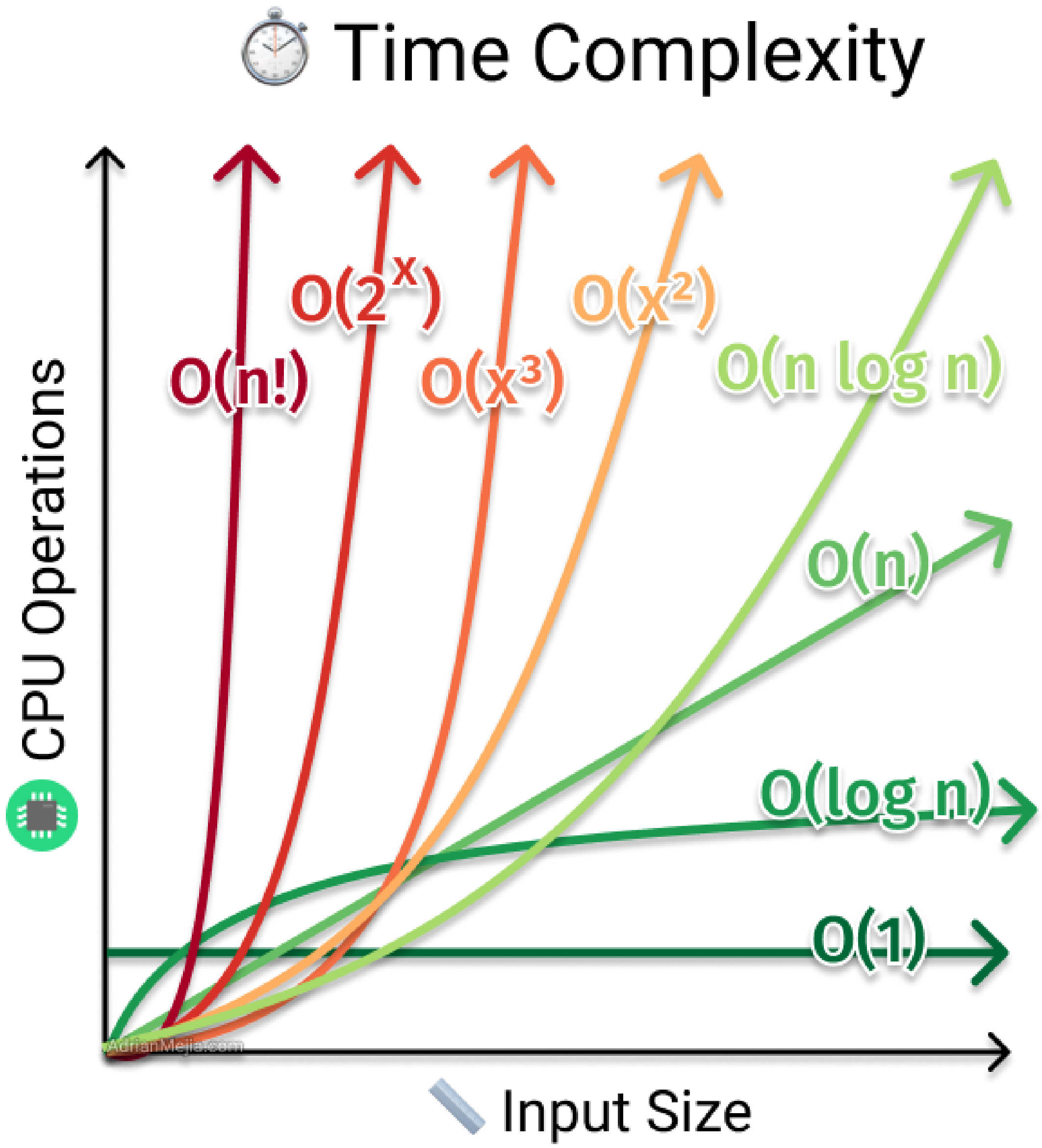
Time Complexity Pdf :

- **Constant** (`O(1)`)
- **Logarithmic** (`O(log n)`)
- **Linear** (`O(n)`)
- **Linearithmic** (`O(n log n)`)
- **Quadratic** (`O(n^2)`)
- **Cubic** (`O(n^3)`)
- **Polynomial** (`O(n^k)`)
- **Exponential** (`O(2^n)`)
- **Factorial** (`O(n!)`)

Order of Growth	Description
1	Constant
log n	Logarithmic
n	Linear
n log n	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential



Example Comparison:

- `O(log n)` **vs.** `O(n)`:
  - `O(n)` **is bigger** because linear growth (`n`) outpaces logarithmic growth (`log n`).
- `O(n)` **vs.** `O(n^2)`:
  - `O(n^2)` **is bigger** because quadratic growth (`n^2`) grows faster than linear growth (`n`).
- `O(n^2)` **vs.** `O(n log n)`:

    - `O(n^2)` **is bigger** because polynomial growth (`n^2`) grows faster than linearithmic growth (`n log n`).
- `O(2^n)` **vs.** `O(n^3)`:
    - `O(2^n)` **is bigger** because exponential growth (`2^n`) outpaces any polynomial growth (`n^3`).

code only :

Here are the codes without comments:

1. Constant Time Complexity (O(1))

```
def constant_example(a, b):
    result = a + b
    return result

print(constant_example(5, 10))
```

2. Logarithmic Time Complexity (O(log n))

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
```

```
        else:
            right = mid - 1

    return -1

print(binary_search([1, 2, 3, 4, 5, 6], 4))
```

3. Linear Time Complexity (O(n))

```
def linear_example(arr):
    total = 0
    for i in arr:
        total += i
    return total

print(linear_example([1, 2, 3, 4, 5]))
```

4. Quadratic Time Complexity (O(n^2))

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr

print(bubble_sort([5, 2, 9, 1, 5, 6]))
```

5. Exponential Time Complexity (O(2^n))

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(5))
```

4o mini

1. Constant Time Complexity (O(1))

```
python
Copy code
def constant_example(a, b):
    # O(1) for variable initialization
    result = a + b  # O(1) for addition
    return result  # O(1) for return operation

# Example execution
print(constant_example(5, 10))  # O(1)
```

Explanation:

- The time complexity of all operations in this function is constant, O(1), because they do not depend on the size of the input.

---

2. Logarithmic Time Complexity (O(log n))

```
python
Copy code
def binary_search(arr, target):
```

```
    # O(1) for variable initialization
    left, right = 0, len(arr) - 1  # O(1)

    while left <= right:  # O(log n) for binary search loop
        mid = (left + right) // 2  # O(1) for mid calculation
        if arr[mid] == target:  # O(1) for comparison
            return mid  # O(1) for return
        elif arr[mid] < target:  # O(1) for comparison
            left = mid + 1  # O(1) for left pointer update
        else:  # O(1)
            right = mid - 1  # O(1)

    return -1  # O(1)

# Example execution
print(binary_search([1, 2, 3, 4, 5, 6], 4))  # O(log n)
```

Explanation:

- The binary search function cuts the search space in half each time, so the time complexity of the loop is O(log n). All other operations are O(1).

---

3. Linear Time Complexity (O(n))

```
python
Copy code
def linear_example(arr):
    # O(1) for variable initialization
    total = 0
    for i in arr:  # O(n) for iterating over the array
        total += i  # O(1) for addition operation
    return total  # O(1) for return operation

# Example execution
```

```
print(linear_example([1, 2, 3, 4, 5]))  # O(n)
```

Explanation:

- The function iterates through all `n` elements in the list, and the time complexity is dominated by the for loop, which is O(n).

---

4. Quadratic Time Complexity (O(n^2))

```
python
Copy code
def bubble_sort(arr):
    # O(1) for variable initialization
    n = len(arr)  # O(1)
    for i in range(n):  # O(n) for outer loop
        for j in range(0, n - i - 1):  # O(n) for inner loop
            if arr[j] > arr[j + 1]:  # O(1) for comparison
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # O
(1) for swap

    return arr  # O(1)

# Example execution
print(bubble_sort([5, 2, 9, 1, 5, 6]))  # O(n^2)
```

Explanation:

- The time complexity is O(n^2) because there are two nested loops, each running up to `n` times, resulting in `n * n` = O(n^2).

---

5. Exponential Time Complexity (O(2^n))

```
python
Copy code
```

```
def fibonacci(n):
    if n <= 1:  # O(1) for base case
        return n  # O(1)
    return fibonacci(n - 1) + fibonacci(n - 2)  # O(2^n) for
recursive calls

# Example execution
print(fibonacci(5))  # O(2^n)
```

Explanation:

- The recursive calls double with each increase in `n`, resulting in O(2^n) time complexity. The function has to compute each Fibonacci number by making two recursive calls for each non-base case input.

Summary of Time Complexity Comments:

Time Complexity	Operations
O(1)	All operations in the function are constant time operations, independent of input size.
O(log n)	Logarithmic complexity arises when the input size is halved in each operation, such as in binary search.
O(n)	Linear complexity occurs when the time required grows directly with the size of the input, like iterating over an array.
O(n^2)	Quadratic complexity arises in algorithms with nested loops that iterate over the input multiple times, such as bubble sort.
O(2^n)	Exponential complexity arises when the number of operations doubles with each increment in the input size, like in recursive Fibonacci computation.

Example 1: Linear + Logarithmic Complexity

```
def example1(arr):
    for i in arr:  # O(n)
        print(i)

    arr.sort()  # O(n log n)
    print(arr)
```

Time Complexity Calculation:

- The first part of the code (`for i in arr`) runs in O(n).

  O(n)O(n)

- The second part (`arr.sort()`) runs in O(nlogn).

  O(nlogn)O(n \log n)

- These are **sequential operations**, so the total complexity is O(n)+O(nlogn). We keep the higher term, so the total complexity is O(nlogn).

  O(n)+O(nlogn)O(n) + O(n \log n)

  O(nlogn)O(n \log n)

Example 2: Nested Loops

```
python
Copy code
def example2(arr):
    for i in range(len(arr)):  # O(n)
        for j in range(len(arr)):  # O(n)
            print(i, j)
```

Time Complexity Calculation:

- The outer loop runs n times, and for each iteration of the outer loop, the inner loop runs n times.

nn

nn

- These loops are **nested**, so the total complexity is O(n)×O(n)=O(n2).

  O(n)×O(n)=O(n2)O(n) \times O(n) = O(n^2)

Example 3: Recursive Example

```
python
Copy code
def recursive_example(n):
    if n <= 1:  # O(1)
        return n
    return recursive_example(n - 1) + recursive_example(n -
2)  # O(2^n)
```

Time Complexity Calculation:

- The function calls itself twice for each value of n. The total number of calls grows exponentially, resulting in O(2n).

  nn

  O(2n)O(2^n)

Final Rule Summary

- **Add** complexities when operations are **sequential**.
- **Multiply** complexities when operations are **nested** or **dependent on each other**.
- **Keep the largest term**: In a function with multiple time complexities, discard the smaller terms. The largest term dominates.

Question 1:

`f(n) = 3n^4 + 7n^3 + 5n^2 + 2n + 8`

Solution:

- The highest degree term is `n^4`.
- **Time Complexity**: **O(n^4)**

Question 2:

`g(n) = 2n^2 + 5n log n + 10n + 100`

Solution:

- The highest degree term is `n^2` because `n^2` dominates over `n log n` and `n`.
- **Time Complexity**: **O(n^2)**

Question 3:

`h(n) = 4n^3 + 3n^2 + 2n log n + 50`

Solution:

- The highest degree term is `n^3`.
- **Time Complexity**: **O(n^3)**

Question 4:

`i(n) = 3log n + 5n log n + n^2 + 50`

Solution:

- The highest degree term is `n^2`.
- **Time Complexity**: **O(n^2)**

Question 5:

`j(n) = 2^n + n^3 + 100n^2`

Solution:

- The highest degree term is `2^n` because exponential growth outpaces polynomial growth.
- **Time Complexity**: **O(2^n)**

Question 6:

`k(n) = 100n + 25n^2 log n + 500n + 30`

Solution:

- The highest degree term is `n^2 log n`.
- **Time Complexity**: **O(n^2 log n)**

Question 7:

`l(n) = n^3 + n^2 + 100log n`

Solution:

- The highest degree term is `n^3`.
- **Time Complexity**: **O(n^3)**

Question 8:

`m(n) = n + 3log n + n^2`

Solution:

- The highest degree term is `n^2`.
- **Time Complexity**: **O(n^2)**

Question 9:

`p(n) = n^5 + n^2 + 1000n log n`

Solution:

- The highest degree term is `n^5`.
- **Time Complexity**: **O(n^5)**

Question 10:

`q(n) = 50n log n + 2000n + log n`

Solution:

- The highest degree term is `n log n`.
- **Time Complexity**: **O(n log n)**

General Rule Recap:

1. **Remove lower-order terms**: Always remove smaller terms, such as `n`, `log n`, `constant values`, and any lower-order polynomial terms.
2. **Keep the highest-order term**: The largest term (in terms of growth) dominates the overall time complexity.

Here's a breakdown of how different time complexities behave in relation to input size (`n`) and how operations grow with respect to `n`. I'll include the growth behaviors for the different types of time complexities, focusing on how the number of operations changes as the input size increases.

Time Complexity Growth and Input-Operation Relation:

Time Complexity	Growth Behavior	Relation Between Input (n) and Operations	Rate of Growth
O(1) (Constant Time)	The operation does not depend on `n`.	**Operations** stay constant regardless of `n`.	No change in the number of operations as `n` increases.
O(log n) (Logarithmic Time)	The number of operations increases logarithmically with `n`.	Operations grow slowly as `n` increases, like `log(n)`.	As `n` doubles, the number of operations increases by a constant amount.
O(n) (Linear Time)	The number of operations grows directly proportional to `n`.	Operations increase linearly as `n` increases, meaning if `n` doubles, the operations also double.	Doubling `n` will double the number of operations.
O(n log n) (Linearithmic Time)	A combination of linear and logarithmic growth.	The operations grow faster than linear time but slower than quadratic.	If `n` doubles, the number of operations increases more than linearly but not as fast as `n^2`.
O(n^2) (Quadratic	The number of operations grows	Operations grow exponentially in relation	Doubling `n` results in a fourfold increase in

Time)	quadratically with `n`.	to the square of `n`. If `n` doubles, the operations increase fourfold.	operations.
O(2^n) (Exponential Time)	The number of operations grows exponentially with `n`.	Operations increase rapidly as `n` increases, making the function inefficient for large `n`.	Doubling `n` results in the number of operations being multiplied by 2.
O(n!) (Factorial Time)	The number of operations grows extremely rapidly.	Operations grow factorially with `n`, so even small increases in `n` result in huge increases in the number of operations.	Doubling `n` results in operations multiplying by `n!` (factorial), which is extremely fast growth.

Detailed Explanation of Growth Behavior:

O(1) (Constant Time):

- **Input (n)** does not affect the number of operations. Regardless of how large `n` is, the algorithm will take the same amount of time to complete.
- **Example**: Accessing an element in an array by its index.
    - **Operations**: Always constant, no matter the size of the array.

O(log n) (Logarithmic Time):

- As `n` increases, the number of operations increases logarithmically. This is typical in algorithms that divide the input in half at each step.
- **Example**: Binary Search.
    - **Operations**: If `n = 1000`, the number of operations is approximately 10 (`log2(1000) ≈ 10`). If `n` doubles to 2000, the number of operations will increase by only 1.

O(n) (Linear Time):

- The number of operations increases directly with the input size. If `n` doubles, the number of operations also doubles.

- **Example**: Iterating over an array.
    - **Operations**: If `n = 1000`, there will be 1000 operations. If `n` doubles to 2000, there will be 2000 operations.

O(n log n) (Linearithmic Time):

- This complexity grows faster than linear time but slower than quadratic time. It is common in efficient sorting algorithms.
- **Example**: Merge Sort, Quick Sort.
    - **Operations**: If `n = 1000`, it will perform about 1000 * log(1000) ≈ 10,000 operations. If `n` doubles to 2000, the number of operations grows by approximately 2 * log(2000), which is a more moderate increase compared to quadratic time.

O(n^2) (Quadratic Time):

- The number of operations grows as the square of `n`. This is typical for algorithms that involve nested loops over the same dataset.
- **Example**: Bubble Sort, Selection Sort.
    - **Operations**: If `n = 1000`, there will be about 1,000,000 operations (1000^2). If `n` doubles to 2000, the number of operations grows to 4,000,000 (2000^2).

O(2^n) (Exponential Time):

- This complexity grows very quickly. Each additional step in `n` doubles the number of operations, making it impractical for large `n`.
- **Example**: Solving the Traveling Salesman Problem using brute force.
    - **Operations**: If `n = 10`, there are about 1024 operations. If `n` increases to 20, the number of operations increases to over 1 million (2^20).

O(n!) (Factorial Time):

- This is the fastest-growing time complexity. Each increase in `n` leads to a massive increase in operations.

- **Example**: Generating all permutations of a list of items.
  - **Operations**: If `n = 5`, there are 120 operations (5!). If `n` increases to 6, there are 720 operations (6!).

Key Points to Remember:

1. **Linear Growth (O(n))**: Each additional input causes a linear increase in operations. Doubling the input size will double the operations.
2. **Logarithmic Growth (O(log n))**: The number of operations increases slowly. Doubling the input size results in only a slight increase in operations.
3. **Quadratic Growth (O(n^2))**: The number of operations grows quickly, so these algorithms are slower as the input size increases.
4. **Exponential and Factorial Growth (O(2^n) and O(n!))**: These complexities are impractical for large inputs due to the extremely rapid growth of operations.

Example 1 :

1. Constant Time Complexity (O(1))

```
def constant_time_operation():
    k = 0
    k = k + 1  # Just a single operation
    return k

print(constant_time_operation())
```

Explanation:

- The function performs a single operation regardless of input size `n`.
- The time complexity is **O(1)** because the execution time is constant.

---

2. Linear Time Complexity (O(n))

```
def linear_time_operation(n):
    k = 0
    for i in range(n):
        k = k + 1
    return k

print(linear_time_operation(1000))
```

Explanation:

- The function iterates over the range of `n` and performs a constant operation on each iteration.
- The time complexity is **O(n)** because the number of operations grows linearly with `n`.

---

3. Quadratic Time Complexity (O(n^2))

```
def quadratic_time_operation(n):
    k = 0
    for i in range(n):
        for j in range(n):
            k = k + 1
    return k
```

```
print(quadratic_time_operation(1000))
```

Explanation:

- The function has two nested loops, each iterating `n` times.
- The time complexity is **O(n^2)** because the number of operations grows quadratically with `n`.

---

4. Logarithmic Time Complexity (O(log n))

```
def logarithmic_time_operation(n):
    k = 0
    i = 1
    while i <= n:
        k = k + 1
        i = i * 2
    return k

print(logarithmic_time_operation(1000))
```

Explanation:

- The function doubles the value of `i` each time, so it takes logarithmic steps to reach `n`.
- The time complexity is **O(log n)** because the loop runs logarithmically.

---

5. Linearithmic Time Complexity (O(n log n))

```
def linearithmic_time_operation(n):
    k = 0
    for i in range(n):
```

```
        j = 1
        while j <= n:
            k = k + 1
            j = j * 2
    return k

print(linearithmic_time_operation(1000))
```

Explanation:

- The outer loop runs `n` times, and the inner loop runs logarithmically with respect to `n` (since `j` is multiplied by 2).
- The time complexity is **O(n log n)** because the inner loop is logarithmic and the outer loop runs `n` times.

Here's the code for **Exponential Time Complexity (O(2^n))** based on the previous examples. This code demonstrates how the number of operations grows exponentially with respect to `n`.

Exponential Time Complexity (O(2^n))

```
def exponential_time_operation(n):
    if n <= 0:
        return 0
    return exponential_time_operation(n-1) + exponential_time
_operation(n-1) + 1

print(exponential_time_operation(5))
```

Example 2 :

1. O(1) - Constant Time Complexity (Best)

- **Description**: The algorithm runs in constant time, regardless of the size of the input. This is the most efficient time complexity.

Example:

```
python
Copy code
def sum_digits(num):
    return sum(int(digit) for digit in str(num))
```

- **Explanation**: Despite the string conversion, this is as close to constant time as possible, though realistically it still behaves in **O(n)** in most cases. However, for small numbers or constant-size inputs, it's essentially constant.

---

2. O(log n) - Logarithmic Time Complexity

- **Description**: The algorithm's time complexity increases logarithmically with the input size. This is more efficient than linear time for large inputs.

Example:

```
python
Copy code
def sum_digits(num):
    total = 0
    while num > 0:
        total += num % 10
        num //= 10
```

```
    return total
```

- **Explanation**: The number of operations needed decreases exponentially as the input grows. This is typical in algorithms that halve the problem size at each step, like binary search.

---

3. O(n) - Linear Time Complexity

- **Description**: The algorithm's time complexity increases linearly with the input size.

Example:

```
python
Copy code
def sum_digits(num):
    total = 0
    num_str = str(num)
    for digit in num_str:
        total += int(digit)
    return total
```

- **Explanation**: The time complexity is directly proportional to the number of digits in the input number. This is typical for problems that need to process each input element exactly once.

---

4. O(n log n) - Linearithmic Time Complexity

- **Description**: The algorithm's time complexity increases more slowly than **O(n^2)** but faster than **O(n)**. It is often seen in divide-and-conquer algorithms like merge sort or quicksort.

Example:

```
python
Copy code
def sum_digits(num):
    total = 0
    while num > 0:
        total += num % 10
        num //= 10
        for i in range(total):
            total += 1
    return total
```

- **Explanation**: The outer loop reduces the number by dividing it by 10 each time (**O(log n)**), but the inner loop has a linear dependency on the `total` value. The complexity of both combined results in **O(n log n)**.

---

5. O(n^2) - Quadratic Time Complexity (Worst)

- **Description**: The algorithm's time complexity grows quadratically with the input size. This is very inefficient for large inputs.

Example:

```
python
Copy code
def sum_digits(num):
    num_str = str(num)
    total = 0
    for i in range(len(num_str)):
        for j in range(i + 1):
            total += int(num_str[j])
    return total
```

- **Explanation**: This is a nested loop where the outer loop runs **n** times, and the inner loop runs up to **i + 1** times. This results in a total of **O(n^2)** operations.

•

Example 3 :

1. Recursive Approach:

In the simple recursive approach, we call the function twice for each number until the base case is reached.

Code:

```
python
Copy code
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Time Complexity Analysis:

- The function calls itself twice for each value of `n`, which leads to overlapping subproblems (i.e., calculating the same Fibonacci numbers multiple times).
- If you want to calculate `fibonacci(5)`, the function will calculate `fibonacci(4)` and `fibonacci(3)` and so on, resulting in many repeated calculations.

  For example:

  - `fibonacci(5)` calls `fibonacci(4)` and `fibonacci(3)`

- `fibonacci(4)` calls `fibonacci(3)` and `fibonacci(2)`
- This leads to repeated calculations for `fibonacci(3)` and `fibonacci(2)`.

- The number of function calls grows exponentially with `n`. Specifically, the time complexity of this approach is **O(2^n)** because the number of calls doubles at each level of recursion.

Example 4 :

1. O(1) - Constant Time Complexity

For **O(1)** time complexity, you need to eliminate loops or operations that grow with input size. Here's an example where we instantly return a result, without iterating:

```
python
Copy code
def find_pair(arr, target_sum):
    return target_sum == 0  # Instant check without any loops
```

Explanation:

- The function just checks if the target sum is 0 and immediately returns the result. This takes constant time regardless of the input size, so the time complexity is **O(1)**.

---

2. O(log n) - Logarithmic Time Complexity

To achieve **O(log n)** complexity, we need a problem where the input is reduced by half each time (e.g., binary search). Here's an example where we use binary search:

```python
python
Copy code
def find_pair(arr, target_sum):
    arr.sort()  # O(n log n)
    for i in range(len(arr)):  # O(n)
        remaining = target_sum - arr[i]
        if binary_search(arr, remaining):  # O(log n)
            return True
    return False
```

Explanation:

- **Sorting the array** takes **O(n log n)** time.
- **Binary search** is used inside the loop, which has a complexity of **O(log n)**.
- The total time complexity is dominated by the sorting step, resulting in **O(n log n)**, but each binary search reduces the problem size logarithmically.

---

3. O(n) - Linear Time Complexity

For **O(n)** complexity, we can use a **hashset** to check for pairs in linear time:

```python
python
Copy code
def find_pair(arr, target_sum):
    seen = set()
    for num in arr:
        if target_sum - num in seen:
            return True
        seen.add(num)
    return False
```

Explanation:

- The function iterates over each element of the array exactly once (**O(n)**).

- The **set** operations (adding and checking for membership) are **O(1)** on average, making the overall time complexity **O(n)**.
- This approach is much faster than using nested loops because it uses hashing to store and check for previously seen numbers.

---

4. O(n log n) - Linearithmic Time Complexity

To achieve **O(n log n)** time complexity, we can use sorting combined with a linear scan. This is common in algorithms like **merge sort** or **quick sort** where the time complexity of sorting dominates:

```
python
Copy code
def find_pair(arr, target_sum):
    arr.sort()  # O(n log n)
    left, right = 0, len(arr) - 1
    while left < right:
        current_sum = arr[left] + arr[right]
        if current_sum == target_sum:
            return True
        elif current_sum < target_sum:
            left += 1
        else:
            right -= 1
    return False
```

Explanation:

- **Sorting** the array takes **O(n log n)** time.
- After sorting, we use two pointers (`left` and `right`) to find the pair. The two-pointer approach runs in **O(n)** time.
- Thus, the overall time complexity is dominated by the sorting step, so it is **O(n log n)**.

---

5. O(n²) - Quadratic Time Complexity (Original Code)

As shown in the original code:

```
python
Copy code
def find_pair(arr, target_sum):
    for i in range(len(arr)):  # Outer loop
        for j in range(i + 1, len(arr)):  # Inner loop
            if arr[i] + arr[j] == target_sum:
                return True
    return False
```

Explanation:

- **Outer loop** runs **n** times.
- **Inner loop** runs a decreasing number of times, from **n-1** down to 1.
- The total number of operations is the sum of the first **n-1** integers, which is **O(n²)**.

Thus, the time complexity of the original code is **O(n²)**.

Summary of Time Complexities

Time Complexity	Code Example	Explanation
O(1)	`return target_sum == 0`	Instant check with no loops.
O(log n)	`binary_search` in sorted array	Halving the problem size at each step (binary search).
O(n)	Using a hashset for lookups (`set`)	Linear scan with **O(1)** set operations.
O(n log n)	Sorting + two-pointer technique	Sorting + linear scan with two pointers.
O(n²)	Original double loop method	Nested loops that check every pair of elements.

Queston:

Here's a modified version of the code with different function and variable names:

```
python
Copy code
def exponentiate(value):
    if value < 1:
        return 0
    elif value == 1:
        print(1)
        return 1
    else:
        previous_result = exponentiate(value // 2)
        current_result = previous_result * 2
        print(current_result)
        return current_result
```

In this version:

- The function name has been changed to `exponentiate`.
- The variable `num` has been changed to `value`.
- The variables `prev` and `curr` have been renamed to `previous_result` and `current_result`, respectively.

The logic and behavior of the code remain the same.

4o mini

The time complexity of the `exponentiate` function can be calculated by analyzing how many times the function is called recursively.

Time Complexity Analysis:

In the given function, the recursion divides the input by 2 at each step. The key observation here is that the function is performing a **recursive division** by 2, which means the recursion depth is logarithmic.

Detailed Breakdown:

1. The base case of the recursion is when `value` is less than or equal to 1. In this case, it returns immediately, taking constant time, O(1).
2. In each recursive step:
    - The function calls itself with `value // 2` (integer division by 2).
    - The number of recursive calls made is proportional to how many times `value` can be halved until it reaches 1.

Since each recursive call reduces `value` by half, the total number of recursive calls will be proportional to **$\log_2$(value)**. This is because halving the value repeatedly leads to a logarithmic number of calls.

Each recursive call involves a constant number of operations (such as multiplying and printing), which is **O(1)**.

Final Time Complexity:

- The number of recursive calls is **O(log n)** where `n` is the input value (`value`).
- Since each call performs constant work (O(1)), the overall time complexity is **O(log n)**.

Thus, the time complexity of the function `exponentiate(value)` is **O(log n)**.

10 Question :

1. Question:

```
python
Copy code
def find_max(arr):
    max_num = arr[0]
    for num in arr:
        if num > max_num:
            max_num = num
    return max_num
```

Answer:

- **Time Complexity: O(n)**
- **Explanation:** The function iterates over the array once to find the maximum value. Since the loop runs **n** times (where **n** is the length of the array), the time complexity is **O(n)**.

---

2. Question:

```
python
Copy code
def sum_pairs(arr):
    total = 0
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            total += arr[i] + arr[j]
    return total
```

Answer:

- **Time Complexity: O(n^2)**

- **Explanation:** The function has two nested loops, each iterating over the array. The outer loop runs **n** times, and the inner loop runs **n-i** times. The total number of operations is proportional to **n^2**.

---

3. Question:

```
python
Copy code
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Answer:

- **Time Complexity: O(log n)**
- **Explanation:** Binary search divides the array into halves at each step. The number of operations reduces exponentially, so the time complexity is logarithmic: **O(log n)**.

---

4. Question:

```
python
Copy code
def reverse_string(s):
    reversed_str = ""
    for i in range(len(s)-1, -1, -1):
```

```
        reversed_str += s[i]
    return reversed_str
```

Answer:

- **Time Complexity: O(n)**
- **Explanation:** The loop runs **n** times, where **n** is the length of the string. Each concatenation operation takes **O(1)** time. Thus, the overall time complexity is **O(n)**.

---

5. Question:

```
python
Copy code
def is_palindrome(s):
    return s == s[::-1]
```

Answer:

- **Time Complexity: O(n)**
- **Explanation:** The slicing operation `s[::-1]` creates a new string with the characters reversed, which takes **O(n)** time. The comparison `s == s[::-1]` also takes **O(n)** time. Thus, the overall time complexity is **O(n)**.

---

6. Question:

```
python
Copy code
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
```

```
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Answer:

- **Time Complexity: O(n log n)**
- **Explanation:** Merge Sort divides the array into halves recursively, which takes **O(log n)** time. The merging process takes **O(n)** time at each level of recursion. So, the total time complexity is **O(n log n)**.

---

7. Question:

```
python
Copy code
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
```

```
    return True
```

Answer:

- **Time Complexity: O(√n)**
- **Explanation:** The loop runs from 2 to √n, so it performs approximately √n iterations. Therefore, the time complexity is **O(√n)**.

---

8. Question:

```
python
Copy code
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Answer:

- **Time Complexity: O(2^n)**
- **Explanation:** The function makes two recursive calls at each level, leading to an exponential growth in the number of function calls. The time complexity is **O(2^n)** due to this exponential recursion.

---

9. Question:

```
python
Copy code
def count_occurrences(arr, target):
    count = 0
    for num in arr:
        if num == target:
            count += 1
```

```
    return count
```

Answer:

- **Time Complexity: O(n)**
- **Explanation:** The function iterates over the array once and checks for each element if it matches the target. Since the loop runs **n** times, the time complexity is **O(n)**.

---

10. Question:

```
python
Copy code
def max_subarray_sum(arr):
    max_sum = float('-inf')
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            subarray_sum = sum(arr[i:j+1])
            max_sum = max(max_sum, subarray_sum)
    return max_sum
```

Answer:

- **Time Complexity: O(n^3)**
- **Explanation:** The function uses two nested loops to generate all possible subarrays. The innermost `sum` function also takes **O(n)** time for each subarray. Therefore, the total time complexity is **O(n^3)** due to the triple nested operations.

Summary of Time Complexities:

Question Number	Time Complexity	Explanation
1	**O(n)**	Linear loop over the array

2	**O(n^2)**	Two nested loops
3	**O(log n)**	Binary search (halving the input each time)
4	**O(n)**	Single loop over the string
5	**O(n)**	String comparison and slicing
6	**O(n log n)**	Merge Sort
7	**O(√n)**	Loop up to square root of `n`
8	**O(2^n)**	Exponential recursive calls
9	**O(n)**	Loop through the array
10	**O(n^3)**	Three nested operations: two loops + sum