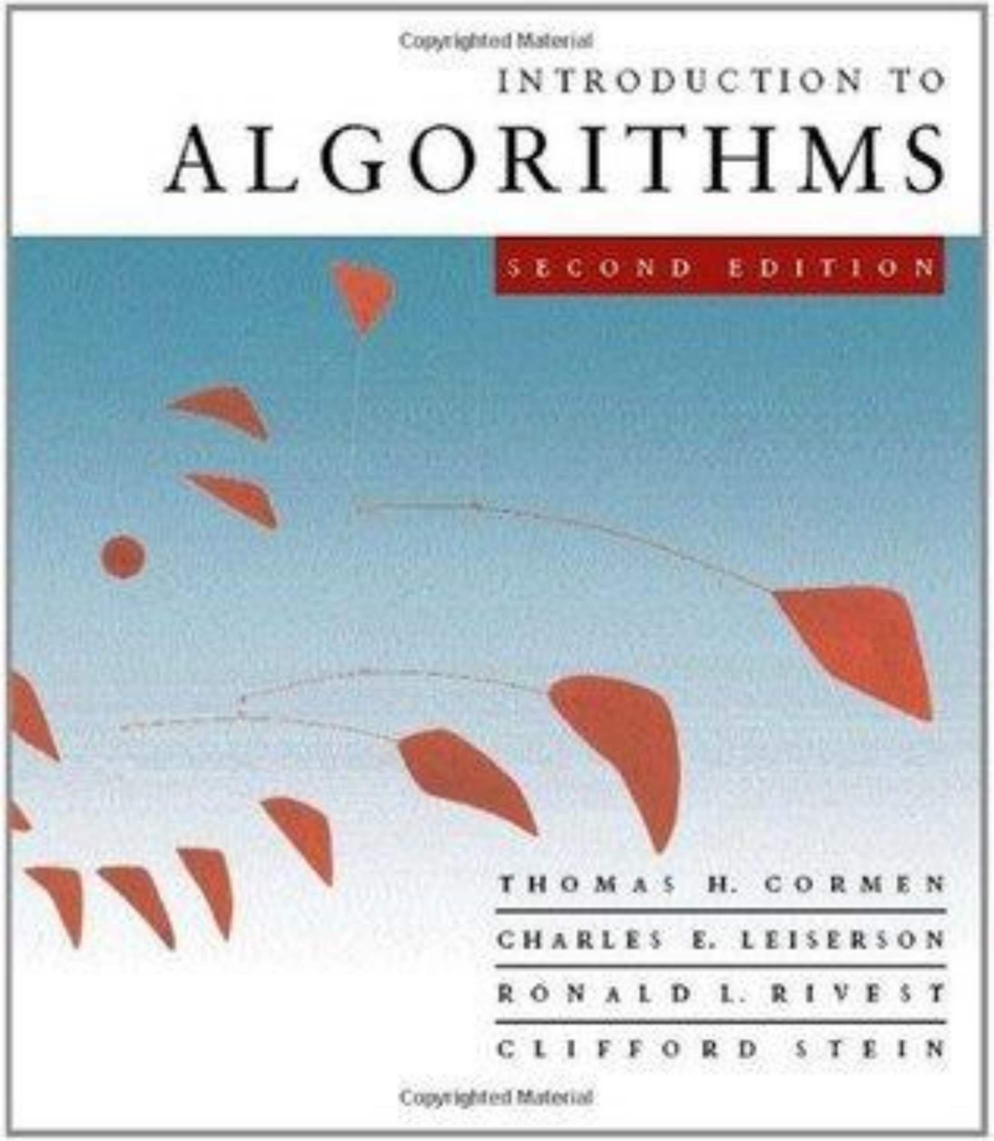
Fall 2024

Prof. Alaa Sheta

Algorithms



Copyrighted Material

INTRODUCTION TO

ALGORITHMS

SECOND EDITION

THOMAS H. CORMEN

CHARLES E. LEISERSON

RONALD L. RIVEST

CLIFFORD STEIN

Copyrighted Material

Why Understanding Time Complexity?

- Understanding the time complexity of an algorithm is essential for several reasons:

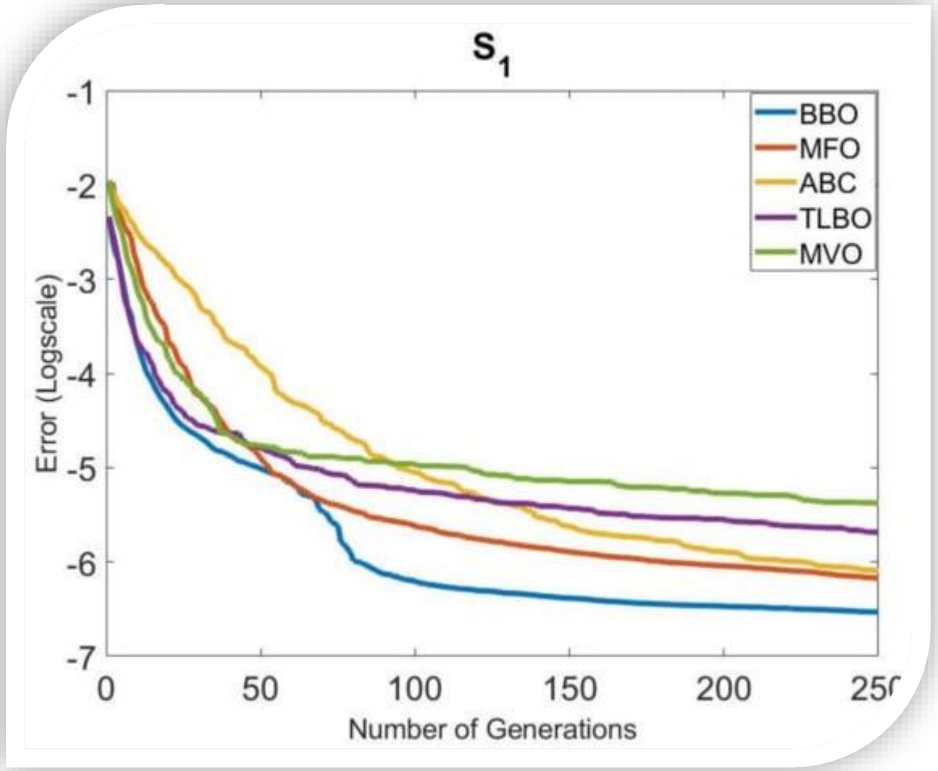
1. **Performance Evaluation**

- **Predicting Efficiency**: Time complexity allows you to estimate how an algorithm's runtime will increase as the input data size grows. This helps select the most efficient algorithm for a given problem, especially when working with large datasets.
- **Comparing Algorithms**: It provides a standardized way to compare different algorithms, enabling you to choose the best performance.

Why Understanding Time Complexity?

2. Scalability

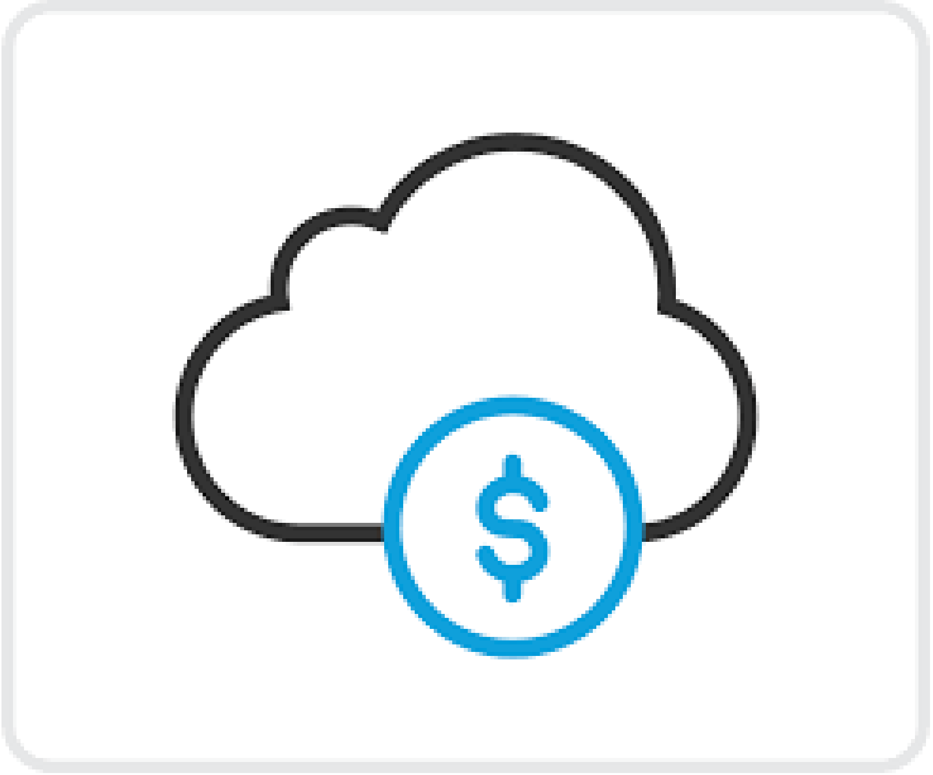
- **Handling Large Inputs**: Time complexity analysis helps understand how well an algorithm scales with increasing input size. This is essential in real-world applications where data can be enormous and inefficient algorithms can become impractical.

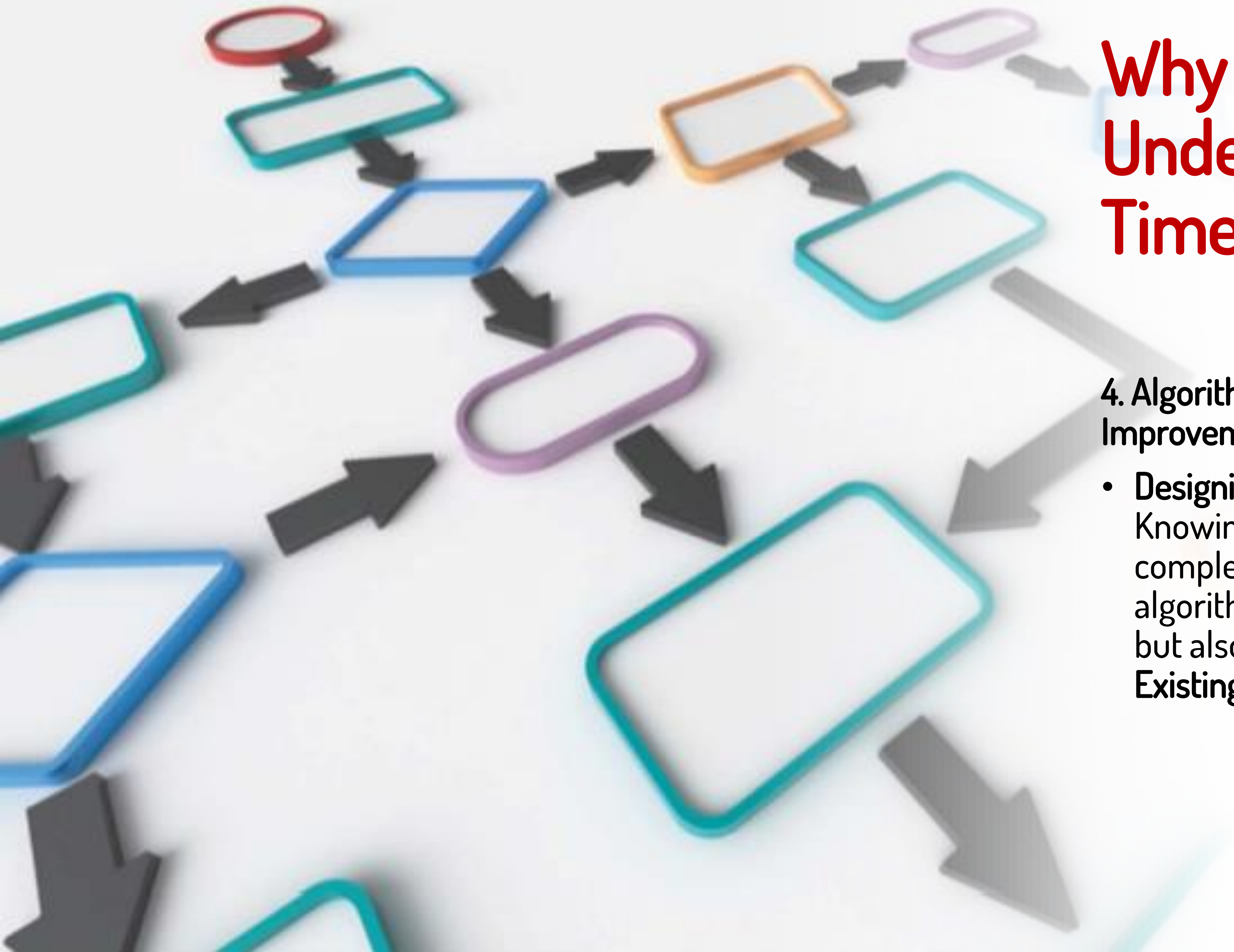


Why Understanding Time Complexity?

3. Resource Management

- **Optimizing Resource Use**: Time complexity gives insights into how much computational resources (like CPU time) an algorithm will require.
- **Cost-Effective Solutions**: In scenarios like cloud computing, where processing power comes at a cost, understanding time complexity can lead to more cost-effective solutions.





Why Understanding Time Complexity?

4. Algorithm Design and Improvement

- **Designing Efficient Algorithms**: Knowing how to evaluate time complexity is essential in designing algorithms that are not just correct but also efficient and help improve **Existing Algorithms**.

Why Understanding Time Complexity?

5. Practical Implications

- **Real-World Applications**: In practical applications, choosing an algorithm with a better time complexity can lead to faster, more responsive software and systems.
- **User Experience**: For applications with critical performance, such as gaming, finance, or real-time systems, understanding and optimizing time complexity can significantly enhance user experience.

Understanding Time Complexity

- Many students find the concept of time complexity frightening.
- Here, we will use simple examples to make it more approachable and less intimidating.

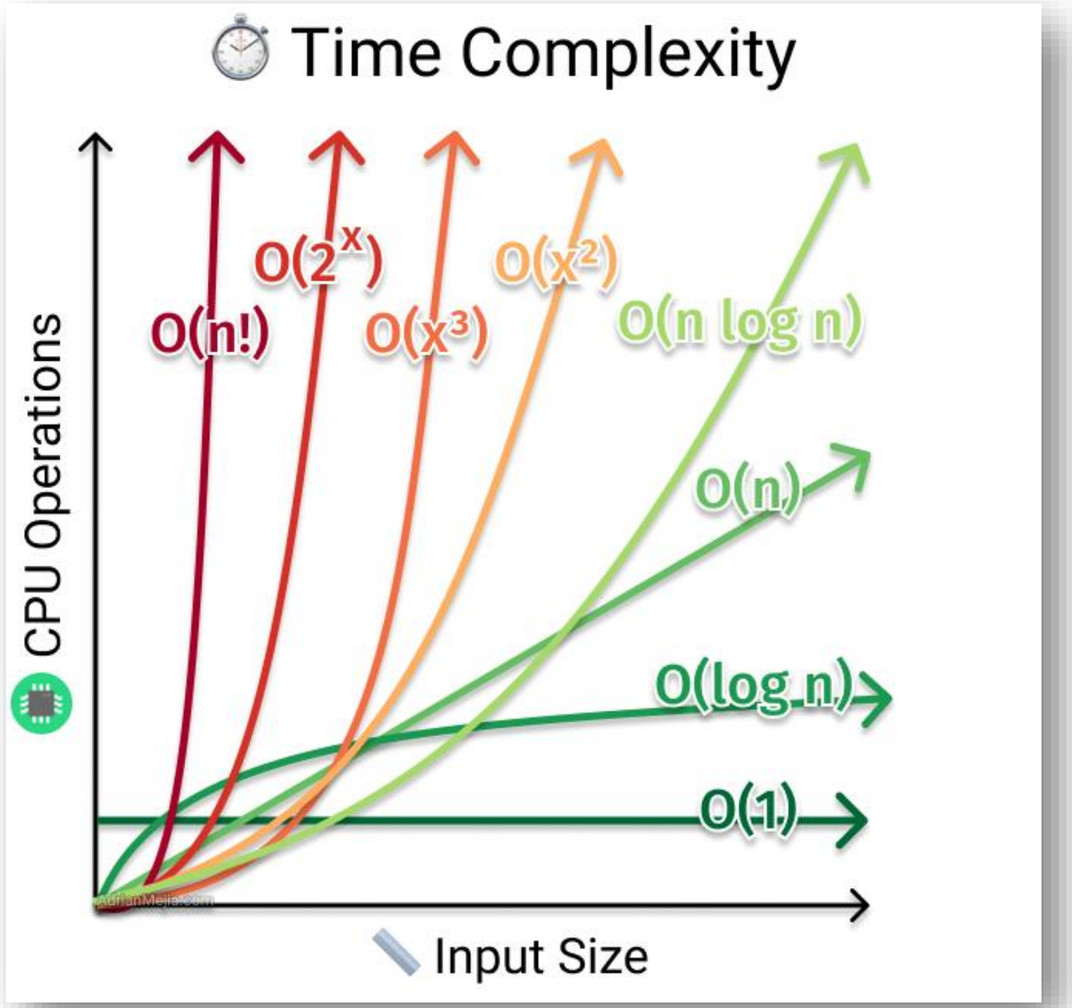
Some definitions

- **Time efficiency**, also called **time complexity**, indicates **how fast an algorithm** in question runs.
- **Space efficiency**, or **space complexity**, refers to the **amount of memory units** the algorithm requires and the space needed for its input and output.

- **Measuring an Input's Size:**
  - Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter *n* indicating its input size.

Time complexity

- The time complexity of an algorithm depends on:
  - How many lines of code to run?
  - How long it takes to run a single line of code?
  - The complexity of the formula to compute time.



Some facts

- Measuring Running Time:
  - **Standard Units of Time:** While we could measure an algorithm's running time in seconds or milliseconds, this approach has drawbacks, such as dependence on the computer's speed, the quality of the implementation, and the difficulty in accurately clocking the running time.
  - **Need for a Better Metric:** A more reliable metric is needed to gauge an algorithm's efficiency independently of these external factors.

Some facts

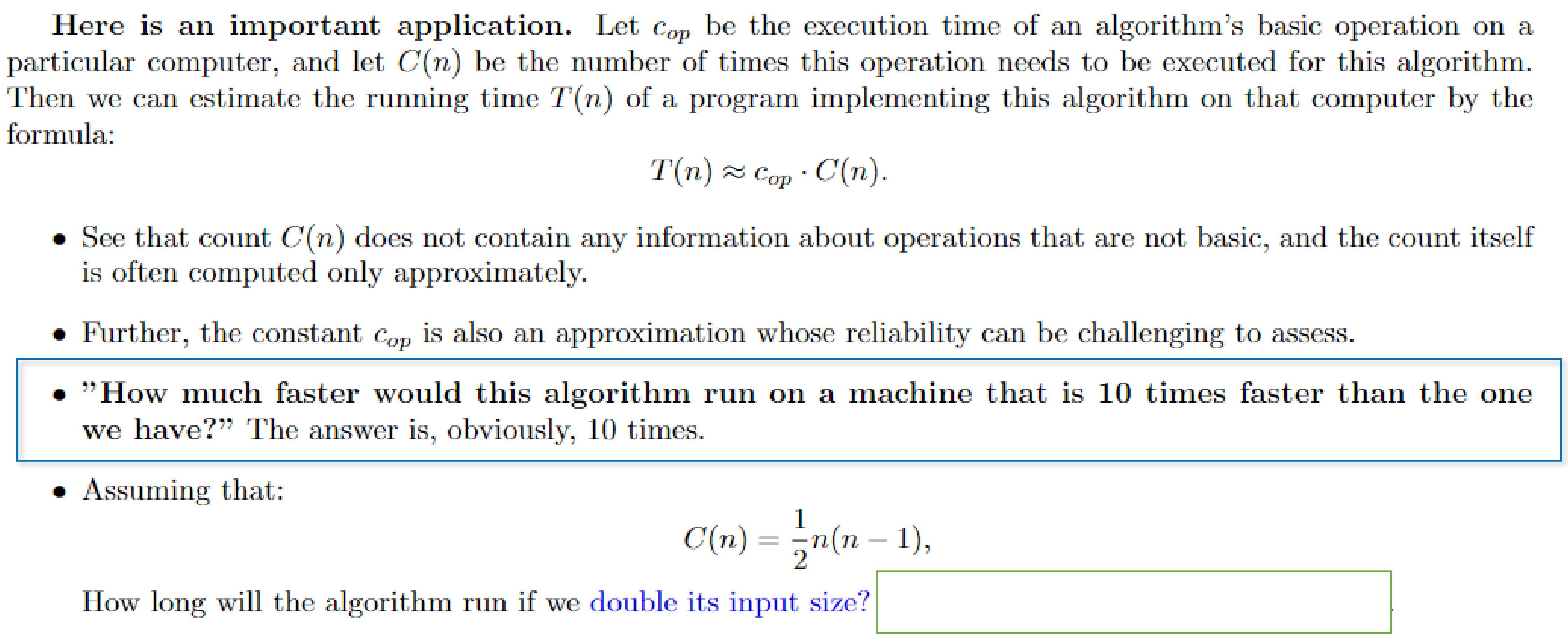
- **Counting Operations:**
  - **Alternative Approach:** One method is to **count the number of times each operation in the algorithm is executed.** However, this is typically too complex and unnecessary.
  - **Basic Operation:** Instead, focus on identifying and counting the most significant operation, known as the *basic operation*, which contributes the most to the algorithm's total running time. **By analyzing the frequency of this operation, we can estimate the algorithm's efficiency.**

Examples of basic operation

- As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop.
- For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison.

- As another example, algorithms for mathematical problems typically involve some or all the four arithmetical operations: addition, subtraction, multiplication, and division.
- Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together

If we are counting on real-time only



Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula:

$$T(n) \approx c_{op} \cdot C(n).$$

- See that count $C(n)$ does not contain any information about operations that are not basic, and the count itself is often computed only approximately.
- Further, the constant c_{op} is also an approximation whose reliability can be challenging to assess.
- **"How much faster would this algorithm run on a machine that is 10 times faster than the one we have?"** The answer is, obviously, 10 times.
- Assuming that:

$$C(n) = \frac{1}{2}n(n-1),$$

How long will the algorithm run if we double its input size?

Prove

Solution:

$$C(2n) = \frac{1}{2}(2n)(2n-1) = \frac{1}{2}(2n)^2 - \frac{1}{2}(2n) \approx \frac{1}{2}(2n)^2,$$

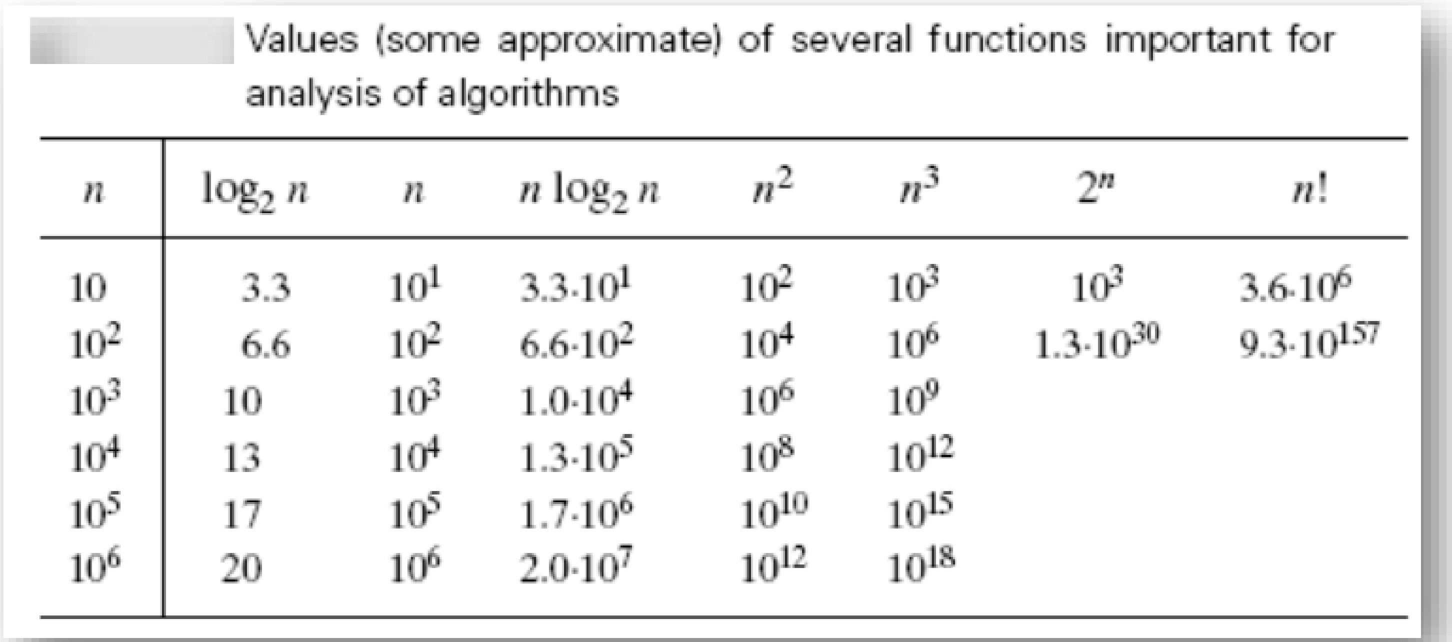
and therefore,

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op} \cdot C(2n)}{c_{op} \cdot C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Observations:

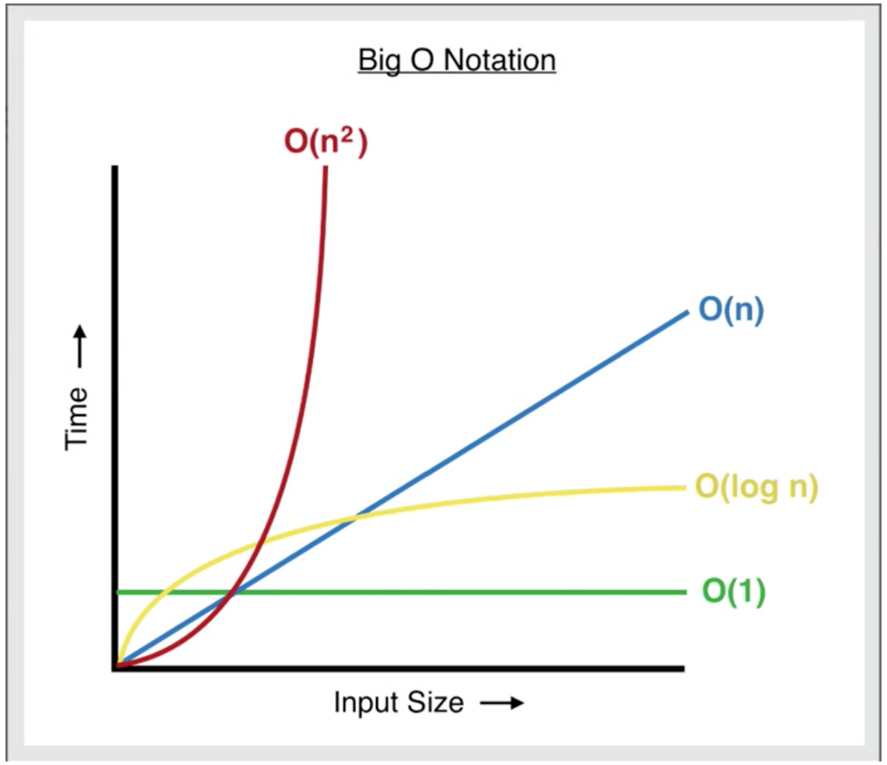
- Note that we could answer the last question without knowing the value of c_{op}: it was neatly canceled out in the ratio.
- Also, note that $\frac{1}{2}$, the multiplicative constant in the formula for the count $C(n)$, was also canceled out.
- For these reasons, the efficiency analysis framework ignores multiplicative constants and concentrates on the count's order of growth.

Orders of Growth

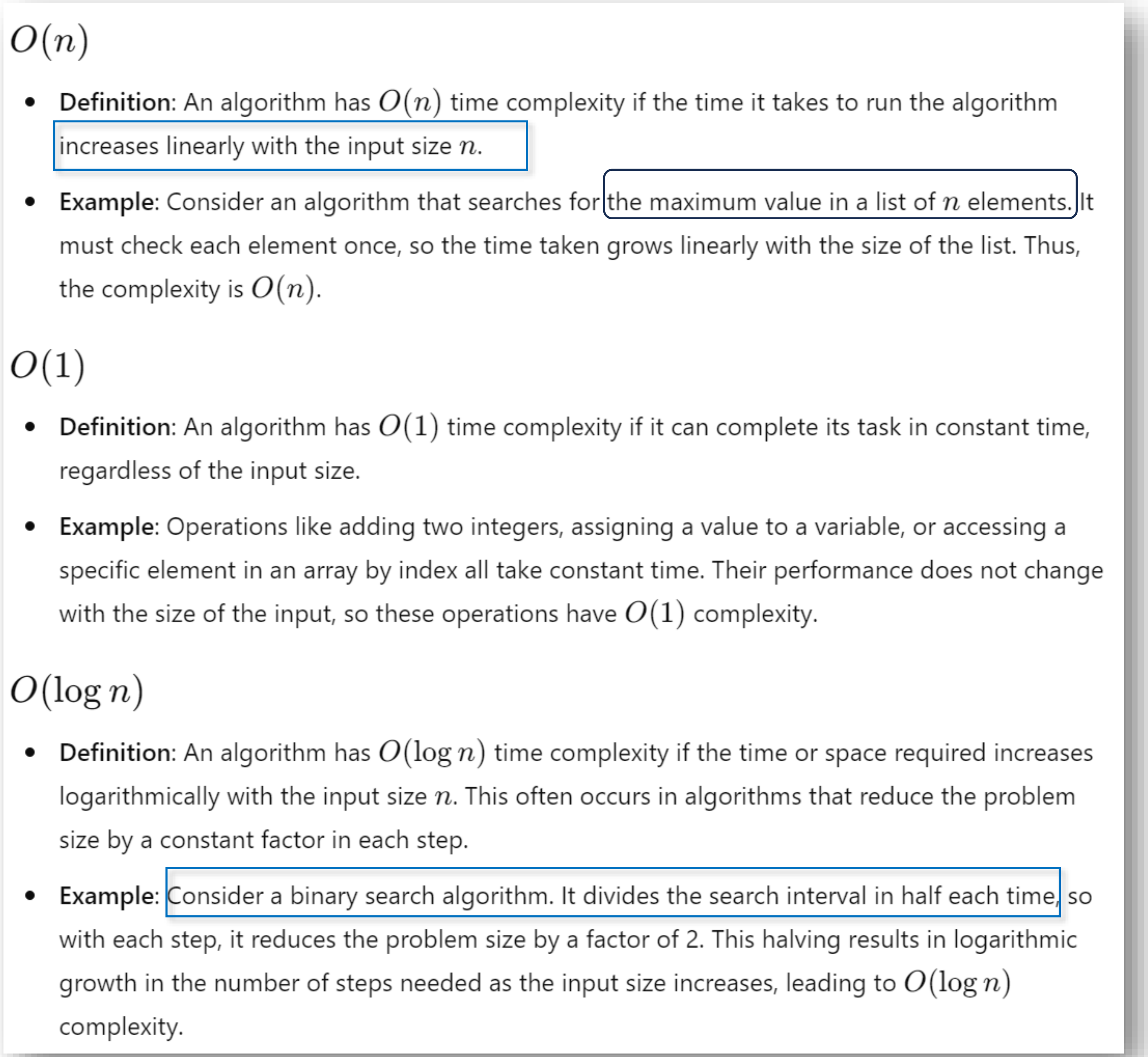


Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		



Examples of time complexity



$O(n)$

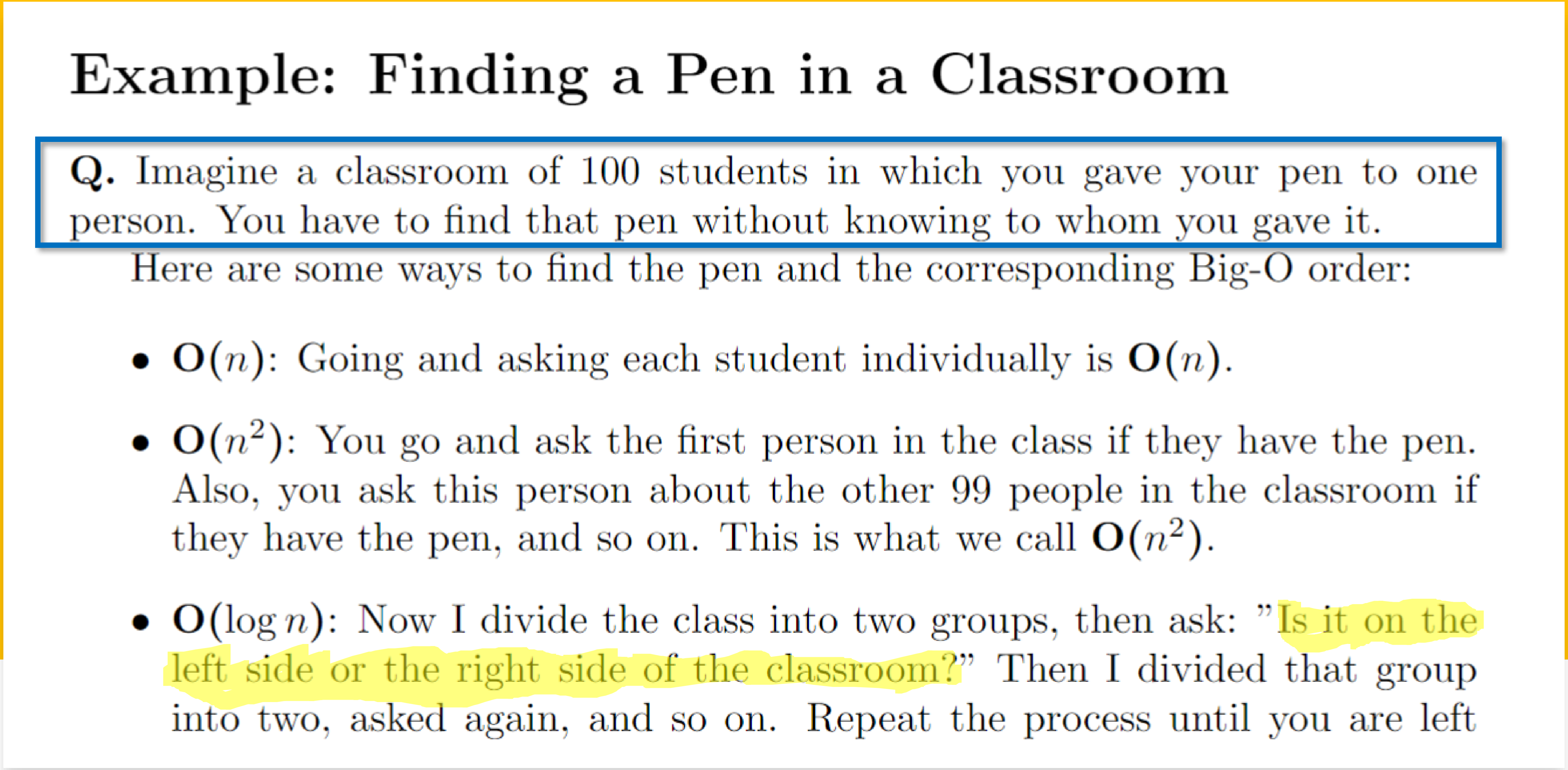
- **Definition**: An algorithm has $O(n)$ time complexity if the time it takes to run the algorithm increases linearly with the input size n.
- **Example**: Consider an algorithm that searches for the maximum value in a list of n elements. It must check each element once, so the time taken grows linearly with the size of the list. Thus, the complexity is $O(n)$.

$O(1)$

- **Definition**: An algorithm has $O(1)$ time complexity if it can complete its task in constant time, regardless of the input size.
- **Example**: Operations like adding two integers, assigning a value to a variable, or accessing a specific element in an array by index all take constant time. Their performance does not change with the size of the input, so these operations have $O(1)$ complexity.

$O(\log n)$

- **Definition**: An algorithm has $O(\log n)$ time complexity if the time or space required increases logarithmically with the input size n. This often occurs in algorithms that reduce the problem size by a constant factor in each step.
- **Example**: Consider a binary search algorithm. It divides the search interval in half each time, so with each step, it reduces the problem size by a factor of 2. This halving results in logarithmic growth in the number of steps needed as the input size increases, leading to $O(\log n)$ complexity.



Example: Finding a Pen in a Classroom

Q. Imagine a classroom of 100 students in which you gave your pen to one person. You have to find that pen without knowing to whom you gave it.

Here are some ways to find the pen and the corresponding Big-O order:

- $\mathbf{O}(n)$: Going and asking each student individually is $\mathbf{O}(n)$.
- $\mathbf{O}(n^2)$: You go and ask the first person in the class if they have the pen. Also, you ask this person about the other 99 people in the classroom if they have the pen, and so on. This is what we call $\mathbf{O}(n^2)$.
- $\mathbf{O}(\log n)$: Now I divide the class into two groups, then ask: "Is it on the left side or the right side of the classroom?" Then I divided that group into two, asked again, and so on. Repeat the process until you are left

Is Time Complexity the Same as the Running/Execution Time of Code?

The time complexity of an algorithm or code is not equal to the actual time required to execute a particular code, but rather the number of times a statement executes. We can prove this by using the `time` command.

Example

Write code in C/C++ or any other language to find the maximum between N numbers, where N varies from 10, 100, 1000, and 10,000. For a Linux-based operating system (Fedora or Ubuntu), use the below commands:

- To compile the program: `gcc program.c -o program`
- To execute the program: `time ./program`

Surprising Results

You will get surprising results, such as:

- For $N = 10$: you may get 0.5 ms time.
- For $N = 10,000$: you may get 0.2 ms time.

You will also get different timings on different machines. Even on the same machine, you might not get the same timings for the same code, due to factors like the current network load.

So, we can say that the actual time required to execute code is machine-dependent (whether you are using Pentium 1 or Pentium 5), and also considers network load if your machine is in LAN/WAN.

Different Types of Time Complexity

As we have seen, time complexity is given by time as a function of the length of the input. There exists a relation between the input data size (n) and the number of operations performed (N) concerning time. This relation is denoted as the order of growth in time complexity and is given by the notation $O[n]$, where O is the order of growth and n is the length of the input. It is also called the 'Big O Notation.'

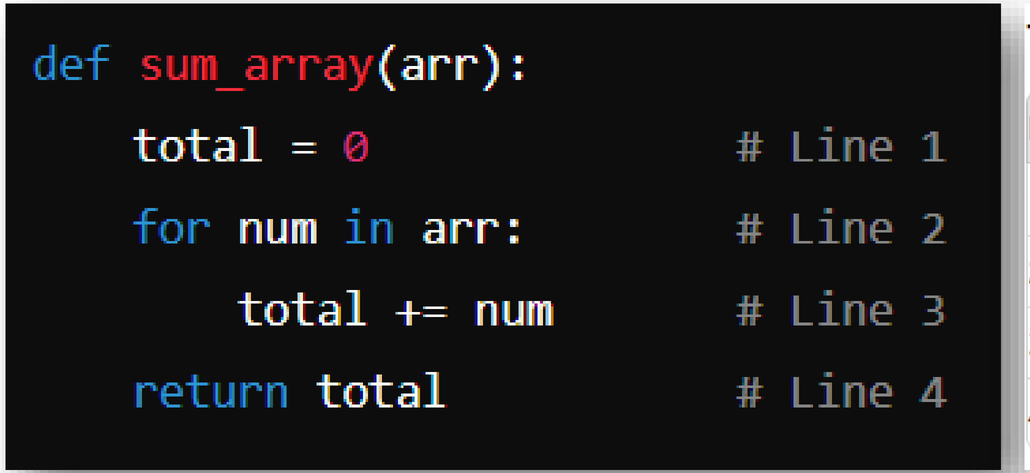
Big O Notation expresses the runtime of an algorithm in terms of how quickly it grows relative to the input n by defining the N number of operations done on it. Thus, the time complexity of an algorithm is denoted by the combination of all $O[n]$ assigned for each line of function.

There are different types of time complexities used. Let's see them one by one:

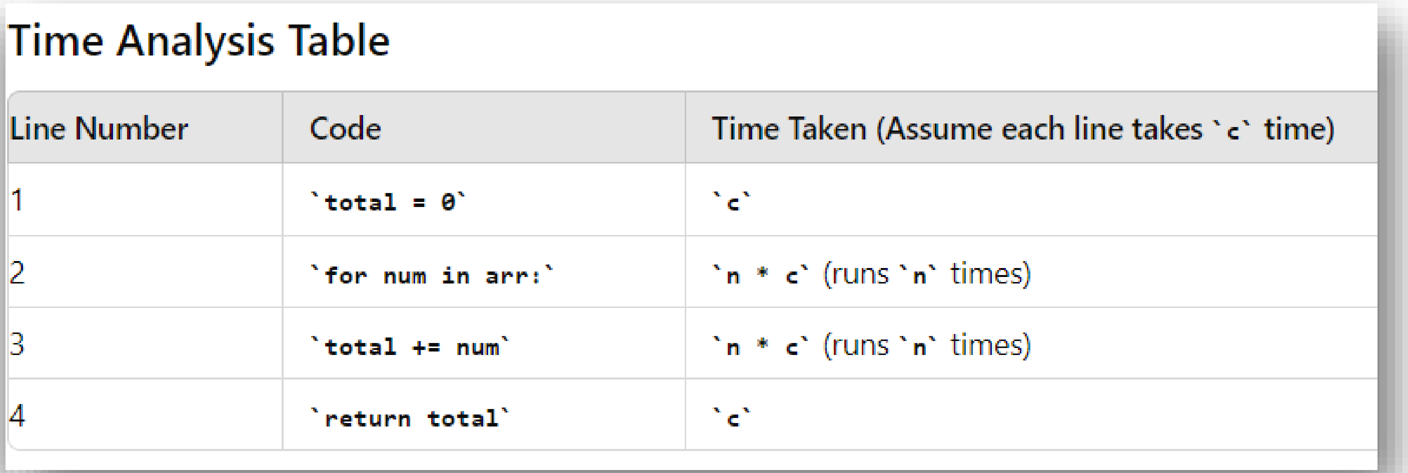
- **Constant time** – $O(1)$
- **Linear time** – $O(n)$
- **Logarithmic time** – $O(\log n)$
- **Quadratic time** – $O(n^2)$
- **Cubic time** – $O(n^3)$

Based on the type of functions defined, many more complex notations, such as exponential time, quasilinear time, factorial time, etc., are used.

Time complexity: Summing the elements of an array

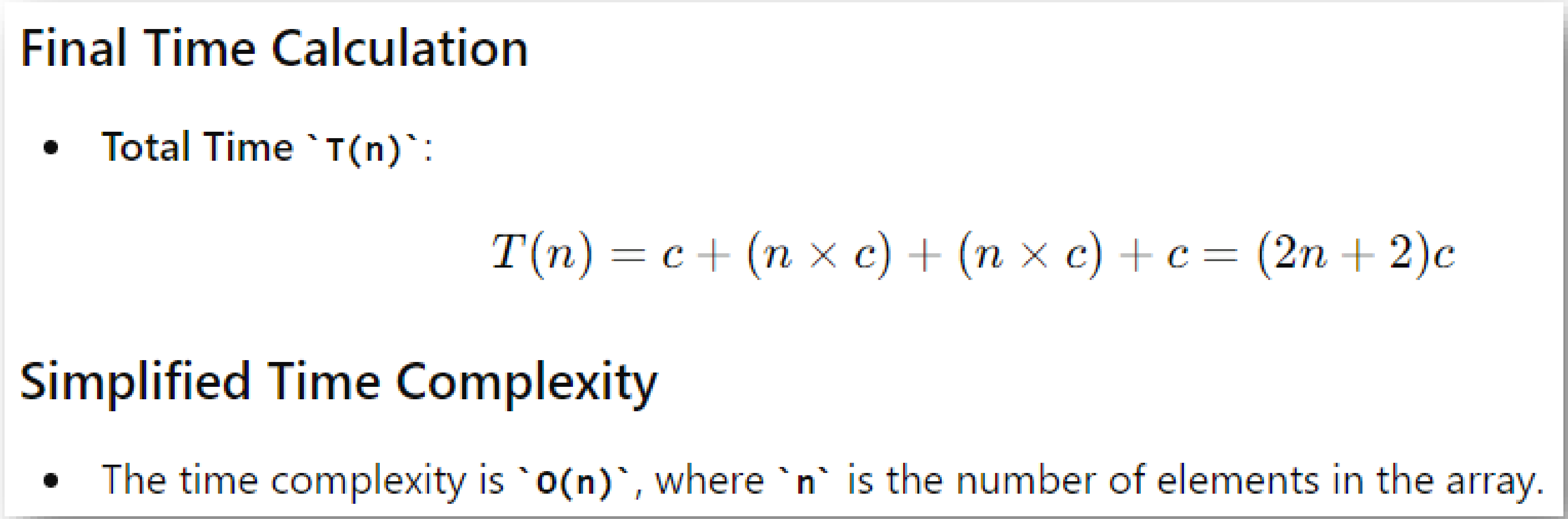


```python
def sum_array(arr):
    total = 0               # Line 1
    for num in arr:         # Line 2
        total += num        # Line 3
    return total            # Line 4
```



Time Analysis Table

Line Number	Code	Time Taken (Assume each line takes `c` time)
1	`total = 0`	`c`
2	`for num in arr:`	`n * c` (runs `n` times)
3	`total += num`	`n * c` (runs `n` times)
4	`return total`	`c`



Final Time Calculation

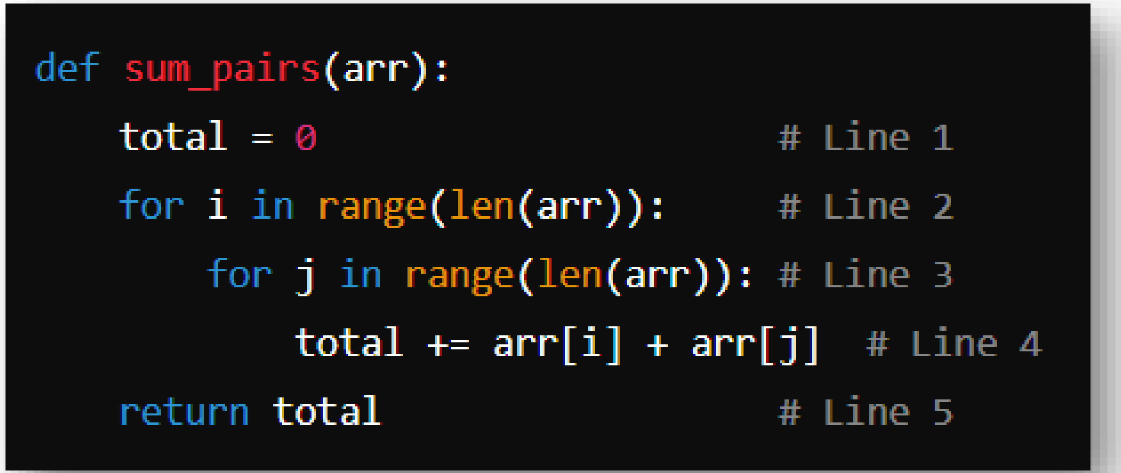
- Total Time `T(n)`:

$$T(n) = c + (n \times c) + (n \times c) + c = (2n + 2)c$$

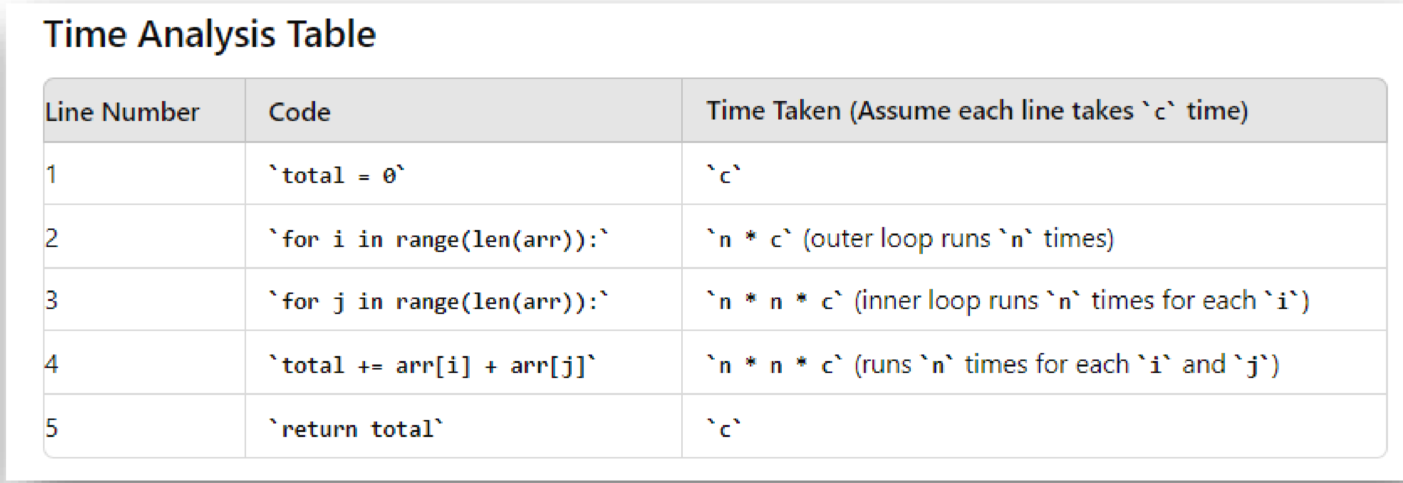
Simplified Time Complexity

- The time complexity is `O(n)`, where `n` is the number of elements in the array.

Time complexity: Nested Loop

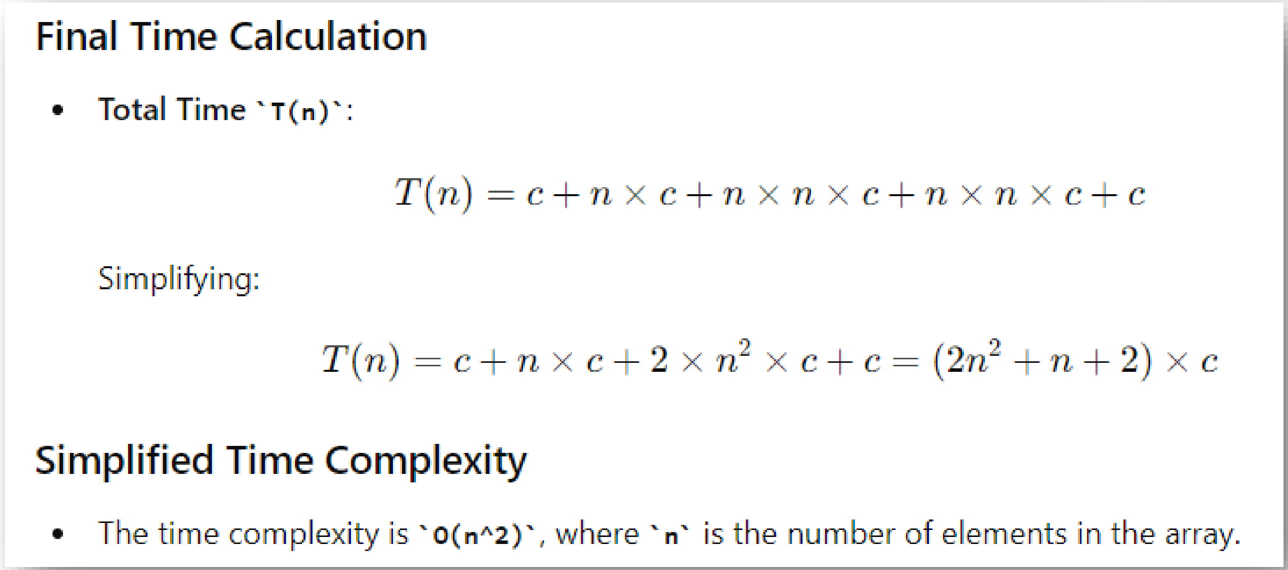


```python
def sum_pairs(arr):
    total = 0                       # Line 1
    for i in range(len(arr)):       # Line 2
        for j in range(len(arr)): # Line 3
            total += arr[i] + arr[j]  # Line 4
    return total                    # Line 5
```



Time Analysis Table

Line Number	Code	Time Taken (Assume each line takes `c` time)
1	`total = 0`	`c`
2	`for i in range(len(arr)):`	`n * c` (outer loop runs `n` times)
3	`for j in range(len(arr)):`	`n * n * c` (inner loop runs `n` times for each `i`)
4	`total += arr[i] + arr[j]`	`n * n * c` (runs `n` times for each `i` and `j`)
5	`return total`	`c`



Final Time Calculation

- Total Time `T(n)`:

$$T(n) = c + n \times c + n \times n \times c + n \times n \times c + c$$

  Simplifying:

$$T(n) = c + n \times c + 2 \times n^2 \times c + c = (2n^2 + n + 2) \times c$$

Simplified Time Complexity

- The time complexity is `O(n^2)`, where `n` is the number of elements in the array.

Matrix Multiplication: Time complexity

```
MatrixMultiply(A, B)
1. Initialize matrix C of size n x n with zeros
2. for i = 0 to n-1:
3.     for j = 0 to n-1:
4.         C[i][j] = 0
5.         for k = 0 to n-1:
6.             C[i][j] = C[i][j] + A[i][k] * B[k][j]
7. return C
```

Line	Operation	Time Complexity
1	Initialize matrix C of size $n \times n$	$O(n^2)$
2	Outer loop: Iterate over i from 0 to $n-1$	$O(n)$
3	Inner loop: Iterate over j from 0 to $n-1$	$O(n)$
4	Initialize $C[i][j] = 0$	$O(1)$
5	Inner loop: Iterate over k from 0 to $n-1$	$O(n)$
6	Update $C[i][j] = C[i][j] + A[i][k] * B[k][j]$	$O(1)$
7	Return matrix C	$O(1)$

Matrix Multiplication: Time complexity

Example: Matrix Multiplication and Time Complexity

Consider multiplying two matrices A and B.

- Matrix A has dimensions $m \times n$.
- Matrix B has dimensions $n \times p$.

To find the resulting matrix C which will have dimensions $m \times p$, we need to compute each element of C by taking the dot product of the corresponding row from A and column from B.

Matrix Multiplication Calculation

1. **Element Calculation**: Each element C_{ij} of matrix C is computed as:

   $$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

   where i ranges from 1 to m and j ranges from 1 to p. For each element C_{ij}, we perform n multiplications and $n - 1$ additions (to sum the products).

2. **Total Operations for One Element**: To compute one element C_{ij}, we need n multiplications and $n - 1$ additions. The total number of operations for one element is $O(n)$.

3. **Total Elements in Matrix C**: Matrix C has $m \times p$ elements.

4. **Total Operations**: Therefore, to compute all elements in C, the total number of operations is:

   $$m \times p \times O(n)$$

   Since multiplication operations dominate, the time complexity for matrix multiplication is:

   $$\boxed{O(m \times n \times p)}$$

Example Calculation

Let's use specific dimensions to illustrate:

- Matrix A is 3×4 (3 rows, 4 columns).
- Matrix B is 4×5 (4 rows, 5 columns).

Here's the step-by-step time complexity calculation:

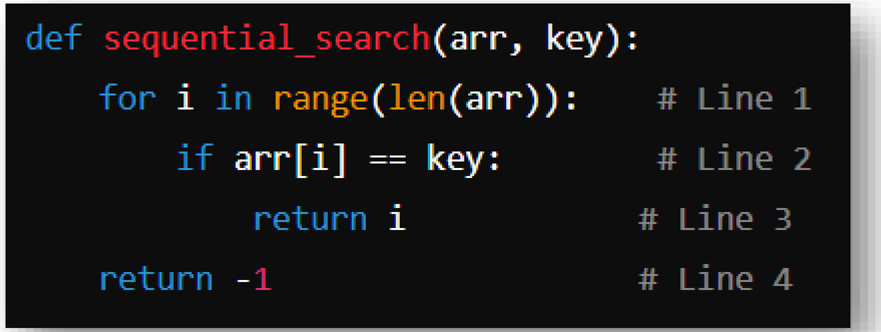
1. **Element Calculation**:
   - Each element of the resulting 3×5 matrix C requires 4 multiplications and 3 additions.
2. **Total Elements**:
   - Matrix C has $3 \times 5 = 15$ elements.
3. **Total Operations**:
   - Each element requires 4 multiplications and 3 additions, so for all elements, the total number of operations is approximately:

$$15 \times 4 = 60 \text{ multiplications}$$

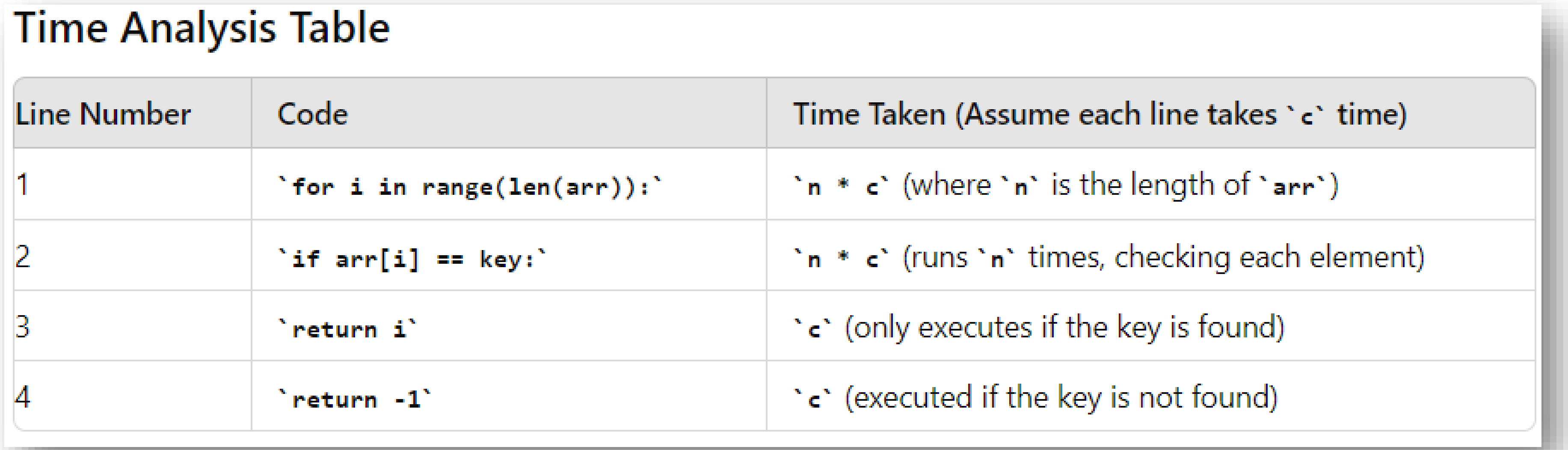
$$15 \times 3 = 45 \text{ additions}$$

4. **Time Complexity**:
   - The time complexity of this matrix multiplication is $O(3 \times 4 \times 5) = O(60)$, which is in line with the general formula $O(m \times n \times p)$.

Time complexity: sequential search



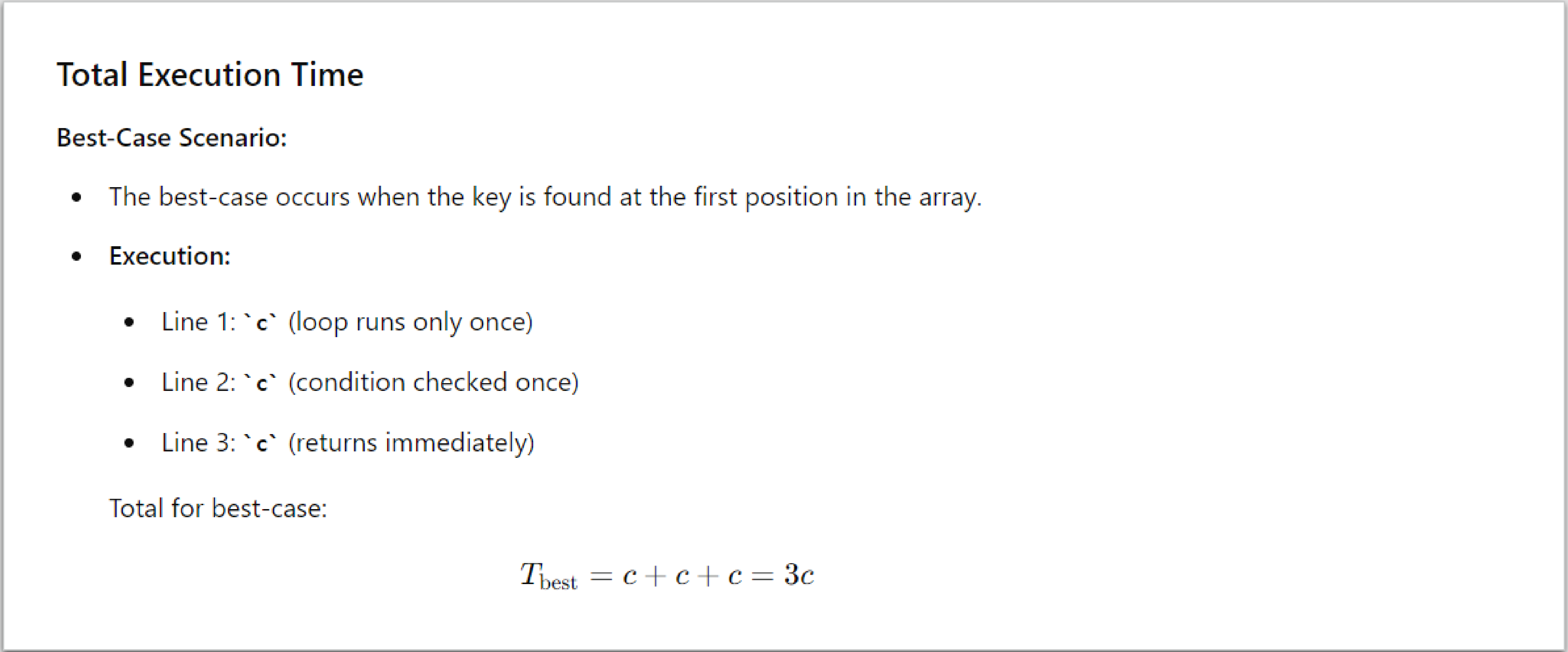
```
def sequential_search(arr, key):
    for i in range(len(arr)):     # Line 1
        if arr[i] == key:         # Line 2
            return i             # Line 3
    return -1                    # Line 4
```



Time Analysis Table

Line Number	Code	Time Taken (Assume each line takes `c` time)
1	`for i in range(len(arr)):`	`n * c` (where `n` is the length of `arr`)
2	`if arr[i] == key:`	`n * c` (runs `n` times, checking each element)
3	`return i`	`c` (only executes if the key is found)
4	`return -1`	`c` (executed if the key is not found)

Time complexity: sequential search



Total Execution Time

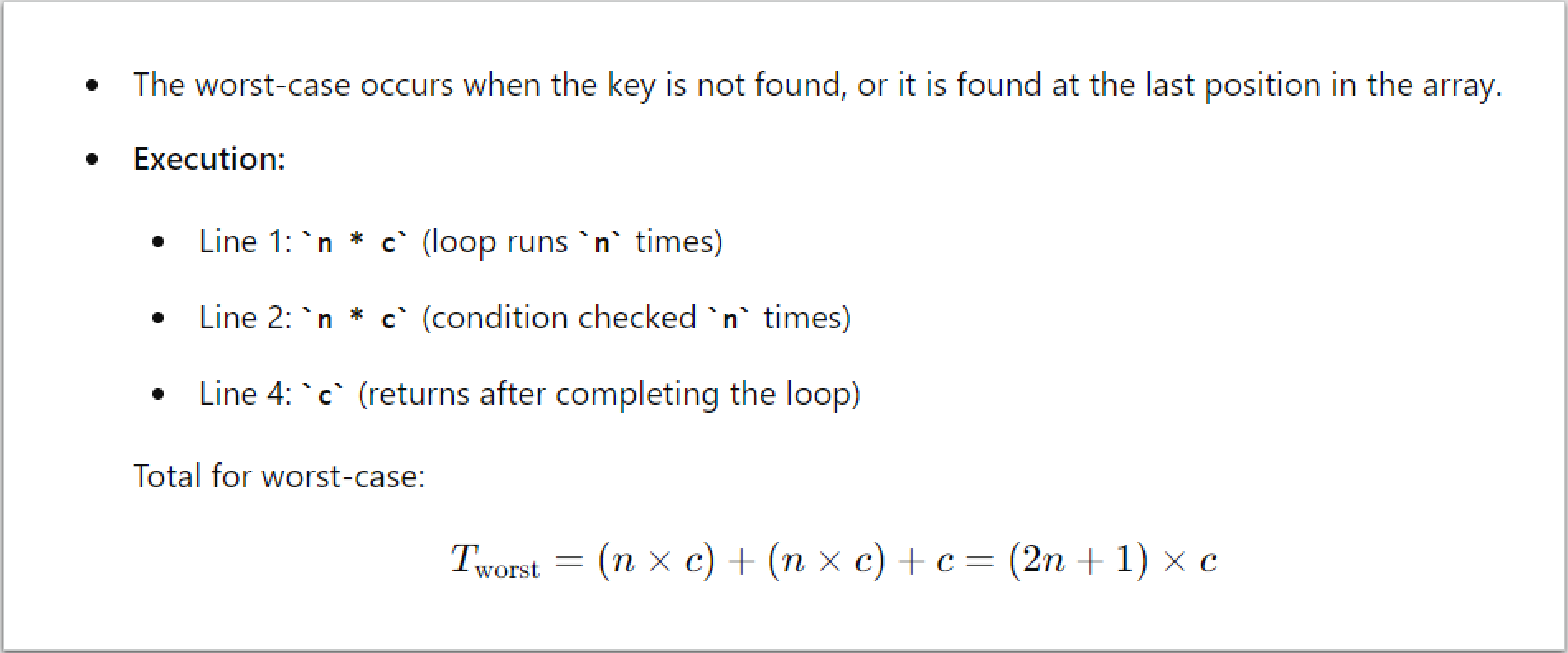
Best-Case Scenario:

- The best-case occurs when the key is found at the first position in the array.
- **Execution:**
    - Line 1: \`**c**\` (loop runs only once)
    - Line 2: \`**c**\` (condition checked once)
    - Line 3: \`**c**\` (returns immediately)

  Total for best-case:

$$T_{\text{best}} = c + c + c = 3c$$

Time complexity: sequential search

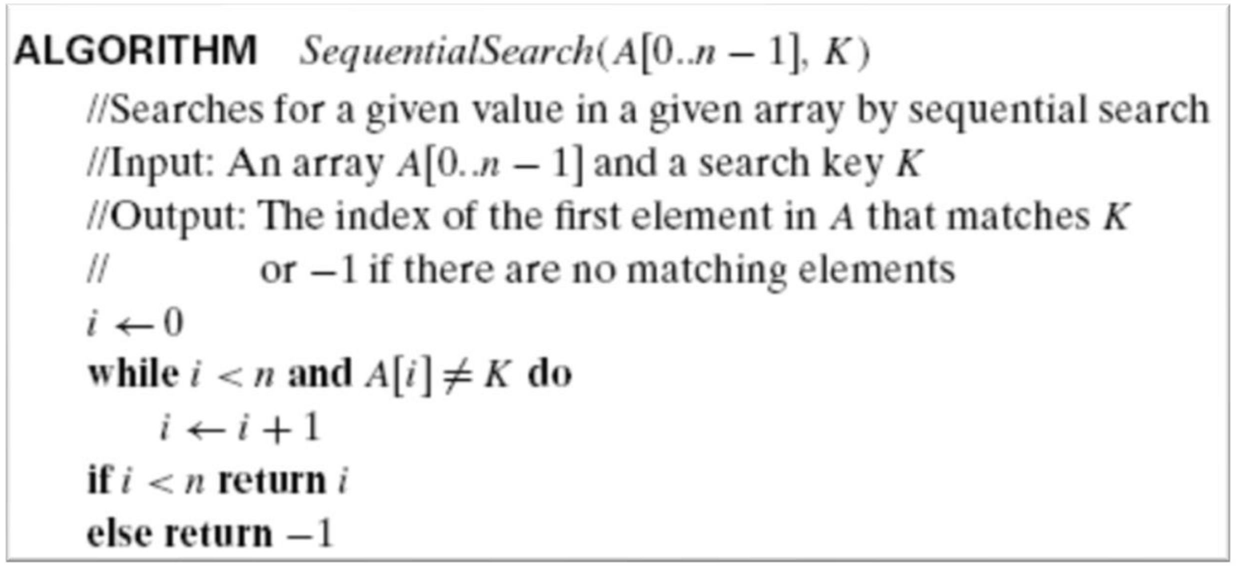


- The worst-case occurs when the key is not found, or it is found at the last position in the array.
- **Execution:**
    - Line 1: `n * c` (loop runs `n` times)
    - Line 2: `n * c` (condition checked `n` times)
    - Line 4: `c` (returns after completing the loop)

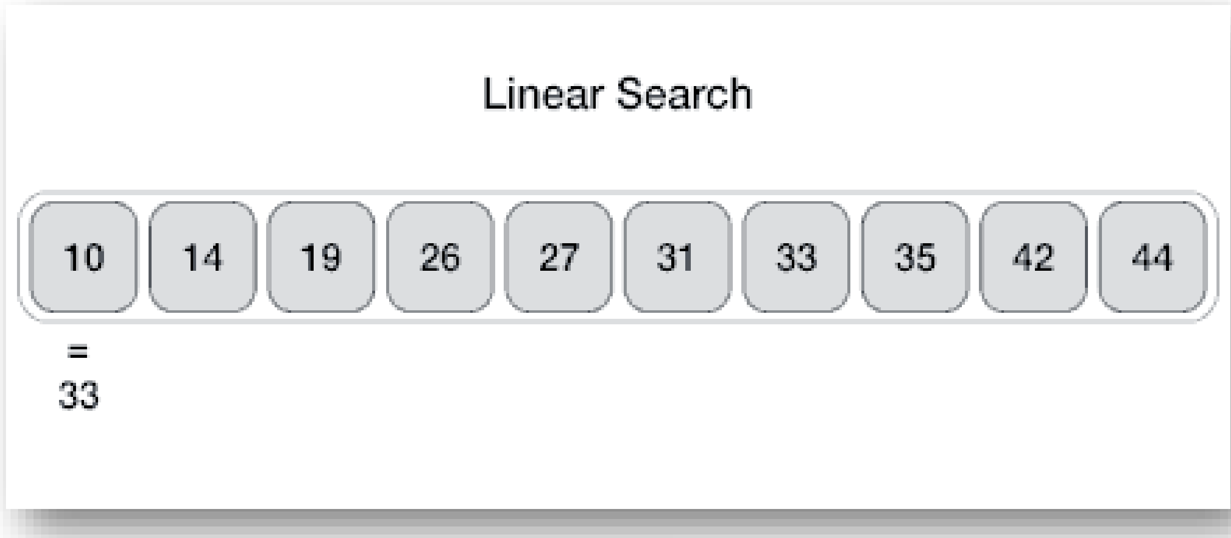
  Total for worst-case:

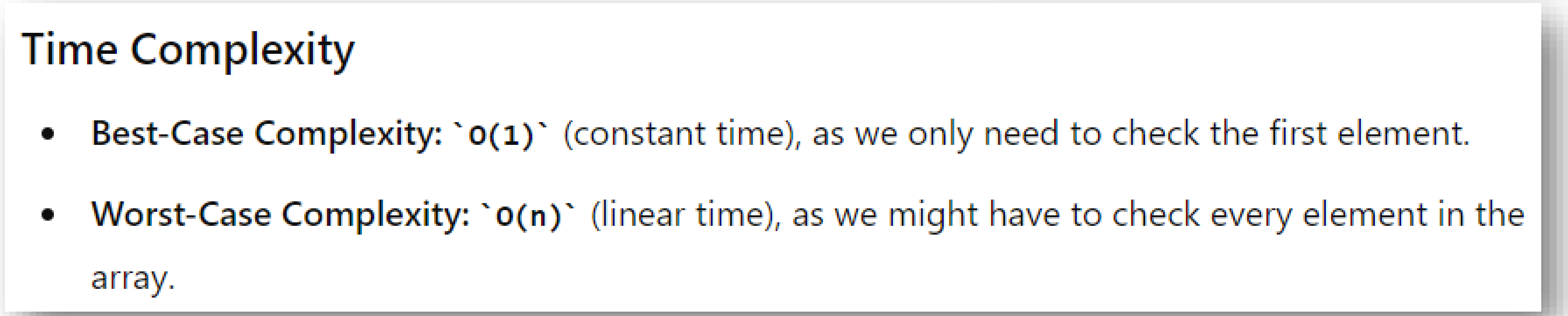
$$T_{\text{worst}} = (n \times c) + (n \times c) + c = (2n + 1) \times c$$

Explore best and worst-case complexity



```
ALGORITHM  $SequentialSearch(A[0..n-1], K)$
    //Searches for a given value in a given array by sequential search
    //Input: An array $A[0..n-1]$ and a search key $K$
    //Output: The index of the first element in $A$ that matches $K$
    //          or $-1$ if there are no matching elements
    $i \leftarrow 0$
    while $i < n$ and $A[i] \neq K$ do
        $i \leftarrow i + 1$
    if $i < n$ return $i$
    else return $-1$
```

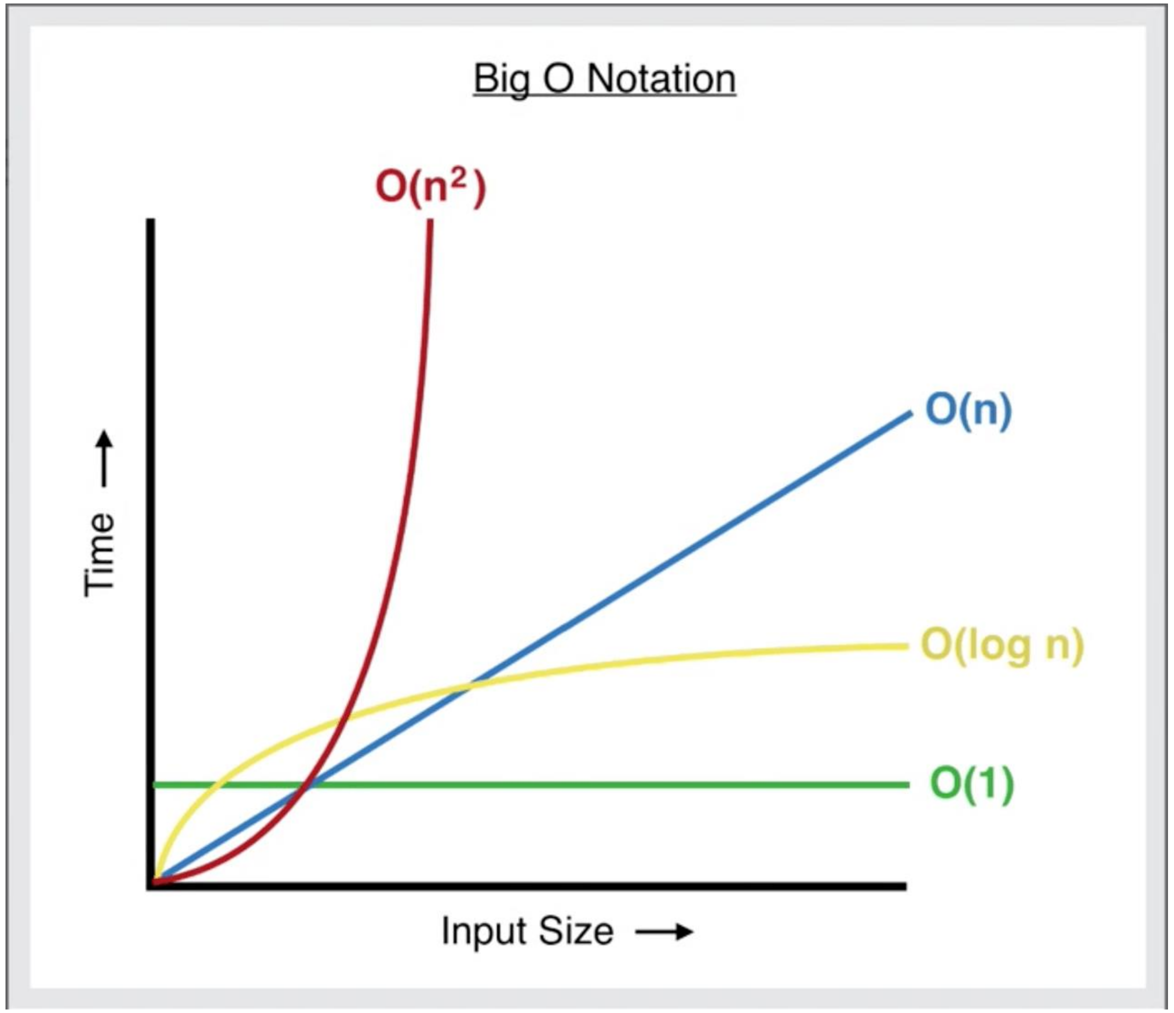




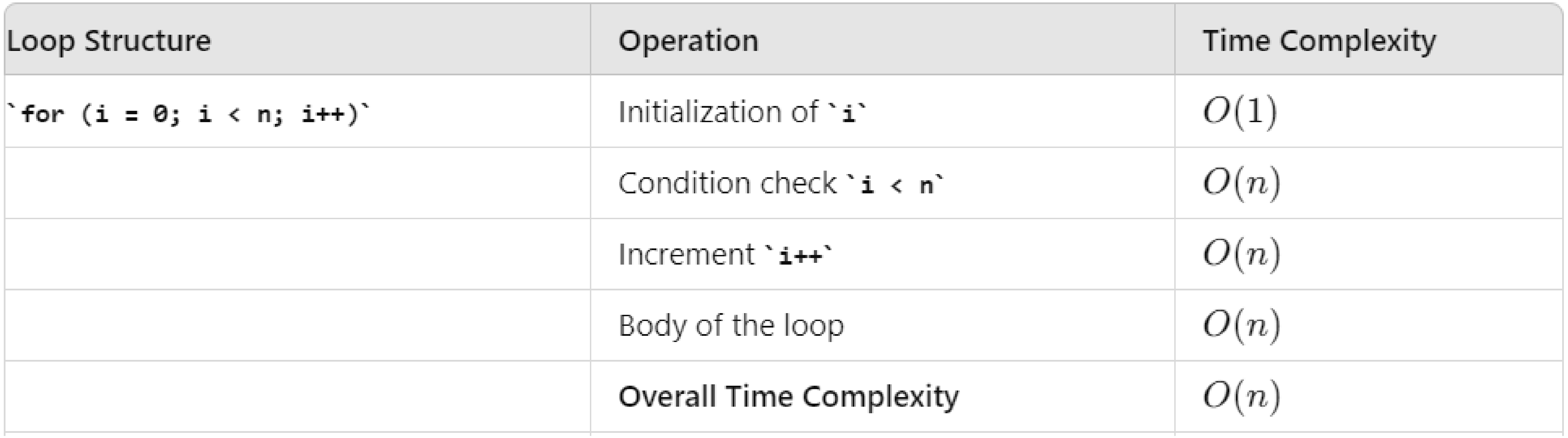
Time Complexity

- **Best-Case Complexity: `O(1)`** (constant time), as we only need to check the first element.
- **Worst-Case Complexity: `O(n)`** (linear time), as we might have to check every element in the array.

More examples of time complexity

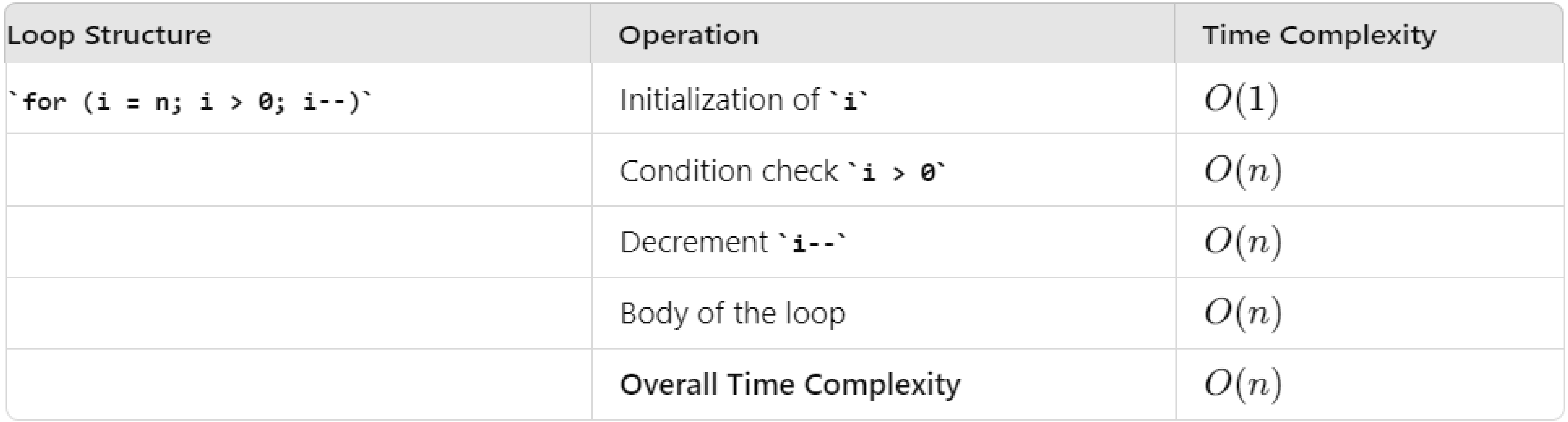


Loops



Loop Structure	Operation	Time Complexity
`for (i = 0; i < n; i++)`	Initialization of `i`	$O(1)$
	Condition check `i < n`	$O(n)$
	Increment `i++`	$O(n)$
	Body of the loop	$O(n)$
	Overall Time Complexity	$O(n)$

Loops

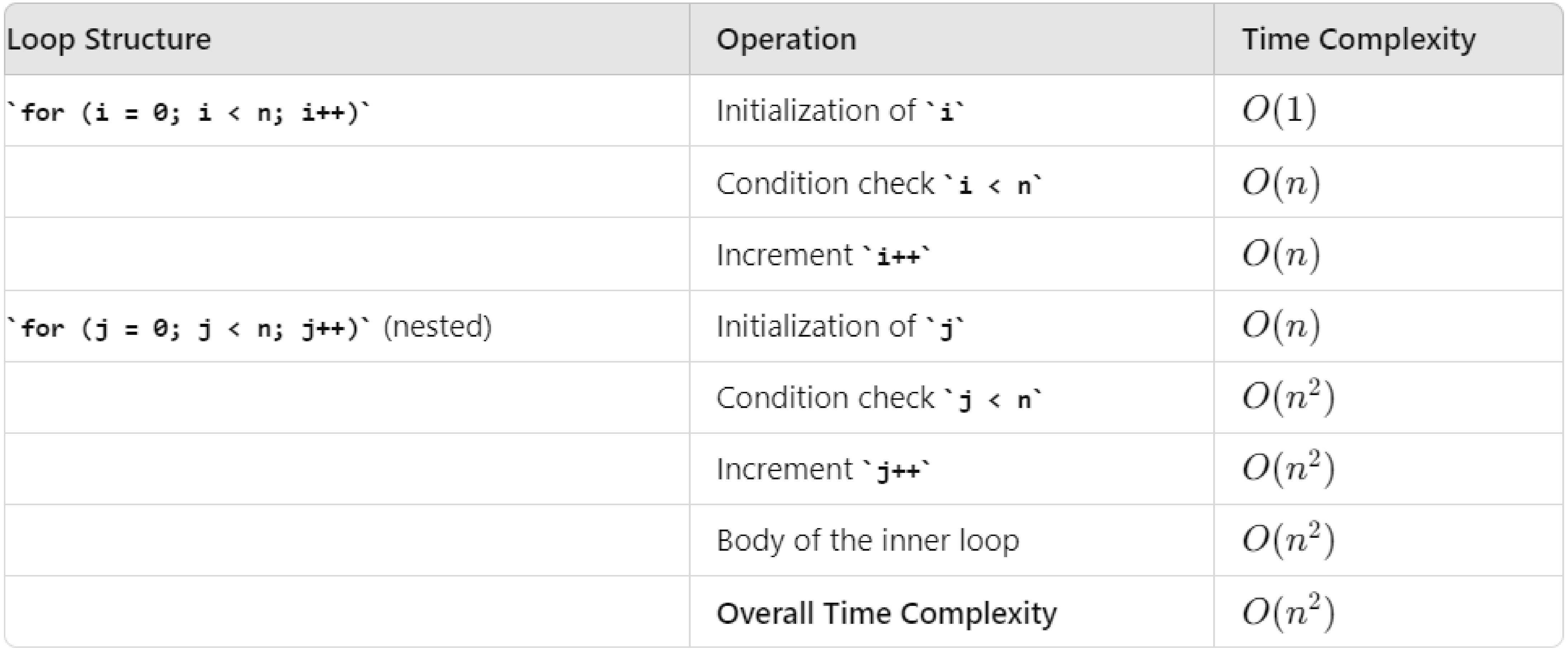


Loop Structure	Operation	Time Complexity
`for (i = n; i > 0; i--)`	Initialization of `i`	$O(1)$
	Condition check `i > 0`	$O(n)$
	Decrement `i--`	$O(n)$
	Body of the loop	$O(n)$
	Overall Time Complexity	$O(n)$

Loops

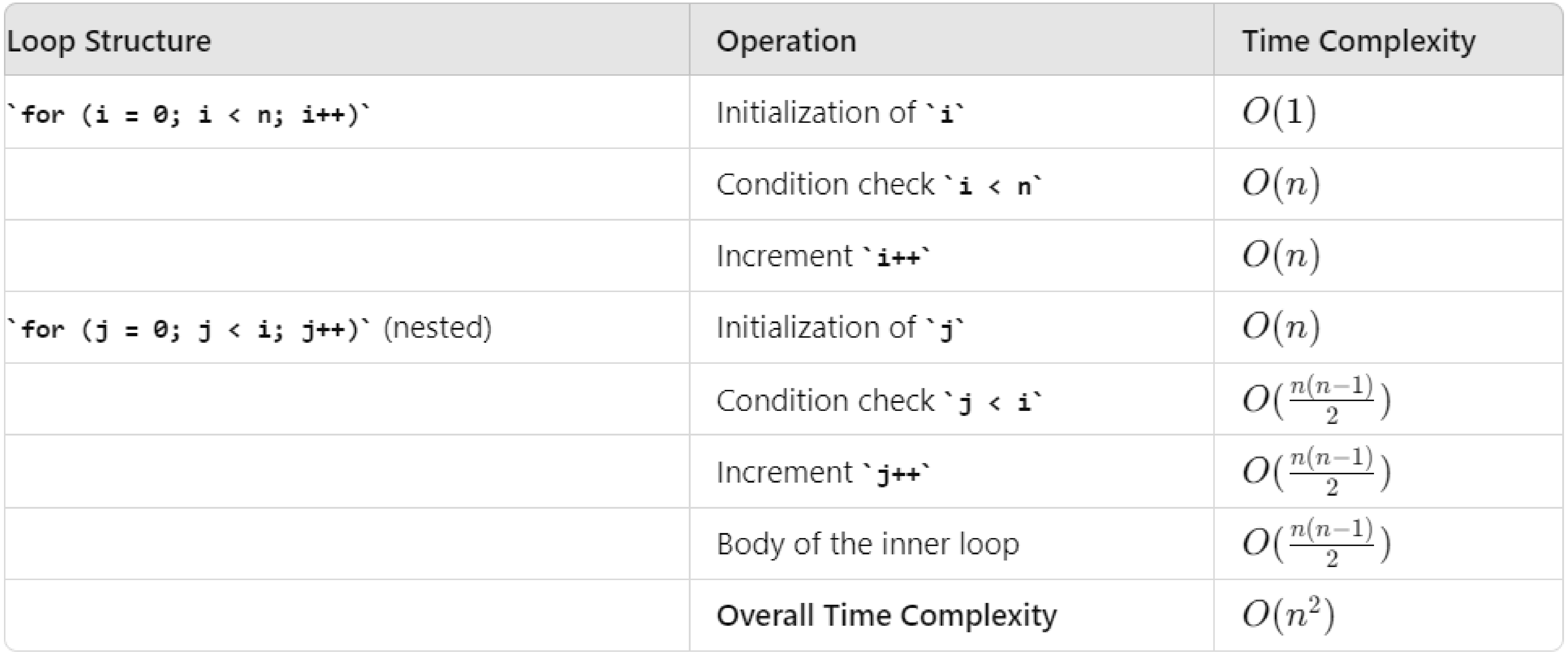
Loop Structure	Operation	Time Complexity
`for (i = 1; i < n; i = i + 2)`	Initialization of `i`	$O(1)$
	Condition check `i < n`	$O(\frac{n}{2})$
	Increment `i = i + 2`	$O(\frac{n}{2})$
	Body of the loop	$O(\frac{n}{2})$
	Overall Time Complexity	$O(\frac{n}{2})$ or $O(n)$

Nested loop



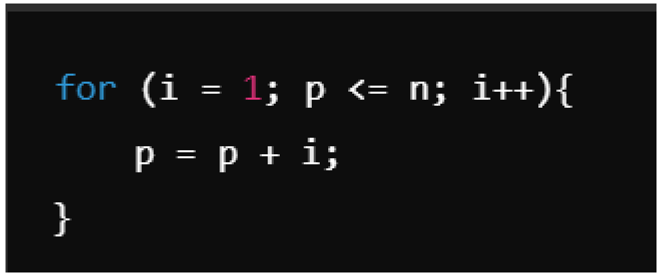
Loop Structure	Operation	Time Complexity
`for (i = 0; i < n; i++)`	Initialization of `i`	$O(1)$
	Condition check `i < n`	$O(n)$
	Increment `i++`	$O(n)$
`for (j = 0; j < n; j++)` (nested)	Initialization of `j`	$O(n)$
	Condition check `j < n`	$O(n^2)$
	Increment `j++`	$O(n^2)$
	Body of the inner loop	$O(n^2)$
	Overall Time Complexity	$O(n^2)$

Nested loop



Loop Structure	Operation	Time Complexity
`` `for (i = 0; i < n; i++)` ``	Initialization of `` `i` ``	$O(1)$
	Condition check `` `i < n` ``	$O(n)$
	Increment `` `i++` ``	$O(n)$
`` `for (j = 0; j < i; j++)` `` (nested)	Initialization of `` `j` ``	$O(n)$
	Condition check `` `j < i` ``	$O(\frac{n(n-1)}{2})$
	Increment `` `j++` ``	$O(\frac{n(n-1)}{2})$
	Body of the inner loop	$O(\frac{n(n-1)}{2})$
	Overall Time Complexity	$O(n^2)$

Loop



```
for (i = 1; p <= n; i++){
    p = p + i;
}
```

Iteration	i	Previous p	New p (cumulative sum)
1	1	0	1
2	2	1	1 + 2 = 3
3	3	3	1 + 2 + 3 = 6
4	4	6	1 + 2 + 3 + 4 = 10
5	5	10	1 + 2 + 3 + 4 + 5 = 15 (exceeds `n`, loop ends)

The sum of the first k positive integers is given by:

$$p = 1 + 2 + 3 + \ldots + k$$

The equation for the sum of the first k positive integers is:

$$p = \frac{k(k+1)}{2}$$

When the loop ends, p becomes greater than n. The sum of the first k integers is:

$$p = 1 + 2 + 3 + \ldots + k = \frac{k(k+1)}{2}$$

To exceed n:

$$\frac{k(k+1)}{2} > n$$

For large k, $k(k+1) \approx k^2$, so:

$$\frac{k^2}{2} > n$$

Thus:

$$k^2 > 2n$$

Taking the square root of both sides:

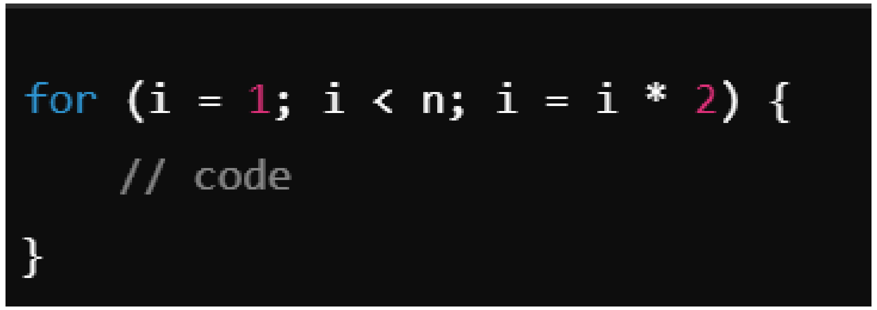
$$k > \sqrt{2n}$$

In big-O notation, we simplify this to:

$$k > \sqrt{n}$$

So, the time complexity of the loop is $O(\sqrt{n})$.

Code complexity



```
for (i = 1; i < n; i = i * 2) {
    // code
}
```

Total Number of Iterations

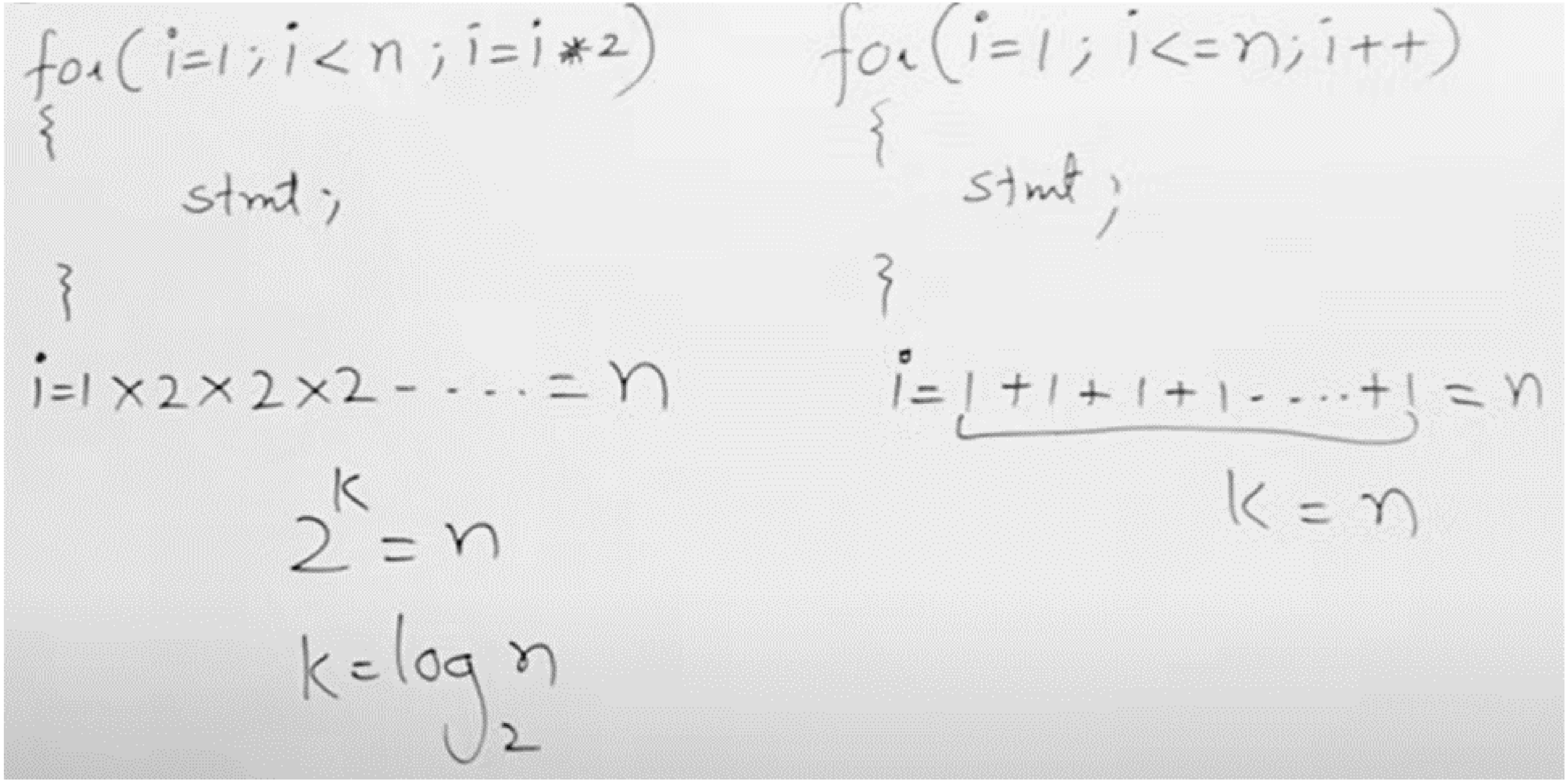
- The value of `i` follows the sequence: $1, 2, 4, 8, 16, ..$

  This means that after k iterations, $i = 2^k$.
- The loop condition is $i < n$, so $2^k < n$.
- Solving for k, we get $k < \log_2(n)$.
- Thus, the number of iterations is $\log_2(n)$.

Overall Time Complexity

- **Overall Time Complexity:** $O(\log n)$

Another simple way



```
for(i=1; i<n; i=i*2)
{
      stmt;
}
```

$i = 1 \times 2 \times 2 \times 2 - \cdots = n$

$2^k = n$

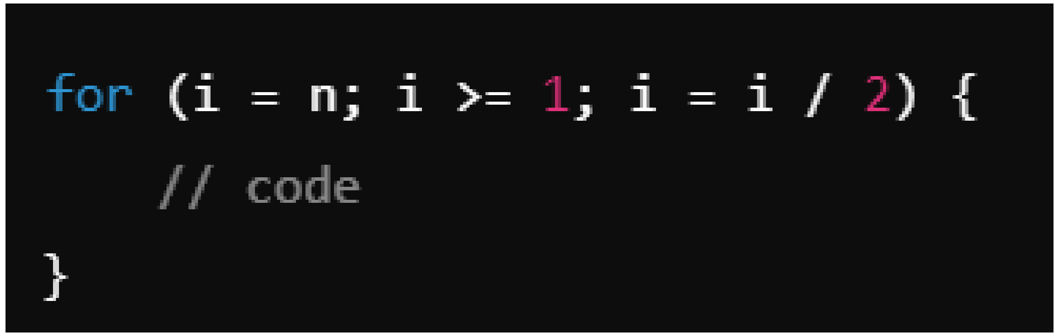
$k = \log_2 n$

```
for(i=1; i<=n; i++)
{
      stmt;
}
```

$i = \underbrace{1 + 1 + 1 + 1 \cdots + 1}_{} = n$

$k = n$

Code complexity



```
for (i = n; i >= 1; i = i / 2) {
    // code
}
```

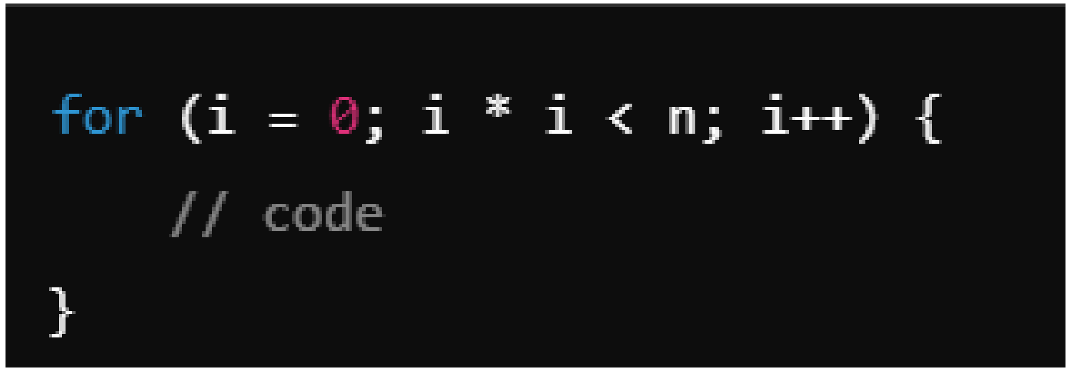
Total Number of Iterations

- The value of `i` follows the sequence: $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$. After k iterations, $i = \frac{n}{2^k}$.
- The loop condition is $i \geq 1$, so $\frac{n}{2^k} \geq 1$.
- Solving for k, we get $2^k \leq n$.
- Taking the logarithm base 2, we get $k \leq \log_2(n)$.
- Thus, the number of iterations is $\log_2(n)$.

Overall Time Complexity

- **Overall Time Complexity**: $O(\log n)$

Code complexity



```
for (i = 0; i * i < n; i++) {
    // code
}
```

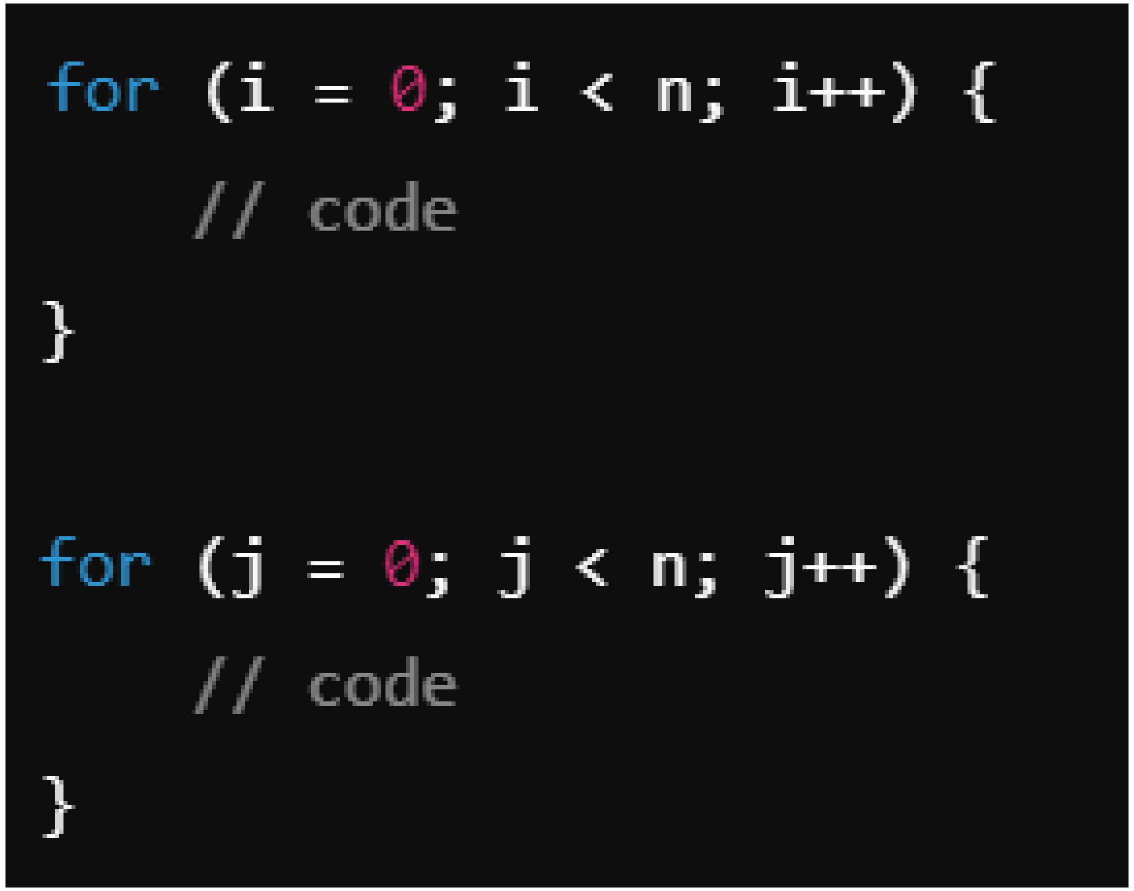
Total Number of Iterations

- The loop continues as long as $i^2 < n$.
- The largest value of `i` will be approximately $\sqrt{n}$, because when i is around $\sqrt{n}$, i^2 will b close to n.
- Therefore, the number of iterations is approximately $\sqrt{n}$.

Overall Time Complexity

- **Overall Time Complexity:** $O(\sqrt{n})$

Code complexity



```
for (i = 0; i < n; i++) {
    // code
}

for (j = 0; j < n; j++) {
    // code
}
```

Combined Time Complexity

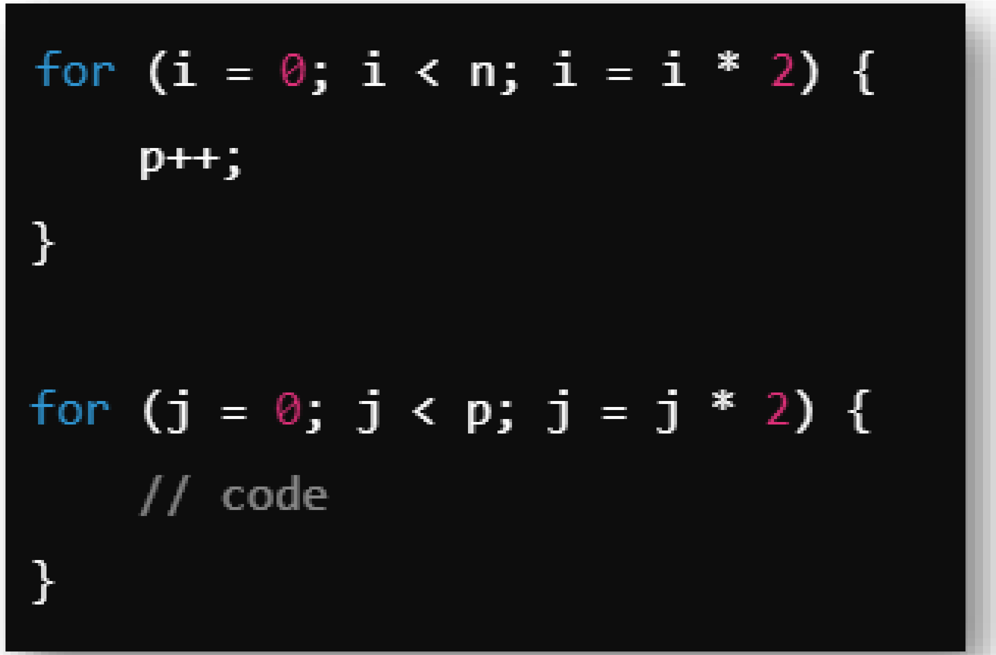
Since these loops are not nested but sequential:

- **First Loop**: $O(n)$
- **Second Loop**: $O(n)$

When combined, the overall time complexity is the sum of both loops:

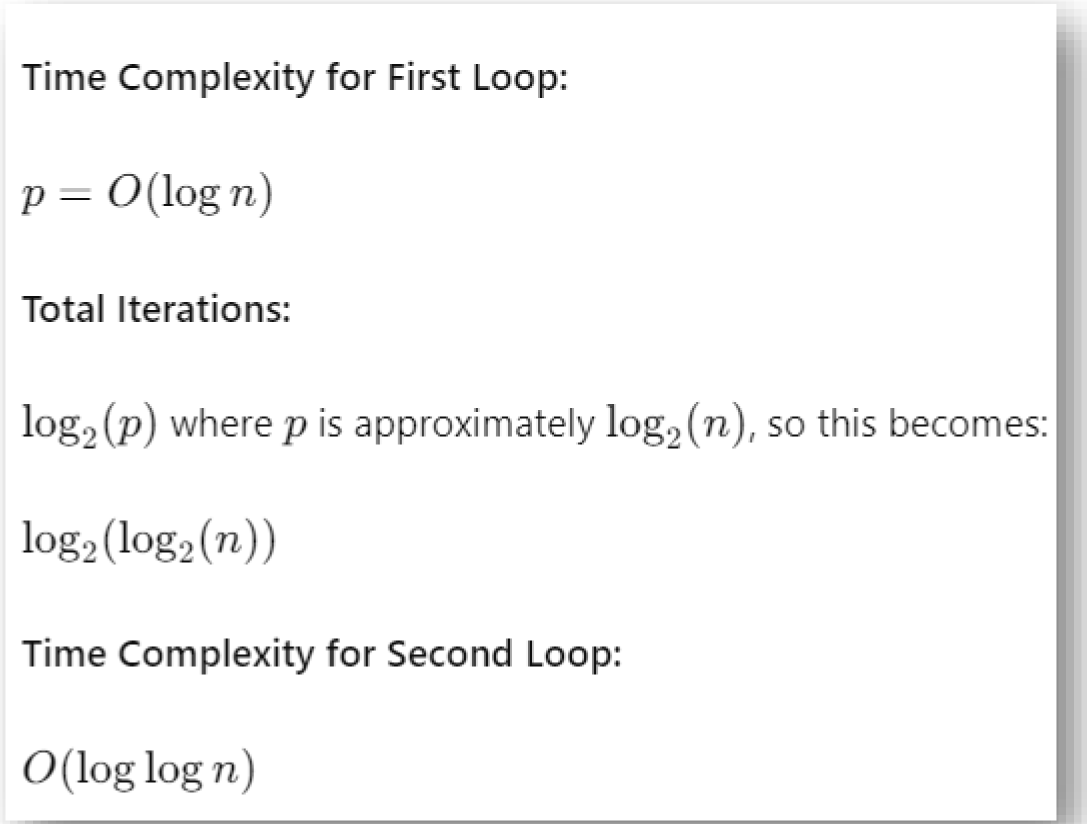
Overall Time Complexity: $O(n) + O(n) = O(n)$

Code complexity



```
for (i = 0; i < n; i = i * 2) {
    p++;
}

for (j = 0; j < p; j = j * 2) {
    // code
}
```



Time Complexity for First Loop:

$p = O(\log n)$

Total Iterations:

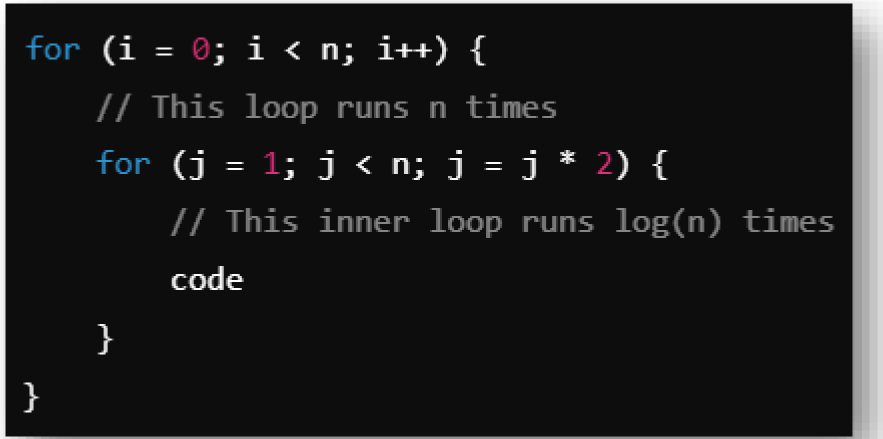
$\log_2(p)$ where p is approximately $\log_2(n)$, so this becomes:

$\log_2(\log_2(n))$

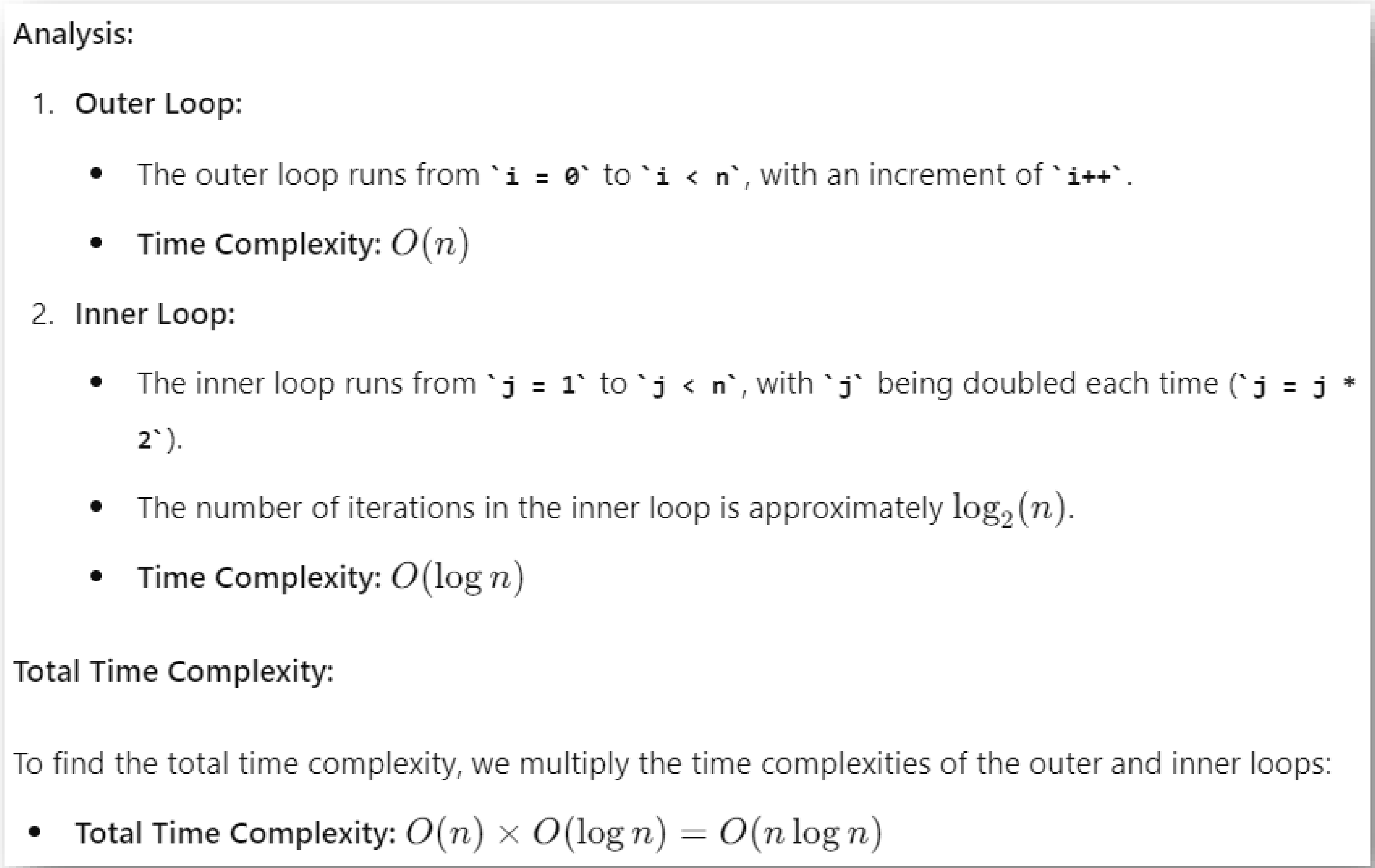
Time Complexity for Second Loop:

$O(\log \log n)$

Code complexity



```
for (i = 0; i < n; i++) {
    // This loop runs n times
    for (j = 1; j < n; j = j * 2) {
        // This inner loop runs log(n) times
        code
    }
}
```



Analysis:

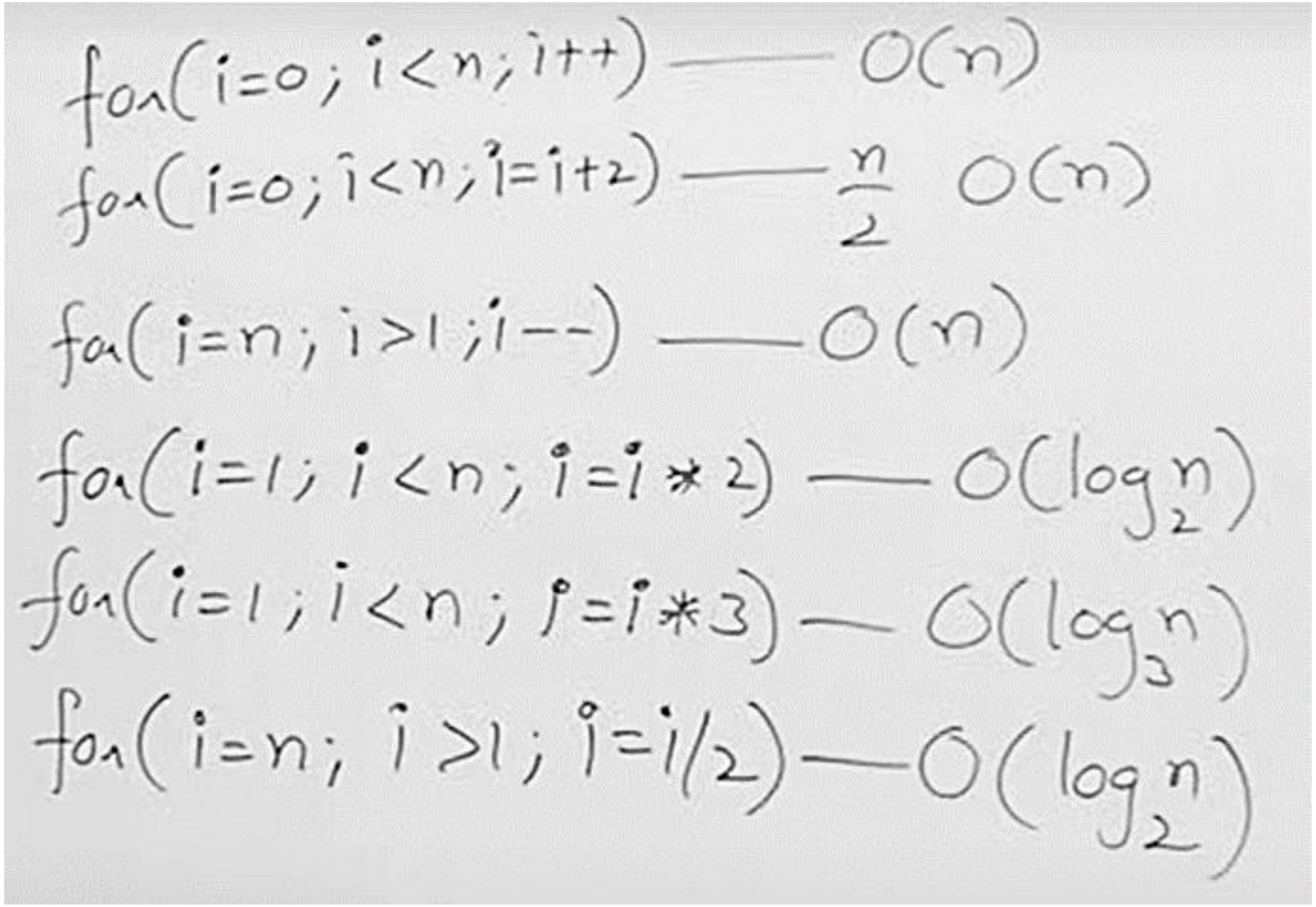
1. **Outer Loop:**
   - The outer loop runs from `i = 0` to `i < n`, with an increment of `i++`.
   - **Time Complexity:** $O(n)$
2. **Inner Loop:**
   - The inner loop runs from `j = 1` to `j < n`, with `j` being doubled each time (`j = j * 2`).
   - The number of iterations in the inner loop is approximately $\log_2(n)$.
   - **Time Complexity:** $O(\log n)$

Total Time Complexity:

To find the total time complexity, we multiply the time complexities of the outer and inner loops:

- **Total Time Complexity:** $O(n) \times O(\log n) = O(n \log n)$

Code complexity



for(i=0; i<n; i++) —— $O(n)$

for(i=0; i<n; i=i+2) —— $\frac{n}{2}$ $O(n)$

for(i=n; i>1; i--) —— $O(n)$

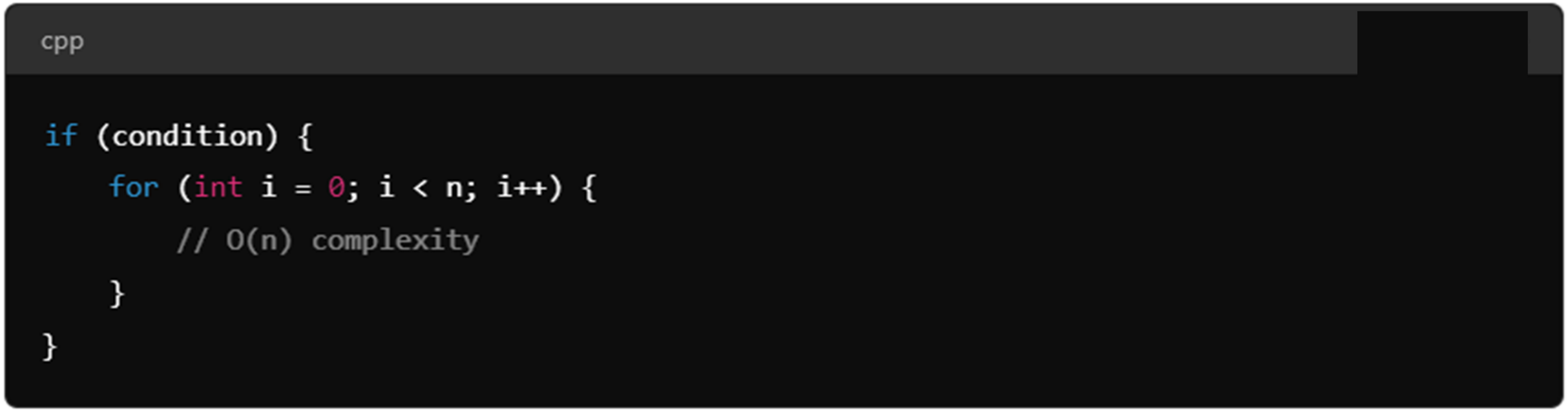
for(i=1; i<n; i=i*2) —— $O(\log_2 n)$

for(i=1; i<n; i=i*3) —— $O(\log_3 n)$

for(i=n; i>1; i=i/2) —— $O(\log_2 n)$

Analysis of if statements

- if Statement: An if statement is used to execute a block of code conditionally. It does not directly affect the overall time complexity, as it only alters the control flow based on a condition. The time complexity of an if statement depends on the complexity of the block of code that is executed when the condition is true.



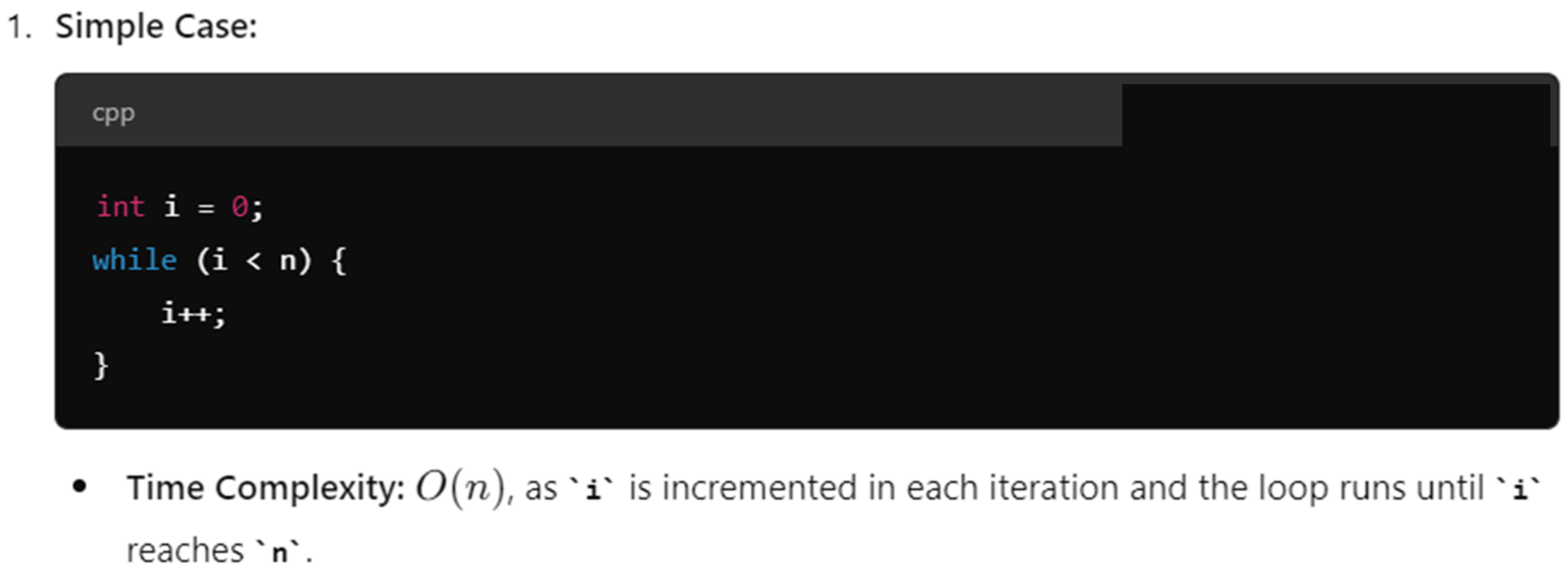
cpp

```cpp
if (condition) {
    for (int i = 0; i < n; i++) {
        // O(n) complexity
    }
}
```

- **Overall Time Complexity:** $O(n)$ if the condition is true, but since it depends on the condition, the complexity is context-specific.

Analysis of while Statement

- while Statement: A while loop repeatedly executes a code block as long as a specified condition remains true. The time complexity of a while loop depends on how many times the loop executes and the complexity of the code inside the loop.



1. **Simple Case:**

```cpp
int i = 0;
while (i < n) {
    i++;
}
```

- **Time Complexity:** $O(n)$, as `i` is incremented in each iteration and the loop runs until `i` reaches `n`.

Analysis of while Statement

Logarithmic Case:

cpp

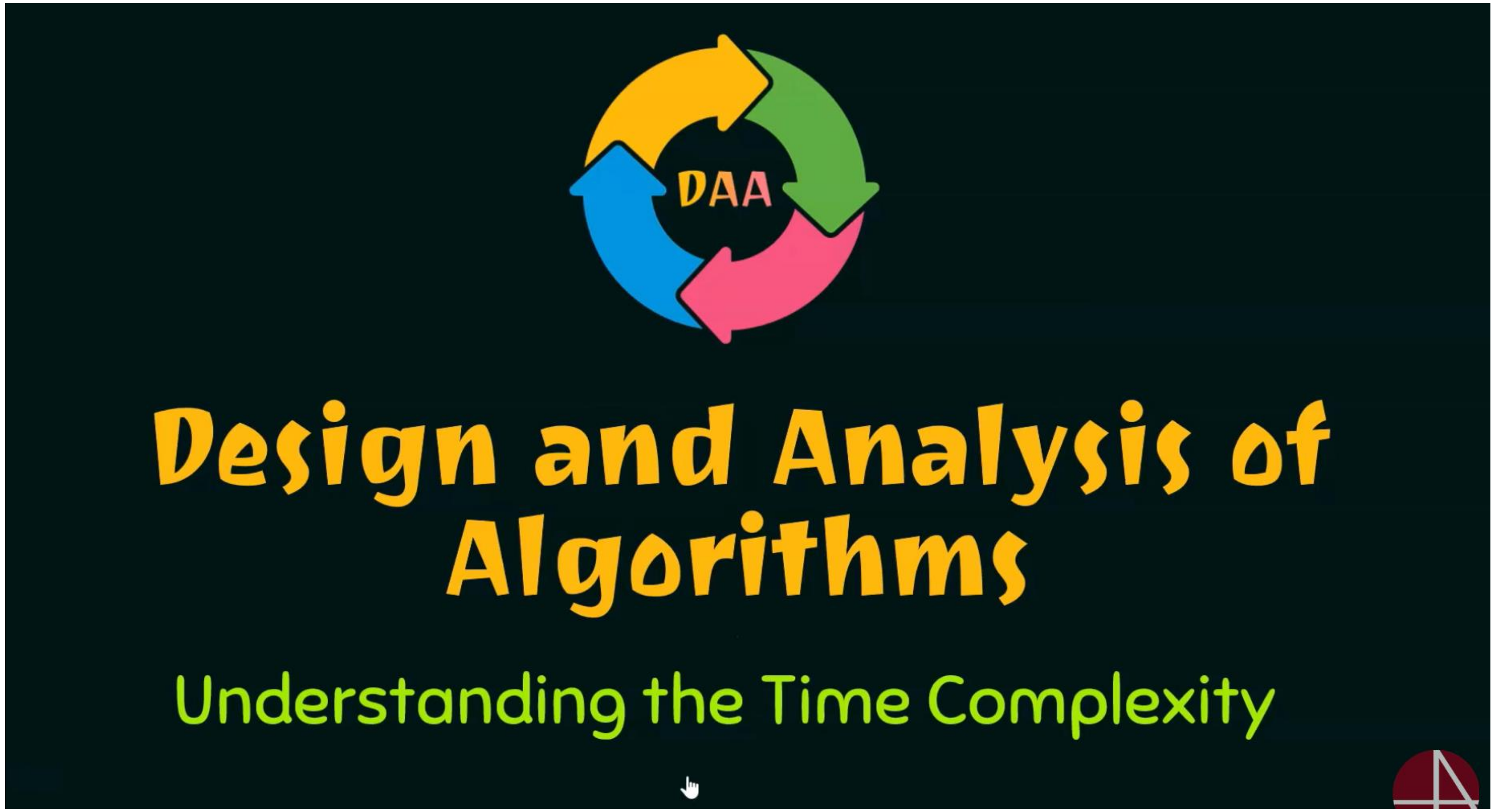
```cpp
int i = 1;
while (i < n) {
    i *= 2;
}
```

- **Time Complexity:** $O(\log n)$, as `i` is multiplied by 2 in each iteration, leading to a logarithmic number of iterations.

Complex Case:

```cpp
int i = 0;
while (i < n) {
    for (int j = 0; j < i; j++) {
        // O(i) complexity
    }
    i++;
}
```

- Time Complexity:
  - Inner loop complexity: $O(i)$
  - Outer loop complexity: Sum of $O(i)$ from $i = 1$ to $n - 1$ is $O(n^2)$
  - **Overall Time Complexity:** $O(n^2)$


DAA
Design and Analysis of Algorithms
Understanding the Time Complexity

Understanding the Time Complexity of an Algorithm

https://www.youtube.com/watch?v=pULw1Fpru0E

Some resources

- https://www.youtube.com/watch?v=9TlHvipP5yA
- https://www.youtube.com/watch?v=9SgLBjXqwd4
- https://www.youtube.com/watch?v=p1EnSvS3urU