

Mohamed Mohamed

CSCI 420 Assignment 3

Task 1:

1. **Code Summarization (Java)** Summarize the functionality of the following method:

```
public Map<String, Integer> countWordFrequency(List<String> words) {  
    Map<String, Integer> freqMap = new HashMap<>();  
    for (String word : words) {  
        freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);  
    }  
    return freqMap;  
}
```

Method 1 Zero Shot:

Input: Summarize the functionality of the following java method:

ChatGPT Output: This method takes a list of words and returns a map where each word is a key and the corresponding value is the number of times that word appears in the list.

ChatGPT Comments: Clearly summarizes the method's purpose but lacks detailed reasoning.

ClaudAI Output: The method counts how many times each word appears in a list and returns a map of words to their frequency counts.

ClaudAI Comments: Functionally correct and high-level. Doesn't describe the use of getOrDefault or the iteration.

Method 2 Chain-of-thought:

Input: think of a step-by-step to understand what the following Java method does. First, look at the parameters. Then, consider how the map is populated, then interpret what the method returns.

ChatGPT Output:

- The method takes a list of strings as input.
- It creates an empty map to count word frequencies.
- It loops through the list and uses getOrDefault to either retrieve the existing count or default to 0, then increments the count.
- Finally, it returns the map containing word frequencies.

Comments: Breaks down the function step-by-step with strong explanatory depth.

ClaudAI Output: The method takes a list of words and uses a HashMap to count the frequency of each word. It checks whether a word already exists in the map using getOrDefault, increments the count if it

does, and adds it with count 1 if it doesn't. It returns the completed frequency map.

ClaudAI Comments: Includes all major logic steps and interprets the use of `getOrDefault`, though not that structured in its explanation.

Analysis: While both models correctly summarized the method, ChatGPT's chain-of-thought response gave a clearer breakdown of how the function works, including numbered steps and use of `getOrDefault`. Claude's CoT was solid too, but a little less structured. Overall, ChatGPT + chain-of-thought made the logic easiest to follow.

Task 2:

2. **Bug Fixing (Python – Off-by-One)** Identify and fix the off-by-one error in this function:

```
def sum_range(start, end):  
    total = 0  
    for i in range(start, end):  
        total += i  
    return total
```

Method 1 Zero Shot:

Input: Identify and fix the off-by-one error in the following Python function:

ChatGPT Output:

```
def sum_range(start, end):  
  
    total = 0  
  
    for i in range(start, end + 1): # Include 'end' in the range  
  
        total += i  
  
    return total
```

ChatGPT Comments: Identifies and fixes the error effectively, but without deeper reasoning.

ClaudAI Output:

```
def sum_range(start, end):  
  
    total = 0  
  
    for i in range(start, end + 1): # Include 'end' in the range  
  
        total += i  
  
    return total
```

ClaudAI Comments: Effectively patches the bug with the appropriate fix. Gives the expected behavior

but offers no further elaboration.

Method 2 Chain-of-thought:

Input: Let's walk through this code step-by-step to find any off-by-one errors. First, observe what the `range(start, end)` does. Then, think about whether this includes the last number in the sum. Finally, decide what needs to be changed.

ChatGPT Output:

```
def sum_range(start, end):  
    total = 0  
  
    for i in range(start, end + 1): # Include 'end' in the range  
        total += i  
  
    return total
```

ChatGPT Comments: Provides a thorough breakdown of the range behavior and justifies the fix.

ClaudAI Output:

```
def sum_range(start, end):  
    total = 0  
  
    for i in range(start, end + 1):  
        total += i  
  
    return total
```

ClaudAI Comments: Recognizes the behavior of Python's range function and corrects it by extending the range. Mentions that `end + 1` ensures inclusion of the last value.

Analysis: Both models fixed the bug accurately in zero-shot mode. But the chain-of-thought prompts encouraged deeper reasoning. ChatGPT's step-by-step explanation of how Python's `range` works was especially helpful. Claude was fine too, but ChatGPT's clarity gave it the edge here.

Task 3

3. **Bug Classification (C++)** Classify the type of bug in the following C++ function:

```
int* getArray(int size) {  
    int arr[size]; // Warning: local array  
    return arr; // Bug: returning pointer to local variable  
}
```

Method 1 Zero Shot:

Input: Classify the type of bug in the following C++ function and explain why it's problematic:

ChatGPT Output:

The bug in the function is a use-after-return error. The array arr is a local variable, so its memory is deallocated when the function returns. Returning a pointer to this invalid memory causes undefined behavior.

Fix:

Use dynamic memory allocation with new or malloc to ensure the memory persists after the function returns.

ChatGPT Comments: Identifies the bug correctly and suggests dynamic memory allocation.

ClaudAI Output: This function returns a pointer to a local variable arr, which is allocated on the stack. Once the function exits, that memory is no longer valid, and using the returned pointer causes undefined behavior.

ClaudAI Comments: Correctly explains the scope issue with stack memory and why returning a pointer to it is unsafe.

Method 2 Chain-of-Thought:

Input: Let's think through the function step-by-step. First, look at how the array is declared. Then, consider the scope and memory duration of that array. Finally, analyze the implications of returning a pointer to it.

ChatGPT Output: Let's break it down step-by-step:

1. Array Declaration:
 - The array `int arr[size];` is declared locally inside the function. It is created on the stack and its size is determined by the size parameter.
2. Scope and Memory Duration:
 - The array `arr` exists only within the scope of the function. Once the function exits, the stack memory for `arr` is deallocated, meaning the array no longer exists.
3. Returning a Pointer:
 - The function returns a pointer to `arr`, but since `arr` is a local variable, the memory it occupies is no longer valid once the function ends. This means the returned pointer points

to invalid memory, leading to undefined behavior.

Implications: Returning a pointer to a local variable causes undefined behavior because the memory for that variable is released when the function scope ends. This can result in crashes, data corruption, or unexpected behavior.

ChatGPT Comments: Explains why the bug occurs with clear attention to memory scope.

ClaudAI Output: the array `arr` is created as a local variable, so it only exists while the function is running. Returning a pointer to it means the caller receives a pointer to memory that has been deallocated. This is unsafe and results in undefined behavior.

ClaudAI Comments: Explains the lifespan of local variables and how this results in an invalid pointer. Offers a good conceptual overview of the problem.

Analysis: Both models identified the stack memory issue, but ChatGPT stood out in chain-of-thought mode. It walked through the memory scope problem and clearly explained why dynamic allocation is needed. Claude understood the bug but didn't go as deep into the fix.

Task 4:

4. **Generating Email Validators (Python + Regex)** Complete the function using regex to validate basic email addresses:

```
def is_valid_email(email):  
    # TODO: Complete using regex  
    pass
```

Method 1 Zero Shot:

Input: Complete the following Python function using regular expressions to validate basic email addresses:

ChatGPT Output:

```
import re
```

```
def is_valid_email(email):  
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'  
    if re.match(pattern, email):  
        return True  
    return False
```

ChatGPT Comments: Correctly implements a regex-based validator and explains the pattern.

ClaudAI Output:

```
import re
```

```
def is_valid_email(email):  
    return bool(re.match(r'^[\w\.-]+@[\w\.-]+\w+$', email))
```

ClaudAI Comments: Implements a simplified but effective regex, it meets the basic goal and works for general use cases.

Method 2 Chain-of-Thought:

Input: Let's approach this step-by-step. First, consider what makes a string a valid email address. Then, think about how to express that using a regex pattern. Finally, use Python's re module to apply it in the function:

ChatGPT Output:

```
def is_valid_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))
```

ChatGPT Comments: This approach outlines the structure of a valid email and builds the regex pattern step-by-step. It helps demystify regex usage for readers unfamiliar with it.

ClaudAI Output:

```
import re

def is_valid_email(email):
    # Step 1: Convert pattern to match valid emails
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w{2,}$'
    return bool(re.match(pattern, email))
```

ClaudAI Comments: Follows a reasoning-first approach by identifying necessary parts of an email. The regex is reasonable

Analysis: Both models returned valid regex-based validators. ChatGPT's pattern was stricter and included more edge-case protection, while Claude used a simpler expression. Chain-of-thought prompts led both models to explain formatting better, with Claude providing a slightly more digestible breakdown of the steps involved so I would say Claude's output using the chain-of-thought method was the best.

Task 5:

5. **Generating Flask APIs (Python)** Create a '/greet/<username>' endpoint that returns a JSON greeting:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
    # TODO: Return a JSON greeting
    pass
```

Method 1 Zero Shot:

Input: Complete the following Flask API endpoint so that it returns a JSON response greeting the given username

ChatGPT Output:

```
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/greet/<username>')
def greet(username):
    return jsonify({"message": f"Hello, {username}!"})
```

ChatGPT Comments: The code correctly returns a greeting using Flask's jsonify, completing the endpoint as intended. It's compact and implementation-focused.

ClaudeAI Output:

```
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/greet/<username>')
def greet(username):
    return jsonify(message=f"Hello, {username}!")
```

ClaudAI Comments: Provides a clean and Pythonic solution. Uses the shorthand message= syntax for clarity but produces the same JSON structure.

Method 2 Chain-of-Thought:

Input: Let's work through this step-by-step. First, recall how Flask handles routes with parameters. Then, think about how to construct a JSON response using the jsonify function. Finally, return a greeting message with the given username.

ChatGPT Output:

```
app = Flask(__name__)
@app.route('/greet/<username>')
def greet(username):
    return jsonify({"message": f"Hello, {username}!"})
```

ChatGPT Comments: This method explains Flask routing and dynamic parameters before constructing the endpoint. It makes the logic and purpose behind each line clearer.

ClaudAI Output:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
    # Step 1: Extract the username from the URL parameter
    # Step 2: Format the message
    # Step 3: Return it as a JSON object
    return jsonify({"message": f"Hello, {username}!"})
```

ClaudAi Comments: Follows a logical breakdown and uses inline steps as comments. Offers a didactic

touch without changing core functionality.

Analysis: Functionally, both models got the endpoint right. But Claude's chain-of-thought output included inline comments that broke the code into logical parts, which made it more instructional. If I were learning Flask for the first time, Claude's approach would be easier to follow.

Task 6:

6. **SQL Schema Design (SQL)** Write the schema for a review app with users, books, and reviews:

```
-- TODO: Design schema with appropriate keys and constraints
-- Tables: users(id, name), books(id, title), reviews(id, user_id, book_id,
rating)
```

Method 1 Zero Shot:

Input: Write an SQL schema for a review app. It should include tables for users, books, and reviews. Ensure that primary and foreign key constraints are included appropriately.

ChatGPT Output:

```
-- Table for users
CREATE TABLE users (
  id INT PRIMARY KEY,          -- Primary key for the user table
  name VARCHAR(100) NOT NULL   -- User's name, cannot be null
);

-- Table for books
CREATE TABLE books (
  id INT PRIMARY KEY,          -- Primary key for the books table
  title VARCHAR(200) NOT NULL  -- Book title, cannot be null
);

-- Table for reviews
CREATE TABLE reviews (
  id INT PRIMARY KEY,          -- Primary key for the reviews table
  user_id INT,                 -- Foreign key referencing users table
  book_id INT,                 -- Foreign key referencing books table
  rating INT CHECK (rating >= 1 AND rating <= 5), -- Rating between 1 and 5
  FOREIGN KEY (user_id) REFERENCES users(id), -- Foreign key constraint for user
  FOREIGN KEY (book_id) REFERENCES books(id)  -- Foreign key constraint for book
);
```

ChatGPT Comments: The schema is correct and includes appropriate keys and constraints. It's practical and follows good SQL practices.

ClaudAI Output:

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);
```



```

CREATE TABLE books (
  id INT PRIMARY KEY,
  title VARCHAR(200) NOT NULL
);

CREATE TABLE reviews (
  id INT PRIMARY KEY,
  user_id INT NOT NULL,
  book_id INT NOT NULL,
  rating INT CHECK (rating BETWEEN 1 AND 5),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (book_id) REFERENCES books(id)
);

```

ClaudAI Comments: Includes NOT NULL constraints for data integrity and uses CHECK BETWEEN for clarity in the rating field, This emphasizes stricter validation.

Method 2 Chain-of-Thought:

Input: Let's design this step-by-step. First, define the users and books tables with primary keys. Then, define the reviews table with foreign keys linking to both users and books. Make sure to include constraints for data integrity.

ChatGPT Output:

```

-- Define the users table
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);

-- Define the books table
CREATE TABLE books (
  id INT PRIMARY KEY,
  title VARCHAR(200) NOT NULL
);

-- Define the reviews table
CREATE TABLE reviews (
  id INT PRIMARY KEY,
  user_id INT,
  book_id INT,
  rating INT CHECK (rating >= 1 AND rating <= 5),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (book_id) REFERENCES books(id)
);

```

ChatGPT Comments: The reasoning builds each table logically, explaining constraints and relationships. It clarifies design decisions and shows how they support data integrity.

ClaudAI Output:

```

-- Step 1: Create users table
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);

-- Step 2: Create books table
CREATE TABLE books (
  id INT PRIMARY KEY,
  title VARCHAR(200) NOT NULL
);

-- Step 3: Create reviews table with foreign keys and rating constraint
CREATE TABLE reviews (
  id INT PRIMARY KEY,
  user_id INT NOT NULL,
  book_id INT NOT NULL,
  rating INT CHECK (rating BETWEEN 1 AND 5),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (book_id) REFERENCES books(id)
);

```

ClaudAI Comments: Uses inline comments to guide the reader through the schema's logic. Emphasizes table relationships and includes NOT NULL for tighter validation.

Analysis: All four outputs created valid SQL schemas, but Claude's chain-of-thought version really stood out. It not only followed good SQL practices but also included better constraint handling and clearer inline comments. It felt more like something you'd see in a real-world project — structured and intentional.

Task 7:

7. **Null Dereference Detection (Java)** Identify any null dereference risk:

```

public int getLength(String s) {
    return s.length(); // What if s is null?
}

```

Method 1 Zero Shot:

Input: Identify and explain any null dereference risk in the following Java method:

ChatGPT Output:

The potential issue in the method is a null dereference risk.

Explanation:

- The method `getLength(String s)` calls `s.length()`.
- If the input `s` is null, attempting to call `s.length()` will throw a `NullPointerException` because null does not have a `length()` method.

Null Dereference Risk:

- Risk: If `s` is null, calling `s.length()` results in a null dereference, which will cause a runtime exception (`NullPointerException`).

Fix:

To avoid this issue, we can check if `s` is null before calling the `length()` method:

```
java
Copy
public int getLength(String s) {
    if (s == null) {
        return 0; // Return 0 or handle the null case appropriately
    }
    return s.length();
}
```

This fix ensures that we handle the case where `s` might be null and avoids a `NullPointerException`.

ChatGPT Comments: The response catches the null dereference issue and offers a valid conditional fix. It's concise and correct.

ClaudAI Output:

The method tries to call `length()` on the string `s`. If `s` is null, this will trigger a `NullPointerException` because null references cannot access methods. The code assumes `s` is always non-null, which is dangerous. Adding a check like `if (s == null) return 0;` before the method call can avoid this issue.

ClaudAI Comments: Thoroughly explains the problem and emphasizes the importance of null validation. Suggests the same fix with added justification.

Method 2 Chain-of-Thought:

Input: Let's analyze this method step-by-step. First, consider what happens when `s` is not null. Then, ask what would occur if `s` were null. Finally, explain whether the current implementation safely handles both cases.

ChatGPT Output:

Let's break it down step-by-step:

1. When `s` is not null:

- If s is not null, the method works as expected. It calls s.length(), which returns the length of the string.
- 2. What happens if s is null?
 - If s is null, attempting to call s.length() will result in a NullPointerException because you cannot call methods on a null object.
- 3. Does the current implementation safely handle both cases?
 - No, the current implementation does not safely handle the case where s is null. It will throw a NullPointerException if s is null, which is a runtime error that needs to be avoided.

Conclusion:

The current implementation does not safely handle both cases, as it lacks a null check. To prevent potential errors, we should add a null check before calling s.length(). For example:

```
java
Copy
public int getLength(String s) {
    if (s == null) {
        return 0; // or handle appropriately
    }
    return s.length();
}
```

ChatGPT Comments: It explains what happens when s is null versus when it's valid, then justifies the need for a null check. This improves understanding of Java's null behavior.

ClaudAI Output:

Step 1: If s is not null, the method works as intended and returns the string's length.

Step 2: If s is null, s.length() will cause a NullPointerException.

Step 3: Since there is no null check, the method is unsafe for null inputs.

Fix: Add a conditional check to avoid accessing length() on a null reference:

```
if (s == null) return 0;
```

ClaudAI Comments: Uses a clear logical flow to explain the risk and offers a standard fix. Emphasizes safety and control flow explicitly

Analysis: Both models caught the null dereference issue and provided a fix, but Claude's chain-of-thought response walked through the valid and invalid cases with really solid explanations. It made the bug easier to understand, especially if you're not yet confident with Java's null behavior.

Task 8:

8. CSV Parser Variants (Python) Improve the parser to support quoted fields:

```
def parse_csv_line(line):  
    return line.split(',') # Incomplete: doesn't handle quoted fields
```

Method 1 Zero Shot:

Input: Improve the following Python CSV parser so that it correctly handles quoted fields in the input string

ChatGPT Output:

```
import csv
```

```
def parse_csv_line(line):  
    # Use Python's built-in csv module to handle quoted fields and commas inside quotes  
    return next(csv.reader([line]))
```

ChatGPT Comments: The model directly substitutes the manual split with csv.reader, which fixes the issue. It's accurate and minimal.

ClaudAI Output:

```
import csv
```

```
def parse_csv_line(line):  
    # Use csv.reader to handle quoted fields and commas inside quotes  
    return next(csv.reader([line]))
```

ClaudAI Comments: Uses Python's csv.reader for robust handling of edge cases like commas inside quoted strings. Matches ChatGPT's approach but includes a clarifying comment.

Method 2 Chain-of-Thought:

Input: Let's improve this function step-by-step. First, think about how splitting by commas fails when fields are quoted and contain commas. Then, consider using Python's csv module to handle these cases. Finally, rewrite the function to correctly parse quoted fields.

ChatGPT Output:

```
import csv
```

```
def parse_csv_line(line):  
    # Use the csv.reader to correctly handle quoted fields and commas inside quotes  
    return next(csv.reader([line]))
```

ChatGPT Comments: This version explains how quoted fields break naive splitting and introduces the csv module as a robust solution. It shows both the problem and why the solution works.

ClaudAI Output:

```
import csv
```

```
def parse_csv_line(line):
    # Step 1: Recognize that fields can contain commas if quoted
    # Step 2: Use csv.reader to correctly parse those cases
    return next(csv.reader([line]))
```

ClaudAi Comments: Includes an explicit breakdown of why `split(',')` fails and why `csv.reader` is appropriate. Slightly more explanatory than the zero-shot version.

Analysis: The solutions were nearly identical in terms of the fix — using `csv.reader()` instead of a naive `split`. But Claude’s chain-of-thought output explained why the old version fails and walked through each part of the fix. That explanation made a big difference in terms of learning.

Task 9:

9. **Data Class to API Conversion (Kotlin)** Convert the data class to a REST API using Ktor:

```
data class Product(val id: Int, val name: String, val price: Double)

// TODO: Create GET and POST endpoints using Ktor
```

Method 1 Zero Shot:

Input: Using Ktor in Kotlin, create GET and POST REST API endpoints for the following data class:

ChatGPT Output:

1. Define the Product class:

```
kotlin
Copy
data class Product(val id: Int, val name: String, val price: Double)
```

2. Ktor Application:

```
kotlin
Copy
import io.ktor.application.*
import io.ktor.features.ContentNegotiation
import io.ktor.http.HttpStatusCode
import io.ktor.jackson.jackson
import io.ktor.request.receive
import io.ktor.response.respond
import io.ktor.routing.*
import io.ktor.server.engine.embeddedServer
import io.ktor.server.netty.Netty

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) { jackson { } }
    }
```

```

routing {
    route("/products") {
        get {
            val products = listOf(Product(1, "Laptop", 1200.0), Product(2, "Smartphone", 800.0))
            call.respond(products)
        }

        post {
            val product = call.receive<Product>()
            call.respond(HttpStatusCode.Created, "Product created with id: ${product.id}")
        }
    }
}
}.start(wait = true)
}

```

3. Dependencies:

```

kotlin
Copy
dependencies {
    implementation("io.ktor:ktor-server-netty:1.6.5")
    implementation("io.ktor:ktor-server-core:1.6.5")
    implementation("io.ktor:ktor-jackson:1.6.5")
    implementation("io.ktor:ktor-server-content-negotiation:1.6.5")
}

```

This creates a simple GET and POST API for the Product class.

Comments: The solution provides a working GET/POST API setup but uses hardcoded values. It assumes some knowledge of server state handling.

Method 2 Chain-of-Thought:

Input: Let's think through this task. First, we have a Kotlin data class named Product. To expose it via a REST API in Ktor, we need to define GET and POST routes. Use a simple in-memory list to simulate database behavior. Then, return appropriate JSON responses in each endpoint.

Output:

1. Define the Product class:

```

kotlin
Copy
data class Product(val id: Int, val name: String, val price: Double)

```

2. Ktor Application:

```
kotlin
Copy
import io.ktor.application.*
import io.ktor.features.ContentNegotiation
import io.ktor.jackson.jackson
import io.ktor.request.receive
import io.ktor.response.respond
import io.ktor.routing.*
import io.ktor.server.engine.embeddedServer
import io.ktor.server.netty.Netty

fun main() {
    val products = mutableListOf(Product(1, "Laptop", 1200.0), Product(2, "Smartphone", 800.0))

    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) { jackson { } }

        routing {
            route("/products") {
                get {
                    call.respond(products)
                }

                post {
                    val product = call.receive<Product>()
                    products.add(product)
                    call.respond("Product created with id: ${product.id}")
                }
            }
        }
    }.start(wait = true)
}
```

Explanation:

- GET /products: Returns the in-memory list of products as JSON.
- POST /products: Accepts a new Product, adds it to the list, and responds with a creation message.

Comments: The model walks through setting up in-memory storage and configuring endpoints dynamically. It gives a complete and realistic example for a REST API.

Analysis: The zero-shot versions worked, but they used hardcoded data and didn't feel very dynamic. Claude's chain-of-thought version added an in-memory list to simulate persistence and made the code more realistic. It was also the only one to explain the structure clearly. That made it the most practical.

Task 10:

10. **Function Summarization (Python)** Write a brief summary of the function:

```
def reverse_words(sentence):  
    return ' '.join(sentence.split()[::-1])
```

Method 1 Zero Shot:

Input: Summarize the functionality of the following Python function:

ChatGPT Output: The reverse_words function splits a sentence into words, reverses the order of the words, and then joins them back into a single string with spaces between them.

ChatGPT Comments: The summary accurately captures the essence of the function in a single sentence. It's simple and to the point.

ClaudAI Output: This Python function reverse_words takes a string sentence as input and reverses the order of words in the sentence while preserving the individual words themselves.

It works by:

1. Splitting the sentence into words using sentence.split()
2. Reversing the order of those words with the slice notation[::-1]
3. Joining the reversed words back together with spaces using ' '.join()

For example, if given "Hello world Python", it would return "Python world Hello".

ClaudAI Comments: The response is informative and gives a concise breakdown of how the function works, even including an example. It borders on a chain-of-thought explanation despite being zero-shot, which enhances clarity for readers unfamiliar with the syntax.

Method 2 Chain-of-Thought:

Input: Let's understand this function step-by-step. First, examine how the sentence is split. Then, notice the use of slicing to reverse the list. Finally, observe how the reversed words are joined back into a string.

ChatGPT Output:

Let's break it down step-by-step:

1. Splitting the Sentence:
 - sentence.split() splits the sentence into a list of words using spaces as the delimiter. For example, "Hello world" becomes ['Hello', 'world'].
2. Reversing the List:
 - [::-1] is a Python slicing operation that reverses the list. So ['Hello', 'world'][::-1] becomes ['world', 'Hello'].
3. Joining the Words:
 - ''.join(...) takes the reversed list and joins the words back into a single string with spaces between them. For example, ['world', 'Hello'] becomes "world Hello".

ChatGPT Comments: This version explains how each part of the function transforms the sentence. It helps users follow the logic from input to output.

ClaudAI Output:

This function `reverse_words(sentence)` reverses the order of words in a sentence while preserving the individual words themselves. Let me break it down step-by-step:

1. `sentence.split()`: This splits the input string at whitespace characters, creating a list of words.
 - For example, if `sentence = "Hello world from Python"`, this would create `["Hello", "world", "from", "Python"]`
2. `[::-1]`: This is a slicing operation that reverses the list.
 - The syntax `[start:stop:step]` is used, but with all positions omitted and a step of `-1`
 - This reverses the entire list, giving `["Python", "from", "world", "Hello"]`
3. `' '.join(...)`: This joins all elements in the list back into a string, with a space character between each element.
 - This creates the final output: `"Python from world Hello"`

So the function takes a sentence and returns a new sentence with the same words but in reverse order.

ClaudAI Comments: This version delivers a step-by-step breakdown that mirrors the function's actual behavior. It uses an illustrative example, and thoroughly explains each transformation. It's pedagogically strong and well-suited for learners.

Analysis: Claude's zero-shot response was already quite detailed, bordering on a chain-of-thought explanation with breakdowns of the logic and an example. However, the chain-of-thought prompt led to an even more structured explanation. ChatGPT performed well in both modes but kept a better focus on summarizing the function behavior rather than unpacking syntax. I would say Claude's chain-of-thought output was the most thorough and helpful for understanding both the logic and implementation.

Task 11:

11. **Prompt from Code Comments (Python)** Write a prompt that could generate the code:

```
# This function checks if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

Method 1 Zero Shot:

Input: Write a Python function that checks if a given number is a prime number.

ChatGPT Output:

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

ChatGPT Comments: The prompt is natural and likely to produce the expected prime-checking function. It's efficient and precise.

ClaudAI Output:

```
def is_prime(number):
    """
    Check if a given number is prime.
    Args:
        number: An integer to check for primality
    Returns:
        bool: True if the number is prime, False otherwise
    """
    # Handle edge cases
    if number <= 1:
        return False
    if number <= 3:
        return True
    if number % 2 == 0 or number % 3 == 0:
        return False

    # Check for divisibility using 6k±1 optimization
    i = 5
    while i * i <= number:
        if number % i == 0 or number % (i + 2) == 0:
            return False
        i += 6

    return True
```

ClaudAI Comments: Implements a more advanced and optimized version using the $6k \pm 1$ rule. Includes docstrings and handles small values efficiently.

Method 2 Chain-of-Thought:

Input: Let's think through how to write a function to check if a number is prime. First, return False for numbers less than or equal to 1. Then, check divisibility for all numbers up to the square root of n. If any number divides evenly, return False. Otherwise, return True. Now write the complete Python function.

ChatGPT Output:

```
import math

def is_prime(n):
```

```

# Return False for numbers less than or equal to 1
if n <= 1:
    return False

# Check divisibility up to the square root of n
for i in range(2, int(math.sqrt(n)) + 1):
    if n % i == 0:
        return False

# If no divisors found, return True
return True

```

ChatGPT Comments: The reasoning builds up the concept of primality before asking for code generation. This helps guide the model toward a more thoughtful solution.

ClaudAI Output:

```

def is_prime(n):
    # Handle numbers less than or equal to 1
    if n <= 1:
        return False

    # Check divisibility up to the square root of n
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False

    # If no divisors were found, the number is prime
    return True

```

ClaudAI Comments: This version is clean and well-commented, guiding the reader through the thought process. Less optimized than the zero-shot version but more beginner-friendly and aligned with the prompt's step-by-step guidance.

Analysis: All four outputs correctly implemented a prime-checking function, but they differed in complexity and pedagogical clarity. Claude's zero-shot output was the most optimized, using the $6k \pm 1$ rule and docstrings, which added depth and real-world performance benefits. The chain-of-thought versions were easier to follow though but Claude's zero-shot output was the best in terms of algorithmic strength.

Task 12:

```

def factorial(n):
    result = 1
    for i in range(1, n):
        result *= i
    return result

```

Method 1 Zero Shot:

Input: Fix the bug in the following Python function that incorrectly calculates the factorial of 0:

ChatGPT Output:

```
def factorial(n):  
    result = 1  
    for i in range(1, n + 1): # include n in the loop  
        result *= i  
    return result
```

ChatGPT Comments: Fixes the off-by-one error with a correct adjustment to the loop range. The solution is clean and accurate but lacks commentary on why the change is needed.

Claud AI Output:

```
def factorial(n):  
    result = 1  
    for i in range(1, n + 1): # Changed from range(1, n) to range(1, n + 1)  
        result *= i  
    return result
```

ClaudAI Comments: The bug is corrected cleanly by adjusting the loop to include n. The solution is functional but minimal in explanation.

Method 2 Chain-of-Thought:

Input: Let's walk through the function step-by-step. First, recognize that the factorial of 0 should return 1. Then, note that the loop range(1, n) skips the case when n = 0 or even when n = 1. Modify the loop or add a base case to fix the behavior for n = 0. Now write the corrected version of the function.

ChatGPT Output:

```
def factorial(n):  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

ChatGPT Comments: Provides the same fix as the zero-shot version but emphasizes the rationale behind adjusting the loop. The step-by-step framing improves comprehension for debugging similar issues.

ClaudAI Output:

```
def factorial(n):  
    result = 1  
    for i in range(1, n + 1): # Changed from range(1, n) to range(1, n + 1)  
        result *= i  
    return result
```

ClaudAI Comments: This version explains why the bug occurs, especially for edge cases like n = 0. It also clarifies how the fix preserves correctness across cases.

Analysis: Every version gave the same fix (adjusting the range to $n + 1$), but ChatGPT's chain-of-thought response actually explained *why* it was wrong in the first place. That reasoning made it better, especially for anyone learning how off-by-one bugs work in loops.

Task 13:

13. **Linked List Node Deletion (C)** Implement node deletion by value:

```
struct Node {
    int data;
    struct Node* next;
};

void deleteNode(struct Node** head, int key) {
    // TODO: Implement node deletion
}
```

Method 1 Zero Shot:

Input: Complete the following C function to delete a node from a singly linked list by its value:

ChatGPT Output:

```
#include <stdlib.h>
```

```
struct Node {
    int data;
    struct Node* next;
};
```

```
void deleteNode(struct Node** head, int key) {
    if (*head == NULL) return;

    struct Node* temp = *head;
    struct Node* prev = NULL;

    // If head node holds the key
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Change head
        free(temp);        // Free old head
        return;
    }

    // Search for the key
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key was not found
    if (temp == NULL) return;
```

```

// Unlink the node
prev->next = temp->next;
free(temp);
}

```

ChatGPT Comments: Handles all necessary cases: empty list, deletion at the head, and deletion in the middle or end. The logic is correct and efficient, though it assumes familiarity with pointer manipulation without explaining much.

ClaudAI Output:

```

void deleteNode(struct Node** head, int key) {
    // Handle empty list case
    if (*head == NULL) {
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Change head
        free(temp);        // Free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) {
        return;
    }

    // Unlink the node from linked list
    prev->next = temp->next;

    // Free memory
    free(temp);
}

```

ClaudAI Comments: Clearly separates each case (head node vs. internal node), improving readability for those new to linked list operations.

Method 2 Chain-of-Thought:

Input: Let's implement this function step-by-step. First, check if the list is empty. Then, handle the case where the node to be deleted is the head. Otherwise, traverse the list, keeping track of the previous node. Once the node with the target value is found, unlink it and free its memory. Now complete the C function:

ChatGPT Output:

```
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

void deleteNode(struct Node** head, int key) {
    if (*head == NULL) return; // Step 1: Check if list is empty

    struct Node* temp = *head;
    struct Node* prev = NULL;

    // Step 2: If head node holds the key
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Change head
        free(temp);        // Free old head
        return;
    }

    // Step 3: Search for the key
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // Step 4: If key was not present in list
    if (temp == NULL) return;

    // Step 5: Unlink the node from linked list
    prev->next = temp->next;
    free(temp);
}
```

ChatGPT Comments: Organized in sequential steps that mirror the CoT prompt. Maintains clarity through stepwise comments, making it ideal for learners or debugging purposes. Covers edge cases without unnecessary complexity.

ClaudAI Output:

```
struct Node {
    int data;
    struct Node* next;
};

void deleteNode(struct Node** head, int key) {
```



```

// Check if list is empty
if (*head == NULL) {
    return;
}

struct Node* temp = *head;

// Case 1: Node to be deleted is the head
if (temp->data == key) {
    *head = temp->next; // Update head
    free(temp);        // Free old head
    return;
}

// Case 2: Node to be deleted is not the head
// Search for the key while keeping track of previous node
struct Node* prev = NULL;
while (temp != NULL && temp->data != key) {
    prev = temp;
    temp = temp->next;
}

// If key was not found
if (temp == NULL) {
    return;
}

// Unlink the node from linked list
prev->next = temp->next;

// Free memory
free(temp);
}

```

ClaudAI Comments: Breaks the process into distinct cases and provides clean transitions between steps. Its structure reinforces the logic behind each operation, particularly the separation between head handling and traversal.

Analysis: The zero-shot outputs are solid. However in the chain-of-thought approach Claude provides better structure and segmentation, especially for educational purposes. Claude's chain-of-thought version is the most readable and pedagogically strong, making it the best method for this task.

Task 14:

14. **Recursive Function Completion (Python)** Complete the recursive function for Fibonacci:

```
def fibonacci(n):  
    # TODO: Base cases and recursive call  
    pass
```

Method 1 Zero Shot:

Input: Complete the following recursive function that computes the nth Fibonacci number:

ChatGPT Output:

```
def fibonacci(n):  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n - 1) + fibonacci(n - 2)
```

ChatGPT Comments: Implements the standard recursive Fibonacci definition correctly. The solution is clean but lacks comments or context for users unfamiliar with recursion.

ClaudAI Output:

```
def fibonacci(n):  
    # Base cases  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    # Recursive case  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

ClaudAI Comments: Same logic as ChatGPT, but slightly more readable due to inline comments. Still basic in structure and does not address the inefficiency of recursion.

Method 2 Chain-of-Thought:

Input: Let's complete this step-by-step. First, recall the base cases for Fibonacci: $\text{fibonacci}(0) = 0$ and $\text{fibonacci}(1) = 1$. Then, define the recursive case using $\text{fibonacci}(n-1) + \text{fibonacci}(n-2)$. Now write the complete function:

ChatGPT Output:

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:
```

```
return fibonacci(n - 1) + fibonacci(n - 2)
```

ChatGPT Comments: Follows a structured explanation through base and recursive cases. Helps clarify the recursion flow but could benefit from inline comments to support learners more explicitly.

ClaudAI Output:

```
def fibonacci(n):  
    # Base cases  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
  
    # Recursive case  
    return fibonacci(n-1) + fibonacci(n-2)
```

Comments: Clearly labels base cases and separates them from the recursive case. More instructional in style and great for building conceptual understanding of recursion.

Analysis: The zero-shot methods are brief and functional, while the chain-of-thought versions offer more structure. Claude's CoT output stands out due to its labeled formatting and logical clarity, making it ideal for educational use. It's the strongest approach for learners new to recursion.

Task 15:

15. **Constructor Completion (Python)** Complete the class constructor:

```
class Person:  
    def __init__(self):  
        # TODO: Add name, age, and optional email  
        pass
```

Method 1 Zero Shot:

Input: Complete the following Python class constructor to include name, age, and an optional email:

ChatGPT Output:

```
class Person:  
    def __init__(self, name, age, email=None):  
        self.name = name  
        self.age = age  
        self.email = email
```

ChatGPT Comments: The constructor is implemented correctly with optional parameters. It's direct and functional but lacks any descriptive cues to clarify the design decisions.

ClaudAI Output:

```
class Person:  
    def __init__(self, name, age, email=None):
```

```
self.name = name
self.age = age
self.email = email
```

ClaudAI Comments: Correct and concise, but purely utilitarian with no context provided for how or why the parameters are used.

Method 2 Chain-of-Thought:

Input: Let's complete this step-by-step. First, decide what parameters the constructor should take: name and age are required, and email is optional. Then, assign those parameters to instance variables using self. Now write the complete `__init__` method:

ChatGPT Output:

```
class Person:
    def __init__(self, name, age, email=None):
        self.name = name
        self.age = age
        self.email = email
```

ChatGPT Comments: Although functionally the same as the zero-shot version, it's framed within a step-by-step prompt. That said, the actual code lacks inline comments or additional structure to reflect that guidance.

ClaudAI Output:

```
class Person:
    def __init__(self, name, age, email=None):
        self.name = name
        self.age = age
        self.email = email
```

ClaudAI Comments: Same output as zero-shot, but benefits from the context of the step-by-step approach. Still, it doesn't reflect the CoT structure in the code, so the value is mainly in how the prompt is framed rather than the output itself.

Analysis: neither method stands out significantly, though zero-shot may be slightly more efficient since it produces the same result with less prompting. For this task, the difference between methods is negligible.

Task 16:

16. **Binary Search Completion (Java)** Complete the binary search implementation:

```
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        // TODO: Compare and adjust bounds
    }
    return -1;
}
```

Method 1 Zero Shot:

Input: Complete the following Java method for performing a binary search on a sorted array

ChatGPT Output:

```
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid; // Found the target
        } else if (arr[mid] < target) {
            left = mid + 1; // Search right half
        } else {
            right = mid - 1; // Search left half
        }
    }
    return -1; // Target not found
}
```

ChatGPT Comments: Implements the binary search correctly using the standard $(left + right) / 2$ formula. It's concise and readable but does not address potential integer overflow or performance edge cases.

ClaudAI Output:

```
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoid potential integer overflow

        if (arr[mid] == target) {
            return mid; // Found the target, return its index
        } else if (arr[mid] < target) {
            left = mid + 1; // Target is in the right half
        } else {
            right = mid - 1; // Target is in the left half
        }
    }
    return -1; // Target not found in array
}
```

ClaudAI Comments: Uses an overflow-safe formula $\text{left} + (\text{right} - \text{left}) / 2$, which is a best practice in binary search. This shows attention to implementation detail and defensive coding.

Method 2 Chain-of-Thought:

Input: Let's complete this step-by-step. First, calculate the middle index as $(\text{left} + \text{right}) / 2$. Then, check if $\text{arr}[\text{mid}]$ equals the target. If it does, return mid. If $\text{arr}[\text{mid}]$ is less than the target, move the left pointer to $\text{mid} + 1$. Otherwise, move right to $\text{mid} - 1$. Now write the full function:

ChatGPT Output:

```
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (arr[mid] == target) {
            return mid; // Target found
        } else if (arr[mid] < target) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }

    return -1; // Target not found
}
```

ChatGPT Comments: Functionally similar to the zero-shot version, but adds inline comments to explain decision points. Still doesn't apply overflow prevention, so it's pedagogically helpful but not optimal.

ClaudAI Output:

```
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;

        // Compare the middle element with the target
        if (arr[mid] == target) {
            return mid; // Target found, return its index
        } else if (arr[mid] < target) {
            left = mid + 1; // Target is in the right half
        } else {
            right = mid - 1; // Target is in the left half
        }
    }
    return -1; // Target not found
}
```

ClaudAI Comments: Maintains the same structure but improves explanation with detailed in-line commentary. However, in this version, it reverts to the standard midpoint formula and loses the overflow protection present in its zero-shot output.

Analysis: Claude's zero-shot output was the strongest technically due to its use of overflow-safe midpoint calculation, while ChatGPT focused on clarity and readability. The chain-of-thought prompts helped reinforce understanding through comments, but neither added complexity or error handling. Claude's zero-shot approach stands out here for best practice.

Task 17:

17. **Self-Consistency Bug Fixing (C++)** Resolve inconsistency between function name and logic:

```
// Supposed to return true if x is even
bool isOdd(int x) {
    return x % 2 == 0; // Logic contradicts function name
}
```

Method 1 Zero Shot:

Input: Fix the logic of the following C++ function so that it matches the function name and returns true when x is odd

ChatGPT Output:

```
bool isOdd(int x) {
    return x % 2 != 0;
}
```

ChatGPT Comments: Uses the most robust and correct condition for checking odd numbers: $x \% 2 \neq 0$. It's concise and handles both positive and negative inputs properly, though it doesn't mention why this is better than $\neq 1$.

ClaudAI Output:

```
bool isOdd(int x) {
    return x % 2 != 0;
}
```

ClaudAI Comments: Selects $\neq 0$, which correctly identifies odd values for all integers. The solution is safe but lacks commentary on edge cases.

Method 2 Chain-of-Thought:

Input: Let's fix this step-by-step. The function is named isOdd, so it should return true when x is an odd number. Recall that an odd number has a remainder of 1 when divided by 2. Therefore, modify the return condition accordingly:

ChatGPT Output:

```
bool isOdd(int x) {
    return x % 2 == 1;
}
```

ChatGPT Comments: Returns $x \% 2 == 1$, which works for positive numbers but fails for negative odd

numbers (e.g. $-3 \% 2 == -1$). This makes it less robust, though it's logically sound in a limited domain.

ClaudAI Output:

```
bool isOdd(int x) {  
    return x % 2 != 0;  
    // or alternatively: return x % 2 == 1;  
}
```

ClaudAI Comments: Explains both options: $!= 0$ and $== 1$, and notes their implications. Suggests $!= 0$ for broader correctness, showing a better grasp of integer behavior in C++.

Analysis: Claude's chain-of-thought output is the most insightful. It acknowledges subtle issues with using $== 1$, which ChatGPT overlooks. ChatGPT's CoT version is technically correct for positive numbers but incomplete overall.

Task 18:

18. **Prompt Chaining: Bug Identification → Fix (JavaScript)** Identify and fix the bug:

```
function isEven(n) {  
    return n % 2; // Returns 1 or 0, not true/false  
}
```

Method 1 Zero Shot:

Input: Identify and fix the bug in the isEven function.

ChatGPT Output:

```
function isEven(n) {  
    return n % 2 === 0;  
}
```

ChatGPT Comments: Recognizes that the original function returns 0 or 1 instead of a boolean. Provides a direct fix using $=== 0$, making the return type match the function's intended boolean behavior.

ClaudAI Output:

```
function isEven(n) {  
    return n % 2 === 0; // Returns true if even, false if odd  
}
```

ClaudAI Comments: Adds clarification in the comment that the fix returns true if even and false if odd. Demonstrates understanding of both the bug and the logical correction using a comparison.

Method 2 Chain-of-Thought:

Input: Let's fix this step-by-step. The function is supposed to return true when n is even. Currently, it returns $n \% 2$, which gives 0 for even and 1 for odd — not a boolean. Instead, we should compare the result of $n \% 2$ with 0 to return a boolean.

ChatGPT Output:

```
function isEven(n) {  
    return n % 2 === 0;
```



```
}
```

ChatGPT Comments: Follows the reasoning path laid out in the prompt. Clearly understands that returning `n % 2` is not enough and that comparing it to 0 is essential for boolean correctness. Simple but accurate.

ClaudAI Output:

```
function isEven(n) {  
    return n % 2 === 0; // Explicitly returns true for even, false for odd  
}
```

ClaudAI Comments: Explains that using `n % 2 === 0` returns a boolean directly and addresses the mismatch in return types. Slightly more instructional by highlighting the return behavior explicitly.

Analysis: All outputs correctly solve the bug, producing functionally identical results. However, the chain-of-thought prompts provide better reasoning context, reinforcing why `n % 2` alone doesn't suffice. ClaudAI gives the clearest comment in both cases by explicitly mentioning boolean output. Still, given the simplicity of the task, zero-shot responses are efficient and sufficient.

Task 19:

19. **Summary Decomposition (C++)** Decompose the high-level comment/summary into logical steps:

```
// Function that validates an input, calculates square, and returns result  
int process(int x) {  
    if (x < 0) return -1;  
    return x * x;  
}
```

Method 1 Zero Shot:

Input: Decompose the following summary into the logical steps shown in the function

ChatGPT Output:

```
// Step 1: Check if the input is negative  
// Step 2: If it is, return -1 as an error indicator  
// Step 3: Otherwise, return the square of the input  
int process(int x) {  
    if (x < 0) return -1;  
    return x * x;  
}
```

ChatGPT Comments: Breaks the logic into three concise steps that clearly explain the control flow. Matches the function's behavior directly and efficiently.

ClaudAI Output:

The function `process(int x)` can be decomposed into these logical steps:

1. Take an integer input x
2. Check if x is less than 0
3. If x is negative, return -1
4. Otherwise, calculate the square of x (multiply x by itself)
5. Return the squared result

This function essentially computes the square of non-negative integers while returning -1 for any negative input.

ClaudAI Comments: Provides a readable bullet-style breakdown of the logic in prose form. While not in comment syntax, it's highly informative and pedagogical.

Method 2 Chain-of-Thought:

Input: Let's break down the high-level summary into specific logical steps. First, the function needs to validate the input: that means rejecting negative numbers. Next, if the input is valid, it must calculate the square. Lastly, the result should be returned. Now rewrite the comments as clear steps:

ChatGPT Output:

```
// Step 1: Validate the input by checking if it's non-negative
// Step 2: If the input is negative, return -1 to indicate invalid input
// Step 3: If valid, compute the square of the input
// Step 4: Return the computed square as the result
int process(int x) {
    if (x < 0) return -1;
    return x * x;
}
```

ChatGPT Comments: Explicitly labels each part of the process with numbered comments, guiding the reader from validation to return value. Great for educational purposes.

ClaudAI Output:

```
int process(int x) {
    // Step 1: Validate input by checking if x is negative
    if (x < 0)
        return -1; // Step 2: If input is invalid (negative), return error code -1

    // Step 3: Calculate the square of x for valid input
    // Step 4: Return the calculated result
    return x * x;
}
```

ClaudAI Comments: Presents clean inline comments inside the function, identifying validation, condition handling, and computation. Balanced between clarity and practicality.

Analysis: ChatGPT's chain-of-thought version is most readable for beginners, while Claude's zero-shot provides the clearest prose explanation. However, Claude's chain-of-thought version strikes the best balance by annotating the actual code with readable comments as you go through each line which is good for wanting to keep up as you progress.

Task 20:

20. **Purpose Inference → Completion (Python)** Complete the function based on intent:

```
def calculate_average(scores):  
    total = 0  
    # TODO: Complete to return average  
    pass
```

Method 1 Zero Shot:

Input: Complete the following function to return the average of a list of scores:

ChatGPT Output:

```
def calculate_average(scores):  
    total = sum(scores)  
    return total / len(scores)
```

ChatGPT Comments: Uses Python's built-in sum() and len() functions for a clean and concise solution. However, it does not explicitly handle edge cases like an empty list, which may lead to division by zero.

ClaudAI Output:

```
def calculate_average(scores):  
    total = 0  
    # Add up all the scores in the list  
    for score in scores:  
        total += score  
    # Calculate the average by dividing the total by the number of scores  
    # Handle empty list case to avoid division by zero  
    if len(scores) > 0:  
        average = total / len(scores)  
        return average  
    else:  
        return 0 # Return 0 for an empty list
```

ClaudAI Comments: Provides a robust version that includes an explicit loop and safeguards against division by zero. Slightly more verbose but safer for edge cases.

Method 2 Chain-of-Thought:

Input: Let's complete this step-by-step. First, we need to compute the total of all values in the list. Then, we divide that total by the number of elements in the list to get the average. Now fill in the function accordingly

ChatGPT Output:

```
def calculate_average(scores):  
    total = 0  
    for score in scores:  
        total += score
```

```
average = total / len(scores)
return average
```

ChatGPT Comments: Implements the logic manually with a loop, which enhances transparency. However, it assumes the list is non-empty, lacking a check for division by zero.

ClaudAI Output:

```
def calculate_average(scores):
    total = 0

    # Calculate the sum of all scores
    for score in scores:
        total += score

    # Calculate the average by dividing the total by the number of elements
    if len(scores) > 0: # Prevent division by zero
        average = total / len(scores)
        return average
    else:
        return 0 # Return 0 for empty lists
```

ClaudAI Comments: Goes beyond the basic logic by explaining each step with inline comments and explicitly checks for empty input. Well-suited for readability and safety.

Analysis: ClaudAI outperforms in both styles by handling empty lists. The chain-of-thought version from ClaudAI stands out for its clarity, making it the best-rounded response for real-world usage and teaching scenarios.

Task 21:

21. **Full-File Bug Detection and Refactoring (Python)** Analyze the following utility script. Identify any potential logic issues or design flaws. Then, refactor it for better readability, correctness, and safety.

Bonus Question — Worth up to 2.5 points toward the total grade

```
# utils.py - Script to parse and summarize numeric CSV files

import csv

def read_csv(filepath):
    with open(filepath, 'r') as f:
        return [row for row in csv.reader(f)]

def summarize_column(data, index):
    values = [float(row[index]) for row in data[1:]] # skip header
    total = sum(values)
    avg = total / len(values)
    return total, avg

def main():
    filepath = 'data.csv'
    data = read_csv(filepath)
    total, avg = summarize_column(data, 1)
    print("Total:", total)
    print("Average:", avg)

if __name__ == '__main__':
    main()
```

Method 1 Zero Shot:

Input: Refactor the following script to improve safety and correctness. Identify any logic or design issues:

ChatGPT Output:

```
import csv
import sys
from typing import List, Tuple

def read_csv(filepath: str) -> List[List[str]]:
    try:
        with open(filepath, 'r', newline='') as f:
            return [row for row in csv.reader(f)]
    except FileNotFoundError:
        print(f'Error: File '{filepath}' not found.')
        sys.exit(1)
    except Exception as e:
        print(f'Error reading file: {e}')
        sys.exit(1)

def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    if not data or len(data) < 2:
        raise ValueError("Insufficient data or missing header.")

    try:
        values = [float(row[index]) for row in data[1:] if len(row) > index and row[index]]
    except ValueError:
        raise ValueError(f'Non-numeric data encountered in column {index}.')

    if not values:
        raise ValueError("No valid numeric values found in column.")

    total = sum(values)
    avg = total / len(values)
    return total, avg

def main(filepath='data.csv', column_index=1):
    data = read_csv(filepath)
    try:
        total, avg = summarize_column(data, column_index)
        print(f'Total: {total}')
        print(f'Average: {avg}')
    except ValueError as ve:
        print(f'Error: {ve}')
        sys.exit(1)

if __name__ == '__main__':
    # Optional CLI argument support
    if len(sys.argv) > 1:
        main(sys.argv[1], int(sys.argv[2]) if len(sys.argv) > 2 else 1)
    else:
```

```
main()
```

ChatGPT Comments: Introduces robust exception handling with try/except, validates file existence, and avoids crashes via sys.exit(1). Enhances correctness and usability while maintaining clarity.

ClaudAI Output:

```
# utils.py - Script to parse and summarize numeric CSV files
import csv
import os
from typing import List, Tuple, Optional, Union
```

```
def read_csv(filepath: str) -> List[List[str]]:
    """
    Reads data from a CSV file.

    Args:
        filepath: Path to the CSV file

    Returns:
        List of rows from the CSV file

    Raises:
        FileNotFoundError: If the file doesn't exist
        PermissionError: If the file can't be accessed
    """
    if not os.path.exists(filepath):
        raise FileNotFoundError(f"File not found: {filepath}")

    try:
        with open(filepath, 'r', newline="") as f:
            return list(csv.reader(f))
    except (PermissionError, IOError) as e:
        raise IOError(f"Error reading file {filepath}: {str(e)}")


def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    """
    Summarizes a numeric column from CSV data.

    Args:
        data: CSV data as a list of rows
        index: Column index to summarize

    Returns:
        Tuple of (total, average)

    Raises:
        IndexError: If the column index is out of range
        ValueError: If the data can't be converted to float
        ZeroDivisionError: If the data has no rows (excluding header)
    """
```

```

"""
if not data or len(data) <= 1:
    raise ValueError("Data must contain at least one row of values (excluding header)")

if not data[0]:
    raise ValueError("Data contains empty rows")

if index < 0 or (len(data) > 0 and index >= len(data[0])):
    raise IndexError(f"Column index {index} out of range")

try:
    values = []
    for row in data[1:]: # Skip header
        if index < len(row):
            values.append(float(row[index]))
        else:
            raise IndexError(f"Column index {index} out of range for row")

    if not values:
        raise ValueError("No numeric values found in the specified column")

    total = sum(values)
    avg = total / len(values)
    return total, avg
except ValueError as e:
    raise ValueError(f"Error converting values to float: {str(e)}")

def main(filepath: Optional[str] = None) -> None:
    """
    Main function to process a CSV file and print summary statistics.

    Args:
        filepath: Path to the CSV file (defaults to 'data.csv' if None)
    """
    try:
        filepath = filepath or 'data.csv'
        print(f"Processing file: {filepath}")

        data = read_csv(filepath)
        if not data:
            print("Warning: File contains no data")
            return

        column_index = 1 # Default to second column

        # Check if the file has enough columns
        if len(data) > 0 and len(data[0]) <= column_index:
            print(f"Error: File doesn't have enough columns (looking for index {column_index})")
            return

        total, avg = summarize_column(data, column_index)

```

```

        print(f"Total: {total:.2f}")
        print(f"Average: {avg:.2f}")
    except Exception as e:
        print(f"Error: {str(e)}")

if __name__ == '__main__':
    main()

```

ClaudAI Comments: Adds comprehensive file validation and raises precise exceptions for readability and safety. Employs typed arguments and detailed docstrings, improving code maintainability and robustness.

Method 2 Chain-of-Thought:

Input:

Let's improve this script step-by-step. First, the function should validate that the file exists and contains rows. Next, handle the possibility of malformed or missing data in the target column. Finally, improve print formatting and refactor for clarity. Now write the refactored version:

ChatGPT Output:

utils.py - Script to parse and summarize numeric CSV files

```

import csv
import os

def read_csv(filepath):
    if not os.path.isfile(filepath):
        raise FileNotFoundError(f"File '{filepath}' does not exist.")

    with open(filepath, 'r', newline=") as f:
        reader = csv.reader(f)
        data = list(reader)

    if not data:
        raise ValueError("CSV file is empty.")

    return data

def summarize_column(data, index):
    if len(data) < 2:
        raise ValueError("CSV must contain a header and at least one row of data.")

    header, *rows = data
    values = []

    for i, row in enumerate(rows, start=2): # Start at 2 to reflect line number in file
        if len(row) <= index or not row[index].strip():
            print(f"Warning: Missing value in row {i}, skipping.")
            continue

```



```

    try:
        values.append(float(row[index]))
    except ValueError:
        print(f"Warning: Non-numeric value '{row[index]}' in row {i}, skipping.")

if not values:
    raise ValueError("No valid numeric data found in specified column.")

total = sum(values)
avg = total / len(values)
return total, avg

def main():
    filepath = 'data.csv'
    try:
        data = read_csv(filepath)
        total, avg = summarize_column(data, 1)
        print(f"Summary for column 1:")
        print(f" Total = {total:.2f}")
        print(f" Average = {avg:.2f}")
    except (FileNotFoundError, ValueError) as e:
        print("Error:", e)

if __name__ == '__main__':
    main()

```

ChatGPT Comments: Walks through improvements step-by-step: validates file, handles edge cases like missing or invalid data, and formats output for clarity. Warns about skipped rows, aiding debugging and reliability.

ClaudAI Output:

```

# utils.py - Script to parse and summarize numeric CSV files
import csv
import os
from typing import List, Tuple, Optional

```

```

def read_csv(filepath: str) -> List[List[str]]:
    """
    Reads data from a CSV file with validation.

    Args:
        filepath: Path to the CSV file

    Returns:
        List of rows from the CSV file

    Raises:
        FileNotFoundError: If the file doesn't exist
        ValueError: If the file is empty
    """

```

```

# Validate file exists
if not os.path.exists(filepath):
    raise FileNotFoundError(f"File not found: {filepath}")

# Read the file
with open(filepath, 'r', newline=") as f:
    data = list(csv.reader(f))

# Validate file contains data
if not data:
    raise ValueError(f"File {filepath} is empty")

return data

def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    """
    Summarizes a numeric column from CSV data.

    Args:
        data: CSV data as a list of rows
        index: Column index to summarize

    Returns:
        Tuple of (total, average)

    Raises:
        IndexError: If the column index is out of range
        ValueError: If the data can't be converted to float or no data rows exist
    """
    # Check that we have rows beyond the header
    if len(data) <= 1:
        raise ValueError("CSV must contain data rows beyond the header")

    # Collect valid numeric values
    values = []
    for i, row in enumerate(data[1:], start=1): # Enumerate for better error messages
        try:
            # Check if the index is valid for this row
            if index >= len(row):
                print(f"Warning: Row {i+1} does not have a column at index {index}. Skipping.")
                continue

            # Try to convert to float
            value = float(row[index])
            values.append(value)
        except ValueError:
            print(f"Warning: Row {i+1}, column {index} contains non-numeric value '{row[index]}'. Skipping.")

    # Check if we have any valid values

```

```

if not values:
    raise ValueError(f"No valid numeric values found in column {index}")

# Calculate statistics
total = sum(values)
avg = total / len(values)

return total, avg

def main(filepath: Optional[str] = None) -> None:
    """
    Main function to process a CSV file and print summary statistics.

    Args:
        filepath: Path to the CSV file (defaults to 'data.csv' if None)
    """
    try:
        # Use default filepath if none provided
        actual_filepath = filepath or 'data.csv'
        print(f"Processing file: {actual_filepath}")

        # Read data
        data = read_csv(actual_filepath)
        print(f"File loaded successfully with {len(data)} rows")

        # Process column with improved formatting
        column_index = 1 # Default to second column

        # Validate column index
        if len(data[0]) <= column_index:
            raise IndexError(f"Column index {column_index} is out of range. File only has {len(data[0])} columns.")

        # Get column name for better output
        column_name = f"{data[0][column_index]}" if len(data[0]) > column_index else f"at index {column_index}"
        print(f"Analyzing column {column_name}...")

        # Calculate summary statistics
        total, avg = summarize_column(data, column_index)

        # Print results with improved formatting
        print(f"Total: {total:.2f}")
        print(f"Average: {avg:.2f}")
        print(f"Sample size: {len(data) - 1} rows")

    except Exception as e:
        print(f"Error: {str(e)}")

```

```
if __name__ == '__main__':  
    main()
```

ClaudAI Comments: Offers a detailed and structured rewrite with validation, inline feedback on skipped rows, and user-friendly output formatting. Improves code safety, readability, and transparency in edge cases.

Analysis: The zero-shot responses focus on correctness and structure, but the chain-of-thought outputs provide more user-focused enhancements such as warnings for bad data and clearer formatting. ClaudeAI's chain-of-thought stands out for its defensive programming and inline feedback, making it the best solution.

Task 22:

22. **Code Completion and Robustness Enhancement (Python)** Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly.

Bonus Question — Worth up to 2.5 points toward the total grade

```
# file_processor.py - Incomplete script for processing text files  
  
import string  
  
def load_file(filepath):  
    with open(filepath, 'r') as f:  
        return f.readlines()  
  
def clean_line(line):  
    # TODO: Remove punctuation and make lowercase  
    pass  
  
def count_words(lines):  
    word_counts = {}  
    for line in lines:  
        clean = clean_line(line)  
        for word in clean.split():  
            word_counts[word] = word_counts.get(word, 0) + 1  
    return word_counts  
  
def main():
```

```
    filepath = 'input.txt'  
    lines = load_file(filepath)  
    counts = count_words(lines)  
    for word, count in sorted(counts.items()):  
        print(f"{word}: {count}")  
  
if __name__ == '__main__':  
    main()
```

Method 1 Zero Shot:

Input: Complete the following Python script to remove punctuation from each line, convert text to lowercase, and count word frequencies correctly

ChatGPT Output:

file_processor.py - Script for processing text files

import string

```

def load_file(filepath):
    with open(filepath, 'r') as f:
        return f.readlines()

def clean_line(line):
    # Remove punctuation and convert to lowercase
    translator = str.maketrans("", string.punctuation)
    return line.translate(translator).lower()

def count_words(lines):
    word_counts = {}
    for line in lines:
        clean = clean_line(line)
        for word in clean.split():
            word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts

def main():
    filepath = 'input.txt'
    lines = load_file(filepath)
    counts = count_words(lines)
    for word, count in sorted(counts.items()):
        print(f'{word}: {count}')

if __name__ == '__main__':
    main()

```

ChatGPT Comments: Implements the expected behavior cleanly—removes punctuation, lowers case, and counts words. The logic is simple, readable, and matches task intent without extra complexity.

ClaudAI Output:

file_processor.py - Script for processing text files and counting word frequencies

import string

```

def load_file(filepath):
    """
    Load all lines from a text file.

    Args:
        filepath: Path to the text file

    Returns:
        List of lines from the file
    """
    with open(filepath, 'r', encoding='utf-8') as f:
        return f.readlines()

```

```

def clean_line(line):
    """

```

Remove punctuation and convert text to lowercase.

Args:

line: The input text line

Returns:

Cleaned string with no punctuation and all lowercase

"""

Remove all punctuation and convert to lowercase

translator = str.maketrans("", "", string.punctuation)

cleaned = line.translate(translator).lower()

return cleaned

def count_words(lines):

"""

Count frequency of each word in the text.

Args:

lines: List of text lines

Returns:

Dictionary with words as keys and frequencies as values

"""

word_counts = {}

for line in lines:

clean = clean_line(line)

for word in clean.split():

if word: # Skip empty strings

word_counts[word] = word_counts.get(word, 0) + 1

return word_counts

def main():

"""

Main function to process a text file and print word frequencies.

"""

try:

filepath = 'input.txt'

print(f"Processing file: {filepath}")

lines = load_file(filepath)

counts = count_words(lines)

Print word counts sorted alphabetically

print(f"\nWord frequencies (total unique words: {len(counts)}):")

for word, count in sorted(counts.items()):

print(f"{word}: {count}")

except FileNotFoundError:

print(f"Error: File '{filepath}' not found.")

except Exception as e:

print(f"Error: {str(e)}")

```
if __name__ == '__main__':  
    main()
```

ClaudAI Comments: Includes robust enhancements like docstrings, encoding specification, and error handling. It's verbose but well-documented, offering a safer and more production-friendly version.

Method 2 Chain-of-Thought:

Input: Let's complete this step-by-step. First, convert the line to lowercase so the word counting is case-insensitive. Then, remove punctuation by translating all characters in `string.punctuation` to `None`. Finally, return the cleaned result.

ChatGPT Output:

```
def clean_line(line):  
    line = line.lower()  
    translator = str.maketrans("", "", string.punctuation)  
    return line.translate(translator)
```

ChatGPT Comments: Isolates and comments each transformation step in `clean_line`, helping clarify the reasoning behind the function changes. Code is correct and well-structured.

ClaudAI Output:

```
def clean_line(line):  
    # Step 1: Convert the line to lowercase for case-insensitive counting  
    lowercase_line = line.lower()  
  
    # Step 2: Remove punctuation using string.translate  
    translator = str.maketrans("", "", string.punctuation)  
    cleaned_line = lowercase_line.translate(translator)  
  
    # Step 3: Return the cleaned result  
    return cleaned_line
```

ClaudAI Comments: Clearly explains each processing step in the cleaning function using inline comments. The use of `lower()` and `translate()` is logical and reinforces proper text preprocessing.

Analysis: ClaudAI's Zero Shot output is the best overall. It not only completes the task correctly but also enhances the script with robust error handling, detailed docstrings, and encoding specification—making it more suitable for real-world usage. While the Chain-of-Thought outputs from both models are helpful for learning and clarity, they don't go as far in improving safety or maintainability. For practical deployment, ClaudAI's zero-shot solution provides the cleanest and most reliable implementation.