SCHOOL OF ENGINERING AND TECHNOLOGY

Course Code – ENCA301

DESIGN AND ANALYSIS OF ALGORITHMS LAB

(2025-26)

Lab Manual

Submitted by – Ankita Bisht

Roll No. – 2301201173

Course – BCA (AI & DS)

Semester – 5th

Submitted to – Ms Aarti Sangwan

Lab Assignment – 1

Assignment Title: Analyzing and Visualizing Recursive Algorithm Efficiency

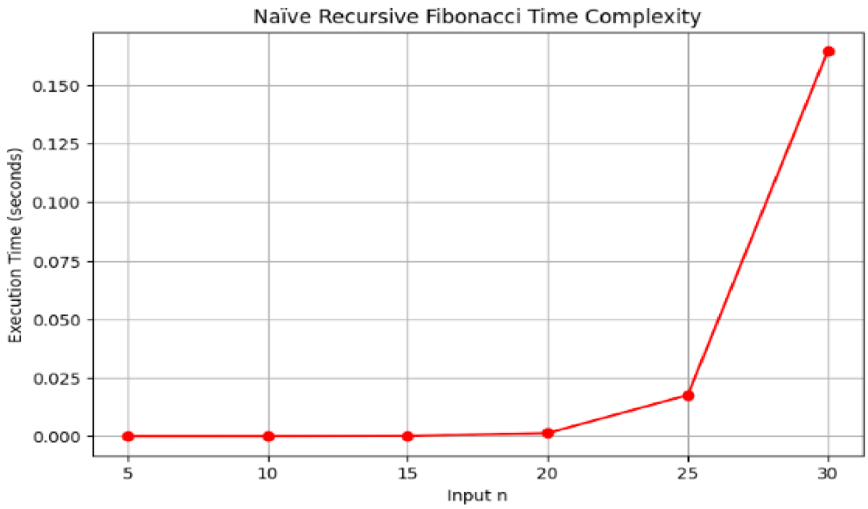
1. Fibonacci Sequence:-

- **Naïve Recursive**
  - **Input:** Integer n ≥ 0
  - **Output:** nth Fibonacci number
  - **Time Complexity:**
    - Best: O(1) (n=0 or 1)
    - Average: O(2^n)
    - Worst: O(2^n)
  - **Space Usage:** O(n) (due to recursion depth)
  - **Trade-offs:** Simple implementation, but highly inefficient due to overlapping subproblems. Risk of stack overflow for large n.

```
!pip install memory_profiler

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

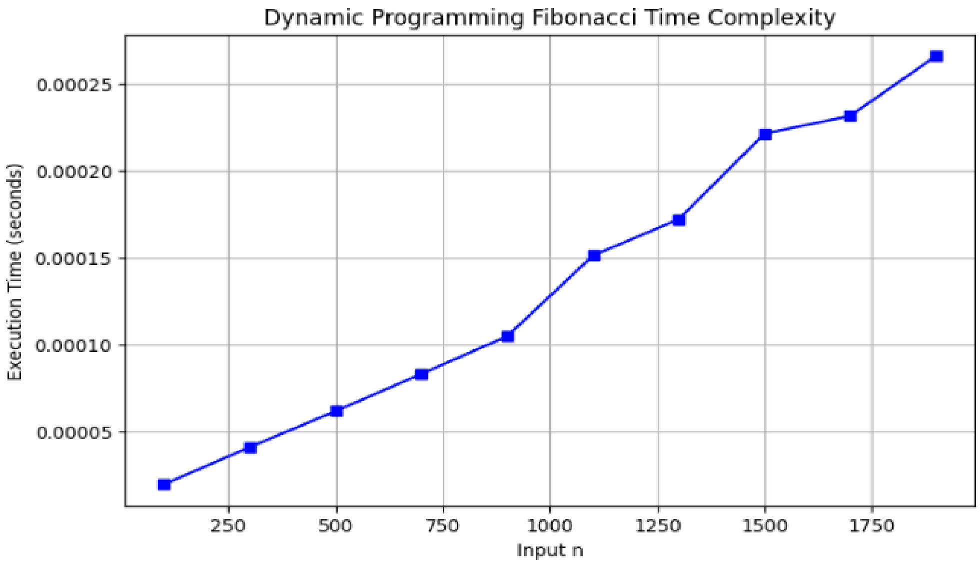
```
Collecting memory_profiler
  Downloading memory_profiler-0.61.0-py3-none-any.whl.metadata (20 kB)
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (from memory_profiler) (5.9.5)
Downloading memory_profiler-0.61.0-py3-none-any.whl (31 kB)
Installing collected packages: memory_profiler
Successfully installed memory_profiler-0.61.0
```

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

# Measure execution time
def measure_time(func, n):
    start = time.time()
    func(n)
    end = time.time()
    return end - start

# Values for recursive
ns_recursive = [5, 10, 15, 20, 25, 30]
times_recursive = [measure_time(fib_recursive, n) for n in ns_recursive]

# Plot
plt.figure(figsize=(8,5))
plt.plot(ns_recursive, times_recursive, marker='o', color='red')
plt.xlabel("Input n")
plt.ylabel("Execution Time (seconds)")
plt.title("Naïve Recursive Fibonacci Time Complexity")
plt.grid(True)
plt.show()
```



- **Dynamic Programming:-**
  - **Input:** Integer $n \geq 0$
  - **Output:** nth Fibonacci number
  - **Time Complexity:**
    - Best: O(1)
    - Average: O(n)
    - Worst: O(n)
  - **Space Usage:** O(n) for memoization, O(1) for iterative version
  - **Trade-offs:** Much more efficient; avoids recomputation. Iterative DP preferred for space efficiency.

```
# Dynamic Programming Fibonacci (Bottom-Up)
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]

# Values for DP
ns_dp = list(range(100, 2001, 200))
times_dp = [measure_time(fib_dp, n) for n in ns_dp]

# Plot
plt.figure(figsize=(8,5))
plt.plot(ns_dp, times_dp, marker='s', color='blue')
plt.xlabel("Input n")
plt.ylabel("Execution Time (seconds)")
plt.title("Dynamic Programming Fibonacci Time Complexity")
plt.grid(True)
plt.show()
```



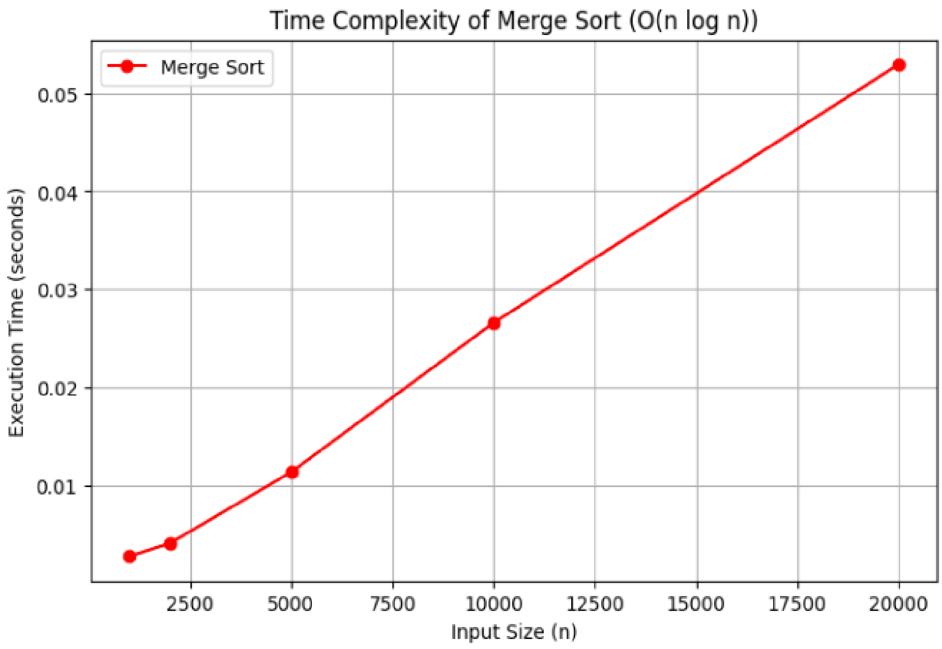
2. Merge Sort:-

- **Input:** Array of n elements
- **Output:** Sorted array
- **Time Complexity:**
  - Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n log n)
- **Space Usage:** O(n) (extra arrays for merging)
- **Trade-offs:** Stable sort; predictable performance. Extra memory required.

```
# Merge Sort Implementation
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)

def merge(left, right):
    result = []
    i = j = 0

# Merge two halves
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

# Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

```python
# Example
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original array:", arr)
print("Sorted array:", merge_sort(arr))

# Measure execution time
def measure_time(func, arr):
    start = time.time()
    func(arr)
    end = time.time()
    return end - start

# Generate test arrays of increasing sizes
sizes = [1000, 2000, 5000, 10000, 20000]
merge_times = []

for n in sizes:
    test_arr = [random.randint(0, 100000) for _ in range(n)]
    merge_times.append(measure_time(merge_sort, test_arr.copy()))

# Plot graph
plt.figure(figsize=(8,5))
plt.plot(sizes, merge_times, marker='o', color='red', label="Merge Sort")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity of Merge Sort (O(n log n))")
plt.legend()
plt.grid(True)
plt.show()
```

```
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```



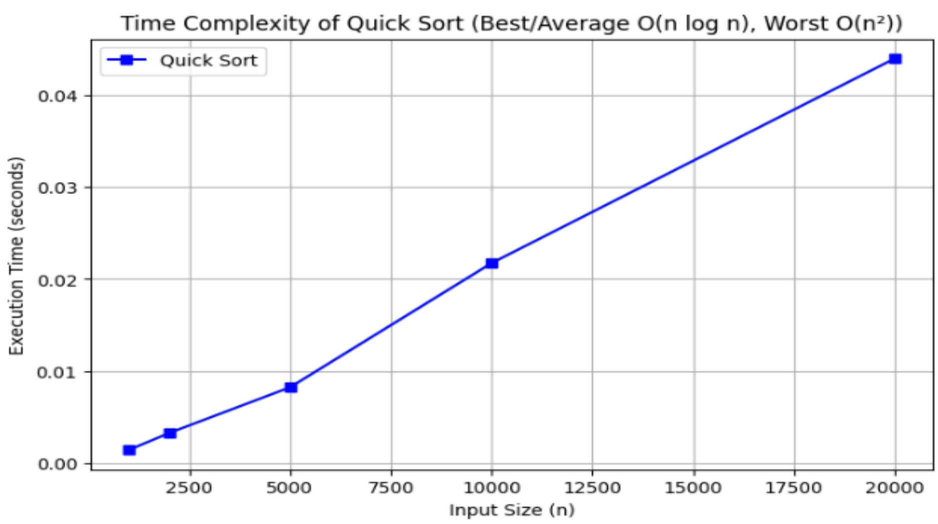
3. Quick Sort:-

- **Input:** Array of n elements
- **Output:** Sorted array
- **Time Complexity:**
  - Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n^2) (when pivot selection is poor)
- **Space Usage:** O(log n) (recursion stack)
- **Trade-offs:** Very fast in practice, in-place, but worst-case risk if pivot chosen poorly. Tail recursion optimizations can help.

```python
# Quick Sort Implementation
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Example run
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original array:", arr)
print("Sorted array:", quick_sort(arr))

# Measure execution time

def measure_time(func, arr):
    start = time.time()
    func(arr)
    end = time.time()
    return end - start
```

```python
# Generate test arrays of increasing sizes
sizes = [1000, 2000, 5000, 10000, 20000]
quick_times = []

for n in sizes:
    test_arr = [random.randint(0, 100000) for _ in range(n)]
    quick_times.append(measure_time(quick_sort, test_arr.copy()))

# Plot graph
plt.figure(figsize=(8,5))
plt.plot(sizes, quick_times, marker='s', color='blue', label="Quick Sort")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity of Quick Sort (Best/Average O(n log n), Worst O(n²))")
plt.legend()
plt.grid(True)
plt.show()
```

```
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```



4. Insertion Sort:-

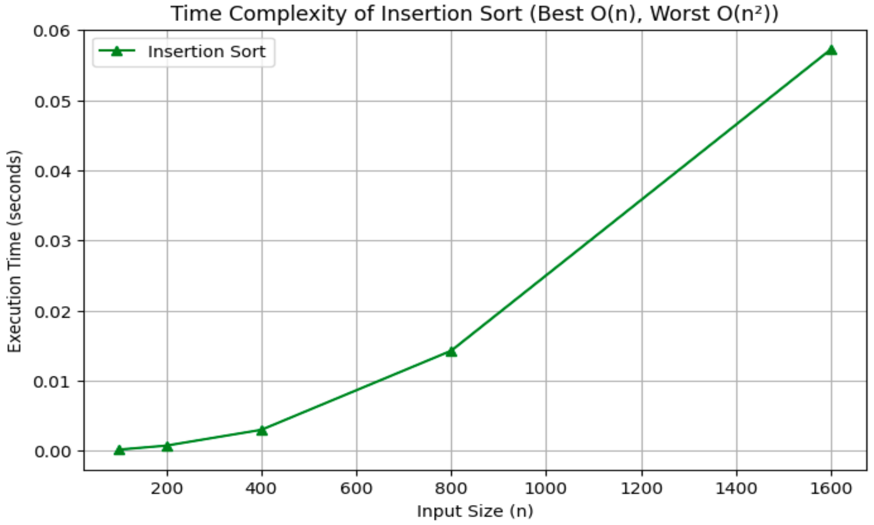
- **Input:** Array of n elements
- **Output:** Sorted array
- **Time Complexity:**
  - Best: O(n) (already sorted)
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space Usage:** O(1)

- **Trade-offs:** Efficient for small datasets or nearly sorted arrays. Poor for large datasets.

```python
# Insertion Sort Implementation
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements greater than key one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Example run
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original array:", arr)
print("Sorted array:", insertion_sort(arr.copy()))

# Measure execution time
def measure_time(func, arr):
    start = time.time()
    func(arr)
    end = time.time()
    return end - start
```

```python
# Generate test arrays of increasing sizes
sizes = [100, 200, 400, 800, 1600]
insertion_times = []

for n in sizes:
    test_arr = [random.randint(0, 100000) for _ in range(n)]
    insertion_times.append(measure_time(insertion_sort, test_arr.copy()))

# Plot graph
plt.figure(figsize=(8,5))
plt.plot(sizes, insertion_times, marker='^', color='green', label="Insertion Sort")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity of Insertion Sort (Best O(n), Worst O(n²))")
plt.legend()
plt.grid(True)
plt.show()
```

```
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```



5. Bubble Sort:-

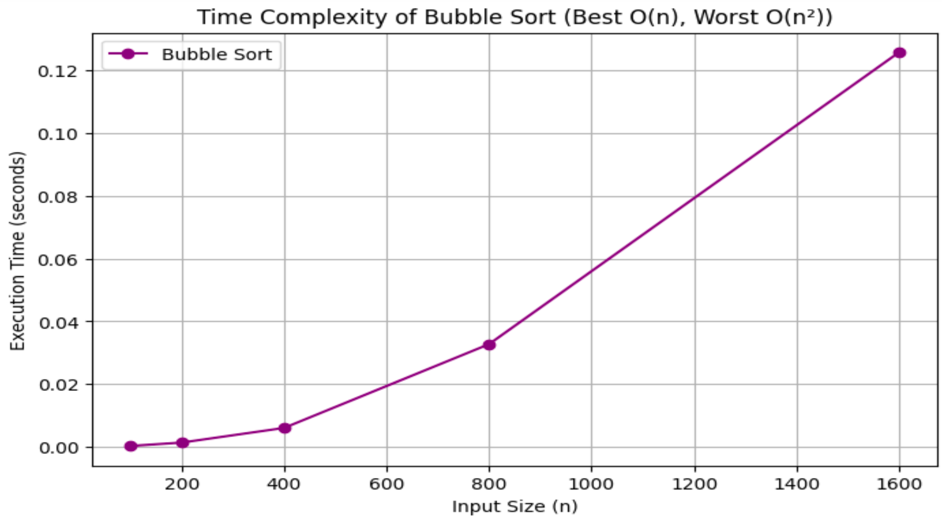
- **Input:** Array of n elements

- **Output:** Sorted array
- **Time Complexity:**
  - Best: O(n) (optimized version with swap check)
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space Usage:** O(1)
- **Trade-offs:** Very simple, but inefficient. Rarely used in real applications.

```python
# Bubble Sort Implementation
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

# Example run
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original array:", arr)
print("Sorted array:", bubble_sort(arr.copy()))

# Measure execution time
def measure_time(func, arr):
    start = time.time()
    func(arr)
    end = time.time()
    return end - start
```

```python
# Generate test arrays of increasing sizes
sizes = [100, 200, 400, 800, 1600]
bubble_times = []

for n in sizes:
    test_arr = [random.randint(0, 100000) for _ in range(n)]
    bubble_times.append(measure_time(bubble_sort, test_arr.copy()))

# Plot graph
plt.figure(figsize=(8,5))
plt.plot(sizes, bubble_times, marker='o', color='purple', label="Bubble Sort")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity of Bubble Sort (Best O(n), Worst O(n²))")
plt.legend()
plt.grid(True)
plt.show()
```

```
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```



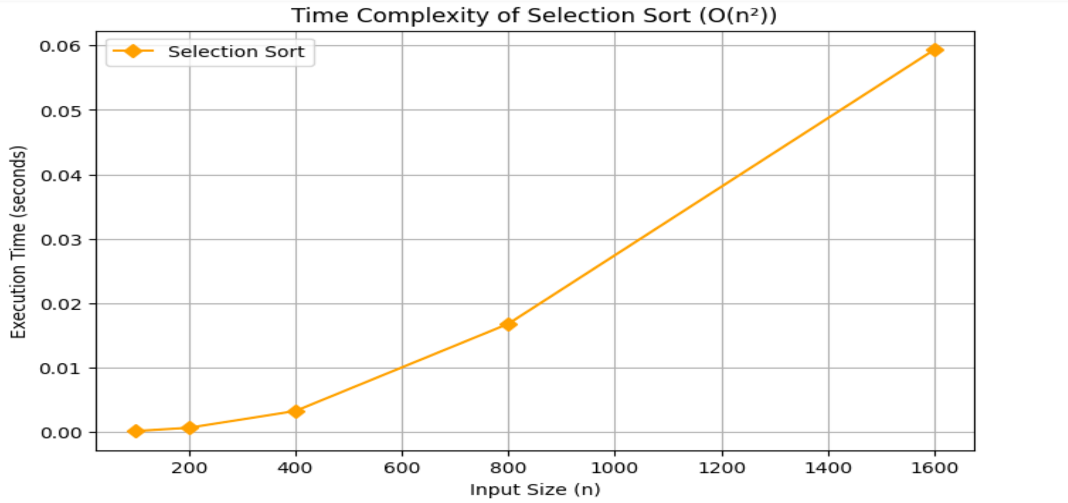
6. Selection Sort:-

- **Input:** Array of n elements
- **Output:** Sorted array
- **Time Complexity:**
  - Best: $O(n^2)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space Usage:** O(1)
- **Trade-offs:** Fewer swaps than Bubble Sort, but still inefficient. Useful when memory writes are costly.

```python
# Selection Sort Implementation
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]  # swap
    return arr

# Example run
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original array:", arr)
print("Sorted array:", selection_sort(arr.copy()))


# Measure execution time
def measure_time(func, arr):
    start = time.time()
    func(arr)
    end = time.time()
    return end - start
```

```python
# Generate test arrays of increasing sizes
sizes = [100, 200, 400, 800, 1600]  # small sizes (quadratic growth)
selection_times = []

for n in sizes:
    test_arr = [random.randint(0, 100000) for _ in range(n)]
    selection_times.append(measure_time(selection_sort, test_arr.copy()))

# Plot graph
plt.figure(figsize=(8,5))
plt.plot(sizes, selection_times, marker='D', color='orange', label="Selection Sort")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity of Selection Sort (O(n²))")
plt.legend()
plt.grid(True)
plt.show()
```

```
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```



7. Binary Search:-

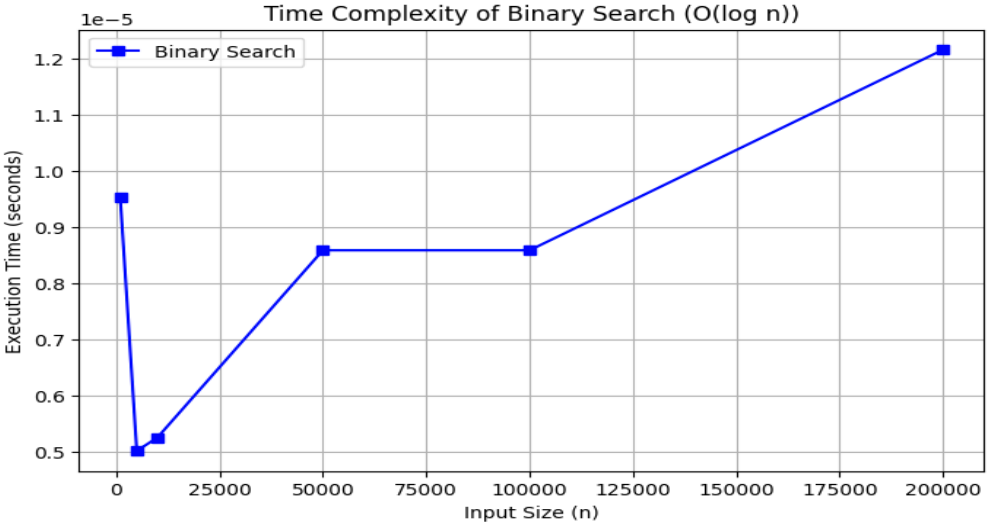
- **Input:** Sorted array of n elements and target key
- **Output:** Index of key (or -1 if not found)
- **Time Complexity:**
    - Best: O(1) (key is middle element)
    - Average: O(log n)
    - Worst: O(log n)
- **Space Usage:** O(1) iterative, O(log n) recursive (stack depth)
- **Trade-offs:** Very efficient but requires sorted input. Recursive version risks stack overflow for huge n.

```python
# Binary Search Implementation (Iterative)
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid  #found
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1  #not found

# Example run
arr = [3, 9, 10, 27, 38, 43, 82]  # must be sorted
target = 43
print("Array:", arr)
print(f"Target {target} found at index:", binary_search(arr, target))


# Measure execution time
def measure_time(func, arr, target):
    start = time.time()
    func(arr, target)
```

```python
    func(arr, target)
    end = time.time()
    return end - start

# Generate test arrays of increasing sizes
sizes = [1000, 5000, 10000, 50000, 100000, 200000]
binary_times = []

for n in sizes:
    test_arr = sorted([random.randint(0, n*10) for _ in range(n)])
    target = random.choice(test_arr)
    binary_times.append(measure_time(binary_search, test_arr, target))

# Plot graph
plt.figure(figsize=(8,5))
plt.plot(sizes, binary_times, marker='s', color='blue', label="Binary Search")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity of Binary Search (O(log n))")
plt.legend()
plt.grid(True)
plt.show()
```

```
Array: [3, 9, 10, 27, 38, 43, 82]
Target 43 found at index: 5
```

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

# 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

# Timer Helper
def measure_time(func, *args):
    start = time.time()
    func(*args)
    return time.time() - start
```



The combined graph of all the algorithms time complexity is:-

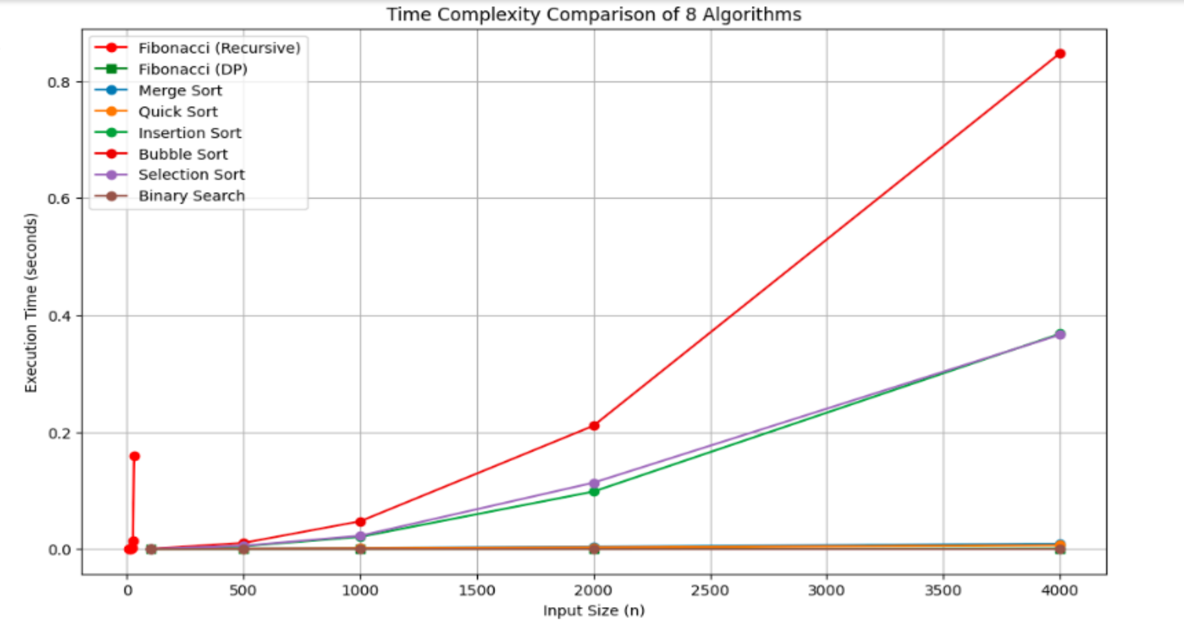
```python
# Fibonacci
def fib_recursive(n):
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)

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, 0
    while i < len(left) and j < len(right):
        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
```

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key, j = arr[i], i-1
        while j >= 0 and arr[j] > key:
            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):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
```

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

# 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

# Timer Helper
def measure_time(func, *args):
    start = time.time()
    func(*args)
    return time.time() - start
```

```
# Experiment
sizes_sort = [100, 500, 1000, 2000, 4000]
sizes_fib_recursive = list(range(5, 35, 5))   # small for recursive
sizes_fib_dp = [100, 500, 1000, 2000, 4000]

results = {
    "Fibonacci (Recursive)": [],
    "Fibonacci (DP)": [],
    "Merge Sort": [],
    "Quick Sort": [],
    "Insertion Sort": [],
    "Bubble Sort": [],
    "Selection Sort": [],
    "Binary Search": []
}

# Fibonacci Recursive
for n in sizes_fib_recursive:
    results["Fibonacci (Recursive)"].append(measure_time(fib_recursive, n))

# Fibonacci DP
for n in sizes_fib_dp:
    results["Fibonacci (DP)"].append(measure_time(fib_dp, n))

# Sorting + Binary Search
for n in sizes_sort:
    arr = [random.randint(0, n*10) for _ in range(n)]
    arr_sorted = sorted(arr)
```

```
    results["Merge Sort"].append(measure_time(merge_sort, arr.copy()))
    results["Quick Sort"].append(measure_time(quick_sort, arr.copy()))
    results["Insertion Sort"].append(measure_time(insertion_sort, arr.copy()))
    results["Bubble Sort"].append(measure_time(bubble_sort, arr.copy()))
    results["Selection Sort"].append(measure_time(selection_sort, arr.copy()))
    results["Binary Search"].append(measure_time(binary_search, arr_sorted, arr_sorted[n//2]))

# Plot
plt.figure(figsize=(12,7))

plt.plot(sizes_fib_recursive, results["Fibonacci (Recursive)"], marker='o', color='red', label="Fibonacci (Recursive)")
plt.plot(sizes_fib_dp, results["Fibonacci (DP)"], marker='s', color='green', label="Fibonacci (DP)")

for algo in ["Merge Sort", "Quick Sort", "Insertion Sort", "Bubble Sort", "Selection Sort", "Binary Search"]:
    plt.plot(sizes_sort, results[algo], marker='o', label=algo)

plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Time Complexity Comparison of 8 Algorithms")
plt.legend()
plt.grid(True)
plt.show()
```



1. Measuring Execution Time

Execution time was tested by gradually increasing the input size.

- **Naïve recursive Fibonacci** grew extremely fast and became impractical beyond n=35 due to exponential growth.
- **Dynamic programming Fibonacci** scaled linearly and handled large values easily.
- **Merge Sort** and **Quick Sort** showed consistent logarithmic growth with input size.
- **Bubble, Insertion, and Selection Sort** slowed down quickly, confirming their quadratic complexity.
- **Binary Search** performed almost instantly, even with large datasets, because of its logarithmic efficiency.

2. Measuring Memory Usage

Memory profiling showed interesting trade-offs:

- Recursive algorithms such as **naïve Fibonacci** and **recursive Quick Sort** consumed extra memory because of the call stack.
- **Merge Sort** required additional memory to store temporary arrays during merging.
- **Iterative Fibonacci, Binary Search, and Insertion Sort** were memory-efficient, using only constant space.

3. Trade-offs in Time vs. Space

From the experiments, we noticed a clear pattern: algorithms that save time often use more memory.

- **Merge Sort** is very fast but needs extra memory.
- **Dynamic Programming Fibonacci** uses more memory than the naïve version but drastically reduces runtime.

- On the other hand, **Selection Sort** and **Insertion Sort** are memory-friendly but slow for large inputs.

4. Recursive Depth and Stack Overflow Risks

Certain algorithms rely heavily on recursion. While recursion simplifies the logic, it increases the risk of stack overflow if the input grows too large:

- **Naïve Fibonacci** has the highest risk because it makes repeated recursive calls.
- **Recursive Quick Sort** may also face stack depth issues if the pivot is consistently chosen poorly.
- **Merge Sort**, though recursive, has a balanced depth of O(log n), which is generally safe.
- **Binary Search (recursive version)** is also safe for practical input sizes, but iterative implementation avoids recursion entirely.

5. Visualization

Using matplotlib, execution times were plotted against input sizes.

- Exponential growth in the naïve Fibonacci stood out sharply.
- Merge Sort and Quick Sort formed smoother curves with logarithmic growth.
- Insertion, Selection, and Bubble Sort showed steep quadratic curves.
- Binary Search appeared almost flat, highlighting its efficiency.

Comparison Table of Algorithms

Algorithm	Input/Output	Best Case	Average Case	Worst Case	Space Usage	Notes & Trade-offs
Fibonacci (Naïve)	n → F(n)	O(1)	O(2^n)	O(2^n)	O(n)	Simple but exponential time
Fibonacci (DP)	n → F(n)	O(1)	O(n)	O(n)	O(1)–O(n)	Very efficient
Merge Sort	Array → Sorted	O(n log n)	O(n log n)	O(n log n)	O(n)	Stable, predictable
Quick Sort	Array → Sorted	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	Very fast, pivot-dependent
Insertion Sort	Array → Sorted	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Best for small/n-early sorted arrays
Bubble Sort	Array → Sorted	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Rarely practical
Selection Sort	Array → Sorted	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Fewer swaps, inefficient
Binary Search	Sorted Array, Key → Index	O(1)	O(log n)	O(log n)	O(1)/O(log n)	Requires sorted input