A blue and white sign

Description automatically generated

**DESIGN AND ANALYSIS OF ALGORITHMS LAB**

**ENCA351**

Bachelor of computer application

(Specialization in AI & DS)

**LAB ASSIGNMENT-1**

**Semester – 5th**

**SUBMITTED BY: SUBMITTED TO:**

Yash Dr AARTI

2301201198

**Assignment Title: Analyzing and Visualizing Recursive Algorithm Efficiency**

**Real-World Problem Context**:

Software engineers and computer scientists often face the challenge of choosing the most efficient algorithm for a given problem. Recursive and iterative algorithms, though conceptually simple, vary significantly in performance based on input size and implementation strategy. The goal of this project is to analyze and compare common sorting and searching algorithms, visualize their time and space behavior, and understand the practical impact of their design.

**SECTION 1: REQUIRED LIBRARIES**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**OUTPUT:**

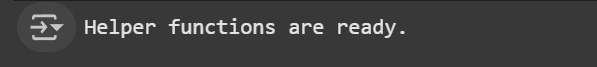
**A screen shot of a computer

AI-generated content may be incorrect.**

**2: Helper Functions for Profiling**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

****

**1a. Naive Recursive Fibonacci**

A black background with white text

AI-generated content may be incorrect.

**1b. Dynamic Programming (Memoization) Fibonacci**

A computer screen shot of a black background

AI-generated content may be incorrect.



**Performance Analysis**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**OUTPUT:**

**A graph with a line graph

AI-generated content may be incorrect.**

**Task 2.2 - Sorting Algorithms**

1. **Merge Sort**

* **Input/Output**: An unsorted list of numbers / A sorted list.
* **Time Complexity**: Best/Average/Worst: O(nlogn).
* **Space Complexity**: O(n) for the temporary arrays used during merging.
* **Suitability**: Excellent for large datasets. It's a stable sort.

**A screenshot of a computer program

AI-generated content may be incorrect.**

1. **Quick Sort**

* **Input/Output**: An unsorted list / A sorted list.
* **Time Complexity**: Best/Average: O(nlogn). Worst: O(n2) (occurs with sorted/reverse-sorted input and a poor pivot choice).
* **Space Complexity**: O(logn) on average (recursion stack), O(n) in the worst case.
* **Suitability**: Generally faster in practice than Merge Sort due to lower constant factors and cache efficiency. Not a stable sort.

**A computer screen shot of a program code

AI-generated content may be incorrect.**

1. **Insertion Sort**
   * **Input/Output**: An unsorted list / A sorted list.
   * **Time Complexity**: Best: O(n) (nearly sorted). Average/Worst: O(n2).
   * **Space Complexity**: O(1) (in-place).
   * **Suitability**: Very efficient for small or nearly sorted datasets.

A computer screen shot of a code

AI-generated content may be incorrect.

1. **Bubble Sort**

* **Input/Output**: An unsorted list / A sorted list.
* **Time Complexity**: Best: O(n) (with optimization). Average/Worst: O(n2).
* **Space Complexity**: O(1) (in-place).
* **Suitability**: Mostly educational. Inefficient for almost all real-world scenarios.

**A computer screen shot of a number

AI-generated content may be incorrect.**

1. **Selection Sort**

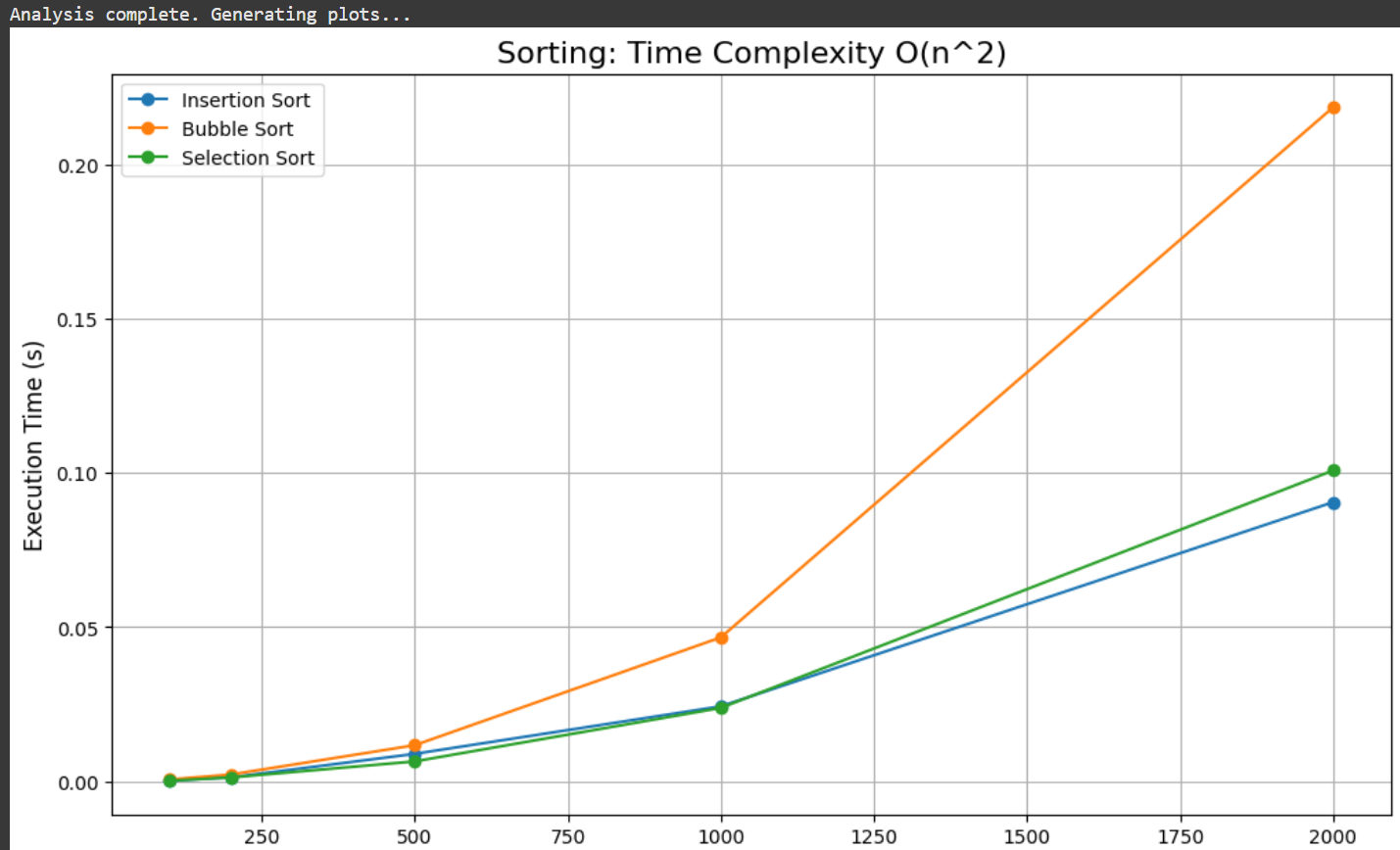
* **Input/Output**: An unsorted list / A sorted list.
* **Time Complexity**: Best/Average/Worst: O(n2).
* **Space Complexity**: O(1) (in-place).
* **Suitability**: Simple to implement but inefficient. Its main advantage is making the minimum number of swaps.

**A computer screen shot of a program code

AI-generated content may be incorrect.**

**OUTPUT:**

****

****

**A graph with a line

AI-generated content may be incorrect.**

**A graph with a purple line and a red line

AI-generated content may be incorrect.**

**2.3 - Binary Search**

**A computer screen with text

AI-generated content may be incorrect.**

**OUTPUT:**

**A graph with a line

AI-generated content may be incorrect.**

**Suitability and Trade-offs:**

1. **Fibonacci (DP vs. Naïve)**

* **Suitability: DP version is efficient for practical use; Naïve version is purely educational.**
* **Trade-off: DP uses extra memory (O(n)) to achieve a huge speedup (from exponential to linear time).**

1. **Merge Sort**

* **Suitability: Best for large datasets where a guaranteed O(nlogn) performance is critical.**
* **Trade-off: Requires significant extra memory (O(n) space).**

1. **Quick Sort**

* **Suitability: Excellent general-purpose sort; often the fastest in practice on average.**
* **Trade-off: Carries a small risk of poor (O(n2)) worst-case performance.**

1. **Insertion Sort**

* **Suitability: Very fast for small lists or lists that are already mostly sorted.**
* **Trade-off: Performance degrades quickly to O(n2) on large, unsorted lists.**

1. **Bubble Sort**

* **Suitability: For educational purposes only; not practical for real-world use.**
* **Trade-off: Simple to understand but extremely inefficient.**

1. **Selection Sort**

* **Suitability: Niche cases where minimizing the number of swaps is the top priority.**
* **Trade-off: Slow (O(n2)) and performance never improves, even on sorted data.**

1. **Binary Search**

* **Suitability: The fastest way to search within a large, pre-sorted list.**
* **Trade-off: The list must be sorted first, which is a significant upfront cost.**
  1. **Conclusion**

The simple, brute-force approach of the O(n2) sorts (Bubble, Selection, Insertion) proved inefficient and unable to scale. In contrast, the "divide and conquer" strategy of Merge and Quick Sort was far more effective on large datasets, showcasing how breaking down a problem is a superior approach.

The most powerful lesson came from the Fibonacci functions. The dramatic speedup of the Dynamic Programming version showed that the smartest strategy is often just remembering the answer to a problem you've already solved, trading a little memory for an enormous gain in time.

**Recursive Depth and Stack Overflow Risks:**

* 1. **Quick Sort:**
* This algorithm was the most practically affected by recursion limits in our tests.
* **Risk:** The recursion depth of Quick Sort depends on the pivot selection. In the worst-case scenario (e.g., on a sorted list), the depth can be linear (O(n)). For an input of 20,000, this would require a stack depth of 20,000, which far exceeds Python's default limit (~1000).
* **Observed Behavior:** This is precisely why we encountered the RecursionError. We had to manually increase the limit using sys.setrecursionlimit() to allow the algorithm to finish on large inputs.
  1. **Merge Sort:**
* This algorithm is highly resistant to stack overflow issues.
* **Risk:** Very low. Because Merge Sort always splits the array in half, its recursion depth is guaranteed to be logarithmic (O(logn)).
* **Observed Behavior:** For an input of 20,000, the recursion depth is only about 15 (log2​20000≈14.3), which is nowhere near the limit. This makes Merge Sort a much safer choice in environments where recursion depth is a concern.
  1. **Fibonacci Algorithms:**
* Both recursive Fibonacci implementations have a potential recursion depth of O(n).
* **Naïve Version Risk:** While theoretically at risk, this algorithm's exponential time complexity (O(2n)) makes it so slow that it becomes unusable for n > 40, long before the stack overflow limit is ever reached.
* **Dynamic Programming Version Risk:** Because this version is very fast (O(n) time), it is entirely possible to cause a RecursionError by calling it with a large number (e.g., n = 1500). For this reason, an iterative (bottom-up) approach is often preferred for calculating Fibonacci numbers when n can be very large.