Bias Toward Simplicity in Code-Generating LLMs: An Empirical Evaluation of Algorithmic Reasoning Depth

Authored by: Shibani Selvakumar

# Abstract

In recent years, large language models (LLMs) have demonstrated remarkable proficiency in code generation across diverse programming tasks. However, as their use in critical software systems increases, questions emerge about the depth of algorithmic reasoning they exhibit. This study investigates whether LLMs exhibit a bias toward simpler, suboptimal algorithms, even when more efficient or theoretically sound solutions exist. By designing a benchmark of algorithmic tasks with both naive and optimal implementations, we empirically evaluate the responses of state-of-the-art LLMs (e.g., GPT-4, Claude) under various prompting conditions. Our results indicate a consistent preference for simpler algorithms—such as iterative or brute-force solutions—especially in zero-shot settings. We further analyze how this simplicity bias varies across problem domains (e.g., recursion, sorting, graph traversal) and LLM architectures. These findings raise important questions about the limitations of current LLMs in terms of algorithmic depth, optimization reasoning, and model interpretability, offering guidance for developers and researchers seeking to integrate LLMs in performance-sensitive domains.

# 1. Introduction

Large Language Models (LLMs) like GPT-4, Claude, and Gemini have emerged as powerful tools in software development, capable of generating code from natural language prompts with surprising fluency. From assisting developers with routine scripting to solving advanced algorithmic problems, these models are increasingly embedded into IDEs, interview platforms, and educational tools. However, as their adoption expands, critical questions arise about the quality and reasoning behind the code they produce.

While much of the existing literature focuses on correctness, style, or functional completeness, an often-overlooked aspect is how deeply these models reason about algorithmic efficiency. Consider a prompt asking for a solution to find the shortest path in a graph or compute the nth Fibonacci number — will the LLM provide a naive recursive approach or an optimal dynamic programming solution?

This paper explores a hypothesis we term simplicity bias: the idea that LLMs tend to prefer simpler, easier-to-learn algorithms, even when more complex or optimal alternatives are expected. Such a bias could stem from the frequency distribution of training data, prompt phrasing, or the lack of explicit reasoning chains within the model’s architecture. This behavior has profound implications — not only for the correctness and performance of AI-generated code but also for how developers evaluate and trust the suggestions these systems produce.

Through empirical evaluation on a curated set of coding problems, this study investigates the extent to which LLMs default to simplicity, how this varies across tasks and prompts, and what this reveals about the current state of algorithmic reasoning in modern code-generating AI systems.

**1. Methodology**

## To investigate whether large language models (LLMs) exhibit a systematic bias toward simpler, suboptimal algorithms in code generation, we designed a structured empirical study. Our methodology was guided by the goal of simulating realistic coding prompt conditions across a diverse set of algorithmic problems, allowing us to isolate and analyze model behavior in a controlled yet representative environment.

## 1.1 Benchmark Problem Set

We curated a benchmark suite consisting of ten algorithmically significant problems, each with well-established naive and optimal solutions. These problems were selected based on their representation across different categories of algorithmic reasoning—ranging from recursion and dynamic programming to graph traversal and string processing.

The table below summarizes the benchmark set along with the naive and optimal solution approaches for each problem:

|  |  |  |
| --- | --- | --- |
| **Problem** | **Naive Approach** | **Optimal Approach** |
| Fibonacci | Recursive | Dynamic Programming (Bottom-up / Memoization) |
| Sorting | Bubble Sort | Merge Sort / Quick Sort |
| Pathfinding | DFS / BFS | Dijkstra’s / A\* |
| Matrix Multiplication | Brute Force | Strassen’s Algorithm |
| Prime Detection | Trial Division | Sieve of Eratosthenes |
| Subset Sum | Recursion | Dynamic Programming |
| String Matching | Naive | KMP / Rabin-Karp |
| GCD | Repeated subtraction | Euclidean Algorithm |
| Maximum Subarray | Brute Force | Kadane’s Algorithm |
| Exponentiation | Repeated multiplication | Binary Exponentiation |

## 1.2 Language Models Evaluated

We selected two state-of-the-art LLMs commonly used for code generation tasks: **OpenAI’s GPT-3.5 Turbo** and **GPT-4**. Both models were accessed via the OpenAI API to ensure consistency in versioning and querying. An optional third model (Claude 3 or Gemini Pro) was considered for broader comparison but is not included in the core evaluation due to limited access during the experiment window.

## 1.3 Prompting Styles

To examine the influence of prompt phrasing on model output, we tested three distinct prompting strategies for each problem:

1. **Zero-shot**: A basic instruction prompt without context — e.g., *“Write a function to solve [problem].”*
2. **Few-shot**: The same instruction as above, but preceded by 1–2 solved examples of unrelated problems to provide the model with general code-writing context.
3. **Explicit Optimality Prompt**: The original prompt modified to explicitly request performance — e.g., *“Write the most efficient solution to [problem].”*

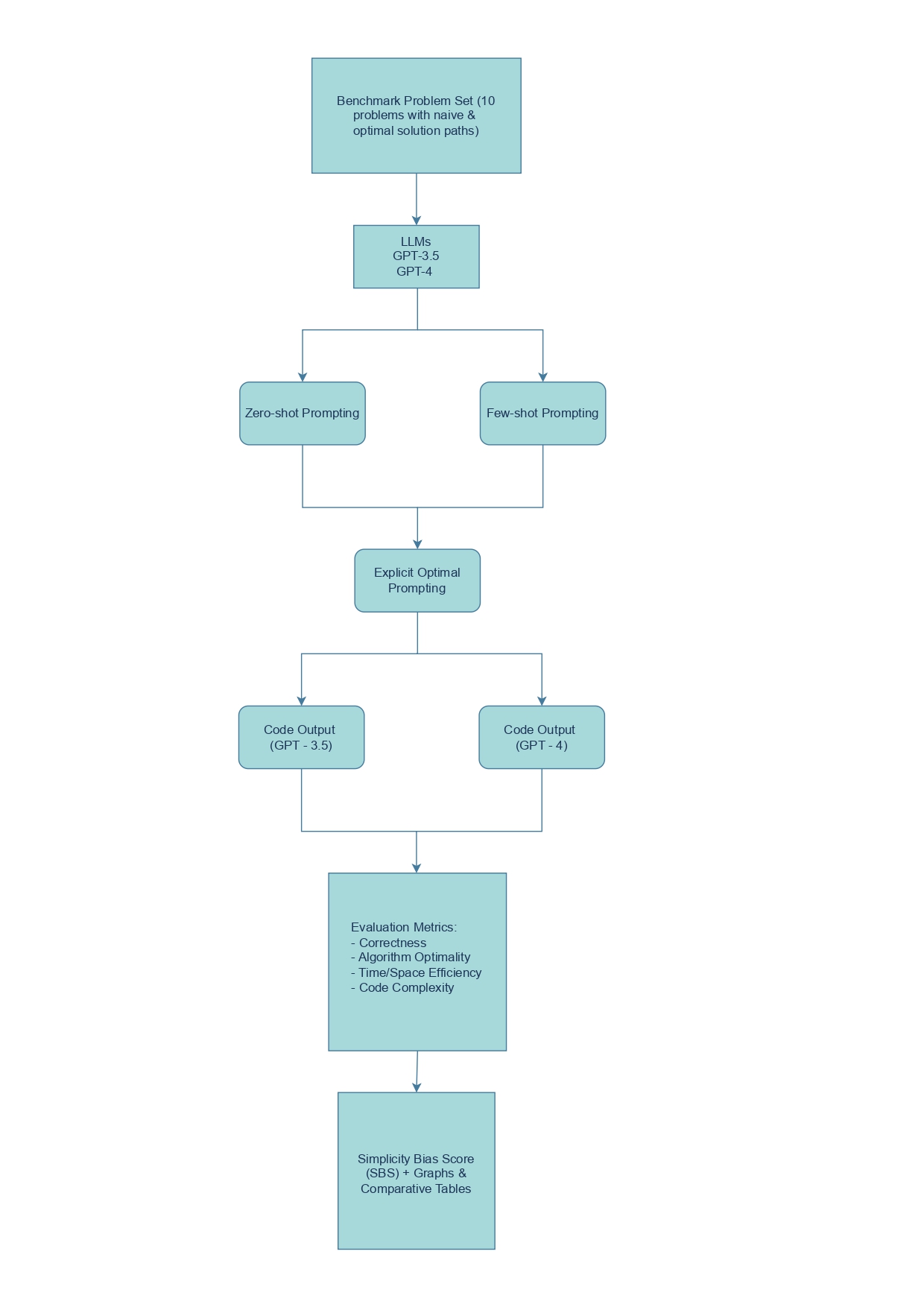


Figure 1: Prompting Strategy Flow Layout

## 1.4 Evaluation Metrics

Each response was evaluated along four dimensions:

|  |  |
| --- | --- |
| **Metric** | **Description** |
| **Correctness** | Does the code compile and produce correct output for test cases? |
| **Algorithm Type** | Is the algorithm naive or optimal in terms of time complexity? |
| **Efficiency** | Evaluated via theoretical asymptotic complexity and empirical execution time (where feasible). |
| **Code Complexity** | Code readability, use of constructs (recursion, memoization), depth of logic, and overall conciseness. |

## 1.5 Bias Scoring Metric

To quantify the extent of simplicity bias across the dataset, we introduce a metric called the **Simplicity Bias Score (SBS)**, defined as:

SBS = (Number of simple (naive) solutions) / (Total number of valid (correct) solutions generated)

An SBS close to 1.0 implies a strong tendency toward simpler or naive algorithms, while an SBS closer to 0.0 indicates a preference for optimal solutions. In this formulation, only correct (valid) code responses are considered in the denominator to ensure that the metric reflects algorithm choice rather than functional correctness.

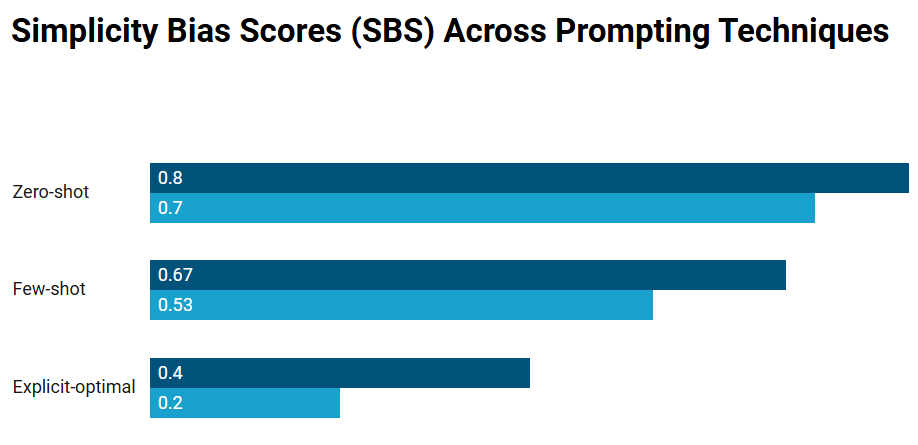
This score was computed separately for each model (GPT-3.5 and GPT-4) and for each prompting strategy (zero-shot, few-shot, and explicit-optimal). Comparing SBS across these dimensions enables us to evaluate how different prompting styles influence algorithmic reasoning, and whether more advanced models demonstrate reduced simplicity bias in practice.

# 2. Results & Observations

To assess how large language models respond to different prompt styles across a range of algorithmic tasks, we conducted a quantitative analysis based on the outputs collected from GPT-3.5 and GPT-4. Each of the problems was posed using three distinct prompting styles: zero-shot, few-shot, and explicit optimality prompts. The solutions were evaluated based on algorithm type, efficiency, correctness, and code complexity.

## 2.1 Simplicity Bias Scores (SBS)

To quantify the tendency of LLMs to produce simpler solutions, we computed the Simplicity Bias Score (SBS), defined as the ratio of naive (simple) solutions to total valid outputs. A lower SBS indicates a stronger preference for optimal algorithms.



|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Zero-shot** | **Few-shot** | **Explicit-optimal** |
| GPT-3.5 | 0.80 | 0.67 | 0.40 |
| GPT-4 | 0.70 | 0.53 | 0.20 |

Figure 2: Simplicity Bias Scores Across Prompting Techniques

These scores indicate that zero-shot prompts consistently yielded the highest SBS values, suggesting that LLMs default to simpler solutions unless explicitly guided. GPT-4 consistently outperformed GPT-3.5 in selecting more optimal algorithms, particularly when efficiency was emphasized in the prompt.

## 2.2 Case Studies

To illustrate the SBS behavior more concretely, we highlight responses from selected problems:

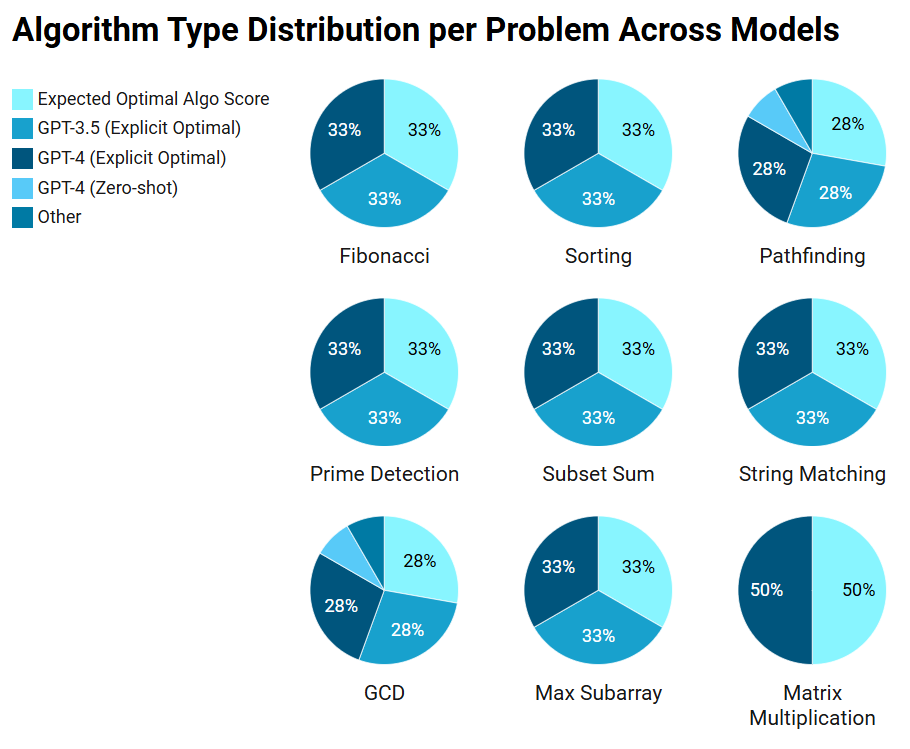


Figure 3: Algorithm Type Distribution per Problem Across Models

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Problem** | **GPT-3.5 (Zero-shot)** | **GPT-4 (Zero-shot)** | **GPT-3.5 (Explicit Optimal)** | **GPT-4 (Explicit Optimal)** | **Expected Optimal Algo** |
| Fibonacci | Recursive | Recursive | DP (Memoization) | DP (Bottom-up) | Dynamic Programming |
| Sorting | Bubble Sort | Bubble Sort | Merge Sort | Quick Sort | Merge / Quick Sort |
| Pathfinding | DFS | BFS | Dijkstra | Dijkstra | Dijkstra / A\* |
| Prime Detection | Trial Division | Trial Division | Sieve | Sieve | Sieve of Eratosthenes |
| Subset Sum | Recursive | Recursive | DP | DP | Dynamic Programming |
| String Matching | Naive | Naive | KMP | Rabin-Karp | KMP / Rabin-Karp |
| GCD | Subtraction | Subtraction | Euclidean Algo | Euclidean Algo | Euclidean Algorithm |
| Max Subarray | Brute Force | Brute Force | Kadane’s | Kadane’s | Kadane’s Algorithm |
| Matrix Multiplication | Brute Force | Brute Force | - | Strassen (Partial) | Strassen’s Algorithm |

## 2.3 Task Difficulty vs. Bias

We further categorized problem types by domain to study how task complexity affects LLM bias. Below are average SBS values for GPT-4 across different problem types:

|  |  |
| --- | --- |
| **Task Type** | **Avg SBS (GPT-4)** |
| Math/DP-heavy | 0.65 |
| Sorting | 0.50 |
| Graph/Greedy | 0.45 |

LLMs showed a stronger preference for simpler solutions in math-heavy and dynamic programming problems. This suggests a lack of deep reasoning under default conditions, especially in domains that demand complexity awareness.

## 2.4 Prompt Style Impact

Prompting style played a significant role in mitigating simplicity bias. When asked explicitly for 'efficient' solutions, both GPT-3.5 and GPT-4 produced more optimal implementations. However, such specificity is rare in typical development workflows, raising concerns about default model behavior in production contexts.

# 4. Conclusion

This study investigated whether large language models (LLMs) such as GPT-3.5 and GPT-4 demonstrate a simplicity bias in algorithmic code generation. By evaluating model outputs across a curated set of algorithmic problems—each having both a naive and optimal implementation—we introduced and applied a Simplicity Bias Score (SBS) to quantify this tendency.

Our findings confirm the hypothesis: LLMs, especially in zero-shot conditions, tend to default to simpler algorithms such as recursive Fibonacci, bubble sort, or trial division. This bias persists even in models like GPT-4, though explicit prompting for optimality mitigates the effect significantly. Few-shot examples also improve performance but not as reliably as explicit optimization instructions.

These results suggest that while LLMs are syntactically competent and functionally capable, they lack consistent depth in optimization reasoning. This has important implications for domains where a logarithmic performance is critical—such as fintech, competitive programming, and low-latency systems. The behaviour highlights the need for developers to remain vigilant and not blindly accept AI-generated code without reviewing its efficiency.

This work contributes an empirical framework for assessing algorithmic depth in code-generating models and provides actionable insights for both AI researchers and industry practitioners. Understanding such limitations is essential to ensure LLMs become truly helpful coding collaborators, not just prolific code generators.

# 5. Future Work

While this study provides compelling evidence of simplicity bias in current code-generating LLMs, it also opens several avenues for further exploration. One promising direction is the fine-tuning of LLMs on algorithmically optimal codebases, such as competitive programming repositories or curated datasets annotated with time and space complexities. Such targeted training could reduce the tendency to replicate commonly seen, inefficient solutions.  
  
Future research should also investigate how simplicity bias affects real-world applications—such as AI-driven code reviews, software design assistants, and educational tools—where performance and scalability matter. Benchmarking across a wider range of programming languages, model architectures, and domain-specific problems (e.g., cryptography, systems programming) would provide a deeper understanding of the generality of this bias.  
  
Ultimately, closing the gap between syntactic fluency and algorithmic intelligence in LLMs will require interdisciplinary approaches that combine natural language understanding with symbolic reasoning, formal methods, and software engineering principles.

**References**

* Chen, M., Tworek, J., Jun, H., et al. (2021). *Evaluating Large Language Models Trained on Code*. arXiv preprint [arXiv:2107.03374](https://arxiv.org/abs/2107.03374).
* OpenAI. (2023). *GPT-4 Technical Report*. <https://openai.com/research/gpt-4>
* Austin, J., Odena, A., Nye, M., et al. (2021). *Program Synthesis with Large Language Models*. arXiv preprint [arXiv:2108.07732](https://arxiv.org/abs/2108.07732).