SOURCE CODE:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from memory_profiler import memory_usage
import random

# Helper function for profiling
def profile_func(func, *args):
    start = time.time()
    mem_usage = memory_usage((func, args), max_iterations=1)
    end = time.time()
    return end - start, max(mem_usage) - min(mem_usage)

# Fibonacci Algorithms
def fib_recursive(n):
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)

def fib_dp(n):
    if n <= 1:
        return n
    dp = [0]*(n+1)
    dp[0], dp[1] = 0, 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Sorting Algorithms

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
```

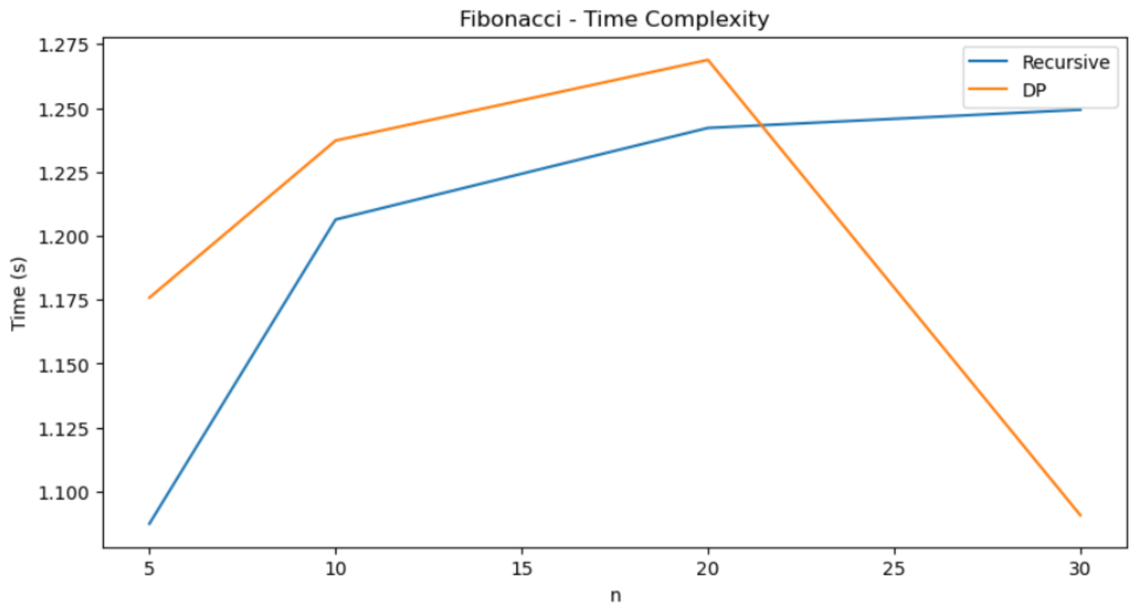
```python
        if left[i] <= right[j]:
            result.append(left[i]); i += 1
        else:
            result.append(right[j]); j += 1
    result.extend(left[i:]); result.extend(right[j:])
    return result

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[random.randint(0, len(arr)-1)]
    less = [x for x in arr if x < pivot]
    equal = [x for x in arr if x == pivot]
    greater = [x for x in arr if x > pivot]
    return quick_sort(less) + equal + quick_sort(greater)

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 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 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
```

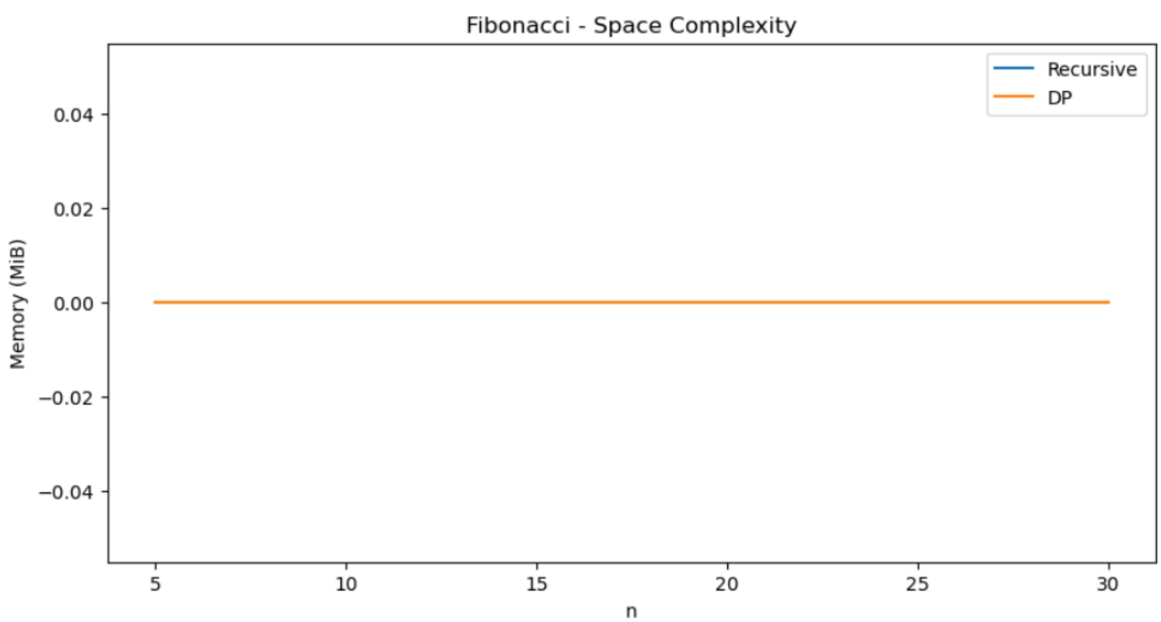
```python
# Searching Algorithm
def binary_search(arr, target):
    low, high = 0, len(arr)-1
    while low <= high:
        mid = (low+high)//2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid+1
        else:
            high = mid-1
    return -1

# Profiling Sorting Algorithms
algorithms = {
    "Merge Sort": merge_sort,
    "Quick Sort": quick_sort,
    "Insertion Sort": insertion_sort,
    "Bubble Sort": bubble_sort,
    "Selection Sort": selection_sort,
}

sizes = [100, 500, 1000, 2000]
results_time = {name: [] for name in algorithms}
results_mem = {name: [] for name in algorithms}

for size in sizes:
    arr = list(np.random.randint(0, 10000, size))
    for name, func in algorithms.items():
        t, m = profile_func(func, arr.copy())
        results_time[name].append(t)
        results_mem[name].append(m)

# Individual graphs for each sorting algorithm
for name in algorithms:
    plt.figure(figsize=(10,5))
    plt.plot(sizes, results_time[name], marker='o')
    plt.xlabel("Input Size"); plt.ylabel("Time (s)")
    plt.title(f"{name} - Time Complexity")
    plt.show()

    plt.figure(figsize=(10,5))
    plt.plot(sizes, results_mem[name], marker='o')
    plt.xlabel("Input Size"); plt.ylabel("Memory (MiB)")
```

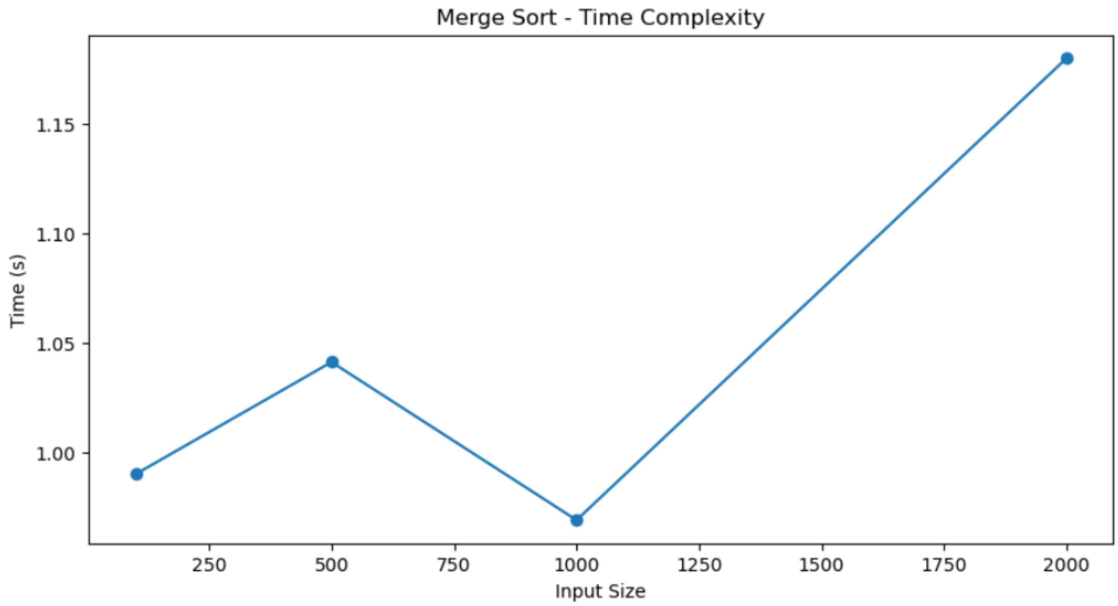
```python
    plt.title(f"{name} - Space Complexity")
    plt.show()

# Profiling Fibonacci
fib_sizes = [5, 10, 20, 30]   # small sizes for recursive
rec_times, rec_mems = [], []
dp_times, dp_mems = [], []

for n in fib_sizes:
    t, m = profile_func(fib_recursive, n)
    rec_times.append(t); rec_mems.append(m)
    t, m = profile_func(fib_dp, n)
    dp_times.append(t); dp_mems.append(m)

# Individual Fibonacci graphs
plt.figure(figsize=(10,5))
plt.plot(fib_sizes, rec_times, label="Recursive")
plt.plot(fib_sizes, dp_times, label="DP")
plt.xlabel("n"); plt.ylabel("Time (s)")
plt.title("Fibonacci - Time Complexity")
plt.legend(); plt.show()

plt.figure(figsize=(10,5))
plt.plot(fib_sizes, rec_mems, label="Recursive")
plt.plot(fib_sizes, dp_mems, label="DP")
plt.xlabel("n"); plt.ylabel("Memory (MiB)")
plt.title("Fibonacci - Space Complexity")
plt.legend(); plt.show()

# Combined Graphs (Sorting + Fibonacci)

combined_time = {**results_time,
                 "Fibonacci Recursive": rec_times,
                 "Fibonacci DP": dp_times}

combined_mem = {**results_mem,
                "Fibonacci Recursive": rec_mems,
                "Fibonacci DP": dp_mems}

plt.figure(figsize=(10,5))
for name, times in combined_time.items():
    if "Fibonacci" in name:
        plt.plot(fib_sizes, times, marker='o', label=name)
```

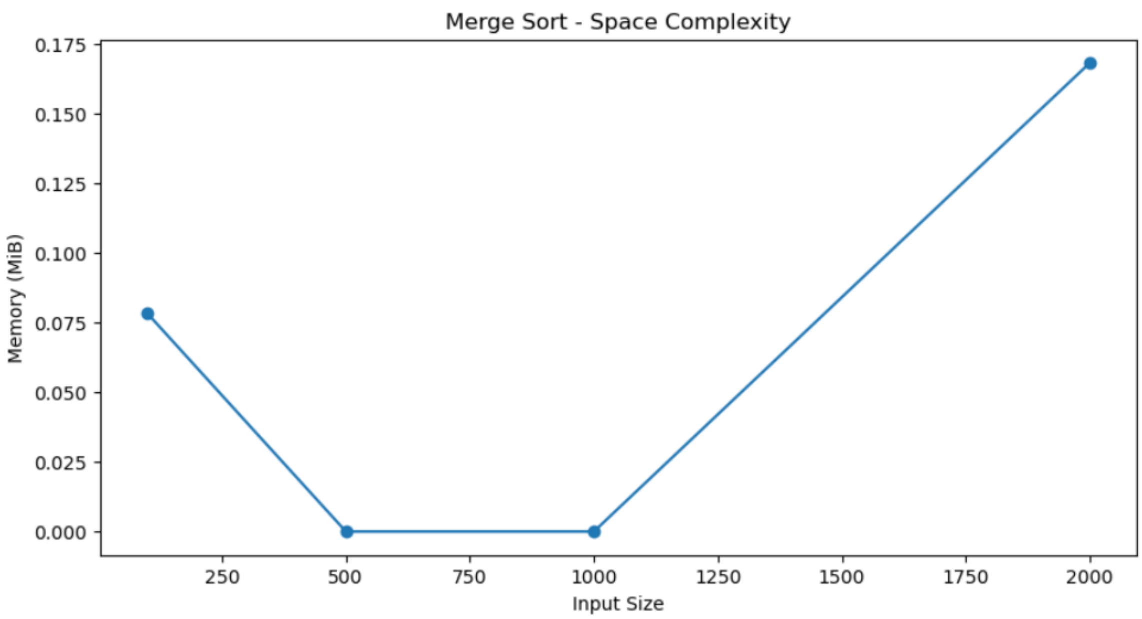
```
    else:
        plt.plot(sizes, times, marker='o', label=name)
plt.xlabel("Input Size (n)")
plt.ylabel("Time (s)")
plt.title("Combined - Time Complexity (Sorting + Fibonacci)")
plt.legend()
plt.show()

plt.figure(figsize=(10,5))
for name, mems in combined_mem.items():
    if "Fibonacci" in name:
        plt.plot(fib_sizes, mems, marker='o', label=name)
    else:
        plt.plot(sizes, mems, marker='o', label=name)
plt.xlabel("Input Size (n)")
plt.ylabel("Memory (MiB)")
plt.title("Combined - Space Complexity (Sorting + Fibonacci)")
plt.legend()
plt.show()
```

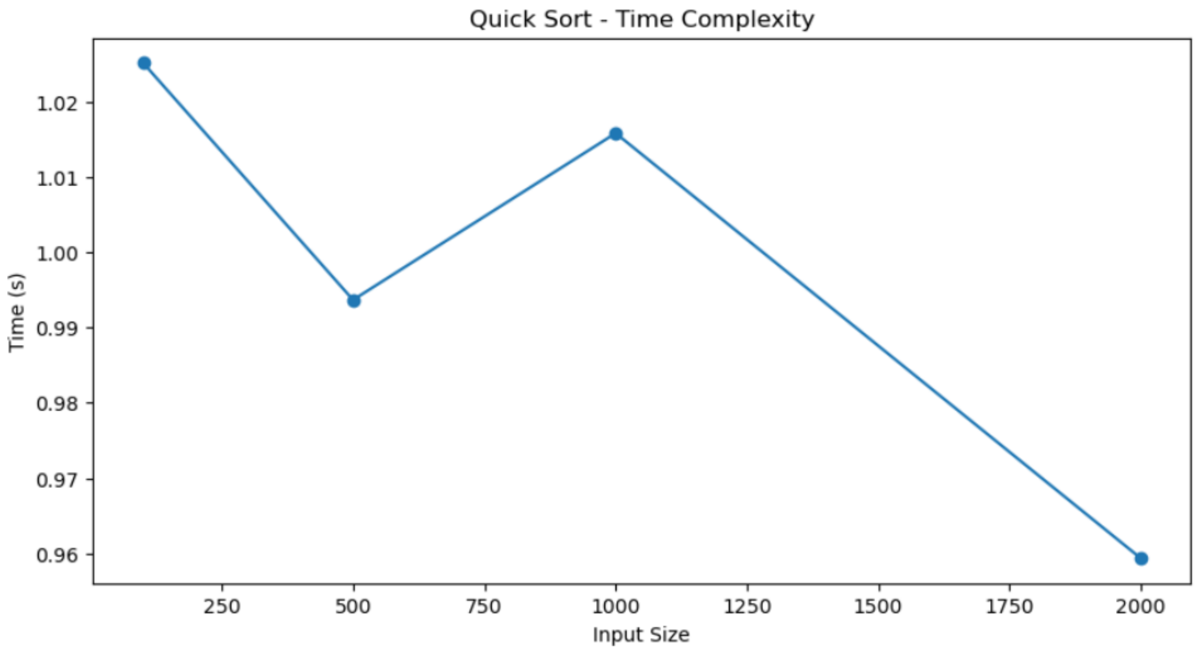
OUTPUT:

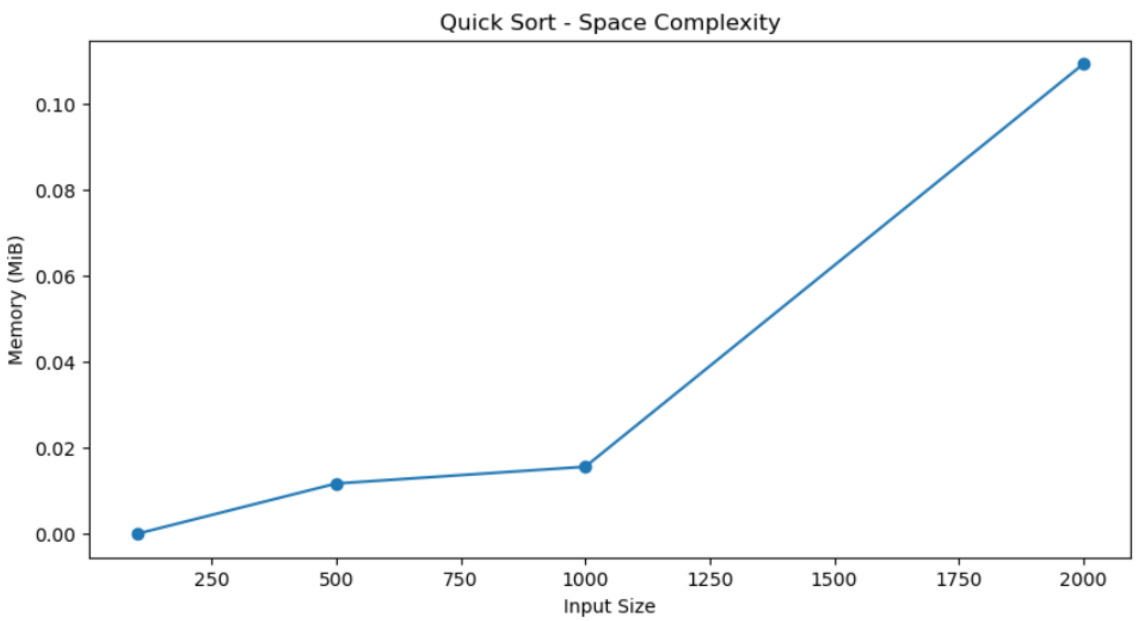

Fibonacci - Time Complexity
Recursive
DP
1.275
1.250
1.225
1.200
1.175
1.150
1.125
1.100
Time (s)
5
10
15
20
25
30
n

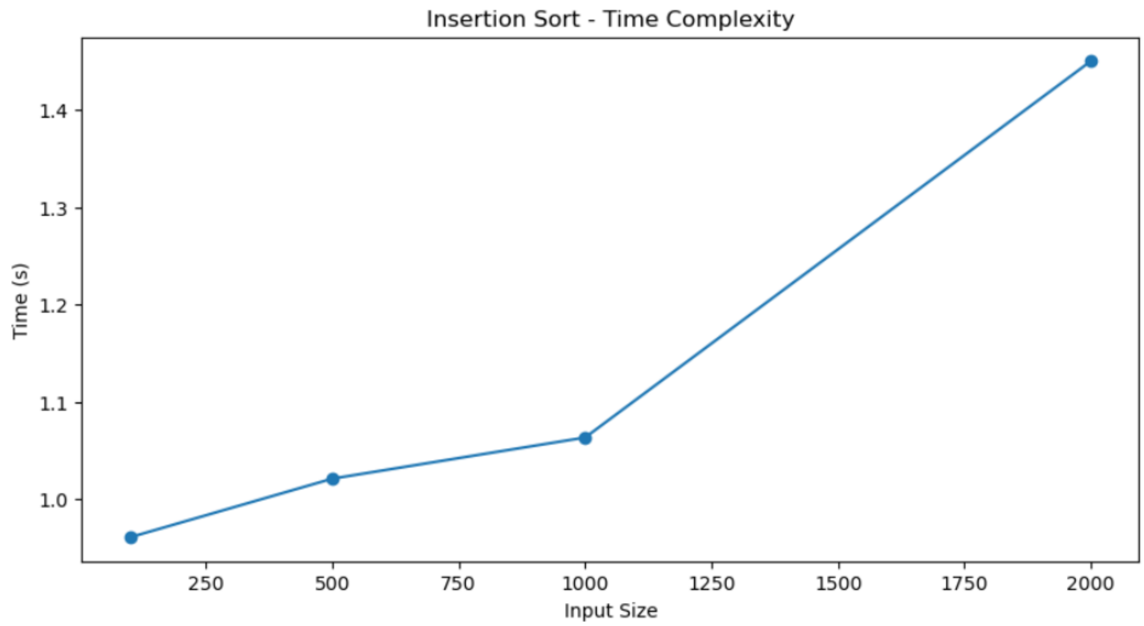

Fibonacci - Space Complexity
Recursive
DP
0.04
0.02
0.00
−0.02
−0.04
Memory (MiB)
5
10
15
20
25
30
n

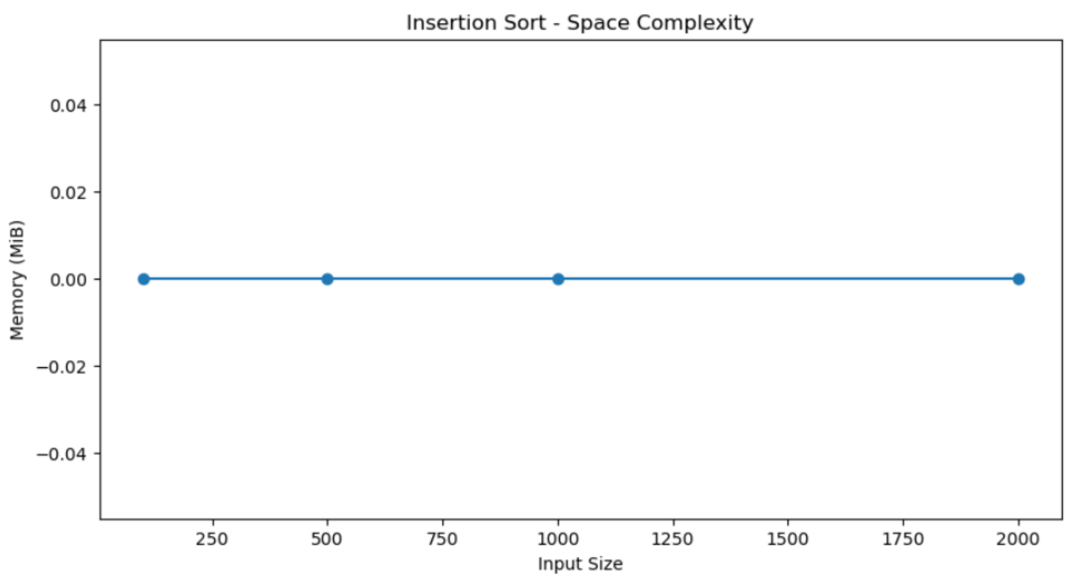

Merge Sort - Time Complexity
Time (s)
Input Size
1.15
1.10
1.05
1.00
250
500
750
1000
1250
1500
1750
2000

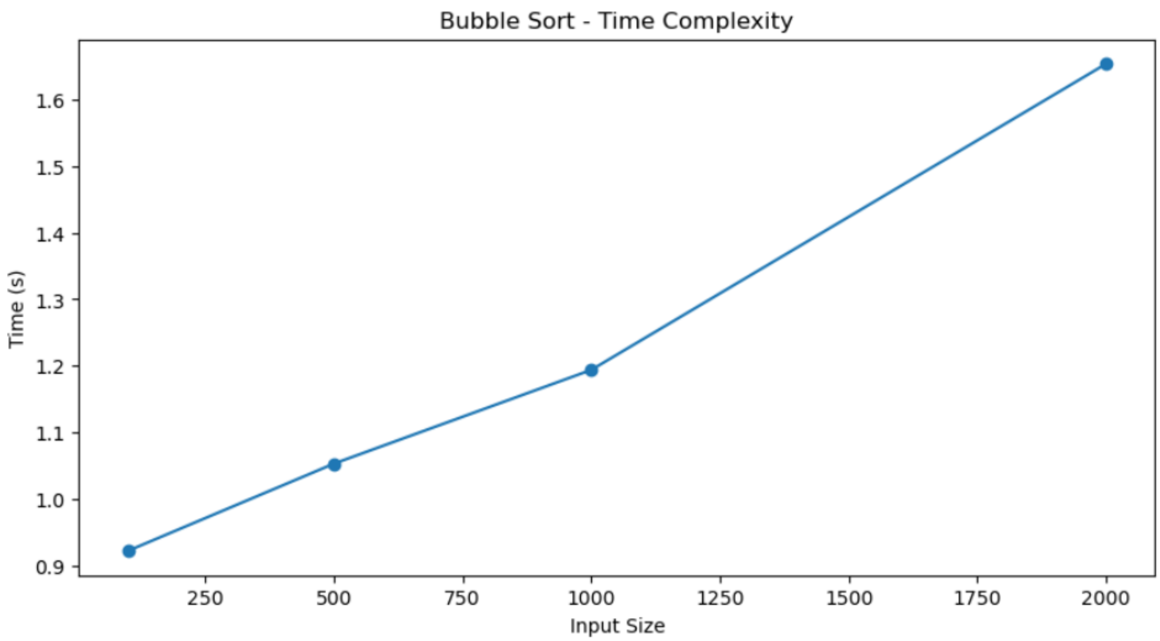

Merge Sort - Space Complexity
Memory (MiB)
Input Size
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
250
500
750
1000
1250
1500
1750
2000

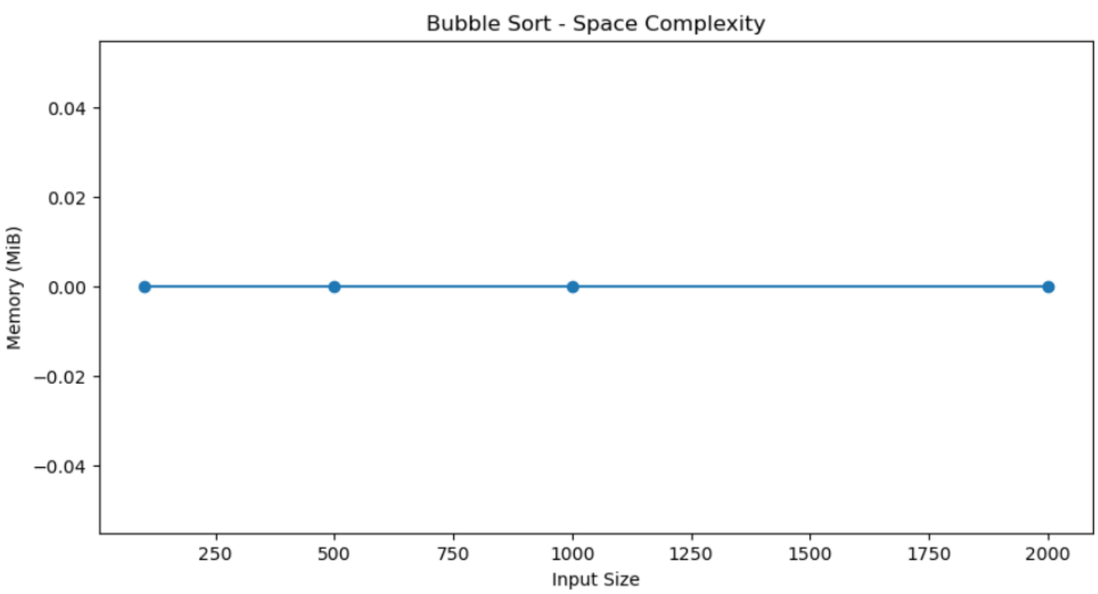

Quick Sort - Time Complexity
Time (s)
Input Size
1.02
1.01
1.00
0.99
0.98
0.97
0.96
250
500
750
1000
1250
1500
1750
2000

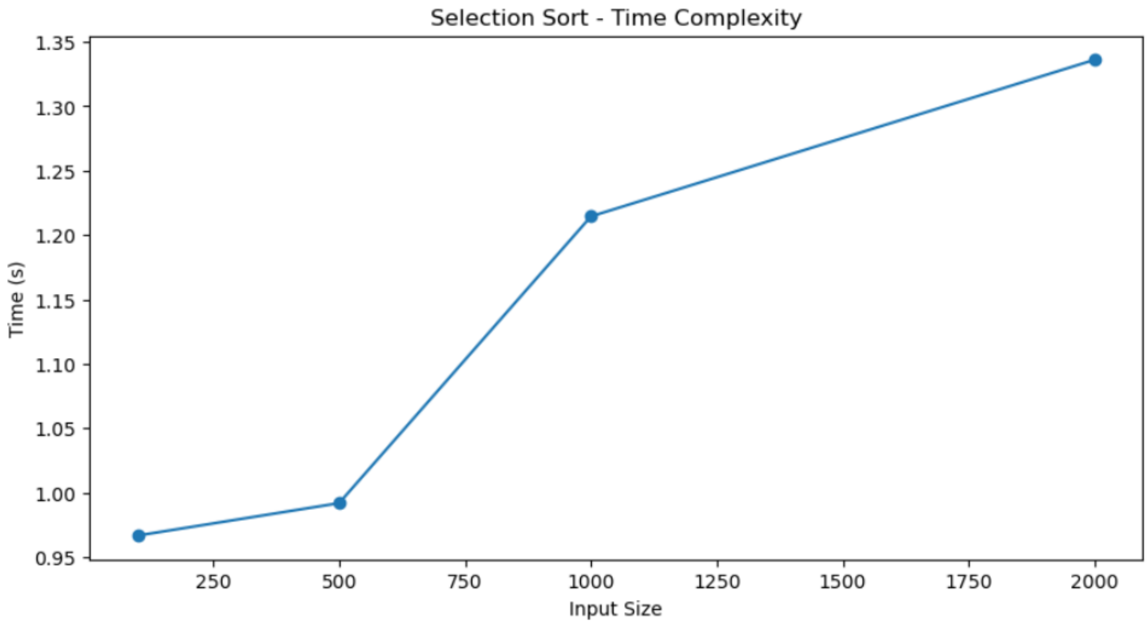

Quick Sort - Space Complexity
Memory (MiB)
Input Size
0.10
0.08
0.06
0.04
0.02
0.00
250
500
750
1000
1250
1500
1750
2000

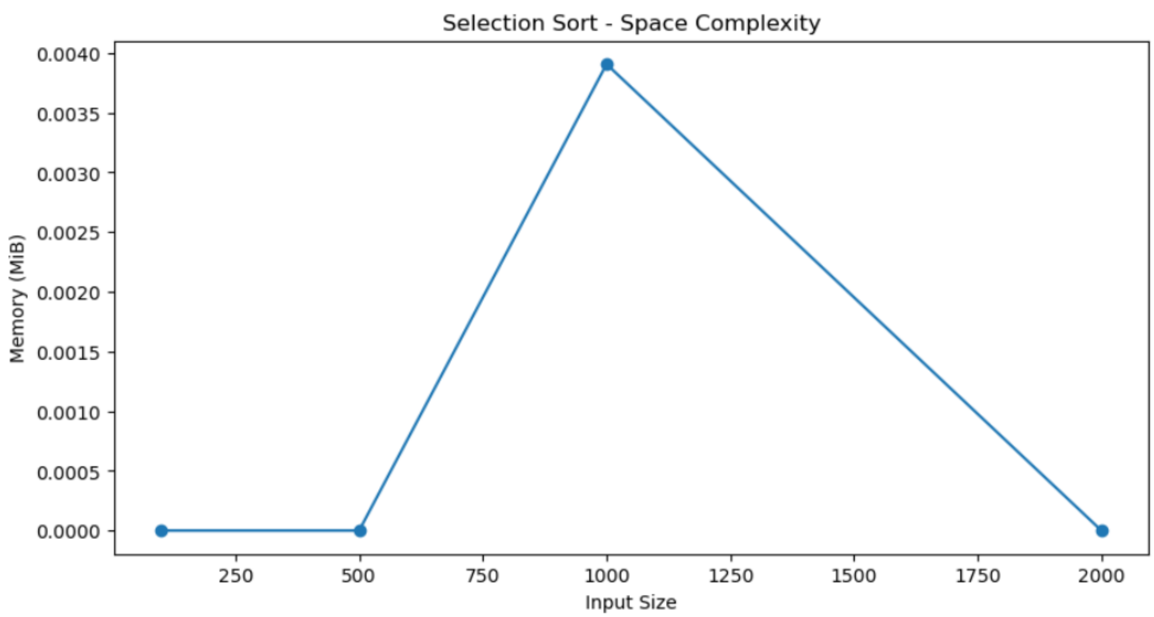

Insertion Sort - Time Complexity
1.4
1.3
1.2
1.1
1.0
Time (s)
250
500
750
1000
1250
1500
1750
2000
Input Size

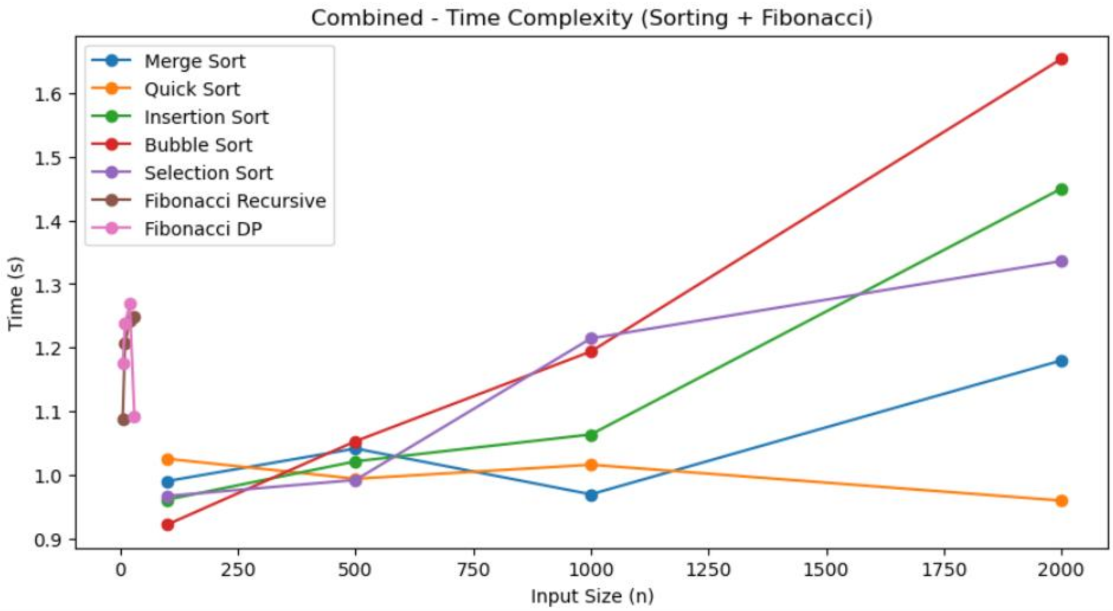

Insertion Sort - Space Complexity
0.04
0.02
0.00
−0.02
−0.04
Memory (MiB)
250
500
750
1000
1250
1500
1750
2000
Input Size

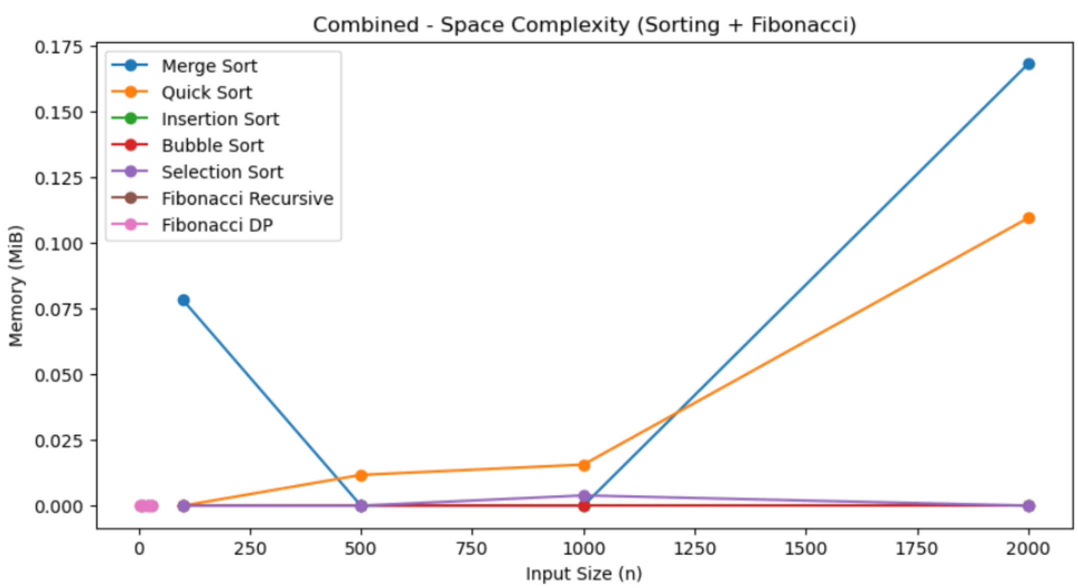

Bubble Sort - Time Complexity
Time (s)
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
250
500
750
1000
1250
1500
1750
2000
Input Size


Bubble Sort - Space Complexity
Memory (MiB)
0.04
0.02
0.00
−0.02
−0.04
250
500
750
1000
1250
1500
1750
2000
Input Size


Selection Sort - Time Complexity
Time (s)
Input Size
1.35
1.30
1.25
1.20
1.15
1.10
1.05
1.00
0.95
250
500
750
1000
1250
1500
1750
2000


Selection Sort - Space Complexity
Memory (MiB)
Input Size
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
250
500
750
1000
1250
1500
1750
2000


Combined - Time Complexity (Sorting + Fibonacci)
Merge Sort
Quick Sort
Insertion Sort
Bubble Sort
Selection Sort
Fibonacci Recursive
Fibonacci DP
Time (s)
Input Size (n)


Combined - Space Complexity (Sorting + Fibonacci)
Merge Sort
Quick Sort
Insertion Sort
Bubble Sort
Selection Sort
Fibonacci Recursive
Fibonacci DP
Memory (MiB)
Input Size (n)

1. Fibonacci (Recursive)

Input: A number n

Output: nth Fibonacci number

Time Complexity: Exponential O(2^n)

Space Complexity: O(n) (recursion stack)

Suitable only for very small inputs. Risk of stack overflow if n is large.

2. Fibonacci (Dynamic Programming)

Input: A number n

Output: nth Fibonacci number

Time Complexity: O(n)

Space Complexity: O(n) (can be optimized to O(1))

Much faster and scalable than recursive.

3. Merge Sort

Time Complexity: O(n log n) (best, average, worst)

Space Complexity: O(n)

Stable and reliable but uses more memory.

4. Quick Sort

Time Complexity: Best & Average $O(n \log n)$, Worst $O(n^2)$

Space Complexity: $O(\log n)$

Very fast in practice but recursive, so risk of stack overflow.

5. Insertion Sort

Time Complexity: Best $O(n)$, Worst $O(n^2)$

Space Complexity: $O(1)$

Works well for small or nearly sorted data.

6. Bubble Sort

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Simple but very slow.

7. Selection Sort

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Simple, not practical for large datasets.

8. Binary Search

Time Complexity: O(log n)

Space Complexity: O(1)

Very efficient but requires sorted data.

Recursive Depth & Stack Overflow:

Recursive algorithms (like Fibonacci recursive and Quick Sort) can cause stack overflow if recursion goes too deep. Python has recursion depth limit (~1000). Iterative or DP methods are safer for large inputs.

1. Plot execution time vs. input size using matplotlib

Execution time increases with input size for all algorithms.

Efficient algorithms (Merge Sort, Quick Sort, Fibonacci DP) scale much better compared to inefficient ones (Bubble Sort, Selection Sort, Recursive Fibonacci).

Graphs clearly show the difference between O(n log n) vs. $O(n^2)$ vs. Exponential $O(2^n)$.

2. Discuss trade-offs in time vs. space complexity

Some algorithms like Merge Sort use extra memory (O(n)) for merging, but they guarantee O(n log n) time.

Quick Sort is faster in practice and uses less memory, but worst-case time is $O(n^2)$ if pivot is bad.

Bubble Sort, Insertion Sort, Selection Sort use very little extra space (O(1)), but are slow ($O(n^2)$).

Fibonacci recursive uses exponential time and large recursion depth, but very little code.

Fibonacci DP uses more memory (array/table), but reduces time to O(n).

Trade-off = Faster algorithms often use more memory, while memory-efficient algorithms may take more time.

3. Comment on recursive depth and stack overflow risks

Recursive Fibonacci grows recursion depth linearly with n, leading to very deep recursion for large inputs. This can cause stack overflow in Python if n is too large (default recursion limit ≈ 1000).

Merge Sort and Quick Sort also use recursion, but depth is O(log n), which is safe even for large inputs.

Dynamic Programming Fibonacci avoids recursion, so no stack overflow risk.

Recursive depth is a serious concern mainly in naïve recursive algorithms (like Fibonacci).