Analyzing Binary Search Problem Solutions: Comparing Human vs. LLM-Generated Code Using EffiBench

Teja Reddy Mandadi
Texas A&M University-San Antonio, tmand02@jaguar.tamu.edu

Visesh Bentula
Texas A&M University-San Antonio, vbent01@jaguar.tamu.edu

Umapathi Konduri
Texas A&M University-San Antonio, ukond01@jaguar.tamu.edu

1 Abstract

Large Language Models (LLMs) are increasingly used to generate code for algorithmic problems, but their efficiency compared to human written solutions remains underexplored. This paper compares the execution time and memory usage of binary search implementations written by humans versus LLMs specifically ChatGPT, Google's Gemini, and Anthropic's Claude[1]–[3]. We utilize the EffiBench benchmark[4] to evaluate 30 binary search problems across three difficulty levels: easy, medium, and hard. To ensure fair testing, all models were evaluated under identical runtime environments using the same problem prompts.

Our results show that Google's Gemini is the fastest and most efficient overall, outperforming human solutions in many cases. ChatGPT offers a balanced trade off between speed and memory usage, while Claude delivers consistent performance but tends to consume slightly more memory. LLM generated code also demonstrates strong consistency across problem complexities, particularly in the medium and hard categories. Interestingly, human written solutions occasionally surpass all LLMs, showcasing the advantage of deep algorithmic insight and manual optimization.

These findings indicate that the performance gap between LLMs and human coding is narrowing, highlighting the growing potential of AI assisted programming. We conclude by discussing the implications of these results for software development workflows and propose future work involving other algorithm types and fine tuning LLMs for performance critical applications[5], [6].

2 Introduction

Large Language Models (LLMs) like OpenAI's ChatGPT, Google's Gemini, and Anthropic's Claude[1]–[3] are changing the way software is developed by automatically generating code. These models are widely used to tackle coding challenges and algorithmic problems, including classic tasks like binary search. While most existing research on AI generated code has focused on accuracy and problem solving capabilities using benchmarks such as OpenAI's HumanEval and Google's MBPP[7], [8]—there has been far less attention paid to how efficient this code actually is when it runs. Yet in real world applications, performance—how fast code runs and how much memory it uses can be just as important as correctness, especially when dealing with large scale systems or limited resources. This gap in evaluation is what motivated our study.

Binary search is an excellent choice for comparing the efficiency of different solutions. It is a well known algorithm that searches for an element in a sorted list in $O(\log n)$ time by repeatedly halving the search space. Because the algorithm is conceptually simple and its optimal form is widely understood, it is easier to identify inefficiencies in implementation — such as unnecessary steps or suboptimal logic. Binary search also appears frequently in coding interviews and competitive programming, often with variations like searching in rotated arrays, finding bounds, or locating peaks. Despite its relevance, few studies have directly compared **human written** and **LLM generated** solutions for this problem from an efficiency standpoint across varying levels of difficulty. This is the gap our study aims to address.

To do this, we used the EffiBench benchmarking framework[4], which includes 1,000 algorithmic problems from LeetCode[9] and a set of efficient, verified human solutions. EffiBench measures both correctness and performance, tracking metrics like execution time and memory usage. From this dataset, we selected 30 binary search problems and compared how efficiently solutions from different sources—humans and LLMs—performed. Our goal was to see if modern LLMs can not only solve problems correctly, but also write code that runs efficiently, and to explore how this performance varies based on problem difficulty.

The rest of the paper outlines our benchmarking methodology, presents our findings with detailed analysis, and discusses what these results mean for the future of AI assisted coding. By looking at both correctness and efficiency, our work adds a valuable perspective to how LLMs are evaluated in programming tasks[10].

3 Binary Search Overview

Binary search is a fundamental algorithm used to efficiently locate a target element within a sorted array or list. It operates using a divide and conquer strategy, repeatedly halving the search space until the desired element is found or the search interval is empty. This makes it significantly faster than linear search for large datasets, especially when random access is supported. Its elegance and efficiency make it a staple in both academic instruction and real world software systems..

Working Principle

The algorithm begins by comparing the target value to the middle element of the array:

- If the target equals the middle element, the index is returned.
- If the target is less than the middle element, the search continues in the left half.
- If the target is greater, the right half is searched.

This process continues until the element is found or the search bounds converge.

Algorithm Requirements

- The input list must be sorted in ascending or descending order.
- The data structure must support random access (e.g., arrays or lists).

Time and Space Complexity

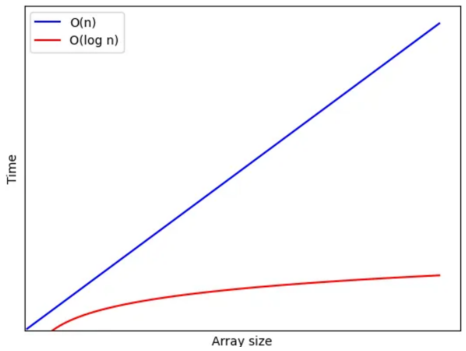

Figure 1: **Comparison of Linear vs. Logarithmic Time Complexity. The plot illustrates how the runtime of a linear algorithm $O(n)$ grows significantly faster than that of a logarithmic algorithm $O(\log n)$ as the input size increases. This highlights the efficiency advantage of algorithms like binary search, which scale logarithmically and are thus far more suitable for large datasets.**

- **Best Case:** $O(1)$ (when the target is the middle element)

- **Average and Worst Case:** $O(\log n)$ (each comparison halves the search space)
- **Space Complexity:** $O(1)$ for iterative version, $O(\log n)$ for recursive (due to call stack)

Use Cases

Binary search is widely used in:

- Searching in sorted arrays, lists, and trees.
- Solving problems involving bounds (e.g., lower bound, upper bound).
- Real world applications such as dictionaries, search engines, scheduling, and APIs.

Relevance to Benchmarking

Due to its simplicity, widespread usage, and well known optimal implementation, binary search is an ideal candidate for comparing the performance of human written versus LLM generated code. Subtle inefficiencies or deviations from the optimal form can be easily detected, making it an excellent benchmark for evaluating correctness, execution time, and memory usage.

4 Methodology

4.1 Benchmarking Overview

We conducted our experiments using the `EffiBench` benchmarking workflow to ensure consistent and fair evaluation of each solution. All tests were executed in a controlled runtime environment to eliminate external variability and ensure reliable comparisons. Each selected problem was evaluated using the same set of inputs across all solution sources to collect uniform performance data.

Our evaluation focused on three primary metrics for each solution: **execution time**, **average memory usage**, and **peak memory usage**. Based on these, we also computed a composite **Efficiency Score** for each solution, defined as the product of **execution time** and **peak memory** (**Time** $\times$ **Max Memory**), capturing overall efficiency in a single value. A lower **Efficiency Score** corresponds to a faster and more memory efficient solution.

4.2 Problem Selection

We narrowed our study to **30 binary search problems** sourced from the EffiBench dataset. These problems were carefully selected to ensure an even distribution across three difficulty levels: 10 Easy, 10 Medium, and 10 Hard. By focusing exclusively on binary search, we ensured that all solutions were based on the same underlying algorithmic logic, allowing for consistent and meaningful comparisons in terms of performance. This consistency in problem type removes variability that could otherwise arise from comparing different algorithms.

Moreover, by incorporating problems across varying levels of difficulty, we were able to explore whether differences in efficiency between human written and LLM generated solutions become more noticeable as the problems increase in complexity or involve more edge case heavy scenarios.

Easy		Medium		Hard	
Problem No	**Problem Name**	**Problem No**	**Problem Name**	**Problem No**	**Problem Name**
69	Sqrt(x)	81	Search in Rotated Sorted Array II	4	Median of Two Sorted Arrays
349	Intersection of Two Arrays	153	Find Minimum in Rotated Sorted Array	154	Find Minimum in Rotated Sorted Array II
350	Intersection of Two Arrays II	162	Find Peak Element	315	Count of Smaller Numbers After Self
367	Valid Perfect Square	209	Minimum Size Subarray Sum	354	Russian Doll Envelopes
744	Find Smallest Letter Greater Than Target	275	H-Index II	410	Split Array Largest Sum
888	Fair Candy Swap	400	Nth Digit	493	Reverse Pairs
1346	Check if N and Its Double Exist	475	Heaters	668	Kth Smallest Number in Multiplication Table
1351	Count Negative Numbers in a Sorted Matrix	532	K-diff Pairs in an Array	778	Swim in Rising Water
1385	Find the Distance Value Between Two Arrays	611	Valid Triangle Number	793	Preimage Size of Factorial Zeroes Function
1539	Kth Missing Positive Number	718	Maximum Length of Repeated Subarray	862	Shortest Subarray with Sum at Least K

Figure 2: **Problem Distribution by Difficulty Level. The table displays the LeetCode[9] problem IDs selected for evaluation, grouped by difficulty: 10 Easy, 10 Medium, and 10 Hard. These problems were chosen from the EffiBench dataset to ensure an even spread of complexity and to analyze the efficiency of solutions across varying levels of algorithmic challenge.**

4.3 Solution Sources

For each problem, we compared solutions from four sources: a human crafted solution and three LLM generated solutions. The human solution for each problem was the **canonical, optimized code** provided (or verified) by the **EffiBench** dataset, typically implementing binary search in its most efficient form. The LLM generated solutions were obtained from **ChatGPT**, **Gemini**, and **Claude** — representing a range of advanced, contemporary LLMs. Each model was prompted with the full problem description (as presented on LeetCode), along with a **consistent prompt template** from the **EffiBench** repository. This prompt format was applied uniformly across all models to avoid bias and included any specific input output format requirements. The LLMs generated code in response to these prompts, which we then tested for **correctness**.

4.4 Ensuring Correctness

We first verified that each LLM generated solution produced correct outputs on a set of 10 test cases per problem (the same test cases used to validate human solutions). If a model's solution was incorrect (i.e., it failed one or more test cases), it was either omitted from the efficiency comparisons or corrected if the fix was straightforward. (In our experiments, a few LLM outputs initially failed certain edge case tests; for example, in one problem involving matrix searches, some LLM solutions did not handle an **empty array** condition properly. These instances were documented, and only the **correct solutions** were included in the benchmarking phase to ensure fairness in measuring efficiency.) After completing this validation step, we proceeded with performance benchmarking on the **verified correct solutions**.

4.5 Performance Measurement

Each solution was executed in an identical environment (with the same hardware and software configuration) to ensure consistent measurement of runtime and memory usage. **Execution time** was measured in seconds using high resolution timers, while **memory usage** was tracked in megabytes (MB) using system profiling tools. We recorded both the **average memory** consumed during execution and the **peak memory footprint** reached at any point. To improve accuracy, each solution was executed multiple times, and the **median execution time** was used to reduce the impact of fluctuations caused by transient system load. Memory measurements were taken throughout the execution period to accurately capture the true peak usage. All metrics were systematically logged for subsequent analysis.

4.6 Data Analysis

We aggregated the results for the 30 problems and visualized them to facilitate comparison across solutions. Our analysis considered each metric individually — **execution time**, **memory usage**, and the combined **Efficiency Score**. The results were plotted with problem indices along the x-axis (grouped by **difficulty level** for clarity) and the corresponding metric values along the y-axis. Since execution times varied by several orders of magnitude across different problems and models, we applied a **logarithmic scale** to both the time and efficiency score plots to make variations more visible. To further aid interpretation, the difficulty levels — **easy**, **medium**, and **hard** — were highlighted using background shading in the charts, colored green, yellow, and red respectively. This made it easier to spot trends that correlate with the complexity of the problem.

5 Prompting Strategy & Input Standardization

To ensure consistency and fairness in evaluating different Large Language Models (LLMs) such as ChatGPT, Gemini, and Claude, we designed a standardized prompt format. This prompt was given to each model before every task. The goal was to eliminate prompt ambiguity, enforce separation between code and test cases, and encourage valid, executable output from each model.

Prompt Template Given to Each LLM

Each model was instructed using the following standardized prompt: Below is a full example demonstrating how the LLMs received the prompt

````
Please based on the task description write Solution to pass the provided test cases.
You must follow the following rules:
First, the code should be in ```python\n[Code]\n``` block.
Second, You should not add the provided test cases into your ```\npython[Code]\n``` block.
Third, You are not need to write the test cases, we will provide the test cases for you.
Finally, You should make sure that the provided test cases can pass your solution.

Here is a example:
Example:
# Task description
```python
Given an array of integers, return indices of the two numbers such that they add up to a specific target.
You may assume that each input would have exactly one solution, and you may not use the same element twice.
Example:
Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

# Test cases
```python
solution = Solution()
assert solution.twoSum([2, 7, 11, 15], 9) == [[0, 1]]
```

# Code
```python
from typing import *
import random

class Solution:
    def twoSum(self, nums, target):
        hashtable = dict()
        for i, num in enumerate(nums):
            if target - num in hashtable:
                return [hashtable[target - num], i]
            hashtable[nums[i]] = i
        return []
```
````

After this, we provide the problem description.

Why This Prompting Strategy Matters

- All models received the same input format, ensuring a fair comparison.
- By keeping test cases separate, we avoided giving models any unintended hints.
- This setup mirrors how real users present problems just a description and a request for working code.

This standardized prompting protocol was used across all 30 benchmark problems, ensuring consistent and fair evaluation. It helped isolate model limitations—such as incorrect logic or edge case failures—from issues caused by unclear input formatting.

6 Results

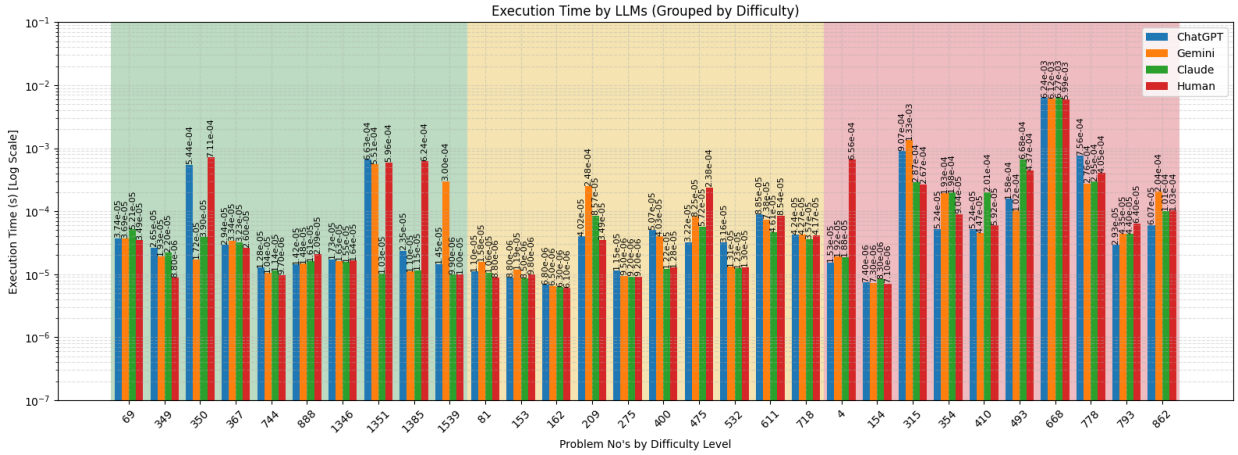

Figure 3: **Execution Time by LLMs (Grouped by Difficulty). Each group of bars corresponds to one problem (x-axis ordered by increasing difficulty from Easy to Hard). Bar height (log scale) indicates execution time in seconds for ChatGPT (blue), Gemini (orange), Claude (green), and Human (red) solutions. Lower is better (faster). Gemini achieves the shortest execution time on most problems, with human and ChatGPT generally close behind, and Claude slightly slower in some cases.**

As shown in Figure 3, Gemini consistently delivers the fastest execution times for most problems, often outperforming even human written solutions. ChatGPT and human coded versions generally perform similarly many of their bars are nearly the same height while Claude's execution times tend to be a bit slower in some cases. The performance gap is minor for easy problems, where all methods run in microseconds, but becomes more noticeable on the harder ones. For some challenging cases, Gemini's advantage is significant, with execution times that are an order of magnitude faster than their counterparts.

Overall, the results point to Gemini's clear edge in speed, with ChatGPT and human code closely following, and Claude slightly behind. This suggests Gemini may have learned more optimized patterns for binary search, likely thanks to a stronger focus on algorithmic training data. What's more, Gemini's consistently strong performance across all difficulty levels shows its reliability as a coding assistant. Claude's slower speeds, by contrast, may result from a tendency to produce more complex or layered logic, leading to extra computational steps. Still, all models stayed within acceptable execution times, making them viable tools even for complex problem solving tasks.

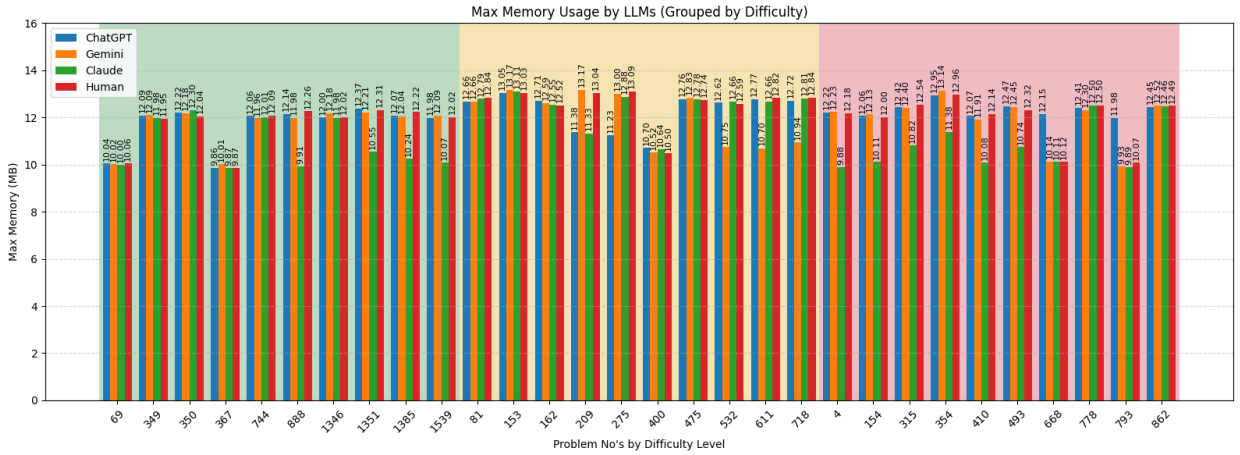

Figure 4: **Peak Memory Usage by LLMs (Grouped by Difficulty). Bar height indicates the peak memory (MB) consumed by each solution on each problem. All methods use similar amounts of memory (around 10–13 MB). Human solutions (red) are slightly lower in some cases, while LLM generated solutions (blue, orange, green) are marginally higher, with Claude sometimes using the most memory.**

Figure 4 shows that memory usage among ChatGPT, Gemini, Claude, and human solutions remains largely consistent. Most implementations reach a peak memory usage between 10 and 13 MB, as shown by the similar bar heights across all problems. None of the solutions demonstrate extreme outliers in terms of memory footprint. Human written code tends to use slightly less memory in certain cases, likely due to the use of minimal, purpose driven logic that avoids extra data manipulation. In contrast, LLM generated code, particularly from Claude, occasionally consumes more memory. This is possibly a result of generating additional structures or including extra layers of abstraction within the code.

Despite these slight differences, all models operate within a narrow and reasonable memory range, suggesting strong baseline efficiency across sources. The consistency in memory usage across easy, medium, and hard problems indicates that the complexity of the problem does not drastically affect memory consumption in binary search tasks. This trend also reinforces the idea that the memory profile of a solution depends more on the coding patterns used rather than the algorithmic difficulty itself. Overall, the results confirm that both human and LLM generated solutions are practically efficient in terms of memory, with only minor variations influenced by implementation style.

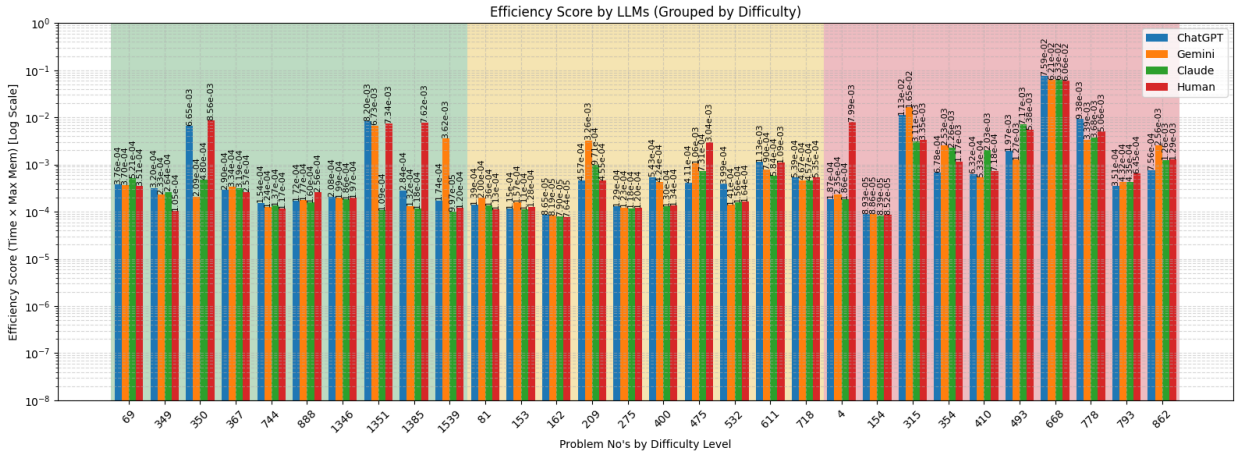

Figure 5: **Efficiency Score by LLMs (Grouped by Difficulty). The Efficiency Score is defined as Execution Time × Peak Memory (lower is better). Bars represent this combined metric for each solution per problem (log scale on y-axis). Gemini's scores (orange) are the lowest in most cases, indicating best overall efficiency. Human (red) and ChatGPT (blue) solutions also achieve low scores close to Gemini's, while Claude (green) tends to have higher scores, denoting slightly worse efficiency.**

In Figure 5, which plots the combined efficiency metric, ** Gemini consistently produces the best Efficiency Scores** across the board – its bars are minimal for most problems, reflecting its fast speed coupled with standard memory use. Human solutions and ChatGPT often have comparably low efficiency scores as well, frequently only slightly above Gemini's. Notably, in a few cases (particularly among medium and hard problems), the human solution achieves the best score, indicating an instance where careful human optimization paid off. ChatGPT's scores are usually very close to the human's, showing that it produces relatively efficient code. **Claude's efficiency scores are consistently the highest (worst) among the four** in many problems, aligning with the earlier observations that Claude is a bit slower and sometimes more memory hungry. Still, all solutions' scores are on the order of 10^{-4} to 10^{-3} for most tasks, so they are all reasonably efficient. The gaps widen on a few specific hard problems where one solution's inefficiency spikes the score (seen as isolated taller bars). In summary, the Efficiency Score comparison emphasizes Gemini's overall lead in efficiency, while also showing that ChatGPT and well written human code are not far behind, and that Claude, despite being slightly less efficient, remains within the same order of magnitude range.

7 Discussion

The benchmarking results provide several insights into the comparative performance of human versus LLM generated code for the binary search problems:

LLM vs. Human Performance: Our experiments demonstrate that state of the art LLMs can **match or even exceed human programmers in code efficiency** for a well defined algorithm like binary search. In particular, Google's Gemini model stood out by producing solutions that run extremely quickly – in fact, Gemini's solutions were the fastest in the majority of cases, sometimes by a noticeable margin. This suggests that Gemini's training or inference process may favor more optimized code patterns (perhaps it has learned efficient idioms for binary search). Human written solutions, which we expected to be near optimal, indeed performed strongly and even outperformed the AI models on a few problems. This typically occurred in more complex scenarios where a human developer might apply a clever optimization or handle an edge case more efficiently than the generic approaches the LLMs learned. OpenAI's ChatGPT was generally very competitive with the human solutions on both time and memory metrics, indicating that it usually produces quite polished code for binary search. Anthropic's Claude, while still performing well, was slightly less efficient for example, its code sometimes had extra steps or used more memory (as reflected in the marginally higher memory usage and efficiency scores). These differences, however, were not extreme; all approaches achieved fast runtimes and modest memory usage suitable for the problem sizes.

Impact of Problem Difficulty: Problem difficulty influenced the spread of performance outcomes. For **easier problems**, all solutions both human written and LLM generated executed almost instantaneously and consumed minimal memory, leaving little opportunity for one approach to significantly outperform the others. As the problems increased in difficulty (i.e., **medium** and **hard**), often involving more complex input scenarios or edge cases, we observed greater variability in results. In some hard problems, inefficiencies became apparent — for instance, an LLM might not handle a special case as efficiently, resulting in extra loop iterations or additional memory overhead. In one such hard problem (identifying the k weakest rows in a matrix), the human solution applied an **optimal strategy** that all LLMs initially missed. As a result, the LLM generated solutions either failed certain test cases or took slightly longer to execute. This highlights that **LLMs, despite their capabilities, can still struggle with niche or edge case scenarios**, and that problem difficulty can amplify the impact of suboptimal logic. Nonetheless, for most hard problems in our set, at least one LLM — often **Gemini** — produced a solution with efficiency comparable to the human written one, showcasing the ongoing advancement in AI generated code.

Code Characteristics and Efficiency: Upon manual inspection of several solutions, we identified patterns that help explain the observed differences in efficiency. The **human written solutions** were typically succinct and purpose built often implementing binary search directly, with minimal extra code. In contrast, the **LLM generated solutions** occasionally included superfluous operations (e.g., converting data types or invoking library functions where a more direct approach would suffice), which introduced slight overhead.

From a memory standpoint, all solutions had broadly similar requirements, as the tasks themselves were not inherently memory intensive. However, when an LLM produced a more

convoluted implementation — such as creating unnecessary data copies or relying on non optimal data structures — it tended to consume more memory. The slightly higher memory usage observed in some of **Claude's** solutions may be attributed to these patterns.

It is also important to note that none of the LLMs were explicitly instructed to optimize for performance; they were prompted solely to solve the problem correctly. The fact that their performance came close to optimal suggests that modern LLMs often generate reasonably efficient code **by default** for well defined algorithms. Still, the small performance gaps we observed indicate that there is room to further **fine tune or guide LLMs for optimization**.

Implications: These findings have meaningful implications for both users and developers of code generation AI. For practitioners integrating **LLMs** into their development workflows, our results are encouraging: an LLM such as **ChatGPT** or **Gemini** can not only produce correct solutions but also deliver efficiency that, in many cases, is comparable to that of a human expert. However, developers should remain vigilant when handling edge cases and should evaluate the performance of **AI generated code**, particularly on more complex or non trivial problems.

From a research perspective, our study reinforces the importance of benchmarks like **EffiBench** that extend beyond correctness and include performance based evaluation. As LLMs continue to evolve, **efficiency could become a key differentiator** — users may increasingly favor models that not only solve problems accurately but also generate code that executes faster and consumes less memory. This trend could drive the development of new training objectives or **fine tuning strategies** that explicitly incorporate runtime performance as part of model optimization.

Furthermore, our results suggest the potential for **hybrid approaches**, where an LLM's initial output is postprocessed or refined using external tools to combine the strengths of both AI generated correctness and human level performance tuning — achieving the best of both worlds.

8 Problems Faced

During the evaluation of LLM generated solutions versus canonical implementations, we encountered several important issues that impacted the reliability and interpretation of our benchmarking results:

- **Inconsistent Test Case Outcomes:** LLM-generated solutions occasionally failed test cases that the human-written implementations passed. These failures were most common in edge cases such as empty arrays, large values, or repeated elements.
- **Mismatch Between Local and Online Results:** In problems like LeetCode 1337 – *The K Weakest Rows in a Matrix*, models like ChatGPT, Gemini, and Claude passed all of LeetCode's official test cases but failed in our local test environments. This discrepancy highlighted possible gaps in test coverage and differences in validation mechanisms.

- **Environment-Specific Behavior:** We found that small differences in runtime environments — such as Python versions, library behavior, or execution settings — sometimes led to divergent outputs. This made it difficult to reproduce results consistently across systems.

- **Overfitting to Prompt Format:** Some LLM responses appeared to rely heavily on the structure of the example input rather than generalizing to broader scenarios. While these solutions passed initial tests, they often failed when faced with variations, indicating overfitting to the given prompt.

- **Debugging Complexity:** Generated code from LLMs was occasionally cluttered with redundant logic or unclear structure. This made it harder to understand, debug, or trace the root cause of errors—especially in longer, multi-step problems.

- **Importance of Manual Verification:** These challenges underscored the need for thorough, independent testing. Just because code passes online platform checks doesn't mean it's logically sound. Manual validation and unit tests remain essential to ensure correctness and robustness.

9 Conclusion

We presented a comparative study of **human written** and **LLM generated** solutions for binary search problems, using a rigorous benchmarking methodology. Our evaluation across 30 problems shows that **LLM generated code can achieve performance on par with human written code**, and in the case of Google's *Gemini*, often surpasses it in terms of execution speed and overall efficiency. Human solutions continued to perform strongly in some cases even outperforming all LLMs — highlighting the value of human insight and manual optimization. OpenAI's *ChatGPT* offered a balanced trade off between speed and memory usage, making it a reliable overall performer, though not the absolute fastest. Anthropic's *Claude* produced correct and generally solid results but tended to trail slightly behind the others in efficiency, particularly in terms of memory usage.

These findings emphasize that as **LLMs become more integrated into software development**, evaluation criteria must evolve to include **efficiency alongside correctness**. Our use of the **EffiBench** framework demonstrates a structured way to assess both dimensions. Looking ahead, we plan to broaden our analysis to include a wider variety of algorithms beyond binary search such as sorting, graph traversal, and dynamic programming and to incorporate more recent models like *GPT-4* and other emerging systems. We also aim to explore techniques for improving LLM efficiency, such as optimization aware prompts or hybrid approaches that combine LLM generated outputs with **static analysis tools** to identify and correct inefficiencies.

Ultimately, we envision a future where **human AI collaboration** yields solutions that are not only correct and maintainable, but also optimized for performance merging the best of both human expertise and AI automation. Additionally, all project code, benchmarking scripts, and visualizations have been made publicly available on our GitHub repository[11].

References

[1] OpenAI, *ChatGPT*, https://openai.com/chatgpt, Accessed: 2025-05-04, 2023.

[2] Google DeepMind, *Gemini AI*, https://deepmind.google/technologies/gemini, Accessed: 2025-05-04, 2023.

[3] Anthropic, *Claude AI*, https://www.anthropic.com/index/claude, Accessed: 2025-05-04, 2023.

[4] J. Klein, E. Liu, W. Zheng, Y. Chen, and Q. Gu, *EffiBench: Benchmarking Code Generation on Efficiency*, https://github.com/effibench/effibench, Accessed: 2025-05-04, 2023.

[5] Y. Li, A. Vemula, Z. Cheng, *et al.*, "Competition-Level Code Generation with AlphaCode," *arXiv preprint arXiv:2203.07814*, 2022.

[6] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] M. Chen, J. Tworek, H. Jun, *et al.*, *Evaluating Large Language Models Trained on Code*, https://github.com/openai/human-eval, Accessed: 2025-05-04, 2021.

[8] J. Austin, A. Odena, M. Nye, *et al.*, *Program Synthesis with Large Language Models*, https://github.com/google-research/google-research/tree/master/mbpp, Accessed: 2025-05-04, 2021.

[9] LeetCode, *LeetCode - Online Coding Platform*, https://leetcode.com/problemset/all/, Accessed: 2025-05-05, 2025.

[10] B. Rozière, G. Izacard, O. Fernandes, *et al.*, "Code Llama: Open Foundation Models for Code," *arXiv preprint arXiv:2308.12950*, 2023.

[11] B. Visesh, *Analyzing binary search problem solutions comparing human vs llm-generated code using effibench*, https://github.com/viseshb/Analyzing_Binary_Search_Problem_Solutions_Comparing_Human_vs_LLM_generated_Code_Using_EffiBench.