Deposited via The University of Leeds.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/237873/

Version: Preprint

Preprint:
La Malfa, E., Weinhuber, C., Torre, O. et al. (2025) Code Simulation as a Proxy for High-order Tasks in Large Language Models. [Preprint]

Code Simulation as a Proxy for High-order Tasks in Large Language Models

*Emanuele La Malfa [1 2] Christoph Weinhuber [1] Orazio Torre [3] Fangru Lin [4] X. Angelo Huang [5]
Samuele Marro [6] Anthony Cohn [2 7] Nigel Shadbolt [1 2] Michael Wooldridge [1]

Project Website

Abstract

Many reasoning, planning, and problem-solving tasks share an intrinsic algorithmic nature: correctly simulating each step is a sufficient condition to solve them correctly. We collect pairs of naturalistic and synthetic reasoning tasks to assess the capabilities of Large Language Models (LLM). While naturalistic tasks often require careful human handcrafting, we show that synthetic data is, in many cases, a good proxy that is much easier to collect at scale. We leverage common constructs in programming as the counterpart of the building blocks of naturalistic reasoning tasks, such as straight-line programs, code that contains critical paths, and approximate and redundant instructions. We further assess the capabilities of LLMs on sorting problems and repeated operations via sorting algorithms and nested loops. Our synthetic datasets further reveal that while the most powerful LLMs exhibit relatively strong execution capabilities, the process is fragile: it is negatively affected by memorisation and seems to rely heavily on pattern recognition. Our contribution builds upon synthetically testing the reasoning capabilities of LLMs as a scalable complement to handcrafted human-annotated problems.

1. Introduction

A major area of interest at the time of writing is understanding the capabilities of Large Language Models (LLMs) beyond the tasks of language understanding and generation.

Many benchmarks, including Theory of Mind (Rabinowitz et al., 2018), planning (Hao et al., 2023; Ouyang et al., 2022), and high-order reasoning (Webb et al., 2023), necessitate turning a prompt, expressed in natural language, into a procedure that must then be carried out faultlessly. Consider a problem where two agents interact and exchange goods, as in Figure 1. An LLM prompted to compute the number of goods at the end of an iteration should be able to sum and assign variables correctly. Such a naturalistic problem has an intrinsic algorithmic nature as code (Figure 1, centre). This work centres on this analogy and turns it into an experimental pipeline to assess the capabilities of LLMs on reasoning tasks via equivalent code simulation.

Our preliminary experiments, which we will expand on in the next sections and pair with others in literature (Lin et al., 2024), evidence a strong performance correlation between naturalistic and synthetic tasks. Beyond the example above, the code simulation capabilities of LLMs are an important object of study for at least two complementary reasons. First, LLMs have shown planning capabilities (Ouyang et al., 2022), which requires reasoning step-by-step and recursively dividing a problem into its elementary components. However, such compositional reasoning performance is highly fragile to variations (Turpin et al., 2024). With highly structured input, code simulation can shed insight into the underlying failure modes of the LLM without the confounding factors of natural language tasks. Second, code simulation requires an LLM to turn instructions formulated in code and natural language into a procedure the model must solve correctly, and thus answers the question of whether LLMs can serve as *digital* computational models (Jojic et al., 2023).

This paper shows that code simulation is a scalable proxy for assessing some core reasoning capabilities of LLMs; this is in contrast with generating naturalistic datasets, which is resource- and time-expensive. While not all the reasoning tasks can be equivalently formulated as code, we showcase the flexibility of our framework by pairing five non-trivial naturalistic tasks with their coding counterpart and show that the performances of GPT-4, GPT-4o (OpenAI,

---

[1]Department of Computer Science, University of Oxford [2]The Alan Turing Institute [3]University of Salerno [4]Faculty of Linguistics, Philology, and Phonetics, University of Oxford [5]Department of Computer Science, ETH Zurich [6]Department of Engineering, University of Oxford [7]Faculty of Engineering and Physical Sciences, University of Leeds. *First author. Correspondence to: Emanuele La Malfa <emanuele.lamalfa@cs.ox.ac.uk>.
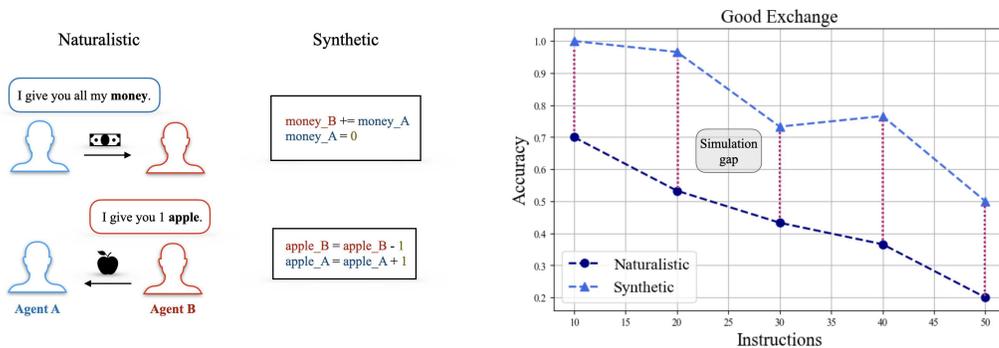
Figure 1. Left: an example of the naturalistic vs. synthetic good exchange settings. The former describes, in natural language, two agents who exchange goods; the latter is an equivalent formulation in code. While GPT-3.5-Turbo performs better on the synthetic task (a "simulation gap"), performance in the synthetic and naturalistic tasks is strongly correlated with respect to the control variable, i.e., the number of operations/exchanges. We conduct experiments on 30 samples per instruction class with $\{10, 20, 30, 40, 50\}$ interactions/lines of code.

2023), and Llama3.1-405B (Grattafiori et al., 2024) correlate with them. As expected, our code simulation approach enables us to identify failure cases for step-by-step execution, namely memorisation and "lazy" pattern recognition. For these cases, we develop a minimal extension of Chain of Thought (Wei et al., 2022b) that mitigates these issues and allows benchmarking in the presence of memorisation. To further lay the groundwork for code simulation as a proxy of naturalistic reasoning tasks, we conduct extensive experiments on pure code simulation and simple programming constructs on a variety of models, including GPT-3.5-Turbo, GPT-4, Jurassic2-Ultra, Llama-2-70B, Llama-3-70B and CodeLLaMA-34b-Instruct. The code to replicate the experiments in the main paper is available here, while the code to replicate the experiments on pure code simulation is available here. Click here to access the project's website.

2. Related Work

LLMs to understand, generate, and improve code have been mainly developed to produce debugging information without invoking a compiler/interpreter (Hou et al., 2023; Santos et al., 2023; Vaithilingam et al., 2022; Widjojo & Treude, 2023; Zan et al., 2023). Code generation and simulation require some degree of compositionality (McCoy et al., 2023a), i.e., the result of complex expressions can be determined by their constituents and the rules used to combine them. Recent works explored compositionality in terms of simple mathematical operations that LLMs can execute (Frieder et al., 2023; Yang et al., 2023; Yuan et al., 2023), and revealed how the most potent models do not achieve that (McCoy et al., 2023b; West et al., 2023). Our work further explores the tension between memorisation and performance on complex tasks (Berglund et al., 2023; Eldan & Russinovich, 2023; Yang et al., 2023), with results that illustrate how the former is at tension with the size of a model, the so-called "inverse scaling law" (Biderman et al.,

2023).

Before the breakthrough of closed-source LLMs (La Malfa et al., 2023), a seminal work tested LLMs on code simulation, showing how keeping track of the variables improves their capabilities (Nye et al., 2021). Successive works explored LLMs and code simulation (Chen et al., 2024b; Tufano et al., 2023; Zhou et al., 2023), particularly in (Liu et al., 2023), where the authors fine-tune Transformer-based models to output the program trace of a code snippet. The code simulation capabilities of LLMs have been explored in several recent works: the first that identified the problem as relevant for LLMs is (La Malfa et al., 2024), of which this work is an extension. Other works successively extended this idea (Lyu et al., 2024). Recent developments in this field go under the name of "code reasoning", as a model's ability to predict a variable's state at runtime (Chen et al., 2024a), the output of a statement/function (Gu et al., 2024; Liu et al., 2024), or their capability to handle recursion (Zhang et al., 2024). At the architectural level, several works studied Transformers and attention-based models regarding the operations and programming languages they interpret and execute (Weiss et al., 2021) and their recursive code simulation capabilities (Zhang et al., 2023). On a broader perspective, past work investigated the Turing-completeness of LLMs (Giannou et al., 2023; Pérez et al., 2021; Schuurmans, 2023; Wei et al., 2022a), and their ability to follow instructions (Ouyang et al., 2022) and policies expressed as code (Liang et al., 2023).

3. Methodology

Formally, a Language Model is defined as a function that predicts the next token (out of a finite vocabulary) conditioned on the sequence of previously fed/generated tokens, namely $\psi : V^* \to \mathbb{P}(V)$. In our setting, a problem is specified as a tuple (x, p), where x instructs the model to solve

a problem p expressed either as code or as an equivalent naturalistic task. Both settings express the same question, with the ground truth label obtained by running the code with an interpreter/compiler Ω and compared, for correctness, against the model's answer. We select Python 3 as our programming language for coding problems as it represents the language of reference in most LLMs, such as Code-LLaMA (Rozière et al., 2023), and is among the most covered languages in open-source LLMs (Gao et al., 2020; Scao et al., 2022) and network Q&A platforms such as Stack Exchange. Furthermore, its syntax, for simple programs, resembles that of pseudo-code, thus abstracting from complex programming constructs.

Metrics such as the accuracy, expressed as $\frac{1}{N}\sum_{i=1}^{N}\mathbb{1}[\psi(y_i|x, p_i) = \Omega(p_i)]$, inform us as to the capabilities of a model. Another metric we consider is how *incorrect* a model is in its prediction, i.e., the average distance between the correct result and the model output, namely $\frac{1}{N}\sum_{i=1}^{N}|\psi(y_i|x, p_i) - \Omega(p_i)|$. On tasks that require simulating multiple independent instructions or returning multiple correct predictions, we measure the prediction error as the Levenshtein distance between the prediction and the ground truth (as tuples), namely $\frac{1}{N}\sum_{i=1}^{N}|\psi(y_i|x, p_i) \cap \Omega(p_i)|$. We also analysed the LLMs' responses to identify the most common reasons for failure in code simulation (and, by extension, task execution).

In summary, our methodology aims to reveal whether code simulation is a good proxy for naturalistic problems; we design reasoning tasks that can be turned into equivalent code, from operations such as addition and assignment to more complex constructs such as nested loops and sorting algorithms. We introduce five naturalistic and synthetic benchmarks: (I) **Straight line code simulation and Good exchange**, which tests the ability of an LLM to solve simple, sequential operations consistently; (II) **Critical path and Critical good exchange**, i.e., problems where only a portion is relevant to output the correct answer; (III) **Parallel path and Clique exchange**, which tests the capacity of an LLM to correctly perform multiple independent operations; (IV) **Nested loops and Recurring calculation**, that connects the computational complexity of a problem with the capabilities performing recurring reasoning operations, and (V) **Sorting and Ranking objects**, which tests sorting as a proxy of ranking objects with varying features.

3.1. Benchmarks

In this section, we introduce the rationale behind each dataset. In particular, drawing from the literature in cognitive psychology (Sweller & Chandler, 1991), we focus on the reasoning capabilities that can be captured by purely synthetic tasks such as code. Our framework benchmarks a model with pairs of synthetic and naturalistic prompts. If the

correlation between the two settings holds strong, one can switch to purely synthetic benchmarks, for which generating new samples is much cheaper and scalable.

I. Straight line and Good exchange. Straight line code simulation can reveal whether a model processes sequential instructions faultlessly and leverages the principle of compositionality (Dziri et al., 2023). In this sense, Straight line can model scenarios that involve keeping track of objects whose state repeatedly changes over time (Kim & Schuster, 2023), as well as planning and Theory of Mind (Kim et al., 2023; Lin et al., 2024). We model a naturalistic task as a Multi-Agent System, where different actors possess and exchange goods. Then, we ask a model to compute the number of goods (the value of a variable) an agent has after the exchanges. We control the complexity of each problem by varying the number of operations/exchanges; other control variables can be introduced to study other capabilities, such as the number of agents and goods exchanged. The synthetic task that acts as a proxy of the naturalistic is a plain sequence of variable declarations, followed by instructions that modify the variables' state with sum and subtraction. Figure 2 (left) illustrates an example of the Straight line and Good exchange task. In Appendix B.1, we evaluate a richer set of operations, such as logical and/or, that is hard to model in naturalistic settings but can serve as a basis for increasing synthetic benchmarks. In addition to the Straight line task, we revisit the entity-tracking benchmark for LLMs (Kim & Schuster, 2023) and pair it with a pure coding task where each object state's change is turned into a synthetic instruction with a variable assignment.

II. Critical path and Critical good exchange. Straight line and Good exchange do not assess a model's capabilities to distinguish relevant states from those that can be ignored. This setting is ubiquitous in machine learning and has been recently addressed in high-order tasks such as Theory of Mind (Huang et al., 2024). We introduce a variation of the Straight line and Good exchange where only a fraction of the problem is sufficient to derive the correct answer. In this setting, agents exchange several goods. The critical path (i.e., the portion sufficient to solve the problem) is intertwined with other instructions that serve the role of noise or "extraneous load" in Cognitive Load Theory (Sweller & Chandler, 1991). We synthetically model this setting with a well-known concept in algorithmic theory, i.e., that of a critical path, which is the stretch of dependent operations in a program. Figure 2 (left) illustrates an example of the Critical path and Critical good exchange task. Each good exchange is modelled as a variable addition, subtraction or assignment to resemble naturalistic operations. In Appendix B.3, we evaluate a richer set of operations, such as logical and/or, that can serve as a basis to test increasingly complex naturalistic settings.
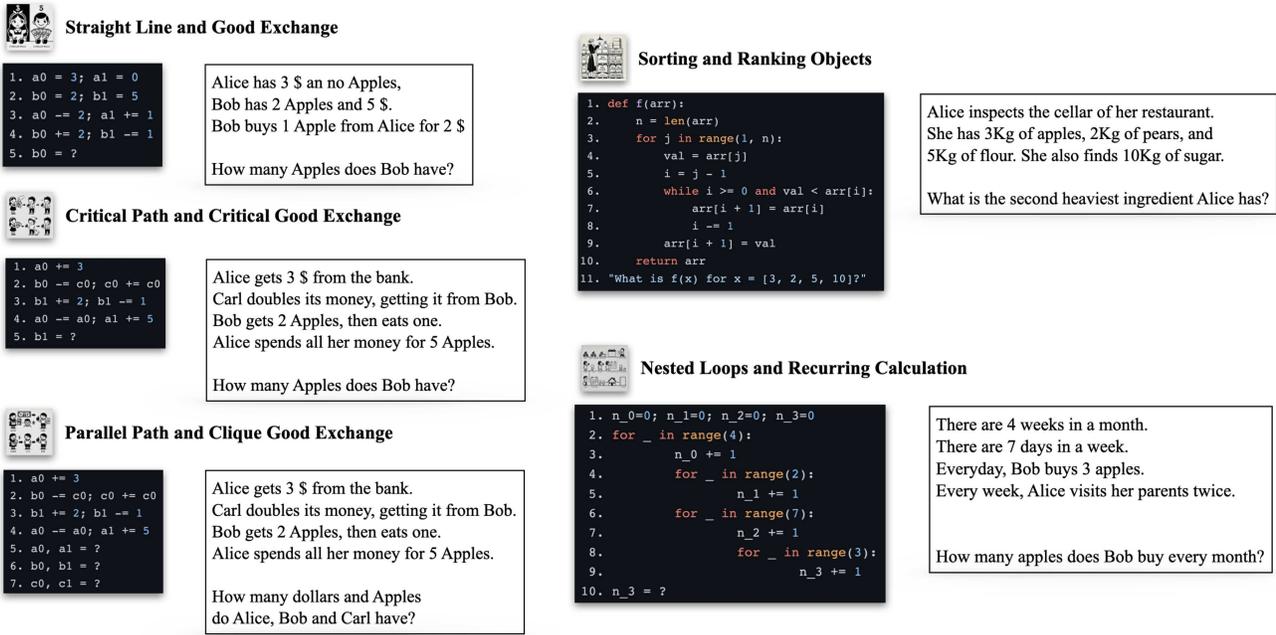
Straight Line and Good Exchange

```
1. a0 = 3; a1 = 0
2. b0 = 2; b1 = 5
3. a0 -= 2; a1 += 1
4. b0 += 2; b1 -= 1
5. b0 = ?
```

Alice has 3 $ an no Apples,
Bob has 2 Apples and 5 $.
Bob buys 1 Apple from Alice for 2 $

How many Apples does Bob have?

Critical Path and Critical Good Exchange

```
1. a0 += 3
2. b0 -= c0; c0 += c0
3. b1 += 2; b1 -= 1
4. a0 -= a0; a1 += 5
5. b1 = ?
```

Alice gets 3 $ from the bank.
Carl doubles its money, getting it from Bob.
Bob gets 2 Apples, then eats one.
Alice spends all her money for 5 Apples.

How many Apples does Bob have?

Parallel Path and Clique Good Exchange

```
1. a0 += 3
2. b0 -= c0; c0 += c0
3. b1 += 2; b1 -= 1
4. a0 -= a0; a1 += 5
5. a0, a1 = ?
6. b0, b1 = ?
7. c0, c1 = ?
```

Alice gets 3 $ from the bank.
Carl doubles its money, getting it from Bob.
Bob gets 2 Apples, then eats one.
Alice spends all her money for 5 Apples.

How many dollars and Apples
do Alice, Bob and Carl have?

Sorting and Ranking Objects

```
1.  def f(arr):
2.     n = len(arr)
3.     for j in range(1, n):
4.        val = arr[j]
5.        i = j - 1
6.        while i >= 0 and val < arr[i]:
7.           arr[i + 1] = arr[i]
8.           i -= 1
9.        arr[i + 1] = val
10.    return arr
11. "What is f(x) for x = [3, 2, 5, 10]?"
```

Alice inspects the cellar of her restaurant.
She has 3Kg of apples, 2Kg of pears, and
5Kg of flour. She also finds 10Kg of sugar.

What is the second heaviest ingredient Alice has?

Nested Loops and Recurring Calculation

```
1.  n_0=0; n_1=0; n_2=0; n_3=0
2.  for _ in range(4):
3.        n_0 += 1
4.        for _ in range(2):
5.           n_1 += 1
6.        for _ in range(7):
7.           n_2 += 1
8.           for _ in range(3):
9.              n_3 += 1
10. n_3 = ?
```

There are 4 weeks in a month.
There are 7 days in a week.
Everyday, Bob buys 3 apples.
Every week, Alice visits her parents twice.

How many apples does Bob buy every month?

Figure 2. On the left, examples of Straight line and Good exchange tasks (top), Critical path and Critical good exchange (middle), and Parallel path and Clique good exchange (bottom). On the right, examples of synthetic and naturalistic Sorting and Ranking objects (top) and Nested loops and Recurring calculation (bottom).

III. Parallel paths and Clique good exchange. Another aspect of reasoning that complements the "extraneous cognitive load" is that of "intrinsic load" as the complexity of processing new information (Sweller & Chandler, 1991). One can make a Straight line and a Good exchange problem more complex by requiring one to keep track of multiple variables/objects simultaneously. A model has to store, access, and modify all the variables consistently to return the correct result. The Clique good exchange is a Multi-Agent Scenario that, paired with the Parallel paths, tests this setting. Figure 2 (left) illustrates an example of such a setting. Similarly to the previous benchmarks, in Appendix B.4, we evaluate a richer set of operations, such as logical and/or, that are hard to model in naturalistic settings but can serve as the basis of increasingly complex synthetic benchmarks.

IV. Nested loops and Recurring calculation. A set of problems often occurring in literature and human settings is recurring calculations, often in the form of math problems (Cobbe et al., 2021). Consider this example: "There are five working days a week. Every day of the week, I buy an apple. How many apples do I buy in 2 months?". This and similar problems stress the capabilities of a model to connect the relevant pieces of information and compute recurring calculations, which can be further made more difficult by injecting noise, as per the previous example. We design the Recurring calculation to test such capabilities. In code, nested loops with different indentations serve as

a faithful proxy of naturalistic tasks with recurring calculations, as illustrated in Figure 2 (right). On top of the experiments we conduct on pairs of synthetic and naturalistic prompts, interesting results emerge by solely analysing the capabilities of different models (such as Llama and GPT) on purely synthetic Nested loops, as reported in Appendix B.5.

V. Sorting and Ranking objects. Ranking is a ubiquitous problem in both synthetic and naturalistic settings. To rank correctly, one must sort a set of objects by an attribute. The Ranking objects problem prompts a model with a problem where an agent has to return the kth-heaviest/lightest object in a group. As Figure 2 (right) illustrates, the synthetic proxy problem consists of sorting a vector of numbers with an algorithm of choice, such as Bubble Sort or Insertion Sort. This benchmark compares the reasoning capabilities of an LLM with synthetic and naturalistic sorting problems.[1] We also extensively evaluate sorting, i.e., pure code capabilities, with different algorithms of varying time- and space-complexity and show a tension between memorisation and code simulation capabilities in LLMs.

---

[1]The optimal synthetic solution for the Objects ranking problem is a min-k heap, which is more efficient yet less intuitive than sorting. We thus focus on sorting.
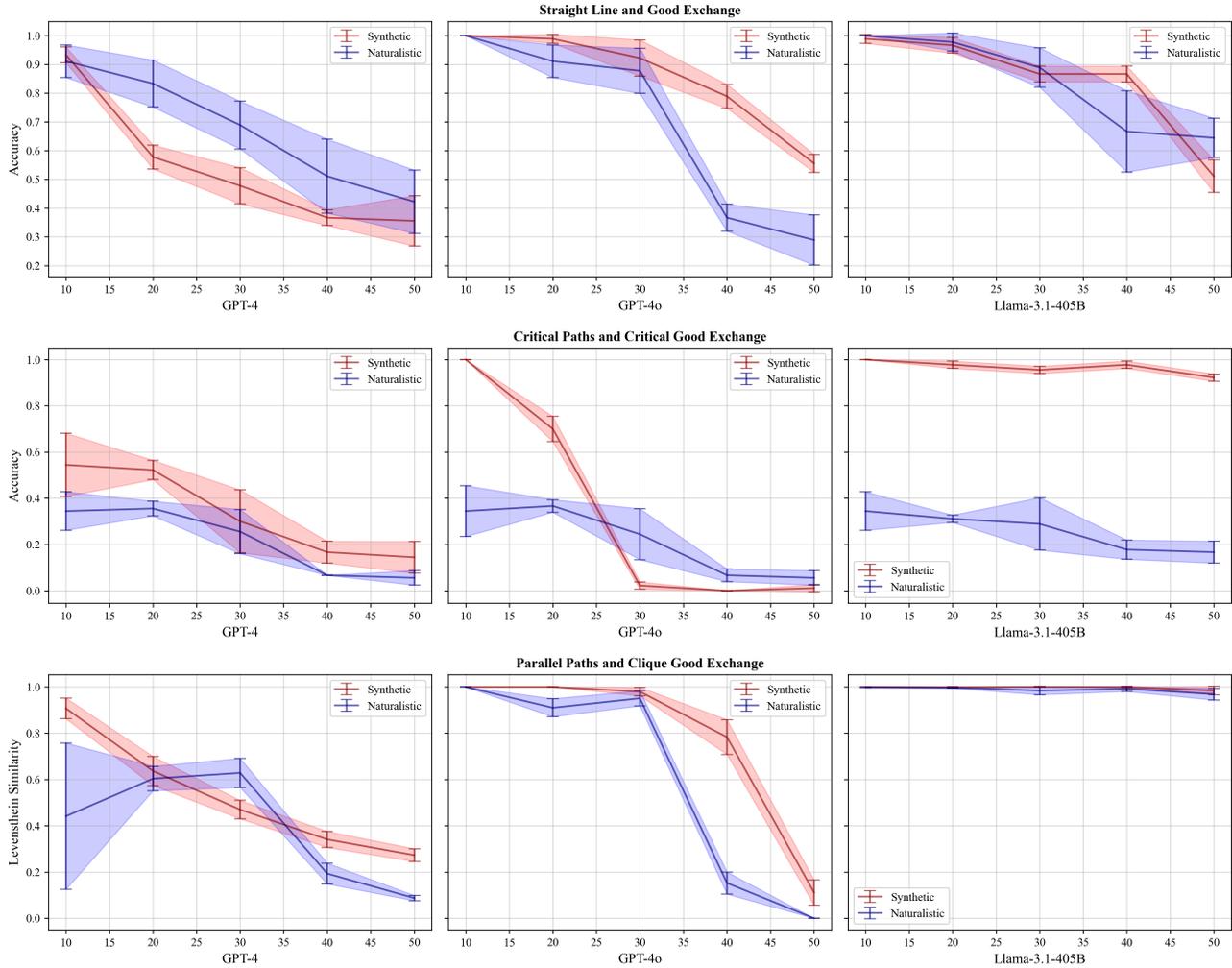
Figure 3. Top: Accuracy of different models on the Straight line and Good exchange. Middle: Accuracy on the Critical path and Critical good exchange. The critical path length is 5. Bottom: Levenshtein similarity of the ground truth and the prediction on the Parallel path and Clique exchange. The control variable for each problem is the number of operations, that spans from 10 to 50 with granularity 10.

4. Experimental Evaluation

We conducted our experiments with open- and closed-source models, i.e., GPT-4, GPT-4o and Llama-3-405B. For each benchmark, we ran three independent runs with 30 programs each to report the standard deviation over multiple trials. The control variable for the Straight line, Critical path, and Parallel path (and their naturalistic counterpart) is the number of instructions/exchanges. For Nested loops and Recurring operations, we vary the complexity of the task with the number of nested loops (i.e., the computational complexity of the algorithm to be simulated) or, equivalently, the number of nested recurring operations in the naturalistic task. For Sorting and Ranking objects, we vary the number of objects to rank. Each run consists of naturalistic or synthetic prompts: we extract the answer and compare it to the ground truth label, obtained with a compiler/interpreter, to compute the performance metrics. We evaluate each model with standard Chain of Thought (Wei et al., 2022b) (CoT). We also conduct an extensive evaluation of the synthetic capabilities of several LLMs (including GPT-3.5-Turbo, GPT-4, Llama-2, Llama-3, and Jurassic) on purely synthetic tasks, to highlight the reasons for failure when prompted to simulate code. Finally, we emphasise that we only evaluate models without access to compilers/interpreters.

4.1. Code Simulation as a Proxy of Naturalistic Tasks

As reported in Figure 3 (top), the performance of Straight line code simulation matches that of the Good exchange, proving the synthetic task a faithful proxy of the performance of the naturalistic setting. Interestingly, GPT-4 is better on the naturalistic task while GPT-4o finds the synthetic task easier, with a "gap" between synthetic and naturalistic for high-complexity problems. In Appendix A, we conducted a linguistically informed analysis of the most frequent errors GPT-4o makes in the naturalistic setting. In the same section, we report the results and an analysis of the object tracking task introduced in (Kim & Schuster, 2023), which we pair with an equivalent coding task. We show that the two are very similar in performance, except for Llama-3.1-405B. For the Critical path, Figure 3 (middle), we notice that the performance of GPT-4 strongly correlates with the synthetic and naturalistic tasks. Interestingly, GPT-4o and Llama-3.1-405B show similar trends for the naturalistic task, yet GPT-4o performs well for short programs, although its performance drops to zero as the program length increases. At the same time, Llama-3.1-405B always has better performance on the synthetic task. We conduct an informed linguistic analysis of the results and report it in Appendix A. In the Parallel path and Clique good exchange (Figure 3, bottom), all the models reveal a correlation between the naturalistic and synthetic settings. As we elaborate in Appendix A, the gap of GPT-4o is caused by

the model mishandling simple operations such as zeroing a variable or erroneously reusing the previous value of a variable.

Regarding Nested loops and Recurring operations, we notice that the performance of the naturalistic and synthetic tasks are correlated, yet the former is lower and affected by noise. We impute this behaviour to the intrinsic noise of natural language, and we confirm this with an in-depth linguistic analysis of the errors in Appendix A. While the correlation between the performances is still present, these results suggest that the naturalistic task is more challenging for LLMs. Finally, for sorting, we notice ambivalent trends that suggest that each LLM handles the two tasks differently. Results reveal that the synthetic task becomes, for GPT-4 and GPT-4o, unexpectedly more straightforward for more challenging instances, while for Llama-3.1-405B, the behaviour of naturalistic and synthetic is monotonic and correlated.

We dedicate the following subsections to providing a rationale for common failures of LLMs when they execute code. We focus on the "simulation gap" caused by memorisation and shallow pattern recognition, revealing that LLMs can decide not to follow the prompt instructions while solving the task correctly, a phenomenon we name "lazy execution regime".

4.2. The "Simulation Gap": Between Shortcuts and Memorisation

A "lazy" code simulation regime. To deepen our understanding of why LLMs fail on sorting routines, we assessed whether they can simulate 8 sorting routines in their iterative and recursive versions and with log-linear or quadratic complexity. Each input is a vector of varying length ($\{10, 20, 30, 40\}$), where each element is an integer randomly sampled between 0 and 100. All the routines are reported in Appendix D.1.

We study GPT-3.5-Turbo, GPT-4, and Llama-3-70B and run 3 independent runs of 30 experiments each are shown in Figure 5. We analyse GPT-3.5-Turbo here as it results in the most interesting findings, though the same trends affect GPT-4 and Llama-3; see Appendix B.6. Beyond confirming the implicit bias of Transformers towards ordered sequences (Dufter et al., 2022) (i.e., they tend to output ordered sequences), Figure 5 (left) shows that LLMs often provide the correct result for classic sorting algorithms such as Insertion Sort. However, with less common algorithms, LLMs trace the expected execution but fail even for vectors of a few inputs. In summary, we find that (i) standard sorting algorithms (e.g., Insertion and Quick Sort) lead to more accurate results; (ii) there is a weak correlation between algorithm complexity and the accuracy of its simulation, thus reinforcing the hypothesis that in this case, the model is not simulating the procedure; (iii) there is a weak correlation
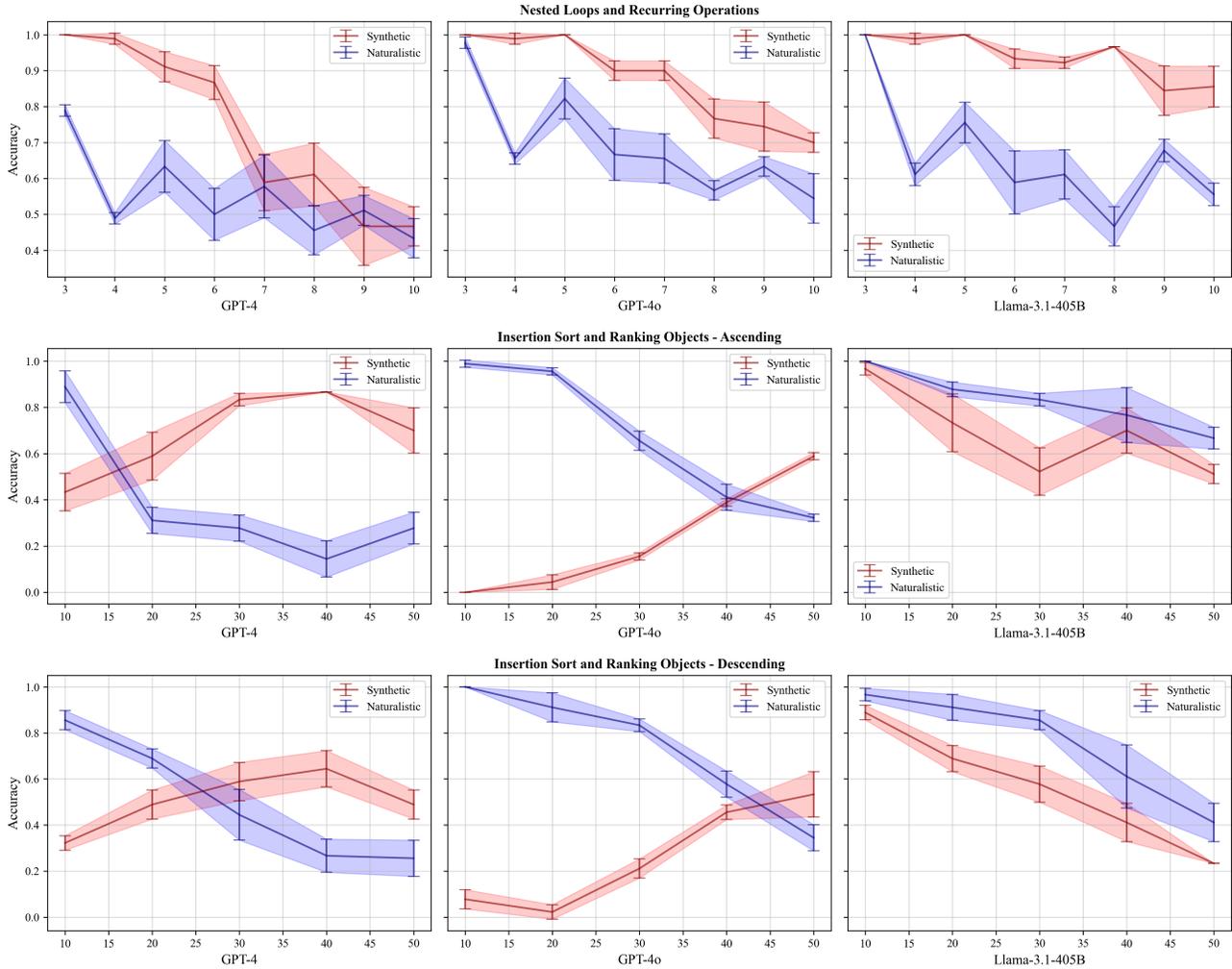
Figure 4. Top: Accuracy of different models on the Nested loops: the control variable is the max computational complexity of the task (synthetic) or, equivalently, the number of recurring operations (naturalistic). Middle and bottom: Accuracy on the Sorting and Ranking objects task: the control variable is the number of elements to sort/rank to give the correct answer.
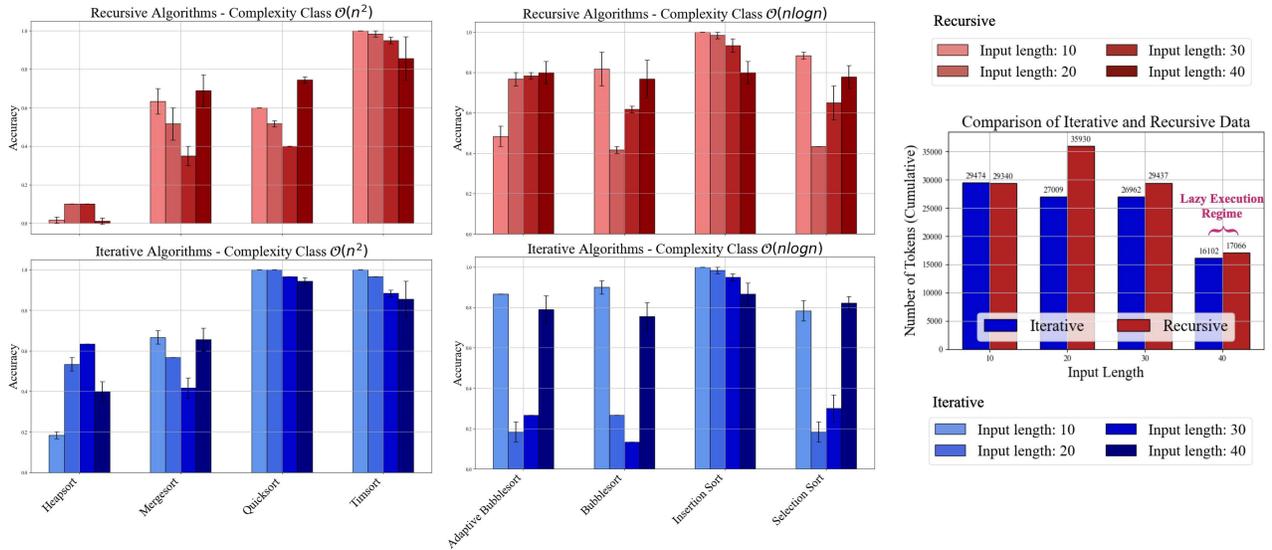
Figure 5. GPT-3.5-Turbo on sorting algorithms with varying complexity ($\mathcal{O}(n^2)$ and $\mathcal{O}(n \log n)$), both in their recursive (top) and iterative (bottom) versions. For long inputs, GPT-3.5 switches to a "lazy execution regime" (in **magenta**, right) where a model no longer simulates but just outputs the ordered sequence. Results on other models, including GPT-4, are reported in Appendix B.6.
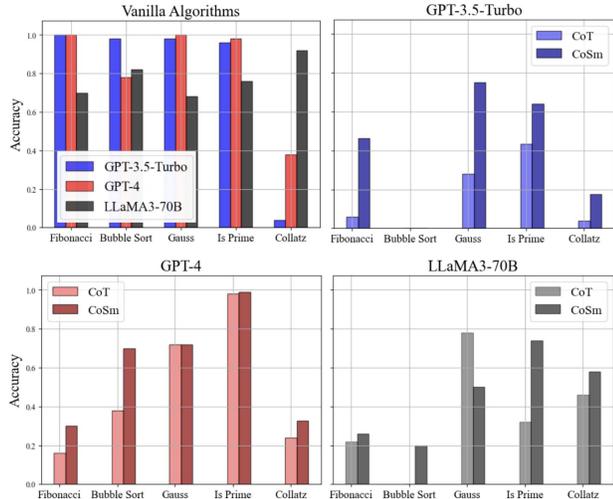
Figure 6. Results of GPT-3.5-Turbo, GPT-4 and Llama-3-70B on 50 independent simulations of classic algorithms and their variations. Top-left: Performance on the *vanilla* implementation of each algorithm with CoT. Top-right and bottom: Performance on the variations for each model with standard CoT vs. CoSm.
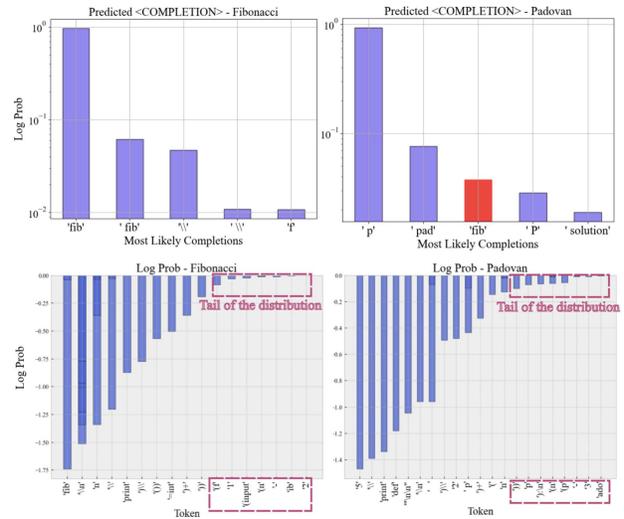
Figure 7. Top: Auto-completion of GPT-3.5-Turbo-Instruct when prompted to complete either the Fibonacci or Padovan function. Bottom: A failure case of an anti-memorisation technique (Shi et al., 2023): the least likely predicted tokens of both Fibonacci and Padovan overlap and are not informative of memorisation.

between the length of the input vector and the accuracy of a model.

By contrast, GPT-3.5-Turbo is accurate on input vectors of max length (i.e., 40). We name this phenomenon "lazy execution regime" (in **magenta**, Figure 5, right): the number of tokens generated for long sequences is considerably smaller than for shorter inputs, hinting that the task's induction bias is more evident when the input is a consistent portion of the prompt. Finally, we document an endemic case of failure with LLMs such as GPT-3.5-Turbo and GPT-4. On standard algorithms such as Bubble Sort, long input sequences often lead to the wrong result when they contain repeated elements, as we document in Appendix B.6. We hypothesise that the probability of a sequence that contains repeated elements is so low, conditioned on what has been generated so far, that a model skips repeating elements, even when we set the 'presence penalty' value to zero.

The pitfalls of memorisation with code. As the main research question of this work is whether one can use code as a proxy of high-order/reasoning tasks, it is important to investigate the role of memorisation on common algorithms, as they may bias the comparison to naturalistic, non-memorised tasks. We paired five standard sorting algorithms with slight variations that neither affect the code length nor their computational complexity. Yet, their semantics are slightly changed to produce a different output. We also introduce a prompting technique that, on top of CoT prompting, explicitly instructs a model to simulate a routine step by step: we name this prompting method Chain of Simulation, or CoSm (the prompt we use is reported in Appendix C). We investigated the following algorithms and their variations: (i) the **Fibonacci** sequence paired with **Padovan**, a slight variation that modifies the return condition; (ii) **ascending Bubble Sort**, paired with the **descending** routine; (iii) the **Gauss algorithm**, paired with a variation that, instead of summing the first n natural numbers, **adds even and subtracts odd numbers**; (iv) a **primality test** paired with the same routine on the **successor** of the input; (v) and the sum of the first n **Collatz numbers**, paired with a variation that returns the sum of the **even numbers** in a Collatz sequence. All the functions and their variations were first anonymised to avoid bias towards known function names and implemented with the code that appears most frequently on GitHub. We report their implementation in Appendix D.2. As shown in Figure 6 (top), GPT-3.5-Turbo, GPT-4 and Llama-3-70B are accurate on each classic algorithm, but their accuracy dropped significantly with the variations. On the other hand, their accuracy is marginally improved when we prompt them explicitly to simulate the routine with CoSm (Figure 6, bottom). We observe that LLMs anticipate the behaviour of a function by looking at specific dominating patterns. As shown in

Figure 7 (right), GPT-3.5-Instruct completes a template of the Fibonacci function with tokens compatible with the function's scope. On Padovan, there is a non-negligible probability (the third most likely token, in **red**) that the predicted function is Fibonacci. These results question the induction head mechanism (Olsson et al., 2022), as a slight variation of a task where an LLM is accurate results in errors of a large order of magnitude. Furthermore, this particular type of uncertainty goes unnoticed with methods that detect memorisation (Shi et al., 2023), as shown in Figure 7 (right). While anti-memorisation techniques are effective with textual inputs (e.g., the Wikipedia dataset), they fail on code. We hypothesise that low-frequency tokens are informative for natural language but not on code as it is more structured, with low-likelihood tokens (in **magenta**) that are not predictive of a model's memorisation.

5. Conclusions and Future Work

In this work, we introduce a way to evaluate some high-order capabilities of LLMs via synthetic tasks in the form of code. We show that many reasoning tasks have a natural reformulation as code, which is easy to obtain and scale. Our experiments show the feasibility of our approach and pave the way to synthetic benchmarks: on the other hand, code simulation may suffer from memorisation (for common routines such as sorting), and one has to carefully consider these drawbacks when designing a synthetic task as a proxy of high-order reasoning.

Acknowledgments

References

Berglund, L., Tong, M., Kaufmann, M., Balesni, M., Stickland, A. C., Korbak, T., and Evans, O. The reversal curse: Llms trained on" a is b" fail to learn" b is a". *arXiv preprint arXiv:2309.12288*, 2023. URL https://arxiv.org/pdf/2309.12288.pdf.

Biderman, S., Prashanth, U. S., Sutawika, L., Schoelkopf, H., Anthony, Q., Purohit, S., and Raf, E. Emergent and predictable memorization in large language models. *arXiv preprint arXiv:2304.11158*, 2023.

Chen, J., Pan, Z., Hu, X., Li, Z., Li, G., and Xia, X. Evaluating large language models with runtime behavior of program execution, 2024a.

Chen, M., Li, G., Wu, L.-I., Liu, R., Su, Y., Chang, X., and Xue, J. Can language models pretend solvers? logic code simulation with llms, 2024b.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Dufter, P., Schmitt, M., and Schütze, H. Position information in transformers: An overview. *Computational Linguistics*, 48(3):733–763, 2022.

Dziri, N., Lu, X., Sclar, M., Li, X. L., Jian, L., Lin, B. Y., West, P., Bhagavatula, C., Bras, R. L., Hwang, J. D., et al. Faith and fate: Limits of transformers on compositionality. *arXiv preprint arXiv:2305.18654*, 2023.

Eldan, R. and Russinovich, M. Who's harry potter? approximate unlearning in llms. *arXiv preprint arXiv:2310.02238*, 2023. URL https://www.thetalkingmachines.com/sites/default/files/2023-10/2310.02238.pdf.

Frieder, S., Pinchetti, L., Chevalier, A., Griffiths, R.-R., Salvatori, T., Lukasiewicz, T., Petersen, P. C., and Berner, J. Mathematical capabilities of ChatGPT. *ArXiv preprint*, abs/2301.13867, 2023. URL https://arxiv.org/abs/2301.13867.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Giannou, A., Rajput, S., Sohn, J.-y., Lee, K., Lee, J. D., and Papailiopoulos, D. Looped transformers as programmable computers. *arXiv preprint arXiv:2301.13196*, 2023.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., and et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Gu, A., Rozière, B., Leather, H., Solar-Lezama, A., Synnaeve, G., and Wang, S. I. Cruxeval: A benchmark for code reasoning, understanding and execution, 2024.

Hao, S., Gu, Y., Ma, H., Hong, J. J., Wang, Z., Wang, D. Z., and Hu, Z. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023. URL https://arxiv.org/abs/2305.14992.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

Huang, X. A., La Malfa, E., Marro, S., Asperti, A., Cohn, A., and Wooldridge, M. A notion of complexity for theory of mind via discrete world models. *arXiv preprint arXiv:2406.11911*, 2024.

Jojic, A., Wang, Z., and Jojic, N. Gpt is becoming a turing machine: Here are some ways to program it. *arXiv preprint arXiv:2303.14310*, 2023.

Kim, H., Sclar, M., Zhou, X., Bras, R. L., Kim, G., Choi, Y., and Sap, M. Fantom: A benchmark for stress-testing machine theory of mind in interactions, 2023. URL https://arxiv.org/abs/2310.15421.

Kim, N. and Schuster, S. Entity tracking in language models, 2023. URL https://arxiv.org/abs/2305.02363.

La Malfa, E., Petrov, A., Frieder, S., Weinhuber, C., Burnell, R., Cohn, A. G., Shadbolt, N., and Wooldridge, M. The arrt of language-models-as-a-service: Overview of a new paradigm and its challenges. *arXiv preprint arXiv:2309.16573*, 2023. URL https://arxiv.org/pdf/2309.16573.pdf.

La Malfa, E., Weinhuber, C., Torre, O., Lin, F., Cohn, A., Shadbolt, N., and Wooldridge, M. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074*, 2024.

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., and Zeng, A. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Lin, F., Malfa, E. L., Hofmann, V., Yang, E. M., Cohn, A., and Pierrehumbert, J. B. Graph-enhanced large language models in asynchronous plan reasoning, 2024.

Liu, C., Lu, S., Chen, W., Jiang, D., Svyatkovskiy, A., Fu, S., Sundaresan, N., and Duan, N. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*, 2023.

Liu, C., Zhang, S. D., Ibrahimzada, A. R., and Jabbarvand, R. Codemind: A framework to challenge large language models for code reasoning, 2024.

Lyu, C., Yan, L., Xing, R., Li, W., Samih, Y., Ji, T., and Wang, L. Large language models as code executors: An exploratory study, 2024. URL https://arxiv.org/abs/2410.06667.

McCoy, R. T., Smolensky, P., Linzen, T., Gao, J., and Celikyilmaz, A. How much do language models copy from their training data? Evaluating linguistic novelty in text generation using RAVEN. *Transactions of the Association for Computational Linguistics*, 11:652–670, 2023a. URL https://aclanthology.org/2023.tacl-1.38.

McCoy, R. T., Yao, S., Friedman, D., Hardy, M., and Griffiths, T. L. Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*, 2023b. URL https://arxiv.org/pdf/2309.13638.pdf.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Olsson, C., Elhage, N., Nanda, N., et al. In-context learning and induction heads. *Transformer Circuits Thread*, 2022. https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html.

OpenAI. GPT-4 technical report. *ArXiv preprint*, abs/2303.08774, 2023. URL https://arxiv.org/abs/2303.08774.

Ouyang, L., Wu, J., Jiang, X., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35: 27730–27744, 2022.

Pérez, J., Barceló, P., and Marinkovic, J. Attention is turing complete. *The Journal of Machine Learning Research*, 22(1):3463–3497, 2021.

Rabinowitz, N., Perbet, F., Song, F., Zhang, C., Eslami, S. M. A., and Botvinick, M. Machine theory of mind. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4218–4227. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/rabinowitz18a.html.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. URL https://arxiv.org/abs/2308.12950.

Santos, E. A., Prasad, P., and Becker, B. A. Always provide context: The effects of code context on programming error message enhancement. In *Proceedings of the ACM Conference on Global Computing Education Vol 1*, pp. 147–153, 2023.

Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., et al. BLOOM: A 176B-parameter open-access multilingual language model. *ArXiv preprint*, abs/2211.05100, 2022. URL https://arxiv.org/abs/2211.05100.

Schuurmans, D. Memory augmented large language models are computationally universal. *arXiv preprint arXiv:2301.04589*, 2023.

Shi, W., Ajith, A., Xia, M., Huang, Y., Liu, D., Blevins, T., Chen, D., and Zettlemoyer, L. Detecting pretraining data from large language models. *arXiv preprint arXiv:2310.16789*, 2023. URL https://arxiv.org/pdf/2310.16789.pdf.

Sweller, J. and Chandler, P. Evidence for cognitive load theory. *Cognition and instruction*, 8(4):351–362, 1991.

Tufano, M., Chandel, S., Agarwal, A., Sundaresan, N., and Clement, C. Predicting code coverage without execution. *arXiv preprint arXiv:2307.13383*, 2023.

Turpin, M., Michael, J., Perez, E., and Bowman, S. Language models don't always say what they think: unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36, 2024.

Vaithilingam, P., Zhang, T., and Glassman, E. L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI '22: CHI Conference on Human Factors in Computing Systems - 5 May 2022, Extended Abstracts*, pp. 332:1–332:7. ACM, 2022. doi: 10.1145/3491101.3519665. URL https://doi.org/10.1145/3491101.3519665.

Webb, T., Holyoak, K. J., and Lu, H. Emergent analogical reasoning in large language models, 2023.

Wei, C., Chen, Y., and Ma, T. Statistically meaningful approximation: a case study on approximating turing machines with transformers. *Advances in Neural Information Processing Systems*, 35:12071–12083, 2022a. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/4ebf1d74f53ece08512a23309d58df89-Paper-Conference.pdf.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *ArXiv preprint*, abs/2201.11903, 2022b. URL https://arxiv.org/abs/2201.11903.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. In *International Conference on Machine Learning*, pp. 11080–11090. PMLR, 2021. URL https://proceedings.mlr.press/v139/weiss21a/weiss21a.pdf.

West, P., Lu, X., Dziri, N., Brahman, F., Li, L., Hwang, J. D., Jiang, L., Fisher, J., Ravichander, A., Chandu, K., et al. The generative ai paradox:" what it can create, it may not understand". *arXiv preprint arXiv:2311.00059*, 2023.

Widjojo, P. and Treude, C. Addressing compiler errors: Stack overflow or large language models? *arXiv preprint arXiv:2307.10793*, 2023.

Yang, Z., Zhao, Z., Wang, C., Shi, J., Kim, D., Han, D., and Lo, D. What do code models memorize? an empirical study on large language models of code. *arXiv preprint arXiv:2308.09932*, 2023.

Yuan, Z., Yuan, H., Tan, C., Wang, W., and Huang, S. How well do large language models perform in arithmetic tasks? *arXiv preprint arXiv:2304.02015*, 2023. URL https://arxiv.org/pdf/2304.02015.pdf.

Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Yongji, W., and Lou, J.-G. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, 2023.

Zhang, D., Tigges, C., Zhang, Z., Biderman, S., Raginsky, M., and Ringer, T. Transformer-based models are not yet perfect at learning to emulate structural recursion, 2024.

Zhang, S. D., Tigges, C., Biderman, S., Raginsky, M., and Ringer, T. Can transformers learn to solve problems recursively? *arXiv preprint arXiv:2305.14699*, 2023. URL https://arxiv.org/pdf/2305.14699.pdf.

Zhou, H., Bradley, A., Littwin, E., Razin, N., Saremi, O., Susskind, J., Bengio, S., and Nakkiran, P. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023.

A. The Simulation Gap: Further Analyses of Naturalistic and Synthetic Tasks
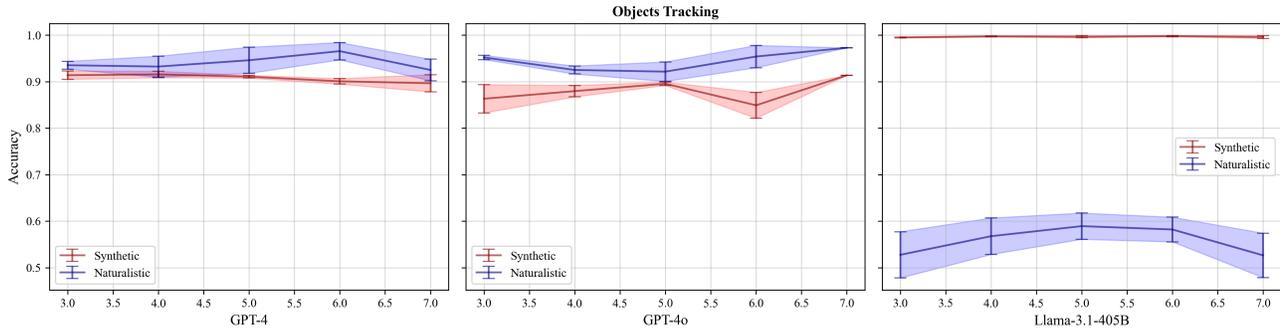

Figure 8. Accuracy of different models on the Object tracking task introduced in (Kim & Schuster, 2023). We pair the naturalistic task with an equivalent coding task.

A.1. Straight Line and Good Exchange

A linguistic analysis of the simulation gap. Across the incorrect solutions, which can be found in the code material and in particular in 'logs/straight-line/n_ops-40_n_vars-3_n_instances-2_batch-1.json', there are recurring issues like incomplete step-by-step tracking—where the model misinterprets large quantities or forgets to zero out a sender's items or incorrectly transferring the full amount to the recipient. The model also often updates only one side of an exchange (like subtracting from one agent but not adding to another), merges or ignores consecutive actions (for example, combining multiple "buy" actions), or makes arithmetic slips that throw off subsequent counts. Sometimes, it simply resets to a wrong intermediate value or cuts off its reasoning prematurely and provides a final answer without incorporating all of the steps.

Object tracking (Kim & Schuster, 2023). As reported in Figure 8, GPT-4 and GPT-4o have very good performance on both the synthetic and naturalistic tasks, while there is a strong gap in favour of the synthetic dataset for Llama-3.1-405B. We run a memorisation test on the naturalistic dataset, which has been well studied in many works and released two years ago. By feeding some truncated inputs of the naturalistic dataset to GPT-3.5-Turbo-Instruct, we were able to recover the remaining part of the input and the label, a strong hint that the dataset has been memorised verbatim by GPT models. We report a test of memorisation in Figure 9.

A.2. Critical Path and Critical Good Exchange and Parallel Path and Clique Good Exchange

Across these incorrect solutions, which can be found in the code material and in particular in which can be found in the code material and in particular in 'logs/critical-path/n_ops-30_n_vars-6_len_critical_path-5_batch-3.json' (the same errors also affect Parallel Path and Clique Good Exchange) the model, frequently truncates its step-by-step reasoning midway, skips the update for certain variables, and makes arithmetic errors involving negative signs or repeated multiplication (by two). It also struggles with handling variables after they are reset to zero, sometimes reusing outdated values or forgetting that the variable is now zero. In many instances, the model fails to complete all lines of code when attempting to explain or execute them, likely due to internal length constraints or confusion as it walks through numerous instructions. As a result, its chain of thought becomes inconsistent, causing final answers to be incorrect or absent altogether. These errors are especially pronounced in problems requiring precise step-by-step arithmetic and frequent reassignment of variables, where any small oversight can render the final result invalid.

Remove the brain from Box 3. Move the coat and the paper from Box 6 to Box 5. Remove the shirt from Box 2. Move the machine from Box 0 to Box 3. Move the brick and the cake and the glass from Box 4 to Box 0. Move the brick and the glass from Box 0 to Box 6. Box 5 contains the chemical and the coat and the paper.", "sentence_masked": "Box 0 contains the machine and the paper and the ticket, Box 1 contains the leaf and the stone and the wire, Box 2 contains the coat, Box 3 contains the brain, Box 4 contains the brick and the cake and the glass, Box 5 contains the chemical, Box 6 is empty. Move the coat from Box 2 to Box 6. Put the bottle and the radio into Box 5. Move the paper from Box 0 to Box 6. Put the shirt into Box 2. Move the bottle and the radio from Box 5 to Box 2. Remove the ticket from Box 0. Remove the brain from Box 3. Move the coat and the paper from Box 6 to Box 5. Remove the shirt from Box 2. Move the machine from Box 0 to Box 3. Move the brick and the cake and the glass from Box 4 to Box 0. Move the brick and the glass from Box 0 to Box 6. Box 5 <extra_id_0> .", "masked_content": "<extra_id_0> contains the chemical and the coat and the paper", "sample_id": 0, "numops": 3}

{"sentence": "Box 0 contains the machine and the paper and the ticket, Box 1 contains the leaf and the stone and the wire, Box 2 contains the coat, Box 3 contains the brain, Box 4 contains the brick and the cake and the glass, Box 5 contains the chemical, Box 6 is empty. Move the coat from Box 2 to Box 6. Put the bottle and the radio into Box 5. Move the paper from Box 0 to Box 6. Put the shirt into Box 2. Move the bottle and the radio from Box 5 to Box 2. Remove the ticket from Box 0. Remove the brain from Box 3. Move the coat and the paper from Box 6 to Box 5. Remove the shirt from Box 2. Move the machine from Box 0 to Box 3. Move the brick and the cake and the glass from Box 4 to Box 0. Move the brick and the glass from Box 0 to Box 6. Box 6 contains the brick and the glass.", "sentence_masked": "Box 0 contains the machine and the paper and the ticket, Box 1 contains the leaf and the stone and the wire, Box 2 contains the coat, Box 3 contains the brain, Box 4 contains the brick and the cake and the glass, Box 5 contains the chemical, Box 6 is empty. Move the coat from Box 2 to Box 6. Put the bottle and the radio into Box 5. Move the paper from Box 0 to Box 6. Put the shirt into Box 2. Move the bottle and the radio from Box 5 to Box 2. Remove the ticket from Box 0. Remove the brain from Box 3. Move the coat and the paper from Box 6 to Box 5. Remove the shirt from Box 2. Move the machine from Box 0 to Box 3. Move the brick and the cake and the glass from Box 4 to Box 0. Move the brick and the glass from Box 0 to Box 6. Box 6 <extra_id_0> .", "masked_content": "<extra_id_0> contains the brick and the glass", "sample_id": 0, "numops": 4}

{"sentence": "Box 0 contains the paper and the string, Box 1 contains the card and the cup, Box 2 contains the glass, Box 3 contains the leaf and the sheet and the tie, Box 4 contains the file and the radio, Box 5 contains the map, Box 6 contains the picture. Box 0 contains the paper and the string.", "sentence_masked": "Box 0 contains the paper and the string, Box 1 contains the card and the cup, Box 2 contains the glass, Box 3 contains the leaf and the sheet and the tie, Box 4 contains the file and the radio, Box 5 contains the map, Box 6 contains the picture. Box 0 <extra_id_0> .", "masked_content": "<extra_id_0> contains the paper and the string", "sample_id": 0, "numops": 1}

{"sentence": "Box 0 contains the paper and the string, Box 1 contains the card and the cup, Box 2 contains the glass, Box 3 contains the leaf and the sheet and the tie, Box 4 contains the file and the radio, Box 5 contains the map, Box 6 contains the picture. Box 0 contains the paper and the string.", "sentence_masked": "Box 0 contains the paper and the string, Box 1 contains the card and the cup, Box 2 contains the glass, Box 3 contains the leaf and the sheet and the tie, Box 4 contains the file and the radio, Box 5 contains the map, Box 6 contains the picture. Box 0 <extra_id_0> .", "masked_content": "<extra_id_0> contains the paper and the string", "

Model

gpt-3.5-turbo-instr...

Temperature   0

Maximum length   256

Stop sequences
Enter sequence and press Tab

Top P   1

Frequency penalty   0

Presence penalty   0

Best of   1

Inject start text
☑

Inject restart text
☑

Show probabilities

Most likely

🔒 API and Playground requests will not be used to train our models. Learn more

Figure 9. Memorisation of the Object tracking dataset (Kim & Schuster, 2023). When GPT-3.5-Turbo-Instruct is fed with some inputs from the dataset, it can recover the continuation almost verbatim (highlighted in yellow), a hint the model has memorised the dataset. For this example, we used the OpenAI Playground.
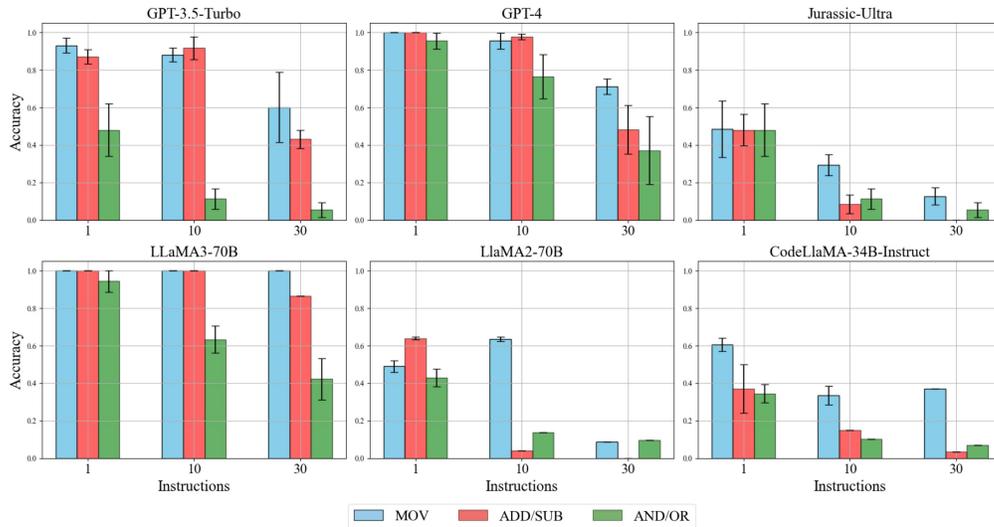
Figure 11. Accuracy on 3 independent runs of 30 experiments each of different LLMs on code snippets with **solely** {and, or}, {add, sub} or {mov} instructions. We group results by codes of varying number of instructions (x-axis), namely $\{1, 10, 30\}$.

B. An Analysis of the Code Simulation Capabilities of LLMs

Replicability. The code to replicate the experiments on pure code simulation is available here.

B.1. Straight Line

This section describes our results with straight-line programs, code with critical paths, and approximate and fault-tolerant instructions. We then study nested loops and sorting algorithms. The key controlled variable for the input is the number of instructions, in line with recent works in the area (Zhou et al., 2023).

B.2. Straight-line Programs Simulation

We first assess the simulation capabilities of different LLMs on code that contains only {add, sub}, {mov}, or logical-{and, or} instructions. Figure 11 shows that for code containing only one type of instruction, Jurassic2-Ultra, Llama-2-70B and CodeLLaMA-34b-Instruct are poor code simulators: their performance significantly downgrades with just 10 instructions, while GPT-3.5-Turbo, GPT-4 and Llama-3-70B are more accurate, though the same detrimental effect is evident, for example, on programs with 30 sequential instructions. Logical instructions are hard to simulate for any model (green bar): we hypothesise that the reason is their low coverage in the training set since even a simple neural network can

```
a0=-1; a1=0; a2=-6
a1 += a2
a0 = a2
a0 -= a0
a1 = a0
a0 -= a2
```

Figure 10. Straight-line code.

correctly compute logical-{and, or}. Since the performance of any model considerably drops with logical-{and, or} instructions, we exclude such operation and synthesise straight-line programs with $\{10, 20, 30, 40, 50\}$ lines of instructions and a fixed number of variables (e.g., 5), as shown in Figure 10. We then prompt an LLM to compute the value of one of such variables at the end of the execution. Both settings prompt each LLM to predict the state of a variable at the end of the computation. Figure 12 shows our results for code with mixed instructions: in this task, they successfully achieve compositionality and reliably simulate code with mixed instructions. GPT-4 and Llama-3 are reliable instruction simulators, followed by GPT-3.5-Turbo. Conversely, Jurassic2-Ultra, Llama-2-70B and CodeLLaMA-34B cannot simulate even short snippets of instructions. Qualitatively, we further note that most errors occur when the output of the LLM consists only of the final result of the computation rather than the complete program execution trace.[2]

B.3. Critical Path

---

[2]We observed this phenomenon with GPT-4 in more than 95% of cases, as per the code attachment.

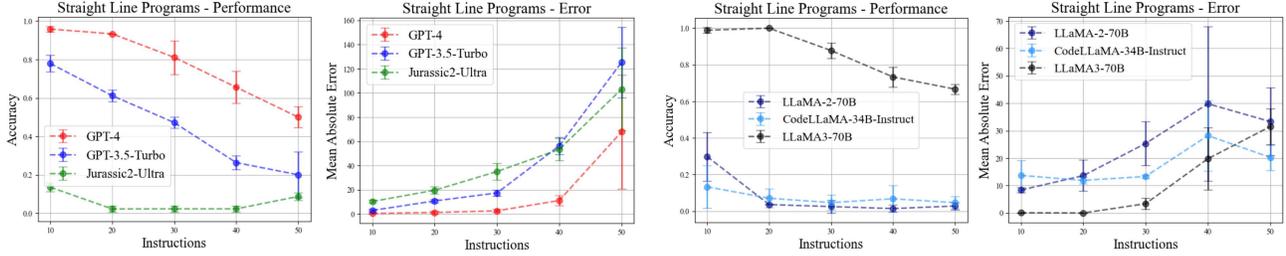*Figure 12.* Accuracy and Mean Absolute Error of different LLMs on code of varying length with **only** {add, sub} and {mov} instructions (out of 3 independent runs of 30 experiments each).
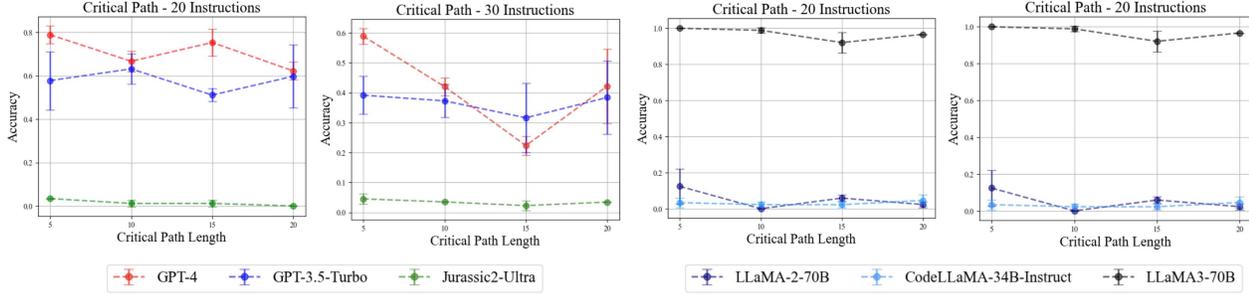

Figure 14. Accuracy of different LLMs on 3 runs of 30 experiments each on programs with varying critical path lengths, for snippets of 20 and 30 lines of code respectively.

Some sequential problems can be solved without executing all the instructions in a program. For instance, consider the code in Figure 13, with a model prompted to predict the value of a3. To compute the value of a3, it suffices to execute only those code blocks highlighted in red, which we refer to as the critical path of a3.[3] We thus perform experiments with programs that contain critical paths shorter than the entire program.

In Figure 14, and for 3 independent runs with 30 programs each, we present the results when GPT-3.5-Turbo, GPT-4, Jurassic2-Ultra, Llama-3-70B, Llama-2-70B and CodeLLaMA-34b-Instruct are prompted to execute snip-

```
a0 = a1 = a2 = 1
a3 = a4 = a5 = -1
a0 -= a1
a3 -= a4
a5 &= a3
a3 |= a5
a0 += a1
a1 -= a3
```

Figure 13. Code with critical path.

pets of 20 and 30 instructions, with critical paths of varying length (i.e., $\{5, 10, 15, 20\}$). GPT-4 and Llama-3-70B can leverage smart execution, though Llama-3-70B is better than GPT-4, especially on 30 lines of code. Although GPT-4's general simulation accuracy is higher than GPT-3.5-Turbo, it is less robust to variations of critical path length, i.e., GPT-4 suffers from a more severe accuracy drop compared to GPT-3.5-Turbo when critical path length approaches that of the entire program. We also notice that Llama-2-70B, CodeLLaMA-34B and Jurassic2-Ultra cannot generally execute instructions reliably. As with straight-line execution, we notice that most errors occur when the output trace contains only the result, not the code simulation.

B.4. Parallel Path

Approximate computation is evaluated with programs of k for loops with n instructions each. Each loop independently contributes to the final function return value, as shown in Figure 15. We denote by δ the probability of wrongly computing the result of each independent loop so that the probability of computing the exact result for a consistent *analog* computer on a program is $(1 - \delta)^k$. For an LLM, δ is computed as the Levenshtein similarity between the ground truth values and the predicted results and is a proxy for the approximation capabilities on programs of varying complexity. Results are reported in Figure 15. GPT-4 and Llama-3 are the best-performing models, with no accuracy degradation even for long programs with up to nine independent *threads*.

A routine is **tolerant to faults** when it can recover from errors occurring during the computation. To test LLMs in this

---

[3]From a theoretical perspective, a neural network that isolates a variable's critical path is straightforward to build, as shown in the Appendix, Section B.3.

Simulate the following program for n=5.

```
def f(n):
    n0=0; n1= 2; n3=-1
    for _ in range(n):
        n0 += 2
    for _ in range(n):
        n1 -= 1
    for _ in range(n):
        n2 *= -1
    return [n0, n1, n2]
```

[10, -6, 2]   [10, -3, 1]
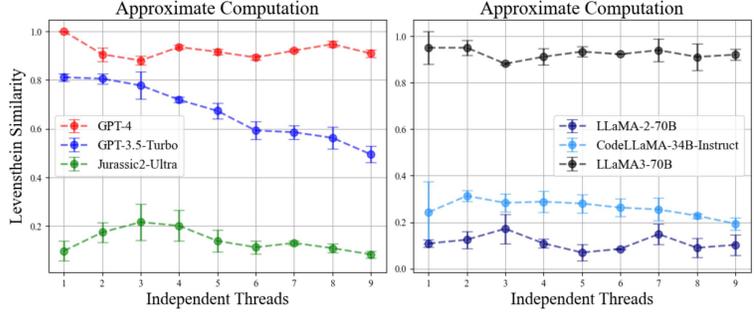
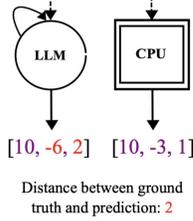Distance between ground
truth and prediction: 2

Figure 15. On the left, an example of an algorithm to test the approximation capabilities of a model. On the right, the Levenshtein similarity between the ground truth and an LLM's output measures the performance of different models (the higher, the better).

Simulate the following programs for n=5.
They should return the same value.

```
def f(n):
    n0=0; n1= 2; n3=-1
    for _ in range(n):
        n0 += 2
    for _ in range(n):
        n1 -= 1
    for _ in range(n):
        n2 *= -1
    return [n0, n1, n2]
```

```
def f(n):
    n0=0; n1= 2; n3=-1
    for _ in range(n):
        n2 *= -1
    for _ in range(n):
        n0 += 2
    for _ in range(n):
        n1 -= 1
    return [n0, n1, n2]
```
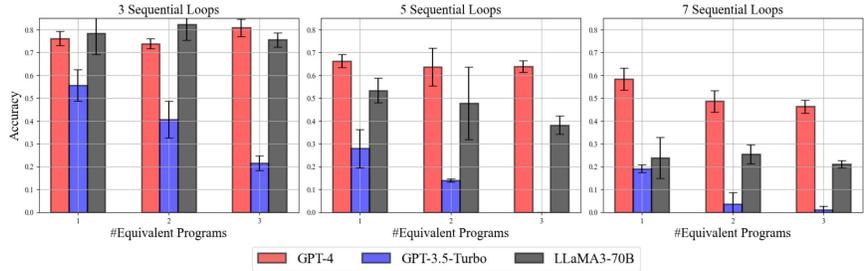
equivalent programs

Figure 16. Left: an example of a fault-tolerant algorithm. We feed an LLM with a few equivalent programs and instruct it to execute all of them to return the same result. Right: how redundancy affects the performances of GPT-3.5-Turbo, GPT-4 and Llama-3-70B on multiple equivalent programs.

setting, we prompt a model with different variations of the same algorithm, specifying that the objective is to demonstrate they yield the same result. Figure 16 (left) reports an example of fault-tolerant prompts. The illustration is complemented by results for different LLMs on three independent runs of 30 experiments each; the control variable is the number of equivalent programs fed to the model. While redundancy neither alters GPT-4 accuracy nor improves its performance, GPT-3.5-Turbo is heavily affected by multiple equivalent programs in the prompt and experiences a severe decrease in performance. Results for Jurassic2-Ultra, Llama-2-70B and CodeLLaMA-34B evidence low accuracy and are excluded from the evaluation (though inspectable in the code).

B.5. Nested Loops

Nested loops are a common instance of programs with polynomial running time $\mathcal{O}(n^k)$, where k is the depth of loop nesting: see, e.g., Figure 17. In this section, we prompt an LLM with programs that consist of k nested loops with n instructions
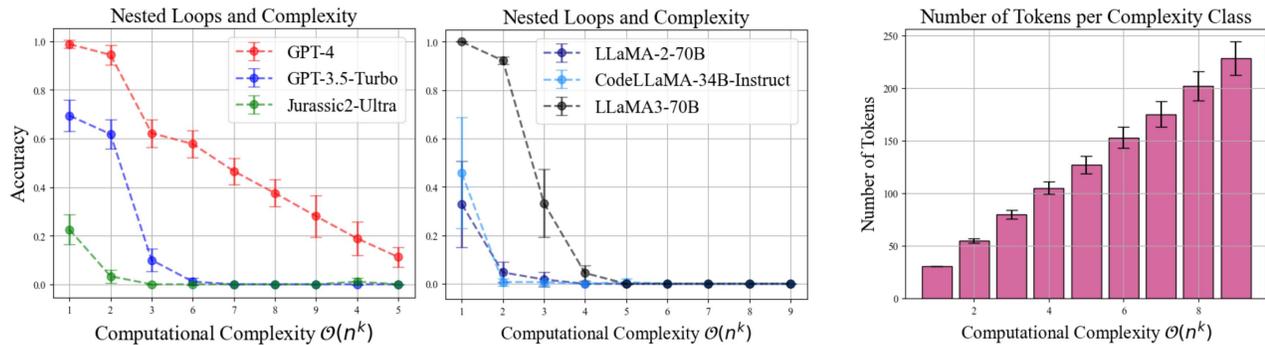

Figure 18. Performances of different LLMs on nested loops with increasing computational complexity. On the right, the number of input tokens per complexity class grows linearly.
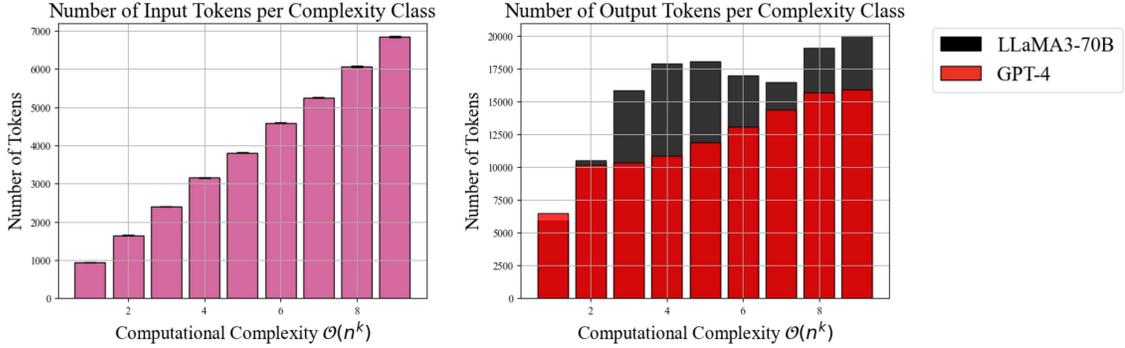
17

Figure 19. On sorting algorithms, the number of input tokens grows linearly (left). At the same time, GPT-4 outputs fewer tokens than Llama-3-70B, especially for complexity larger than $\mathcal{O}(n^2)$. The number is approximately the same for linear complexity. Both the graphs report the **cumulative** number of tokens over 30 experiments.

each, i.e., with time complexity that ranges from $\mathcal{O}(n^{k=1})$ (linear) to $\mathcal{O}(n^{k=9})$. We measure the performance of a model to predict the exact result of the computation. By construction, the return value is an integer bounded between $\pm 2^k$, with overall upper- and lower-bounds bounded between ± 1024, so we prevent high-order magnitude operands from influencing an LLM's performances. We run 3 independent runs with batches of 30 programs each and report the results for GPT-4, GPT-3.5-Turbo, Jurassic2-Ultra, Llama-3-70B, Llama-2-70B and CodeLLaMA-34b-Instruct.

Results in Figure 18 evidence a **strong non-linear negative correlation** between the accuracy of GPT-3.5-Turbo, GPT-4 and Llama-3-70B and the computational complexity of the function (right). In contrast, a strong linear correlation characterises the complexity of a function and its length (left). For high-performing LLMs (e.g., GPT-3.5, GPT-4, and Llama-3-70B), algorithms whose complexity is beyond quadratic induce the most significant drop in performance. This suggests that the current state-of-the-art models cannot reliably simulate routines whose complexity is cubic or beyond. This phenomenon necessitates further investigations to connect the work in (Zhou et al., 2023), or other works on the computational capabilities of Transformers (Weiss et al., 2021), with the computational complexity of a routine. Interestingly, Llama-3-70B was the best-performing model on the straight-line, approximate and critical path code, yet on nested loops, GPT-4 outperforms it by a solid margin. By inspecting the log results, we noticed that GPT-4, for

Simulate the following program for n=5.

```
def f(n):
  n0=0
  for _ in range(n):
    n0 += 2
  return sum([n0])
```

O(n)

```
def f(n):
  n0=0; n1= 2; n3=-1
  for _ in range(n):
    n0 += 2
    for _ in range(n):
      n1 -= 1
      for _ in range(n):
        n2 *= -1
  return sum([n0, n1, n2])
```

O(n³)

Figure 17. Examples of programs with varying computational complexity: on the left, linear ($\mathcal{O}(n)$), on the right, cubic ($\mathcal{O}(n^3)$).

high complexity programs (i.e., beyond $\mathcal{O}(n^2)$) implicitly unrolls the loops and **correctly guesses** the final result via **pattern matching**, surpassing any other model performance, including Llama-3-70B.

To give empirical evidence that GPT-4 does implicit computation without unrolling the loops, while Llama-3-70B tries to execute each instruction sequentially, we computed the number of tokens each model outputs in response to programs with different computational complexity. As reported in Figure 19, the **cumulative** number of input tokens grows linearly (left). At the same time, GPT-4 outputs fewer tokens than Llama-3-70B, especially for complexity larger than $\mathcal{O}(n^2)$. The number is approximately the same for linear and quadratic complexity.

B.6. Sorting

We report details on each sorting algorithm's space and time complexity in Table 1, while results for GPT-4 and Llama-3-70B on all the sorting routines are reported in Figure 20 and 21.

Repetita non iuvant. We report a case of emblematic failure that appears frequently with LLMs such as GPT-3.5-Turbo and GPT-4.
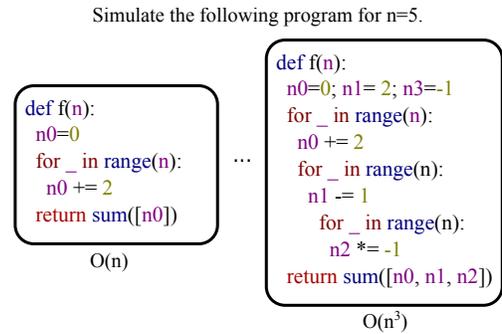
Table 1. Sorting algorithms space and time complexity. They reference results in Figure 5.

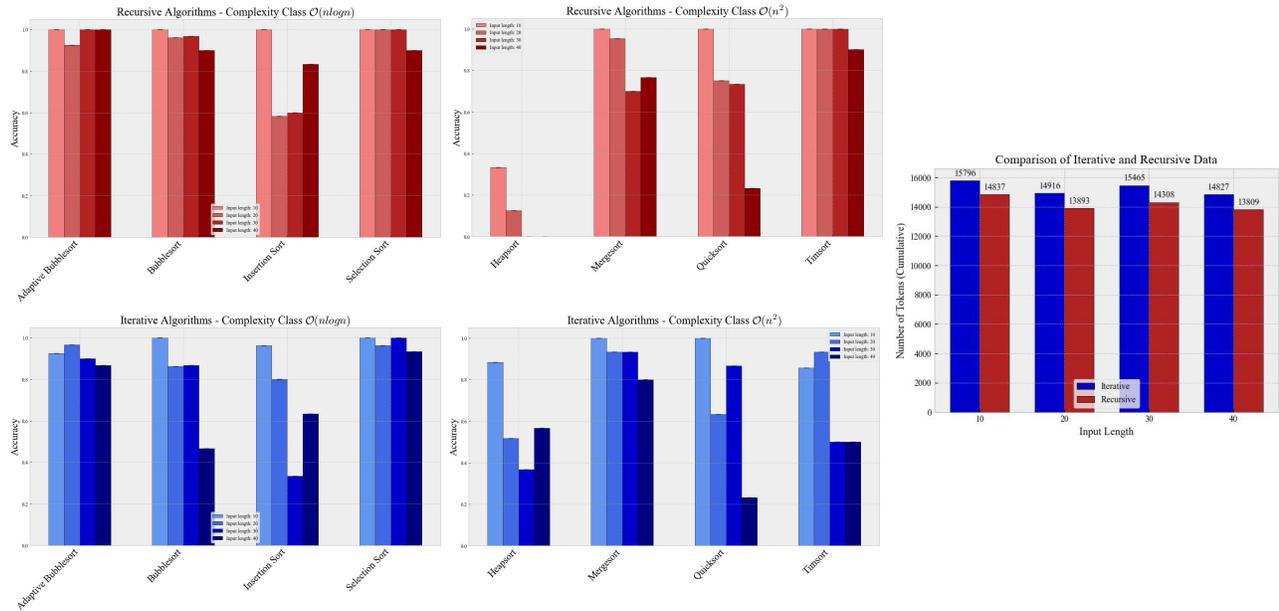| Algorithm	Worst Time Complexity	Average Time Complexity	Best Time Complexity	Space Complexity
Insertion Sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	It: $O(1)$ Rec: $O(n)$
Selection Sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	It: $O(1)$ Rec: $O(n)$
Bubblesort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	It: $O(1)$ Rec: $O(n)$
Adaptive Bubblesort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	It: $O(1)$ Rec: $O(n)$
Quicksort	$O(n^2)$	$\Theta(n\log(n))$	$\Omega(n\log(n))$	It: $O(n)$ Rec: $O(n)$
Mergesort	$O(n\log(n))$	$\Theta(n\log(n))$	$\Omega(n\log(n))$	It: $O(n)$ Rec: $O(n)$
Timsort	$O(n\log(n))$	$\Theta(n\log(n))$	$\Omega(n)$	It: $O(1)$ Rec: $O(n)$
Heapsort	$O(n\log(n))$	$\Theta(n\log(n))$	$\Omega(n\log(n))$	It: $O(1)$ Rec: $O(\log n)$

Figure 20. On top, results of GPT-4-Turbo with Chain of Thought prompting technique on different sorting algorithms, both in their recursive (top) and iterative (bottom) versions. Differently from GPT-3.5-Turbo, GPT-4 forces a model to simulate a routine and does not suffer from "lazy execution" for longer input vectors (as illustrated in Figure 5 and the relative section).
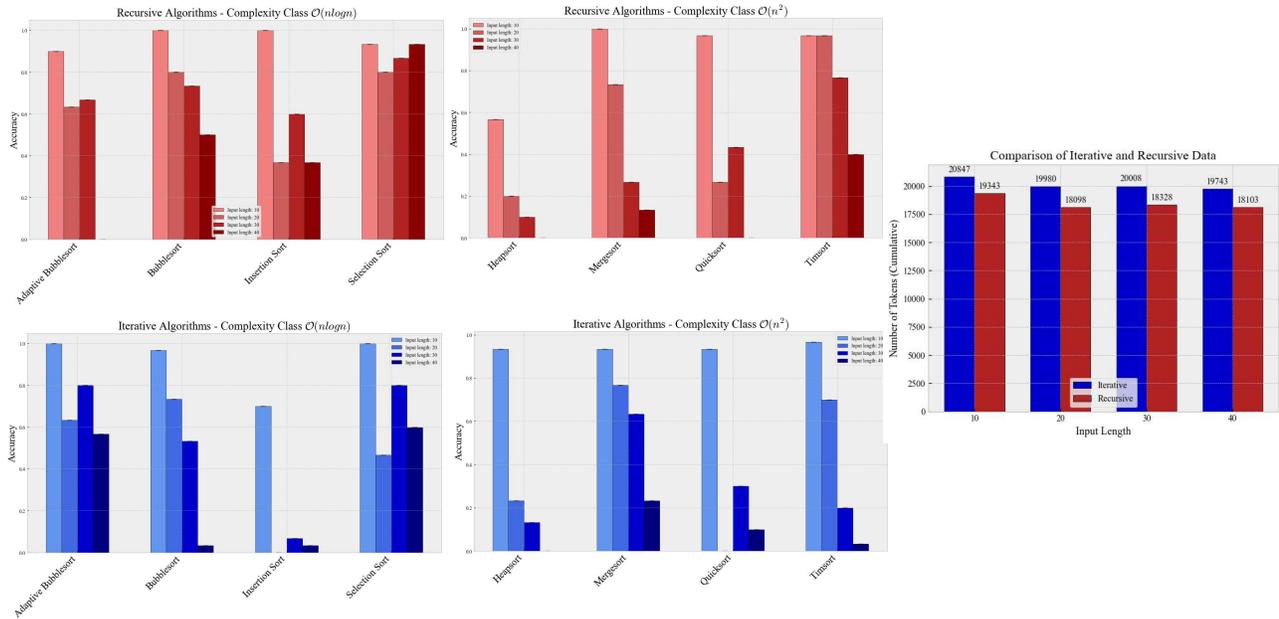
Figure 21. Results of Llama-3-70B with CoT prompting technique on different sorting algorithms, both in their recursive (top) and iterative (bottom) versions. Llama-3-70B is not a good simulator for sorting algorithms and, in general, does not understand the underlying task is sorting.

```
[85, 58, 6, 58, 34, 58, 93, 47, 5, 89, 86, 12, 51, 76, 0, 3, 63, 6, 74, 52, 46, 61, 34,
    92, 50, 56, 21, 25, 58, 80].
```

The previous sequence, alongside the code for Bubble Sort, is fed to GPT-4. The input vector contains the number 58 four times, highlighted in **magenta**). While GPT-3.5-Turbo correctly sorts the input vector, it reports the value 58 thrice. The LLM output is reported below, with the sequence of interest highlighted in **magenta**:

```
[0, 3, 5, 6, 6, 12, 21, 25, 34, 34, 46, 47, 50, 51, 52, 56, 58, 58, 58, 61, 63, 74, 76,
    80, 85, 86, 89, 92, 93]
```

Our hypothesis is that the probability of a sequence that contains the number 58 four times is so low, conditioned on what has been generated so far, that the model skips one of them, even though we set the presence penalty value to zero. In this case, which is not isolated and appears frequently for similar inputs, code simulation and correctness are in tension with the LLM output's probability distribution. We hypothesise that the probability of a sequence that contains repeated elements is so low, conditioned on what has been generated so far, that a model skips repeating elements, even when we set the 'presence penalty' value to zero.[4]

---

[4]https://platform.openai.com/docs/guides/text-generation/frequency-and-presence-penalties

C. Eliciting Code Simulation

Below, we report the exact implementation used for Chain of Simulation (CoSm).

```
"""
@code@
# 1. Simulate the above program instruction by instruction.
# 2. Report the trace of the program at the end of each iteration.
# 3. Think step by step and reply with the output of the function for the following
    input: @input@.
"""
```

D. Algorithms Implementation

D.1. Sorting Algorithms

D.1.1. RECURSIVE ALGORITHMS

Insertion Sort:

```python
def main(array, size, start=0):
    if start >= len(array) - 1:
        return array
    min_index = start
    for j in range(start + 1, len(array)):
        if array[j] < array[min_index]:
            min_index = j
    array[start], array[min_index] = array[min_index], array[start]
    return main(array, size, start + 1)
```

Bubble Sort:

```python
def main(list_data, length) :
    for i in range(length - 1):
        if list_data[i] > list_data[i + 1]:
            list_data[i], list_data[i + 1] = list_data[i + 1], list_data[i]
    return list_data if length<2 else main(list_data, length - 1)
```

Selection Sort:

```python
def main(array, size, start=0):
    if start >= len(array) - 1:
        return array
    min_index = start
    for j in range(start + 1, len(array)):
        if array[j] < array[min_index]:
            min_index = j
    array[start], array[min_index] = array[min_index], array[start]
    return main(array, size, start + 1)
```

Adaptive Bubblesort:

```python
def main(list_data, length) :
    swapped = False
    for i in range(length - 1):
        if list_data[i] > list_data[i + 1]:
            list_data[i], list_data[i + 1] = list_data[i + 1], list_data[i]
            swapped = True
    return list_data if not swapped else main(list_data, length - 1)
```

Quicksort:

```python
def main(array, high, low=0):
    if high==len(array):
        high=high-1
    if low < high:
        pi = f1(array, low, high)
        main(array, pi - 1, low)
        main(array, high, pi + 1)
    return array

def f1(array, low, high):
    pivot = array[high]
```

```
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    return i + 1
```

Merge Sort:

```
def main(arr, r, l=0):
    if r==len(arr):
        r=r-1
    if l < r:
        m = l+(r-l)//2
        main(arr, m, l)
        main(arr, r, m+1)
        f1(arr, l, m, r)
    return arr

def f1(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]
    i = 0
    j = 0
    k = l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
```

Tim Sort:

```
def main(lst, size):
    length = len(lst)
    runs, s_runs = [], []
    new_run = [lst[0]]
    s_array = []
    i = 1
    while i < length:
        if lst[i] < lst[i - 1]:
            runs.append(new_run)
            new_run = [lst[i]]
        else:
            new_run.append(lst[i])
```

```
        i += 1
    runs.append(new_run)
    for run in runs:
        s_runs.append(f2(run))
    for run in s_runs:
        s_array = f1(s_array, run)
    return s_array

def f1(left, right):
    if not left:
        return right
    if not right:
        return left
    if left[0] < right[0]:
        return [left[0], *f1(left[1:], right)]
    return [right[0], *f1(left, right[1:])]

def f2(lst):
    length = len(lst)
    for index in range(1, length):
        value = lst[index]
        pos = f3(lst, value, 0, index - 1)
        lst = lst[:pos] + [value] + lst[pos:index] + lst[index + 1 :]
    return lst

def f3(lst, item, start, end):
    if start == end:
        return start if lst[start] > item else start + 1
    if start > end:
        return start
    mid = (start + end) // 2
    if lst[mid] < item:
        return f3(lst, item, mid + 1, end)
    elif lst[mid] > item:
        return f3(lst, item, start, mid - 1)
    else:
        return mid
```

Heap Sort:

```
def main(u_arr,size):
    n = len(u_arr)
    for i in range(n // 2 - 1, -1, -1):
        f1(u_arr, i, n)
    for i in range(n - 1, 0, -1):
        u_arr[0], u_arr[i] = u_arr[i], u_arr[0]
        f1(u_arr, 0, i)
    return u_arr

def f1(u_arr, index, heap_size):
    largest = index
    left_index = 2 * index + 1
    right_index = 2 * index + 2
    if left_index < heap_size and u_arr[left_index] > u_arr[largest]:
        largest = left_index

    if right_index < heap_size and u_arr[right_index] > u_arr[largest]:
        largest = right_index

    if largest != index:
        u_arr[largest], u_arr[index] = u_arr[index], u_arr[largest]
        f1(u_arr, largest, heap_size)
```

D.1.2. ITERATIVE ALGORITHMS

Insertion Sort:

```python
def main(arr, size):
    for j, val in enumerate(arr[1:]):
        i = j
        while j >= 0 and val < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        if j != i:
            arr[j + 1] = val
    return arr
```

Bubble Sort:

```python
def main(collection, size=0):
    length = len(collection)
    for i in reversed(range(length)):
        for j in range(i):
            if collection[j] > collection[j + 1]:
                collection[j], collection[j + 1] = collection[j + 1], collection[j]
    return collection
```

Selection Sort:

```python
def main(collection, size=0):
    length = len(collection)
    for i in reversed(range(length)):
        for j in range(i):
            if collection[j] > collection[j + 1]:
                collection[j], collection[j + 1] = collection[j + 1], collection[j]
    return collection
```

Adaptive Bubblesort:

```python
def main(collection, size=0):
    length = len(collection)
    for i in reversed(range(length)):
        swapped = False
        for j in range(i):
            if collection[j] > collection[j + 1]:
                swapped = True
                collection[j], collection[j + 1] = collection[j + 1], collection[j]
        if not swapped:
            break
    return collection
```

Quicksort:

```python
def main(arr, h, l=0):
    if h==len(arr):
        h=h-1
    size = h - l + 1
    stack = [0] * (size)
    top = -1
    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h
    while top >= 0:
```

```
        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1
        p = f1( arr, l, h )
        if p-1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p - 1
        if p + 1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h
    return arr
```

Merge Sort:

```
def main(a, size):
    width = 1
    n = len(a)
    while (width < n):
        l=0;
        while (l < n):
            r = min(l+(width*2-1), n-1)
            m = min(l+width-1,n-1)
            f1(a, l, m, r)
            l += width*2
        width *= 2
    return a

def f1(a, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * n1
    R = [0] * n2
    for i in range(0, n1):
        L[i] = a[l + i]
    for i in range(0, n2):
        R[i] = a[m + i + 1]
    i, j, k = 0, 0, l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            a[k] = L[i]
            i += 1
        else:
            a[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        a[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        a[k] = R[j]
        j += 1
        k += 1
```

Tim Sort:

```
def main(arr,n):
    min_run = 32
```

```
    n = len(arr)
    for i in range(0, n, min_run):
        f2(arr, i, min((i + min_run - 1), n - 1))
    size = min_run
    while size < n:
        for start in range(0, n, size * 2):
            middle = min((start + size - 1), (n - 1))
            end = min((start + size * 2 - 1), (n - 1))
            if middle < end:
                f1(arr, start, middle, end)
        size *= 2
    return arr

def f2(arr, left=0, right=None):
    if right is None:
        right = len(arr) - 1
    for i in range(left + 1, right + 1):
        key_item = arr[i]
        j = i - 1
        while j >= left and arr[j] > key_item:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key_item

def f1(arr, left, middle, right):
    if arr[middle] <= arr[middle + 1]:
        return
    left_copy = arr[left:middle + 1]
    right_copy = arr[middle + 1:right + 1]
    left_copy_index = 0
    right_copy_index = 0
    s_index = left
    while left_copy_index < len(left_copy) and right_copy_index < len(right_copy):
        if left_copy[left_copy_index] <= right_copy[right_copy_index]:
            arr[s_index] = left_copy[left_copy_index]
            left_copy_index += 1
        else:
            arr[s_index] = right_copy[right_copy_index]
            right_copy_index += 1
        s_index += 1
    while left_copy_index < len(left_copy):
        arr[s_index] = left_copy[left_copy_index]
        left_copy_index += 1
        s_index += 1
    while right_copy_index < len(right_copy):
        arr[s_index] = right_copy[right_copy_index]
        right_copy_index += 1
        s_index += 1
```

Heap Sort:

```
def main(arr, n):
    f1(arr, n)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        j, index = 0, 0
        while True:
            index = 2 * j + 1
            if (index < (i - 1) and
                arr[index] < arr[index + 1]):
                index += 1
            if index < i and arr[j] < arr[index]:
                arr[j], arr[index] = arr[index], arr[j]
            j = index
```

```
        if index >= i:
            break
    return arr

def f1(arr, n):
    for i in range(n):
        if arr[i] > arr[int((i - 1) / 2)]:
            j = i
            while arr[j] > arr[int((j - 1) / 2)]:
                (arr[j],
                 arr[int((j - 1) / 2)]) = (arr[int((j - 1) / 2)],arr[j])
                j = int((j - 1) / 2)
```

D.2. Standard Algorithms and Variations

Fibonacci (iterative):

```
def f(n):
    a, b = 0, 1
    if n <=1:
        return n
    else:
        for i in range(1, n):
            c = a + b
            a = b
            b = c
        return b
```

Padovan (iterative):

```
def g(n):
    a, b = 1, 1
    c, d = 1, 1
    for i in range(3, n+1):
        d = a + b
        a = b
        b = c
        c = d
    return d
```

Bubble Sort (iterative):

```
def f(v):
    n = len(v)
    for i in range(n):
        for j in range(0, n-i-1):
            if v[j] > v[j+1]:
                v[j], v[j+1] = v[j+1], v[j]
    return v
```

Bubble Sort Descending (iterative):

```
def g(v):
    n = len(v)
    for i in range(n):
        for j in range(0, n-i-1):
            if 0 > v[j] - v[j+1]:
                v[j], v[j+1] = v[j+1], v[j]
    return v
```

Gauss Sum:

```python
def f(n):
    tot = 0
    for i in range(n):
        tot += i
    return tot
```

Gauss Sum and Subtraction:

```python
def g(n):
    tot = 0
    for i in range(n):
        tot += (i if i%2==0 else -i)
    return tot
```

Is Prime:

```python
def f(n):
    if n < 2: return False
    for x in range(2, int(n**0.5) + 1):
        if n % x == 0:
            return False
    return True
```

Is Prime on Successor:

```python
def g(n):
    n = n+1
    if n < 2: return False
    for x in range(2, int(n**0.5) + 1):
        if n % x == 0:
            return False
    return True
```

Collatz Sum Even:

```python
def g(n):
    s = n
    while n != 1:
        if n % 2 == 0:
            n = n // 2
            s += n
        else:
            n = 3 * n + 1
    return s
```