indeed an efficient code.

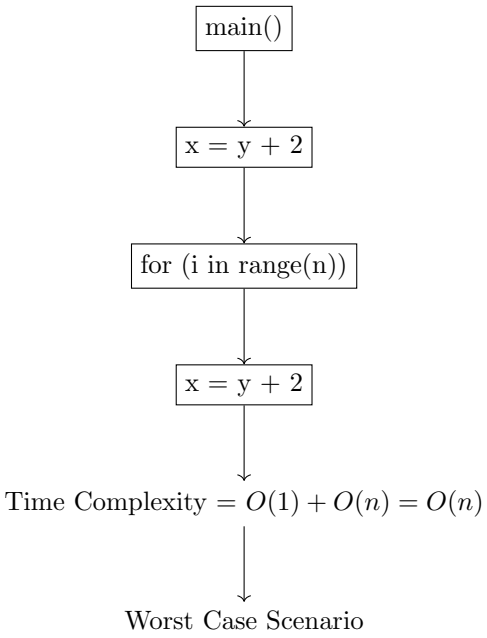


Example 2

Consider the following function:

```
main() {
    x = y + 2
    for (i in range(n)) {
        x = y + 2
    }
}
```

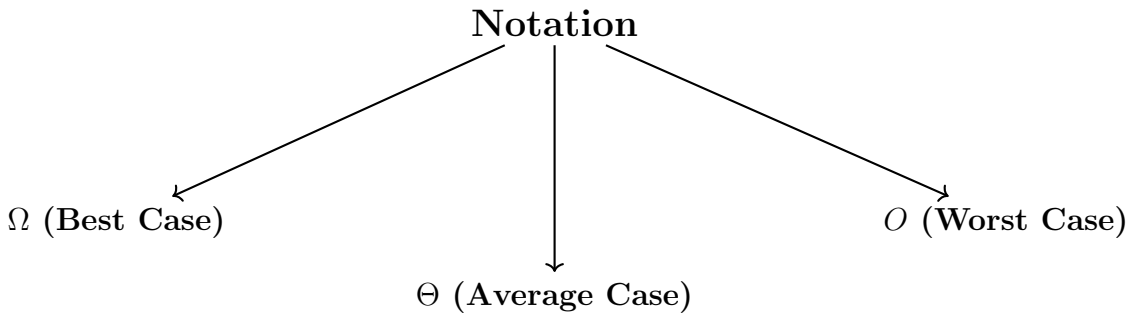
- The line 'x = y + 2' outside the loop runs once.
- The code inside the loop runs n times.
- For $n = 4$:
  - $i = 0$: x = y + 2
  - $i = 1$: x = y + 2
  - $i = 2$: x = y + 2
  - $i = 3$: x = y + 2
- Therefore, the time complexity is $O(1) + O(n) = O(n)$.



Time Complexity Notations

When calculating the time complexity of an algorithm, we often use different notations to describe different scenarios:

- Ω (Omega): Best Case
- Θ (Theta): Average Case
- O (Big O): Worst Case



Understanding the Time Complexity of Example 2

- The line 'x = y + 2' runs once, hence $O(1)$.
- The loop runs n times, hence $O(n)$.

- Therefore, the total time complexity is $O(1) + O(n) = O(n)$.
- We do not calculate exact runtime in milliseconds, but we express the complexity in Big O notation.
- For example, in an interview, we would say the algorithm has a time complexity of $O(n)$, indicating linear time complexity.

Example 4: Complicated Code Example

Let's look at a slightly more complicated code example.

Example 4

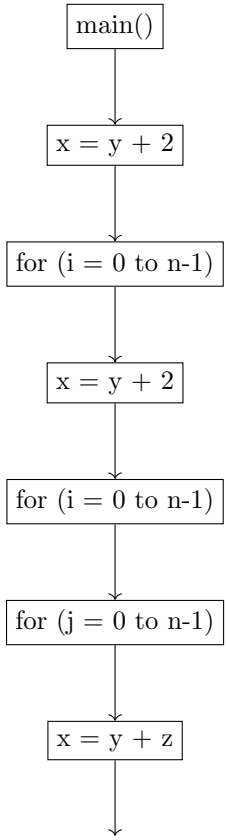
```
main() {
    x = y + 2;
    for (i = 0 to n-1) {
        x = y + 2;
    }
    for (i = 0 to n-1) {
        for (j = 0 to n-1) {
            x = y + z;
        }
    }
}
```

Question: What is the time complexity of this code?

1. $O(n^2)$
2. $O(n) + O(n^2)$
3. $O(1) + O(n) + O(n^2)$
4. All of the above

The correct answer is: All of the above.
Explanation:

- The line 'x = y + 2' runs once, hence $O(1)$.
- The first loop runs n times, hence $O(n)$.
- The nested loop runs n^2 times, hence $O(n^2)$.
- Therefore, the total time complexity is $O(1) + O(n) + O(n^2)$.
- We always take the highest degree when it is cumulative. Therefore, we approximate the overall complexity to $O(n^2)$.



Time Complexity $= O(1) + O(n) + O(n^2) = O(n^2)$

Detailed Example with $n = 3$

Let's suppose the value of $n = 3$:

- When $i = 0$:
  - $j = 0$: $x = y + z$
  - $j = 1$: $x = y + z$
  - $j = 2$: $x = y + z$
- When $i = 1$:
  - $j = 0$: $x = y + z$
  - $j = 1$: $x = y + z$
  - $j = 2$: $x = y + z$
- When $i = 2$:

- $j = 0$: $x = y + z$
- $j = 1$: $x = y + z$
- $j = 2$: $x = y + z$

This results in a total of 9 operations, which confirms that the time complexity is $O(n^2)$.

Nested and Individual Loops

```
for (i = 0 to n-1) {
    for (j = 0 to n-1) {
        a = b + c;
        d = y + 2;
        c = a + x;
    }
}
```

- This is an example of nested loops, leading to a time complexity of $O(n^2)$.
- When loops are nested, we multiply their complexities.
- When loops are individual, we add their complexities.

```
a = b + c;
d = y + 2;
c = a + x;
```

- These statements run once each, hence the time complexity is $O(1)$.

```
for (i = 0 to n-1) {
    a = b + c;
    d = y + 2;
    c = a + x;
}
```

- These statements are inside a loop that runs n times, hence the time complexity is $O(n)$.

```
for (i = 0 to n-1) {
    for (j = 0 to n-1) {
        a = b + c;
        d = y + 2;
        c = a + x;
    }
}
```

- These statements are inside nested loops that run n^2 times, hence the time complexity is $O(n^2)$.

Quiz

Question: What is the complexity of this code?

```
x = y + 2;
x = y + 2;
x = y + 2;
x = y + 2;
x = y + 2;
```

Answer: $O(1)$

Code Complexity Analysis

Example of $O(n) + O(m)$ Complexity

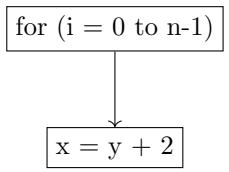
```
for (i = 0 to n-1) {
    x = y + 2;
}

for (i = 0 to m-1) {
    x = y + 2;
}
```

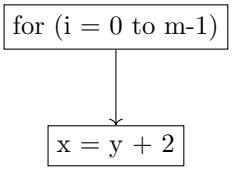
We have two notations:

- $O(n) + O(m)$
- $O(n + m)$

Question: Are both statements the same or not?
Answer:

- Check the complexity of the loops.
- The first loop runs $O(n)$ times.
- The second loop runs $O(m)$ times.
- Therefore, the complexity is $O(n) + O(m)$.
- The answer is $O(n) + O(m)$.



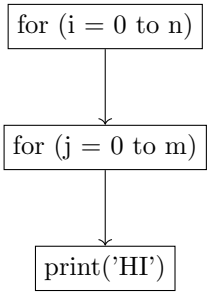


Example of $O(n * m)$ Complexity

```
for (i = 0 to n) {
    for (j = 0 to m) {
        print('HI');
    }
}
```

Question: What is the complexity of this code?
Answer: $O(n * m)$
Explanation:

- The outer loop runs n times.
- For each iteration of the outer loop, the inner loop runs m times.
- Therefore, the total number of iterations is $n * m$.
- Hence, the complexity is $O(n * m)$.

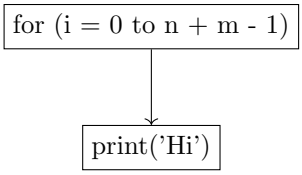


Example of $O(n + m)$ Complexity

```
for (i = 0 to n + m - 1) {
    print('Hi');
}
```

We are iterating over $n + m$ elements in a single loop.
Explanation:

- The loop runs $n + m$ times.
- Therefore, the complexity is $O(n + m)$.



Time Complexity Analysis

Iterative Code and Loops

Note:

1. Time complexity is determined by the presence of loops in the code.
2. The higher the number of loops, the higher the time complexity.

In a nutshell, if a code does not contain any loops (apart from recursion), its complexity is constant, denoted as $O(1)$.

There are two ways to write code:

- Iterative code - uses loops.
- Recursive code - uses recursive functions.

For iterative code, if there are no loops, the time complexity is $O(1)$. If there are loops, the complexity depends on the number and type of loops.

Example: Sum of Natural Numbers

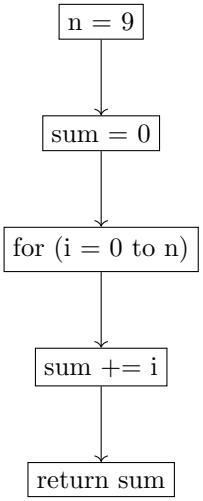
Let's consider the problem of finding the sum of natural numbers up to n, where n is a user-provided value.

There are two approaches:

Solution 1 (Iterative Approach):

```
n = 9  // user input

sum = 0
for (i = 0 to n) {
    sum += i
}
return sum
```

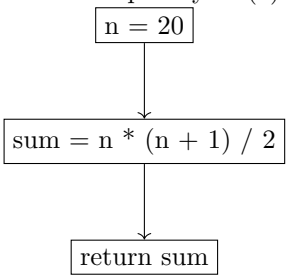
Time Complexity: $O(n)$



Solution 2 (Mathematical Formula):

```
n = 20  // user input

sum = n * (n + 1) / 2
return sum
```

Time Complexity: $O(1)$



Explanation:

- Solution 1 ($S1$) uses a loop to sum the numbers, resulting in a time complexity of $O(n)$.
- Solution 2 ($S2$) uses a mathematical formula to calculate the sum directly, resulting in a time complexity of $O(1)$.

In practice, always prefer the optimized solution with lower time complexity. In this case, Solution 2 is preferable because it has a constant time complexity.