CompSci 520 - Exercise 1

Part 1: Prompt Design & Code Generation

I selected 10 problems from the MBPP-ET dataset, as listed. I used two distinct LLM families: **Qwen3** and **Claude**.

For each problem I applied two prompting strategies:

- **Chain-of-Thought (CoT)**
- **Stepwise Chain-of-Thought (SCOT).**

I evaluated the generated code using the pass@k metric (k=1,3,5) and summarized the results in below table. The exact prompts used are included below, followed by the results and a discussion

#	Task ID	Function Name	Description
1	200	`position_max`	Find all index positions of the maximum values in a list
2	540	`find_Diff`	Find the difference between highest and least frequencies in an array
3	175	`is_valid_parenthese`	Verify validity of a string of parentheses
4	466	`find_peak`	Find the peak element in the given array
5	67	`bell_number`	Find the number of ways to partition a set (Bell numbers)
6	426	`filter_oddnumbers`	Filter odd numbers using a lambda function
7	399	`bitwise_xor`	Perform bitwise XOR across given tuples
8	347	`count_Squares`	Count the number of squares in a rectangle
9	282	`sub_list`	Subtract two lists using map and lambda function

10	159	`month_season`	Print the season for the given month and day

Prompting Strategies Used

1. Chain-of-Thought (CoT)

Prompt:

{####problem statement}

Lemme thought process you this step by step:

1. *Firstly you will understand what the functions should do*
2. *Then, you will analyze the expected inputs and outputs*
3. *Next, you will think of how to approach this problem*
4. *Finally, you will implement the solution*

2. Stepwise Chain-of-Thought (SCOT)

Prompt:

{####problem statement}

Let's solve this using a systematic and proper stepwise approach:
Step 1: Problem Analysis: Understand what the function needs to do and know the inputs and expected outputs. Determine the main requirements.
Step 2: Algorithm Design: Choose the most appropriate approach or algorithm, take computational complexity under consideration (should be optimal), and plan the main logic flow.
Step 3: Edge Case Consideration: Identify potential edge cases (like empty inputs, single elements.) and design how to handle special scenarios. Consider input validation as needed.
Step 4: Implementation: Write the function step by step, implement the core logic, and handle the edge cases so that you don't fail on unseen test data.
Step 5: Verification: After writing the solution, review for correctness and check if it fulfills all the requirements. Now I'll implement the function.

Results Table

The table below summarizes the pass@k results for each problem selected, LLM, and prompting strategy chosen. Each row represents the pass@k value (fraction of solutions passing all tests out of 5 attempts).

#	Function	Qwen3-CoT	Qwen3-SCOT	Claude-CoT	Claude-SCOT
1	`position_max`	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00	1.00 / 1.00 / 1.00	1.00 / 1.00 / 1.00
2	`find_Diff`	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00
3	`is_valid_parenthese`	0.25 / 0.75 / 0.00	1.00 / 1.00 / 1.00	0.00 / 0.00 / 0.00	0.20 / 0.60 / 1.00
4	`find_peak`	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00
5	`bell_number`	1.00 / 1.00 / 0.00	0.80 / 1.00 / 1.00	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00
6	`filter_oddnumbers`	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00
7	`bitwise_xor`	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00
8	`count_Squares`	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00
9	`sub_list`	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00	1.00 / 1.00 / 0.00	1.00 / 1.00 / 1.00

10	`month_season`	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00	0.00 / 0.00 / 0.00

Table 1: pass@k results (k = 1, 3, 5) for 10 MBPP-ET problems across Qwen3 and Claude using CoT and SCoT prompting.

Note In the above table each cell is **pass@1 / pass@3 / pass@5**
From my experiment i have analysed that

- **High Success Problems:** Problems like `position_max`, `bell_number`, `filter_oddnumbers`, `bitwise_xor`, and `sub_list` were solved perfectly by both models and both prompting strategies (pass@1=1.00).
- **Challenging Problems:** Problems such as `find_Diff`, `find_peak`, `count_Squares`, and `month_season` had a 0% pass rate among both models and strategies, indicating either a clear misunderstanding of the problem, insufficient prompt clarity, or model limitations.
- **Prompting Impact**: For some problems (like `is_valid_parenthese`), the ***stepwise CoT strategy improved results especially for Qwen3***(this can be attributed to clear step thought process through prompting). For others, both strategies performed similarly.
- Model Differences: In some cases, Qwen3 outperformed Claude (e.g., `is_valid_parenthese` with SCOT), while in others, both models struggled equally (e.g., `find_Diff`).
- Average Success: The overall average pass@1 across all problems and models was around 0.55–0.6, showing that while LLMs can solve many problems reliably but some types of problems stil remain challenging.

Part 2: Debugging & Iterative Improvement

For this part I considered two problems from above, `find_Diff`, `count_Squares` and experimented how debugging and prompt refinement strategies helped improve the results.

Problem 1: find_Diff

Claude:

- Failure: Used wrong function signature (one parameter instead of two), causing all tests to fail.
- Debugging: Added explicit parameter analysis, clarified signature, gave a sample failed test case in the prompt for better understanding
- Improved Prompt: Clearly specified two parameters and correct usage.
- Result: All tests passed after fix.
- The reason why the model failed was it focused only on the algorithm and missed interface details.

Qwen3:

- Failure: Even with Qwen3 it was the same case it missed the details of function signature. It should have asked for the details instead of hallucinating.
- Debugging: Recognized the issue, matched the test signature, and handled edge cases.
- Improved Prompt: Matched function signature
- Result: All tests passed after fix.
- Why it failed: Assumed standard Python signature

Comparison:

Both models failed for the same reason (signature mismatch), and both were fixed by clarifying by providing the sample test case. Debugging was effective once the prompt was explicit.

Old Prompt:

Prompt:

{####problem statement}

'''

Let's solve this using a systematic and proper stepwise approach:
Step 1: Problem Analysis: Understand what the function needs to do and know the inputs and expected outputs. Determine the main requirements.
Step 2: Algorithm Design: Choose the most appropriate approach or algorithm, take computational complexity under consideration (should be optimal), and plan the main logic flow.
Step 3: Edge Case Consideration: Identify potential edge cases (like empty inputs, single elements.) and design how to handle special scenarios. Consider input

validation as needed.
Step 4: Implementation: Write the function step by step, implement the core logic, and handle the edge cases so that you don't fail on unseen test data.
Step 5: Verification: After writing the solution, review for correctness and check if it fulfills all the requirements. Now I'll implement the function.
'''

Improvised Prompt:

{####problem statement}

"

Above generated code is failing some test cases. Write a function `find_Diff` that takes a list of integers and returns the difference between the highest and lowest frequencies of elements in the list. Use `collections.Counter` to count frequencies. Handle empty lists by returning 0. Show the frequency dictionary before returning the result.
sample test cases

['assert find_Diff([1,1,2,2,7,8,4,5,1,4],10) == 2',
 'assert find_Diff([1,7,9,2,3,3,1,3,3],9) == 3',]
'''

Problem 2: count_Squares

Claude:

- Failure: Used standard iterative approach but in test cases there were few test cases which expected a different mathematical formula.
- Debugging: provided sample test cases, and asked it to ask follow up question if anything is unclear
- Improved Prompt: Explicitly stated to use different mathematical approaches other than iterative.
- Result: All tests passed after fix.
- Why it failed: Ambiguous problem statement, there are multiple valid interpretations for this problem statement.

Qwen3:

- Failure: Used standard geometric definition, but test cases were inconsistent or ambiguous.
- Debugging: Pointed out ambiguity, requested clarification, and asked .

- Improved Prompt: Explicitly stated to use different mathematical approaches other than iterative.
- Result: Correct for standard definition; test cases may need review, model pointed out that the test case might be wrong and the approach is correct. It was trying to fit the given test cases(in prompt) and give the solution which led to wrong solutions for other problems.
- Why it failed: Problem ambiguity, unclear requirements.

Old Prompt:

Prompt:

{####problem statement}

‘’’

Let's solve this using a systematic and proper stepwise approach:
Step 1: Problem Analysis: Understand what the function needs to do and know the inputs and expected outputs. Determine the main requirements.
Step 2: Algorithm Design: Choose the most appropriate approach or algorithm, take computational complexity under consideration (should be optimal), and plan the main logic flow.
Step 3: Edge Case Consideration: Identify potential edge cases (like empty inputs, single elements.) and design how to handle special scenarios. Consider input validation as needed.
Step 4: Implementation: Write the function step by step, implement the core logic, and handle the edge cases so that you don’t fail on unseen test data.
Step 5: Verification: After writing the solution, review for correctness and check if it fulfills all the requirements. Now I'll implement the function.
‘’’

Improvised Prompt:

{####problem statement}

‘’’Let's debug your code. For m=4, n=3, print the number of squares for each possible size (1x1, 2x2, 3x3), then sum them. If m or n is 0, return 0.

sample test case where it failed

Test 4 FAILED: assert count_Squares(5, 2) == 10 , the generated code gives 14

Test 13 FAILED: assert count_Squares(3, 7) == 28 , the generated code give 38

Show me your test case verification like For count_Squares(5,2), show me step-by-step what your code returns and break down the calculation. Try different

approaches other than iterative ones. Also please Don't guess, if the test cases don't match your understanding ask me for more clarification and test cases
”’

Comparison:

Claude gave the correct code when asked for some mathematical approach. Qwen3 highlighted ambiguity and requested clarification, showing good engineering practice but ultimately couldn't give the right solution. Both models did show some relatively better results but **claude performed well.**

My understanding

- Both models failed due to unclear/vague requirements or signature mismatches.
- Explicit prompts and parameter analysis with possible test cases are crucial for success.
- Providing formulas and clarifying expectations resolves most failures.
- Qwen3 is more likely to request clarification when requirements are ambiguous; Claude responds well to direct instructions.

Good Practice

- Always match function signatures to test cases.
- Provide explicit formulas for mathematical problems.
- Clarify ambiguous requirements with examples outputs.
- **Debugging prompts to guide LLMs really helped to get better solutions.**

Part 3: Innovation - Propose Your Own Strategy

STREW (Specification-Test-Refine Workflow)

From the experiments that I conducted I understood that most LLM failures were due to function signature mismatches, ambiguous problem statements, or missed edge cases. Stepwise COT prompting seems to improve reasoning to an extent but not verification. Particular sample tests and debug prompts increased accuracy(as seen in part B). Qwen3 seems to ask clarifying questions, while Claude follows strict instructions. I would like to propose a STREW workflow which combines structured specification, self-testing, and iterative refinement to address these issues.

Steps

1. Specification & Clarification (Spec Phase)

- Goal: Prevent function's signature and ambiguity errors.
- Output: Function name and signature, input/output types, edge cases, and couple of sample input-output pairs.
- If ambiguous: Model notes down the clarifying questions before proceeding to implement.

2. Initial Implementation & Self-Test (Test Phase)

- Goal: Generate code and verify correctness internally for the first time.
- LLM Role: Implementation + Test Engineer.
- Steps: Implement function based on Specification Phase, generate a couple of self-tests (including good edge cases), simulate execution, and compare results to expected outputs.

3. Error Analysis & Iterative Refinement (Refine Phase)

- Goal: Given the failure cases try to include this case and redefine solution(dont over fit)
- LLM Role: Debugger.
- Steps: Identify failure test cases, reason out why it happened, suggest minimal changes, update code, and re-run to evaluate these cases again. Repeat until all tests pass or after 2–3 iterations.

Example Prompts

Spec Phase

Prompt:

‴

Problem: {#####problem statement}

Task: Produce a precise function specification:
- Function name & signature
- Input/output types
- Edge cases & assumptions
- couple of sample input/output pairs

If ambiguous: include a clarifying question or assumption.
Output Json only
‴

Test Phase Prompt:

“”

Based on the specification JSON:
- Implement the function exactly as specified.
- Generate a couple of self-tests (combination of normal + edge cases).
- For each test, show input, expected output, and the output your code returns.
- Indicate which tests fail.

“”

Refine Phase Prompt:

‴

Some self-tests failed:
- Explain the failure cause.
- Correct the code with minimal changes (do not rewrite unrelated parts).
- Re-run the tests and show updated outputs.
- Repeat until all self-tests pass or after 2 iterations.

“”

Application to selected previously failed Problems

`find_peak`: Spec Phase clarifies correct signature and edge cases. Test Phase exposes failure for empty list. Refine Phase adds conditional for empty list, all self-tests pass.
`month_season`: Spec Phase defines combinatorial formula. Test Phase detects mismatch with sample test. Refine Phase adjusts formula with minimal code changes.

Problem	Model	Baseline pass@1	After STREW pass@1	Key Failure Fixed by STREW
find_peak	Claude	0.00	1.00	Signature mismatch, edge cases
find_peak	Qwen3	0.00	1.00	Signature mismatch, edge cases
month_season	Claude	0.00	1.00	Incorrect data type data(month number instead of string)
month_season	Qwen3	0.00	1.00	Incorrect data type data(month number instead of string)

Why STREW Works

Failure Source	STREW Solution
Signature mismatch	Spec Phase enforces exact signature and types
Ambiguous requirements	Spec Phase clarifies assumptions and edge cases
logic errors	Test + Refine Phase exposes and corrects them

Edge-case omission	Self-tests cover normal + edge + extreme cases

---

Scenarios where this strategy may FAIL?

STREW Worked for the above problems because of their less complexity. But there might be scenarios where this might fail sometime mainly because of flawed assumptions. Firstly it assumes LLMs can *accurately self-test* code but in reality, they only simulate execution, often missing hidden edge cases. Secondly the multi-phase prompting assumes consistent memory across all the steps but in UI-based runs, there is a good chance of context loss and can cause the model to forget earlier constraints. Finally, when the problem is too convoluted its not really easy to iteratively make small changes and end up with good solution.

Table : Experimenting with STREW i found the below observations

Model	**Observations**
Claude	• Follows instructions strictly • Strong at logic and structure • Needs explicit signature and formula • Responds well to stepwise and test-driven prompts. It did ask for relevant questions before proceeding like : The main ambiguities are: 1. **Astronomical vs meteorological seasons** - which system to use? 2. **"fall" vs "autumn"** - which term should be returned? 3. **Hemisphere** - Northern or Southern? 4. **Input validation** - should invalid dates be handled?
Qwen3	• Asks clarifying questions when ambiguous • Good at catching edge cases • May request more examples • Excels with structured, multi-step workflows Had to provide explicit failed test cases to

	produce the correct solution.

From above strategy we can expect

- Pass@1 expected to improve from ~0.55 baseline to ~0.75–0.8 for previously failing cases.
- Especially effective for signature-sensitive(number of params, input/output data types) and ambiguous mathematical problems.

Summary

The STREW workflow systematically improves LLM code generation by making sure clear specifications, internal self-testing, and focused iterative refinement on failure cases. It is simple, effective, which leads to higher reliability and accuracy for challenging problems

Methodology for the experiment :

- **Models used:** Claude, Qwen3 (UI)
- **Prompting approach:** CoT and SCoT(Stepwise), number of generations per problem
- **Evaluation metric:** explain pass@k (k=1,3,5)
- **Testing setup:** I ran a python code to evaluate the generated codes and found the pass@k and appropriate accuracies.
- **Dataset or problems:** source (MBPP-ET) and selection criteria (10 random diverse problems).
- **Any constraints:** All generations were collected manually via the web UI.