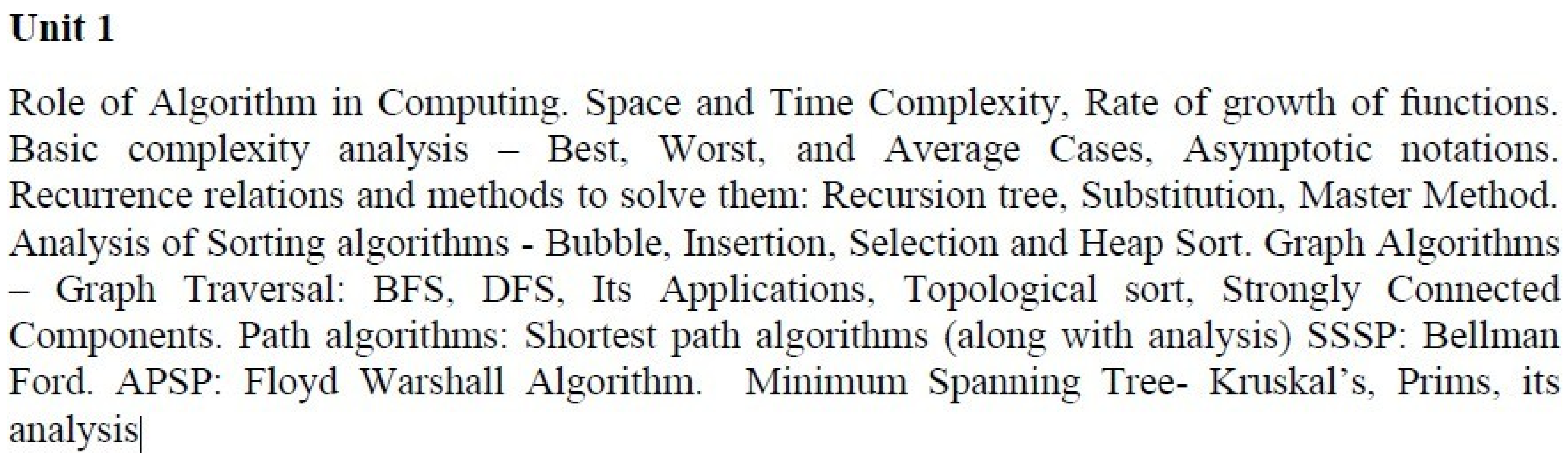
20CYS305 ALGORITHMS: DESIGN AND ANALYSIS L-T-P-C:3-0-0-3

Pre-Requisite(s): 20CYS214 - Data Structures and Algorithms

Syllabus



Unit 1

Role of Algorithm in Computing. Space and Time Complexity, Rate of growth of functions. Basic complexity analysis – Best, Worst, and Average Cases, Asymptotic notations. Recurrence relations and methods to solve them: Recursion tree, Substitution, Master Method. Analysis of Sorting algorithms - Bubble, Insertion, Selection and Heap Sort. Graph Algorithms – Graph Traversal: BFS, DFS, Its Applications, Topological sort, Strongly Connected Components. Path algorithms: Shortest path algorithms (along with analysis) SSSP: Bellman Ford. APSP: Floyd Warshall Algorithm. Minimum Spanning Tree- Kruskal's, Prims, its analysis

AMRITA VISHWA VIDYAPEETHAM

20CYS305 ALGORITHMS: DESIGN AND ANALYSIS L-T-P-C:3-0-0-3

Pre-Requisite(s): 20CYS214 - Data Structures and Algorithms

Syllabus

Unit 2

Divide and Conquer: Merge Sort and Binary search type strategies, Pivot based strategies. Strassens's Algorithm for matrix multiplication, Long integer multiplication – Maximum subarray sum - Closest Pair problem as examples. Greedy Algorithm - Introduction to the method, Fractional Knapsack problem, Task Scheduling Problem, Huffman coding as examples. Dynamic Programming: Introduction to the method, Fibonacci numbers, 0-1 Knapsack problem, Matrix chain multiplication problem, Longest Common Subsequence, and other problems including problems incorporating combinatorics as examples.

Unit 3

Backtracking, Branch and Bound 0-1 Knapsack, N-Queen problem, subset sum as some examples. String Matching: Rabin Karp, Boyer Moore, Knuth-Morris-Pratt (KMP). Network Flow and Matching: Flow Algorithms - Maximum Flow - Cuts Maximum Bipartite Matching. Introduction to NP class: Definitions P, NP, NP complete, NP hard, Examples of P and NP.

Text Book

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, Third Edition, The MIT Press Cambridge, Massachusetts; 2009 (Indian reprint: Prentice-Hall).

Reference(s)

1. *Michael T. Goodrich and Roberto Tamassia, Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley and Sons; 2001.*
2. *Dasgupta S, Papadimitriou C and Vazirani U. Algorithms, Eighth Edition, Tata McGraw-Hill; 2009.*
3. *Jon Kleinberg, Eva Tardos. Algorithm Design, First Edition, Pearson New International Edition, Pearson Education Limited; 2014.*

Evaluation Pattern.

S.No		Weightage
1.	Mid Term	30
2.	Continuous Assessment	30
3.	End Semester Theory	40
	Total	100

Continuous Assessment Theory (CAT):

S.NO	Assessment	Marks
1.	Quiz	10
2.	Problem Solving	10
3.	Class Test	10
	TOTAL	30

An ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output.

An algorithm is a step-by-step procedure or set of rules designed to solve a specific problem or perform a particular task. It's a finite sequence of well-defined instructions that, when executed, produces a desired output from a given input.

Key Characteristics of Algorithms

1. **Finiteness**: An algorithm must have a finite number of steps.
2. **Definiteness**: Each step must be clearly and unambiguously defined.
3. **Input**: An algorithm should have zero or more inputs.
4. **Output**: An algorithm must produce at least one output.
5. **Effectiveness**: Each step can be carried out with finite amount of effort, in principle, by a person using a pencil and paper.
6. **Correctness**: They should solve the problem they are designed for.

Importance of Algorithms in Computing

- **Efficiency**: Algorithms **optimize the use of resources** such as time and memory, making processes faster and more efficient.
- **Problem Solving**: They provide a systematic approach to solving problems, which is essential in programming and software development.
- **Reusability**: Well-designed algorithms can be reused in different programs and applications.
- **Foundation of Computing**: They are fundamental to the functioning of computers and software, forming the basis for data processing and task automation.

- They solve real-world problems, from sorting data to route planning.

Basic Components of Algorithm Design

1. **Input**: What data does the algorithm need?
2. **Output**: What result does it produce?
3. **Control Structures**: Loops, conditionals, and function calls.
4. **Data Structures**: Arrays, lists, trees, etc. Decide on the appropriate data structures for storing and managing data.

Steps in Algorithm Analysis

1. **Correctness**: Does it solve the problem correctly?
2. **Efficiency**: How fast does it run? (Time complexity)
3. **Space Usage**: How much memory does it need? (Space complexity)

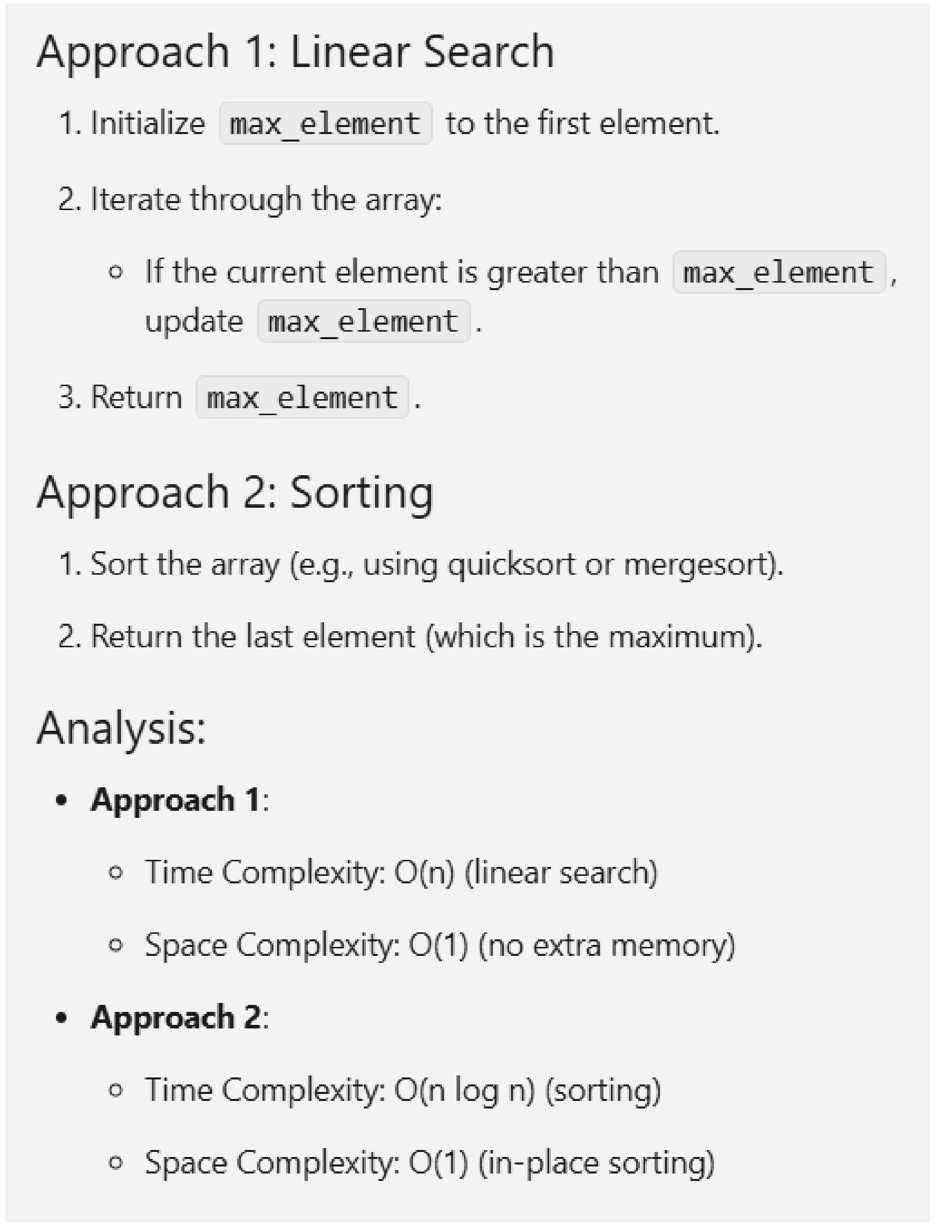
Steps in Algorithm Analysis

- **Correctness**: Ensure the algorithm produces the correct output for all possible valid inputs.
- **Analyze input size**: Determine how the input affects the algorithm's performance.
- **Identify operations**: List the basic operations the algorithm performs.
- **Count operations**: Estimate how many times each operation is executed.
- ***Efficiency**: Evaluate the time and space complexity of the algorithm, often using Big O notation.*
  - ***Time Complexity**: Measure of the amount of time an algorithm takes to complete as a function of the length of the input.*
  - ***Space Complexity**: Measure of the amount of memory an algorithm uses as a function of the length of the input.*
  - ***Express complexity**: Use Big O notation to describe the algorithm's time and space complexity.*
- **Scalability**: Assess how the algorithm performs as the size of the input grows.
- **Robustness**: Ensure the algorithm can handle unexpected inputs gracefully.
- **Compare alternatives**: Evaluate different approaches to solve the same problem.

Example: Finding the Maximum Element in an Array

Approach 1: Linear Search

Approach 2: Sorting



Approach 1: Linear Search

1. Initialize `max_element` to the first element.
2. Iterate through the array:
   - If the current element is greater than `max_element`, update `max_element`.
3. Return `max_element`.

Approach 2: Sorting

1. Sort the array (e.g., using quicksort or mergesort).
2. Return the last element (which is the maximum).

Analysis:

- **Approach 1**:
  - Time Complexity: O(n) (linear search)
  - Space Complexity: O(1) (no extra memory)
- **Approach 2**:
  - Time Complexity: O(n log n) (sorting)
  - Space Complexity: O(1) (in-place sorting)

Linear Search

Problem

Search for a target element in an array of n integers using linear search.

Algorithm Definition

1. Start at the first element of the array.
2. Compare the current element with the target element.
3. If they match, return the index.
4. If they don't match, move to the next element.
5. Repeat steps 2-4 until you find the target or reach the end of the array.

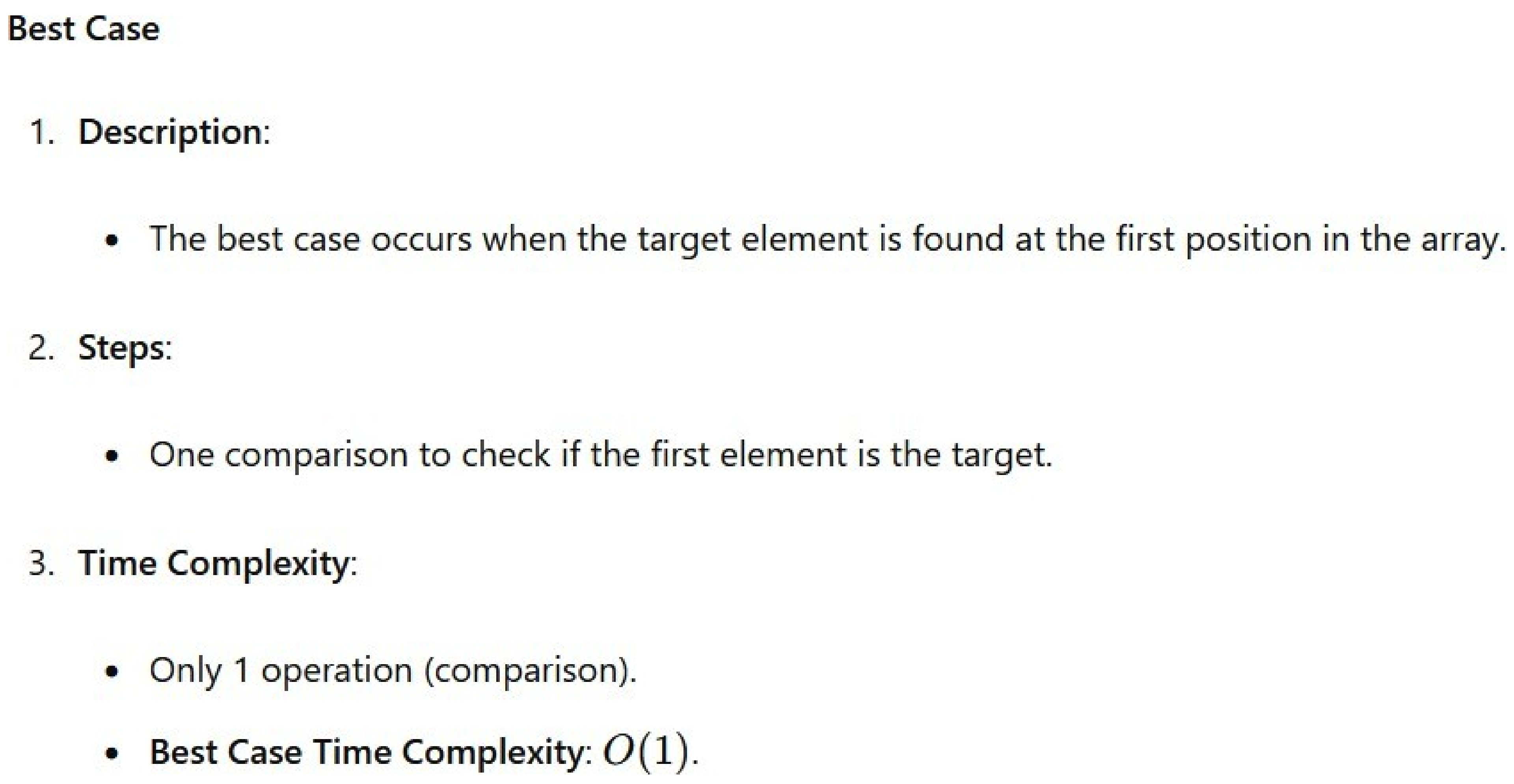
Pseudocode

```
function linearSearch(array, target):

  for i from 0 to n-1:
     if array[i] == target:
         return i
  return -1
```

3 cases of Complexity

- Best,
- Worst and,
- Average



Best Case

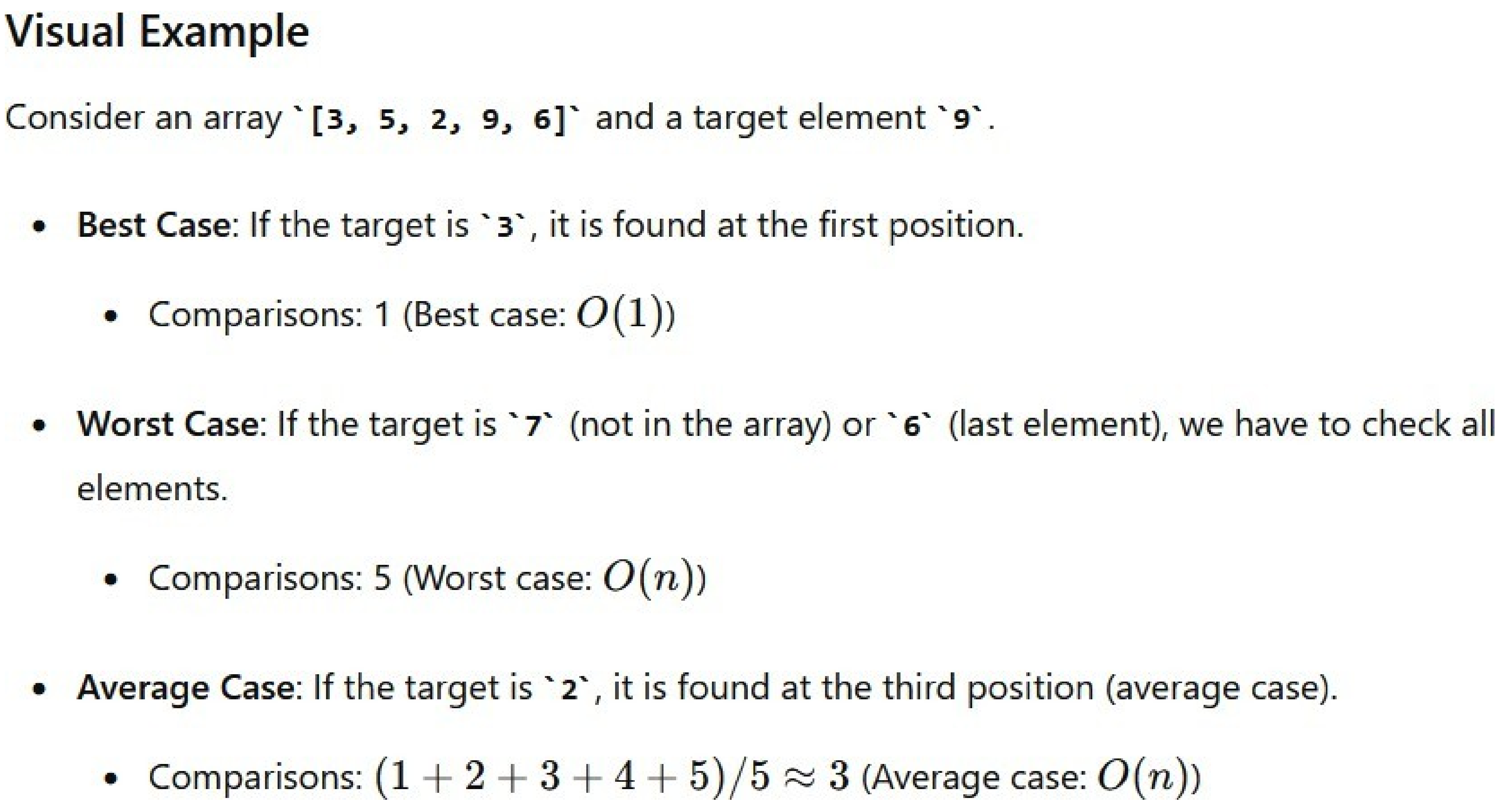
1. **Description**:
   - The best case occurs when the target element is found at the first position in the array.
2. **Steps**:
   - One comparison to check if the first element is the target.
3. **Time Complexity**:
   - Only 1 operation (comparison).
   - **Best Case Time Complexity**: $O(1)$.

Worst Case

1. **Description**:
   - The worst case occurs when the target element is not present in the array or is at the last position.
2. **Steps**:
   - Compare each element of the array with the target until the end.
   - For an array of n elements, this means n comparisons.
3. **Time Complexity**:
   - n comparisons.
   - **Worst Case Time Complexity**: $O(n)$.

Average Case

1. **Description**:
   - The average case considers the scenario where the target element could be in any position in the array with equal probability.
2. **Steps**:
   - On average, the target element will be found halfway through the array.
   - This means $n/2$ comparisons, on average.
3. **Time Complexity**:
   - While the exact number is $n/2$, in Big O notation, we drop constant factors.
   - **Average Case Time Complexity**: $O(n)$.



Visual Example

Consider an array `[3, 5, 2, 9, 6]` and a target element `9`.

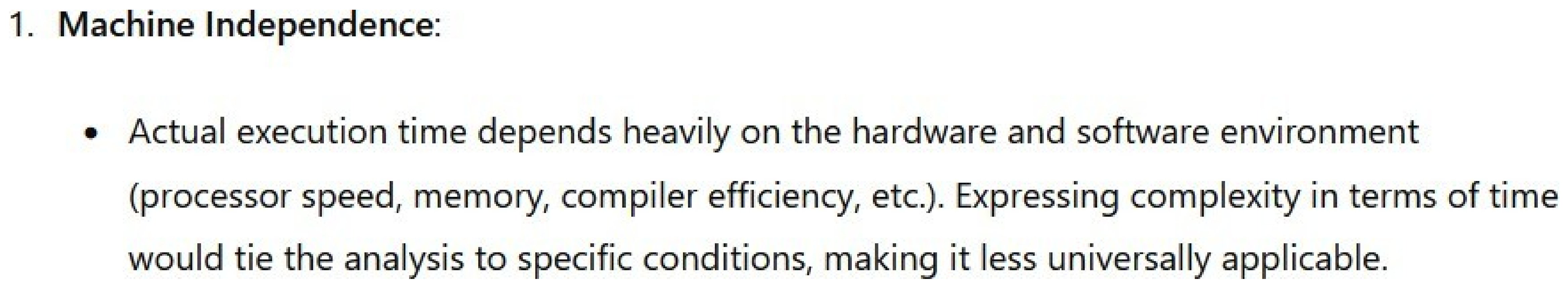
- **Best Case**: If the target is `3`, it is found at the first position.
  - Comparisons: 1 (Best case: $O(1)$)
- **Worst Case**: If the target is `7` (not in the array) or `6` (last element), we have to check all elements.
  - Comparisons: 5 (Worst case: $O(n)$)
- **Average Case**: If the target is `2`, it is found at the third position (average case).
  - Comparisons: $(1+2+3+4+5)/5 \approx 3$ (Average case: $O(n)$)

Summary

- **Best Case:** $O(1)$ - The target is at the beginning.
- **Worst Case:** $O(n)$ - The target is at the end or not in the array.
- **Average Case:** $O(n)$ - The target is somewhere in the middle.

Why the time complexity is not expressed in time that is seconds or minutes?

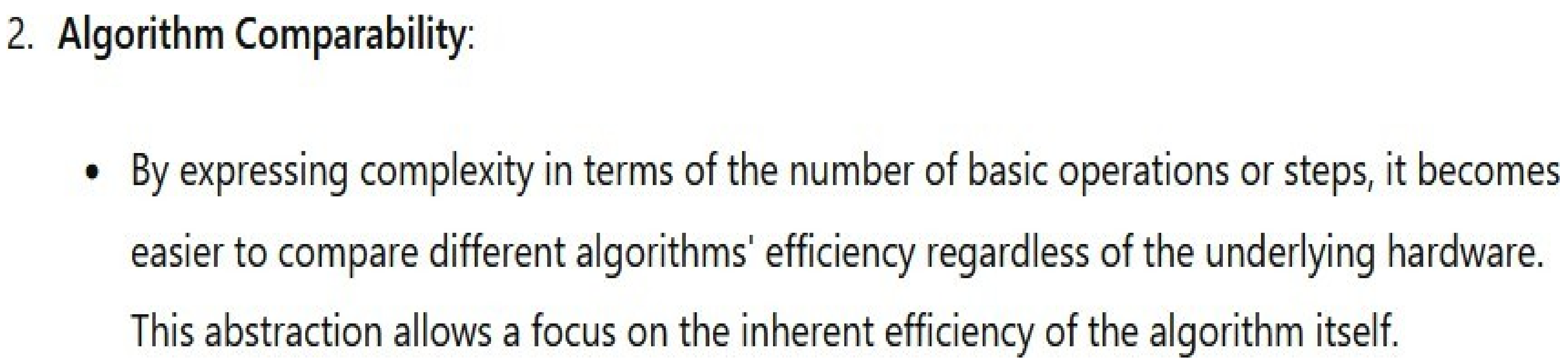
Why the time complexity is not expressed in time that is seconds or minutes?



1. **Machine Independence:**

   - Actual execution time depends heavily on the hardware and software environment (processor speed, memory, compiler efficiency, etc.). Expressing complexity in terms of time would tie the analysis to specific conditions, making it less universally applicable.

For example, if we said "Algorithm A takes 2 seconds and Algorithm B takes 4 seconds," that might be true on one computer, but on a faster computer, it might be "1 second and 2 seconds" instead. The relative performance is the same, but the actual times differ.

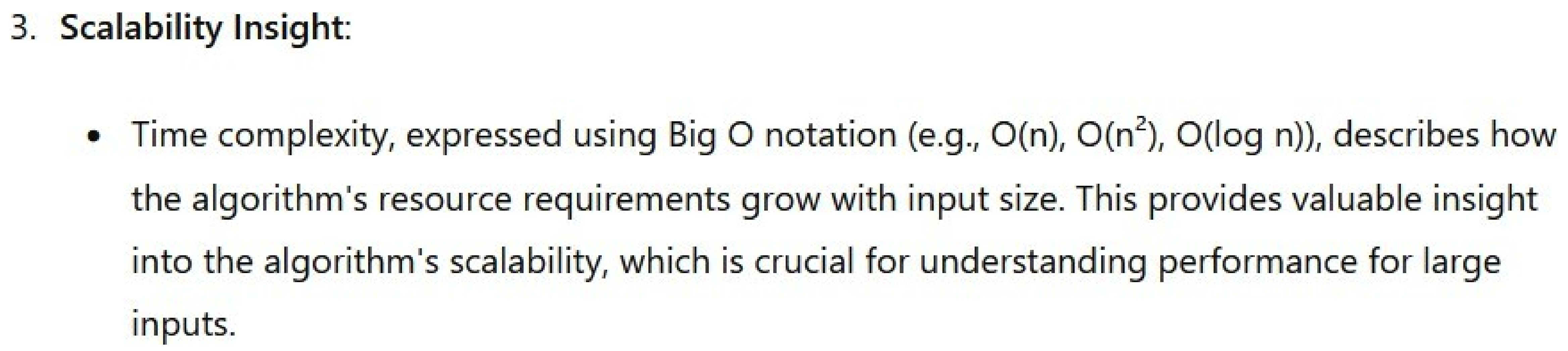


2. Algorithm Comparability:

- By expressing complexity in terms of the number of basic operations or steps, it becomes easier to compare different algorithms' efficiency regardless of the underlying hardware. This abstraction allows a focus on the inherent efficiency of the algorithm itself.

Abstraction: ***By using abstract units like "number of operations," we can analyze algorithms without actually implementing them.*** This allows us to make theoretical comparisons and improvements.

Comparability: We can compare this algorithm's O(n) complexity with another algorithm, say O(n^2) without worrying about the exact execution time.



3. **Scalability Insight:**

- Time complexity, expressed using Big O notation (e.g., O(n), O(n^2), O(log n)), describes how the algorithm's resource requirements grow with input size. This provides valuable insight into the algorithm's scalability, which is crucial for understanding performance for large inputs.

1. Time Complexity: **Time complexity measures how the runtime of an algorithm grows as the input size increases.**

2. Space Complexity: **Space complexity measures how much additional memory an algorithm needs as the input size increases.**

Input size focus: The real power of complexity analysis comes from **understanding how an algorithm's performance changes as the input grows.** This is crucial because an algorithm that works well for small inputs might become impractical for larger ones.

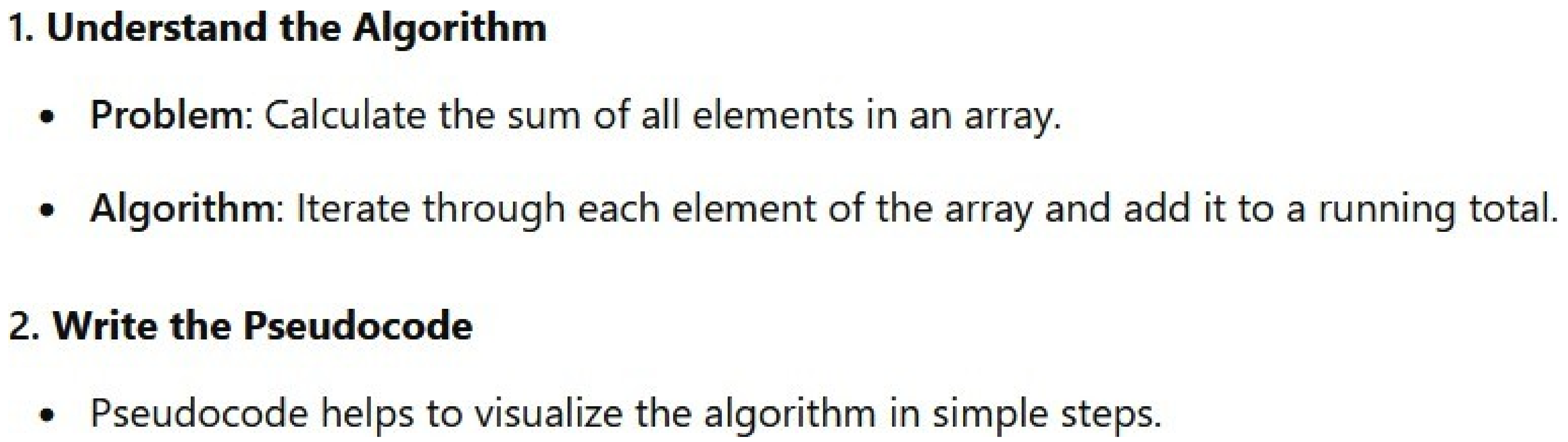
Scalability: Knowing the complexity is O(n) tells us that if the input size doubles, the number of operations approximately doubles, providing insight into how the algorithm scales.

Simplification: Using Big O notation, we ignore less significant terms and constants, simplifying the analysis and focusing on the most impactful part of the algorithm's behavior.

6. **Big O Notation:** This is the standard way to express algorithm complexity. It gives us an upper bound on the growth rate of the number of operations. Common complexities include:

1. O(1): Constant time (best)
2. O(log n): Logarithmic time
3. O(n): Linear time
4. O(n log n): Linearithmic time
5. O(n^2): Quadratic time
6. O(2^n): Exponential time (worst)

Step-by-Step Procedure to Express Algorithm Complexity



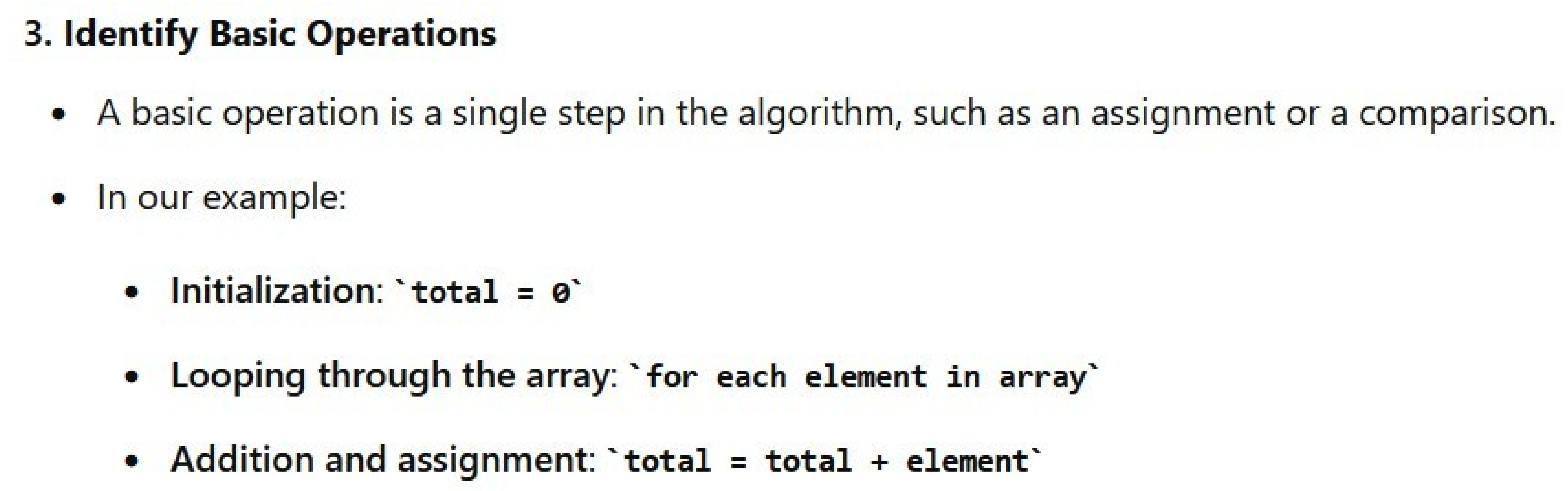
1. Understand the Algorithm

- **Problem**: Calculate the sum of all elements in an array.
- **Algorithm**: Iterate through each element of the array and add it to a running total.

2. Write the Pseudocode

- Pseudocode helps to visualize the algorithm in simple steps.

```
function sumArray(array):
    total = 0
    for each element in array:
        total = total + element
    return total
```



3. Identify Basic Operations

- A basic operation is a single step in the algorithm, such as an assignment or a comparison.
- In our example:
  - Initialization: `total = 0`
  - Looping through the array: `for each element in array`
  - Addition and assignment: `total = total + element`

4. Count the Number of Basic Operations

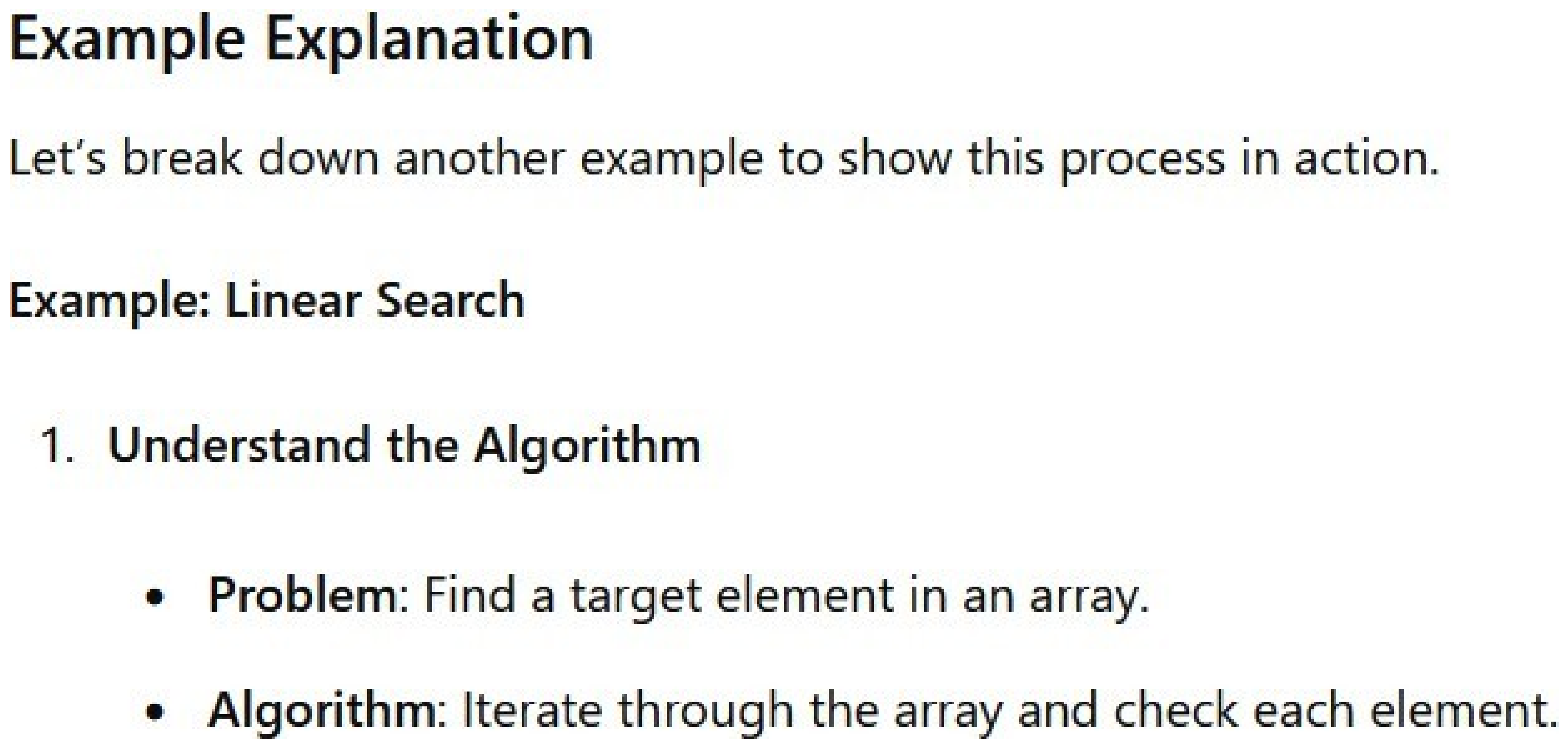
- Count how many times each basic operation occurs based on the input size n (number of elements in the array).
- **Initialization**: 1 operation.
- **Looping**: n times, once for each element.
- **Addition and assignment**: n times, once for each element.

5. Express in Terms of Input Size n

- Total operations = 1 (initialization) + n (loop iterations) + n (additions and assignments).
- Sum = $1 + n + n = 1 + 2n$.

6. Simplify Using Big O Notation

- Big O notation focuses on the term that grows the fastest as n increases.
- In $1 + 2n$, the term $2n$ dominates as n grows larger.
- Ignore constant factors and lower-order terms.
- **Time Complexity**: $O(n)$.



Example Explanation

Let’s break down another example to show this process in action.

Example: Linear Search

1. **Understand the Algorithm**

   - **Problem**: Find a target element in an array.
   - **Algorithm**: Iterate through the array and check each element.

2. Write the Pseudocode

```
function linearSearch(array, target):
  for i from 0 to n-1:
    if array[i] ==
      target: return i
  return -1
```

3. Identify Basic Operations

- **Initialization**: None explicitly in this pseudocode.
- **Looping through the array**: `for i from 0 to n-1`
- **Comparison**: `if array[i] == target`
- **Return statement**: `return i` (only once when the target is found)

4. **Count the Number of Basic Operations**

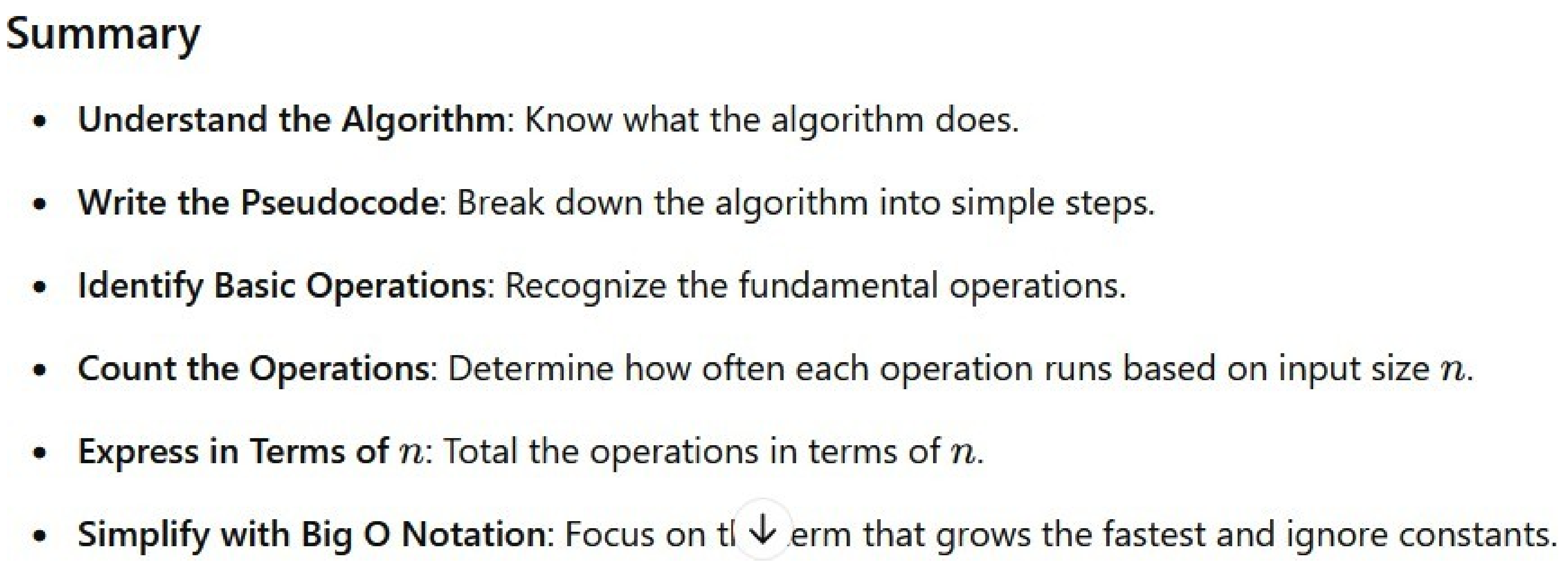
- **Looping**: n times (one for each element).
- **Comparison**: n times (one for each element).
- **Return statement**: Executed once when the target is found or not found after n comparisons.

5. **Express in Terms of Input Size n**

- Total operations = n (loop iterations) + n (comparisons).

6. Simplify Using Big O Notation

- Sum = $n + n = 2n$.
- The dominant term is n.
- **Time Complexity**: $O(n)$.



Summary

- **Understand the Algorithm**: Know what the algorithm does.
- **Write the Pseudocode**: Break down the algorithm into simple steps.
- **Identify Basic Operations**: Recognize the fundamental operations.
- **Count the Operations**: Determine how often each operation runs based on input size n.
- **Express in Terms of n**: Total the operations in terms of n.
- **Simplify with Big O Notation**: Focus on tl erm that grows the fastest and ignore constants.

Time Complexity: Time complexity measures how the runtime of an algorithm grows as the input size increases.

Space Complexity: Space complexity measures how much additional memory an algorithm needs as the input size increases.

Input Size (n)

- The input size, denoted as **n**, represents the size of the problem instance.
- For example:
  - If we're sorting an array, **n** is the number of elements in the array.
  - If we're searching in a graph, **n** is the number of nodes or edges.

Runtime Function T(n)

- We measure an algorithm’s efficiency using a **runtime function** denoted as **T(n)**.
- It represents the time (or other resources) required by the algorithm to solve a problem of size **n**.
- For example, if we’re sorting an array, **T(n)** could be the time taken by the sorting algorithm.

What Is the Rate of Growth?

- When we analyze algorithms, we want to estimate how their performance changes as the input size increases.
- ***The rate of growth helps us understand how quickly the algorithm's runtime or resource usage (such as memory) increases concerning the input size.***
- We mainly focus on **time complexity**, which measures how long an algorithm takes to run.

Purpose of rate of growth: ***The rate of growth helps us understand how an algorithm's performance changes as the input size increases. This is crucial for comparing algorithms and predicting their behavior on large datasets.***

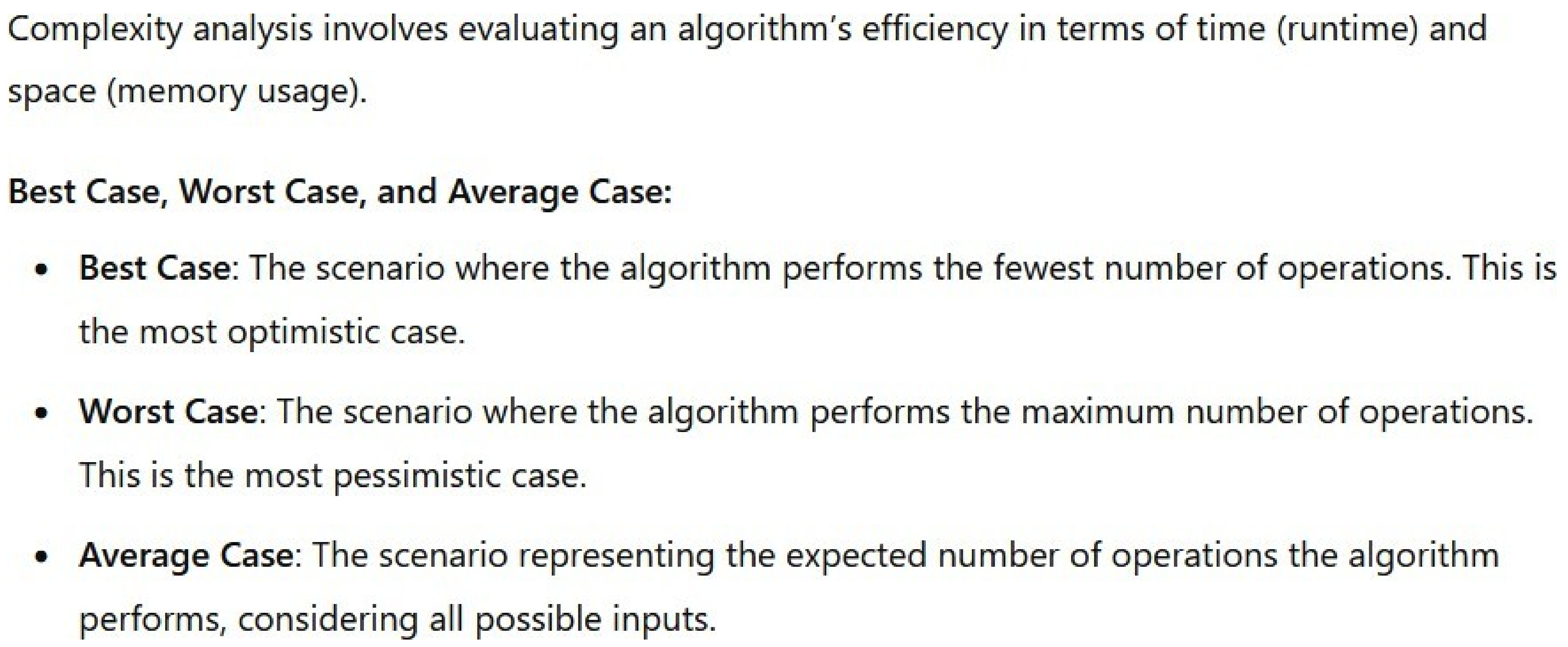
Example:

Consider two functions:

- $f(n) = 2n$
- $g(n) = n^2$

As n (input size) increases, $f(n)$ grows linearly, while $g(n)$ grows quadratically. For large values of n, $g(n)$ will always be much larger than $f(n)$, indicating that an algorithm with runtime n^2 will be slower than one with runtime $2n$ for large inputs.

Basic Complexity Analysis



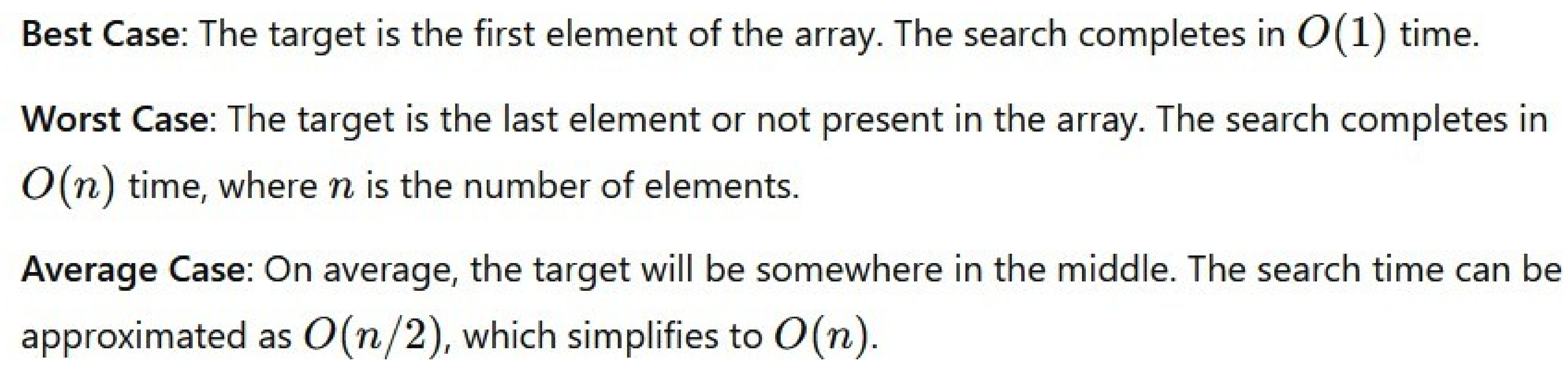
Complexity analysis involves evaluating an algorithm's efficiency in terms of time (runtime) and space (memory usage).

Best Case, Worst Case, and Average Case:

- **Best Case**: The scenario where the algorithm performs the fewest number of operations. This is the most optimistic case.
- **Worst Case**: The scenario where the algorithm performs the maximum number of operations. This is the most pessimistic case.
- **Average Case**: The scenario representing the expected number of operations the algorithm performs, considering all possible inputs.

Consider a simple linear search algorithm to find an element in an array.

```
def linear_search(arr,
  target): for i in
  range(len(arr)):
     if arr[i] == target:
       return
i return -1
```



- **Best Case**: The target is the first element of the array. The search completes in $O(1)$ time.
- **Worst Case**: The target is the last element or not present in the array. The search completes in $O(n)$ time, where n is the number of elements.
- **Average Case**: On average, the target will be somewhere in the middle. The search time can be approximated as $O(n/2)$, which simplifies to $O(n)$.

Asymptotic Analysis

- ***Defining the exact runtime function T(n) is challenging because it depends on various factors (machine speed, programming language, etc.).***
- Instead, ***we use asymptotic analysis, which focuses on how the runtime grows asn becomes large.***
- We’re interested in the dominant term that affects the runtime the most.

Asymptotic Notations

Asymptotic notations describe the running time of an algorithm as a function of the input size, providing a way to compare the efficiency of different algorithms. The most common notations are:

Big O Notation (O):

Omega Notation (Ω):

Theta Notation (Θ):

- **Big O Notation (O)**: Describes the upper bound of an algorithm's running time. It gives the worst-case scenario.

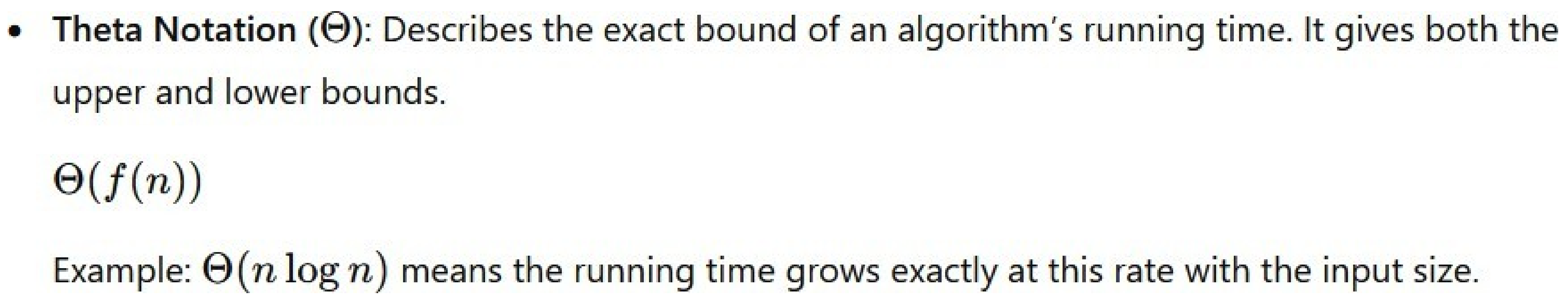
  $O(f(n))$

  Example: $O(n^2)$ means the running time grows at most quadratically with the input size.

- **Omega Notation (Ω)**: Describes the lower bound of an algorithm's running time. It gives the best-case scenario.

  $\Omega(f(n))$

  Example: $\Omega(n)$ means the running time grows at least linearly with the input size.



- **Theta Notation (Θ)**: Describes the exact bound of an algorithm's running time. It gives both the upper and lower bounds.

  $\Theta(f(n))$

  Example: $\Theta(n \log n)$ means the running time grows exactly at this rate with the input size.

Theta notation (Θ) provides a tight bound on the running time of an algorithm. ***It means that the running time grows at the same rate as the given function both in the upper and lower bounds.***

- **Theta Notation (Θ)**: Provides a tight bound (both upper and lower) on the running time of an algorithm.
- **Example (Insertion Sort)**: The insertion sort algorithm has a time complexity of $\Theta(n^2)$ because its performance can be tightly bound by the function n^2 in both best and worst cases.

This tight bound gives a precise characterization of the algorithm's performance, representing its running time in both the best and worst scenarios.

Find the minimum element in an array of (n) integers.

```
function findMin(array):
    min = array[0]
     for i from 1 to n-1:
        if array[i] < min:
            min = array[i]
    return min
```

1. **Identify Basic Operations**:

- Initialization: min = array[0] - (1 operation)
- Looping through the array: for i from 1 to n-1 - (n-1) operations
- Comparison: if array[i] < min - (n-1) operations
- Assignment: min = array[i] - (at most (n-1) operations

2. Count the Number of Operations

Total operations = 1 (initialization) + (n-1) loop iterations + (n-1)comparisons + at most (n-1)assignments

Sum = (1 + (n-1) + (n-1) + (n-1) = 3n - 2)

Sum = 3n-2

3. ***Simplify Using Big O Notation:***

The dominant term is (n).

Time Complexity O(n)

Example 2: Checking if an Array is Sorted

Check if an array of (n) integers is sorted in ascending order.

```
function isSorted(array):
    for i from 0 to n-2:
        if array[i] > array[i+1]:
            return false
    return true
```

1. Identify Basic Operations:

• Looping through the array: for i from 0 to n-2	(n-1) operations
• Comparison: if array[i] > array[i+1]	(n-1) operations
• Return statement: return false	(executed at most once)//negligible

2. Count the Number of Operations:

Total operations = (n-1) (loop iterations) + (n-1) (comparisons)

3. Simplify Using Big O Notation:

- Sum = (n-1) + (n-1) = 2(n-1)
- The dominant term is (n).
- Time Complexity: O(n)

Example 4: Counting the Number of Elements Greater Than a Given Value

Problem : Count the number of elements in an array that are greater than a given value (x).

```
function countGreaterThan(array, x):
    count = 0
    for each element in array:
        if element > x:
            count = count + 1
    return count
```

1. Identify Basic Operations:

- Initialization: count = 0 (1 operation)
- Looping through the array: for each element in array (n) operations
- Comparison: if element > x (n) operations
- Increment: count = count + 1 at most (n) operations
- Return statement: return count (1 operation)

2. Count the Number of Operations:

Total operations = 1 (initialization) + (n) (loop iterations) +(n) (comparisons) + at most (n) (increments) + 1 (return)

Sum = (1 + n + n + n + 1 = 3n + 2)

3. Simplify Using Big O Notation:

The dominant term is (n).

Time Complexity: O(n)

Find the Complexity

```
def print_all_pairs(arr):
    for i in arr:
      for j in arr:
          print(i,
          j)
```

Basic Operations:

1. Outer loop iterating through each element of the array.
2. Inner loop iterating through each element of the array.
3. Printing each pair of elements.

Counting Basic Operations:

1. The outer loop runs n times for an array of size n.
2. The inner loop runs n times for each iteration of the outer loop, resulting in $n \times n = n^2$ iterations.
3. The print operation is performed once for each pair, resulting in n^2 print operations.

Time Complexity:

- **O(n^2)**: The time complexity is quadratic because the number of operations grows as the square of the size of the array.

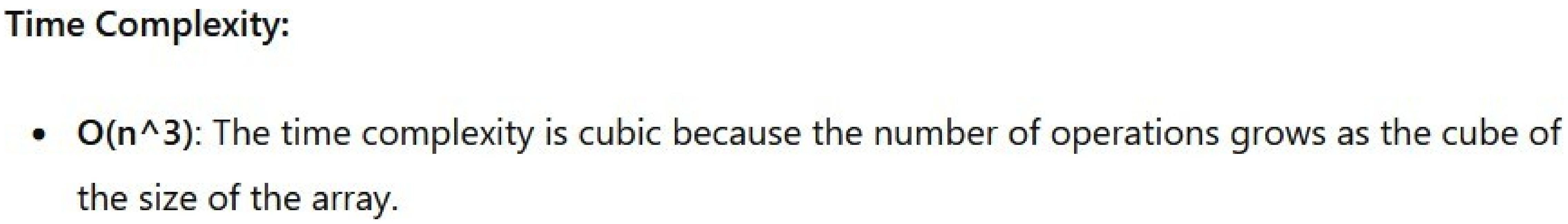
```
def print_all_triplets(arr):
    for i in arr:
      for j in arr:
          for k in arr:
          print(i, j, k)
```

Basic Operations:

1. Outer loop iterating through each element of the array.
2. Middle loop iterating through each element of the array.
3. Inner loop iterating through each element of the array.
4. Printing each triplet of elements.

Counting Basic Operations:

1. The outer loop runs n times for an array of size n.
2. The middle loop runs n times for each iteration of the outer loop.
3. The inner loop runs n times for each iteration of the middle loop, resulting in $n \times n \times n = n^3$ iterations.
4. The print operation is performed once for each triplet, resulting in n^3 print operations.

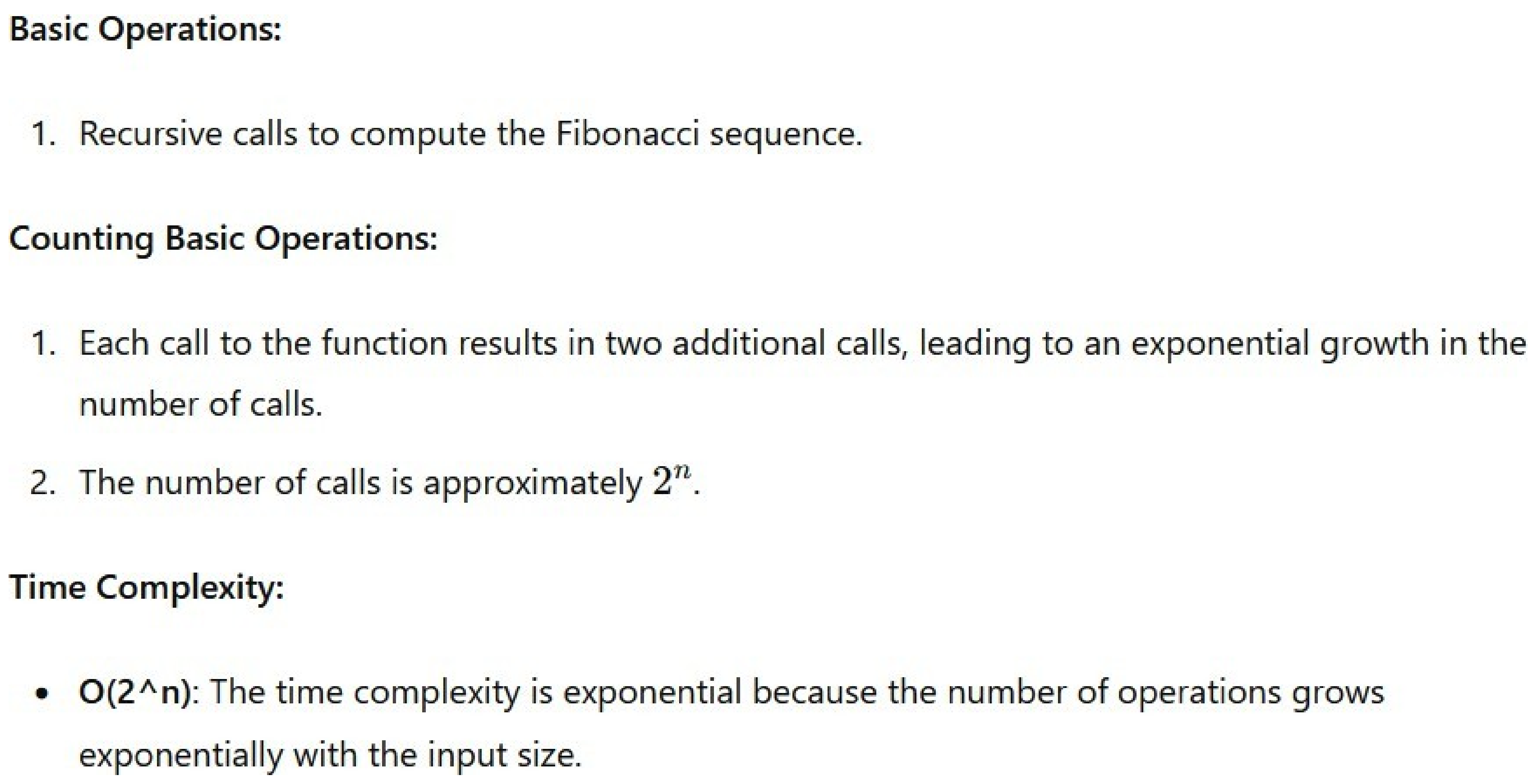


Time Complexity:

- **O(n^3)**: The time complexity is cubic because the number of operations grows as the cube of the size of the array.

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

Exponential Time Complexity (O(2^n))



Basic Operations:

1. Recursive calls to compute the Fibonacci sequence.

Counting Basic Operations:

1. Each call to the function results in two additional calls, leading to an exponential growth in the number of calls.
2. The number of calls is approximately 2^n.

Time Complexity:

- **O(2^n)**: The time complexity is exponential because the number of operations grows exponentially with the input size.

Asymptotic notation: formal defenitions

Big O notation is used to describe the upper bound of the time complexity of an algorithm. It provides a way to express how the runtime of an algorithm grows relative to the input size in the worst-case scenario.

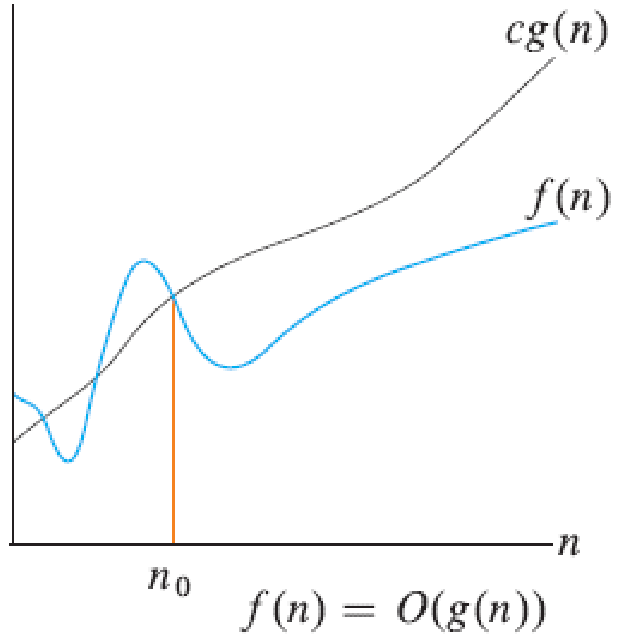


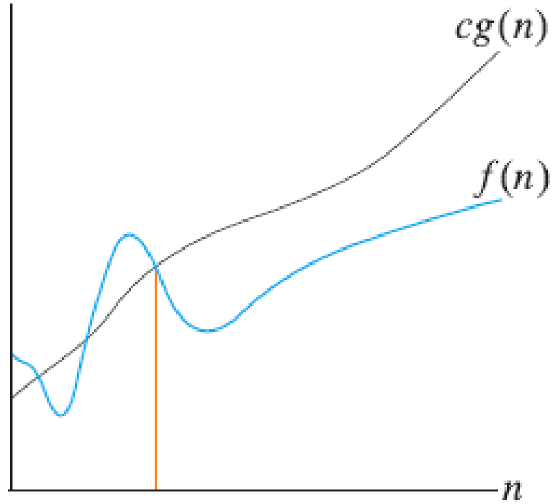
Figure 3.2 a

An algorithm is said to have a time complexity of $O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geq n_0$:

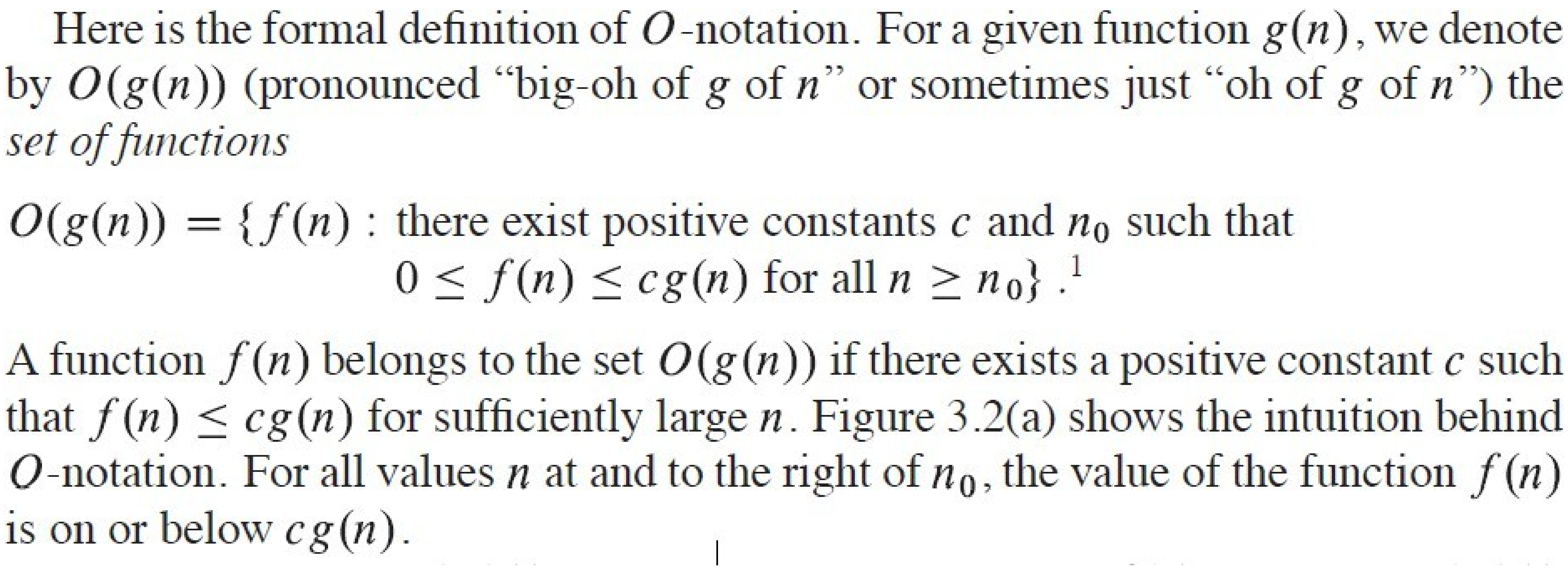
$$f(n) \leq c \cdot g(n)$$

where:

- $f(n)$ is the actual running time of the algorithm for an input of size n.
- $g(n)$ is a function that describes an upper bound on $f(n)$.
- c and n_0 are constants.



$f(n) = O(g(n))$



Here is the formal definition of O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the *set of functions*

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .^1$$

A function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that $f(n) \le cg(n)$ for sufficiently large n. Figure 3.2(a) shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

A formal definition from the syllabus text

Find the constants *C* and *n0* for ***f(n) = 5n+3***

Goal: Show that $f(n)$ is $O(n)$.

f(n) =O(?)

5n+3 <= c.g(n)

1. **Identify the dominant term:**
   - The dominant term is $5n$.
2. **Simplify by ignoring constants and lower order terms:**
   - Approximate $5n + 3$ by n.

3. Apply the formal definition:

- We need to find constants c and n_0 such that $5n + 3 \leq c \cdot n$ for all $n \geq n_0$.

5n +3 <= c. n

apply diff values for c , c=1, c=2, c=3, ….

- Choose $c = 6$.
- For $n \geq 1$, we have:

  $5n + 3 \leq 6n$

- This inequality holds because $3 \leq n$ for $n \geq 3$.

Thus, $f(n) = 5n + 3$ is $O(n)$ with $c = 6$ and $n_0 = 3$.

Example 1: Linear Function

Function: $f(n) = 5n + 3$

Goal: Show that $f(n)$ is $O(n)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $5n$.
2. **Simplify**:
   - Approximate $5n + 3$ by n.
3. **Form the Inequality**:
   - We need $f(n) \leq c \cdot g(n)$.
   - $5n + 3 \leq c \cdot n$.
4. **Choose c**:
   - Let's choose $c = 6$.
   - Then we need $5n + 3 \leq 6n$.
5. **Solve for n**:
   - Rearrange the inequality: $5n + 3 \leq 6n$.
   - $3 \leq n$.
   - Thus, $n \geq 3$.
6. **Determine n_0**:
   - Here, $n_0 = 3$.

So, $f(n) = 5n + 3$ is $O(n)$ with $c = 6$ and $n_0 = 3$.

Example 2: Quadratic Function

Function: $f(n) = 2n^2 + 4n + 8$

Goal: Show that $f(n)$ is $O(n^2)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $2n^2$.
2. **Simplify**:
   - Approximate $2n^2 + 4n + 8$ by n^2.
3. **Form the Inequality**:
   - We need $f(n) \leq c \cdot g(n)$.
   - $2n^2 + 4n + 8 \leq c \cdot n^2$.
4. **Choose c**:
   - Let's choose $c = 3$.
   - Then we need $2n^2 + 4n + 8 \leq 3n^2$.
5. **Solve for n**:
   - Rearrange the inequality: $2n^2 + 4n + 8 \leq 3n^2$.
   - $4n + 8 \leq n^2$.
   - Rearrange: $n^2 - 4n - 8 \geq 0$.

6. **Solve the Quadratic Inequality:**
   - Factor or use the quadratic formula: $n = \frac{4 \pm \sqrt{16+32}}{2} = \frac{4 \pm \sqrt{48}}{2} = \frac{4 \pm 4\sqrt{3}}{2} = 2 \pm 2\sqrt{3}$.
   - Approximate: $n \approx 5.46$.
7. **Determine n_0:**
   - n_0 should be greater than the largest root, so $n_0 = 6$.

So, $f(n) = 2n^2 + 4n + 8$ is $O(n^2)$ with $c = 3$ and $n_0 = 6$.

Example 3: Exponential Function

Function: $f(n) = 3 \cdot 2^n + 4 \cdot 2^n$

Goal: Show that $f(n)$ is $O(2^n)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $3 \cdot 2^n$.
2. **Simplify**:
   - Approximate $3 \cdot 2^n + 4 \cdot 2^n$ by 2^n.
3. **Form the Inequality**:
   - We need $f(n) \leq c \cdot g(n)$.
   - $3 \cdot 2^n + 4 \cdot 2^n \leq c \cdot 2^n$.
4. **Choose c**:
   - Let's choose $c = 7$.
   - Then we need $3 \cdot 2^n + 4 \cdot 2^n \leq 7 \cdot 2^n$.
5. **Solve for n**:
   - Combine like terms: $7 \cdot 2^n \leq 7 \cdot 2^n$.

6. **Determine n_0:**

- This inequality holds for all $n \geq 1$.

So, $f(n) = 3 \cdot 2^n + 4 \cdot 2^n$ is $O(2^n)$ with $c = 7$ and $n_0 = 1$.

Example 4: Logarithmic Function

Function: $f(n) = \log(n) + 5$

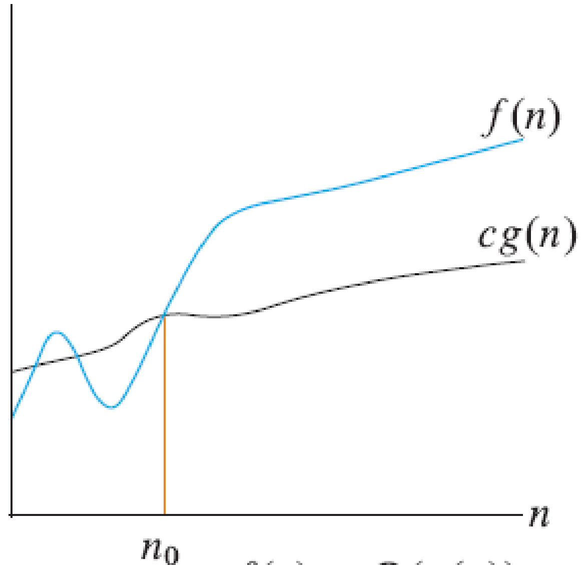
Goal: Show that $f(n)$ is $O(\log(n))$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $\log(n)$.
2. **Simplify**:
   - Approximate $\log(n) + 5$ by $\log(n)$.
3. **Form the Inequality**:
   - We need $f(n) \leq c \cdot g(n)$.
   - $\log(n) + 5 \leq c \cdot \log(n)$.

4. **Choose c:**
   - Let's choose $c = 2$.
   - Then we need $\log(n) + 5 \leq 2\log(n)$.
5. **Solve for n:**
   - Rearrange the inequality: $5 \leq \log(n)$.
   - Exponentiate both sides: $n \geq 2^5 = 32$.
6. **Determine n_0:**
   - Here, $n_0 = 32$.

So, $f(n) = \log(n) + 5$ is $O(\log(n))$ with $c = 2$ and $n_0 = 32$.



$f(n) = \Omega(g(n))$

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

Asymptotic Notation: Formal Definition of Big Omega (Ω)

Big Omega (Ω) notation provides a lower bound for the running time of an algorithm. It describes the best-case scenario, giving a way to express the minimum amount of time an algorithm will take to complete for large input sizes.

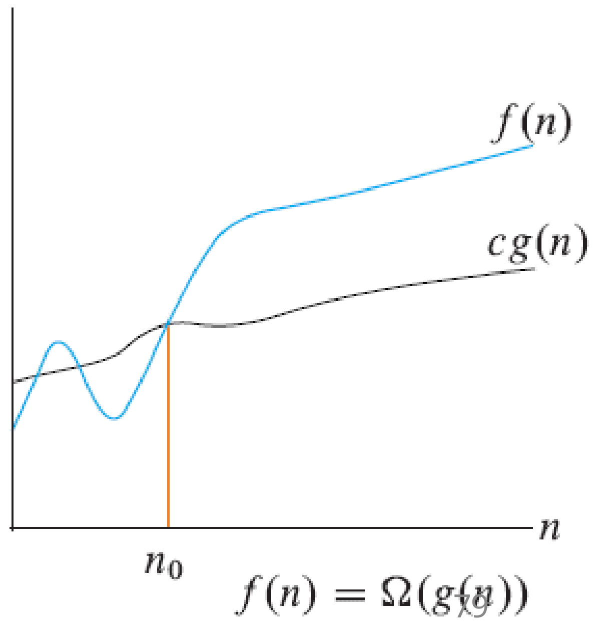
Formal Definition

An algorithm is said to have a time complexity of $\Omega(g(n))$ if there exist positive constants c and n_0 such that for all $n \geq n_0$:

$$f(n) \geq c \cdot g(n)$$

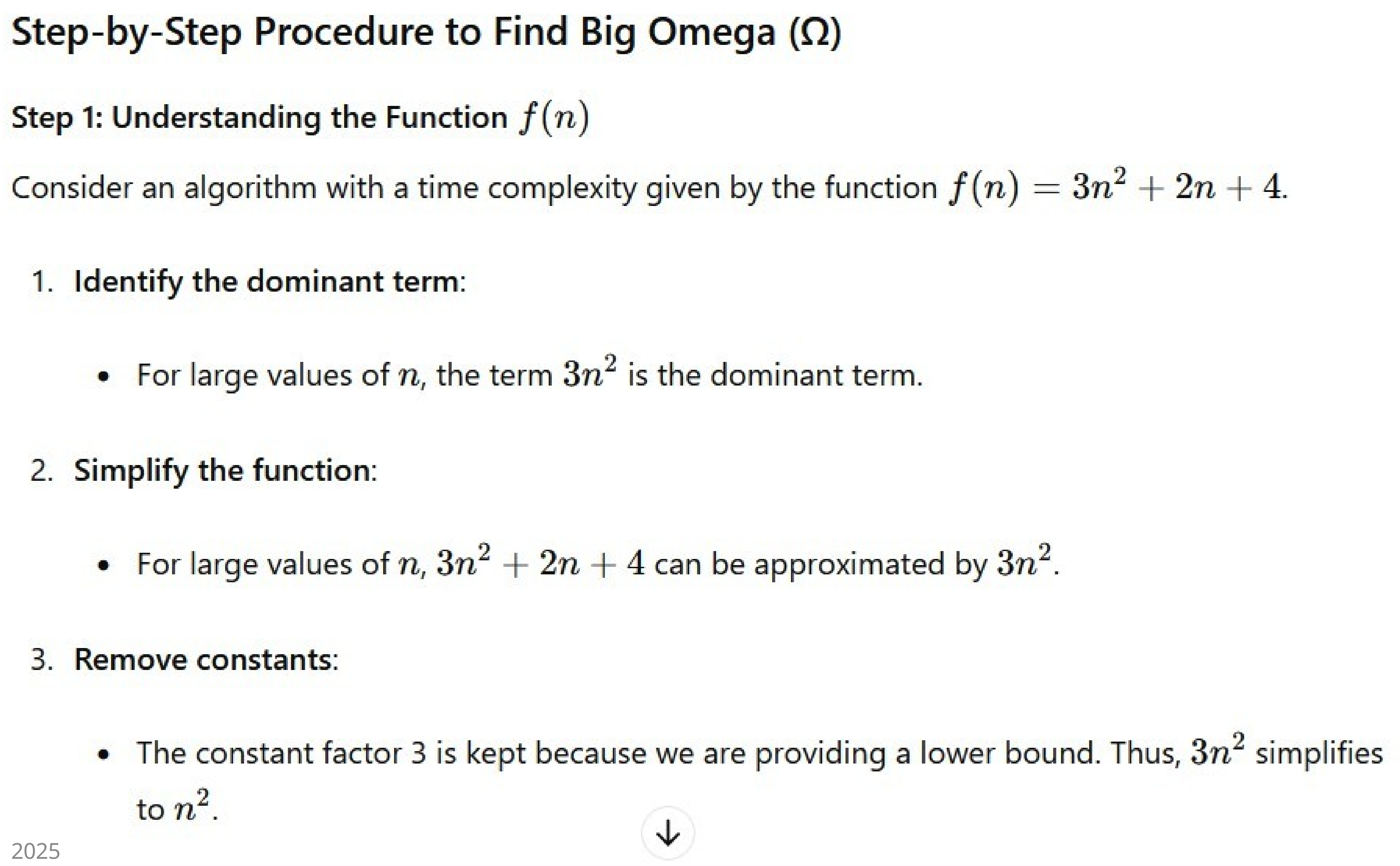
where:

- $f(n)$ is the actual running time of the algorithm for an input of size n.
- $g(n)$ is a function that describes a lower bound on $f(n)$.
- c and n_0 are constants.



In Omega we are considering the lower bound ***but we have to select the Greatest lower bound***

But in Big O we consider the least upper bound



Step-by-Step Procedure to Find Big Omega (Ω)

Step 1: Understanding the Function $f(n)$

Consider an algorithm with a time complexity given by the function $f(n) = 3n^2 + 2n + 4$.

1. **Identify the dominant term**:
   - For large values of n, the term $3n^2$ is the dominant term.
2. **Simplify the function**:
   - For large values of n, $3n^2 + 2n + 4$ can be approximated by $3n^2$.
3. **Remove constants**:
   - The constant factor 3 is kept because we are providing a lower bound. Thus, $3n^2$ simplifies to n^2.

4. **Big Omega notation:**

   - Therefore, $f(n)$ is $\Omega(n^2)$.

Step 2: Formal Definition Application

To apply the formal definition, let's find constants c and n_0 such that for all $n \geq n_0$:

$$f(n) = 3n^2 + 2n + 4 \geq c \cdot g(n) = c \cdot n^2$$

- Choose $c = 1$:
  - For $n \geq 1$, we have:

    $$3n^2 + 2n + 4 \geq 1n^2$$

  - This inequality holds because $3n^2$ dominates the other terms for sufficiently large n.

Thus, we can say that $f(n)$ is $\Omega(n^2)$.

Example 1: Linear Function

Function: $f(n) = 5n + 3$

Goal: Show that $f(n)$ is $\Omega(n)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $5n$.
2. **Simplify by ignoring constants and lower order terms**:
   - Approximate $5n + 3$ by $5n$.
3. **Form the Inequality**:
   - We need $f(n) \geq c \cdot g(n)$.
   - $5n + 3 \geq c \cdot n$.
4. **Choose c**:
   - Let's choose $c = 5$.
   - Then we need $5n + 3 \geq 5n$.

We have to choose the greatest lower bound that Is the closest.

C=1, 2,3,4,5 will satisfy, select 5

5. **Solve for n:**

   - The inequality $5n + 3 \geq 5n$ is true for all $n \geq 0$.

6. **Determine n_0:**

   - Here, $n_0 = 0$.

So, $f(n) = 5n + 3$ is $\Omega(n)$ with $c = 5$ and $n_0 = 0$.

Example 2: Quadratic Function

Function: $f(n) = 2n^2 + 4n + 8$

Goal: Show that $f(n)$ is $\Omega(n^2)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $2n^2$.
2. **Simplify by ignoring constants and lower order terms**:
   - Approximate $2n^2 + 4n + 8$ by $2n^2$.
3. **Form the Inequality**:
   - We need $f(n) \geq c \cdot g(n)$.
   - $2n^2 + 4n + 8 \geq c \cdot n^2$.
4. **Choose c**:
   - Let's choose $c = 2$.
   - Then we need $2n^2 + 4n + 8 \geq 2n^2$.

5. **Solve for n:**

   - The inequality $2n^2 + 4n + 8 \geq 2n^2$ is true for all $n \geq 0$.

6. **Determine n_0:**

   - Here, $n_0 = 0$.

So, $f(n) = 2n^2 + 4n + 8$ is $\Omega(n^2)$ with $c = 2$ and $n_0 = 0$.

Example 3: Cubic Function

Function: $f(n) = 6n^3 + 5n^2 + 4n + 2$

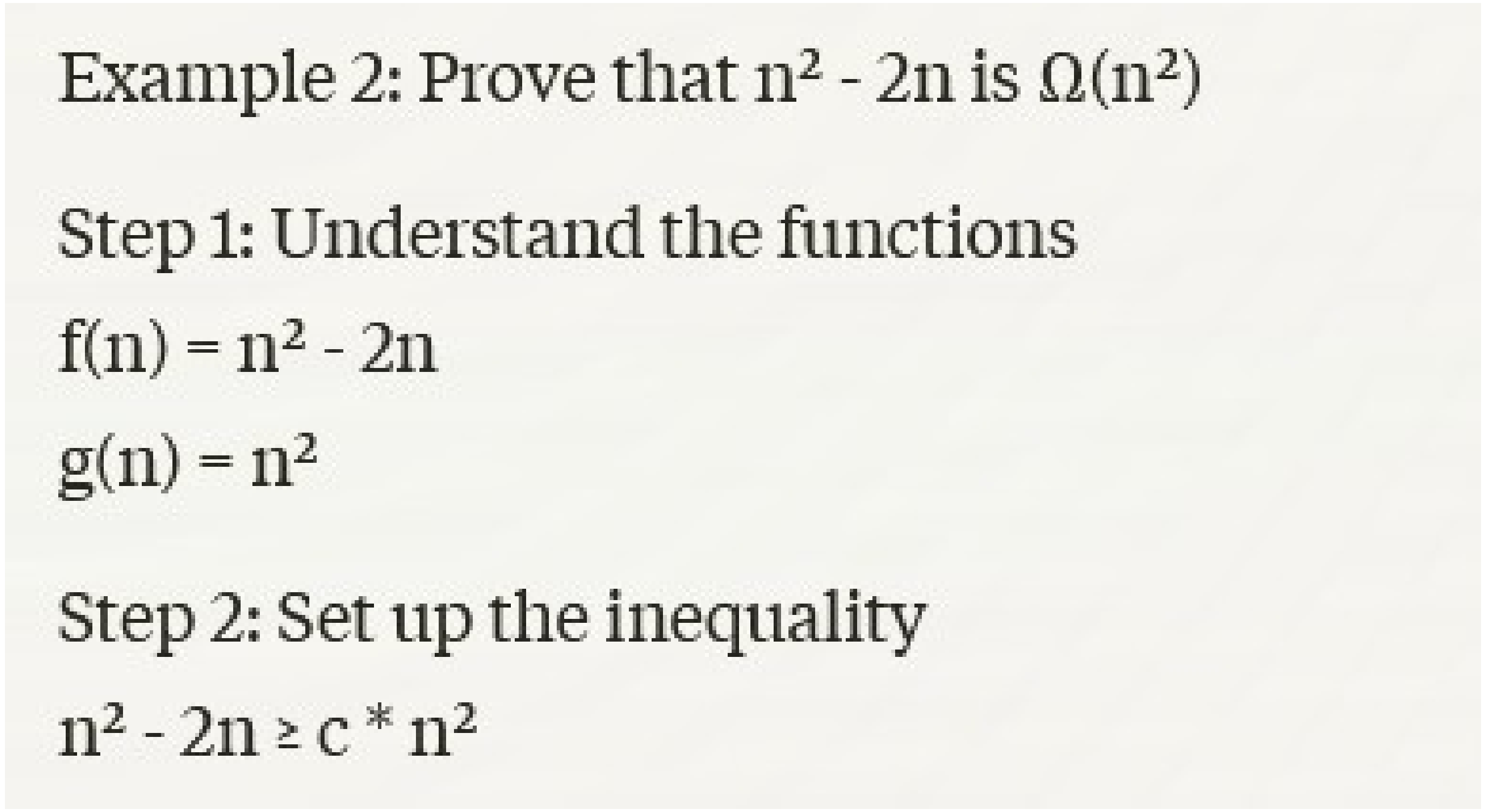
Goal: Show that $f(n)$ is $\Omega(n^3)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
   - The dominant term is $6n^3$.
2. **Simplify by ignoring constants and lower order terms**:
   - Approximate $6n^3 + 5n^2 + 4n + 2$ by $6n^3$.
3. **Form the Inequality**:
   - We need $f(n) \geq c \cdot g(n)$.
   - $6n^3 + 5n^2 + 4n + 2 \geq c \cdot n^3$.

4. **Choose c:**
   - Let's choose $c = 6$.
   - Then we need $6n^3 + 5n^2 + 4n + 2 \geq 6n^3$.
5. **Solve for n:**
   - The inequality $6n^3 + 5n^2 + 4n + 2 \geq 6n^3$ is true for all $n \geq 0$.
6. **Determine n_0:**
   - Here, $n_0 = 0$.

So, $f(n) = 6n^3 + 5n^2 + 4n + 2$ is $\Omega(n^3)$ with $c = 6$ and $n_0 = 0$.



Example 2: Prove that $n^2 - 2n$ is $\Omega(n^2)$

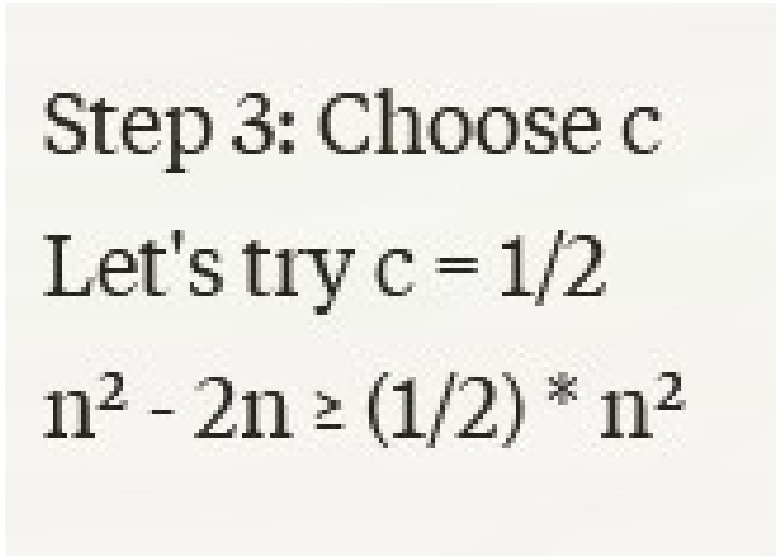
Step 1: Understand the functions

$f(n) = n^2 - 2n$

$g(n) = n^2$

Step 2: Set up the inequality

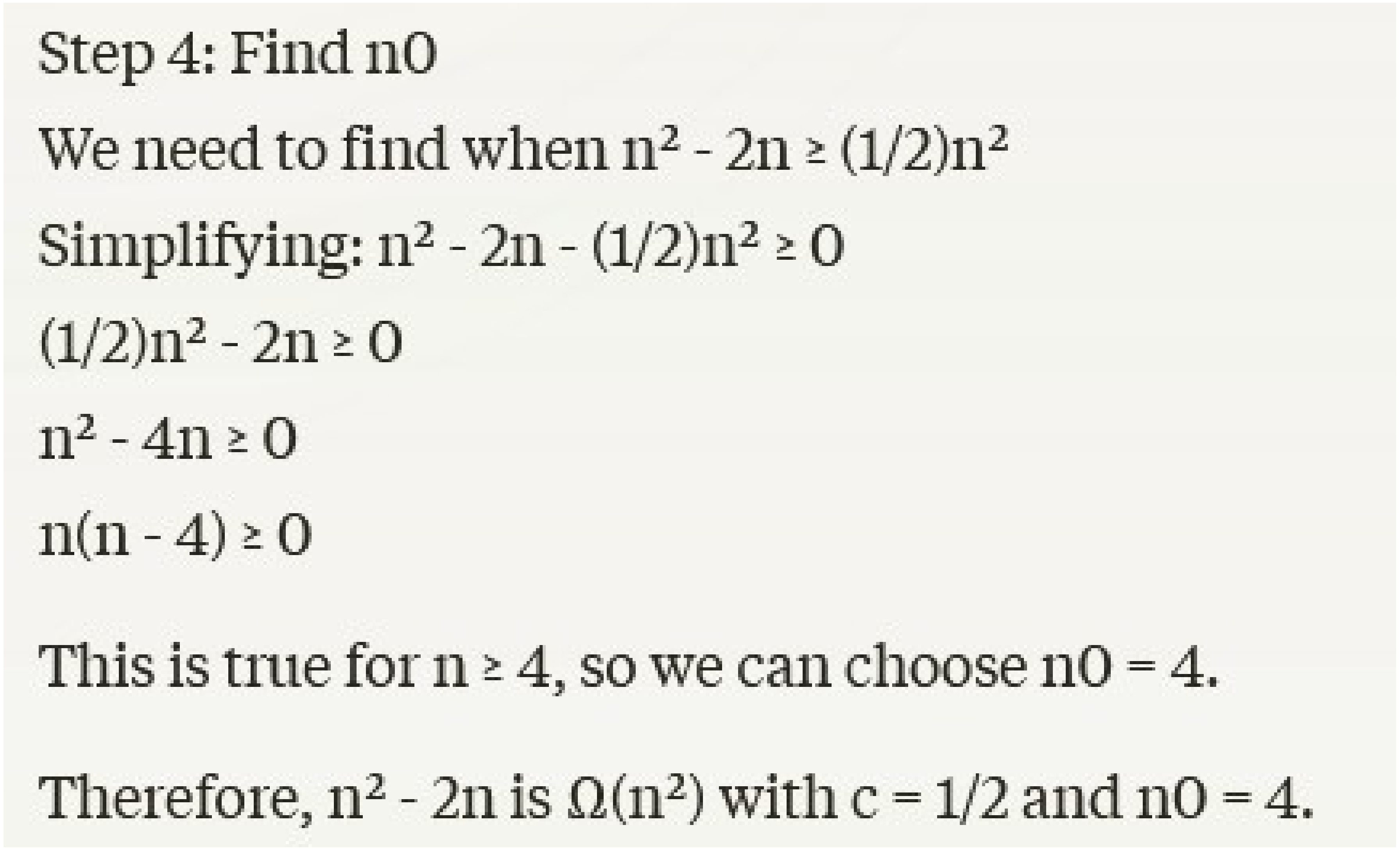
$n^2 - 2n \geq c * n^2$



Step 3: Choose c

Let's try c = 1/2

$n^2 - 2n \geq (1/2) * n^2$



Step 4: Find n0

We need to find when $n^2 - 2n \geq (1/2)n^2$

Simplifying: $n^2 - 2n - (1/2)n^2 \geq 0$

$(1/2)n^2 - 2n \geq 0$

$n^2 - 4n \geq 0$

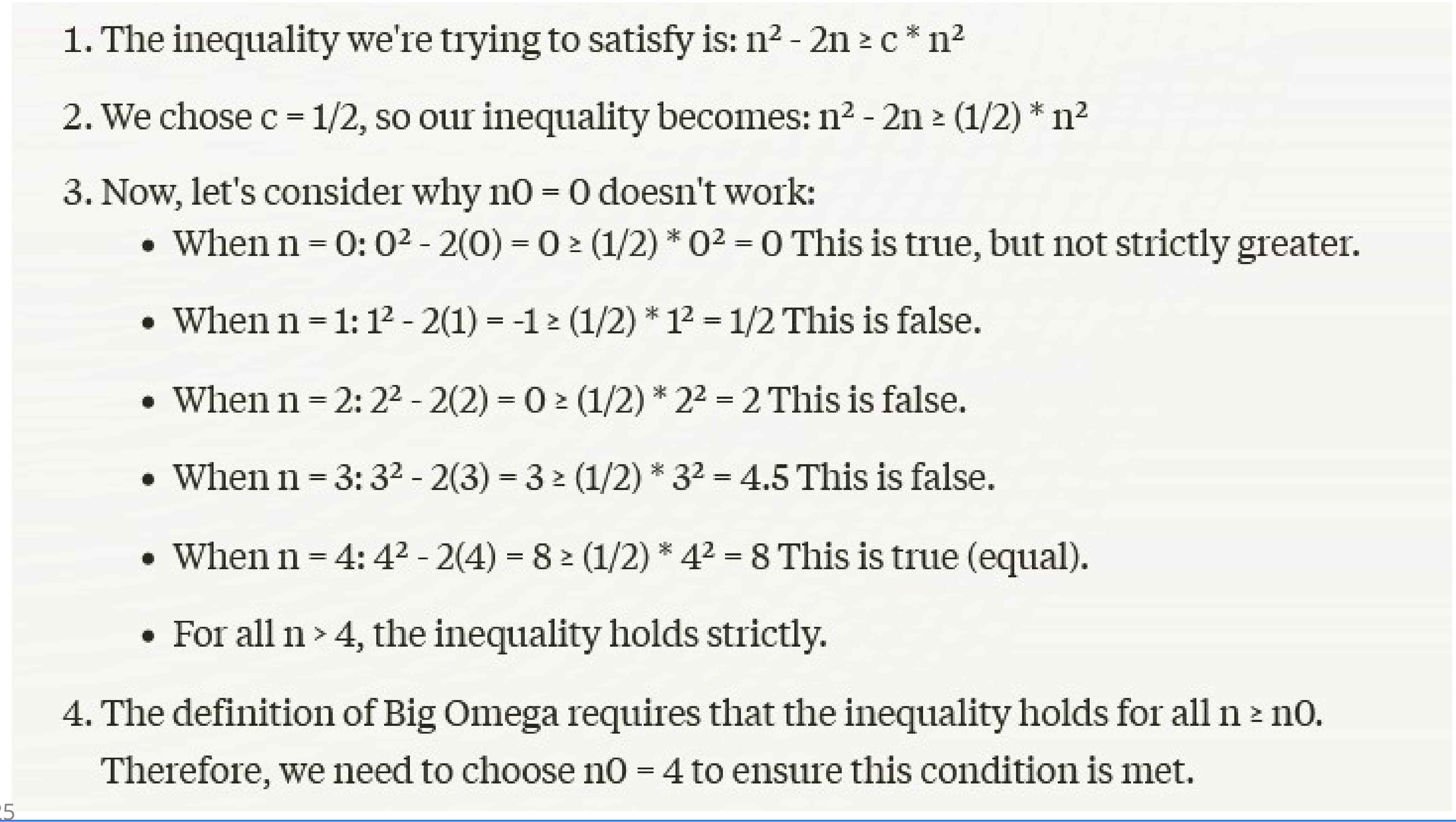
$n(n - 4) \geq 0$

This is true for $n \geq 4$, so we can choose n0 = 4.

Therefore, $n^2 - 2n$ is $\Omega(n^2)$ with c = 1/2 and n0 = 4.

n = 0 will also true for this case, then why choose n= 4 ?

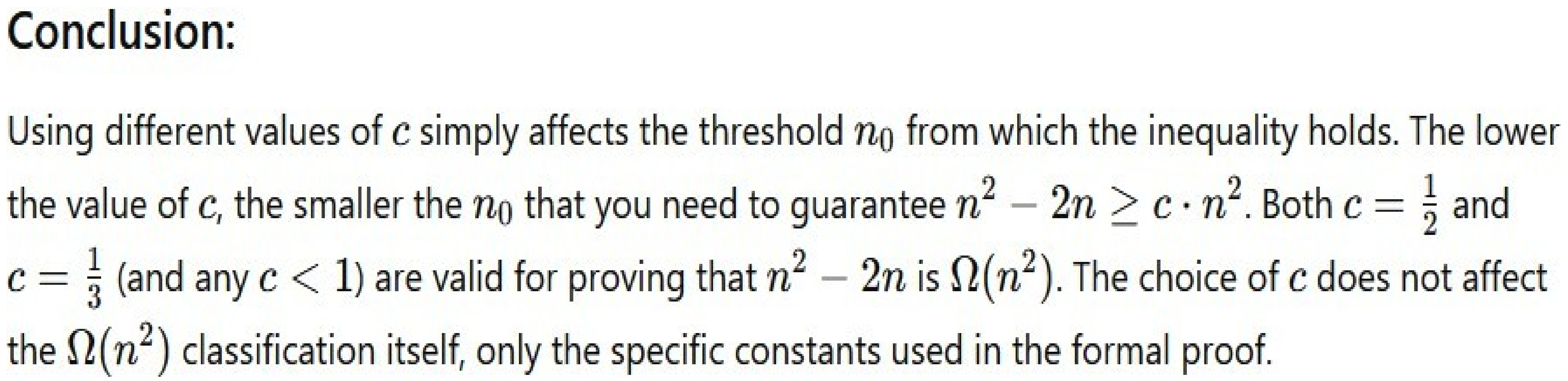
why n0 = 4 was chosen, and why n0 = 0 isn't actually correct in this case.



1. The inequality we're trying to satisfy is: $n^2 - 2n \geq c * n^2$
2. We chose $c = 1/2$, so our inequality becomes: $n^2 - 2n \geq (1/2) * n^2$
3. Now, let's consider why n0 = 0 doesn't work:
   - When $n = 0$: $0^2 - 2(0) = 0 \geq (1/2) * 0^2 = 0$ This is true, but not strictly greater.
   - When $n = 1$: $1^2 - 2(1) = -1 \geq (1/2) * 1^2 = 1/2$ This is false.
   - When $n = 2$: $2^2 - 2(2) = 0 \geq (1/2) * 2^2 = 2$ This is false.
   - When $n = 3$: $3^2 - 2(3) = 3 \geq (1/2) * 3^2 = 4.5$ This is false.
   - When $n = 4$: $4^2 - 2(4) = 8 \geq (1/2) * 4^2 = 8$ This is true (equal).
   - For all $n > 4$, the inequality holds strictly.
4. The definition of Big Omega requires that the inequality holds for all $n \geq n0$.

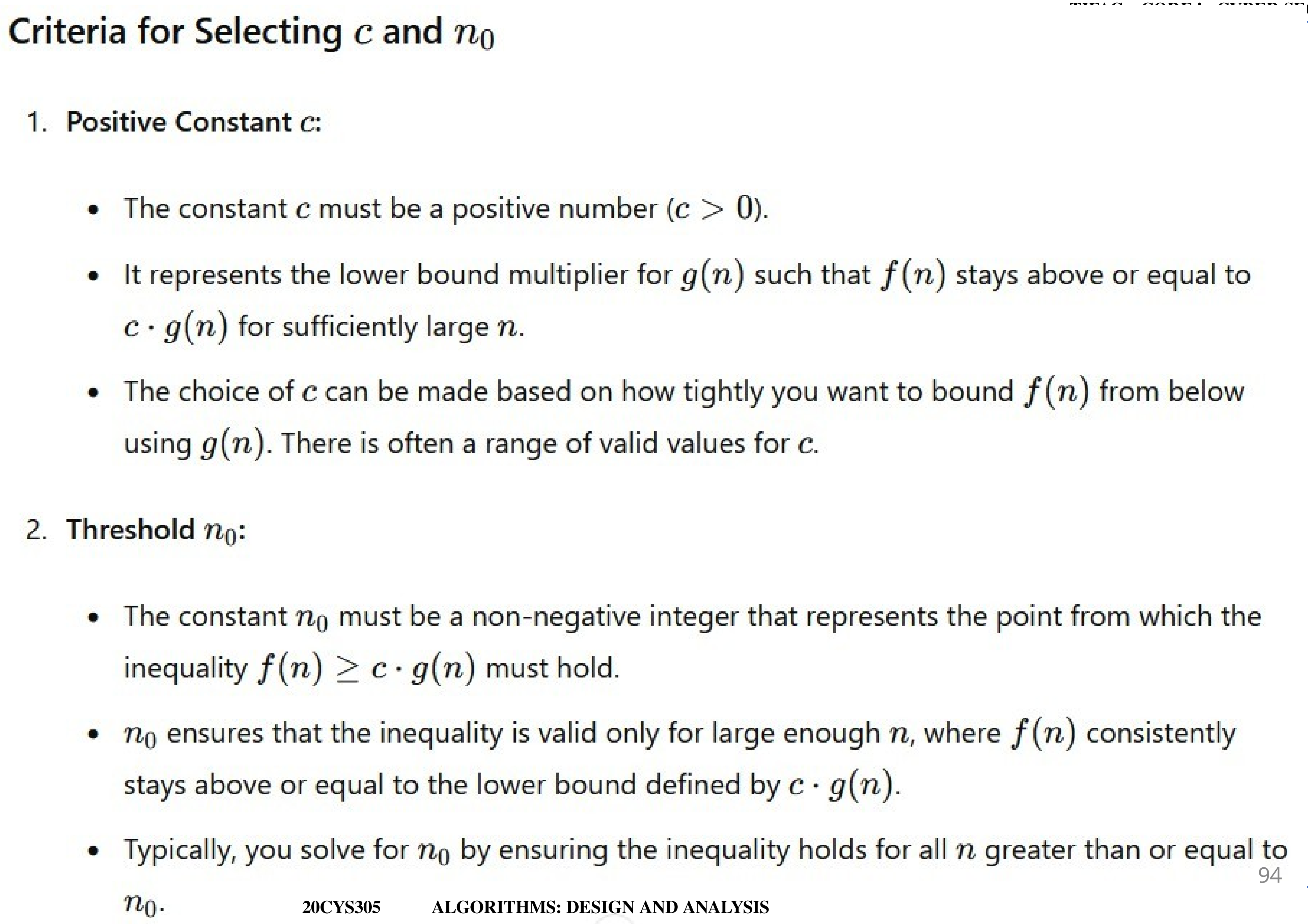
   Therefore, we need to choose n0 = 4 to ensure this condition is met.

It is right that the ***<u>inequality is technically true for n = 0, but it's not strictly greater</u>***, and it **fails for n = 1, 2, and 3**. ***In Big Omega notation, we need to find the point from which the inequality consistently holds for all larger values.***



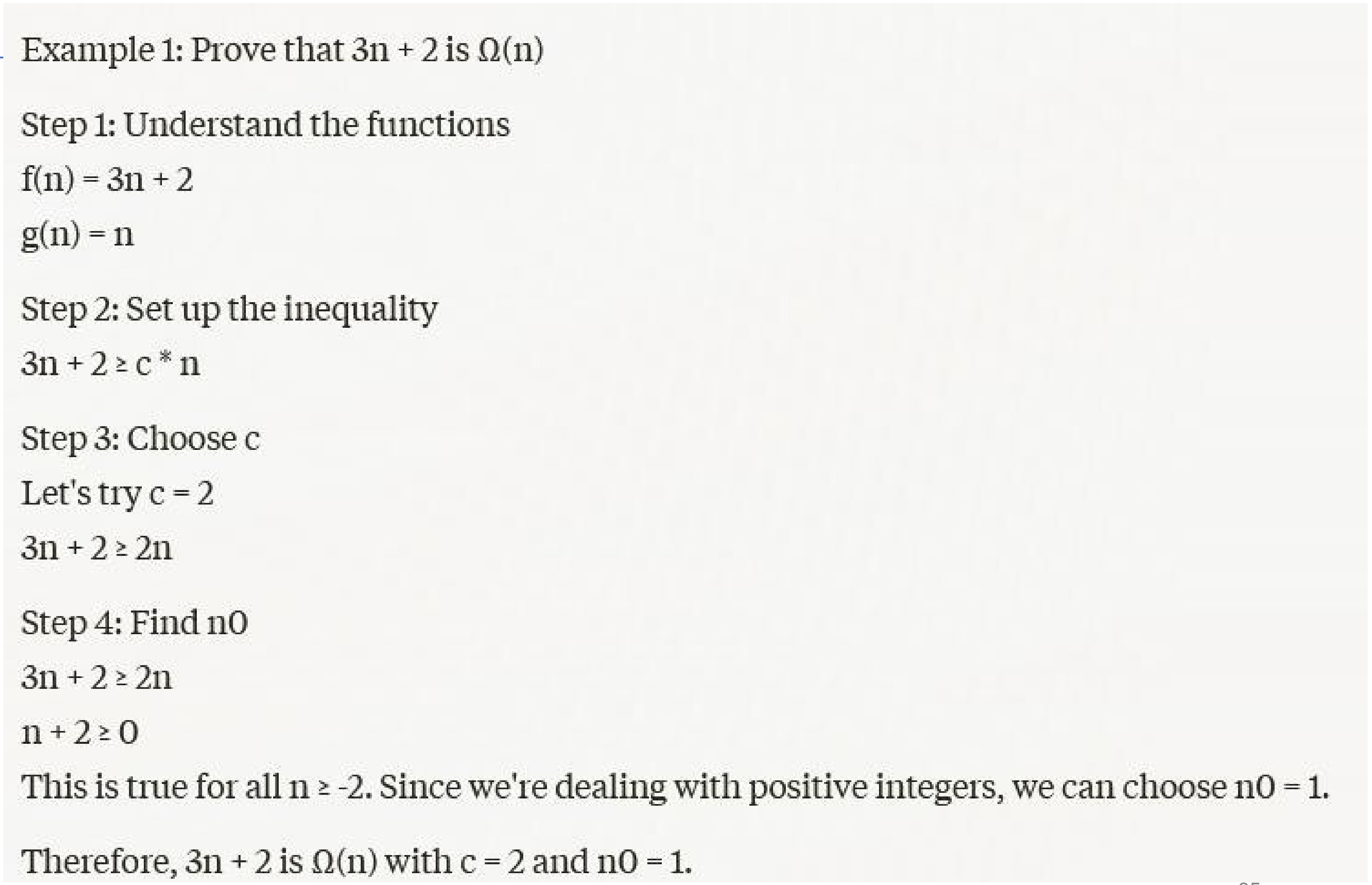
Conclusion:

Using different values of c simply affects the threshold n_0 from which the inequality holds. The lower the value of c, the smaller the n_0 that you need to guarantee $n^2 - 2n \geq c \cdot n^2$. Both $c = \frac{1}{2}$ and $c = \frac{1}{3}$ (and any $c < 1$) are valid for proving that $n^2 - 2n$ is $\Omega(n^2)$. The choice of c does not affect the $\Omega(n^2)$ classification itself, only the specific constants used in the formal proof.



Criteria for Selecting c and n_0

1. **Positive Constant c:**
   - The constant c must be a positive number ($c > 0$).
   - It represents the lower bound multiplier for $g(n)$ such that $f(n)$ stays above or equal to $c \cdot g(n)$ for sufficiently large n.
   - The choice of c can be made based on how tightly you want to bound $f(n)$ from below using $g(n)$. There is often a range of valid values for c.
2. **Threshold n_0:**
   - The constant n_0 must be a non-negative integer that represents the point from which the inequality $f(n) \geq c \cdot g(n)$ must hold.
   - n_0 ensures that the inequality is valid only for large enough n, where $f(n)$ consistently stays above or equal to the lower bound defined by $c \cdot g(n)$.
   - Typically, you solve for n_0 by ensuring the inequality holds for all n greater than or equal to n_0.



Example 1: Prove that $3n + 2$ is $\Omega(n)$

Step 1: Understand the functions

$f(n) = 3n + 2$

$g(n) = n$

Step 2: Set up the inequality

$3n + 2 \geq c * n$

Step 3: Choose c

Let's try $c = 2$

$3n + 2 \geq 2n$

Step 4: Find n0

$3n + 2 \geq 2n$

$n + 2 \geq 0$

This is true for all $n \geq -2$. Since we're dealing with positive integers, we can choose $n0 = 1$.

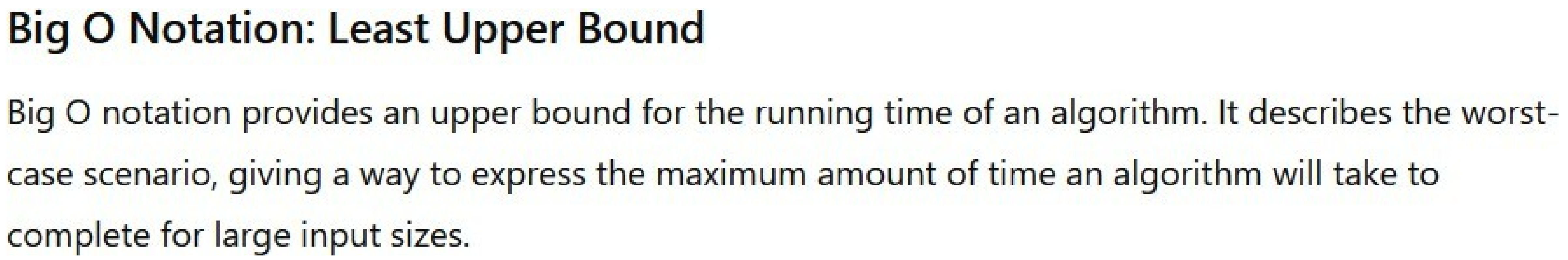
Therefore, $3n + 2$ is $\Omega(n)$ with $c = 2$ and $n0 = 1$.

Key Points in Finding n_0:

- **Ensure Non-Trivial Bounds**: n_0 should be chosen so that the bound $c \cdot g(n)$ is non-trivial. This means $f(n)$ should actually grow significantly larger than $c \cdot g(n)$ for all $n \geq n_0$.
- **Choose a Reasonable n_0**: While mathematically, n_0 could be very small, in practical algorithm analysis, n_0 should reflect a realistic starting point where the comparison makes sense.
- **Check the Entire Range $n \geq n_0$**: Always verify the inequality holds over the entire range of n starting from n_0.

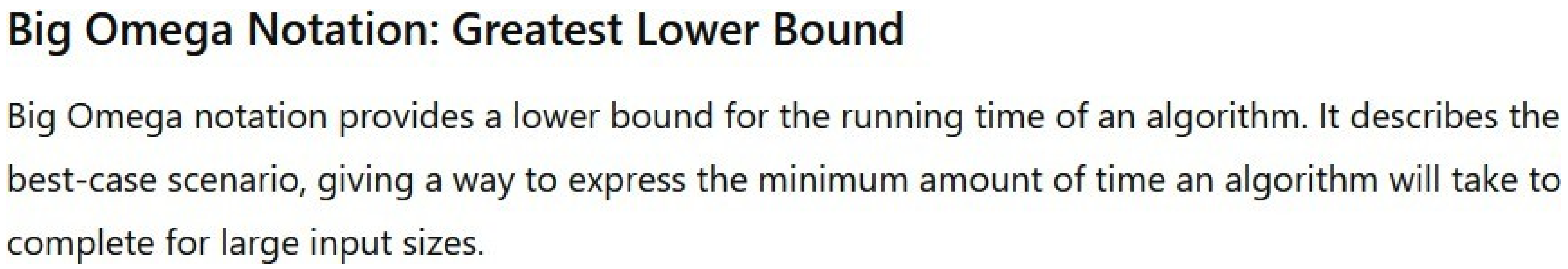
Understanding Upper and Lower Bounds in Asymptotic Notation

In Big O notation, we are interested in the least upper bound of the growth rate of a function, while in Big Omega notation, we are concerned with the greatest lower bound.



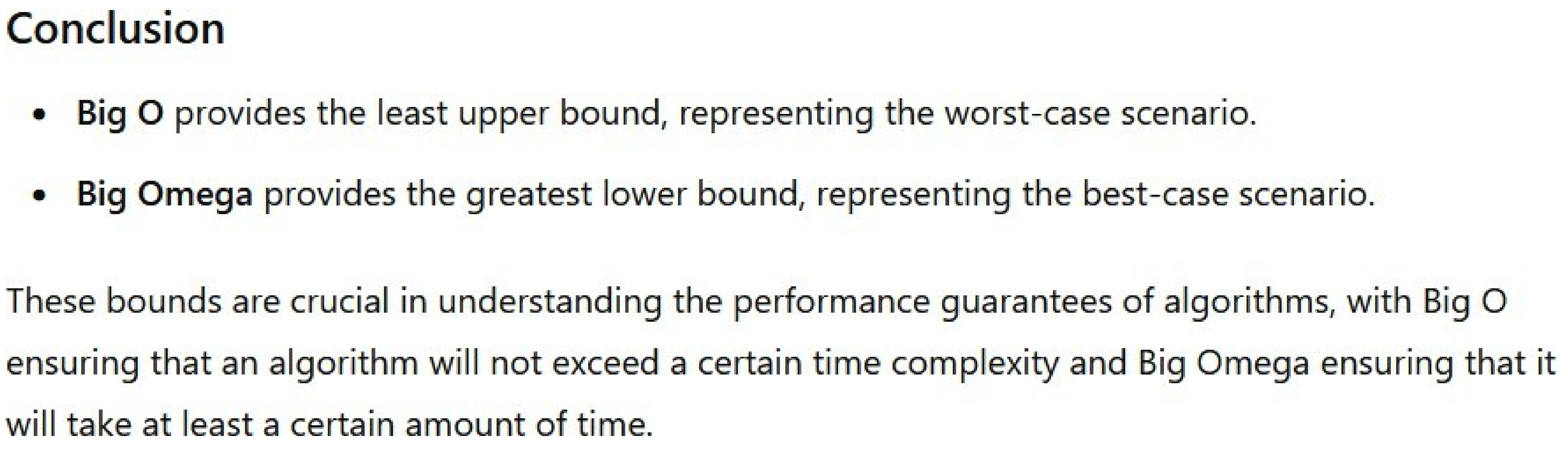
Big O Notation: Least Upper Bound

Big O notation provides an upper bound for the running time of an algorithm. It describes the worst-case scenario, giving a way to express the maximum amount of time an algorithm will take to complete for large input sizes.



Big Omega Notation: Greatest Lower Bound

Big Omega notation provides a lower bound for the running time of an algorithm. It describes the best-case scenario, giving a way to express the minimum amount of time an algorithm will take to complete for large input sizes.



Conclusion

- **Big O** provides the least upper bound, representing the worst-case scenario.
- **Big Omega** provides the greatest lower bound, representing the best-case scenario.

These bounds are crucial in understanding the performance guarantees of algorithms, with Big O ensuring that an algorithm will not exceed a certain time complexity and Big Omega ensuring that it will take at least a certain amount of time.

Justification with Greatest Lower Bound and Least Upper Bound

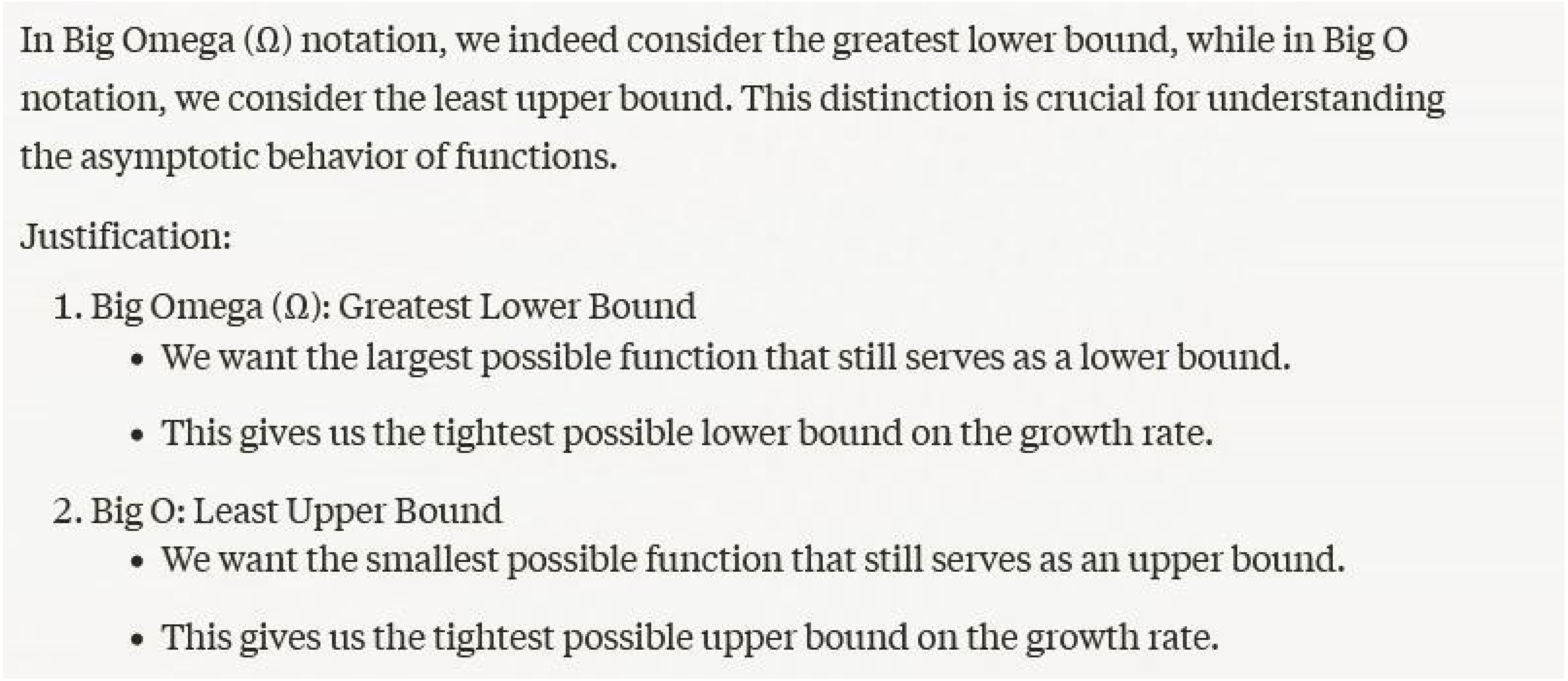
Big O Notation:

- In Big O notation, we look for the smallest function $g(n)$ that is an upper bound for $f(n)$. This ensures that $f(n)$ does not grow faster than $g(n)$.
- For $f(n) = 3n^2 + 2n + 1$, we found that n^2 is the least upper bound because no smaller function can guarantee the inequality $f(n) \leq c \cdot g(n)$.

Big Omega Notation:

- In Big Omega notation, we look for the largest function $g(n)$ that is a lower bound for $f(n)$. This ensures that $f(n)$ does not grow slower than $g(n)$.
- For $f(n) = 3n^2 + 2n + 1$, we found that n^2 is the greatest lower bound because no larger function can guarantee the inequality $f(n) \geq c \cdot g(n)$.

Another Explanation



In Big Omega (Ω) notation, we indeed consider the greatest lower bound, while in Big O notation, we consider the least upper bound. This distinction is crucial for understanding the asymptotic behavior of functions.

Justification:

1. Big Omega (Ω): Greatest Lower Bound
   - We want the largest possible function that still serves as a lower bound.
   - This gives us the tightest possible lower bound on the growth rate.
2. Big O: Least Upper Bound
   - We want the smallest possible function that still serves as an upper bound.
   - This gives us the tightest possible upper bound on the growth rate.

Asymptotic Notation: Formal Definition of Big Theta (Θ)

Big Theta (Θ) notation provides a tight bound on the running time of an algorithm. It describes both the upper and lower bounds, giving a way to express the exact growth rate of the algorithm for large input sizes.

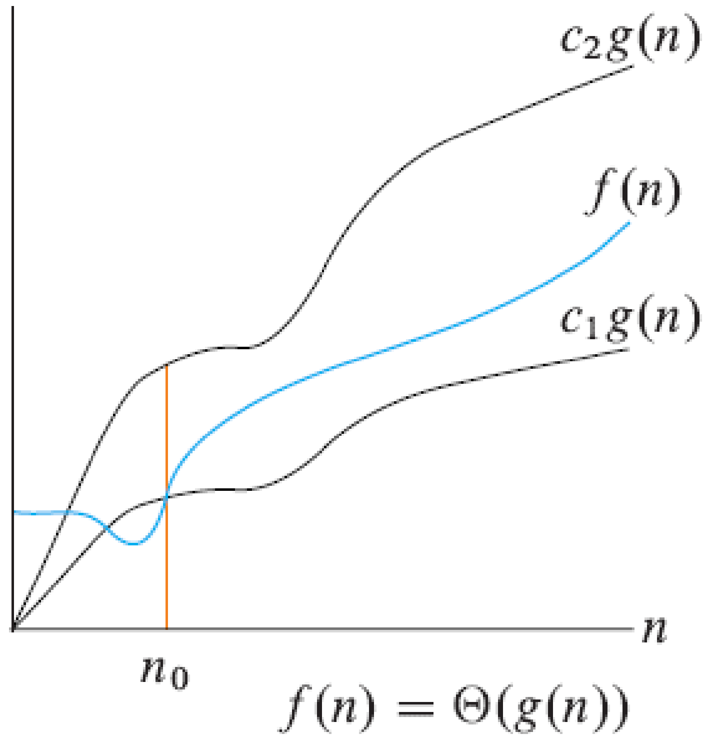


Figure 3.2
C

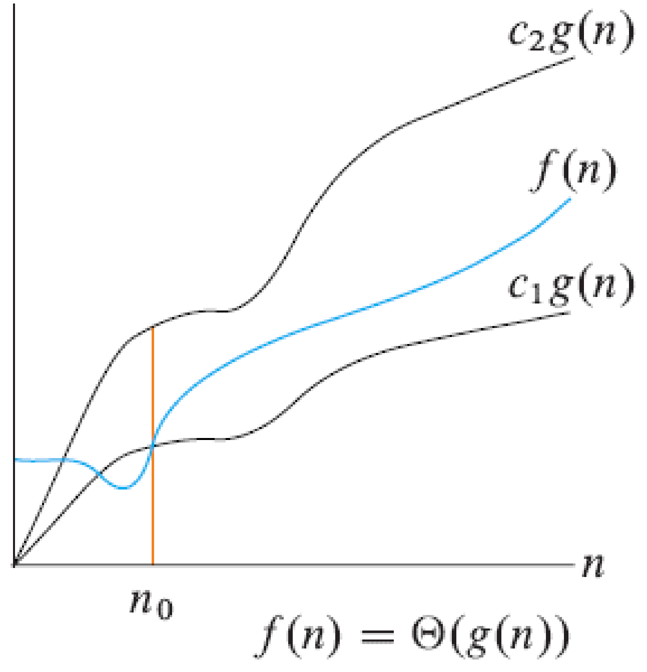
Formal Definition

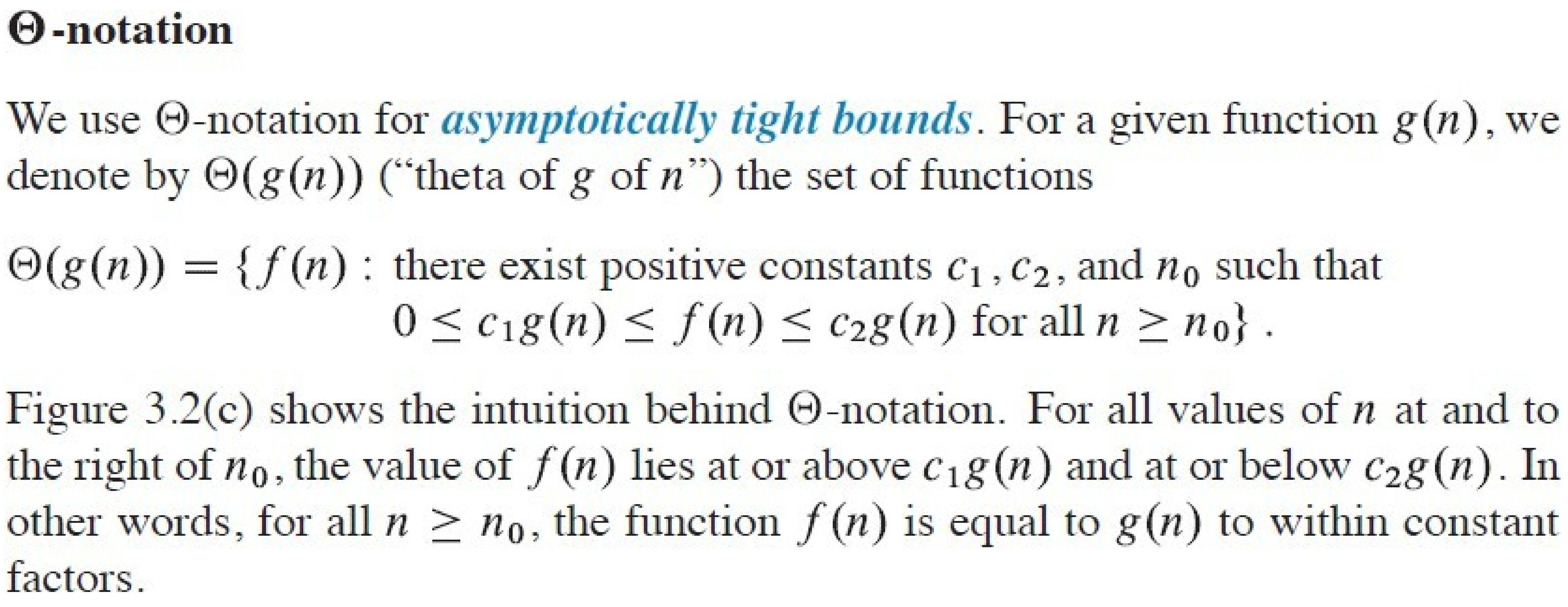
An algorithm is said to have a time complexity of $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that for all $n \geq n_0$:

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

where:

- $f(n)$ is the actual running time of the algorithm for an input of size n.
- $g(n)$ is a function that describes a tight bound on $f(n)$.
- c_1, c_2, and n_0 are constants.





Θ-notation

We use Θ-notation for *asymptotically tight bounds*. For a given function $g(n)$, we denote by $\Theta(g(n))$ ("theta of g of n") the set of functions

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

Figure 3.2(c) shows the intuition behind Θ-notation. For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.

Theorem 3.1

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 $f(n) = \Omega(g(n))$. ■

Step-by-Step Procedure to Find Big Theta (Θ)

Step 1: Understanding the Function $f(n)$

Consider an algorithm with a time complexity given by the function $f(n) = 3n^2 + 2n + 4$.

1. **Identify the dominant term:**
   - For large values of n, the term $3n^2$ is the dominant term.
2. **Simplify the function:**
   - For large values of n, $3n^2 + 2n + 4$ can be approximated by $3n^2$.

3. **Remove constants:**

   - The constant factor is kept for both the upper and lower bounds. Thus, $3n^2$ can be considered for both bounds.

4. **Big Theta notation:**

   - Therefore, $f(n)$ is $\Theta(n^2)$.

Step 2: Formal Definition Application

To apply the formal definition, let's find constants c_1, c_2, and n_0 such that for all $n \geq n_0$:

$$c_1 \cdot g(n) = c_1 \cdot n^2 \leq f(n) = 3n^2 + 2n + 4 \leq c_2 \cdot g(n) = c_2 \cdot n^2$$

Upper Bound:

- Choose $c_2 = 4$:
  - For $n \geq 1$, we have:
    $3n^2 + 2n + 4 \leq 4n^2$
  - This inequality holds because the additional terms $2n + 4$ are less than n^2 for large n.

Lower Bound:

- Choose $c_1 = 3$:
  - For $n \geq 1$, we have:
    $3n^2 + 2n + 4 \geq 3n^2$
  - This inequality holds because $2n + 4$ are positive terms that only add to the left-hand side.

Thus, we can say that $f(n)$ is $\Theta(n^2)$.

Example 1: Linear Function

Function: $f(n) = 5n + 3$

Goal: Show that $f(n)$ is $\Theta(n)$.

Step-by-Step Solution:

1. **Identify the Dominant Term**:
    - The dominant term is $5n$.
2. **Simplify by ignoring constants and lower-order terms**:
    - Approximate $5n + 3$ by $5n$.

3. **Form the Inequality:**

   - We need $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.
   - $c_1 \cdot n \leq 5n + 3 \leq c_2 \cdot n$.

4. **Choose c_1 and c_2:**

   - Let's choose $c_1 = 5$ and $c_2 = 6$.

5. **Solve for n:**

   - For the lower bound: $5n \leq 5n + 3$ holds for all $n \geq 0$.
   - For the upper bound: $5n + 3 \leq 6n$ holds for $n \geq 3$.

6. Determine n_0:

- Here, $n_0 = 3$.

So, $f(n) = 5n + 3$ is $\Theta(n)$ with $c_1 = 5$, $c_2 = 6$, and $n_0 = 3$.

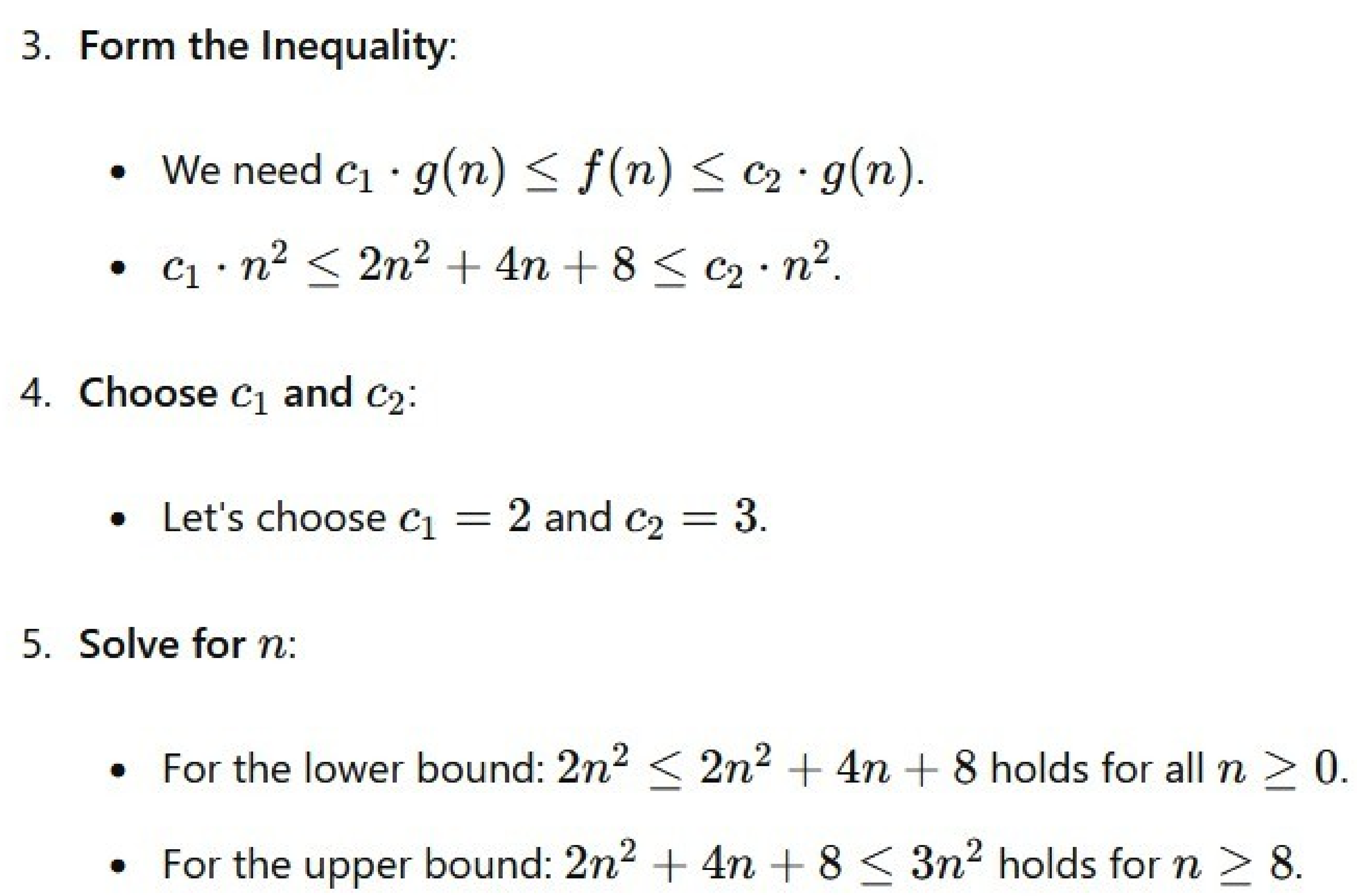
Example 2: Quadratic Function

Function: $f(n) = 2n^2 + 4n + 8$

Goal: Show that $f(n)$ is $\Theta(n^2)$.

Step-by-Step Solution:

1. **Identify the Dominant Term:**
   - The dominant term is $2n^2$.
2. **Simplify by ignoring constants and lower-order terms:**
   - Approximate $2n^2 + 4n + 8$ by $2n^2$.



3. **Form the Inequality:**
   - We need $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.
   - $c_1 \cdot n^2 \leq 2n^2 + 4n + 8 \leq c_2 \cdot n^2$.
4. **Choose c_1 and c_2:**
   - Let's choose $c_1 = 2$ and $c_2 = 3$.
5. **Solve for n:**
   - For the lower bound: $2n^2 \leq 2n^2 + 4n + 8$ holds for all $n \geq 0$.
   - For the upper bound: $2n^2 + 4n + 8 \leq 3n^2$ holds for $n \geq 8$.

6. **Determine n_0:**

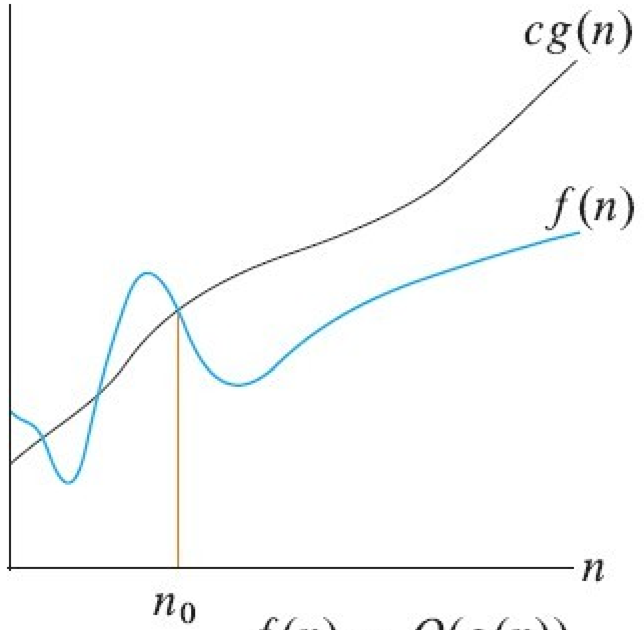
- Here, $n_0 = 8$.

So, $f(n) = 2n^2 + 4n + 8$ is $\Theta(n^2)$ with $c_1 = 2$, $c_2 = 3$, and $n_0 = 8$.

Example 3: Cubic Function

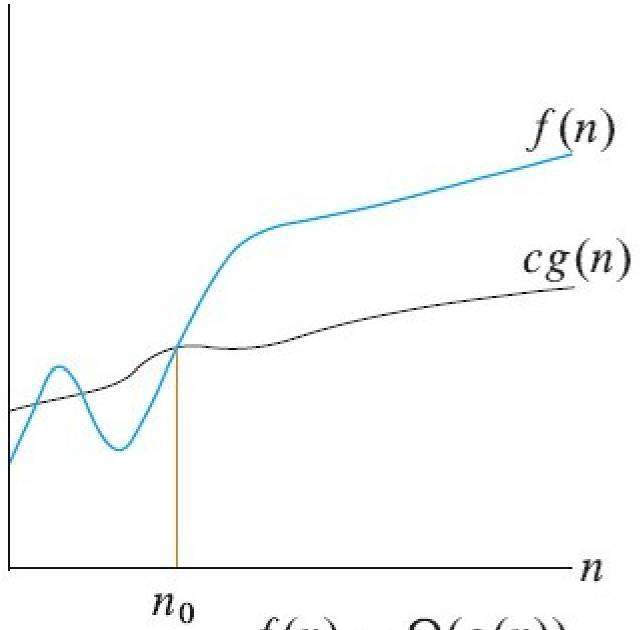
Function: $f(n) = 6n^3 + 5n^2 + 4n + 2$

Goal: Show that $f(n)$ is $\Theta(n^3)$.



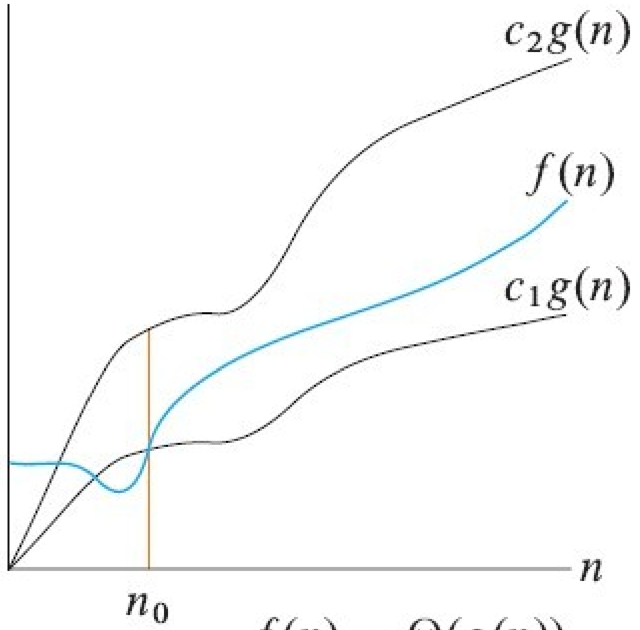
$f(n) = O(g(n))$

(a)



$f(n) = \Omega(g(n))$

(b)



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

(c)

Little Oh notation - o

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

Definition of Little-oh Notation

Little-oh notation, written as $f(n) = o(g(n))$, describes the behavior of $f(n)$ as it grows much slower than $g(n)$ when n becomes large. Specifically, $f(n)$ is said to be little-oh of $g(n)$ if, for any positive constant c, there exists a threshold n_0 such that:

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

In "o" notation the function f(n) becomes insignificant relative to g(n) as n gets large:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 .$$

Little-omega Notation ω

Definition of Little-omega Notation

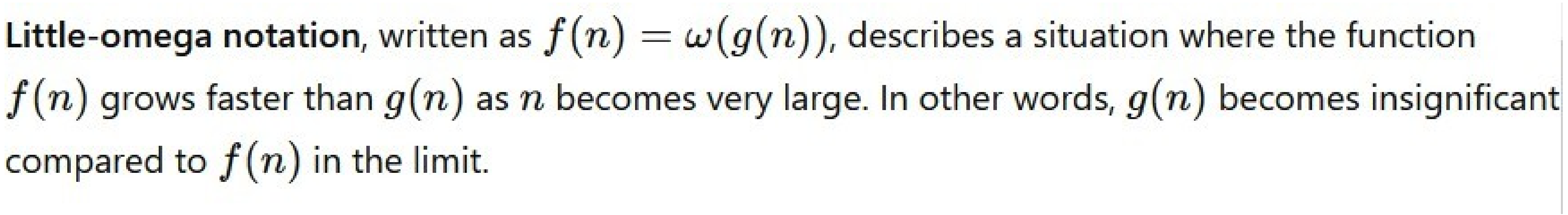
Formally, $f(n) = \omega(g(n))$ means that for any positive constant c, there exists a constant n_0 such that:

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

This indicates that $f(n)$ grows faster than $g(n)$ as $n \to \infty$.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \,,$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n gets large.



Little-omega notation, written as $f(n) = \omega(g(n))$, describes a situation where the function $f(n)$ grows faster than $g(n)$ as n becomes very large. In other words, $g(n)$ becomes insignificant compared to $f(n)$ in the limit.