LABSHEET 1

Design and Thinking Lab

Submitted by:

Aditya Singh

(2301201182)

Under the supervision of

Aarti Sangwan

Department of Computer Science and Engineering

School of Engineering and Technology

K.R Mangalam University, Gurugram- 122001, India

2025

Basic Installation of Environment:

```
!pip install -q memory_profiler
%load_ext memory_profiler

import time
import random
import matplotlib.pyplot as plt
import numpy as np
import sys

sys.setrecursionlimit(2000)

print("Setup Complete!")
```

Output:

```
Setup Complete!
```

1. Fibonacci (Naïve Recursive)

- **Description**: This is the most direct implementation of the Fibonacci sequence, translating the mathematical formula F(n)=F(n−1)+F(n−2) directly into a recursive function.

- **Time Complexity**: O(2n). The function's runtime is exponential because it recalculates the same Fibonacci values multiple times, leading to an explosive number of function calls.

- **Space Complexity**: O(n). The space usage is linear and is determined by the maximum depth of the recursion call stack.

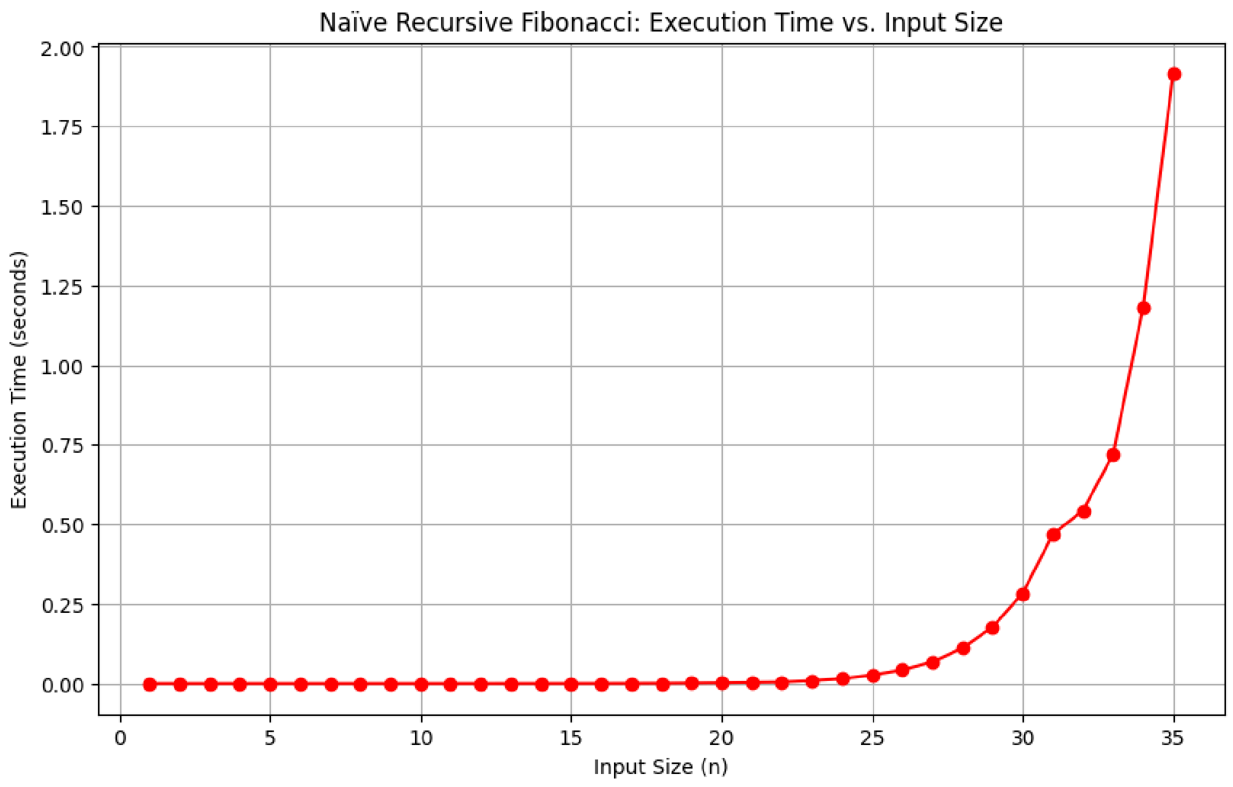
- **Suitability & Trade-offs**: This method is primarily for educational purposes to demonstrate recursion. It is extremely inefficient and unsuitable for practical use with inputs larger than approximately n=40. Its main trade-off is simplicity of understanding for extremely poor performance.

Code:

```
#Fibonacci (Naïve Recursive)

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

print("Algorithm Analysis: ")
print("Time Complexity: O(2^n) - Exponential.")
print("Space Complexity: O(n) - Linear, due to recursion stack depth.\n")

print("Demonstration:")
n_demo = 10
print(f"The {n_demo}th Fibonacci number is: {f_recursive(n_demo)}")
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = range(1, 36)
execution_times = []
for n in input_sizes:
    start_time = time.time()
    f_recursive(n)
    end_time = time.time()
    execution_times.append(end_time - start_time)

plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-', color='r')
plt.title('Naïve Recursive Fibonacci: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

print("\nMemory Profile for n=30:")
%memit f_recursive(30)
```

Output:

```
Algorithm Analysis:
Time Complexity: O(2^n) - Exponential.
Space Complexity: O(n) - Linear, due to recursion stack depth.

Demonstration:
The 10th Fibonacci number is: 55
```



```
Memory Profile for n=30:

peak memory: 123.75 MiB, increment: 0.52 MiB
```

2. Fibonacci (Dynamic Programming)

- **Description**: This version avoids the redundant calculations of the naïve approach by storing previously computed results. It uses a "bottom-up" method called tabulation, building an array of Fibonacci numbers from F(0) up to F(n).

- **Time Complexity**: O(n). The runtime is linear because each Fibonacci number up to n is calculated exactly once in a single loop.

- **Space Complexity**: O(n). Space is required to store the table of Fibonacci numbers up to n. This can be optimized to O(1) by only storing the two most recent numbers.

- **Suitability & Trade-offs**: This is the standard, efficient method for calculating Fibonacci numbers in any practical application. It is vastly more performant than the recursive version at the cost of slightly more complex code.

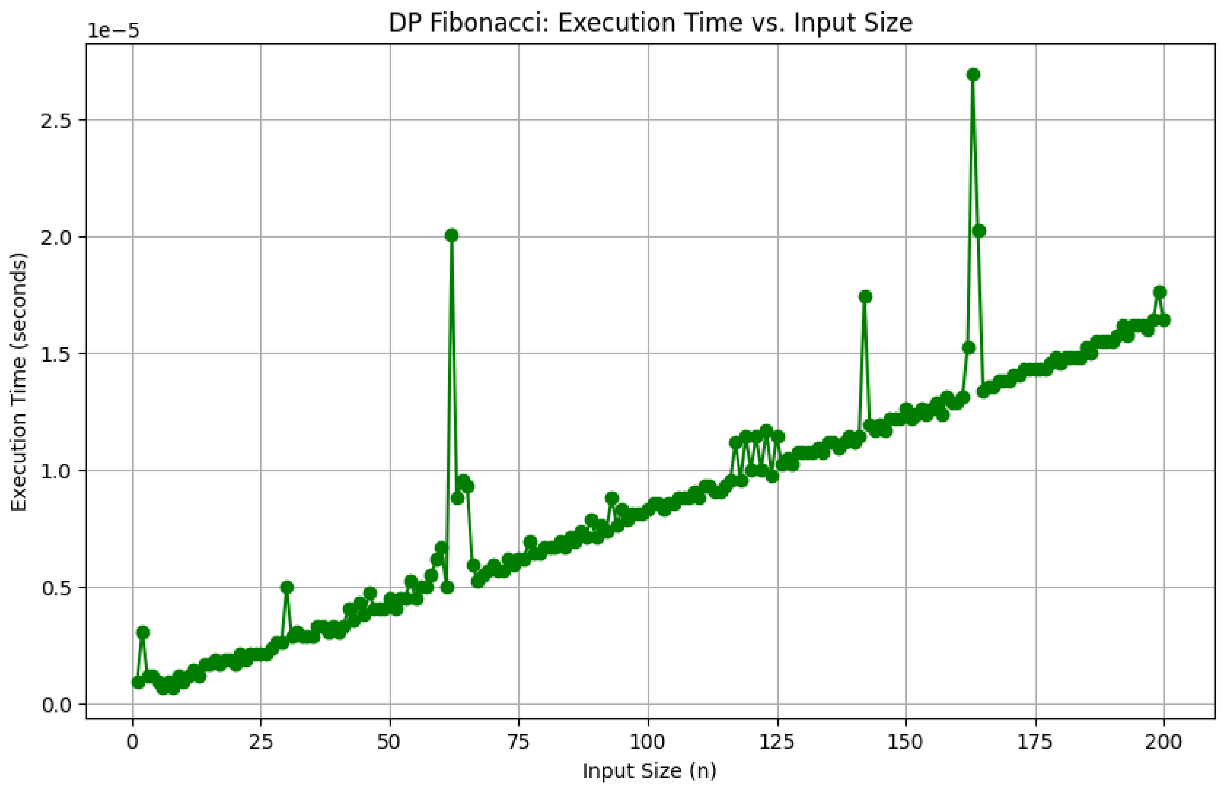
Code:

```
# Fibonacci (Dynamic Programming)

def f_dp(n):
    """Calculates the n-th Fibonacci number using dynamic programming
(tabulation)."""
    if n <= 1:
        return n
    fib_table = [0] * (n + 1)
    fib_table[1] = 1
    for i in range(2, n + 1):
        fib_table[i] = fib_table[i - 1] + fib_table[i - 2]
    return fib_table[n]

print("Algorithm Analysis: ")
print("Time Complexity: O(n) - Linear.")
print("Space Complexity: O(n) - Linear (can be optimized to O(1)).\n")

print("Demonstration:")
n_demo = 20
print(f"The {n_demo}th Fibonacci number is: {f_dp(n_demo)}")
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = range(1, 201)
execution_times = []
for n in input_sizes:
    start_time = time.time()
    f_dp(n)
    end_time = time.time()
    execution_times.append(end_time - start_time)
```

```
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-', color='g')
plt.title('DP Fibonacci: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

print("\nMemory Profile for n=200:")
%memit fibonacci_dp(200)
```

Output:

```
Algorithm Analysis:
Time Complexity: O(n) - Linear.
Space Complexity: O(n) - Linear (can be optimized to O(1)).

Demonstration:
The 20th Fibonacci number is: 6765
```



```
Memory Profile for n=200:

peak memory: 124.31 MiB, increment: 0.00 MiB
```

3. Merge Sort

- **Description**: A "divide and conquer" sorting algorithm. It works by recursively splitting the input list into two halves until single-element lists are created. It then repeatedly merges these sub-lists back together in sorted order.

- **Time Complexity**: O(nlogn) for the best, average, and worst cases. Its performance is highly consistent and does not depend on the initial order of elements.

- **Space Complexity**: O(n). It requires extra linear space to store the temporary sub-arrays during the merging process.

- **Suitability & Trade-offs**: Excellent for large datasets where guaranteed O(nlogn) performance is necessary. It is also a **stable sort**, meaning it preserves the original order of equal elements. Its main drawback is the O(n) space requirement.

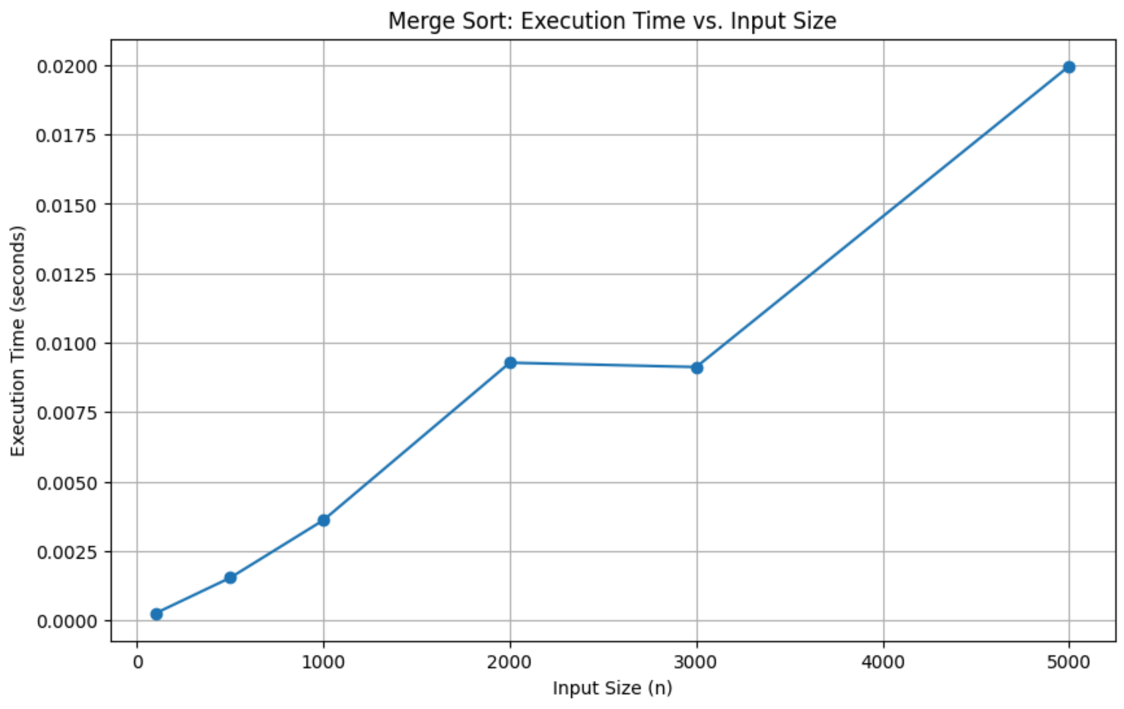
Code:

```
# Merge Sort

def m_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half, right_half = arr[:mid], arr[mid:]
        m_sort(left_half)
        m_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]: arr[k] = left_half[i]; i += 1
            else: arr[k] = right_half[j]; j += 1
            k += 1
        while i < len(left_half): arr[k] = left_half[i]; i += 1; k += 1
        while j < len(right_half): arr[k] = right_half[j]; j += 1; k += 1
    return arr

print("Algorithm Analysis: ")
print("Time Complexity: O(n log n) for all cases (Best, Average, Worst).")
print("Space Complexity: O(n) - Linear, for temporary arrays.\n")

print("Demonstration:")
demo_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", demo_list)
sorted_demo_list = m_sort(demo_list.copy())
print("Sorted list:  ", sorted_demo_list)
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = [100, 500, 1000, 2000, 3000, 5000]
```

```
execution_times = []
for size in input_sizes:
    test_list = [random.randint(0, size) for _ in range(size)]
    start_time = time.time()
    m_sort(test_list.copy())
    end_time = time.time()
    execution_times.append(end_time - start_time)

plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-')
plt.title('Merge Sort: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

sample_list = [random.randint(0, 5000) for _ in range(5000)]
print("\nMemory Profile for a list of size 5000:")
%memit m_sort(sample_list.copy())
```

Output:

```
Algorithm Analysis:
Time Complexity: O(n log n) for all cases (Best, Average, Worst).
Space Complexity: O(n) - Linear, for temporary arrays.

Demonstration:
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]
```



```
Memory Profile for a list of size 5000:
peak memory: 124.59 MiB, increment: 0.00 MiB
```

4. Quick Sort

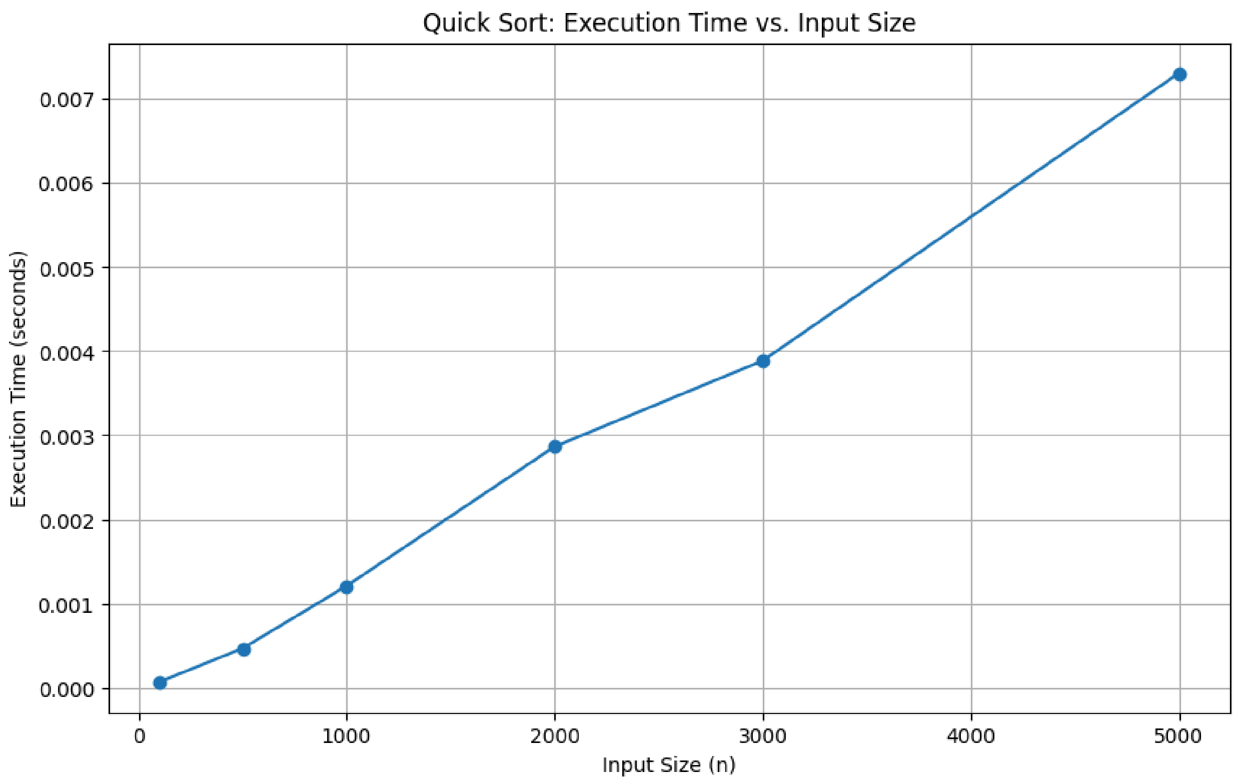
- **Description**: Another "divide and conquer" algorithm. It works by selecting a 'pivot' element and partitioning the rest of the list into two sub-lists: elements less than the pivot and elements greater than the pivot. It then recursively sorts the sub-lists.
- **Time Complexity**:
    - **Best/Average**: O(nlogn). This occurs when the pivot consistently divides the list into roughly equal halves.
    - **Worst**: O(n2). This rare case occurs with consistently poor pivot choices (e.g., always picking the smallest or largest element).
- **Space Complexity**:
    - **Average**: O(logn), determined by the depth of the recursion stack in a balanced partition.
    - **Worst**: O(n), if the recursion stack becomes as deep as the list itself in an unbalanced partition.
- **Suitability & Trade-offs**: It is often faster in practice than other O(nlogn) algorithms due to low constant factors and good cache performance. However, its O(n2) worst-case can be a concern, and it is an **unstable sort**.

Code:

```
# Quick Sort

def quick_sort_partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort_recursive(arr, low, high):
    if low < high:
        pi = quick_sort_partition(arr, low, high)
        quick_sort_recursive(arr, low, pi - 1)
        quick_sort_recursive(arr, pi + 1, high)

def quick_sort(arr):
    quick_sort_recursive(arr, 0, len(arr) - 1)
    return arr

print("Algorithm Analysis: ")
print("Time Complexity: Average O(n log n), Worst O(n^2).")
print("Space Complexity: Average O(log n), Worst O(n).\n")
print("Demonstration:")
demo_list = [10, 7, 8, 9, 1, 5]
print("Original list:", demo_list)
sorted_demo_list = demo_list.copy()
quick_sort(sorted_demo_list)
print("Sorted list:  ", sorted_demo_list)
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = [100, 500, 1000, 2000, 3000, 5000]
execution_times = []
for size in input_sizes:
    test_list = [random.randint(0, size) for _ in range(size)]
    start_time = time.time()
    quick_sort(test_list.copy())
    end_time = time.time()
    execution_times.append(end_time - start_time)

plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-')
plt.title('Quick Sort: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

sample_list = [random.randint(0, 5000) for _ in range(5000)]
print("\nMemory Profile for a list of size 5000:")
%memit quick_sort(sample_list.copy())
```

Output:

```
Algorithm Analysis:
Time Complexity: Average O(n log n), Worst O(n^2).
Space Complexity: Average O(log n), Worst O(n).

Demonstration:
Original list: [10, 7, 8, 9, 1, 5]
Sorted list:   [1, 5, 7, 8, 9, 10]
```



```
Memory Profile for a list of size 5000:
peak memory: 129.86 MiB, increment: 0.00 MiB
```

5. Insertion Sort

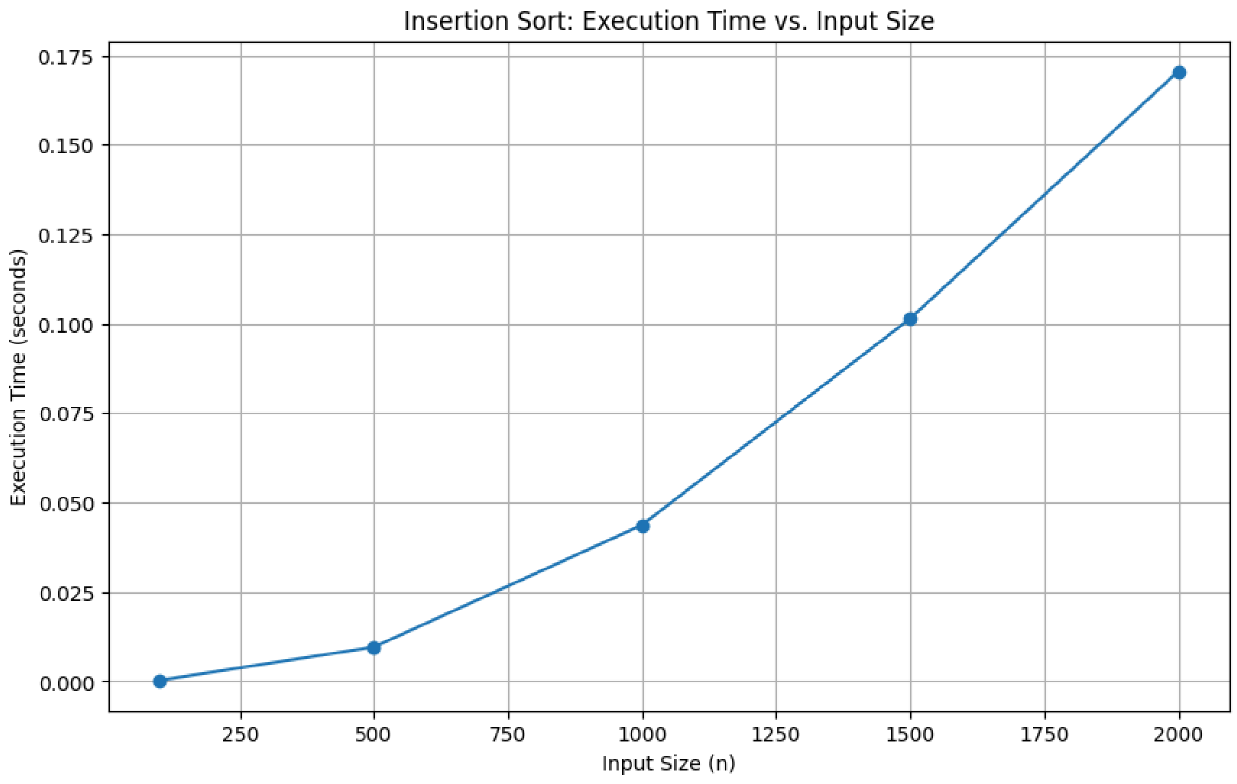
- **Description**: A simple algorithm that builds the final sorted list one item at a time. It iterates through the input list and "inserts" each element into its correct position within the already sorted part of the list.
- **Time Complexity**:
  - **Best**: O(n). This occurs when the list is already sorted.
  - **Average/Worst**: O(n2).
- **Space Complexity**: O(1). It is an **in-place** sorting algorithm that requires no significant extra storage.
- **Suitability & Trade-offs**: Very efficient for small datasets or lists that are already nearly sorted. Its performance degrades quickly for large, unordered lists.

Code:

```
# Insertion Sort

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

print("Algorithm Analysis: ")
print("Time Complexity: Average O(n^2), Best O(n).")
print("Space Complexity: O(1) - Constant (in-place).\n")

print("Demonstration:")
demo_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", demo_list)
insertion_sort(demo_list)
print("Sorted list:  ", demo_list)
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = [100, 500, 1000, 1500, 2000]
execution_times = []
for size in input_sizes:
    test_list = [random.randint(0, size) for _ in range(size)]
    start_time = time.time()
    insertion_sort(test_list.copy())
    end_time = time.time()
    execution_times.append(end_time - start_time)

plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-')
plt.title('Insertion Sort: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

sample_list = [random.randint(0, 2000) for _ in range(2000)]
print("\nMemory Profile for a list of size 2000:")
%memit insertion_sort(sample_list.copy())
```

Output:

```
Algorithm Analysis:
Time Complexity: Average O(n^2), Best O(n).
Space Complexity: O(1) - Constant (in-place).

Demonstration:
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]
```



```
Memory Profile for a list of size 2000:
peak memory: 129.86 MiB, increment: 0.00 MiB
```

6. Bubble Sort

- **Description**: A straightforward algorithm that repeatedly steps through the list, compares adjacent pairs of elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.
- **Time Complexity**:
    - **Best**: O(n). This occurs if the list is already sorted and an optimization is used to detect this.
    - **Average/Worst**: O(n2).
- **Space Complexity**: O(1). It is an **in-place** sort.
- **Suitability & Trade-offs**: Almost exclusively used for educational purposes to introduce sorting concepts. It is too slow for nearly all practical applications due to its poor performance.

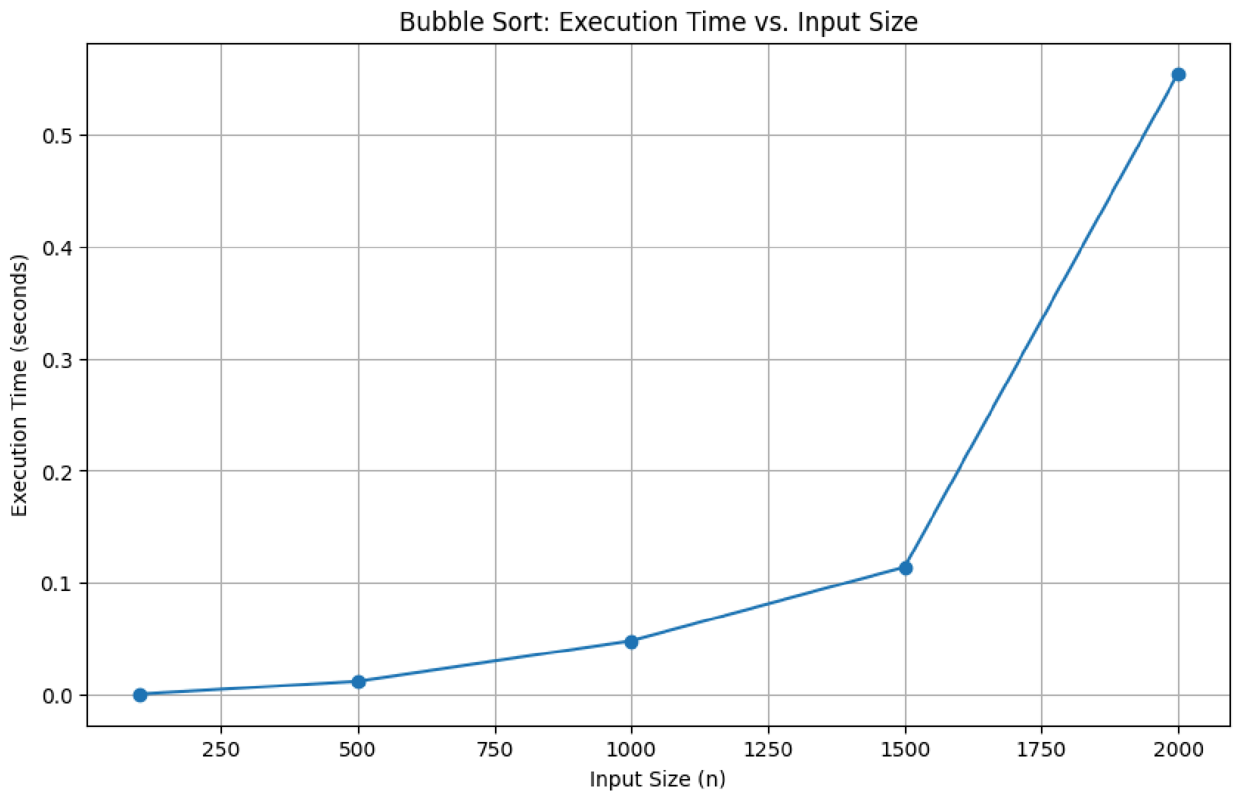
Code:

```
# Bubble Sort

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

print("Algorithm Analysis: ")
print("Time Complexity: Average O(n^2), Best O(n) (with optimization).")
print("Space Complexity: O(1) - Constant (in-place).\n")

print("Demonstration:")
demo_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", demo_list)
bubble_sort(demo_list)
print("Sorted list:  ", demo_list)
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = [100, 500, 1000, 1500, 2000]
execution_times = []
for size in input_sizes:
    test_list = [random.randint(0, size) for _ in range(size)]
    start_time = time.time()
    bubble_sort(test_list.copy())
    end_time = time.time()
    execution_times.append(end_time - start_time)
```

```
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-')
plt.title('Bubble Sort: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

sample_list = [random.randint(0, 2000) for _ in range(2000)]
print("\nMemory Profile for a list of size 2000:")
%memit bubble_sort(sample_list.copy())
```

Output:

```
Algorithm Analysis:
Time Complexity: Average O(n^2), Best O(n) (with optimization).
Space Complexity: O(1) - Constant (in-place).

Demonstration:
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]
```



```
Memory Profile for a list of size 2000:
peak memory: 129.88 MiB, increment: 0.00 MiB
```

7. Selection Sort

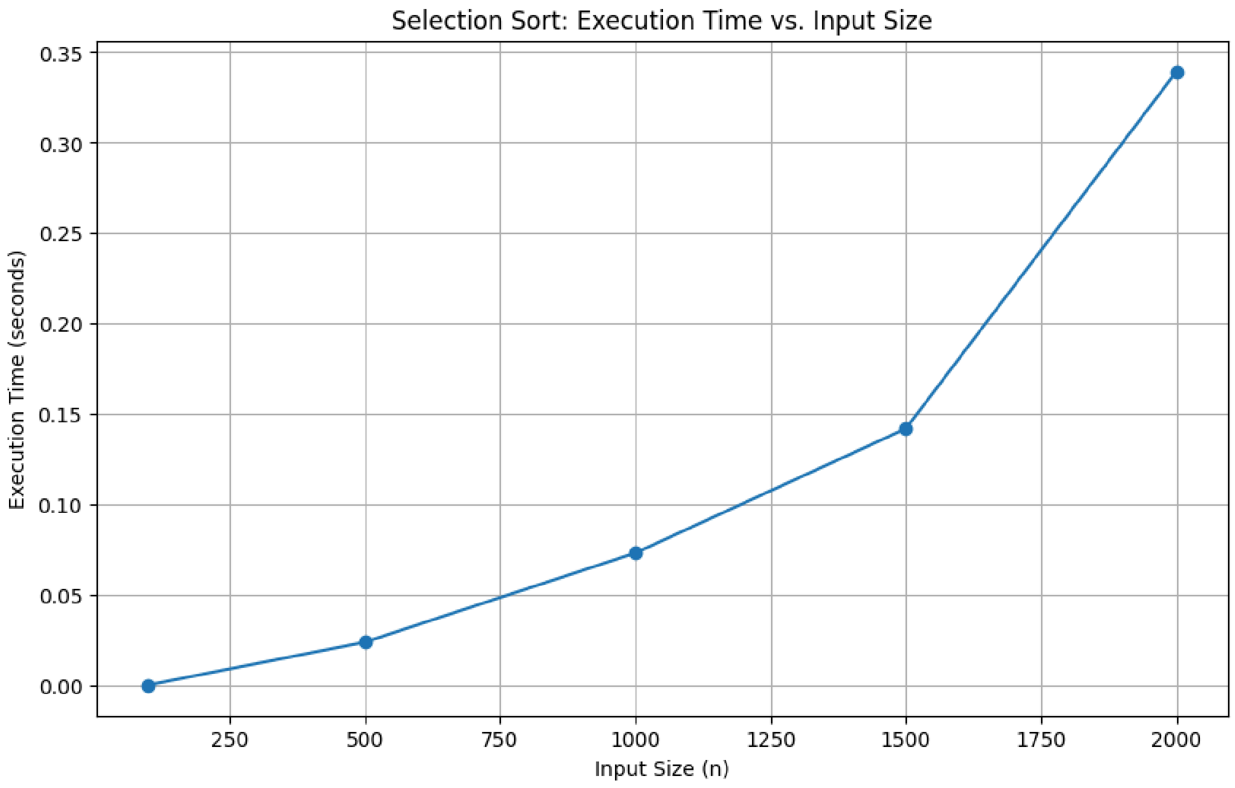
- **Description**: This algorithm sorts a list by repeatedly finding the minimum element from the unsorted portion and swapping it with the element at the beginning of the unsorted portion.
- **Time Complexity**: O(n2) for the best, average, and worst cases. The number of comparisons is fixed regardless of the input's initial order.
- **Space Complexity**: O(1). It is an **in-place** sort.
- **Suitability & Trade-offs**: Its primary advantage is that it makes the minimum possible number of swaps (n−1). This can be useful in specific scenarios where the cost of swapping is extremely high. Otherwise, it is inefficient for large lists.

Code:

```
# Selection Sort

def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

print("Algorithm Analysis: ")
print("Time Complexity: O(n^2) for all cases (Best, Average, Worst).")
print("Space Complexity: O(1) - Constant (in-place).\n")

print("Demonstration:")
demo_list = [64, 25, 12, 22, 11]
print("Original list:", demo_list)
selection_sort(demo_list)
print("Sorted list:  ", demo_list)
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = [100, 500, 1000, 1500, 2000]
execution_times = []
for size in input_sizes:
    test_list = [random.randint(0, size) for _ in range(size)]
    start_time = time.time()
    selection_sort(test_list.copy())
    end_time = time.time()
    execution_times.append(end_time - start_time)

plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-')
plt.title('Selection Sort: Execution Time vs. Input Size')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

sample_list = [random.randint(0, 2000) for _ in range(2000)]
print("\nMemory Profile for a list of size 2000:")
%memit selection_sort(sample_list.copy())
```

Output:

```
Algorithm Analysis:
Time Complexity: O(n^2) for all cases (Best, Average, Worst).
Space Complexity: O(1) - Constant (in-place).

Demonstration:
Original list: [64, 25, 12, 22, 11]
Sorted list:   [11, 12, 22, 25, 64]
```



```
Memory Profile for a list of size 2000:
peak memory: 129.88 MiB, increment: 0.00 MiB
```

8. Binary Search

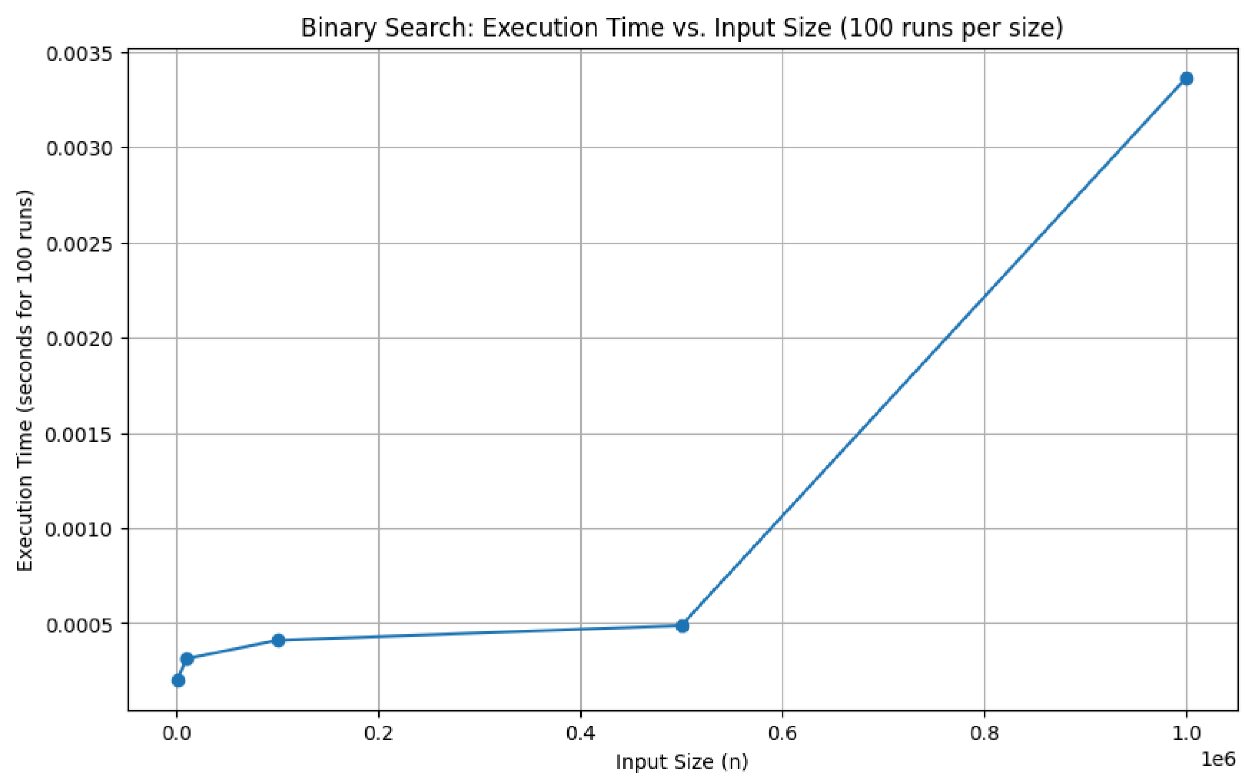
- **Description**: An efficient algorithm for finding a target value within a **sorted list**. It works by repeatedly dividing the search interval in half. If the middle element is the target, the search is over. Otherwise, the half in which the target cannot lie is eliminated, and the search continues on the remaining half.
- **Time Complexity**:
    - **Best**: O(1). The target is found on the first check (the middle element).
    - **Average/Worst**: O(logn).
- **Space Complexity**: O(1) for the iterative version. A recursive version would have O(logn) space complexity due to the call stack.
- **Suitability & Trade-offs**: It is the standard method for searching in large, sorted arrays due to its extreme speed. Its major trade-off is the absolute requirement that the list must be sorted before the search begins.

Code:

```
# Binary Search

def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target: return mid
        elif arr[mid] < target: low = mid + 1
        else: high = mid - 1
    return -1 # Not found

print("Algorithm Analysis: ")
print("Time Complexity: Average O(log n), Best O(1).")
print("Space Complexity: O(1) for this iterative version.\n")

print("Demonstration:")
demo_list = [2, 5, 7, 8, 11, 12, 25, 30]
target1 = 11
target2 = 6
print(f"Searching in list: {demo_list}")
print(f"Index of {target1}: {binary_search(demo_list, target1)}")
print(f"Index of {target2}: {binary_search(demo_list, target2)}")
print("-" * 30)

# Experimental Profiling & Visualization
input_sizes = [1000, 10000, 100000, 500000, 1000000]
execution_times = []
for size in input_sizes:
    sorted_list = list(range(size))
    target = -1 # Worst case
    start_time = time.time()
    for _ in range(100): # Run multiple times to get a measurable time
        binary_search(sorted_list, target)
    end_time = time.time()
    execution_times.append(end_time - start_time)

plt.figure(figsize=(10, 6))
plt.plot(input_sizes, execution_times, marker='o', linestyle='-')
plt.title('Binary Search: Execution Time vs. Input Size (100 runs per size)')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds for 100 runs)')
plt.grid(True)
plt.show()

sample_list = list(range(1000000))
print("\nMemory Profile for a list of size 1,000,000:")
%memit binary_search(sample_list, -1)
```

Output:

```
Algorithm Analysis:
Time Complexity: Average O(log n), Best O(1).
Space Complexity: O(1) for this iterative version.

Demonstration:
Searching in list: [2, 5, 7, 8, 11, 12, 25, 30]
Index of 11: 4
Index of 6: -1
```



```
Memory Profile for a list of size 1,000,000:
peak memory: 208.44 MiB, increment: 0.00 MiB
```

9. Comparative Analysis Summary

- **Objective**: To visually demonstrate the performance differences between sorting algorithms.
- **Conclusion**: A graphical comparison reveals two distinct performance groups:
    - **O(n2) Algorithms (Bubble, Insertion, Selection)**: These are significantly slower. Their execution time grows quadratically, making them unsuitable for large datasets.
    - **O(nlogn) Algorithms (Merge, Quick)**: These are vastly more efficient and scale well with increasing input size. Their log-linear growth is far superior to quadratic growth. Quick Sort is often observed to be slightly faster in practice than Merge Sort for average cases.

Code:

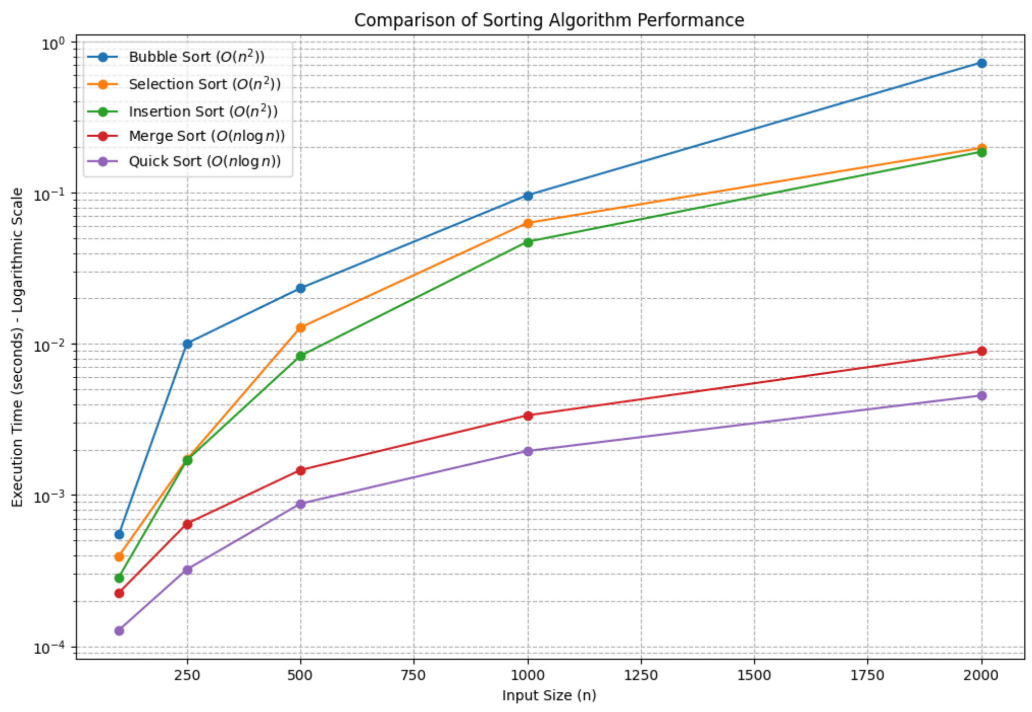
```
#Comparative Analysis of All Sorting Algorithms

import time
import random
import matplotlib.pyplot as plt
import sys

sys.setrecursionlimit(5000)

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

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half, right_half = arr[:mid], arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]: arr[k] = left_half[i]; i += 1
            else: arr[k] = right_half[j]; j += 1
            k += 1
        while i < len(left_half): arr[k] = left_half[i]; i += 1; k += 1
        while j < len(right_half): arr[k] = right_half[j]; j += 1; k += 1
    return arr

def quick_sort(arr):
    def _quick_sort_recursive(arr, low, high):
        if low < high:
            pi = _partition(arr, low, high)
            _quick_sort_recursive(arr, low, pi - 1)
            _quick_sort_recursive(arr, pi + 1, high)
    def _partition(arr, low, high):
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1
    _quick_sort_recursive(arr, 0, len(arr) - 1)
    return arr


algorithms = {
    "Bubble Sort ($O(n^2)$)": bubble_sort,
    "Selection Sort ($O(n^2)$)": selection_sort,
    "Insertion Sort ($O(n^2)$)": insertion_sort,
    "Merge Sort ($O(n \log n)$)": merge_sort,
    "Quick Sort ($O(n \log n)$)": quick_sort
}

input_sizes = [100, 250, 500, 1000, 2000]
results = {name: [] for name in algorithms}

print("Running comparative analysis... (This may take a moment)")
for size in input_sizes:
    print(f"  Testing with list size: {size}")
    # Use the same random list for all algorithms at a given size for a fair
comparison
    test_list = [random.randint(0, size) for _ in range(size)]
    for name, algo_func in algorithms.items():
        list_copy = test_list.copy()
        start_time = time.time()
        algo_func(list_copy)
        end_time = time.time()
        results[name].append(end_time - start_time)
```

```
print("Analysis complete.")


plt.figure(figsize=(12, 8))
for name, times in results.items():
    plt.plot(input_sizes, times, marker='o', linestyle='-', label=name)

plt.title('Comparison of Sorting Algorithm Performance')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds) - Logarithmic Scale')
plt.grid(True, which="both", ls="--")
plt.legend()
plt.yscale('log') # ogarithmic scale to see all results clearly
plt.show()
```

Output:

```
Running comparative analysis... (This may take a moment)
  Testing with list size: 100
  Testing with list size: 250
  Testing with list size: 500
  Testing with list size: 1000
<>:75: SyntaxWarning: invalid escape sequence '\l'
<>:76: SyntaxWarning: invalid escape sequence '\l'
<>:75: SyntaxWarning: invalid escape sequence '\l'
<>:76: SyntaxWarning: invalid escape sequence '\l'
/tmp/ipython-input-2168183009.py:75: SyntaxWarning: invalid escape sequence
'\l'
  "Merge Sort ($O(n \log n)$)": merge_sort,
/tmp/ipython-input-2168183009.py:76: SyntaxWarning: invalid escape sequence
'\l'
  "Quick Sort ($O(n \log n)$)": quick_sort
  Testing with list size: 2000
Analysis complete.
```


Comparison of Sorting Algorithm Performance
Bubble Sort ($O(n^2)$)
Selection Sort ($O(n^2)$)
Insertion Sort ($O(n^2)$)
Merge Sort ($O(n \log n)$)
Quick Sort ($O(n \log n)$)
Execution Time (seconds) - Logarithmic Scale
Input Size (n)
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
250
500
750
1000
1250
1500
1750
2000