On the Efficacy of Large Language Models as Models of Computation

1st Sebastian Just
College of Computing & Informatics
Drexel University
Philadelphia, USA
sj992@drexel.edu

2nd Greg Morgan
College of Computing & Informatics
Drexel University
Philadelphia, USA
gm655@drexel.edu

3rd Frank Ottey
College of Computing & Informatics
Drexel University
Philadelphia, USA
fmo28@drexel.edu

4th Trevor Pawlewicz
College of Computing & Informatics
Drexel University
Philadelphia, USA
tmp365@drexel.com

***Index Terms*—Language models, Computation, Software security, WebAssembly**

***Abstract*—The rise of ChatGPT and the increasing effectiveness of large-language models, driven in part by the efficient neural transformer architecture defined in the seminal paper [VSP+17], has captured significant public consciousness. As the use of the transformer architecture to encode domain-specific sequence-to-sequence information, as seen OpenAI's work on the automatic speech recognition system, Whisper, [RKX+22], has been met with success, we sought to apply the self-same techniques to virtual machine bytecode sequences. As large language models have shown promise in both source code generation, source code summary, and source code evaluation, and considering prior work in the area, we were hopeful that language models would prove to be capable of modeling basic computation. In particular, we wanted to evaluate if Web Assembly bytecode sequences encoded by contemporary language models using the transformer architecture maintained enough semantic information that a model of computation emerged. What we found however, is that there are numerous challenges in attempting to use language models to develop a model of computation. And while language models have worked well on program source code, their ability to reason about bytecode sequences can be quite limited as difference between source code and bytecode sequences are non-trivial. Therefore, instead, in this paper, we we aim to explore these differences, the challenges in utilizing language models to develop models of computation, and evaluate how and where these models break down in their performance.**

I. Introduction: Language Models as Models of Computation

A. Language Models as Programmers

The recent effectiveness and proliferation of the transformer architecture in language models has given rise this year to a number of both open and closed source generative pre-trained transformer networks; large autoregressive language models with state-of-the-art performance in many multi-task settings dealing with both human language and source code. Perhaps no tool has captured more public attention than that of OpenAI's ChatGPT. This model has proven capable of analyzing source code, writing source code, and explaining existing source code to users. Much has been made of this models and its open-source kin's effectiveness in these tasks, such that there is a notion that these language models themselves are quickly becoming capable "programmers" in their own right.

With all the recent success in this area, our research team had originally intended to explore potential extensions of these applications of language models as "programmers". If language models could function as programmers, we hope that we leverage these models' intuition or model of computation itself, specifically with respect to the task of automatically analyzing and detecting attributes of specific code sequences. While much existing research and evaluation has been spent on these models' capability to operate on program representations as source code, less attention has been placed on their ability to operate on more constrained program representations, such as abstract syntax trees, or compiled bytecode / assembly code. We had hoped to evaluate the effectiveness of these kinds of models in this area by utilizing existing language models to encode a specific bytecode format sequence, Web Assembly, and then use supervise learning to train a classifier to predict whether or not program sequences contained any malicious or problematic bytecode sequences. A similar approach has even been attempted by others in the past. See [RKO20] or [KZP+13] for inspiration.

To this end, our initial research was delving into the following key question: Could large language models developed for analyzing sequences of text be naively applied to bytecode sequences and still retain the ability to reason about the sequences and models of computation regarding those sequences? Our original hypothesis was that indeed, today's contemporary large language models do indeed include within them, models of computation, and could be used to analyze program sequences, fragements, and reliably function as pan-lingual "programmers" after a fashion, regardless of the target language in question, whether it was a high-level source code language like Python, or a low-level bytecode format languages like Web Assembly.

B. What is Web Assembly?

Web Assembly is a novel, platform-agnostic virtual-machine and bytecode instruction format that has been designed specifically for the domain of web development. Specifically, the VM and bytecode are intended for embedding within the context of a web browser, while providing a robust and secure sandbox for running portable and efficient code on client devices. As a language and platform-agnostic compilation target, source code of any language could theoretically be compiled down to a Web Assembly bytecode sequence. This would provide increased performance and security for existing web applications while also allowing existing code written for native applications, such as C, C++, Rust, etc. to be ported to the web platform.

Due to its focus on embeddability, the bytecode format of the virtual machine is structured to be compact and fast to load. This would make it suitable for running client-side applications that demand near-native performance such as computational simulations, graphics rendering, and/or virtualization. The bytecode generated by compiling code to Web Assembly is designed to be platform-independent and can be executed on various architectures. This characteristic makes it possible to run consistent and reliable experiences across various devices and browsers without the need for constant modifications and adjustments.

Finally the Web Assembly's virtual machine platform has the capability to interface with JavaScript and the document object model (DOM) present in web browsers, which offers developers flexibility in integrating it with, and deploying it to, existing web applications and platforms. This interoperability allows developers to optimize performance-critical software components while retaining the overall structure and functionality of their client-based web software.

C. Why use LLMs for Bytecode Analysis?

LLMs have shown initial promise at generating and evaluating code for various programming languages. Historically, the analysis of code is generally performed either statically or dynamically: Static analysis tends to have less reliable coverage while dynamic analysis tends to be non-performant. (Ie. Compiler analysis vs. products like Valgrind and other sanitizers). Dynamic analysis requires running code, which, if the code is untrusted / malicious is problematic on it's own.

Thus we hope to use LLMs to analyze bytecode fragments before execution by the runtime environment (usually the virtual machine) addressing the shortcomings of the dynamic and static analysis but without the risk of actually executing the code.

D. Similar / Prior Work

Others have ventured into this area before and we want to highlight simmilar work that has influenced our research: Huaiguang et.al. analyzed JavaScript bytecode for malicious intend [RKO20] which is close to our research area but limited to (platform-specific) code. Ao et.al. showed how to improve the code recognition of a specific malicious program in JSP by leveraging XGBoost [PFZ+22] which is promising but uses transfer learning for a specific target.

Finally, Ashizawa et.al. researched how to apply code analysis for vulnerability detection on Ethereum smart contracts. [AYCO21]

E. A Contemporary Adaptation

Based on Huaiguang et.al's work, our original research aimed to adopt their approach, leveraging contemporary open-source LLM infrastructure in place of word2vec as the sequence-to-sequence encoder-decoder step for bytecode sequences. In addition, as apposed to operating on the VM-specific V8 bytecode, we proposed focusing on Web Assembly bytecode instead, as a cross-platform, performant compilation target for the web. Our intent was to follow through on the original experiment's plan, with further featurization of the inputs, finally culminating with a supervised classification layer to build a system for vulnerability detection..

We intended for this adaptations to allow our work to be more applicable for real-world use cases: as more performance-sensitive code is required to run client-side for privacy and security reasons, there is a need to have tools to automatically verify the transmitted code as non-malicious. This is especially important for our work in the field of video streaming over IP networks on untrusted playback devices.

II. Challenges: Language Models vs. Models of Computation

Unfortunately, the original research goal proved quite ambitious for a 10-week timeline. We ran into numerous challenges during our research and investigation that forced us to reckon with theoretical limitations inherent in the chosen approach. Particularly, we ran into three key challenges: the nature of language and tokenization, the mis-match between computation models and auto-regressive langauge models, and the inherent lack of available Web Assembly bytecode datasets of sufficient size that could be used to produce a proper sequence-to-sequence encoder-decoder. In addition, we were faced with the very real challenge of gaining access to the necessary compute resources to see our vision come to fruition. This section explores these challenges in more detail and finally outlines the key research questions we aim to answer in this paper.

A. The Nature of Language and Tokenization

The first key challenge encountered in our approach is that for sequence-to-sequence encoding to work efficiently and with any accuracy, a tokenizer needs to be applied to the input sequences, breaking up the input characters into tokens that the model then has some inherent understanding of. Our initial approach was to leverage existing open-source language models to perform this tokenization. What we found, however, was that this tokenization activity, particularly when working with Web Assembly bytecode sequences, was not effective. Because tokenization is such a sensitive process, what we found is that good tokenization could greatly impact various pre-trained models output and performance on even a very limited sample selection of inputs.

Given that Tokenization is contextual, the meaning of any given token is dependent on the tokens around it and different characters, words, or even grapheme sequences could tokenize in different ways. The tokenization of existing open-source language models is dependent on the input training data. What we found during our research is that while many existing open-source language mdodels have been trained on a variety of textual data on the web, the amount of code sequences those language models have been trained on is far less. In addition, there is an even more limited selection of bytecode sequence data in the training data sets and an even further limited selection of Web Assembly bytecode sequences in the training data set.

Fundamentally, this means that very few existing open-source language models were well-equipped to properly tokenize Web Assembly bytecode without extensive modification. Furthermore, testing with existing tokenization efforts revealed that existing models often semantically confused Web Assembly token sequences with similar assembly code sequences or or with source code sequences from the Lisp family of languages.

A final challenge that emerged was that while existing open-source language models have worked well with tokenizing and understanding both natural language and source code, Web Assembly byte code sequences operate at a much lower level of abstraction. More than a language, these byte code sequences define computation, which includes state, sequencing, and specific constraints. Given that generative pre-trained transformer language models are built to stochastically predict the next token in a sequence of tokens, while this task works well when operating on natural language or high-level source, we observed that as the abstraction level the sequence was reduced, and more constraints were introduced into the target language, hallucinations and mis-predictions had a much greater negative impact as it was increasingly easier to produce not only semantically incorrect sequences, but also easier to produce syntactically incorrect sequences. This finding, its implications and further work are explored more in section III.

B. Computation Models are not Autoregressive

Another key challenge our research uncovered is that while contemporary generative pre-trained transformer language models are autoregressive, fundamentally, models of computation are not. Generative autoregressive language models are built by using only the previous tokens within a sequence to predict the next upcoming tokens in a sequence. While this appears to work well for natural language, it seems to be less effective in the face of languages / computation models with low-level levels of abstraction like assembly or bytecode. Specifically, the state of any given token is dependent on not just the token before, but also the tokens after it. In addition, any given token proceeding token sequence doesn't necessarily carry enough contextual information to drive the generation of further tokens in the sequence. As such, these models ability to build a model around the language / computation is limited. An example of this phenomen is a function signature. Given a high-level source code programming language like Python, and a function signature, with enough Python code in the training datasets, contemporary models are fairly effective at generating the correct function given a specification and the function signature. The same task, applied to Web Assembly source code, however rarely produces correct results. This is a consequence of the previously aforementioned mis-match.

Considering the above mis-match, another issue is that low-level assembly / bytecode languages like Web Assembly are fairly inexpressive. This means that it takes a great many tokens to express a given computation, constraint, or expression. Because of this, the input sequence length required for these models to produce any sort of reasonable output is much higher than that of similar languages. This is a particular problem with large generative, pre-trained transformer language models as the input sequence length is often capped and/or limited during training in order to efficiently train the large model on the vast quantities of input data. Asking the model to correctly generate the requisite number of tokens for a large input sequence is prohibitively expensive from a computational point of view.

C. Dataset Size and Availability

These two challenges taken together, specifically the lack of high-quality tokenization for the particular language in question and the missing flexibility in autoregressive models to deal with low-level abstractiopns present in assembly / bytecode languages like Web Assembly could theoretically have been managed by some high-quality transfer-learning. Indeed, our first thought encountering these issues was to "simply" update an existing open-source language models with a dataset of Web Assembly bytecode sequences in an attempt to update the tokenizer, and to then leverage the existing model to continue on with our experiment. What we encountered, however, is that to achieve their high-quality results, existing language models are trained with massive amounts of data, often obtained by researchers and organizations scraping large sections of the public internet. This strategy works really well for gathering many sequences of common languages and their tokens, such as English, or, if scraping source code repository websites like Github, where popular programming languages like Python often utilize a great many English language tokens in their source code.

Unfortunately, in our case, the novelty of Web Assembly bytecode sequences and it's abstraction level proved to be fatal challenges when it came to procuring sufficient training data for the transfer learning activity. Because Web Assembly is a rather novel invention, there are not a lot of example high-quality byte code sequences for it, nor have existing language models encountered much of it in their training datasets, leading to poor-quality tokenization and semantic understanding. In addition, due to its low abstraction level (i.e. it's a bytecode language vs. a source code language) it is often not actually written and used by humans; rather it is instead more often produced by a compiler after being fed in a higher-level langauge. Taken together, these two factors lead to there being only a very few repositories that contain samples of Web Assembly bytecode sequences, certainly not enough to actually produce an effective tokenization / language model from existing sources.

D. Resource Acquisition and Environmental Impact

The final critical challenge our research faced was one of resource acquisition. As we searched to obtain and build a dataset of sufficient size for the transfer learning activity, in parallel, we searched to obtain sufficient resources to run the transfer learning activity on an existing pre-trained, open-source, generative large-language model. What we found is that these models are, in practice, much larger that can reasonably be expected to fit within the confines of the compute of modern desktop / laptop computers. In addition, with all the recent interest and activity in the language modeling space, availability of vertical scaling resources on large cloud compute platforms is quite limited as contention for these limited resources is at a peak.

The effect of this was that even if we were able to obtain a proper dataset to work around the challenges of tokenization and minimize the negative mis-match between auto-regresive language token-production and computational modeling, we would have no computation resources with which to execute the transfer learning task. And even if we were able to obtain a resource, these resources would be prohibitively expensive to run for any length of time, which would be a baseline requirement for training a model well enough to achieve consistent results.

Finally, for the environmentally conscious among us, the impact of all this ML-training is not negligible for the environment. As for any Information and Communication Technology (ICT) service, there are at least four sources of CO2 equivalent emission sources that should be taken into account to assess the environmental impact of computational experiments: 1/ production of hardware equipment: router, PC, server; 2/idle use of the hardware; 3/ dynamic use of the hardware; and 4/ end of life of equipment. [BGNL21]

E. Research Questions

Faced with numerous challenges, our team realized that we needed to reduce the scope of our research in order to meet our 10-week timelines. In addition, what we found was that after several weeks of research, we were left with more questions than answers, specifically regarding where exactly the performance of existing large language models breaks down when used to evaluate source code and/or bytecode sequences for the purpose of building a model of computation. To this end we realized that we wanted to explore and measure this distinction with the remaining time and resources available to us. Therefore, we came to formulate the following four research questions

RQ1. How effective are LLM's at being "programmers" in the small case?.

RQ2. How effective are LLM's at translating a given computation between languages?

RQ3. To what extent are LLMs simply understanding the semantic meaning of the tokens vs. understanding computation?

RQ4. How effective are LLMs at understanding computation given the changing context of language?

The first two research questions above, respectively, will be further explored and evaluated in section III of this paper, where we will perform the analysis between C++ source code and Web Assembly bytecode sequences. The second two research questions respectively, will be further explored in section IV of this paper, using Python source code as a basis for evaluation.

III. Evaluating LLMs with WebAssembly

In this section, we will evaluate LLMs ability to work with WebAssembly Text (WAT). To do this, we will prompt ChatGPT 3.5 to interpret and generate WebAssembly Text code. In this case, we are evaluating ChatGPT's ability to translate (compile and decompile) code between C++ and WAT format. We chose to work with WebAssebmly Text instead of the raw binaries because they are much more human readable to validate and understand. Additionally, since LLMs are trained to work with human language rather than machine code, we hypothesized they would perform better with the assembly language rather than the raw binary files. To evaluate our model the prompts that were use are as follows:

- "Can you translate this C++ code to WebAssembly Text? <C++ Code>"
- Can you translate this WebAssembly Text code to C++? <WAT Code>".

Because ChatGPT has the ability to use results from previous prompts from the same chat environment as context to improve it responses, we asked each prompt in an isolated chat environment to insure no additional context was provided to the model.

A. Methodology

We evaluated ChatGPT using the following basic C++ programs:

- Sum of Two Integers
- Factorial Function
- Fibonacci Sequence
- IsPrime Function

These functions range from extremely basic in the case of Sum of Two integers which simply defines two variables, adds them together and returns the response to functions like Fibonacci which require basic use of for loops and conditionals. All function were written in less than 20 lines of C++ code. Importantly, these are also very common functions that are likely well represented in ChatGPT's training data set. Therefore, if ChatGPT is unable to translate these common functions, we can infer that it is unlikely to do better with more complex or unique code. When compiling C++ to WebAssembly, external libraries are compiled in their entirety and represented the resulting binaries. This provided WAT codes that were far to long and distracted the model from the intended purpose of the prompt. They also made it much harder to visually validate the WebAssembly code. Because of this, our sample functions are written without the use of any external C++ libraries.

TABLE I
RESULTS

Code	Readable	Compilable	Runnable	Usable
Sum Two Ints WAT	✓	✓	✓	✓
Sum Two Ints C++	✓	✓	✓	✓
Factorial WAT	✓			
Factorial C++	✓	✓	✓	✓
Fibonacci WAT	✓	✓	✓	✓
Fibonacci C++	✓	✓	✓	
isPrime WAT	✓			
isPrime C++	✓	✓	✓	

To evaluate the performance of ChatGPT, we defined the following criteria:

- Readable: Does the code generate look reasonable to the human eye?
- Compilable: Does the code compile?
- Runnable: Does the code run?
- Usable: is the output given the correct expected output or something different? Is the resulting code actually usable?

For our evaluation we used WasmFiddle [https://wasdk.github.io/WasmFiddle/] and WasmExplorer [https://mbebenita.github.io/WasmExplorer/] for compiling and running our C++/WAT code respectively.

B. Results & Analysis

Our results can be seen in Table I. They suggest that ChatGPT already possess some ability to interpret and generate WebAssembly Text. Across the board, we found that the code produced was visually readable however we did see some sort of error in most cases. Common errors for translating C++ to WAT where parsing syntax errors such as "wasm text error: parsing wasm text at 4:6". This suggests that ChatGPT is familiar enough with the syntax of WebAssembly text to generate something that looks correct but not consistent enough to generate code that actually compiles. On the flip side, when translating WAT to C++, ChatGPT consistently generated code that compiled and ran but in some cases generated the wrong output. For example, our C++ code generated for the `isPrime` function should return a value of `0` or `1` (representing False or True) but in same cases returned `2` instead. This implies that while ChatGPT is familiar enough with the syntax of C++ to generate code that actually runs, it was not able to generate code that was actually usable.

```
(module
  (func (param i32) (result i32)
    (local i32)

    (local.set 1 (i32.const 1))
    (loop
      (local.get 0)
      (i32.const 1)
      (i32.gt_s)
      (br_if 1)
      (local.get 1)
      (local.get 0)
      (i32.mul)
      (local.tee 1)
      (i32.const 1)
      (i32.sub)
      (local.set 0)

      (br 0)
    )

    ;; return result
    (local.get 1)
  )

  (func (result i32)
    ;; int main()
    (i32.const 5)
    (call 0)
  )
)
```

Listing 1. Generated WAT

Looking at the generated WAT in listing 1, it is actually shorter and more concise than the compiled WAT. In this case, this code does not compile which suggests that ChatGPT may be missing some additional information when translating to WAT. When generating WAT, we found that the resulting code either did not compile or ran and worked as expected. This suggests that the primary issue is that ChatGPT is not familiar enough with the WAT syntax to consistently generate compilable code. But that when it does generate compilable code, that it is able to translate the intention of the C++ code effectively to WAT format. Because WebAssembly is relatively new and less common, simply exposing ChatGPT to more WAT code in its training data set could significantly improve ability to generate compilable WAT.

Research from Rémi Louf et.al. showed the regular expression guided generation greatly reduced the number of syntax errors [WL23]. We believe that a similar approach of using a WebAssembly Context Free Grammar to assist in generating code or using a compiler in a reinforcement learning training loop could greatly improve its performance. Since ChatGPT is already able to pick up on a lot of semantic meaning in code, we believe that passing this output through a compiler filter would help improve the odds that the generated output from ChatGPT can be run out of the box with little or no refactoring.

```cpp
#include <iostream>

int32_t factorial(int32_t n) {
    int32_t result = 1;

    if (n >= 2) {
        do {
            result *= n;
            n -= 1;
        } while (n >= 2);
    }

    return result;
}

int32_t main() {
    int32_t result = factorial(5);
    return result;
}
```

Listing 2. Generated C++

Similarly with previous listing, when generating C++ from WAT (Listing 2) the resulting C++ tended to be longer and more verbose than the target C++. The model tended to be very literal with the translation. In the case of Factorial, we see that the model is trying to allocate memory as well as specific sized integers (int32_t) rather than generalizing to code to use standard integers. This is likely because the model is trying to translate the code line by line or token by token rather demonstrating a deeper, holistic understanding of what the code is doing and generating the appropriate output.

Though ChatGPT was far from perfect when translating between WebAssembly and C++, we found that it was almost always able to pick up on the intent of the code even if the code generated was not syntactically correct or gave the wrong response. Since the model was struggling with the underlying logic but always understood the intent, this lead us to wonder if the model was using key works such as variable and function names in the code to provide more context and help with interpreting the given code and generating new code.

This leads us to the next section of this paper where we will be attempting to answer the follow question: Is ChatGPT using key words in the WAT such as "Factorial" or "Fibonacci" to gain additional context or does it have a deeper understanding of the logic and language structure?

IV. Evaluating LLMs with Python

A. Assessing LLMs current ability to interpret Python Code

The following approach is used to compare the performance of LLMs to estimate their performance on programming languages. The guiding assumption is that the LLM only understands the programming code as it uses keywords of the English language and thus fairs best with a programming language like Python: Many descriptive keywords from the English language and few pure symbols that could be interpreted as token boundaries of the LLM only.

Thus we developed the following experiment:

1) Select common algorithms from the Computer Science field:
   - Bubble-Sort
   - Quick-Sort
   - Reverse String
   - Longest common sub-string
2) Implement each algorithm in a common way
3) Let the LLM analyse the code with the prompt

   what does this code do? ``` <actual code> ```
4) Obscure the code by replacing all method and variable names with simple letters and nondescript method names like `foo()` or `bar()`
5) In a new session, let the LLM analyse the updated code again with the same prompt structure as before
6) Obscure the code by replacing simple conditions with their lambda equivalent - allowing for simple inlining by the compiler
7) In a new session, let the LLM analyse the updated code again with the same prompt structure as before
8) Obscure the code further by replacing the Python core methods like `len()` with self-written `for` loops or similar
9) In a new session, let the LLM analyse the updated code again with the same prompt structure as before
10) Obscure the code further by adding empty loops to implementation and consuming method of variables to the code that do not alter the variables of the original algorithm
11) In a new session, let the LLM analyse the updated code again with the same prompt structure as before

To evaluate the performance of the LLM, we defined the following criteria:

- algorithm recognized: The response from the LLM names the the algorithm in question correctly and briefly describes its use.
- purpose recognized: The response from the LLM recognizes the overall intention (ie. sorting, working with strings) but doesn't further qualify
- not recognized: The response from the LLM is non-specific to the purpose of the code

ChatGPT-3.5 was used as the LLM to run these experiments.

As it is not possible to look-up the LLM's responses of this experiment and there is no consensus yet on how to cite generative AI artifacts [Chaa] [Chab] [McA] we refrained from citing them as references. Instead all ChatGPT responses from Monday, 22 Aug 2024 have been captured in the appendix and are referenced accordingly throughout the chapter.

B. Experiments

1) Quick-Sort: This experiment on the model's abilities to recognize a Quick-Sort algorithm. The Quick-Sort is implemented in its recursive version taking advantage of Python's `//` operator to calculate the pivot point as per listing 5.

The response from the LLM according to the experiment plan above is correct:

> The provided code is an implementation of the Quick Sort algorithm, which is a widely used sorting algorithm known for its efficiency. (B-A1)

and all parts of the code are properly described in their function for the algorithm.

Applying the first level of obfuscating by using single character variable names and nondescript method (Listing 6) results in the LLM's still correct response:

> This code defines a recursive function named foo that takes a single argument bar, which is assumed to be a list of comparable elements (e.g., numbers, strings, etc.). The purpose of this function is to perform a sorting algorithm using a divide-and-conquer approach known as QuickSort. (B-A2)

and identifies the implemented algorithm followed by a detailed description of each line of the code.

The next level of obfuscating is done by extracting all Boolean comparisons into their own lambda implementation. The goal is to keep the implementation in a form that the compiler is very likely to inline those changes and result in the same executed code though this was not verified on the machine code level (Listing 7).

The response from the LLM does not mention Quick-Sort anymore but the general idea of the algorithm is still described:

> In essence, the code is implementing a recursive sorting algorithm where the pivot element is chosen as the middle element of the input list, and the list is divided into three parts. (B-A3)

The next level of obfuscating is done by even removing Python key functions like then **`len()`** method (Listing 8) At this point the LLM's response resorts to a pure line-by-line description of the code:

> This code defines two functions, `fuzz(foo)` and `foo(bar)`, which appear to manipulate and process a list-like data structure called bar. (B-A4)

and the overall idea of Quick-Sort in the implementation is still somewhat recognizable in the response:

> Essentially, the function recursively divides the input list into three segments: those smaller than the pivot, those equal to the pivot, and those greater than the pivot. It applies the same process to the segments smaller and greater than the pivot, creating a recursive tree structure. Finally, it combines the results of the recursive calls along with the elements equal to the pivot to form the final output. (B-A4)

The last level of obfuscating is done by adding loops that do not alter the output of the method itself but use intermediate values of the implementation without changing the output of the implementation (Listing 9). This causes the LLM's response resorts to a pure line-by-line description of the code and no mention of any sorting is provided:

> In essence, this code seems to be constructing a string z by appending various representations of lists and elements based on their relationships with the middle element a of the input sequence bar. The behavior is complex and somewhat convoluted, and it's not immediately clear what practical purpose this code serves without more context. (B-A4)

2) Bubble-Sort: For the Bubble-Sort sample, a classic and straight forward swapping implementation is used (Listing 10), This common implementation of Bubble-Sort is recognized by the LLM in the first sentence:

> The provided code is an implementation of the Bubble Sort algorithm, a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. (B-B1)

The obfuscated version with replaced variable and method names (Listing 11) is also still recognized:

> The given code defines a function foo that implements the bubble sort algorithm.

The usage of lambdas for the Boolean conditions (Listing 12) starts to confuse the LLM. While the line by line description of the implementation remains correct, the overall intention of the code is only understood as some kind of sorting algorithm:

> In summary, the code appears to be a sorting algorithm implementation that uses three lambda functions (a, b, and c) to define certain conditions and arithmetic operations. The sorting logic within the loops is a bit unconventional and not immediately recognizable as a standard sorting algorithm like bubble sort, insertion sort, etc. It seems to involve swapping elements based on the conditions defined by the lambda functions. The exact behavior and efficiency of this sorting algorithm would require further analysis and testing. (B-B3)

By replacing the **`len()`** method calls with a self-written method (Listing 13) results in the response of the LLM to be very generic about the overall implementation behavior:

> In summary, the foo(bar) function performs a series of operations involving nested loops and conditional statements. It swaps elements in the input bar list based on the conditions specified by the lambda functions a, b, and c, as well as the logic within the loops. The code might be part of a larger program and could have a specific purpose depending on the context in which it's used. (B-B4)

By finally adding loops that do not contribute to the output (Listing 14), the LLM's responses shifts its focus to the "empty" loops and the actual purpose of the implementation is lost:

> In summary, the code is quite complex and appears to be manipulating lists or sequences (bar) based on

> various nested loops and lambda functions. The exact purpose and intended behavior of the code depend on the specific inputs provided to the foo function and the values within the bar list. The lambda functions and nested loops seem to be performing a combination of string concatenation and value manipulation operations. (B-B5)

3) Reverse string: Reversing a string is one of the shorter algorithms which have a huge bandwidth of possible implementations. A simple list-based approach is used for this experiment to allow for the obfuscating steps later on (Listing 15).

The response from the LLM for this implementation identifies the algorithm right away:

> takes an `input_string` as an argument and returns the reversed version of that string. (B-C1).

Also replacing the method and variable names as before with (for humans) meaningless characters (Listing 16) returns a similar response as before from the LLM identifying the algorithm correctly:

> The purpose of the function is to reverse the characters in the input string bar using a loop. (Listing B-C2)

As there exist no conditional logic or similar constructs in the implementation, the code was changed to operate on reversing a list (Listing 17). This is recognized rightfully by the LLM:

> This code attempts to reverse the order of characters in the input string bar using a somewhat convoluted process (B-C3)

but a bug is mentioned by the LLM which is not present in the code.

Also replacing the `len()` method (Listing 18) as in other samples causes the response from the LLM to (falsely) find more issues with the presented code and has less of an understanding of what the code actually does:

> the function returns the string a, which would be the result of concatenating characters from the input bar in a specific order. (B-C4)

The last iteration of this sample introduces empty loops that have no impact on the actual result from the algorithm (Listing B-C5). This causes the LLM to respond just in a generic manner about the code and not expressing the general intentions of the code:

> In summary, the foo(bar) function seems to process each element in the iterable bar by manipulating the contents of a list l and a string z. The final output is a string that reflects the changes made during the loop iterations

As before, all responses include line-by-line description of the implementation though this provides little information about the overarching goal of the presented code.

4) Longest common sub-string: Determining the Longest Common Sub-String (LCS) is a common algorithm and used as the final experiment. The plain implementation (Listing 20) is easily understood by the LLM and properly reported:

> calculates the longest common substring between two input strings, s1 and s2. A common substring is a sequence of characters that appears in both strings in the same order. (B-D1)

As seen before, replacing the method and variable names (Listing 21 has no impact on the response from the LLM:

> This function appears to be implementing a common algorithm known as the Longest Common Substring (LCS) algorithm, which is used to find the longest contiguous subsequence that is common to both input strings. (B-D2)

Adding lambdas to express Boolean conditions that can be inlined (Listing 22) seems to have no tangible impact on the ability of the LLM to still identify the algorithm used:

> This function appears to be aimed at finding the longest common substring between the two input strings. (B-D3)

Also re-implementing `len()` (Listing 23) as before has no impact on the LLM to identify the algorithm though it seems less certain:

> This code [...] seem to be related to finding the longest common substring between two sequences of elements. (B-D4)

Finally, adding empty loops and variables to the implementation again (Listing 24) results in the LLM not being able to decipher what the overall intend of the code is:

> The main purpose of the [...] function is to perform some sort of comparison between two input sequences (foo and bar) and return a subsequence of foo based on certain conditions. (B-D5)

C. Findings

A comprehensive summary of the experiments and the respective ChatGPT answer is available in Table II:

The first finding is that obscuring just method and variable names seems to have no impact on the ability of the LLM to identify a given algorithm from the Python code. While this is a pretty simple obscuring technique as it works on search-and-replace method it introduced on longer algorithms a significant mental load for human readers to understand an implementation of an algorithm as the method and/or variable names are not properly named. On the other hand replacing simple conditions with lambda expressions in Python code seems to be significantly harder for the LLM to follow a long. Even though these are also very close to search-and-replace they can be easily understood (and most compilers likely inline them, too) it seems to impact the LLM the most. Introducing dead code into the implementation seems to have a significant impact causing the LLM to lose the context of the actual algorithm. Additionally, re-implementing the common Python method **`len`**`()` seems to have a significant impact on the LLM's ability to identify the purpose of an implementation.

These four observations combined with the fact that all tokens (the symbols in the Python code) are represented by a matrix in the LLM can help explaining that the LLM only

TABLE II
FINDINGS

Modification	Quick-Sort	Bubble-Sort	Reverse string	LCS
None	algorithm recognized	algorithm recognized	algorithm recognized	algorithm recognized
Foorbar'ed	algorithm recognized	algorithm recognized	algorithm recognized	algorithm recognized
Lambda'ed	purpose recognized	purposes recognized	algorithm recognized	algorithm recognized
`len()`'ed	purpose recognized	not recognized	purpose recognized	algorithm recognized
no-op loop'ed	not recognized	not recognized	not recognized	not recognized

matches the statistical distribution of tokens and thus matches known algorithms examples from its training. This would explain why renaming symbols has no measurable effect but introducing "no-op" differences to standard implementation of an algorithm leads the LLM quickly to respond with less and less precise responses.

The outlier remains the LCS implementation: Despite applying all the patterns that distracted the LLM before, the responses from the LLM still clearly identified the LCS implementation in the responses. Chances are this is due to very unique way an LCS implementation distinguishes itself against other algorithms.

It is worth mentioning that the LLM seems to have no understanding of scope: The same variable name in different scopes (ie. methods) seems to be translated to the same token internally and thus causing the LLM to express concerns and (falsely) identify bugs around those variables - even though the variable scope is clearly defined as per the Python language specification [Pyt]. This can be mostly observed with loop index variables.

Also all responses from the LLM always included a line-by-line transcription of the code into English. This part of the response was not included here as it never conveyed any overarching context of the implementation but only a transcription of code into the English language.

V. CONCLUSION

With all the results from our experiments we try to present a conclusion below. Based on all the challenges lined out above we want to caution everyone and remind that more research in the field is nessassary:

A. How effective are LLM's at being "programmers" in the small case?

Our experiments showed a large variety of results. For some (small) cases, the results are fairly effective. For other cases the provided results by the LLM are not usable.

B. How effective are LLM's at translating a given computation between languages?

It depends – level of abstraction and complexity of computation are key factors regarding the ability to perform this task. Overall it seems heavily dependent on how well covered the code/algorithm is in the training dataset.

C. To what extent are LLMs simply understanding the semantic meaning of the tokens vs. understanding computation?

Token replacement in common programming languages does not seem to affect computation reasoning (although we haven't explored the effect in less common programming languages). Though it is unclear if this behavior can be attributed to an understanding of the semantic meaning in the first place or if this is just an outcome of the pattern recognition of the implementation itself.

D. How effective are LLMs at understanding computation given the changing context of language?

This is dependent on the complexity of the computation and the target language. A major obstacle seems to be the lack of understanding of the scope of symbols (variables) in a given implementation of a programming language which in turn leads to false statements about the shown implementation. This suggested that the understanding of both the computation and the semantic meaning is not given.

REFERENCES

[AYCO21] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, *Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts*, 2021. arXiv: 2101.02377 [cs.CR].

[BGNL21] N. Bannour, S. Ghannay, A. Névéol, and A.-L. Ligozat, "Evaluating the carbon footprint of NLP methods: A survey and analysis of existing tools," in *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*, Virtual: Association for Computational Linguistics, Nov. 2021, pp. 11–21. DOI: 10.18653/v1/2021.sustainlp-1.2. [Online]. Available: https://aclanthology.org/2021.sustainlp-1.2.

[Chaa] "Citation, documentation of sources." (), [Online]. Available: https://www.chicagomanualofstyle.org/qanda/data/faq/topics/Documentation/faq0422.html.

[Chab] "Ieee referencing: Generative ai." (), [Online]. Available: https://libraryguides.vu.edu.au/ieeereferencing/generativeAI.

[KZP+13] S. Kim, T. Zimmermann, R. Premraj, N. Bettenburg, and S. Shivaji, "Predicting method crashes with bytecode operations," in *Proceedings of the 6th India Software Engineering Conference*, ser. ISEC '13, New Delhi, India: Association for Computing Machinery, 2013, 3–12, ISBN: 9781450319874. DOI: 10.1145/2442754.2442756. [Online]. Available: https://doi.org/10.1145/2442754.2442756.

[McA] T. McAdoo. "How to cite chatgpt." (), [Online]. Available: https://apastyle.apa.org/blog/how-to-cite-chatgpt.

[PFZ+22] A. Pu, X. Feng, Y. Zhang, X. Wan, J. Han, and C. Huang, "Bert-embedding-based jsp webshell detection on bytecode level using xgboost," *Security and Communication Networks*, vol. 2022, p. 4 315 829, 2022, ISSN: 1939-0114. DOI: 10.1155/2022/4315829. [Online]. Available: https://doi.org/10.1155/2022/4315829.

[Pyt] "The nonlocal statement." (), [Online]. Available: https://docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement.

[RKO20] M. F. Rozi, S. Kim, and S. Ozawa, "Deep neural networks for malicious javascript detection using bytecode sequences," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–8. DOI: 10.1109/IJCNN48605.2020.9207134.

[RKX+22] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, *Robust speech recognition via large-scale weak supervision*, 2022. arXiv: 2212.04356 [eess.AS].

[VSP+17] A. Vaswani *et al.*, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.

[WL23] B. T. Willard and R. Louf, *Efficient guided generation for large language models*, 2023. arXiv: 2307.09702 [cs.CL].

Appendix A
Python code listings

A. Test-beds

The code used to run all sort methods to confirm each modified version of the code still provides the correct answer:

```
arr = [64, 34, 25, 12, 22, 11, 90]
sort_method(arr)
print(arr)
```

Listing 3. Test-bed for sorting algorithms

The code used to run all string methods to confirm each modified version of the code still provides the correct answer:

```
s1 = "ABABC"
s2 = "BABCABA"
result = string_method(s1, s2)
print(result)
```

Listing 4. Test-bed for string algorithms

B. Quick-Sort

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + \
        quick_sort(right)
```

Listing 5. Quick-Sort plain

```
def foo(bar):
    if len(bar) <= 1:
        return bar

    a = bar[len(bar) // 2]
    b = [x for x in bar if x < a]
    c = [x for x in bar if x == a]
    d = [x for x in bar if x > a]
    return foo(b) + c + foo(d)
```

Listing 6. Quick-Sort Foobar'ed

```
def foo(bar):
    if len(bar) <= 1:
        return bar

    aa = lambda a,b: a > b
    bb = lambda a,b: a < b
    cc = lambda a,b: a == b

    a = bar[len(bar) // 2]
    b = [x for x in bar if bb(x, a)]
    c = [x for x in bar if cc(x, a)]
    d = [x for x in bar if aa(x,a)]
    return foo(b) + c + foo(d)
```

Listing 7. Quick-Sort Lambda'ed

```
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foo(bar):
    if fuzz(bar) <= 1:
        return bar

    aa = lambda a,b: a>b
    bb = lambda a,b: a<b
```

```
    cc = lambda a,b: a == b

    a = bar[len(bar) // 2]
    b = [x for x in bar if bb(x, a)]
    c = [x for x in bar if cc(x, a)]
    d = [x for x in bar if aa(x,a)]
    return foo(b) + c + foo(d)
```

Listing 8. Quick-Sort len()'ed

```
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foo(bar):
    if fuzz(bar) <= 1:
        return bar

    z = ""
    for i in range(0, 10):
        z = z + str(i)
    aa = lambda a,b: a>b
    bb = lambda a,b: a<b
    cc = lambda a,b: a == b

    a = bar[len(bar) // 2]
    for i in range(0, 10):
        z = z + str(a)
    b = [x for x in bar if bb(x, a)]
    for i in range(0, 10):
        z = z + str(b)
    c = [x for x in bar if cc(x, a)]
    for i in range(0, 10):
        z = z + str(c)
    d = [x for x in bar if aa(x,a)]
    for i in range(0, 10):
            z = z + str(d)
    return foo(b) + c + foo(d)
```

Listing 9. Quick-Sort no-op loop'ed

C. Bubble-Sort

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = \
                    arr[j+1], arr[j]
```

Listing 10. Bubble-Sort as recursive implementation

```
def foo(bar):
    n = len(bar)
    for i in range(n):
        for j in range(0, n-i-1):
            if bar[j] > bar[j+1]:
                bar[j], bar[j+1] = \
                    bar[j+1], bar[j]
```

Listing 11. Bubble-Sort Foobar'ed

```
def foo(bar):
    n = len(bar)
    a = lambda a, b: a > b
    b = lambda a, b: a-b-1
    c = lambda a: a+1
    for i in range(n):
        for j in range(0, b(n,i)):
            if a(bar[j], bar[c(j)]):
                bar[j], bar[c(j)] = \
                    bar[c(j)], bar[j]
```

Listing 12. Bubble-Sort Lambda'ed

```
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foo(bar):
    n = fuzz(bar)
    a = lambda a, b: a > b
    b = lambda a, b: a-b-1
    c = lambda a: a+1
    for i in range(n):
        for j in range(0, b(n,i)):
            if a(bar[j], bar[c(j)]):
                bar[j], bar[c(j)] = \
                    bar[c(j)], bar[j]
```

Listing 13. Bubble-Sort len()'ed

```
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foo(bar):
    n = fuzz(bar)

    z = ""
    for i in range(0, 10):
        z = z + str(i)
    a = lambda a, b: a > b
    b = lambda a, b: a-b-1
    c = lambda a: a+1
    for i in range(n):
        for i in range(0, 10):
            z = z + str(i)
```

```
        for j in range(0, b(n,i)):
            for i in range(0, 10):
                    z = z + str(j)
            if a(bar[j], bar[c(j)]):
                bar[j], bar[c(j)] = \
                    bar[c(j)], bar[j]
```

Listing 14. Bubble-Sort no-op loop'ed

D. Reverse string

```
def reverse_string(input_string):
    reversed_string = ""
    for char in input_string:
        reversed_string = char + \
            reversed_string
    return reversed_string
```

Listing 15. Reverse string implementation

```
def foo(bar):
    a = ""
    for c in bar:
        a = c + a
    return a
```

Listing 16. Reverse string Foobar'ed

```
def foo(bar):
    a = ""
    l = []
    for i in range(len(bar)):
        c = bar[i]
        l.reverse()
        l.append(c)
        l.reverse()
        a = "".join(l)
    return a
```

Listing 17. Reverse string Lambda'ed

```
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foo(bar):
    a = ""
    l = []
    for i in range(fuzz(bar)):
        c = bar[i]
        l.reverse()
        l.append(c)
        l.reverse()
        a = "".join(l)
    return a
```

Listing 18. Reverse string len()'ed

```
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foo(bar):
    a = ""
    l = []
    z = ""
    for ii in range(0, 10):
        z = z + str(ii)
    for i in range(fuzz(bar)):
        for ii in range(0, 10):
            z = z + str(ii)
        c = bar[i]
        l.reverse()
        l.append(c)
        l.reverse()
        a = "".join(l)
    return a
```

Listing 19. Reverse string no-op loop'ed

E. Longest common sub-string

```
def longest_common_substring(s1, s2):
    m = len(s1)
    n = len(s2)

    dp = [[0] * (n + 1) for _ in range(m + 1)]

    max_length = 0
    ending_pos = 0

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = \
                    dp[i - 1][j - 1] + 1
                if dp[i][j] > max_length:
                    max_length = dp[i][j]
                    ending_pos = i

    longest_substring = s1[ending_pos - \
        max_length:ending_pos]
    return longest_substring
```

Listing 20. LCS implementation

```
def foobar(foo, bar):
    m = len(foo)
    n = len(bar)
    x = [[0] * (n + 1) for _ in \
        range(m + 1)]
    a = 0
```

```python
    b = 0

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if foo[i - 1] == bar[j - 1]:
                x[i][j] = \
                    x[i - 1][j - 1] + 1
                if x[i][j] > a:
                    a = x[i][j]
                    b = i
    return foo[b - a:b]
```

Listing 21. LCS Foobar'ed

```python
def foobar(foo, bar):
    a = 0
    b = 0
    m = len(foo)
    n = len(bar)
    eq = lambda a,b: a == b
    lg = lambda a,b: a>b
    x = [[0] * (n + 1) for _ in \
        range(m + 1)]


    m1 = m+1
    n1 = n+1
    for i in range(1, m1):
        for j in range(1, n1):
            o = j-1
            q = i-1
            if eq(foo[q], bar[o]):
                x[i][j] = x[q][o] + 1
                if lg(x[i][j], a):
                    a = x[i][j]
                    b = i
    return foo[b - a:b]
```

Listing 22. LCS Lambda'ed

```python
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foobar(foo, bar):
    a = 0
    b = 0
    m = fuzz(foo)
    n = fuzz(bar)
    eq = lambda a,b: a == b
    lg = lambda a,b: a>b
    inc = lambda a: a+1
    dec = lambda a: a-1
    x = [[0] * inc(n) for _ in \
        range(inc(m))]
```

```python
    m1 = m+1
    n1 = n+1
    for i in range(1, m1):
        for j in range(1, n1):
            o = dec(j)
            q = dec(i)
            if eq(foo[q], bar[o]):
                x[i][j] = inc(x[q][o])
                if lg(x[i][j], a):
                    a = x[i][j]
                    b = i
    return foo[b - a:b]
```

Listing 23. LCS len()'ed

```python
def fuzz(foo):
    bar = 0
    for _ in foo:
        bar += 1
    return bar

def foobar(foo, bar):
    a = 0
    b = 0
    z = ""
    for ii in range(0, 10):
        z = z + str(i)
    m = fuzz(foo)
    n = fuzz(bar)
    eq = lambda a,b: a == b
    lg = lambda a,b: a>b
    x = [[0] * (n + 1) for _ in \
        range(m + 1)]

    for ii in range(0, 10):
        z = z + str(i)

    m1 = m+1
    n1 = n+1
    for i in range(1, m1):
        for ii in range(0, 10):
            z = z + str(ii)
        for j in range(1, n1):
            for ii in range(0, 10):
                z = z + str(ii)
            o = j-1
            q = i-1
            if eq(foo[q], bar[o]):
                for ii in range(0, 10):
                    z = z + str(ii)
                x[i][j] = x[q][o] + 1
                if lg(x[i][j], a):
                    for ii in range(0, 10):
                        z = z + str(ii)
```

```
                a = x[i][j]
                b = i

    for ii in range(0, 10):
        z = z + str(i)
    return foo[b - a:b]
```

Listing 24. LCS no-op'ed

Appendix B
ChatGPT answers

A. Quick-Sort

1) Plain: The response to the prompt

what does this code do? ``` <actual code> 5 ```

was:

The provided code is an implementation of the Quick Sort algorithm, which is a widely used sorting algorithm known for its efficiency. It works by selecting a pivot element from the array and then partitioning the array into three parts: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. This partitioning process is repeated recursively on the left and right sub-arrays until the entire array is sorted.
Here's a breakdown of how the code works:
The quick_sort function takes an array arr as input.
The base case is defined where if the length of the array is 1 or less, it is already considered sorted, so the function simply returns the array.
If the array has more than one element, the pivot is chosen as the element at the middle index (approximately). This pivot selection is a common practice in Quick Sort.
The array is then partitioned into three sub-arrays:
left: contains elements that are smaller than the pivot.
middle: contains elements that are equal to the pivot.
right: contains elements that are greater than the pivot.
The quick_sort function is recursively called on both the left and right sub-arrays. The results of these recursive calls will be sorted versions of the left and right sub-arrays.
Finally, the function returns the concatenation of the sorted left sub-array, the middle sub-array, and the sorted right sub-array. This forms the sorted version of the original array.
It's important to note that while Quick Sort is an efficient algorithm, the provided code might not be the most optimal implementation in terms of performance for larger arrays. The repeated list comprehensions to create the left, middle, and right sub-arrays can lead to additional memory usage and potentially slower execution. More efficient implementations of Quick Sort often involve manipulating indices directly within the array without creating separate sub-arrays.

2) Foobar'ed: The response to the prompt

what does this code do? ``` <actual code> 6 ```

was:

This code defines a recursive function named floo that takes a single argument bar, which is assumed to be a list of comparable elements (e.g., numbers, strings, etc.). The purpose of this function is to perform a sorting algorithm using a divide-and-conquer approach known as QuickSort.
Here's how the code works:
The base case of the recursion is when the length of the input list bar is 1 or less. In this case, the function simply returns the input list as it is already sorted (or doesn't need sorting).
If the length of bar is greater than 1, the algorithm does the following:
It selects a pivot element a from the middle of the list bar. It then divides the list into three sublists: elements less than a, elements equal to a, and elements greater than a, represented by the lists b, c, and d, respectively. The algorithm then recursively calls itself on the lists b and d, which contain the elements less than and greater than the pivot respectively. The result of these recursive calls will be the sorted versions of these sublists.
The result of the sorting process is obtained by concatenating the sorted sublist b, the equal elements c, and the sorted sublist d.
By repeatedly dividing the input list into smaller sublists based on the pivot element and then sorting those sublists recursively, the QuickSort algorithm efficiently sorts the entire list.
It's important to note that the efficiency of QuickSort largely depends on the choice of pivot. In this implementation, the middle element is chosen as the pivot, which could lead to suboptimal performance in certain cases, especially when the input list is already partially sorted. More advanced implementations of QuickSort might use techniques like random pivots or the "median of three" pivot selection to improve performance in various scenarios.

3) Lambda'ed: The response to the prompt

what does this code do? ``` <actual code> 7 ```

was:

The given code defines a recursive function foo(bar) that takes a list bar as input and performs the following steps:
The function checks if the length of the input list bar is less than or equal to 1. If so, it immediately returns the input list as there's no need for further processing.
Inside the function, three lambda functions aa, bb, and cc are defined. These lambda functions take two arguments each and are used for comparison purposes.
aa returns True if the first argument is greater than

the second, bb returns True if the first argument is less than the second, and cc returns True if the two arguments are equal.
The code then calculates the middle element of the input list bar using integer division (//). This element is assigned to the variable a.
The code creates three new lists:
b contains elements from the input list bar that are less than a (according to the comparison using bb lambda function). c contains elements from the input list bar that are equal to a (according to the comparison using cc lambda function). d contains elements from the input list bar that are greater than a (according to the comparison using aa lambda function). The function then applies itself recursively to the lists b and d and concatenates the results of these recursive calls with the c list in between. This creates a new list that has been divided and sorted around the pivot element a.
The final sorted and divided list is returned.
In essence, the code is implementing a recursive sorting algorithm where the pivot element is chosen as the middle element of the input list, and the list is divided into three parts: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. This process is repeated recursively on the "less than" and "greater than" partitions until the base case of having only one element (or none) is reached.

4) len()'ed: The response to the prompt

what does this code do? ``` <actual code> 8 ```

was:

This code defines two functions, fuzz(foo) and foo(bar), which appear to manipulate and process a list-like data structure called bar. Let's break down what each part of the code does:
fuzz(foo): This function takes a parameter foo, which is assumed to be an iterable (e.g., a list or string). It iterates through each element of foo and counts the number of iterations, effectively counting the number of elements in the iterable. The final count is returned as the output.
foo(bar): This function takes a parameter bar, which is assumed to be a list-like data structure. The function's purpose seems a bit more complex:
a. It first checks if the result of fuzz(bar) is less than or equal to 1. If so, it simply returns the bar without any modifications. This means that if the length of bar is 1 or less, the function returns the input unchanged.
b. If the length of bar is greater than 1, the function does the following:
It defines three lambda functions aa, bb, and cc that compare two arguments. It calculates the median index of bar using integer division: a = bar[len(bar) // 2]. This is used as a pivot for further processing. It creates three lists: b, c, and d, by filtering elements from bar based on comparisons with the pivot a. b: Contains elements that are less than a. c: Contains elements that are equal to a. d: Contains elements that are greater than a. The function then performs recursive calls to foo on the lists b and d, and concatenates the results with c in between. Essentially, the function recursively divides the input list into three segments: those smaller than the pivot, those equal to the pivot, and those greater than the pivot. It applies the same process to the segments smaller and greater than the pivot, creating a recursive tree structure. Finally, it combines the results of the recursive calls along with the elements equal to the pivot to form the final output.
The behavior and output of the foo function can be complex and may require careful testing and analysis with actual input data to fully understand its functionality and potential results.

5) no-op loop'ed: The response to the prompt

what does this code do? ``` <actual code> 9 ```

was:

This code defines two functions, fuzz(foo) and foo(bar), and it appears to be a somewhat convoluted piece of code that involves manipulating lists and strings in a recursive manner. Let's break down what each function does:
fuzz(foo): This function takes a single argument foo, which is presumably a sequence (e.g., a list, string, etc.). It initializes a variable bar to 0 and then iterates through each element in the foo sequence using a loop. In each iteration, it increments the bar counter by 1. Finally, the function returns the value of bar, which is essentially the length of the sequence foo.
foo(bar): This function takes a sequence bar as an argument. It first checks if the length of the sequence, determined by calling fuzz(bar), is less than or equal to 1. If this condition is met, it returns the sequence bar itself.
If the length of bar is greater than 1, the function proceeds with some complex manipulations:
It initializes a string variable z as an empty string. It defines three lambda functions aa, bb, and cc, each of which takes two arguments and returns a boolean value based on comparisons between the arguments. It calculates the middle element of the sequence bar using bar[len(bar) // 2] and assigns it to the variable a. It appends the string representation of a to the string z 10 times. It creates a list b containing elements from the sequence bar that are smaller than a. It appends the string representation of the list b to the string z 10 times. It creates a list c containing elements from the sequence bar that are equal to a. It appends the string representation of the list c to the string z 10

times. It creates a list d containing elements from the sequence bar that are greater than a. It appends the string representation of the list d to the string z 10 times. It recursively calls foo(b) and concatenates the result with the list c and the result of foo(d). In essence, this code seems to be constructing a string z by appending various representations of lists and elements based on their relationships with the middle element a of the input sequence bar. The behavior is complex and somewhat convoluted, and it's not immediately clear what practical purpose this code serves without more context. It might be helpful to know the context or intention behind this code in order to provide a more meaningful explanation.

B. Bubble-Sort

1) Plain: The response to the prompt

what does this code do? ``` <actual code> 10 ```

was:

The provided code is an implementation of the Bubble Sort algorithm, a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated for each element in the list until the entire list is sorted.
Here's how the code works:
The function bubble_sort takes a single argument arr, which is the list of elements to be sorted.
The variable n is assigned the length of the input array arr.
The outer loop runs n times, where n is the length of the input array. This loop controls the number of passes needed to sort the array.
The inner loop runs for each element in the array, except for those that have already been sorted in previous passes. This is achieved by using the range range(0, n-i-1), where i is the current pass number.
Inside the inner loop, the code compares each element at index j with the element at the next index j+1. If arr[j] is greater than arr[j+1], it means that the elements are out of order and need to be swapped.
The swapping of elements is done using a common technique in Python: arr[j], arr[j+1] = arr[j+1], arr[j]. This line of code swaps the values of arr[j] and arr[j+1].
After completing the inner loop for a single pass, the largest unsorted element in the array will "bubble up" to the last position of the array.
The outer loop continues this process for n passes until the entire array is sorted in ascending order.
After the bubble_sort function is executed with a given input array, the input array will be modified in place, with its elements arranged in ascending order.
While Bubble Sort is a simple sorting algorithm, it's not very efficient for large lists due to its average and worst-case time complexity of O(n2̂). There are more efficient sorting algorithms available, such as QuickSort, MergeSort, and HeapSort, which have better performance characteristics.

2) Foobar'ed: The response to the prompt

what does this code do? ``` <actual code> 11 ```

was:

The given code defines a function foo that implements the bubble sort algorithm. Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, indicating that the list is sorted.
Here's a breakdown of how the code works:
The foo function takes one argument, bar, which is expected to be a list of elements to be sorted.
n = len(bar) calculates the length of the input list bar.
The nested for loops are used to iterate through the list and perform the sorting:
The outer loop iterates through each element in the list using the index i. The inner loop iterates through the range 0 to n - i - 1. The range is limited to this size because, after each iteration of the outer loop, the largest unsorted element "bubbles up" to the end of the list, so there's no need to compare it again. Inside the inner loop, the code compares adjacent elements bar[j] and bar[j+1]. If bar[j] is greater than bar[j+1], it means that they are in the wrong order, so a swap is performed using the tuple unpacking technique: bar[j], bar[j+1] = bar[j+1], bar[j]. This swaps the values of bar[j] and bar[j+1] in-place. The process continues until the entire list is iterated over, and no more swaps are needed. At this point, the list is sorted in ascending order.
In summary, the foo function sorts the input list bar in ascending order using the bubble sort algorithm. However, it's worth noting that bubble sort is not the most efficient sorting algorithm, especially for large lists, as its average and worst-case time complexity is O(n2̂), where n is the number of elements in the list.

3) Lambda'ed: The response to the prompt

what does this code do? ``` <actual code> 12 ```

was:

The given code defines a function foo(bar) that takes a single argument bar, which is presumably a list or some iterable. Let's break down the code step by step to understand what it does:
n = len(bar): This line calculates the length of the input list bar and assigns it to the variable n.
a = lambda a, b: a > b: This line defines a lambda function a that takes two arguments a and b, and it returns True if a is greater than b, otherwise False.

b = lambda a, b: a-b-1: This line defines another lambda function b that takes two arguments a and b, and it returns the result of subtracting b from a and then subtracting 1 from the result.
c = lambda a: a+1: This line defines a lambda function c that takes a single argument a, and it returns the result of adding 1 to a.
The following nested loops iterate over the range of indices from 0 to n - 1 using i as the outer loop variable and the range of indices from 0 to b(n, i) - 1 using j as the inner loop variable.
Inside the nested loops, the code checks if the condition if a(bar[j], bar[c(j)]): is true. This condition compares the elements at indices j and c(j) in the list bar using the lambda function a.
If the condition is true, it means that bar[j] is greater than bar[c(j)] according to the lambda function a, so a swapping operation is performed between the elements at indices j and c(j) in the list bar.
In summary, the code appears to be a sorting algorithm implementation that uses three lambda functions (a, b, and c) to define certain conditions and arithmetic operations. The sorting logic within the loops is a bit unconventional and not immediately recognizable as a standard sorting algorithm like bubble sort, insertion sort, etc. It seems to involve swapping elements based on the conditions defined by the lambda functions. The exact behavior and efficiency of this sorting algorithm would require further analysis and testing.

4) len()'ed: The response to the prompt

what does this code do? ``` <actual code> 13 ```

was:

This code defines two functions, fuzz(foo) and foo(bar), which perform certain operations on input data. Let's break down what each function does:
def fuzz(foo): This function takes a single argument foo, which is expected to be an iterable (like a list or tuple). The function initializes a variable bar to 0. It then iterates over each element in foo, incrementing the value of bar for each element encountered. Finally, it returns the value of bar, which will be the number of elements in the iterable foo.
def foo(bar): This function takes a single argument bar, which is again expected to be an iterable. It then performs the following steps:
a. It calls the fuzz() function on the input bar and stores the result in n. b. It defines three lambda functions:
a takes two arguments a and b and returns True if a is greater than b, otherwise False. b takes two arguments a and b and returns the result of a - b - 1. c takes one argument a and returns the result of a + 1. c. It enters a nested loop using the range of n (the value returned by fuzz(bar)): The outer loop iterates from 0 to n - 1 (values of i). The inner loop iterates from 0 to the value returned by b(n, i). This inner loop is executed with different ranges depending on the value of n and i. Inside the inner loop, there is a conditional statement: It checks if a(bar[j], bar[c(j)]) is True, where j is the loop variable of the inner loop. If the condition is satisfied, it swaps the elements at indices j and c(j) in the bar list. In summary, the foo(bar) function performs a series of operations involving nested loops and conditional statements. It swaps elements in the input bar list based on the conditions specified by the lambda functions a, b, and c, as well as the logic within the loops. The code might be part of a larger program and could have a specific purpose depending on the context in which it's used.

5) no-op loop'ed: The response to the prompt

what does this code do? ``` <actual code> 14 ```

was:

This code defines two functions, fuzz and foo, which seem to operate on lists or sequences.
The fuzz function: This function takes a single argument foo, which is expected to be an iterable (like a list or tuple). It initializes a variable bar with a value of 0. It then iterates over the elements in the input foo using a for loop, incrementing the bar value by 1 for each element encountered. Finally, it returns the value of bar.
The foo function: This function takes an argument bar, which seems to be a list (or sequence) of values. Inside the function, it calculates the value of n by calling the fuzz function with the bar as an argument. Then, the function initializes an empty string z.
The next part of the function involves nested loops: The outer loop runs n times. The first inner loop runs from 0 to 9 and appends the string representation of the loop variable to z. The second inner loop uses a lambda function b to calculate a value, which is then used to run the next loop. The third inner loop runs from 0 to 9 and appends the string representation of j to z. Inside this loop, there's a conditional statement that checks whether a(bar[j], bar[c(j)]) is True, where a is another lambda function, and c is yet another lambda function. If the conditional statement is true, it swaps the values of bar[j] and bar[c(j)].
In summary, the code is quite complex and appears to be manipulating lists or sequences (bar) based on various nested loops and lambda functions. The exact purpose and intended behavior of the code depend on the specific inputs provided to the foo function and the values within the bar list. The lambda functions and nested loops seem to be performing a combination of string concatenation and value manipulation operations.

C. Reverse string

1) Plain: The response to the prompt

what does this code do? ``` <actual code> 15 ```

was:

This code defines a function named reverse_string that takes an input_string as an argument and returns the reversed version of that string.
Here's how the code works:
It initializes an empty string called reversed_string, which will be used to build the reversed version of the input string.
It iterates over each character char in the input_string using a for loop.
Inside the loop, it concatenates the current character char with the existing reversed_string, effectively placing the current character at the beginning of the reversed_string.
After looping through all the characters in the input_string, the loop concludes, and the reversed_string containing the characters in reverse order is returned as the result.
To illustrate with an example:
If you call the function like this: reverse_string("hello"), the output will be "olleh", because the code iterates through each character of the input string "hello" and builds the reversed string "olleh".

2) Foobar'ed: The response to the prompt

what does this code do? ``` <actual code> 16 ```

was:

The code defines a Python function named foo that takes a single argument bar. The purpose of the function is to reverse the characters in the input string bar using a loop.
Here's a step-by-step explanation of what the code does:
It initializes an empty string a. It starts iterating through each character c in the input string bar from the beginning. Inside the loop, it concatenates the current character c with the existing content of the string a. This effectively adds the current character to the beginning of the string, effectively reversing the order of characters. After looping through all characters in bar, the reversed string a is returned from the function. In essence, the function takes a string as input and returns a new string with the characters in reverse order. For example, if you call foo("hello"), it will return "olleh".

3) Lambda'ed: The response to the prompt

what does this code do? ``` <actual code> 17 ```

was:

This code defines a function foo that takes a single argument bar, which is expected to be a string. Let's break down what the code does step by step:
a = "": Initializes an empty string a that will be used to store the result.
l = []: Initializes an empty list l that will be used to temporarily store characters from the string bar.
The for loop iterates through each character in the input string bar.
a. c = bar[i]: Retrieves the character at the current index i from the input string bar. b. l.reverse(): Reverses the order of elements in the list l. c. l.append(c): Appends the current character c to the end of the list l. d. l.reverse(): Reverses the order of elements in the list l again. e. a = "".join(l): Joins the elements in the list l into a single string and assigns it to the variable a.
In essence, the code is reversing the order of elements in the list l, appending the current character c to the list, and then reversing the order of elements again. This effectively adds the current character to the beginning of the list l instead of the end.
The line a = "".join(l) creates a string by joining the elements in the list l in their current order.
The loop continues to iterate through the characters in the input string, performing the same steps for each character.
After the loop finishes, the function returns the final value of the string a, which is the result of joining all the characters in the list l.
In summary, this code attempts to reverse the order of characters in the input string bar using a somewhat convoluted process involving a temporary list. However, the code has a bug: the list l is not being cleared after each iteration, so it accumulates characters from previous iterations. To fix this, you should move the initialization of the list l inside the loop to clear it before processing each character.

4) len()'ed: The response to the prompt

what does this code do? ``` <actual code> 18 ```

was:

The provided code defines two functions: fuzz(foo) and foo(bar). Let's break down what each function does:
fuzz(foo): This function takes a single argument foo, which is expected to be an iterable (like a string or a list). Inside the function, a variable bar is initialized to 0. Then, a loop iterates over each element in the input foo, and for each iteration, the value of bar is incremented by 1. Finally, the function returns the value of bar, which would be the count of elements in the input foo.
foo(bar): This function takes a single argument bar, which is expected to be an iterable (like a string or a list). Inside the function, an empty string a and an empty list l are initialized. Then, a loop runs for each value of i in the range of the result of the fuzz(bar) function (i.e., the count of elements in the input bar).

Inside the loop:

The character at index i of the input bar is extracted and stored in the variable c. The list l is reversed using the reverse() method. The character c is appended to the reversed list l. The list l is reversed again to bring it back to the original order. The list l is joined into a string using the join() method and stored in the variable a. After the loop completes, the function returns the string a, which would be the result of concatenating characters from the input bar in a specific order.

However, there's a small issue in the code: the fuzz(bar) function is used within the range of the loop in the foo(bar) function, but the fuzz() function takes an iterable as an argument, not an integer bar. This would result in an error when trying to access elements of bar as if it were an iterable.

In summary, the code seems to have some logical errors and might not behave as intended due to the issue mentioned above. If you provide more context about what the code is supposed to achieve, I can help you correct and refine the implementation.

5) no-op loop'ed: The response to the prompt

what does this code do? ``` <actual code> 19 ```

was:

The provided code defines two functions: fuzz(foo) and foo(bar). Let's break down what each function does:

fuzz(foo): This function takes a single argument foo, which is presumably an iterable (like a string or a list). It initializes a variable bar to 0. It then iterates through each element in the foo iterable using a loop. In each iteration, it increments the bar variable by 1. Finally, it returns the value of bar, which essentially represents the number of elements in the foo iterable.

foo(bar): This function takes an iterable bar as an argument. It initializes an empty string a, an empty list l, and a string z containing the numbers from 0 to 9. The loop that populates z essentially creates a string "0123456789".

The function then proceeds with a loop that iterates fuzz(bar) times. In each iteration:

It goes through each digit from 0 to 9 and appends them to the string z. It retrieves the i-th element from the iterable bar. It reverses the contents of the list l. It appends the retrieved element c to the reversed list l. It reverses the list l again. It joins the elements in the list l into a single string and assigns it to the variable a. After all iterations are complete, the function returns the string a.

In summary, the foo(bar) function seems to process each element in the iterable bar by manipulating the contents of a list l and a string z. The final output is a string that reflects the changes made during the loop iterations. The purpose of this code is a bit unclear without more context, as it appears to be manipulating data in a somewhat convoluted manner.

D. Longest common sub-string

1) Plain: The response to the prompt

what does this code do? ``` <actual code> 20 ```

was:

The provided code defines a function called longest_common_substring that calculates the longest common substring between two input strings, s1 and s2. A common substring is a sequence of characters that appears in both strings in the same order. The function utilizes dynamic programming to find the longest common substring and returns it.

Here's a step-by-step explanation of how the code works:

The function longest_common_substring takes two input strings, s1 and s2.

It initializes two variables m and n to store the lengths of s1 and s2, respectively.

It creates a 2D table dp with dimensions (m+1) x (n+1) to store the lengths of common substrings. dp[i][j] will store the length of the longest common substring of s1[:i] and s2[:j].

Two additional variables, max_length and ending_pos, are initialized to keep track of the length of the longest common substring and the ending position of that substring in s1.

The code enters a nested loop, where i iterates from 1 to m, and j iterates from 1 to n.

For each pair of characters s1[i-1] and s2[j-1], if they are equal, it means there is a continuation of a common substring. So, dp[i][j] is set to dp[i-1][j-1] + 1. This means that the length of the current common substring is one character longer than the previous common substring.

If the value of dp[i][j] (current common substring length) is greater than the current value of max_length, max_length is updated, and ending_pos is set to the current value of i. This way, we keep track of the ending position of the longest common substring in s1.

Once the nested loop completes, the longest common substring's length and ending position have been found.

Finally, the function retrieves the longest common substring by slicing s1 using the ending_pos and max_length values and returns it.

In summary, this code uses dynamic programming to efficiently find the longest common substring between two input strings s1 and s2. The function returns this longest common substring.

2) Foobar'ed: The response to the prompt

what does this code do? ``` <actual code> 21 ```

was:

The given code defines a function called foobar that takes two input strings, foo and bar. This function appears to be implementing a common algorithm known as the Longest Common Substring (LCS) algorithm, which is used to find the longest contiguous subsequence that is common to both input strings.

Here's a breakdown of how the code works:

It starts by getting the lengths of the input strings foo and bar and storing them in variables m and n respectively.

It initializes a 2D list x with dimensions (m + 1) rows and (n + 1) columns, filled with zeros. This list will be used to store the lengths of common substrings for different prefixes of the input strings.

It initializes two variables, a and b, both set to 0. These will be used to keep track of the length of the longest common substring found so far (a) and the ending index of that substring in the foo string (b).

The code then uses two nested loops to iterate over each character in the foo and bar strings, comparing them to find common substrings. Specifically, it starts comparing characters from the first character of both strings.

If the current characters being compared from foo and bar are the same, the value at x[i][j] (representing the length of the common substring ending at position i in foo and position j in bar) is set to the value at the diagonal element x[i - 1][j - 1] plus 1.

If the current common substring length (x[i][j]) is greater than the previously recorded maximum length (a), the variables a and b are updated to store the new maximum length and the ending index of that common substring in foo.

After both loops complete, the function returns the longest common substring found in the foo string, starting from the index b - a and ending at index b - 1. This substring is extracted using slicing notation.

In summary, the foobar function implements the Longest Common Substring algorithm and returns the longest common substring found in the foo string that matches a substring in the bar string.

3) Lambda'ed: The response to the prompt

what does this code do? ``` <actual code> 22 ```

was:

The given code defines a function named foobar that takes two input strings foo and bar. This function appears to be aimed at finding the longest common substring between the two input strings. A common substring is a sequence of characters that appears in both strings in the same order.

Here's a breakdown of the code:

The function initializes variables a and b to 0. These variables will store the length of the longest common substring and the ending index of the substring in foo respectively.

The lengths of the input strings foo and bar are calculated and stored in variables m and n.

Two lambda functions, eq and lg, are defined. eq checks whether two values are equal, and lg checks whether the first value is greater than the second.

A 2D list x of dimensions (m + 1) x (n + 1) is created and initialized with 0s. This list will be used to store the length of the common substring ending at position (i, j).

Two variables m1 and n1 are assigned the values m + 1 and n + 1 respectively.

The code then enters a nested loop that iterates through each position (i, j) within the dimensions of the 2D list x.

For each position (i, j), the indices o and q are calculated by subtracting 1 from j and i respectively. These indices represent positions in the strings bar and foo.

The code then checks if the characters at positions q and o in foo and bar are equal using the eq lambda function.

If the characters are equal, the value at position (i, j) in the 2D list x is updated to be one greater than the value at position (q, o). This represents extending the length of the common substring by one character.

If the length of the common substring at (i, j) is greater than the current maximum length a, a is updated to this new value, and b is updated to i.

Finally, the function returns the substring of foo starting at index b - a and ending at index b, which represents the longest common substring between foo and bar.

In summary, this code defines a function to find the longest common substring between two input strings foo and bar using dynamic programming techniques. The function returns the longest common substring found.

4) len()'ed: The response to the prompt

what does this code do? ``` <actual code> 23 ```

was:

This code defines two functions, fuzz and foobar, which seem to be related to finding the longest common substring between two sequences of elements.

The fuzz function: This function takes a single argument foo, which is presumably a sequence (list, string, etc.). It initializes a variable bar to 0 and then iterates through each element in the sequence, incrementing bar for each iteration. Finally, it returns the value of bar. Essentially, it's counting the number of elements in the foo sequence.

The foobar function: This function takes two sequences as arguments, foo and bar. It also uses the fuzz function to calculate the lengths (m and n, respectively) of the two sequences.

It then defines two lambda functions, eq and lg, which are used for comparison operations (eq checks if two elements are equal, and lg checks if the first element is greater than the second).

The function initializes a 2D list x with dimensions (m + 1) x (n + 1) filled with zeros.

The main part of the function consists of a nested loop that iterates over the indices of the 2D list x (excluding the first row and column). For each index (i, j), it checks if the elements at the corresponding positions in foo and bar are equal using the eq lambda function. If they are equal, it updates the value at x[i][j] to be one more than the value at x[q][o] (where q = i - 1 and o = j - 1). This step effectively keeps track of the length of the longest common substring ending at position (i, j).

Additionally, if the current value in x[i][j] is greater than the current maximum length a, it updates a with the new maximum value and updates b with the row index i.

Finally, the function returns a subsequence of foo starting from index b - a and ending at index b, effectively returning the longest common substring between the two sequences.

In summary, the foobar function calculates the longest common substring between two sequences foo and bar using a dynamic programming approach. The code is finding the length of the longest common substring and then extracting that substring from foo.

5) no-op loop'ed: The response to the prompt

what does this code do? ``` <actual code> 24 ```

was:

This code defines two functions fuzz and foobar, and the main purpose of the foobar function is to perform some sort of comparison between two input sequences (foo and bar) and return a subsequence of foo based on certain conditions.

Let's break down the code step by step:

The fuzz function takes a sequence foo as input and counts the number of elements in the sequence by iterating over it using a loop. The count is stored in the variable bar, which is then returned.

The foobar function takes two sequences, foo and bar, as inputs. It initializes some variables a, b, and z. The z variable seems to be used as a string for appending characters, but its purpose is unclear as it is manipulated in multiple places without being used for anything meaningful.

The function then enters a loop that iterates from 0 to 9 (inclusive) using the variable ii. In each iteration of this loop, the character representation of ii is appended to the string z. This loop is repeated twice within the foobar function.

The m variable is assigned the result of calling the fuzz function with foo as an argument, which computes the count of elements in foo. Similarly, the n variable is assigned the count of elements in bar using the fuzz function.

Two lambda functions, eq and lg, are defined. These functions are meant to perform comparisons later in the code. eq checks for equality between two values, and lg checks if the first value is greater than the second.

A 2D list x is created with dimensions (m + 1) by (n + 1).

The code then enters a loop that iterates from 0 to 9 (inclusive) using the variable ii. In each iteration of this loop, the character representation of ii is appended to the string z.

The nested loops following this section iterate over the ranges (1, m1) and (1, n1), where m1 is m + 1 and n1 is n + 1. These loops iterate through the 2D list x and perform calculations based on the values in foo and bar.

Inside the nested loops, the variables o and q are assigned values based on the current iteration variables. The code then checks if the elements at indices q and o in foo and bar are equal. If they are equal, certain operations are performed on the x list, and the a and b variables might be updated.

The final loop iterating from 0 to 9 (inclusive) using the variable ii appends the character representation of i to the string z.

The function returns a subsequence of foo using slicing based on the values of a and b. It returns foo[b - a:b].

Overall, the code appears to be performing some kind of comparison between the sequences foo and bar, and it calculates a subsequence of foo based on certain conditions. However, the code seems to contain several unused variables and unclear operations involving the string z, which makes its actual purpose and functionality difficult to discern without more context. Additionally, there are inconsistencies and errors in the code (e.g., referencing i before it is defined) that would need to be addressed for the code to run correctly.