Time Complexity

Big-O notation

Table of contents

1 Big-O notation

1.1 Time Complexity

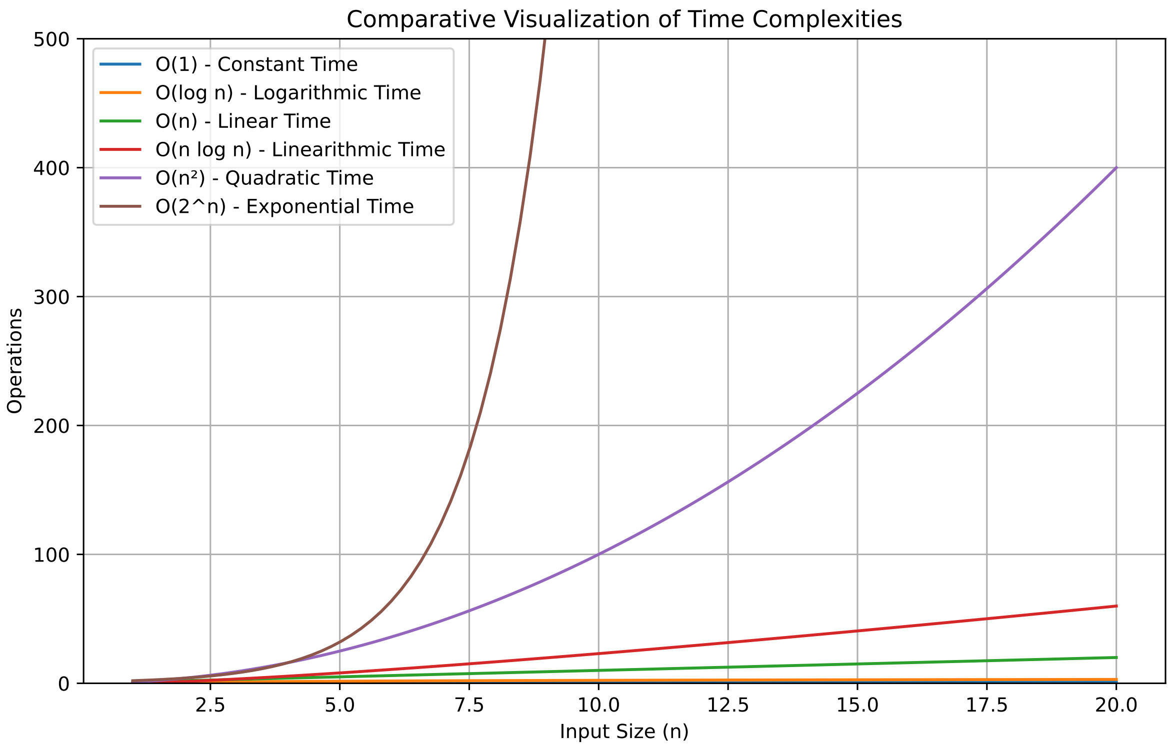
- The **Big-O Notation** describes how an algorithm's runtime grows as the input size increases.
- Understanding time complexity helps in selecting the right algorithm for a specific task.

1.2 Time Complexities Comparison Table

Complexity	Class Name	Description	Real-World Example	Graph
$O(1)$	Constant Time	Always takes the same time regardless of input size	Accessing an element in an array	Flat Line
$O(\log n)$	Logarithmic Time	Runtime grows slowly with input size	Binary search in a phone book	Slowly Increasing Curve
$O(n)$	Linear Time	Runtime grows directly proportional to input size	Reading through a list of names in a document	Straight Line
$O(n \log n)$	Linearithmic Time	Efficient sorting, slower than linear	Efficient sorting algorithms like Merge Sort	Rising Curve
$O(n^2)$	Quadratic Time	Runtime grows exponentially with input size	Bubble Sort on a long list of elements	Steep Curve
$O(2^n)$	Exponential Time	Runtime doubles with each additional element	Recursive solutions to combinatorial problems	Exponential Curve
$O(n!)$	Factorial Time	Runtime grows factorially with input size	Generating all permutations of a list	Extremely Steep Curve

1.3 Visualization of Time Complexities

```python
import matplotlib.pyplot as plt
import numpy as np

# Define input sizes
n = np.linspace(1, 20, 100)

# Define various time complexities
O_1 = np.ones_like(n)
O_log_n = np.log(n)
O_n = n
O_n_log_n = n * np.log(n)
O_n2 = n**2
O_2n = 2**n

# Plot the complexities
plt.figure(figsize=(10, 6))

plt.plot(n, O_1, label="O(1) - Constant Time")
plt.plot(n, O_log_n, label="O(log n) - Logarithmic Time")
plt.plot(n, O_n, label="O(n) - Linear Time")
plt.plot(n, O_n_log_n, label="O(n log n) - Linearithmic Time")
plt.plot(n, O_n2, label="O(n²) - Quadratic Time")
plt.plot(n, O_2n, label="O(2^n) - Exponential Time")

# Add labels and title
plt.ylim(0, 500)  # Limit the y-axis for better comparison
plt.xlabel("Input Size (n)")
plt.ylabel("Operations")
plt.title("Comparative Visualization of Time Complexities")
plt.legend(loc="upper left")

plt.grid(True)
plt.show()
```


Comparative Visualization of Time Complexities
O(1) - Constant Time
O(log n) - Logarithmic Time
O(n) - Linear Time
O(n log n) - Linearithmic Time
O(n²) - Quadratic Time
O(2^n) - Exponential Time
Operations
Input Size (n)
0
100
200
300
400
500
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

2 Bubble Sort

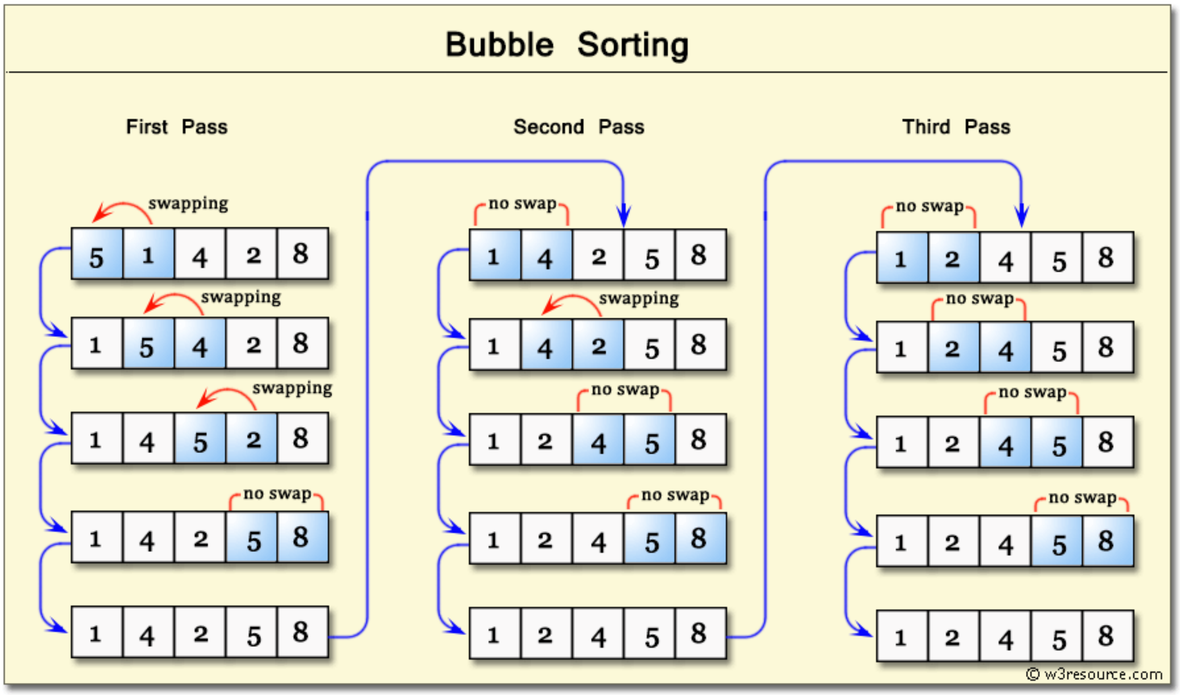
Figure 1: Bubbles

2.1 Introduction to Bubble Sort

- **Bubble Sort** is a simple, comparison-based sorting algorithm. It repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order.
- The smaller elements "bubble" to the top, while larger elements "sink" to the bottom. This process repeats until the list is sorted.

2.2 How Bubble Sort Works

1. Start at the beginning of the list.
2. Compare each pair of adjacent elements.
3. If they are in the wrong order, swap them.
4. Repeat this process until no more swaps are needed.


Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
no swap
© w3resource.com

2.3 Bubble Sort Flowchart

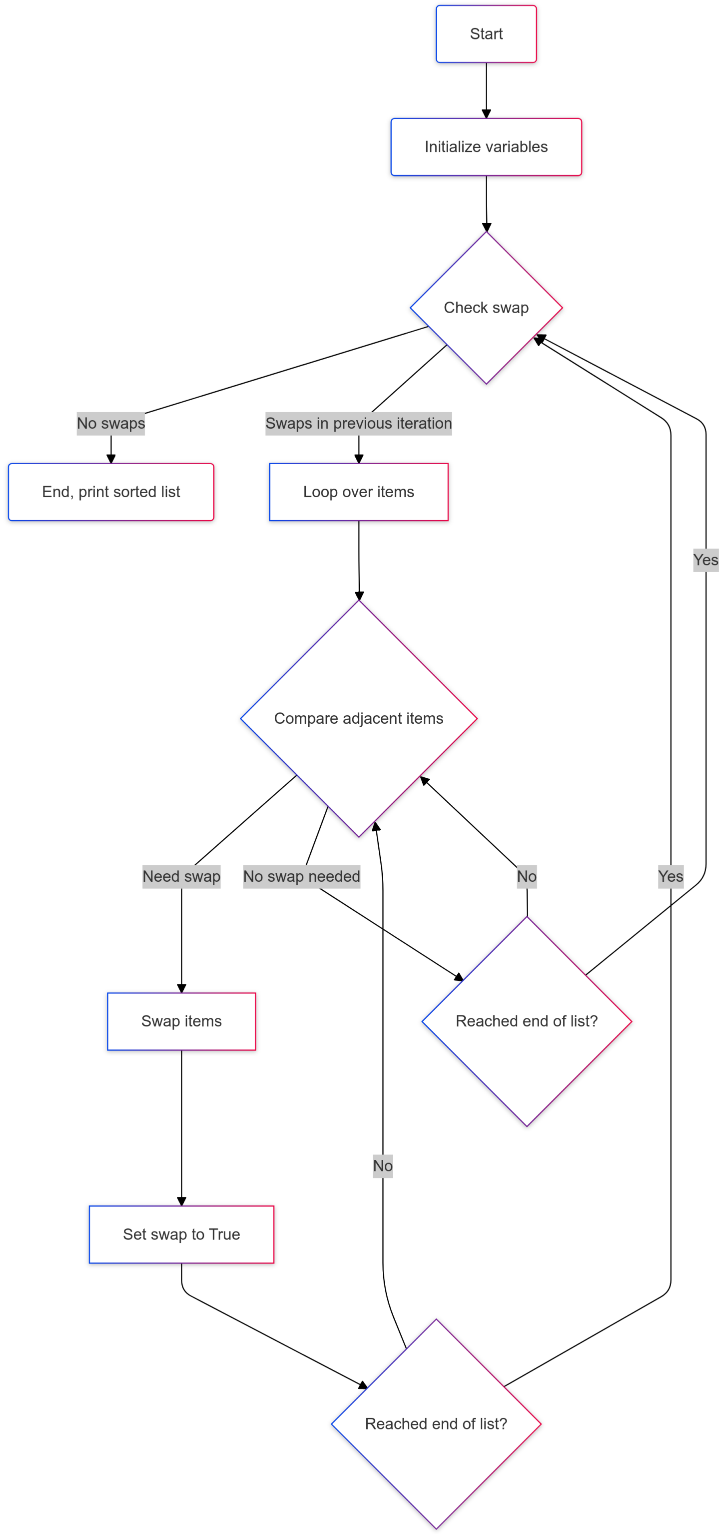


Figure 2: Bubble Sort Flowchart

2.4 Bubble Sort Activity

- We need 7-10 student volunteers of different heights to stand in a row.
- Use the Bubble Sort algorithm to sort the volunteers by height from shortest to tallest.
- Compare adjacent volunteers, swap if necessary, and repeat until sorted.

Figure 3: Celebrity Heights

2.5 Visualizing Bubble Sort Swaps

- Let's sort the list `[5, 3, 8, 6, 7, 2]` step-by-step:

1. **First Pass**:
   - Compare 5 and 3 → Swap → `[3, 5, 8, 6, 7, 2]`
   - Compare 5 and 8 → No swap.
   - Compare 8 and 6 → Swap → `[3, 5, 6, 8, 7, 2]`
   - Compare 8 and 7 → Swap → `[3, 5, 6, 7, 8, 2]`
   - Compare 8 and 2 → Swap → `[3, 5, 6, 7, 2, 8]`
   - List after first pass: `[3, 5, 6, 7, 2, 8]`
2. **Second Pass**:
   - Compare 3 and 5 → No swap.
   - Compare 5 and 6 → No swap.
   - Compare 6 and 7 → No swap.
   - Compare 7 and 2 → Swap → `[3, 5, 6, 2, 7, 8]`
   - List after second pass: `[3, 5, 6, 2, 7, 8]`
3. **Third Pass**:
   - Compare 3 and 5 → No swap.
   - Compare 5 and 6 → No swap.

- Compare 6 and 2 → Swap → `[3, 5, 2, 6, 7, 8]`
- List after third pass: `[3, 5, 2, 6, 7, 8]`

4. **Final Pass**:

   - The final pass will continue comparing adjacent elements until no more swaps are needed.

2.6 Bubble Sort in Python

Write a Python program to perform a bubble sort

- Input: `[5, 3, 8, 6, 7, 2]`
- Expected Output: `[2, 3, 5, 6, 7, 8]`

```python
# Initialize a list of unsorted items
items = [5, 3, 8, 6, 7, 2]

# Determine the number of items in the list
n = len(items)

# Initialize a boolean variable to track swaps
swap = True

# Continue looping while swaps occur
while swap:
    swap = False  # Reset swap flag
    for i in range(1, n):
        if items[i-1] > items[i]:  # Compare adjacent items
            items[i-1], items[i] = items[i], items[i-1]  # Swap if needed
            swap = True  # Set swap flag to True

# Print the sorted list
print(items)
```

```
[2, 3, 5, 6, 7, 8]
```

2.7 Time Complexity of Bubble Sort

- **Best Case**: $O(n)$ – This occurs when the list is already sorted. In this case, Bubble Sort makes only one pass.

- **Worst Case**: $O(n^2)$ – This occurs when the list is in reverse order. Every element needs to be compared and swapped.

3 Matrix Multiplication

Figure 4: The Matrix Code

3.1 Introduction to Matrix Multiplication

- **Matrix Multiplication** is an operation that combines two matrices to produce a third matrix.
- Matrices represent data in rows and columns, and multiplying them allows us to transform or process data in powerful ways.

3.2 How Matrix Multiplication Works

3.2.1 Example:

Given two matrices A and B:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Their product $C = A \times B$ is:

$$C = \begin{bmatrix} (1 \times 5 + 2 \times 7) & (1 \times 6 + 2 \times 8) \\ (3 \times 5 + 4 \times 7) & (3 \times 6 + 4 \times 8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

3.2.2 Formula:

For each element $C_{i,j}$ in the resulting matrix:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

Where A is an $m \times n$ matrix, and B is an $n \times p$ matrix.

3.2.3 Notes:

- The product of $A \times B$ will be of dimension $m \times p$.
- The number of columns in matrix A (i.e., n) must match the number of rows in matrix B for the multiplication to be possible.

3.3 Matrix Multiplication in Python

Write a Python function `matrix_multiply` that takes to input matrices, `A` and `B` and returns the product of their matrix multiplication

- **Example:**

```python
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
matrix_multiply(A, B)
```

```
[[19, 22], [43, 50]]
```

- **Answer:**

```python
def matrix_multiply(A, B):
  # Initialize an empty list to store the result of the matrix multiplication
  result = []

  # Create a matrix with the same number of rows as A and columns as B,
   ↪  filled with zeros
  for i in range(len(A)):
    row = []  # Initialize a new row
    for j in range(len(B[0])):  # Iterate over columns of B
      row.append(0)  # Append 0 to represent the initial state
    result.append(row)  # Append the row to the result matrix

  # Perform matrix multiplication
  for i in range(len(A)):  # Loop over rows of A
    for j in range(len(B[0])):  # Loop over columns of B
      for k in range(len(B)):  # Loop over rows of B (and columns of A)
        # Multiply corresponding elements and add to the current cell in
         ↪  result
        result[i][j] += A[i][k] * B[k][j]

  return result  # Return the result matrix

# Example matrices to test the function
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
matrix_multiply(A, B)  # Expected output: [[19, 22], [43, 50]]
```

```
[[19, 22], [43, 50]]
```

- **Explanation:**

1. **Matrix Setup:** The first loop constructs a result matrix with the appropriate dimensions (rows of A × columns of B), initialized to zeros.
2. **Matrix Multiplication:** The nested loops calculate the dot product for each element in the result matrix by summing the products of corresponding elements in each row of A and column of B.
3. **Return Statement:** Finally, the function returns the resulting matrix after the multiplication.

3.4 Key Points:

- The outer loops traverse the rows and columns.

- The inner loop calculates the dot product between rows of A and columns of B.

3.5 Time Complexity of Matrix Multiplication

- For two matrices, A of size $m \times n$ and B of size $n \times p$, the time complexity of standard (naive) matrix multiplication is: $O(m \times n \times p)$
- If the matrices are square (i.e., $m = n = p$), the time complexity simplifies to $O(n^3)$.

3.6 Applications of Matrix Multiplication

- **Transformation**: In **computer graphics**, matrix multiplication is used to rotate, scale, or translate objects.
- **Data Representation**: In **machine learning**, data is often represented as matrices, such as weights in neural networks.
- **Solving Systems of Equations**: In **linear algebra**, matrix multiplication helps solve systems of equations.

4 Recursive Fibonacci

Figure 5: Nautilus shell

4.1 Recursive Fibonacci Sequence

- Write a recursive function that calculates the Fibonacci number for a given n.
- Test it with small values like $n = 5$ and $n = 6$.

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

print(fibonacci(5))  # Output: 5
print(fibonacci(6))  # Output: 8
```

```
5
8
```

4.2 Recursive Tree Visualization

- Draw the recursion tree to visualize the recursive calls $n = 5$:

```
fibonacci(5)
   |
   +-- fibonacci(4)
   |      |
   |      +-- fibonacci(3)
   |      |      |
   |      |      +-- fibonacci(2)
   |      |      |      +-- fibonacci(1)
   |      |      |      +-- fibonacci(0)
   |      |      +-- fibonacci(1)
   |      +-- fibonacci(2)
   |             +-- fibonacci(1)
   |             +-- fibonacci(0)
   +-- fibonacci(3)
          |
          +-- fibonacci(2)
          |      +-- fibonacci(1)
          |      +-- fibonacci(0)
          +-- fibonacci(1)
```

- Notice how the same values are recomputed multiple times.

4.3 Counting the Number of Recursive Calls

- Modify the **fibonacci** function to count the number of recursive calls.

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

- Number of calls for `fibonacci(5)`

```
n = 5
call_count = 0
fibonacci(n)
print("Number of recursive calls for n = ", 5, ": " , call_count)
```

```
Number of recursive calls for n =  5 :  15
```

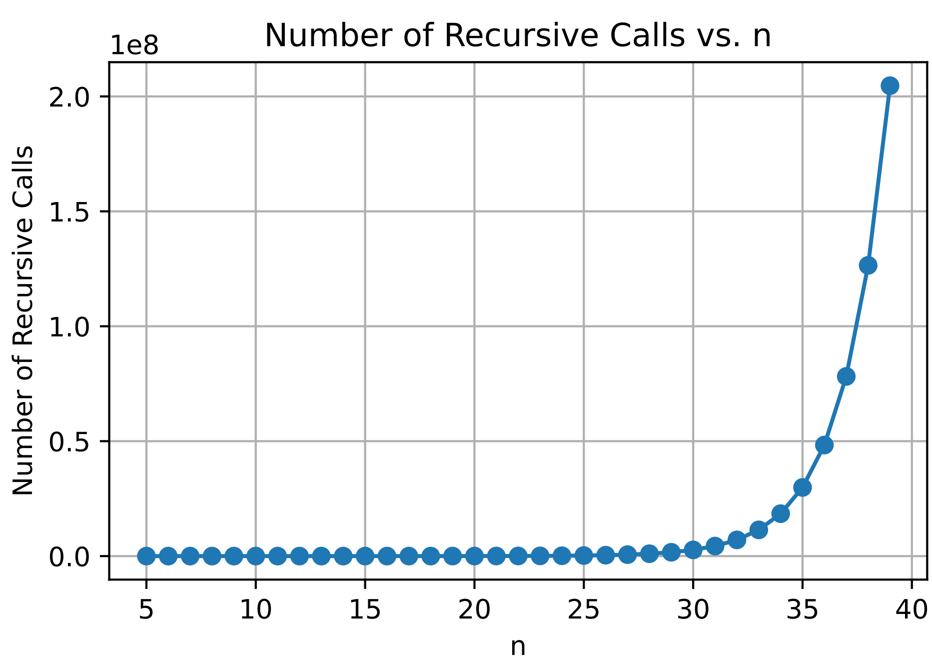
- Number of calls for `fibonacci(6)`

```
n = 6
call_count = 0
fibonacci(n)
print("Number of recursive calls for n = ", 6, ": " , call_count)
```

```
Number of recursive calls for n =  6 :  25
```

4.4 Visualizing the Number of Recursive Calls

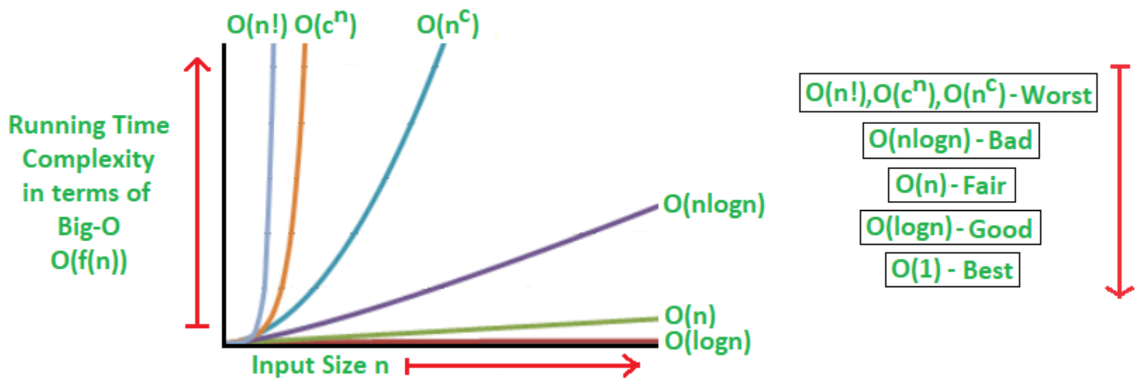
- Count the recursive calls for multiple values of n and plot the results.



4.5 Time Complexity of Fibonacci

- Every call to `fibonacci(n)` splits into two subproblems: `fibonacci(n-1)` and `fibonacci(n-2)`.
- The total number of nodes in the tree is approximately 2^n, leading to the time complexity of $O(2^n)$.

5 Summary



[Source: Big O Notation Tutorial – A Guide to Big O Analysis]