UNIVERSITAT DE BARCELONA

FUNDAMENTAL PRINCIPLES OF DATA SCIENCE MASTER'S THESIS

Evaluating Large Language Models as computer programming teaching assistants

Author: Maria Magdalena Pol Pujadas Supervisors: Dr. Daniel Ortiz Dr. Eloi Puertas

A thesis submitted in partial fulfillment of the requirements for the degree of MSc in Fundamental Principles of Data Science

in the

Facultat de Matemàtiques i Informàtica

UNIVERSITAT DE BARCELONA

Abstract

Facultat de Matemàtiques i Informàtica

MSc

Evaluating Large Language Models as computer programming teaching assistants

by Maria Magdalena Pol Pujadas

The principal aim of this project is to conduct an analysis of how different Large Language Models (LLMs) operate in diverse context and situations in the field of education. In particular, we aim to assess the suitability of LLMs for specific tasks within the domain of algorithmic subjects within computer science studies.

The tasks under analysis are designed to assist both students and teachers. With regard to students, we will assess the capacity of the models to implement a specified code. When it comes to teachers, we will evaluate the models' abilities to identify the target of the introduced code and potential errors introduced by students in their codes, enabling students to become more self-taught and seek assistance from teachers when necessary.

To evaluate these tasks, we have considered eight models. Two closed-source models were evaluated: GPT-3.5 and GPT-4. Five open-source models were also considered: Llama2, Codellama instruct, Llama3, Platypus2, Deepseek Coder and Qwen-1.5.

Acknowledgements

I would like to express my gratitude to all my friends, family and colleagues who have supported me throughout my master's degree, particularly those who have encouraged me to persevere and trusted me during the past months full of new emotions and changes. I would also like to express my gratitude to my supervisors, Daniel and Eloi, for their invaluable assistance and flexibility in adapting to the project's evolving requirements, to complete the project in the most optimal manner.

Chapter 1

Introduction

1.1 LLM in Education

The question of whether to permit the use of Large Language Models (LLM) in education is frequently a subject of debate. On the one hand, some adhere to more traditional pedagogical approaches and would prefer to prohibit using such models. On the other hand, some educators are committed to innovation and believe that the potential of these models can be utilised in a controlled manner.

It is important to recognise that not all students will have the same intentions when engaging with these models. Our focus is on students who are enrolled in courses with the specific purpose of learning. Such students would be interested in utilising the capabilities of these models to facilitate accelerated learning and to allocate more time to activities that are perceived to be of greater value.

It is our assumption that, as education has evolved to accommodate new technologies in the recent years, it will also integrate the use of artificial intelligence in a controlled and secure manner. The objective of this project is to conduct an analysis of how different LLMs operate in diverse contexts and situations.

By analysing this, we aim to identify potential areas where LLMs could be of significant benefit, to both students and teachers, as well as to identify areas where further development is required for LLMs to be truly useful. The objective of this analysis is to identify both the strengths and weaknesses of the various models, thereby enabling the differentiation of those models that are suitable for specific tasks. Furthermore, it is also our intention to demystify LLMs in the eyes of students, demonstrating that LLMs are not infallible and cannot be expected to perform all tasks perfectly on their first attempt. We aim to identify the contexts in which they are useful, while also pointing out their inherent limitations and noting that they are not a 100% effective solution to every problem.

Students are habituated to seeking answers on the internet. Having grown up with the internet, they are aware that not everything they find there is accurate and that there are reliable sources and others that are best avoided.

In this regard, there is a significant distinction between utilizing the internet and using an LLM. In the context of LLMs, it is more challenging to establish a traceability that enables the differentiation between responses from reliable sources and those from unreliable ones. For this reason, it is of great importance to subject the responses to critical analysis, and it would be beneficial to identify instances where the LLM responses may contain inaccurate or incomplete information. It is therefore essential for students to analyse these answers with a critical eye, to determine which parts of the answer to use and which parts to contradict. This will ensure that the answer obtained is valid.

In this project, we will focus on the field of computer science and, more specifically, algorithm-related subjects. In the context of information technology, the use of Large

Language Models is typically oriented towards assisting programmers in accelerating and improving the efficiency of their work. However, the question remains as to how these technologies can be best utilised by those who are still in the process of learning to code. As this question appears to be a relatively uncommon one, we have decided to direct our attention towards the group of individuals who are new to the field of computer science and programming. The tasks we will be focusing on are typical of those that a first-year student on an algorithmic-related subject within computer science would undertake. The objective is to assess the potential utility of LLMs for this particular group of learners.

1.1.1 Use of the LLM in Algorithms

In algorithm-related subjects, one of the primary objectives is to educate students on the comprehension and interpretation of mathematical problems, as well as the efficient and accurate implementation of these solutions. As these students are in their first year, the majority of them haven't yet learned how to program. With these considerations in mind, we have determined that it would be beneficial to focus on a selection of classic problems that could be taught in an introductory algorithmics course. Additionally, we aim to identify situations where language models could be used effectively.

The objective is to consider a scenario in which a problem is presented in class and students are required to develop a code that performs all the functions previously explained by the teacher. Before implementing the code, it would be anticipated that the student would have a comprehensive understanding of the problem they are working and would be able to consider the most effective means of implementing the solution.

One method for achieving this is to initially conceptualise the problem in a manual, pen-and-paper format, before subsequently adapting this concept to align with the specific syntax and semantics of the chosen programming language. Students will probably require external assistance to successfully generate code. We have assumed that the use of LLMs would be one of their first attempts. To gain insight into how students and teachers might approach the use of LLMs, we have placed ourselves in the position of both the student who wants to complete the task and the teacher who wants to help students overcome their doubts.

In light of the aforementioned considerations, we propose to analyse two distinct phases of the process. The following items will be considered in the analysis:

- Implementing the code
- Code interpretation and error detection

Implementing the code

To implement the code or a preliminary version of it, it would be optimal for students to attempt to solve the problem using the techniques that were discussed in class or those that they have already mastered. Once a partial solution has been obtained, it is recommended that students compare it with another classmate who has completed the same exercise. If this is not possible, students should request assistance from their teacher. It should be noted that this scenario is not always feasible, as in many cases students lack the time to fully comprehend all the concepts. Consequently, with convenient access to LLMs, it is evident that students will frequently seek solutions from them. The objective is to adopt the perspective of students and request that the model provides a code capable of performing a specific action. It would be beneficial to be able to simulate a variety of potential approaches when requesting information from the model.

Due to time and space constraints, it is not feasible to run more than three versions for each algorithm. In the initial two versions, it is assumed that the student possesses a solid mathematical background and that their role is to explain to the model the underlying principles of the algorithm through mathematical reasoning.

It is considered a feasible scenario to assume that the student requests the model to provide the code directly, without any previous steps. The final of the three versions will consider another feasible scenario. We now assume that the students have dedicated themselves to the task and have developed a pseudocode representation of the desired implementation. In this scenario, the student would provide the model with the pseudocode and request for it to be converted it into a code with the appropriate syntax.

This second option appears to be more beneficial from a pedagogical standpoint, as it requires the student to demonstrate their understanding of the concepts by creating the pseudocode. If the generated solution is comprehensible and accurate, the student will also gain familiarity with the syntax, thereby reducing the need to rely on LLM for subsequent tasks.

This exercise allows us to assess the efficacy of different models and identify potential limitations that students may encounter. To this end, we will pose a series of questions designed to facilitate an analysis of the responses, including queries such as: "Do the generated answers perform exactly the functions they are asked to perform?" and "Do the generated answers provide code that performs the expected actions and can be executed directly?". Furthermore, it would be beneficial to analyse whether the models perform better when they are required to simply convert pseudocode into code, as opposed to having to comprehend mathematical reasoning.

Code Interpretation and Error Detection

Another situation that requires our attention is when a student has developed a code but is uncertain about its functionality, or has encountered an error in their code that they are unable to resolve. In such a case, we may be confronted with one of the following scenarios:

- 1. The code is free of errors and, therefore, ready for submission to the teacher.
- 2. The code exhibits conceptual errors that are potentially dangerous because they can be challenging to identify. While the code may run smoothly and lead to an incorrect solution, students should always verify that the solution their code reaches and the actual solution is identical. If this is not the case, it can be difficult to determine where the error lies, because there is no compilation error.
- 3. The code is free of conceptual errors, but contains syntax errors. These errors arise when the syntax of the code is incorrect.

If students were required to seek assistance from the instructor each time they encountered an error, it would be a significant challenge for the instructor to provide individual attention to all students during each delivery. We wondered what would happen if the teacher provided the students with a tool to assist them in correcting their codes when they encountered situations 2 or 3.

In the same manner as we previously considered the perspective of the student, we now consider the position of the teacher. Our objective is to provide students with the necessary tools to become more self-taught and to ask only those questions that are essential.

In this context, we propose the utilisation of LLM to assist with the completion of a specific task. We consider a scenario in which educators develop an interface that enables students to input their code and receive feedback regarding its purpose and any errors. For this to occur, the model must be provided with explicit instructions regarding how it should process the code that has been entered. Similarly, the output of the LLMs must have a fixed structure. This is because we are not necessarily interested in directly answering the learner; instead, we are interested in associating the LLM's output with a fixed structure.

The objective is to create a prompt that is capable of effectively detecting the target of the code or errors in the desired format. To achieve this, several versions of the same prompt will be run, and the output will be analysed to identify any notable differences between them.

These experiments have, as objective, to ascertain whether the issues identified in scenarios 2 and 3 can be addressed. The focus will extend beyond erroneous codes to encompass those codes that are correct.

It is equally important to ensure that, in the case of correct codes, the model does not erroneously assign errors. For the purpose of this study, the most common errors made when learning to program will be considered in relation to the identification of errors. In the analysis of the responses, particular attention will be paid to the structure of the output, as any format that does not align with the desired specifications will be unsuitable for use in future tools.

Chapter 2

State of the art

2.1 Theoretical framework

2.1.1 Natural Language Process from the beginning to the Transformers

The field of linguistics and artificial intelligence first came into contact in the early 1950s, resulting in the emergence of Natural Language Processing (NLP). In the beginning, the tasks performed were relatively simple, such as word-by-word translation from one language to another. This initial approach subsequently evolved to encompass more complex linguistic tasks, such as identifying words with the same spelling but multiple meanings or detecting metaphors. Subsequently, so-called grammars emerged, which serve as the foundation for regular expressions, which are currently employed to identify patterns within text. Up to this point, NLP was focused on handcrafted rules. However, natural language is a much more complex phenomenon than that, and it can contain ambiguities. Consequently, there was a need to extract semantics from the text to address syntax. Later on, machine learning methods employing probabilities were first employed, giving rise to statistical NLP. (Nadkarni, Ohno-Machado, and Chapman, 2011).

The exponential growth in computer capabilities, the vast amount of data available, and the development of new machine learning architectures, such as deep neural networks, have led to a radical transformation in the field of natural language processing. Initially, progress was gradual, but it accelerated exponentially with the emergence of transformer architectures. (Chernyavskiy, Ilvovsky, and Nakov, 2021) Natural Language models are machine learning models that calculate the probability of the next word given the preceding ones. These models have emergent capabilities as they can process, understand, and generate human language. ELIZA was one of the first natural language models. It was a chatbot programmed in the 1960s that simulated a conversation with a therapist (Weizenbaum, 1966). N-grams is another natural language model that emerged from the necessity to comprehend and process human language. Despite its simplicity, the model proved to be an effective one. The concept of n-grams is based on the premise that the context of each word contributes to the overall meaning of a sentence (Jurafsky and Martin, 2023). These models operate by dividing a given sentence into groups of n elements or words, to identify the context of any given word within the sentence. The ability to detect context is useful in that a given word may have a completely different meaning depending on the context in which it is used. One of the limitations of these models is that they are unable to capture long-term relationships between words. Additionally, as the value of n increases, the algorithm becomes computationally expensive. To address this issue, neural network algorithms were introduced.

Neural networks (NN) are inspired by the structure and functionality of the brain (Jurafsky and Martin, 2023). The structure of neural networks is comprised of interconnected nodes, which are known as neurons, organised in layers. The neurons are linked together, with a weight assigned to each connection. This weight can be understood as the significance of the variable under consideration. A NN is comprised of three principal layers: the input layer, one or more hidden layers, and the output layer. The neurons in the hidden layer are responsible for recognising patterns within the input data. As patterns are identified, the neural network modifies its parameters until it identifies the optimal combination that generates the desired output. The inclusion of multiple hidden layers enables the capture of diverse and pertinent information. A neural network comprising multiple layers is a deep learning model. A prominent example of such a model is transformers. To comprehend the architecture of transformers, it is essential to be familiar with two key concepts: the encoder-decoder framework and the self-attention mechanism. The encoderdecoder framework is responsible for processing an input sequence and generating an output sequence. Such a model is designated a sequence-to-sequence model. The encoder and decoder are two interconnected subnetworks. The encoder is responsible for transforming the input sequence into a concise and compact representation, ensuring that all relevant information is retained. This representation will subsequently be utilised by the decoder input to generate the output. The self-attention mechanism is also a sequence-to-sequence model. To determine the output vector, the model computes a weighted average across all input vectors. This enables the model to focus its attention on the information in the input sequence that is deemed most relevant. "Self-attention can be thought of as a way to build contextual representations of a word's meaning that integrate information from surrounding words, helping the model learn how words relate to each other over large spans of text." (Jurafsky and Martin, 2023).

With an understanding of these foundational concepts, the transformer's architecture emerges. "The intuition of a transformer is that across a series of layers, we build up richer and richer contextualized representations of the meanings of input words or tokens." (Jurafsky and Martin, 2023).

The transformer architecture comprises both encoder and decoder stacks. The encoder comprises N identical layers, each containing two sublayers. The first sublayer employs a multi-head self-attention mechanism. The multi-head self-attention mechanism accommodate the complex relationships between different words within a sentence, as multiple words may relate to each other in various ways simultaneously. The second sublayer of the encoder is a feed-forward network, which may be conceived of as a neural network. Subsequently, each of these sublayers is followed by a normalisation step, which serves to stabilise the training process. Similarly, the decoder is structured in a manner analogous to that of the encoder, with the primary distinction being the incorporation of a third sublayer. This additional sublayer also employs multi-head attention, but it focuses on the output of the encoder stack. It is notably, the multi-head self-attention layer within the decoder is masked to ensure that predictions for position *i* are based solely on the known outputs at positions less than *i*. This effectively restricts the model's access to future information and ensures that only past context is considered (Vaswani et al., 2017).

The transformer architecture enabled the emergence of pre-trained language models (PLMs), which were designed to learn a representation of language that could be used to perform a range of natural language processing (NLP) tasks. These models demonstrated improved performance as the number of parameters and the amount of data used in the training stage increased. This led to the development of large language models (LLMs), which have billions of parameters. "LLMs appear to have emergent abilities such as reasoning, planning, decision-making, in-context learning, answering in zero-shot settings, etc" (Naveed et al., 2023). These models demonstrate the capacity to perform a range of tasks, even in the absence of direct training on those tasks. A suitable description of the activity to be performed and examples of how to do it enable the model to learn how to perform the task correctly and achieve satisfactory results.

The architectural design of these models may vary depending on how attention is applied to them and the manner of connection between the transformer blocks. Some of the most widely used architectural approaches include encoder-decoder, causal-decoder, and prefix-decoder architectures. The primary distinction between the Causal-Decoder and Prefix-Decoder architectures lies in the manner of attention estimation. In the initial architectural configuration, the subsequent token is solely contingent upon the preceding context. Conversely, in the second architectural configuration, the attention mechanism is bidirectional. The selection of the architectural approach and the training methodology are determined by the specific objectives of the model. Two of the principal objectives of pretraining are to predict the subsequent word in a sequence based on the preceding words (full language modelling) and to fill in the blanks in a text based on both the preceding context and the context that follows (masked language modelling). (Wang et al., 2022). To create a model's base, a significant volume of data must be made available for the model to learn from during a process known as pretraining. It is of crucial importance to utilise high-quality data, as the LLM will be as proficient as the data it has been trained on. For the model to possess knowledge of a specific downstream task, a process of fine-tuning is employed. This process of adapting the model to a new task is known as transfer learning. However, there are other fine-tuning processes, such as instruction tuning and alignment tuning. The objective of instruction tuning is to guide the model to respond in a specific way, whereas alignment tuning aims to ensure that the model is helpful, honest, and harmless. The process of finetuning a model is computationally complex and requires the use of new data for that specific approach. An alternative approach to finetuning is to design natural language instructions to guide LLMs in providing effective responses. This is a much simpler method. This technique is referred to as prompt engineering. Prompt engineering helps to create instructions that can be provided to the system to perform a specific task. To achieve this, a variety of strategies can be employed, including the use of (Saravia, 2022):

- **Zero-shot prompting** is the process of instructing the LLM to perform a specified action without the provision of an example of how to do so.
- **In-context learning**, also known as **few-shot learning**, involves providing examples of inputs and outputs to an LLM so that it can learn how to respond expectedly.
- Reasoning in LLMs: Instructions are provided to the model to assist it in reasoning. Chain-of-Thoughts represents a specific instance of prompting, characterised by the incorporation of a step-by-step reasoning process within the prompt, to facilitate the model's comprehension of the optimal approach for generating the desired response.

It is important to consider that LLM models are trained with limited contextual windows, as attention and memory require significant resources. To achieve the desired outcomes, it is essential to provide a suitable prompt that is neither excessively lengthy nor brief, as it should encompass sufficient information to provide context to the model, thereby preventing hallucinations or the generation of erroneous responses. (Naveed et al., 2023)

Given the constraints on the capacity for attention and memory exhibited by the models, it is crucial to design effective prompts by employing techniques that facilitate prompt engineering. It is crucial to recognise that prompts serve as the input for LLMs, and their function is of paramount importance, as they influence the generated output. Minor alterations to the input prompts can result in markedly different outputs. Furthermore, changes that are not immediately apparent to humans may have a significant impact on the model. It is therefore of the utmost importance to ensure that instructions are clear, precise and delivered in a well-paced manner. (Fulford and Ng, 2023)

Some of the techniques which can be employed to generate these prompts are as follows:

- Use delimiters to clearly distinguish between different sections of the text. The term "delimiter" is used to describe any punctuation that serves to clearly distinguish one section of a text from another. Potential indications may include the following: triple quotation marks (""), triple dashes (—), angular brackets (<>), and others.
- Request the output in a specific structure, such as a JSON document with defined fields. This would allow the output to be used as a dictionary in a Python function.
- Verify that the prerequisites for the performance of the task have been satisfied.
- Provide examples of how to complete the task and then ask the model to perform the task.
- Specify the steps required to complete the task.
- Ask the model to work on its solution before any conclusions are drawn.

2.2 Models used during the project

LLMs can be classified according to whether they are open-source or closed-source. An open-source model can be used freely and openly. Such models are accessible to the general public and can be adapted and tailored to suit specific requirements, given that their underlying architecture is publicly available. As they are open models, they are transparent, which allows the community to propose constant improvements to the quality and security of the codes. In contrast, closed-source models are also a possibility where the code remains private. The community is therefore limited to the information disclosed by the owner company, which is primarily concerned with the business requirements. (Ahmed et al., 2024)

The utilisation of LLM models is associated with a significant computational cost and necessitates the allocation of substantial memory and GPU resources. This is the reason why large open-source models cannot be run on a conventional computer. One alternative is to connect to the models through an Application Programming Interface (API), which enables the user to make requests to the models and receive responses without the need to install the models locally. To establish a connection with a model through an API, it is necessary to be registered with a model provider and to possess an API key. In this study, we aim to utilise both closed-source models and open-source. To this end, we used two distinct model providers. On the one hand, we have "OpenAI", and on the other hand, "Together.ai". (*openai API* 2024 and *together.ai API* 2024)

The closed-source models selected were "gpt-3.5" and "gpt-4" due to their popularity and capabilities. Given that we were working with a conventional computer, we were unable to select any open-source LLM and run it. Accordingly, we sought the assistance of the provider "together.ai". This platform offers access to a substantial number of open-source models at a reduced cost. Among the available models, we have selected those with a greater number of parameters and those specialised in tasks related to implementation or code understanding. The selected models are as follows: "Llama-2", "Llama-3", "Codellama", "Qwen1.5", "Depseek-code" and "Platypus2".

2.2.1 GPT-3.5

This is a closed-source model developed by OpenAI. The "gpt-3.5-turbo" version of the model has been considered. The model has been trained with information up to September 2021. The aforementioned model permits the understanding and generation of natural language and code. (*gpt-3.5-turbo* 2022)

2.2.2 GPT-4

This is a closed-source model developed by OpenAI. In this study, we have considered the "gpt-4" version of the model, which has been trained with data up to September 2021. The GPT-4 model can solve more difficult problems with higher accuracy than the GPT-3.5 model. (gpt-4 2023)

2.2.3 LLaMA-2

This is a family of models, ranging from 7B to 70B parameters, developed by Meta. The Llama-2-70B-chat-hf model, which is the most complex of the family, will be the subject of our study. The model has been trained to generate dialogue in a more secure manner than the base Llama-2 model. The model has been trained on publicly available online data on a range of topics. The model has a contextual window of 4,000 tokens and was trained between January 2023 and July 2023. In the pre-training phase, 2 trillion tokens of data from publicly available sources were utilised. In the finetuning phase, datasets with public instructions as well as human-annotated examples were employed. In the pre-training phase, data up to September 2022 was utilised, while in the finetuning phase, data up to July 2023 was employed (Touvron et al., 2023). For the sake of simplicity, we will refer to this model as *llama2* throughout this project.

2.2.4 Code Llama Instruct

This is a proprietary model developed by Meta. This model has been constructed upon the LLaMA-2 framework through a process of fine-tuning, enabling the generation and discussion of code utilising code-specific datasets. The model is capable of both code generation and natural language response generation about a given code. The model with 70B parameters, trained on 1T tokens to generate the model and subsequently on 500B tokens of code and code-related data, was considered. The Code Llama Instruct version has undergone additional finetuning, which has enhanced its ability to comprehend the desired outcomes of prompts. Code Llama Instruct has undergone fine-tuning to generate responses in natural language that are both helpful and safe. (Roziere et al., 2023). For the sake of simplicity, we will refer to this model as *codellama* throughout this project.

2.2.5 Llama3

This is an updated version of the LLaMA-2 model, which was launched in April 2024. This model offers new capabilities and improvements, including enhanced reasoning capabilities in code generation and in following instructions. LLaMA-3 is a decoder-only structure, as is the case with all models in the Llama family. However, it employs a 128K-tokens tokeniser, enabling it to encode the language with greater effectiveness. The pre-training phase was conducted on 15 T tokens, all of which were sourced from publicly available data sources. (AI@Meta, 2024)

2.2.6 Platypus2-70B-instruct

The model was created by combining garage-bAInd's Platypus2-70B and Llama-2-70-B instruct. It should be noted that this is a fine-tuning of LLaMA2, which has been trained on a set of STEM and logic data. (Lee, Hunter, and Ruiz, 2023). For the sake of simplicity, we will refer to this model as *platypus* throughout this project.

2.2.7 Qwen1.5.Chat

This model is licensed by Tongyi-QIANWEN-Research. The 72B version was selected from the family of models. The architecture of this model follows the decoderonly approach and is based on the Llama model family. In particular, supervised finetuning (SFT) has been performed to align the model to human preferences. The model has been pre-trained on 3T of text diversity tokens in both English and Chinese, as well as codes. The data utilised in the pre-training process encompasses language capabilities and advanced skills, including those about arithmetic, coding and logical reasoning. (Bai et al., 2023). For the sake of simplicity, we will refer to this model as *qwen* throughout this project.

2.2.8 Deepseek Coder Instruct

This is a model developed by Deepseek. The base model is "deepseek-coder-33bbase" and has undergone further finetuning through the incorporation of 2B instruction data tokens. The base model has been trained on 2T tokens from 87 programming languages, thereby ensuring an understanding of the code and its syntax. A total of 87% of the tokens are code, with the remaining 13% being natural language in English and Chinese. The architecture is decoder-only, and both the next token prediction and fill-in-the-middle strategies have been employed during training. The data utilised in this study was sourced from public GitHub repositories created before February 2023. (Guo et al., 2024).For the sake of simplicity, we will refer to this model as *deepseek* throughout this project.

Chapter 3

Methodology

3.1 Introduction to Classic Problems used in the Experimental Analysis

As previously stated in the motivation section, the focus of this study is on classical problems that could be taught in an introductory algorithmics course. The selected problems are the Sieve of Eratosthenes algorithm, the Euclidean algorithm, the Binary search algorithm, and the Knapsack algorithm. It is challenging to objectively rank them from easiest to hardest. Our criteria for saying that one problem is harder than another is based on how familiar students are with what the algorithm computes and the complexity of the methods under study. For instance, the Sieve of Eratosthenes algorithm and Euclidean algorithm can be considered the easiest. These are methods for determining prime numbers and the greatest common divisor of two numbers, and students are familiar with these processes from primary school. When we start studying algorithms, we already know what it means for a number to be prime or how to calculate the greatest common divisor of two numbers. However, students may be unfamiliar with the Sieve of Eratosthenes algorithm algorithm or the Euclidean *algorithm*. The other two methods, although intuitively simple to solve, may never have been encountered by students from an academic point of view. The binary search is an efficient way of finding a number in an ascending list, and the Knapsack algorithm is an optimisation problem with constraints.

These methods are well established, and the selected LLM models are likely to have a substantial corpus of information about them.

We will now examine each method in detail, to refresh our understanding of its particular characteristics, before proceeding to explain how we approached the experiments we conducted on them.

3.1.1 Sieve of Eratosthenes algorithm

The Sieve of Eratosthenes represents an elegant and efficient method of computing prime numbers. The fundamental concept is straightforward. A given number can be classified as either prime or composite. A number is defined as a prime if it has only two possible divisors, namely 1 and itself. To identify all the primes less than a given number, one may consider a list comprising all the numbers up to that number and exclude all the multiples of the primes that have already been considered. In other words, all multiples of 2 must be discarded, and the next number on the list, which is 3, must be considered the next prime. All multiples of 3 up to the number being considered must then be discarded, and the process is repeated for each subsequent prime. The distinctive feature of this methodology is that it ceases when it encounters the first number that is greater than the square root of the given number. This is because, from this point onwards, if the number in question is a multiple of

a larger number, it is also a multiple of a smaller number that has already been discarded. Furthermore, the list of all the prime numbers that are smaller than a given number is already available.

3.1.2 Euclidean Algorithm

The Euclidean method is employed to identify the greatest common divisor of two given numbers. Euclid demonstrated that to ascertain the greatest common divisor of two given numbers, it was not necessary to identify all the divisors of each of the numbers and to subsequently identify the common divisors. He demonstrated that if the objective is to identify the greatest common divisor of two integers, it is sufficient to consider the smaller of the two and the remainder of the division of the larger by the smaller. Upon iteration, the remainder will be zero. Since the greatest common divisor between a number n and 0 is n, we may conclude that this is the greatest common divisor we are looking for.

3.1.3 Binary search

The objective of the binary search algorithm is to identify a specific element within an ascendingly sorted list without having to examine every element in the list. The concept is to identify the element situated at the median position within the ordered set and determine whether the target value is equal to, less than, or greater than the found value. This information is used to determine whether to terminate the search or to repeat the same process in the sub-interval to the left or right. If the element is present in the list, it will be identified. If the element is not found, it can be concluded that it is not present in the list, given that the list is finite and ordered.

3.1.4 Knapsack Algorithm

This is a classic optimisation problem where there are n objects, each with a weight and a value. The objective is to select a subset of objects from the set of n objects, each with a weight and a value, that simultaneously satisfy two conditions. The first condition is that the sum of the weights of all the selected objects must be less than or equal to the maximum capacity of the backpack. The second condition is that the value of the selected objects must be the maximum possible. The objective of the problem is, therefore, to identify the objects that should be placed in the backpack and to quantify the benefit of selecting these objects.

3.2 Experimental analysis 1: Obtaining code that implements a described algorithm

The objective of this experiment is to place ourselves in the position of students who request a code from the model that will perform a specific action. To achieve this, two distinct strategies will be employed. In the initial strategy, we provide the model with an explanation of the algorithmic process, employing mathematical reasoning. In the second strategy, the pseudocode of the algorithm is described, and the model is then asked to generate a code with the correct syntax.

To identify areas where the model is particularly effective and those where it is less so, we have tried to consider a particular aspect of the task. To this end, we considered how an individual might incorporate code that executes a specified action. We concluded that an implementation can be either recursive or iterative. It is typical for students to create iterative algorithms at the beginning of their programming journey. However, they are also capable of doing recursive implementations if requested to do so. The objective is to analyse the behaviour of LLMs. In the absence of a specified approach, it is necessary to identify whether they will implement an iterative or recursive solution. Furthermore, if we direct the model to produce a recursive implementation, is it capable of doing so? To test this hypothesis, the first strategy comprises two prompts that are highly analogous. The sole distinction between the two is that the first does not stipulate how the method is to be implemented, whereas the second specifies that it should be recursive. The prompts have been designed to be introduced to a language model by a student. Consequently, they are not preoccupied with minor details and simply explain what the learner is attempting to achieve.

In each prompt of this initial strategy, we avoid mentioning the name of the algorithm. At the end of the prompt, we request that the model indicate whether the method described has a name. The objective is to determine the ease or difficulty with which each model can identify this information. Our objective is to obtain code that is both functional and compliant with the specifications outlined in the instructions. This will permit the analysis of instances where code has been provided that performs the algorithm under study, although not in the exact manner requested. Furthermore, we do not request that the model justify or explain the code. Therefore, if the model does so, it is because this is a characteristic feature of the model.

In the second strategy, it is assumed that the student has a pseudocode of the desired implementation. The design of the prompts is based on the a priori assumption that the student will require assistance in implementing the code. This is achieved by providing the model with a list of steps to follow by the pseudocode generated by the student.

We have posed the following questions that we want to evaluate:

- 1. Does the model recognise the problem?
- 2. Does the model produce working Python code?
- 3. Does the model explain the code?
- 4. Does the model reason correctly (if it provides any kind of reasoning)?
- 5. Does the model produce recursive code?
- 6. Does the model perform the steps requested in the description?

As not all challenges are designed in the same way, the responses to these questions are of greater value in some challenges than others. The objective is to implement functional code generated from the descriptions provided in the prompts. Consequently, the responses to questions 2 and 6 will be evaluated for all prompts executed.

We only want to look at recursion in the prompts generated by strategy 1, to see what happens when recursion is not explicitly requested and when the model is instructed to generate recursive code. Consequently, the response to question 5 will be evaluated exclusively based on the prompts associated with strategy 1. Similarly, the response to question 1 will be analysed solely about the prompts of the first strategy, as this is the only instance where the question is directly posed.

The following section will examine the various prompts that have been formulated for each of the problems, with a particular focus on the two strategies under consideration. Please refer to Appendix A for a comprehensive list of all the prompts referenced in this section.

3.2.1 Strategy 1: Prompts to implement the code from mathematical reasoning

For each of the four algorithms previously described, two prompts have been designed that are essentially identical. The principal distinction between the two prompts lies in the manner of their formulation. In one instance, the model is explicitly requested to provide a recursive code, whereas in the other, this is not the case. The prompts are structured uniformly. Firstly, the algorithm under consideration is explained in detail, after which the model is asked to provide a code that performs the described function. In addition to the code, the model is also required to respond to the following question: "Does this algorithm have a name?". The question is posed to determine whether models possess sufficient information to identify algorithms when describing their functionality. Furthermore, this question is posed to determine whether, once the algorithm for which the code is required has been identified, the models provide the code as described, or merely provide code that implements the algorithm in question.

Accordingly, one of the questions we have posed is whether the model performs the steps requested in the description. In this context, the term "steps" refers to the specific details mentioned in the prompt that are both distinctive and enable us to ascertain whether the model has generated the code by the provided description or based on its prior knowledge.

The steps to implement the *Sieve of Eratosthenes algorithm*, as described in the prompts of figures A.1 and A.2, are as follows:

- 1. *x* is considered to be the number up to which the primes are to be computed.
- 2. all multiples of primes are eliminated until the square root of *x* is reached.

The figures A.3 and A.4 contain the prompts of the *Euclidean algorithm*. In this instance, the requisite steps involve the consideration of two numbers, a and b, and the subsequent computation of the greatest common divisor, which is not performed directly on these numbers but rather on x and y. In this case, x is defined as the maximum value of a and b, while y is defined as the minimum value of a and b.

In the case of the *Binary search algorithm*, we refer to the figures A.5 and A.6. In this algorithm, the element is located by dividing the intervals into two parts. If the element is not found, the result should be a - 1.

Finally, we find the *Knapsack algorithm* in the figures A.7 and A.8. To implement this method, it is necessary to define two variables: *p*, which represents the list of values associated with the *N* objects, and *w*, which is a vector of *N* elements, each of which represents the weight of the corresponding object. The algorithm is then queried to return two elements: the sum of the selected elements of p and the indices of p corresponding to the selected elements.

3.2.2 Strategy 2: Prompts to implement the code from a pseudocode description

For the same four algorithms, we have designed a prompt for each of them, in which the student provides a step-by-step explanation of the pseudocode developed, to implement the code. In all the prompts, we first provide context to situate the model within the relevant context. We then proceed to outline the steps that must be followed to implement the code. The steps are highly detailed as if the student were describing the pseudocode they had created to implement the Python code for it. The prompts for the methods *Sieve of Eratosthenes algorithm, Euclidean algorithm, Binary search algorithm* and *Knapsack algorithm* can be found in figures A.9, A.10, A.11 and A.12 respectively.

3.3 Experimental analysis 2: Code interpretation and error detection

As previously discussed, it is beneficial to adopt the perspective of the students to gain a deeper understanding of their learning process. Once the students have implemented a solution to the problem, it may be correct or it may contain syntax or conceptual errors. If students have doubts about their code, they will want to seek assistance before submitting their answers. It can be challenging for teachers to provide support to all students, given the limited time and resources available. One strategy that has been employed in numerous subjects is the establishment of online forums where students can pose their doubts and provide assistance to one another. In such instances, the teacher's role is primarily that of a facilitator, intervening only when necessary. While this approach is beneficial, there are instances where more personalised assistance may be required.

The objective of this study is to examine the potential of LLM technology for the execution of specialized tasks. In this section, we consider a scenario in which educators wish to develop a tool to assist students in becoming more self-taught. Assuming that the tool employs generative AI, we have designed prompts that could be executed on the tool when the learner enters the code to be evaluated. Two of the primary areas in which students require feedback are the functionality of their code and the presence of syntax errors. Consequently, it is assumed that the tool will be capable of providing this information, and prompts must therefore be designed to extract it from the code. It is of critical importance to consider how instructions are conveyed to the models. The focus is on defining the desired outcome, as well as the how and the expected format of the output. It is important to note that language models have limitations in terms of the length of prompts. Rather than building one very complex prompt that would provide us with all the information we were interested in, we decided to build two prompts.

The first prompt has been devised in such a way that a student is required to input a code, with the model then indicating the specific function of the code. If the student inputs a code containing conceptual errors, it would be intriguing for the model to indicate that the student is unaware of the model's functionality. It would also be of interest to evaluate the response of the model when errors are introduced at the level of syntax. The objective is to ascertain whether the models will evade the introduction of a code with syntax errors and respond as if there were no such errors, or whether they will be unable to identify the programming language in which the code is written. For this reason, the initial prompt is designed to ascertain the language in which the provided code is written and to identify the code's intended purpose.

If a student submits a code containing syntax errors, they will require assistance in resolving these errors without the need to get an entire code corrected. In the second prompt, we will concentrate on the syntactic errors present in the submitted code.

In both prompts, the model must attend to the code under analysis, the desired output format, and the steps required to achieve this output. The prompts assist the model in reasoning and limiting errors or hallucinations. To facilitate the model's understanding of the task, instructions were therefore provided in a clear and precise manner. The same structure has been employed in both prompts. Firstly, the model is informed that it should analyse the code by the instructions provided and that its output should be a dictionary with a specific structure. To facilitate the model's ability to distinguish between the code and instructions and the remainder of the prompt, delimiters have been employed. This approach ensures that the model is aware that the code to be analysed is located between the "&" symbols, while the instructions are contained within the "<>" delimiters. The instructions are presented in a sequential order, guiding the model and providing structured instructions.

Three distinct strategies have been employed: zero-shot prompting, one-shot prompting, and two-shot prompting. This implies that the two prompts have been designed in such a way that, in the absence of examples, the models can still perform the task. However, in addition to this, one or even two examples have been included, to enable the model to learn not only from the instructions given but also from examples of inputs and the outputs it should generate.

Furthermore, the language models permit the specification of the system's role. In the absence of any indication, the system assumes its default role. However, it is possible to specify a specific role, thereby informing the model of the expected behaviour for each activity. Consequently, for each of the generated prompts, both the modified and unmodified system roles were executed.

We will now proceed to analyse each of the specific tasks. Please refer to Appendix C for a comprehensive list of all the prompts referenced in this section.

3.3.1 Experimental Analysis 2.1: Code objective

In this section, we will analyze the inputs introduced to the model when we intend to obtain the target of the code. As previously stated, we are going to analyze the responses given by the model in two distinct scenarios: when the system role is modified and when it is not.

In the experiments where the system role has been modified, the following description has been provided:

.....

Given a code, you're an expert at analyzing it to identify its theme, and if it contains a specific method, you mention its name for a clearer target. Follow the <INSTRUCTIONS>.

In this description, it is emphasised that the model must determine the objective of the code by following the instructions given at the user prompt. Given that we are dealing with algorithms that have been given a name, we would like the model to indicate it in the answer. For instance, Euclid's algorithm calculates the greatest common divisor. Consequently, the answer should indicate that it employs Euclid's method. Similarly, the Eratosthenes algorithm calculates prime numbers. However, the answer should not merely state that it calculates prime numbers; it should also indicate that it employs Eratosthenes' method.

The experiments are conducted with the system role modified by the aforementioned comment and with the default system role. In the user role, we consider zero-shot, one-shot, and two-shot prompting. The prompts are designed to be completed with a code entered by the user. In this instance, the user is expected to input a code that implements one of the four algorithms presented in the preceding sections. The examples added to the prompt are elementary mathematical problems. If we provide a single example, we are referring to an algorithm that provides the actual solutions to the equation $c \cdot x^2 + b \cdot x + a = 0$, given the specified parameters of (a, b, c). If we provide two examples, we also include an additional example illustrating the calculation of the i-th term in the Fibonacci series using the golden ratio. By indicating that the calculation is performed using the golden ratio, we intend that the model appreciate the significance of the method employed in the algorithm, rather than focusing solely on the outcome.

Figure C.1 illustrates the zero-shot prompting that has been designed. Figure C.2 depicts the one-shot prompting. Figure C.2 combined with C.3 provides the two-shot prompting.

In the prompt illustrated in Figure C.1, we direct the model to construct a dictionary with the fields "code_language" and "code_objective". In the first field, the model is instructed to enter the language of the code. In the second field, a concise description of the purpose of the code must be provided. To prevent the models from providing responses that are either invented or irrelevant, we stipulate the required inputs in the event of an unidentified answer. Finally, we stipulate that the dictionary is the only acceptable answer, without any supplementary information.

The answers will be evaluated by the criteria outlined in the aforementioned prompt. Firstly, the format of the given answers will be analysed, as the prompt states that only the dictionary should be provided. Subsequently, the responses that adhere to the format will be subjected to an examination of the values assigned to the fields "code_language" and "code_objective." The responses to the "code_language" can be classified into two categories: those that identify the language and those that do not. The remaining field, "code_objective", can be divided into three categories: those for which the model has not identified the target, those for which the model has provided an incorrect target, and those that correctly identify the target. To ascertain whether the modified system role or the Fibonacci example has taken effect, we shall examine the instances where the algorithm method has been mentioned.

3.3.2 Experimental Analysis 2.2: Code syntax error

Once the student is aware of the intent behind the code and is satisfied with the code structure and logic, the next step is to identify potential issues related to the code's syntax. Similarly to the approach taken in the prompt to ascertain the code's objective, we will analyse the responses generated by the model in both the modified and default system role scenarios. Additionally, experiments have been conducted utilising zero-shot, one-shot and two-shot prompting with the same two examples, incorporating additional syntax errors.

In the experiments where the system role has been modified, the following description has been provided.

.....

You are an AI expert in detecting if a code has syntax errors and in case of having them explain which ones following the <INSTRUCTIONS>."

In this description, it is emphasised that the model must be an expert at detecting errors by following the instructions given at the user prompt.

Figure C.4 illustrates the zero-shot prompting that has been designed. Figure C.5 depicts the one-shot prompting. Figure C.5 combined with C.6 provides the two-shot prompting.

In the prompt illustrated in Figure C.4, we direct the model to construct a dictionary with the fields "code_sintax_error" and "code_sintax_explanation". In the first field, the model is instructed to enter if there are syntax errors or not. To ensure uniformity, responses are limited to "Yes" or "No". In designing this prompt, we aimed to establish a connection between the prompt's response and the objective prompt described above. To this end, the code language was specified as –code_language–. As in all the examples that have been evaluated, the language used was Python. In this instance, the prompt was adjusted to "–code_language–:–Python–". If the preceding response is affirmative, it is necessary to identify the errors to facilitate their correction by the student. Finally, we stipulate that the dictionary is the only acceptable answer, without any supplementary information.

The answers will be evaluated by the criteria outlined in the aforementioned prompt. The responses will be classified as either valid or invalid and subsequently analysed separately for each model. A response is considered invalid if it is based on information from the examples and not on the code entered, if it contains additional information apart from the requested dictionary if it identifies false errors, if it gives incorrect corrections, if it does not follow the given instructions, or if it provides in its response codes from the training of the model that have no relation to the task to be performed. A valid response is one that, in the event of errors, identifies them and provides details thereof; alternatively, in the absence of errors, it should indicate that no such issues exist.

Figures C.5 and C.6 illustrate the examples introduced in the one and two-shot prompting. These examples address the same two issues previously discussed, but to align them with this task, we have incorporated syntax errors. In the solution of the second-degree equation, the symbol "=" is employed in place of "==" for comparison. In the example of finding the i-th term of the Fibonacci sequence, an erroneous syntax was employed, "round" was incorrectly written as "rount". Furthermore, all lines were terminated with a semicolon which is not correct in Python.

3.3.3 Experimental analysis 2: analysed codes

All prompts in Appendix C have been designed to be evaluated by entering a code. To ensure consistency across all experiments, we have considered some of the codes that might have been implemented by students in response to some of the descriptions presented in Experimental Analysis 1. The intention is to introduce diverse situations, and not all codes are correct. The analysis will encompass correct codes, codes exhibiting conceptual errors, and codes displaying syntax errors. A conceptual error is defined as an error that involves the use of incorrect variables or the performance of improper operations. Such errors result in the code failing to perform the intended function. In contrast, there are also syntax errors. This type of error results in execution errors due to the erroneous grammar of the programming language employed.

The Appendix D, contains a comprehensive list of all the codes used. This section will provide detailed information about the codes used for each algorithm, including any errors that may have been introduced.

Sieve of Eratosthenes algorithm

In Figure D.1 and D.2 illustrate two correct implementations of the Sieve of Eratosthenes algorithm. In Figure D.4 we observe an iterative version of the code that exhibits a syntax error. This error is characterised by the use of "append{}" instead of "append()". Figure D.3 contains a code that exhibits both conceptual and syntax errors. The conceptual error is that the order of the inputs in the recursive call of the function has been altered.

Euclidean algorithm

Figure D.5 illustrates a recursive implementation of the algorithm without errors. Figure D.6 depicts an iterative implementation of the algorithm that exhibits a conceptual error. The error lies in the calculation of the greatest common divisor, whereby the quotient resulting from the division of the largest number by the smallest is considered, rather than the remainder. Figure D.7 illustrates a recursive implementation of the algorithm that exhibits four distinct errors. The first error is the use of a different name for the function than that used in the definition. The second error is the replacement of the comma with a semicolon. The third error is the replacement of the comma with a semicolon. The third error is the replacement of the assignment operator. The final figure for this algorithm is Figure D.8, which presents an iterative implementation of the algorithm with two errors. The first error is the substitution of "int" for "ent". The second error is the use of "=!" to check if two values are different instead of "!="

Binary search algorithm

Figure D.9 depicts an iterative implementation free of any errors. Figure D.10, illustrates another iterative implementation that contains three syntax errors. These include the use of ";" instead of "," in the assignment of variables, the elimination of all ":" and, finally, the indistinct use of "D" or "d" in a variable.

Knapsack algorithm

Figure D.11 depicts a recursive implementation without errors. Figure D.12 illustrates a recursive implementation that exhibits both conceptual and syntax errors. The conceptual errors are as follows: firstly, the order of two of the inputs was altered when making the recursive call; secondly, the value "n-1" was incorrectly entered as "-1". The syntax error is the use of the variable "wt" or "Wt" without distinction.

3.4 Technical aspects of the experiments

We performed all experiments by connecting to the models we were analysing via API. For the closed-source models, we used the openai provider, while for the open-source models, we used together.ai.

For both open-source and closed-source models, several hyperparameters can be used to control certain aspects of the generated response. Firstly, when a request is made to start a conversation with the LLM, the messages that are part of the conversation so far must be included. Each of these messages has a role and content associated with it. The role determines who introduced the message and the content is the message itself. When we change the system role, we are telling the model how to behave and what tone to use in its output. By introducing a message with the role assigned to the user, we tell the model that it is interacting with the end user. Apart from the messages, it is important to determine which model we want to communicate with. There are other hyperparameters that we can modify that will determine how the output will be. Among them we find (*Chat completion openai* 2024 and *Chat completion together.ai* 2024):

- max_tokens: A token is the minimum unit of measurement for LLMs. Note that the price of queries depends on the number of tokens entered and generated, so it may be interesting to enter the maximum number of tokens you want the model to generate.
- temperature: determines the randomness of the model responses. It is a value between 0 and 1, the closer to 0 the lower the randomness and the closer to 1 the higher the randomness.
- top-p: Alternative to temperature sampling. Select the minimum percentage of relevance that tokens must have to be considered in the model. The value of this parameter also ranges from 0 to 1.
- top-k: limits to k the token options for the next predicted word.
- frequency penalty: Value between -2 and 2 where a positive value decreases the probability of repeating tokens mentioned above.
- repetition penalty: Number that controls the diversity of the generated text. Higher values reduce the likelihood of repeated sequences.

As mentioned above, we ran experiments with and without changing the system role. The other hyperparameters were not changed in the experiments and therefore we kept the default hyperparameters of each of the models. In the openai platform the default hyperparameters are:

- temperature = 1
- $max_tokens = 256$
- $top_p = 1$
- *frequencypenalty* = 0

In the together.ai platform the default hyperparameters are:

- $max_tokens = 512$
- temperature = 0.7
- $top_p = 0.7$
- $top_k = 50$
- repetition penalty = 1

Chapter 4

Results

4.1 Experimental analysis 1

This section will present and analyse the results obtained from the execution of each prompt presented in Section 3.2. A total of four problems are under study, with three prompts designed for each. Eight models are being evaluated in total. Given the inherent randomness of LLM responses, each prompt was run three times. Accordingly, we have generated 288 data points, which form the basis of the subsequent analysis.

The two questions are as follows: "Does the model produce working Python code?" and "Does the model perform the steps requested in the description?".

The objective is to identify those models that are more suited to generating the requested code when going from mathematical reasoning to code (Strategy 1) than when going from pseudocode to code (Strategy 2). Consequently, the percentage of responses that generated functional code and the percentage of responses that followed the steps described in the prompt for each of the strategies were compared. The percentages were calculated for each model, strategy, and question.

	% of Responses wit	th Functional Code	% of Responses that follows the steps		
Model	Strategy 1	Strategy 2	Strategy 2 Strategy 1		
llama3	100.0	100.0	75.0	100.0	
gpt-4	100.0	100.0	75.0	100.0	
gpt-3.5	91.7	91.7	87.5	91.7	
codellama	87.5	100.0	37.5	100.0	
deepseek	75.0	75.0	37.5	75.0	
llama2	25.0	75.0	12.5	50.0	
platypus	62.5	25.0	50.0	25.0	
qwen	87.5	50.0	75.0	50.0	

TABLE 4.1:	Percentage of	of satisfactory	responses	grouped	by	model
		and strate	gy _			

Table 4.1 reveals that the models can be classified into two broad categories. Those that have demonstrated a greater ability for moving from pseudocode to code than for generating code from mathematical reasoning include *llama3*, *gpt-4*, *gpt-3.5*, *codelama*, *deepseek* and *llama2*. In contrast, the remaining two models, *platypus* and *qwen*, demonstrated a higher success rate in generating correct code when prompted to do so using mathematical reasoning.

A comparison of the answers generated with the same strategy reveals that the percentage of answers that produce a functional code is higher or equal to the percentage of answers that produce a code that adheres to all the steps described. This phenomenon occurs because, on occasion, the model produces code that effectively responds to the algorithm under study, yet the code differs slightly from the requested output. The non-compliance with the steps outlined in the prompt does not necessarily indicate that the model is unable to reason and execute the described steps. This demonstrates that, in response to these prompts, the model occasionally prioritises other aspects over strict adherence to the prescribed steps. For instance, in the case of *gpt-4* and *llama3* models, some of the generated answers did not fully adhere to all the prescribed steps. Let us undertake an analysis of one such response, generated by *gpt-4*. Figure B.1 illustrates a response generated by *gpt-4* in response to the prompt presented in Figure A.3. It can be affirmed that this response fails to adhere to all the prescribed steps, as it assumes that *a* is greater than *b*. However, in Figure B.2, we observe an alternative response generated by the same model and under identical conditions, in which all the steps are followed. In the case of *llama3*, it has been unable to generate recursive codes when explicitly requested in the Euclid and Eratosthenes problems. However, it does comply with all the other steps, as demonstrated in B.3.

We now turn our attention to the disaggregated answers, organised according to the algorithms in question and the models. The objective is to identify instances where the expected answer was generated and instances where it was not. Given that three iterations were performed, it is possible that in one or two iterations the answer was correct and in the others, it was not, as was observed in the previous example of *gpt*-4. This idea will be illustrated by means of a colour map in the graphs from Figures 4.1, 4.2, 4.3, 4.4 and 4.5. Each cell represents the responses generated by a specific prompt and a particular model. A green cell indicates that the condition under study has been met in three out of three iterations. If the cell is yellow, this indicates that the condition was met in two of the three iterations. If the cell is orange, this indicates that the condition was met in one of the three iterations. Finally, if the cell is white, this indicates that the condition was not met in any of the iterations. Figures 4.1 and 4.3 illustrate the models that have been able to generate a functional code for each iteration. The visual representation in Figures 4.2 and 4.4 facilitates the identification of the models that have been able to comply with the steps described in the prompt. In Figures 4.1 and 4.2, we can observe that:

- The *llama2* model is unable to generate a functional code based on a description that employs mathematical reasoning.
- The mathematical representation of the *knapsack* problem presents certain challenges for some models, making it more difficult to achieve a functional code. This may be attributed to the complexity of the problem.

In Figures 4.3 and 4.4 we can observe that:

- The *platypus* model is unable to generate functional code by the instructions provided in the prompt, which describe the code in the form of pseudocode.
- In the context of the *Binary Search* problem, when described as a pseudocode, the *deepseek*, *llama2*, *platypus* and *qwen* models are unable to generate a functional code following the prescribed steps. The models are primarily confused when attempting to return the position of the searched element within the array. In numerous instances, the position of the subarray where the element has been found is returned, yet this response is of limited utility in the absence of the subarray. An illustrative example of this phenomenon can be observed in Figure B.6, where we present the response generated by the *qwen* model when

executing the prompt in Figure A.11. If we simply execute the model without examining the results in detail, it appears to be functioning correctly. However, if we modify the input, for instance by setting a = 9, we observe that the model returns a value of 1 instead of 4. These models seem to exhibit greater difficulty than *llama3*, *gpt-4*, *gpt-3.5* and *codellama* in reasoning and interpreting the instructions they are given.

Figure 4.2 illustrates the difficulties encountered by the models in generating codes that adhere to the prescribed steps in the *Eratosthenes* and *Euclid* problems. This hypothesis is aligned with the proposition that, given the vast quantity of information present in the models, they may offer an implementation of the same once they recognise that they are confronted with such a problem, without undertaking a detailed analysis of the prompt in question. Figure 4.2 illustrates also that requesting a recursive implementation of the aforementioned problems may present a challenge to the models.

In the absence of a clear indication from the user as to whether recursive or iterative code is desired, all models have generated iterative code. The only exception to this rule has been the *deepseek* model, which has generated recursive code for the *Binary Search* algorithm.

Figure 4.5 illustrates the generation of recursive codes when explicit requests are made at the prompt. About the *Binary Search*, it can be observed that the generation of recursive codes has been consistently achieved, although as illustrated in Figure 4.1 they have not always been functional. The *Eratosthenes* problem has presented the greatest challenge in generating recursive codes, even though in all models except *llama2*, functional executions of the code have been generated. This indicates that the focus of the model has not been on the fact that a recursive code was requested.

In all instances where Strategy 1 was employed, the model was asked if the method described had a name. In every case, the model correctly identified the algorithm in question.

Despite not requesting such information directly, we have identified that some of the responses provide reasoning in support of the code, while others offer explanations or commentary on the code itself. As we did not request this information, we presume that it is a feature of the models. As illustrated in Table 4.2, the *gpt-4* model exhibits the strongest tendency to explain the code, with the *codellama*, *llama3*, and *deepseek* models following in order. In addition to providing explanations for the code, the models also tend to reason out their answers.

Model	% of responses that explain the code	% of responses thet provide a correct
		reasoning
llama3	75.0%	66.7%
gpt-4	94.4%	72.2%
gpt-3.5	8.3%	0%
codellama	83.3%	55.6%
deepseek	41.7%	41.7%
llama2	33.3%	8.3%
platypus	0%	0%
qwen	66.7%	50.0%

 TABLE 4.2: Percentage of responses by model that have commented on the generated code or provided correct reasoning

It is important to note that *gpt-3.5* did not provide erroneous reasoning, but it demonstrated a lack of reasoning present in the answers. In contrast, the results from *llama2* were characterized by incorrect reasoning. To illustrate, we may consider the responses generated by both models to the prompt presented in Figure A.4.

Figure **B.5** illustrates the response of *gpt-3.5* to the *Euclidean* problem when prompted for recursion directly. The response is recursive and performs the requested operation, but does not include additional reasoning or comment code, which is appropriate given that this was not a direct request.

Conversely, Figure B.4 depicts the response provided by *llama2* to the aforementioned problem. It can be observed that this response is not recursive and contains erroneous statements, such as "so we can see 12 is a divisor of 18" or "it works by repeatedly subtracting the smaller number (b) from the larger number (a) until b is 0."



4.2 Experimental Analysis 2

In this section, we will analyse the results of the experiments presented in section 3.3. The responses of the two experiments will be analysed: the one to obtain the target code presented in section 3.3.1 and the one to obtain the syntax errors presented in section 3.3.2.



FIGURE 4.5: Prompts from strategy 1 that generate a recursive code when asked

For each of the experiments, we evaluated the zero-shot, one-shot, and two-shot prompting, both with and without modifying the system role, resulting in six different prompts. The prompts were completed with the 12 codes from Appendix D, resulting in 72 different inputs for each of the models. In the previous experiments, each input was executed three times. In this experiment, however, we have decided to increase the number of executions of the open-source models to five, as they have a lower associated cost. In contrast, the closed-source models will be executed three times. This results in a total of 2,592 responses to analyse for each of the two experiments. It should be noted that the open source models are *llama3*, *llama2*, *codellama*, *deepseek*, *platypus* and *qwen*, while the closed source models are *gpt-3.5* and *gpt-4*.

4.2.1 Results of Experimental Analysis 2.1

The objective of this section is to analyse the results presented in section 3.3.1 The model *codellama* is unable to perform this type of activity correctly. A total of 97.2% of responses indicated that the model could not perform the requested task due to ethical considerations. Figure E.1 illustrates an example of a response. The remaining 2.8% of responses were not in the desired format. This is why the *codellama* model is rejected.

The responses of the remaining models will now be analysed. The format of the responses was found to be incorrect in 9.8% of cases, which equates to 219 answers out of a total of 2, 232. The analysis revealed that 214 of the erroneous responses were generated by the *llama2* model. The remaining five iterations correspond to the five runs of the *deepseek* model when the system role was not modified by running zeroshot prompting in Figure C.1 with the error-free *Knapsack* code in Figure D.11. Figure E.2 illustrates the incorrect format of the given answer. However, the content of the dictionary is accurate. Upon comparison of the results obtained from executing the same process in the model *llama2*, it has been observed that not only does the structure fail, but also the content. This is evident in the figure E.3.

A total of 146 answers were identified as being incorrect format out of 360 executions of the *llama2* model. Upon analysis, it was observed that 6.8% of these responses did not align with the intended functionality of the algorithm. Additionally, 13.7% of the responses provided information about the examples used in the solution, which indicates a potential issue with the correctness of the answer. Furthermore, it was observed that 17.1% of the instances, the code input exhibited a conceptual inconsistency, yet the model responded as if no such inconsistency existed. In light of the aforementioned observations, it can be stated that 37.6% of the responses that

adhere to the prescribed format yield erroneous outcomes. This is why we consider that the llama2 model is also not a suitable model to carry out the task evaluated in this section.

By eliminating *codellama*, *llama2*, and the five erroneous executions of *deepseek*, we are left with a total of 1,867 results for analysis. Thus far, all responses that were not in the correct format have been discarded. The following stage of the process involves the analysis of the content of the responses that have been entered in the correct format. An invalid answer is defined as one which: (a) provides a false response; (b) fails to identify the programming language; or (c) is lacking in clarity regarding the objective of the code. This latter category encompasses instances in which the code entered exhibits a conceptual error, yet the model responds as if there were no such error.

Of the total number of results, 1,867, to be analysed, 931 originate from runs where the code entered contains syntax errors, while 936 are from runs where the code is free of such errors. It is necessary to analyse both scenarios separately, as it is logical that if the code contains errors, the model is uncertain about the programming language. In such a case, it would be incorrect to consider the response incorrect. Conversely, if the code is free of errors, it is incorrect not to detect the language.

Of the executions originating from code with syntax errors, 79.5% identified the relevant programming language; conversely, 20.5% failed to do so. In the case of executions that do not originate from erroneous code, 98.5% of executions successfully identified the underlying programming language.

Table 4.3 provides a detailed overview of the models in question. Notably, the GPT models have tended to fail to detect the programming language on a few occasions, despite the code being correct. It can be observed that these models are less likely to provide the language name in the event of syntax errors, as evidenced by the data presented in the table.

Model	llama3	gpt-3.5	gpt-4	deepseek	platypus	qwen
No syntax error	180 (of 180)	95 (of 108)	107 (of 108)	180 (of 180)	180 (of 180)	180 (of 180)
Syntax error	180 (of 180)	31 (of 108)	39 (of 108)	150 (of 175)	170 (of 180)	170 (of 180)

TABLE 4.3: Number of responses that identify the programming language of the code grouping by model and distinguishing between codes with syntax error and correct codes

We now proceed to an analysis of the responses provided in the code_objective field. In this field, we have identified both incorrect and correct responses. The incorrect responses will now be analysed. The *qwen* model has encountered difficulties when executing the code depicted in Figure D.1. Figures E.4, E.5 and E.6 illustrate the responses generated by the zero-shot, one-shot and two-shot prompting, respectively, without modifying the system role. Modifications to the system role result in largely similar responses. These responses are not accurate because they provide a detailed explanation of a specific code segment, rather than offering an overall understanding of the code.

Another situation of concern is the generation of numerous erroneous responses in the presence of codes with inherent conceptual errors. Out of 463 runs with conceptual errors on 409 occasions, the models have responded as though the codes entered

had no conceptual errors. Such errors are particularly problematic as they are not easily detected. The table **4.4** illustrates the number of responses generated as if the code entered had no concept errors, out of the total number of responses generated with concept errors for each of the models. As illustrated in Table **4.5**, the models have been unable to identify the intended target of the code in the majority of cases. This answer is considered correct as the introduction of a conceptual error to the methods under investigation results in a lack of clarity regarding their functionality. The model *gpt-3.5* is unable to discern the purpose of the code half of all instances. Similarly to the occurrence of conceptual errors that do not identify the objective of the code, there are also instances of other codes that do not identify the objective of the code. A total of 65 with syntax errors failed to identify the purpose of the code. It is reasonable to conclude that these results are acceptable, given that it is not possible to determine whether the identified errors would present a significant challenge in identifying the target. Furthermore, there are 13 instances that do not present any error and correspond to the *Knapsack* problem.

 TABLE 4.4: Responses containing conceptual errors that yield the same responses as those without such errors.

Model	llama3	gpt-3.5	gpt-4	deepseek	platypus	qwen
incorrect	90 (of 90)	27 (of 54)	42 (of 54)	80 (of 85)	80 (of 90)	90 (of 90)
answers)0 (01)0)	27 (01 54)	42 (01 54)	00 (01 05)	00 (01 70)	<i>)</i> 0 (01 <i>)</i> 0)

TABLE 4.5 :	Responses with	conceptual	errors for	which	the purpose
	of the c	ode is not ic	dentified.		

Model	llama3	gpt-3.5	gpt-4	deepseek	platypus	qwen
no ob- jective	0 (of 90)	27 (of 54)	12 (of 54)	5 (of 85)	10 (of 90)	0 (of 90)

This leaves a total of 1,296 responses that have correctly identified the target. The objective is to analyse the percentage of responses that mention the method used in their answer. Although the system role was explicitly modified to include the method of the algorithm, the resulting data indicate that this modification did not yield superior results compared to the default system role executions, as evidenced in Table 4.6.

 TABLE 4.6:
 Responses mentioning the name of the method used in the algorithm

Model	# System role default	# System role modified
llama3	60 (of 135)	70 (of 135)
gpt-4	41 (of 64)	41 (of 71)
gpt-3.5	40 (of 71)	35 (of 65)
deepseek	75 (of 125)	75 (of 130)
platypus	60 (of 130)	60 (of 130)
qwen	90 (of 120)	80 (of 120)

It can be observed that the system role, whether modified or unmodified, did not result in a significant difference. The proportion of responses mentioning the method name is now examined to ascertain whether it is higher when two examples are entered than when one or no examples are entered. Table 4.7 shows that this is the case

for all models except *platypus* and *qwen*

Model	#0 or 1 examples	# 2 examples
llama3	80 (of 180)	50 (of 90)
gpt-4	(47 of 89)	35 (of 46)
gpt-3.5	49 (of 100)	26 (of 36)
deepseek	(100 of 175)	50 (of 80)
platypus	80 (of 170)	40 (of 90)
qwen	115 (of 160)	55 (of 80)

 TABLE 4.7:
 Responses mentioning the name of the method used in the algorithm

A detailed examination of the aforementioned scenarios has enabled us to discern that in both the *Knapsack* and *Binary Search*, the models consistently identify the name of the algorithm. The only model that does not always mention the name of the method is *gpt-3.5*, and sometimes for the *Binary Search*, *qwen* as well. Figure **E**.7 provides an illustrative example of the answer given by *qwen*. The response is accurate in that it provides an accurate description of the model's functionality. However, it fails to mention that the problem in question is *Binary Search*. It can be observed that the examples do influence the generated answer, as evidenced by the fact that the answer initially begins with "define a function," which is also present in the examples. Figure **E**.8 demonstrates that when the prompt is executed without examples on the same code, the answer is also correct. However, it mentions the method in a different style.

About the *Knapsack* and *Binary Search* methods, the name of the method is almost always mentioned, with a few exceptions as previously discussed we are now going to focus on *Erathostenes* and *Euclides*. The objective here is to analyse the outcome of not modifying the system role and providing 0 or 1 examples, or of not modifying the system role while offering two examples or modifying the system within identical scenarios.

	System role de		System role modifie		
Model	0 or 1	2 examples	0 or 1	2 examples	
	examples		examples		
llama3	8.3%	33.3%	25.0%	33.3%	
gpt-4	30.8%	64.3%	20.0%	60.0%	
gpt-3.5	26.5%	63.6%	28.1%	50.0%	
deepseek	36.4%	50.0%	33.3%	50.0%	
platypus	33.3%	33.3%	33.3%	33.3%	
qwen	60.0%	60.0%	50.0%	60.0%	

 TABLE 4.8: Percentatge of responses mentioning the name of the method used in the algorithm

It can be observed that the most effective approach is to maintain the system role default and integrate two illustrative examples. However, this does not hold for the *platypus* model, which performs always the same.

4.2.2 **Results of Experimental Analysis 2.2**

As evidenced in the results of section 3.3.1, the models *codellama* and *llama2* are incapable of fulfilling the requisite task. This is the reason why we wanted to analyse

whether the same is true for the results of the section 3.3.2.

In the case of *codellama*, we find ourselves in precisely the same situation as that described above, since 98.6% of the responses indicated that it was unable to perform the requested task due to ethical considerations. The *llama2* model is similarly irrelevant to the current task. It is of interest to note that 67.5% of the responses include erroneous information, 9.7% of the time the model provides data from the examples and 8% of the responses are not in the expected format. Furthermore, in ten of the executions of *llama2* the answer given includes codes from their training that have no relevance to the task at hand. An exemplar of the latter can be observed in Figure E.9.

Consequently, the responses provided by *codellama* and *llama2* are deemed invalid, leaving a total of 1,872 remaining responses. It would be beneficial to determine the proportion of invalid responses for each model. It should be noted that an answer is classified as invalid when errors presented are in the provided example, not in the studied code, responses are in an incorrect format, those providing false errors, answers offering corrections that are not correct, or those ignoring the instructions and presenting an entire corrected code or training codes within their answer.

Model	llama3	gpt-3.5	gpt-4	deepseek	platypus	qwen
valid	319 (of 360)	186 (of 216)	211 (of 216)	242 (of 360)	290 (of 360)	316 (of 360)
answers	$\equiv 88.6\%$	$\equiv 86.1\%$	$\equiv 97.7\%$	$\equiv 67.2\%$	$\equiv 80.6\%$	$\equiv 87.8\%$
invalid	41 (of 360)	30 (of 216)	5 (of 216)	118 (of 360)	70 (of 360)	44 (of 360)
answers	$\equiv 11.4\%$	≡ 13.9%	$\equiv 2.3\%$	$\equiv 32.8\%$	$\equiv 19.4\%$	≡ 12.2%

TABLE 4.9: Answers generated by each model classified as valid and invalid

Table 4.9 presents the number and percentage of valid answers provided by each model. The model that has generated the most optimal responses is *gpt-4* followed by *llama3*, *gpt-3.5* and *qwen*.

The *deepseek* model has generated 32.8% of incorrect answers, which is a cause for concern. To identify the nature of these incorrect answers, we will analyse them in detail. It would be beneficial to observe the model's behaviour in the presence of both syntax errors and no syntax errors in the code entered. Of the 118, 15 were erroneous in the absence of syntax errors, in contrast to 103 erroneous response executions with syntax errors. In other words, *deepseek* has provided erroneous responses, incorrect corrections, or responses containing all the corrected code 57.2% of the times when errors were to be detected and there were errors in the code. It is for this reason that we consider this model unsuitable for this task, as we cannot guarantee with an acceptable level of confidence that its responses do not contain errors or false corrections.

A further area of investigation is the identification of models that demonstrate a tendency to fail to detect errors when such errors are present in the code. The objective of this analysis is to examine the 756 results derived from erroneous codes. Table 4.10, presents the number of responses from each model that did not identify an error in the code when errors were present.

Model	llama3	gpt-3.5	gpt-4	platypus	qwen
No error	0 (of 180)	2 (of 108)	0 (of 108)	90 (of 180)	35 (of 180)
detected	$\equiv 0\%$	$\equiv 1.9\%$	$\equiv 0\%$	$\equiv 50.0\%$	$\equiv 19.4\%$

 TABLE 4.10: Responses that do not identify any errors when there are errors in the code

The model *platypus* demonstrates that in half of the cases where an erroneous code has been introduced, it has been unable to identify this. Consequently, we conclude that this model is also unsuitable for the task at hand.

We now proceed to analyse the responses of the 1,152 runs of the models *gpt-3.5*, *gpt-4*, *llama3* and *qwen*. One-half of the runs were conducted with error-free code, while the other half involved the introduction of errors into the code. In the absence of any errors in the code, no specific instructions were provided in the prompt regarding the contents of the "code_sintax_explanation" field. It has been observed that the *gpt-3.5* and *qwen* models have responded with a dictionary that lacks the "code_sintax_explanation" field on several occasions due to the lack of clarity in the prompt.

As can be observed in Table 4.11, all models have indicated that no errors were identified, except three iterations of *gpt-3.5* and six runs of *gpt-4*. Figure E.10 illustrates the three responses from *gpt-3.5*. The errors the model has identified are not significant. The model has noted instances where spaces are absent when they are not required, as well as the inclusion of elements that are not present in the source text. It is important for students to be aware that this type of error can occur and to critically analyse the output. Conversely, the responses of *gpt-4* which indicate the presence of errors are because, in the absence of syntactic errors, the model identifies conceptual inconsistencies. This indicates that at least *gpt-4* is capable of identifying these errors, although in this context it was not required to do so. The response may be found in Figure E.11.

Model	llama3	gpt-3.5	gpt-4	qwen
Correct complete	180 (of 180)	19 (of 108)	102 (of 108)	160 (of 180)
dictionary	$\equiv 100\%$	$\equiv 17.6\%\%$	$\equiv 94.4\%$	$\equiv 88.9\%$
Partial dictionary	0 (of 180)	86 (of 108)	0 (of 108)	20 (of 180)
	$\equiv 0\%$	$\equiv 79.6\%$	$\equiv 0\%$	$\equiv 11.1\%$
Invalid answer	0 (of 180)	3 (of 108)	6 (of 108)	0 (of 180)
	$\equiv 0\%$	$\equiv 2.8\%$	$\equiv 5.6\%$	$\equiv 0\%$

TABLE 4.11: Responses when there are no errors in the code

We now proceed to analyse the responses of the runs that have errors. For these responses, we have already considered the possibility of not identifying them previously. The remaining options are presented in Table 4.12. For each model, we observe the executions that have identified some, but not all, errors; those that have identified all errors; and those that have generated invalid answers.

TABLE 4.12: Responses when there are errors in the code

Model	llama3	gpt-3.5	gpt-4	qwen	
All errors detected	125 (of 180)	40 (of 108)	96 (of 108)	21 (of 180)	
	$\equiv 69.4\%$	$\equiv 37.0\%$	$\equiv 88.9\%$	$\equiv 11.7\%$	
Model	llama3 gpt-3.5		gpt-4	qwen	
-----------------------	-----------------	-----------------	----------------	-----------------	--
Some orrers detected	14 (of 180)	39 (of 108)	7 (of 108)	80 (of 180)	
Joine errors delected	$\equiv 78.0\%$	$\equiv 36.1\%$	$\equiv 6.5\%$	$\equiv 44.4\%$	
Invalid answer	41 (of 180)	27 (of 108)	5 (of 108)	44 (of 180)	
invanu answei	$\equiv 22.8\%$	$\equiv 25.0\%$	$\equiv 4.6\%$	$\equiv 24.4\%$	

TABLE 4.12: Responses that do not identify any errors when there are errors in the code(cont.)

A significant proportion of the responses appear to be invalid. The objective is to analyse the circumstances under which these responses were obtained. To this end, we can examine Table 4.13, which presents the number of incorrect responses obtained for each model, based on whether the system role was modified or not. Additionally, we can categorize the incorrect responses based on the prompts, distinguishing between instances where no examples were provided, one example was provided, or two examples were provided. It can be observed that the greatest number of erroneous responses was obtained when the system role was modified without the incorporation of exemplars. Conversely, the lowest number of incorrect answers was yielded when the system role was modified with the inclusion of one or two examples.

TABLE 4.13: Number of incorrect responses obtained for each model, based in where the system role was modified and the number of examples

	System role default			System role modified		
Model	0	1	2	0	1	2
	examples	example	examples	examples	example	examples
llama3	10 (of 41)	5 (of 41)	0 (of 41)	15 (of 41)	11(of 41)	0 (of 41)
	$\equiv 24.4\%$	$\equiv 12.2\%$	$\equiv 0\%$	$\equiv 36.6\%$	$\equiv 26.8\%$	$\equiv 0\%$
gpt-4	1 (of 5)	1 (of 5)	1 (of 5)	1 (of 5)	0 (of 5)	1 (of 5)
	$\equiv 20.0\%$	$\equiv 20.0\%$	$\equiv 20.0\%$	$\equiv 20.0\%$	$\equiv 0\%$	$\equiv 20.0\%$
gpt-3.5	6 (of 27)	3 (of 27)	6 (of 27)	7 (of 27)	1 (of 27)	4 (of 27)
	$\equiv 22.2\%$	$11.1 \equiv \%$	$\equiv 22.2\%$	$\equiv 25.9\%$	$\equiv 3.7\%$	$\equiv 14.8\%$
qwen	5 (of 44)	5 (of 44)	9 (of 44)	15 (of 44)	10 (of 44)	0 (of 44)
	$\equiv 11.3\%$	$\equiv 11.3\%$	$\equiv 20.5\%$	$\equiv 34.1\%$	$\equiv 22.7\%$	$\equiv 0\%$

We shall now proceed to analyse the problems for which each model has identified the errors, distinguishing between those instances where the full range of errors has been detected and those where only some of the errors have been identified. About the *Eratosthenes* problem with the iterative code and the *Knapsack* problem with the recursive code, it can be observed that there is only one syntax error. Consequently, they are unable to identify only some of the errors. The tables display the total number of answers for each condition about the total number of executions with error codes performed for each code of a given algorithm.

	Recursi	ve code	Iterative code		
gpt-4	Detects all	Detects all Detects some		Detects some	
	errors	of the errors	errors	of the errors	
Eratosthenes	14 (of 18)	4 (of 18)	18(of 18)	-	
Euclid	15 (of 18)	3 (of 18)	18(of 18)	0 (of 18)	
Binary search	-	-	17 (of 18)	0 (of 18)	
Knapsack	14(of 18)	-	-	-	

TABLE 4.14:	gpt-4 responses	detecting errors	grouped b	oy algorithms
	())	~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~	()	/ ()

As illustrated in Table 4.14 the *gpt-4* model consistently provides accurate responses in nearly all instances. In the case of the *Erathostenes* problem, the model detects all errors in the majority of instances when executing the code from Figure D.3. However, in four instances, it fails to identify all errors, including the lack of definition of the variable c and the need to set it to the current value. In the four executions in which this situation was identified, the system role was not modified. However, it should be noted that the examples have not affected the outcome, as there is one case with zero examples, two cases with one example, and one case with two examples. Figure E.12 illustrates an example of an answer that does not detect all the errors, but rather identifies a conceptual error. In the case of *Euclid*, there are three runs where all errors are not identified. In this instance, the incorrect use of == in the assignments of the usage example is not identified, as illustrated in Figure E.13. For the problems of Binary Search and Knapsak, there are instances where the obtained answers are not valid. In the case of *Binary Search* the error is a single instance. In contrast, for *Knapsack*, there are four instances incorrect. With regard to the *Binary Search* problem, as illustrated in the answer to Figure E.14 the problem is that it tells us that there is an unnecessary ";" when it is not so, what happens is that it should be a ",". In the case of *Knapsack*, we do not provide a detailed analysis of the errors. The four invalid answers can be attributed to three instances of erroneous input and one instance of an incorrect correction. An example of these responses can be found in Figure E.15. It is important to note that while *gpt-4* has occasionally generated erroneous responses, it has also provided correct responses on a vast majority of occasions. Furthermore, these responses have proven to be useful in identifying and correcting the aforementioned errors. Furthermore, it should be noted that when the same prompt is executed, the model has never provided an invalid answer on the three iterations. As an illustrative example, Figure E.16 presents one of the responses generated by *gpt-4* for the *Euclid* problem, free from errors. This response has been selected as an exemplar of a straightforward answer that accurately identifies the errors and provides clear guidance on how to rectify them.

TABLE 4.15:	llama3	responses	detecting	errors	grouped	by	algo-
		ri	thms			-	-

	Recursive codea3Detects allDetects of all		Iterative code		
llama3			Detects all	Detects some	
	errors	of the errors	errors	of the errors	
Eratosthenes	26 (of 30)	4 (of 30)	30 (of 30)	-	
Euclid	14 (of 30)	5(of 30)	20(of 30)	0(of 30)	
Binary search	-	-	20(of 30)	0 (of 30)	
Knapsack	15 (of 30)	5 (of 30)	-	-	

Table 4.15 presents the identical responses that were previously examined in the

context of *gpt-4*, now for the *llama3* model. In the case of the *Eratosthenes* algorithm, it is observed that the model almost always detects all the errors, except four runs where, as in *gpt-4*, it does not indicate that the variable c is not defined and should be current. This is illustrated in Figure E.17. The Euclidean method presents a more challenging problem to address. The five answers that did not detect all the errors were due to the model's failure to identify the incorrect use of a semicolon in place of a comma, as illustrated in Figure E.18 as an example. In the case of the *knapsack* problem, it has been determined that five of the solutions exhibit some of the errors identified. This is contrary to the previously stated assertion that only a single syntax error exists. It can be observed that the model has identified a conceptual error that could be considered a syntax error. This is illustrated in Figure E.19.

Finally, in Tables 4.16 and 4.17, we observe the outcomes of the experiments conducted with *gpt-3.5* and *qwen*. There are numerous instances where the model has identified some of the errors but not all of them. In the case of *gpt-3.5*, it generates erroneous responses in instances where it fails to accurately identify the *Knapsack* errors. For instance, consider the Figure E.20, the *qwen* model provides corrections that are not accurate, as evidenced in Figure E.21.

	Recursive code		Iterative code		
gpt-3.5	Detects all Detects some		Detects all	Detects some	
	errors	of the errors	errors	of the errors	
Eratosthenes	10 (of 18)	6(of 18)	11(of 18)	-	
Euclid	1 (of 18)	11 (of 18)	3 (of 18)	15 (of 18)	
Binary search	-	-	7(of 18)	7 (of 18)	
Knapsack	8 (of 18)	-	-	-	

TABLE 4.16: gpt-3.5 responses detecting errors grouped by algorithms

TABLE 4.17: quen responses detecting errors grouped by algorithms

	Recursi	ve code	Iterative code		
qwen	Detects all	Detects some	Detects all	Detects some	
	errors of the errors		errors	of the errors	
Eratosthenes	0 (of 30)	25 (of 30)	0 (of 30)	-	
Euclid	5 (of 30)	20 (of 30)	0 (of 30)	25(of 30)	
Binary search	-	-	6 (of 30)	10 (of 30)	
Knapsack	10 (of 30)	-	-	-	

Chapter 5

Conclusions

In conclusion, the results obtained for each of the models will be analysed and an overall view of the results will be provided.

Codellama did not yield the expected results. As it is a large model, specialised in code and finetuned to better understand human instructions, it was expected to give good results. However, we observed that this model, despite its efficacy in transforming pseudocode into code and producing well-structured and commented code, has encountered difficulties in executing specific tasks. When attempting to ascertain the purpose of the code or identify potential syntax errors, the model indicates that such activities are unethical and does not provide the anticipated response. Consequently, it can be concluded that the model's objective of ensuring safety has resulted in a threshold that is excessively high in terms of what it can answer, rendering it unusable in certain use cases, as proposed.

The Llama-2 model has yielded results that are not applicable to any of the specified use cases. As an alternative, the improved version of the model, Llama-3, may be employed. The latter model has yielded results that are comparable to those produced by the gpt models. The experiments conducted have revealed that the model has difficulty generating recursive codes, suggesting that its reasoning capacity may be inferior to that of the gpt models. The model has consistently produced the response in the expected format, in the context of identifying syntactic errors, has never erroneously identified the presence of errors when none exist. Conversely, it has demonstrated the capacity to detect a greater number of errors after gpt-4 when they do exist. In conclusion, it can be stated with confidence that this model has provided the most favourable results of all the open-source models analysed.

We now turn our attention to the gpt models. Out of these, gpt-4 has generated the most promising results. It is important to note that the ability to provide accurate responses is not absolute. On occasions, we have observed instances where the system has made errors, failed to generate the desired output, or failed to adhere to the prescribed steps. It is therefore important to exercise one's own judgement before accepting a gpt-4 answer as valid and not questioning it, despite its proven abilities. It is important to highlight certain aspects of the gpt-3.5 model. For instance, when introducing a code with conceptual errors, it was observed that half of the generated answers did not provide the objective of the code. Consequently, it can be posited that the gpt-3.5 model attempts to align itself with the specifications outlined in the prompt, while omitting information that has not been explicitly requested which is considered as a correct answer.

Finally, we analysed the qwen, deepseek, and platypus models. The platypus model proved to be of limited assistance to students in implementing the code, as it encountered difficulties in translating pseudocode into code. Furthermore, it is not an effective method for detecting syntax errors, and thus it is not a model that we would recommend for use in the tasks presented. With regard to the deepseek model, it

should be noted that it is the smallest model considered, with only 33B parameters. The outcomes were therefore deemed satisfactory, given the constraints associated with its size. It is noteworthy that 75% of the time, the model has been able to translate pseudocode into code. However, its reasoning abilities are not particularly noteworthy, and it does not detect syntax errors either, as it tends to invent errors. Consequently, we can infer that this model is suitable for translating from pseudocode to code, yet it lacks the capacity to detect errors within the code itself. Finally, we consider the gwen model. It is important to note that this model has demonstrated superior performance in generating code from mathematical reasoning compared to pseudocode. Qwen exhibits greater performance in Experiment 2 (Code interpretation and error detection) than in Experiment 1 (Implementation of a described algorithm). This is evidenced by its ability to identify the method of the algorithm in question and provide valid answers when identifying syntax errors. In this regard, it outperforms gpt-3.5, achieving a slightly higher percentage of correct responses. It should be noted that this model has certain limitations. In 19.4% of cases, it fails to identify errors, while in 24.4% of cases, it provides invalid answers. However, it is important to note that this percentage is comparable to that of gpt-3.5, which provides invalid responses 25% of the time when there are syntax errors. A comparison of deepseek and qwen reveals that deepseek tends to identify false errors, whereas quen tends to identify only some of them in order to avoid mistakes.

It is evident from this analysis that the system roles introduced are not effective and have not resulted in an improvement in outcomes. Indeed, they have had the opposite effect, with results becoming worst. In relation to the examples, it has been established that they do indeed have an effect on the response. In consequence, we believe that improving these examples could lead to better results.

It is evident that conceptual errors represent a point of improvement for all the models. When introducing codes with conceptual errors, they have practically always responded as if there were no error. Consequently, if a student encounter such an error, it is likely that they would require assistance from their teacher in order to identify it. It is noteworthy that the only model that performs exceptionally well in identifying syntax errors is gpt-4. In over 20% of cases, the other models provided erroneous responses. It would be beneficial to investigate whether this percentage could be reduced by providing more effective examples or by implementing a more structured prompting process.

Future steps include the improvement of the prompts to make them a little more guided, as well as the testing of these prompts with other algorithms and real students in order to determine the extent to which the answers provided are helpful. In this manner, if the responses prove to be beneficial, a proof of concept could be established, and a tool capable of executing these functions could be implemented with improved versions of the prompts and more illustrative examples. These could then be tested with new students for potential incorporation in class.

Appendix A

Prompts used in the experimental analysis 1

•••••

Given a natural number x, we want to calculate all prime numbers smaller than x. To do this, the first thing to do is to have a clear definition of a prime number. A number is prime when it has no divisors except one and itself. Another concept to keep in mind is that if a is a multiple of b and b is prime, then a cannot be prime. Since we have said before that if a is prime then its only divisors are 1 and itself. Therefore, if b is prime, it is not 1, and since b is a divisor of a, there are only two options: either a is b, or a is not prime. Thus, 2 is the first prime number. We remove all multiples of 2 up to x. The next number that we have not removed will be the next prime number, and we do the same process of removing its multiples up to x. We will finish this process once we get to the nearest integer to the square root of x. This is because any non-prime number after the square root of x will have been discarded before.

I want a code that will be able to replicate this process and give me all the numbers up to x that have not been discarded.

Additionally, answer this question: does this algorithm have a name?

FIGURE A.1: Strategy 1: Sieve of Eratosthenes algorithm mathematical reasoning prompt (no recursion required) Given a natural number x, we want to calculate all prime numbers smaller than x. To do this, the first thing we need is a clear definition of a prime number. A number is prime when it has no divisors except one and itself. Another concept to keep in mind is that if a is a multiple of b and b is prime, then a cannot be prime. Since we have previously stated that if a is prime, then its only divisors are 1 and itself, therefore if b is prime, it is not 1. Since b is a divisor of a, there are only two options: either a is b or a is not prime. Thus, 2 is the first prime number. We start by considering a list with all numbers from 2 to x. We remove from this list all multiples of 2 up to x. The next number that we have not removed in the list will be the next prime number, and we repeat the process of removing its multiples up to x in a recursive manner. We will finish this process once we reach the nearest integer to the square root of x. This is because any non-prime number after the square root of x will have been discarded before.

I want code that can replicate this process recursively and give me all the numbers up to x that have not been discarded.

Additionally, does this algorithm have a name?

FIGURE A.2: Strategy 1: Sieve of Eratosthenes algorithm mathematical reasoning prompt (recursion required)

.....

Let's suppose we have two numbers "a" and "b". The first step is to verify that they are natural numbers. If this is true, let "x" be the maximum between a and b, and "y" be the minimum between a and b. Then there will exist two natural numbers "c" and "r" such that $y \ge c > r \ge 0$, so we can write x = y * c + r. There are two options: either r is 0 or r is not 0. If r is not 0, then there will exist two other natural numbers "d" and "s" such that y = r * d + s, with $r \ge d > s \ge 0$. The naturals between r and 0 are finite. Therefore, this process will have an end. If "s" is not 0, since "r" is a natural number, there will exist "e" and "f" such that r = s * e + f with $s \ge e > f \ge 0$. If f is not 0, we would repeat the process. I want a code that can replicate this process and give me the denominator of the division that has a remainder of 0. In this case, if s = 0, it would be r; if f = 0, it would be s, and if f is not 0, it would imply continuing iterating until finding the solution.

Additionally, answer this question: Does this algorithm have a name?

FIGURE A.3: Strategy 1: Euclidean algorithm mathematical reasoning prompt (no recursion required)

.....

•••••

Let's suppose we have two numbers "a" and "b". The first step is to verify that they are natural numbers. If this is true, let "x" be the maximum between "a" and "b", and "y" be the minimum between "a" and "b". Then, there will exist two natural numbers "c" and "r" such that $y \ge c > r \ge 0$, allowing us to write x = y * c + r. There are two options: either r is 0 or r is not 0. If r is not 0, then there will exist two other natural numbers "d" and "s" such that y = r * d + s, with $r \ge d > s \ge 0$. The naturals between r and 0 are finite, so this process will have an end. If "s" is not 0, as "r" is a natural number, there will exist "e" and "f" such that r = s * e + f, with $s \ge e > f \ge 0$. If f is not 0, we would repeat the process.

I want code that can replicate this process recursively and give me the denominator of the division that has a remainder of 0. In this case, if s = 0, it would be r; if f = 0, it would be s; and if f is not 0, it would involve continuing the iteration until finding the solution.

Additionally, answer this question: Does this algorithm have a name?

FIGURE A.4: Strategy 1: Euclidean algorithm mathematical reasoning prompt (recursion required)

•••••

We want to find an element in an ordered list of elements. To do this, we divide the list in half. If the number we are looking for is smaller than the value that occupies the middle position of the list, it means that it is in the first half of the list. If it is larger, it means that it is in the second half of the list. Depending on which of the two conditions is met, we consider the corresponding sublist and repeat the process. As soon as the value in the middle of the list is the desired value, we stop. In this way, if we reduce the length of the list where we look by half, we will reach a point where either we will have found it or we will have looked at all possible locations, and we can say that the number is not in the list. Give me a Python code that does this process and that returns the position in the list where the value it was looking for is if it finds it and "-1" if it does not find it. Additionally, answer the following question: Does this algorithm have a name?

.....

FIGURE A.5: Strategy 1: Binary search algorithm mathematical reasoning prompt (no recursion required)

We want to find an element in an ordered list of elements. To do this, we divide the list in half. If the number we are looking for is smaller than the value that occupies the middle position of the list, it means that it is in the first half of the list. If it is larger, it means that it is in the second half of the list. Depending on which of the two conditions is met, we consider the corresponding sublist and repeat the process. As soon as the value in the middle of the list is the desired value, we stop. In this way, if we reduce the length of the list where we are looking by half, we will reach a point where either we will have found it or we will have looked at all possible locations and we can say that the number is not in the list. Give me a Python code that does this process recursively and that returns the position in the list where the value it was looking for is if it finds it and "-1" if it does not find it. Additionally, answer the following question: Does this algorithm have a name?

FIGURE A.6: Strategy 1: Binary search algorithm mathematical reasoning prompt (recursion required)

•••••

.....

We have an optimization problem that we want to solve. We have N elements. Each one of them has two associated parameters, p and w. So we can consider p as a list $p_1, ..., p_N$ and w as a list $w_1, ..., w_N$. The objective of the algorithm is to keep the subset of elements whose sum of their w_i is less than or equal to a certain constant W and whose sum of the p_i is maximal at the same time. To do this, we consider all subsets of w with a sum <= W and then select that subset whose sum of the maximum. Give me a Python code that does this and that returns the maximum sum of the selected elements in "p" and the i-th elements that have been used. Also, answer the following question: Does this algorithm have a name?

FIGURE A.7: Strategy 1: Knapsack algorithm mathematical reasoning prompt (no recursion required)

.....

.....

We have an optimization problem that we want to solve recursively. We have N elements. Each one of them has two associated parameters, p and w. So we can consider p as a list $p_1, ..., p_N$ and w as a list $w_1, ..., w_N$. The objective of the algorithm is to keep the subset of elements whose sum of their w_i is less than or equal to a certain constant W and whose sum of the p_i is maximal at the same time. To do this, we consider all subsets of w with a sum <= W and then select that subset whose sum of the p_i is maximum. Give me a Python code that does this recursively and that returns the maximum sum of the selected elements in "p" and the i-th elements that have been used. Also, answer the following question: Does this algorithm have a name?

FIGURE A.8: Strategy 1: Knapsack algorithm mathematical reasoning prompt (recursion required)

""" Help me implement a Python code that performs the following steps:

1. Given a natural number n, we define an array of n positions, which we will call x.

2. We will initialize the array x with all its positions set to True.

3. We change the values of the first two positions to False.

4. For each number j from 2 to the nearest integer to the square root of n, we do the following:

4.1. If the array x at position j takes the value True, we update the values of x to False at positions j*j through n by jumping from j to j.

4.2. We return the indices of the array x that take the value True.

FIGURE A.9: Strategy 2: Sieve of Eratosthenes algorithm pseudocode description prompt

Help me implement a Python code that performs the following steps:

1. Verify if these are two positive integers.

2. If they are, the program must perform the Euclidean division of the larger by the smaller.

3. If the remainder of the division is 0, the program must return the denominator.

4. If the remainder is not 0, you will do the following until the remainder of the division you perform is 0:

4.1. Consider the denominator and the remainder of the previous division and repeat the Euclidean division between them.

4.2. Do this process until you reach a denominator that is 0.

4.3. When the remainder is 0, the program must return the denominator of that division.

.....

FIGURE A.10: Strategy 2: Euclidean algorithm pseudocode description prompt

.....

.....

Help me implement a Python code that performs the following steps:

1.Given a number "a" and a sorted list "x", we define a new variable, which is the middle position of this list "x", and we call this variable "mid".

2.We check if x at position "mid" takes the value "a". If it does, we stop and return "mid".

2.1. If "x" at position "mid" takes a value "b" which is not "a", we do the following steps:

2.1.1. If "b" is less than "a", we update the list and consider only the sublist from position 0 to "mid".

2.1.2. If "b" is greater than "a", we update the list and consider only the sublist from position "mid" to the end.

2.2. We update x with the corresponding sublist.

2.3. We repeat the process until x at position "mid" equals "a" or "x" has a length of 2. If x at position "mid" equals "a", we return "mid", and if the length of x is 2, we check if either of the two positions has "a", and we return that position. If not, we return -1 to indicate that the element was not found

FIGURE A.11: Strategy 2:Binary search algorithm pseudocode description prompt "" Help me implement a Python code that performs the following steps:

1. Given two vectors "p" and "w" of the same length, n, and a natural number W.

2.We define a function which takes as input a constant A, a list z, a list t and a constant m.

2.1. This function is defined in a recursive way. Let's call this function "fun_1". This function returns two outputs. The first output is a number and the second output is a list.

2.2. If m or A are 0, the output of the function should be 0 and [].

2.3. If the value of z at the (m-1) position (starting from 0) is bigger than A, then fun_1 evaluated on A, z, t, m should be the same as fun_1 evaluated on A, z, t, m-1.

2.4. If the value of z at the position (m-1) is smaller than A, then we calculate two numbers, c and d.

2.4.1. c is the sum of the value of t at the (m-1) position and the first output of fun_1 evaluated on A-z[m-1], z, t and m-1.

2.4.2. d is the first output of fun_1 when evaluated to A, z, t and m-1.

2.5. If c is bigger than d. In this case, we return as the first output c and as the second output we add "m-1" to the second output of fun1 evaluated on A-z[m-1], z, t and m-1.

2.6. If d is bigger than c we return fun_1 evaluated on A, z, t and m-1

3.Evaluate fun_1 on (W, w, p, n). """

FIGURE A.12: Strategy 2:Knapsack algorithm pseudocode description prompt

Appendix **B**

Sample of Responses generated by the different models and prompts during the experimental analysis 1

What you're describing is essentially the process used in the Euclidean algorithm to find the greatest common divisor (GCD) of two numbers *a* and *b*. The Euclidean algorithm repeatedly subtracts the smaller number from the larger one (or uses the modulus operator, which is a more efficient computational representation of the same process) until the remainder is 0. The last non-zero remainder is the GCD of the original pair.

Here's a Python code snippet that implements the Euclidean algorithm based on your description, finding the GCD of two integers:

"'python

```
def euclidean_algorithm(a, b):
    # Ensure a and b are natural numbers (positive integers)
    if a < 1 or b < 1:
        raise ValueError("Both_a_and_b_must_be_natural_numbers")
    # Continue finding remainders until one of them is 0
    while b != 0:
        a, b = b, a % b
    # a is now the GCD of the original a and b
    return a
# Example usage
a = 48
b = 18
print("The_GCD_of", a, "and", b, "is:", euclidean_algorithm(a, b))
""</pre>
```

This Python function first checks if 'a' and 'b' are both natural numbers. It then proceeds to iterate, replacing 'a' with 'b' and 'b' with 'a % b' (the remainder when 'a' is divided by 'b') until 'b' becomes 0. At that point, 'a' contains the GCD.

To answer your question about the name of the algorithm: This is called the **Euclidean Algorithm**, which has been known since ancient times and is a fundamental algorithm for number theory and computation of the greatest common divisor (GCD).

FIGURE B.1: Response generated by gpt-4 when executing the prompt from the Figure A.3

•••••

The algorithm described closely mimics the Euclidean algorithm, which is used to find the greatest common divisor (GCD) of two natural numbers. In your description, each step replaces the larger number with the remainder of the division until a remainder of 0 is obtained. When the remainder is zero, the smaller number at that step is the GCD.

Here is a Python code that implements the described algorithm and returns the smallest non-zero remainder which ends the cycle (the GCD):

```
"'python
```

```
def euclidean_algorithm(a, b):
    if not (isinstance(a, int) and isinstance(b, int)
    and a > 0 and b > 0):
        raise ValueError("Both_a_and_b_must_be_natural
    unubers_(positive_integers)")
    x, y = max(a, b), min(a, b)
    while y != 0:
        c = x // y
        r = x % y
        print(f"Calculating:_{x}_=_{y}_*_{c}+_{r}")
        x, y = y, r # update x to y, and y to r
    return x
# Example usage
    a = 48
    b = 18
```

gcd = euclidean_algorithm(a, b)
print("The_greatest_common_divisor_(last_non-zero_remainder):", gcd)

"' This code will calculate stepwise the combinations as described and print each division step. The algorithm concludes when it finds a remainder of zero, and the other non-zero number associated with that division would be the GCD of the original two numbers.

Regarding your question, as mentioned, the described algorithm is known as the "Euclidean algorithm". It is an ancient algorithm used for finding the greatest common divisor of two integers and is very efficient and commonly used in various applications, including in mathematics and computer science for problems related to number theory.

FIGURE B.2: Response generated by gpt-4 when executing the prompt from Figure A.3

""" This algorithm is known as the Sieve of Eratosthenes. Here is a Python implementation of the algorithm:

"'python

```
def sieve_of_eratosthenes(x):
    # Create a list of boolean values, all initially set to True
    sieve = [True] * (x + 1)
    sieve[0:2] = [False, False]
# 0 and 1 are not prime numbers
    # Iterate from 2 to the square root of x
    for current_prime in range(2, int(x ** 0.5) + 1):
        # If the current number is prime,
        #mark its multiples as not prime
        if sieve[current_prime]:
            for multiple in range(current_prime ** 2, x + 1,
            current_prime):
                sieve[multiple] = False
    # Return a list of all prime numbers in the range
    return [num for num, is_prime in enumerate(sieve) if is_prime]
# Example usage:
x = 30
prime_numbers = sieve_of_eratosthenes(x)
print("Prime_numbers_up_to", x, ":", prime_numbers)
....
```

This code creates a list of boolean values, where 'sieve[i]' represents whether 'i' is a prime number or not. It then iterates over the list, marking the multiples of each prime number as not prime. Finally, it returns a list of all prime numbers in the range.

Note that this implementation uses a recursive approach, but it's not a traditional recursive function. Instead, it uses a loop to iterate over the list and mark the multiples of each prime number. This is because Python has a recursion limit, and using a loop is generally more efficient and scalable.

The time complexity of the Sieve of Eratosthenes is O(n log log n), making it a very efficient algorithm for finding prime numbers up to a given number.

.....

FIGURE B.3: Response generated by llama3 when executing the prompt from figure A.2

""" The algorithm you described is known as the Euclidean algorithm, which is a well-established method for computing the greatest common divisor (GCD) of two integers. It is named after the ancient Greek mathematician Euclid, who described it in his book "Elements".

Here is an example of how the Euclidean algorithm works:

Suppose we want to find the GCD of 12 and 18. We can start by writing 18 = 12 * 1 + 6, so we can see that 12 is a divisor of 18. Now, we can repeat the process by writing 12 = 6 * 2 + 0, so we can see that 6 is a divisor of 12. We can continue this process by writing 6 = 2 * 3 + 0, so we can see that 2 is a divisor of 6. We can repeat this process again by writing 2 = 1 * 2 + 0, so we can see that 1 is a divisor of 2.

At this point, we have found the GCD of 12 and 18, which is 6. We can express 18 as 6 * 3 + 0, and 12 as 6 * 2 + 0, so we can see that 6 is the greatest common divisor of 12 and 18.

The Euclidean algorithm can be written in Python as follows:

"'python

```
def euclidean_algorithm(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

" This function takes two integers a and b as input, and it returns the GCD of a and b. It works by repeatedly subtracting the smaller number (b) from the larger number (a) until b is 0, at which point a is the GCD of a and b.

To use this function, you can call it with two integers as arguments, like this:

print(euclidean_algorithm(12, 18)) # Output: 6

This will print 6, which is the GCD of 12 and 18.

I hope this helps! Let me know if you have any questions.

.....

FIGURE B.4: Response generated by llama2 when executing the prompt from Figure A.4

""" The process you described is known as the Euclidean algorithm. It is an algorithm for finding the greatest common divisor of two integers. In your case, you are describing a way to find the denominator of the division that has a remainder of 0, which can be achieved by applying the Euclidean algorithm recursively.

Here is a Python implementation of the recursive function that replicates the process you described:

"'python

```
def find_denominator(a, b):
    if b == 0:
        return a
    else:
        q = a // b
        r = a % b
        return find_denominator(b, r)
# Test the function
a = 357
b = 234
result = find_denominator(max(a,b), min(a,b))
print("Denominator_of_the_division_with_remainder_0_=", result)
""
```

You can input any two natural numbers "a" and "b" to find the denominator of the division that has a remainder of 0 using this function.

FIGURE B.5: Response generated by gpt-3.5 al when executing the prompt of the Figure A.4

""" Here's a Python function that implements the described algorithm:

```
"'python
```

```
def find_element_position(a, x):
    while len(x) > 2:
        mid = len(x) / / 2
        if x[mid] == a:
            return mid
        elif x[mid] < a:
            x = x [mid:]
        else:
            x = x [:mid]
    if len(x) == 2:
        if x[0] == a:
            return 0
        elif x[1] == a:
            return 1
    return -1
# Example usage:
a = 5
x = [1, 3, 5, 7, 9]
print(find_element_position(a, x)) # Output: 2
....
```

This function takes two arguments: 'a' (the number to search for) and 'x' (the sorted list). It uses a while loop to repeatedly update the sublist based on the middle element until the element is found or the sublist has a length of 2. If the element is found, the function returns the index; otherwise, it returns -1. """

FIGURE B.6: Response generated by qwen when executing the prompt of the figure A.11

Appendix C

Prompts used in the experimental analysis 2

•••••

Please analyze the &&code&& as indicated in <INSTRUCTIONS>. You have to return the result of the <INSTRUCTIONS> in a dictionary format, please. It is VERY IMPORTANT that you follow the <INSTRUC-TIONS>.

<INSTRUCTIONS>:

<1.Create a dictionary with the following key fields: "code_language", "code_objective">

<2. Set the answer "What programming language is the code written in?" as the value of "code_language". If you don't know the answer don't make it up, just put "pseudocode".>

<3. Set the answer "What is the code doing?" as the value of "code_objective". Use only a few words. If you don't know the answer don't make it up. If there is no clear answer put "unkown".>

<4.Return only the dictionary . Don't give any further information>

&&code&&:

&& {} && RESULT:

FIGURE C.1: Code objective: zero-shot prompting

Please analyze the &&code&& as indicated in <INSTRUCTIONS>. You have to return the result of the <INSTRUCTIONS> in a dictionary format, please. It is VERY IMPORTANT that you follow the <INSTRUCTIONS>.

<INSTRUCTIONS>:

<1.Create a dictionary with the following key fields: "code_language", "code_objective">

<2. Set the answer "What programming language is the code written in?" as the value of "code_language". If you don't know the answer don't make it up, just put "pseudocode".>

<3. Set the answer "What is the code doing?" as the value of "code_objective". Use only a few words. If you don't know the answer don't make it up. If there is no clear answer put "unkown".>

<4.Return only the dictionary . Don't give any further information>

&&code&&:

```
&&
import math
def solution(a, b, c):
    if b*b-4*a*c =0:
        solution = [-b/(2*c),]
    if b*b-4*a*c <0:
        print("No_solution")
        break
else:
        solution = [(-b+math. sqrt(b*b -4*a*c))/(2*c),
        (-b-math. sqrt(b*b -4*a*c))/(2*c)]
    return solution
&&&
RESULT:
{code_language: "python",</pre>
```

code_objective: "define a function that solves a second order equation"}
&& {} &&
RESULT:
"""

.....

```
.....
&&code&&:
&&
from math import sqrt
def fib(i):
     if i < 2:
         return i
     else:
         phi = ((1 + sqrt(5))/2)
         fibonacci = ((phi**i-(1-phi)**i)/sqrt(5))
         return round(fibonacci)
fib(9)
&&
RESULT:
{code_language: "python", code_objective: "define a function that calcu-
lates the i-th term of the fibonacci series using the golden ratio",}
&& {} &&
```

RESULT:

.....

FIGURE C.3: Code objective: 2-shot prompting (We add to the prompt in the figure C.2 this second example Please analyze the &&code&& as indicated in <INSTRUCTIONS>. You have to return the result of the <INSTRUCTIONS> in a dictionary format, please. It is VERY IMPORTANT that you follow the <INSTRUCTIONS>.

<INSTRUCTIONS>:

.....

<1.Create a dictionary with the following key fields: "code_sintax_error", "code_sintax_explanation">

<2. Considering that the code is written in -code_language-, Set the answer to the question "Is there any syntax error in the code?" as the value of "code_sintax_error". The answers can only be "Yes" or "No" >

<3. If the answer to the previous question was "Yes", which are this errors? Set the answer as the value of "code_sintax_explanation". Gives enough details to identify this errors.>

<4.Return only the dictionary . Don't give any further information>

-code_language-: -Python-

&&code&&:

&& {} && RESULT:

.....

FIGURE C.4: Code syntax error: zero-shot prompting

Please analyze the &&code&& as indicated in <INSTRUCTIONS>. You have to return the result of the <INSTRUCTIONS> in a dictionary format, please. It is VERY IMPORTANT that you follow the <INSTRUC-TIONS>.

<INSTRUCTIONS>:

<1.Create a dictionary with the following key fields: "code_sintax_error", "code_sintax_explanation">

<2. Considering that the code is written in -code_language-, Set the answer to the question "Is there any syntax error in the code?" as the value of "code_sintax_error". The answers can only be "Yes" or "No" >

<3. If the answer to the previous question was "Yes", which are this errors? Set the answer as the value of "code_sintax_explanation". Gives enough details to identify this errors.>

<4. Return only the dictionary . Don't give any further information>

-code_language-: -Python-

```
&&code&&:

&&

import math

def solution (a, b, c):

if b*b-4*a*c =0:

solution = [-b/(2*c),]

if b*b-4*a*c <0:

print("No_solution")

break

else:

solution = [(-b+math. sqrt(b*b -4*a*c))/(2*c),

(-b-math. sqrt(b*b -4*a*c))/(2*c)]
```

return solution

&&

RESULT:

{ code_sintax_error: "Yes", code_sintax_explanation: "In 'b*b-4*a*c =0' the sintaxis for comparison is not correct." } -code_language-: -Python-&&code&&: && {} && RESULT: """

FIGURE C.5: Code syntax error: 1-shot prompting

RESULT: { code_sintax_error: "Yes",

code_sintax_explanation: "The function rount does not exist, you wanted to use the function round. Additionally you used ; when it was not necessary." }

-code_language-: -Python- &&code&&:

FIGURE C.6: Code syntax error:2-shot prompting (We add to the prompt in the figure C.5 this second example

Appendix D

.....

Codes used in Experimental Analysis 2

```
def fun_1(x, p=None, c=2):
    if p is None:
        p = [i for i in range(2, x + 1)]
    if c * c > x:
        return p
    for j in range (c*c, x+1, c):
        if j in p:
            p.remove(j)
    return fun_1(x, p, c + 1)
```

FIGURE D.1: Code Sieve of Eratosthenes: Recursive implementation without errors

```
•••••
```

.....

```
def fun_1(x):
    p = []
    s = [True] * (x + 1)
    s[0] = s[1] = False
    for i in range(2, int(x**0.5) + 1):
        if s[i]:
            p.append(i)
            for j in range(i*i, x + 1, i):
               s[j] = False
    for i in range(int(x**0.5) + 1, x + 1):
        if s[i]:
            p.append(i)
    return p
```

.....

FIGURE D.2: Code Sieve of Eratosthenes: Iterative implementation without errors

```
def fun_1(x, primes=None, current=2):
    if p is None:
        p = [i for i in range[2, x + 1]]
    if c * c > x:
        return p
    for j in range [c*c, x+1, c]:
        if j in p:
            p.remove(j)
    return fun_1(x, c + 1, p)
""""
```

FIGURE D.3: Code Sieve of Eratosthenes: Recursive implementation with conceptual and syntax errors

```
def fun_1(x):
    p = []
    s = [True] * (x + 1)
    s[0] = s[1] = False
    for i in range(2, int(x*0.5) + 1):
        if s[i]:
            p.append{i}
            for j in range(i*i, x + 1, i):
               s[j] = False
    for i in range(int(x*0.5) + 1, x + 1):
            if s[i]:
               p.append{i}
```

.....

FIGURE D.4: Code Sieve of Eratosthenes: Iterative implementation with syntax errors

.....

```
def euclidean_gcd(a, b):
    # Verify that a and b are natural numbers
    if a < 0 or b < 0:
        raise ValueError("Inputs_must_be_natural_numbers")
    # Ensure that a \ge b
    if b > a:
        a, b = b, a
    # Base case: if b = 0, a is the GCD
    if b == 0:
        return a
    # Find the remainder of a/b
    r = a \% b
    # Recursive call with a = b and b = r
    return euclidean_gcd(b, r)
# Example usage
a = 10
b = 6
print("GCD_of", a, "and", b, "is", euclidean_gcd(a, b))
```

FIGURE D.5: Code Euclidean algorithm: Recursive implementation without errors

```
.....
```

.....

```
def fun_1(a, b):
    if not isinstance(a, int) or not isinstance(b, int) or a <= 0 or b <
        return "Ambos_n meros_deben_ser_enteros_positivos."</pre>
```

```
x = max(a, b)
y = min(a, b)
while y != 0:
    c = x // y
    r = x % y
    if r == 0:
        return y
    else:
        x = y
        y = c
return "No_se_encontr ."
```

```
.....
```

FIGURE D.6: Code Euclidean algorithm: Iterative implementation with conceptual errors

```
def fun_1(a, b):
    & Verify that a and b are natural numbers
    if a < 0 or b < 0:
         raise ValueError("Inputs_must_be_natural_numbers")
    & Ensure that a \ge b
    if b > a:
        a; b == b; a
        & Base case: if b = 0, a is the output
    if b == 0:
        return a
    & Find the remainder of a/b
    r = a \% b
    & Recursive call with a = b and b = r
    return fun_1(b, r)
& Example usage
a == 10
b == 6
print("GCD_of", a, "and", b, "is", euclidean_gcd(a, b))
.....
 FIGURE D.7: Code Euclidean algorithm: Recursive implementation
```

```
with syntax errors
```

```
.....
def fun_1(a, b):
    if not isinstance(a, ent) or not isinstance(b, ent) or a <= 0 or
        return "Ambos_numeros_deben_ser_enteros_positivos."
    x = max(a, b)
    y = min(a, b)
    while y = ! 0:
        c = x // y
        r = x \% y
        if r == 0:
             return y
        else:
            x = y
             y = r
    return "No_se_encontro."
.....
```

FIGURE D.8: Code Euclidean algorithm: Iterative implementation with syntax errors

.....

```
def fun_1(1, t):
    x, y = 0, len(1) - 1
    while x <= y:
        d = (x + y) // 2
        if 1[d] == t:
            return d
        elif 1[d] < t:
            x = d + 1
        else:
            y = d - 1
        return -1
```

```
.....
```

.....

FIGURE D.9: Code Binary Search: Iterative implementation without errors

```
def fun_1(1, t)
    x; y = 0; len(1) - 1
    while x <= y
        D = (x + y) // 2
        if 1[d] == t:
            return D
        elif 1[d] < t
            x = D + 1
        else
            y = d - 1
        return -1</pre>
```

```
.....
```

FIGURE D.10: Code Binary Search: Iterative implementation with syntax errors

```
.....
```

```
def fun_1(W, wt, val, n):
    if n == 0 or W == 0:
        return 0, []
    if wt[n - 1] > W:
        return fun_1(W, wt, val, n - 1)
    else:
        v_i, i_i = fun_1(W - wt[n - 1], wt, val, n - 1)
        v_n_i, i_n_i = fun_1(W, wt, val, n - 1)
        if v_i + val[n - 1] > v_n_i:
            return v_i + val[n - 1], i_i + [n]
        else:
            return v_n_i, i_n_i
```

.....



```
"""
def fun_1(W, wt, val, n):
    if n == 0 or W == 0:
        return 0, []
    if Wt[n - 1] > W:
        return fun_1(W, wt, val, n - 1)
    else:
        v_i, i_i = fun_1(W - Wt[n - 1], wt, n - 1, val)
        v_n_i, i_n_i = fun_1(W, Wt, - 1, val)
        if v_i + val[n - 1] > v_n_i:
            return v_i + val[n - 1], i_i + [n]
        else:
            return v_n_i, i_n_i
```

FIGURE D.12: Code Knapsack algorithm: Recursive implementation with conceptual and syntax errors

Appendix E

Sample of Responses generated by the different models and prompts during the experimental analysis 2.1

""" I apologize, but as a responsible AI language model, I must clarify that I cannot provide a response that may promote or facilitate harmful or unethical activities, including those that may violate intellectual property rights or security protocols.

Therefore, I cannot provide a response that analyzes the code provided in the instructions, as it may contain sensitive or proprietary information that should not be shared publicly.

Instead, I would like to emphasize the importance of respecting the privacy and security of individuals and organizations, and promoting ethical and responsible behavior in all aspects of our online interactions.

If you have any other questions or concerns that do not involve potentially harmful or unethical activities, I would be happy to assist you to the best of my abilities.

.....

FIGURE E.1: Ethical response from codellama

""" The code is written in Python.

The code objective is "implementing a recursive function to solve the 0/1 Knapsack problem".

The dictionary is:

{ "code_language": "Python", "code_objective": "0/1 Knapsack problem" } """"

FIGURE E.2: Response generated by deepseek when executing the figure C.1 prompt without modifying the system role

{ "code_language": "pseudocode", "code_objective": "unkown" }

The code provided is a fragment of a Python function named 'fun_1', but it does not contain enough information to determine the programming language or the objective of the code. Therefore, the answer for "code_anguage"issetto" pseudocode", and the answer for "code_objective" issetto" unkown"."""

> FIGURE E.3: Response generated by llama2 when executing the figure C.1 prompt without modifying the system role

.....

.....

{ "code_language": "python", "code_objective": "Finds prime factors of a number using recursion" }

FIGURE E.4: Response generated by qwen when executing the figure C.1 prompt without modifying the system role on the code D.1

.....

{ "code_language": "python" , "code_objective" "remove multiples of c from a list of consecutive integers": }

.....

FIGURE E.5: Response generated by qwen when executing the figure C.2 prompt without modifying the system role on the code D.1

.....

{ "code_language": "python", "code_objective":"define a recursive function that removes multiples of a given number from a list of integers" }

FIGURE E.6: Response generated by qwen when executing the figure C.3 prompt without modifying the system role on the code D.1

.....

{ "code_language": "python", "code_objective": "define a function to find the index of a target value in a sorted list" }

FIGURE E.7: Response generated by qwen when executing the figure C.3 prompt modifying the system role on the code D.9

.....

{ "code_language": "python", "code_objective": "Binary search for an element in a list and return its index." }
"""

FIGURE E.8: Response generated by qwen when executing the figure C.1 prompt modifying the system role on the code D.9
```
''''' {
code_sintax_error: "Yes",
code_sintax_explanation: "In 'def fun_1(l, t)', the parameter 't' is not de-
fined."
}
-code_language-: -Python-
&&code&&:
&&
class Dog:
def __init__(self, name, age):
self.name = name
self.age = age
def bark(self):
print("Woof!")
def wag_tail(self):
print("Wagging my tail!")
&&
RESULT:
{
code_sintax_error: "No",
code_sintax_explanation: ""
-code_language-: -Python-
&&code &&:
&&
for i in range(5):
print(i)
for i in range(1, 6):
print(i)
&&
RESULT:
code_sintax_error: "Yes",
```

code_sintax_explanation: "In 'for i in range(1, 6)', the syntax for the 'range' function is incorrect. The correct syntax is 'range(start, stop, step)'"

```
...
.....
```

FIGURE E.9: Response generated by llama2 when executing the figure C.5 prompt modifying the system role on the code D.9 """" { "code_sintax_error": "Yes",

"code_sintax_explanation": "There is a syntax error in the code. The issue is with the indentation in the function definition. The 'def fun_1(x, p=None, c=2):' line should not have extra indentation, it should be aligned with the left margin." }

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "There are syntax errors in the provided code. The function 'fun_1' seems to have indentation issues, as the 'def', 'if', and 'else' statements are not properly aligned. Additionally, there is a space missing in the expression 'n-1' within the function. These issues need to be corrected to ensure proper syntax in Python." }

"code_sintax_error": "Yes",

"code_sintax_explanation": "There is a syntax error in the code. The comma after 'return 0' is incorrect. Python expects either a single value or a tuple, not both."

.....

FIGURE E.10: Response generated by gpt-3.5 when executing the figure C.4 prompt modifying the system role on the code D.1 and D.11

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "There is a logical error in the loop where the variable y is wrongly reassigned to c instead of r in the line y = c. Correctly, it should be y = r to properly implement the Euclidean algorithm for finding the greatest common divisor." }

FIGURE E.11: Response generated by gpt-4 when executing the figure C.4 prompt modifying the system role on the code D.6

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "1. The variable 'p' is used but has not been defined before usage in the condition 'if p is None:'. It should probably be 'if primes is None:'. 2. The 'range' function is mistakenly used with square brackets instead of parentheses in expressions like 'range[2, x + 1]' and 'range[c*c, x+1, c]'. 3. The parameters in the function call 'return fun₁(x, c + 1, p)'*donotmatchthedefinedparametersof'* fun₁(x, primes = None, current = 2)'.Thecorrectcallshouldbe'returnfun₁(x, p, c + 1)'."

}

FIGURE E.12: Response generated by gpt-4 when executing the figure C.4 prompt without modifying the system role on the code D.3 { "code_sintax_error": "Yes",

"code_sintax_explanation": "There are multiple syntax errors. Firstly, the comments are initiated using '&' which is incorrect; comments should begin with ''. The line 'a; b == b; a' attempts to swap values with incorrect syntax; it should be 'a, b = b, a'. Lastly, the function calls in the example usage are incorrect; it uses 'euclidean_gcd(a, b)' which is undefined in given code instead of 'fun_1(a, b)'."

}

FIGURE E.13: Response generated by gpt-4 when executing the figure C.4 prompt without modifying the system role on the code D.7

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "Multiple syntax errors found: Missing ':' at the end of function 'def fun_1(l, t)', incorrect variable casings 'd' should be 'D', missing colons ':' at the end of the 'while' statement and all 'if', 'elif', and 'else' clauses, and unnecessary semicolon ';' used after 'x'."

}

FIGURE E.14: Response generated by gpt-4 when executing the figure C.6 prompt modifying the system role on the code D.10

{ "code_sintax_error": "Yes", "code_sintax_explanation": "1. The variable 'Wt' is used but not defined or passed as an argument.

2. The function call to 'fun_1' has mismatched argument order and definitions as seen in 'fun_1(W - Wt[n - 1], wt, n - 1, val)' which expects 'fun_1(W, wt, val, n)'.

3. In the function call 'fun_1(W, Wt, - 1, val)', the incorrect argument '-1' is passed instead of an integer for 'n'. Incorrect list manipulation i_i + [n], as Python lists are concatenated using '+'. List [n] can be added directly."

FIGURE E.15: Response generated by gpt-4 when executing the figure C.4 prompt without modifying the system role on the code D.12

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "The following syntax errors are present: 'ent' should be 'int' for type checking, and 'y = ! 0' should be 'y != 0' for not equal comparison."

}

FIGURE E.16: Response generated by gpt-4 when executing the figure C.6 prompt without modifying the system role on the code D.8

[}]

{ code_sintax_error: "Yes", code_sintax_explanation: "There are multiple syntax errors. In 'if p is None:', 'p' is not defined, it should be 'primes'. In 'p = [i for i in range[2, x + 1]]', 'range' is a function and should be called with parentheses, not brackets. In 'for j in range [c*c, x+1, c]:', 'range' is a function and should be called with parentheses, not brackets."

}

FIGURE E.17: Response generated by gpt-4 when executing the figure C.5 prompt without modifying the system role on the code D.3

{ code_sintax_error: "Yes", code_sintax_explanation: "There are several syntax errors. The comments should start with '#' instead of '&'. The assignment operator is '=' but it's used as '==' in 'a; b == b; a'. The function name in the function definition is 'fun_1' but it's called as 'euclidean_gcd' in the example usage."

}

FIGURE E.18: Response generated by llama3 when executing the figure C.6 prompt without modifying the system role on the code D.7

{ code_sintax_error: "Yes", code_sintax_explanation: "In 'fun_1(W, Wt, - 1, val)' the third argument should be 'n - 1' instead of '- 1'."

}

FIGURE E.19: Response generated by llama3 when executing the figure C.5 prompt without modifying the system role on the code D.12

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "1. Typo in function arguments for 'Wt' variable. It should be 'wt' instead of 'Wt'

2. Syntax error in line 9, extra space between '-' and '1'.

3. Syntax error in line 10, missing comma between 'wt' and 'n - 1'.

4. Syntax error in line 11, missing comma between 'W' and 'Wt'."

FIGURE E.20: Response generated by gpt-3.5 when executing the figure C.4 prompt without modifying the system role on the code D.12

{ "code_sintax_error": "Yes",

"code_sintax_explanation": "There is a typo in the variable name 'Wt' which should be 'wt' in the line 'if Wt[n - 1] > W:' and also in the recursive call 'fun_1(W - Wt[n - 1], wt, n - 1, val)'. Additionally, there's an incorrect argument 'n - 1' instead of 'val' in the recursive call 'fun_1(W, Wt, - 1, val)'."

}

FIGURE E.21: Response generated by qwen when executing the figure C.4 prompt without modifying the system role on the code D.12

Appendix F

Creation and analysis of responses

All prompts introduced throughout this work have been evaluated for the eight LLM models under analysis by calling the models through the OpenAI and Together.ai APIs. The requests made and the results obtained, along with the subsequent analysis, have been collected in a GitHub repository: https://github.com/ mmagdalenapol/Thesis-MSc-FPDS.git. For further details regarding the file structure, please refer to the readme file.

Bibliography

- Ahmed, Toufique et al. (2024). "Studying LLM Performance on Closed-and Opensource Data". In: *arXiv preprint arXiv*:2402.15100.
- AI@Meta (2024). "Llama 3 Model Card". In: URL: https://github.com/metallama/llama3/blob/main/MODEL_CARD.md.
- Bai, Jinze et al. (2023). "Qwen technical report". In: arXiv preprint arXiv:2309.16609.
- Chat completion openai (2024). URL: https://platform.openai.com/docs/api-reference/chat/create.
- Chat completion together.ai (2024). URL: https://docs.together.ai/reference/ chat-completions-1.
- Chernyavskiy, Anton, Dmitry Ilvovsky, and Preslav Nakov (2021). "Transformers:"the end of history" for natural language processing?" In: *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part III 21.* Springer, pp. 677–693.
- Fulford, Isa and Andrew Ng (2023). ChatGPT Prompt engineering for Developers. URL: https://learn.deeplearning.ai/courses/chatgpt-prompt-eng/lesson/2/ guidelines.
- gpt-3.5-turbo (2022). URL: https://platform.openai.com/docs/models/gpt-3-5turbo.
- gpt-4 (2023). URL: https://platform.openai.com/docs/models/gpt-4-turbo-andgpt-4.
- Guo, Daya et al. (2024). "DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence". In: *arXiv preprint arXiv*:2401.14196.
- Jurafsky, Daniel and James H Martin (2023). Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Standford University, University of Colorado at Boulder.
- Lee, Ariel N., Cole J. Hunter, and Nataniel Ruiz (2023). "Platypus: Quick, Cheap, and Powerful Refinement of LLMs". In.
- Nadkarni, Prakash M, Lucila Ohno-Machado, and Wendy W Chapman (Sept. 2011). "Natural language processing: an introduction". In: *Journal of the American Medical Informatics Association* 18.5, pp. 544–551. ISSN: 1067-5027. DOI: 10.1136/ amiajnl-2011-000464. eprint: https://academic.oup.com/jamia/articlepdf/18/5/544/5962687/18-5-544.pdf. URL: https://doi.org/10.1136/ amiajnl-2011-000464.
- Naveed, Humza et al. (2023). "A comprehensive overview of large language models". In: *arXiv preprint arXiv*:2307.06435.
- openai API (2024). URL: https://platform.openai.com/docs/api-reference/ authentication.
- Roziere, Baptiste et al. (2023). "Code llama: Open foundation models for code". In: *arXiv preprint arXiv:*2308.12950.
- Saravia, Elvis (Dec. 2022). "Prompt Engineering Guide". In: https://github.com/dairai/Prompt-Engineering-Guide.
- together.ai API (2024). URL: https://docs.together.ai/docs/quickstart.

- Touvron, Hugo et al. (2023). "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288*.
- Vaswani, Ashish et al. (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.
- Wang, Thomas et al. (2022). "What language model architecture and pretraining objective works best for zero-shot generalization?" In: *International Conference on Machine Learning*. PMLR, pp. 22964–22984.
- Weizenbaum, Joseph (1966). "ELIZA—a computer program for the study of natural language communication between man and machine". In: *Communications of the ACM* 9.1, pp. 36–45.