"Why is my code slow?" Efficiency Bugs in Student Code

Hope Dargan
hoped@mit.edu
MIT CSAIL
Cambridge, MA, USA

Adam Gilbert-Diamond
adam2718@mit.edu
MIT CSAIL
Cambridge, MA, USA

Adam J. Hartz
hz@mit.edu
MIT EECS
Cambridge, MA, USA

Robert C. Miller
rcm@mit.edu
MIT CSAIL
Cambridge, MA, USA

Abstract

While prior research has categorized common errors and code quality issues of student programmers, little attention has been paid to researching student efficiency bugs. Qualitative content analysis of 250 slow student submissions across five CS2 assignments yielded over 750 efficiency bugs. Extracting general themes resulted in an efficiency bug taxonomy with three main categories: superfluous computation, suboptimal data structure design, and suboptimal algorithm design, with 12 subcategories. Analysis of specific bug frequencies across the assignments provided insights that may inform content design for programming courses.

CCS Concepts

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → **Software performance**.

Keywords

programming education; programming mistakes; software performance antipatterns; efficiency; code quality

ACM Reference Format:
Hope Dargan, Adam Gilbert-Diamond, Adam J. Hartz, and Robert C. Miller. 2025. "Why is my code slow?" Efficiency Bugs in Student Code. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701939

1 Introduction

Programming students often start by learning how to write correct code before learning how to write code with good style. As students progress and encounter more complex assignments with runtime constraints, they learn how to write efficient code. While prior research has devoted considerable attention to categorizing common student errors [1, 5, 9–11] and code quality issues [2, 4, 6, 8, 12, 16, 17], we are not aware of any prior work that focuses on categorizing efficiency bugs in student code.

A recent study by Izu and Mirolo [6] found that by the end of a CS1 course many students thought performance was an important aspect of code quality, but they often had flawed reasoning about program performance. Specifically, their study had 79 students at week 22 in a CS1 course rank five correct code samples from best to worst and explain the criteria for their ranking. They found that the top criterion students used for quality assessment was performance, but noted that 68% of flawed ranking explanations were due to assessing performance incorrectly.

In order to design course content that can better equip students to assess program performance and address efficiency bugs, we need to understand what issues they are encountering. This paper aims to categorize the early efficiency bugs that students make in order to help instructors teach students explicitly about common efficiency bugs and design assignments with these bugs in mind.

This paper explores the efficiency bugs in student code that is mostly correct but slow by investigating the following research questions. First, what efficiency bugs occur in student code? Second, which efficiency bugs occur only in particular assignments, and which occur generally across assignments? Investigation of these questions through qualitative content analysis of 250 randomly-selected slow submissions across five assignments revealed over 750 specific time-related efficiency bugs. Iteratively extracting general themes and creating terms and definitions for the most common efficiency bugs resulted in a taxonomy with 12 specific labels that were then applied to the selected submissions.

The taxonomy proposed in this paper consists of three main efficiency bug categories: superfluous computation, suboptimal data structure design, and suboptimal algorithm design. Within these three categories we defined 12 specific subcategories and created illustrative code snippets. Additionally, superfluous computations such as Failing to Cache, Failing to Short Circuit, or Executing Dead Code were present to varying degrees across all assignments. Other efficiency bugs such as Solving Irrelevant Subproblems or Misordering Subproblems tended to be very frequent within specific assignments. These insights may inform assignment-design around specific bugs.

2 Background and Related Work

Many prior works have investigated errors in student code and created taxonomies of common mistakes. For example, Hristova et al. [5] conducted a survey of students and teachers to identify Java programming bugs and extracted common themes to create a taxonomy of 20 student errors that were split into categories of syntax, semantic, and logic errors. McCall and Kölling [11] manually analyzed 197 code samples from an introductory Java course and

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0531-1/25/02
https://doi.org/10.1145/3641554.3701939

found 80 specific errors that they classified as syntax, semantic, and logic errors, the same category labels used by Hristova et al. McCall and Kölling found that analyzing student code samples by hand, rather than by using compiler error messages, led to more precise and reliable identification and ranking of student error frequencies, due to both the disconnect between student errors and automated diagnostic messages and the fact that student code could contain multiple distinct errors on the same line. Altadmri and Brown [1] developed a script to automatically analyze over 37 million code samples from the Blackbox database to study the frequency and time-to-fix of different errors, using 18 of the 20 student errors identified by Hristova et al. Mase and Nel [9] used manual content analysis to review over 650 C# code samples generated by 38 novice CS students to identify 21 common student code errors. The study expanded on the error categories of Altadmri and Brown, resulting in nine new errors and rearranging the error classification to include type errors in addition to the categories of style, semantic, and logic errors proposed by Hristova et al.

In addition to [6], other CS education papers have encountered efficiency as a factor related to code quality. In 2017, an ITiCSE working group [2] administered a survey on code quality to students, teachers, and developers. Through qualitative analysis, the group determined that all three groups listed readability and structure as important aspects of code quality. The group also found that over a quarter of respondents listed dynamic behavior, which includes good performance, as an important aspect of code quality.

Other studies have designed taxonomies for student style and code quality errors, which have some efficiency implications. For example, Stegeman et al. [17] developed a style rubric that includes ten categories; the first five categories relate to the readability and code style whereas the second five categories relate to algorithms and code structure. Keuning et al. [8] used the Java PMD tool to detect 24 code quality issues in 2.6 million novice code samples from the Blackbox dataset, dividing the issues into five categories taken from Stegeman et al. relating to algorithms and code structure. While these two studies did not focus on efficiency specifically, some of their general code quality categories have efficiency implications. For example, the idiom category specifies that students should use library functionality, the expressions category encourages appropriate data types, and the flow category discourages code duplication, all of which may have an impact on efficiency.

Other prior work in the area of software performance engineering has identified common software performance antipatterns and methods for detecting, refactoring, and modeling such performance issues [13, 15]. However, this work mainly relates to large scale systems involving hardware, databases, and concurrency issues that are mostly beyond the scope of CS2 courses.

3 Method

This paper seeks to study the efficiency bugs that students make by investigating the following questions:

- RQ1. What efficiency bugs occur in student code?
- RQ2. Which efficiency bugs occur only in particular assignments, and which occur generally across assignments?

Like some previous studies that categorized mistakes made by programming students [9, 11], we investigated our research questions primarily through qualitative content analysis of pseudonymized student code samples. The student code samples in this study came from the files submitted to our automatic grading server by students who registered for our CS2 course during the Fall 2023 semester. This study obtained an IRB exemption because the assignment files were collected as a regular part of running the course.

3.1 Data Collection

Our CS2 course covers topics such as higher-order functions, data structure design, graph search, recursion, backtracking, and object-oriented programming. While efficiency is not our primary focus, we do touch on it, for example, discussing data structure choice and linear-time versus constant-time operations, optimizing graph search with a visited-node set, and pruning a backtracking search.

Students learn to apply course topics by completing 12 week-long Python assignments that involve implementing multiple related functions in a single file in order to pass given test cases. A majority of the grade for each assignment is determined by the automatic grading server, which allows students to submit repeatedly before the deadline. The server assesses assignment files by running various groups of test cases and recording which tests in each group pass within a hand-tuned time limit. Because code efficiency can act as a proxy for understanding of some of the topics we cover, some test case groups are intentionally designed with large inputs and time limits that are intended to make code with poor algorithm or data structure choices exceed the time limit.

For the purposes of this study, we are primarily concerned with categorizing the most severe runtime efficiency bugs, so we only examined student submissions that experienced a timeout on our automatic grading server. During the Fall 2023 semester, 408 students submitted 19,824 assignment files to the server. Of these files, roughly 43% passed all test cases under the specified time limit, 15% timed out on at least one group of tests, and the remaining 42% experienced a correctness error. We filtered the assignment files to only include timeouts, since timing out is evidence that there is at least one severe efficiency bug preventing the submission from passing the tests within the time limit. We also removed timeout submissions that had major correctness issues, as evidenced by failing most of the test cases before the timeout, because we encourage students to focus on correctness before efficiency. Because students were allowed to submit repeatedly, we sought to avoid sampling duplicate submissions by including each student's first and last timeout submission for each assignment. If the first and last timeout submission for a student was sampled, we manually inspected the files and discarded one if they were nearly identical.

From our twelve assignments, we selected five assignments which accounted for 80% of the submissions that timed out. Our analysis focused on 11 specific test cases across the five assignments that were the most common bottlenecks for students who experienced timeouts. Multiple criteria were considered as part of this selection process including the percentage of students who experienced a timeout on the assignment, the time difference between the fastest 10% of submissions and the timeout submissions on the test case, what part of assignment functionality the test case covered,

Efficiency Bug Category	Definition	% Subs (/250)	% Bugs (/753)
Superfluous Computation			
Failing to Cache	Not storing or not using data resulting in repeated computation.	36% (90)	18% (136)
Failing to Short Circuit	Continuing a computation after the answer has been determined or ordering computation in a way that prevents exiting sooner.	40% (101)	15% (111)
Executing Dead Code	Code that can be removed with very slight changes that would increase efficiency without sacrificing correctness. Does not involve copying data.	32% (80)	13% (101)
Extra Copying	Creating shallow or deep copies of data that will not be modified or that can safely be mutated.	26% (65)	10% (72)
Failing to Hoist	Performing a computation at every recursive step or loop iteration when a single computation external to the structure would suffice.	13% (32)	4% (32)
Extra Loop / Recursive Overhead	Performing necessary distinct steps with similar repeated iterative or recursive structures, when the steps could be performed more efficiently in one structure.	5% (13)	2% (14)
	Total	84% (210)	62% (466)
Suboptimal Data Structure Design			
Slow Data Structure	Poor choice of data structure resulting in inefficient operations.	40% (101)	18% (132)
Expensive Data Transformation	The amount of time taken to create a data structure is significantly greater than the amount of time saved by using this new structure.	8% (20)	3% (23)
	Total	43% (108)	21% (155)
Suboptimal Algorithm Design			
Solving Irrelevant Subproblems	A brute-force approach involving solving unnecessary subproblems before checking constraints.	20% (51)	8% (58)
Misordering Subproblems	Failing to explore subproblems according to a heuristic when doing so would lead to efficiency gains.	15% (38)	5% (39)
Repeatedly Solving Subproblems	Re-exploring parts of the problem space.	10% (24)	3% (25)
Reimplementing Builtins	Implementing an algorithm instead of using a more efficient builtin.	4% (10)	1% (10)
	Total	46% (116)	18% (132)

Table 1: Efficiency Bug Taxonomy and Overall Submission and Bug Frequency

and how long it would take the test case to be profiled. To help find efficiency bugs for classification, we profiled each of the filtered slow submissions for up to five minutes using a profiling tool based on the Python line_profiler tool [7]. Profiling submissions highlighted the most time-intensive functions and lines, which informed our qualitative descriptions of efficiency bugs.

3.2 Data Analysis

After filtering and profiling timeout submissions, two of the authors collaboratively performed qualitative content analysis as described by Richards and Hemphill [14]. In order to get a representative sample of efficiency bugs, we labeled submissions in batches consisting of three random submissions from each of the five assignments until we had labeled 50 submissions for each of the five assignments. We iteratively developed our taxonomy by extracting common themes as we labeled efficiency bugs with a description of their root cause. When creating categories, we sought to create definitions that were both precise enough to reduce potential overlap and general enough to apply to contexts outside of the assignments we examined.

We took a number of steps to maintain consistency between the two authors who labeled the efficiency bugs, including labeling the 45 submissions in the first three batches together to create an initial taxonomy and a list of common bugs for each category and assignment. Once the initial taxonomy was established, both labelers independently labeled the 250 randomly selected submissions, overlapping on 76, with the first author labeling 142 additional submissions and the second labeling the remaining 26.

After the initial labeling was completed, we consolidated the 342 bugs from the 76 submissions labeled by both authors down to 131 duplicate bugs and 80 unique bugs (43 from the first author and 37 from the second author). We verified bugs that were unique and created a summary of labeling differences. Then, we analyzed all submissions again specifically for those bugs to ensure consistency.

After examining the frequency of efficiency bugs across the 250 submissions, we adjusted the categories so that all labels were time-related and most subcategories were present in at least 5% of samples across multiple assignments. Once we were satisfied with the final taxonomy, we relabeled all of the recorded bugs.

Throughout the labeling process, we focused on recording all efficiency bugs that presented potential barriers to the code passing within the time limit. The number of efficiency bugs labeled for a single submission varied from as few as one to as many as eight. Some submissions had multiple unrelated occurrences of the same kind of efficiency bug recorded. If in doubt of the severity or cause of an issue, we tested fixing a submission, and would only record the bug if there was a noticeable difference in runtime.

4 Results

This section presents the results of the qualitative code analysis: first the taxonomy of efficiency bugs, and then which bugs tended to occur in particular assignments and which occurred more generally across assignments.

4.1 Efficiency Bug Taxonomy

Analysis of the 250 randomly-selected timeout submissions resulted in 753 time-related efficiency bugs, which we categorized into three main categories and 12 subcategories. The definitions of subcategories and their frequencies can be found in Table 1. The following sections provide further details and illustrative Python examples. Additional examples can be found in a copy of the efficiency reading we created for our students based on this work[3].

4.1.1 Superfluous Computation. We defined superfluous computation as code that can be deleted or computation that can be repeated fewer times without sacrificing correctness. The first two bugs in this category, Failing to Cache and Failing to Short Circuit, together accounted for almost a third of all labeled efficiency bugs.

Failing to Cache: Fixing this bug generally requires caching or making use of cached values. The code below is an example of both calling `slow_call(x)` multiple times instead of caching it, and calling `slow_call(y)` again instead of using the value in `new_y`.

```
new_y = slow_call(y)  # unused cached value
if slow_call(x) < 0:  # Fix: cache slow_call(x)
    foo(slow_call(x), slow_call(y)) # Fix: use new_y
else:
    bar(slow_call(x), slow_call(y))
```

Failing to Short Circuit: Fixing this bug generally requires adding conditional checks to exit earlier, or checking less expensive exit conditions sooner. For example, in the code below, we can break the loop as soon as we find the first spelling error.

```
has_error = False
for word in book:
    if spelling_error(word):
       has_error = True # Fix: break here
```

Executing Dead Code: Fixing this bug requires deleting unnecessary code. 17% of observed instances of this bug involved debugging-print statements, which are easy to find and remove. However, sometimes this bug involved other unnecessary additional computation, as in the if statement below.

```
for r in range(height):
    for c in range(width): # Fix: remove conditional
        if 0 <= r < height and 0 <= c < width:
            array[r][c] += 1
```

Extra Copying: This bug can generally be fixed by not copying unmodified structures or by modifying existing structures instead of creating new ones. For instance, the following code snippet has five instances of unnecessary copying.

```
names = []
for fname in first_name_strings[:]: # remove [:]
  for lname in last_name_strings[:]: # remove [:]
    full_name =str(fname)+" "+str(lname) # no str
    names = names + [full_name] # list + list copies
```

Failing to Hoist: This efficiency bug can be fixed by moving computation that only needs to get executed once outside of repeated structures. While this bug was often observed in recursive functions, an iterative example is shown below.

```
def update(pond, moves):
    for direction in moves:
        frog = get_frog_loc(pond)
        pond_size = get_pond_size(pond)
        # Fix: previous two lines go before loop
        frog = move(frog, direction, pond_size)
        # Fix: line below goes after loop
        pond = make_pond(pond_size, frog)
    return pond
```

Extra Loop / Recursive Overhead: Fixing this efficiency bug generally requires combining loops or recursive functions. For example, the adjacent loops with identical loop-control logic below can be combined into a single loop.

```
xo_matrix = []
for i in range(100_000):
    xo_matrix.append("X")
for i in range(100_000): # Fix: use one loop not two
    if (i % 2) == 1:
        xo_matrix[i] = "O"
```

4.1.2 Suboptimal Data Structure Design. We defined suboptimal data structure design as creating or using a data structure that has slow operations, when a more efficient alternate structure exists.

Slow Data Structure: This bug can generally be fixed by changing the data structure to make frequent operations more efficient. Like the example below, nearly 32% of Slow Data Structures that we labeled used a list or tuple instead of a set.

```
unique_numbers = [] # Fix: use a set
for phone_number in call_history:
    if phone_number not in unique_numbers:
        unique_numbers.append(phone_number)
```

Expensive Data Transformation: Fixing this bug generally involves finding a more efficient way to transform the data or simply not creating the structure. The example below creates a completely extraneous data structure, which was relatively rare in the submissions we examined.

```
chosen_num = int(input("Pick a number: "))
ten_million_nums = set(range(10_000_000))
num_in_range = chosen_num in ten_million_nums
# Fix: replace with 0 <= chosen_num < 10_000_000
```

4.1.3 Suboptimal Algorithm Design. We define a suboptimal algorithm as a correct algorithm that does not take advantage of problem characteristics that would enable a significantly more efficient algorithm. This includes code that fails to fully use optimizations or problem constraints, resulting in additional computation.

Solving Irrelevant Subproblems: This bug can generally be fixed by enforcing constraints to avoid wasted computation. In the code below, we can fix the efficiency bug by limiting the area of possible moves to adjacent horizontal or vertical locations instead of all locations.

```
def get_valid_moves(board, loc):
    valid_moves = set()
    for r, c in get_all_locations(board):
        distance = abs(r - loc[0]) + abs(c - loc[1])
        if distance == 1:
            valid_moves.add((r, c))
    return valid_moves
```

Misordering Subproblems: This efficiency bug can often be fixed by using a valid heuristic to select the next subproblem. In the code below, it would be faster to loop through words from shortest to longest and stop at the first word that starts with "z".

```
shortest_z_word = None
for word in longest_to_shortest_words:
    if word[0:1] == "z":
        shortest_z_word = word
```

Repeatedly Solving Subproblems: This efficiency bug is typically fixed by exploring the problem space in a way that avoids redundant work by keeping track of which parts of the problem space have been visited previously, as the example below shows.

```
all_pairwise_sums = set()
for i in range(0, len(nums)):
    for j in range(0, len(nums)): # Fix: start at i
        all_pairwise_sums.add(nums[i] + nums[j])
```

Reimplementing Builtins: Fixing this bug involves using an existing operation in the programming language or library. While this category represented only 1% of recorded efficiency bugs, it is an example of a language-specific bug that depends on both the available builtins and how efficiently they are implemented.

```
has_zebra = False
for word in animal_set:
    if word == "zebra": has_zebra = True
# Fix: has_zebra = "zebra" in animal_set
```

4.2 Assignment-Specific vs. General Bugs

Because the five assignments we examined covered a diverse array of topics, we expected variation by assignment in frequency of efficiency bug categories, especially across the subcategories of suboptimal data structure design and suboptimal algorithm design. Results in Table 2, which show the percentage of selected submissions that had at least one occurrence of a particular bug, indicate that this was indeed the case.

The assignments we selected covered a variety of topics including graph search, data transformation, recursion, backtracking, and object-oriented programming. Specific assignment names, topics, and brief descriptions are listed below.

- **Bacon Number** (graph search, data transformation): Given a raw database of actors, students first transform data. They then implement functions that find actors with a certain "Bacon" number and find a shortest path between actors.
- **Sokoban** (graph search, data transformation): Students implement an interface for a Sokoban game, then find the shortest sequence of moves that can solve the puzzle.
- **Minesweeper** (recursion): Given an inefficienct 2D implementation of minesweeper with bad style, students refactor the 2D version and implement an N-Dimensional version.
- **SAT Solver** (backtracking, recursion): Given a CNF formula, students implement an algorithm to either find a mapping of variables to values that satisfies the formula or determine that no such mapping exists.
- **Word Filter** (object-oriented programming, recursion): Students implement a prefix-tree class to efficiently autocomplete words and filter words that match a wildcard pattern.

The sections below discuss the assignment-dependent and assignment-independent subcategories of each of the three main efficiency bug classes.

4.2.1 Superfluous Computation. Some bugs in this category were prevalent in specific assignments, likely as a result of assignment design. For example, a majority of selected Minesweeper submissions had Failing to Hoist bugs likely because the assignment was designed with a single step function that both uncovered squares and determined the state of the game. As students recursively uncovered squares surrounding zeroes, they often repeatedly performed expensive game state checks. If we had designed Minesweeper to have a separate game state check function like in Sokoban, Failing to Hoist would likely not have been an issue at all. As another example, Extra Copying was most common in Sokoban because students often preemptively copied their game representation on every move before checking whether an obstacle would prevent a move. Extra Copying bugs were less common in Minesweeper because the game state was mutable.

The most common bugs of Failing to Cache, Failing to Short Circuit, and Executing Dead Code appeared across all assignments to varying degrees. The frequency of Failing to Cache bugs in particular may be a function of the fact that all our assignments feature interrelated functions. For example, in Sokoban, Minesweeper, SAT Solver, and Word Filter, many operations were performed by calling helper functions. While Executing Dead Code sometimes involved print statements, it was more often caused by over-complicating the problem. This was especially easy to do in the recursive assignments because in addition to being more complex than earlier assignments, students new to recursion often added unnecessary base cases and recursive cases.

4.2.2 Suboptimal Data Structure Design. Of the five assignments we looked at, Slow Data Structures were most present in Bacon Number and Sokoban, likely because by design computing the neighbor states efficiently during a graph search required transforming the original Slow Data Structure that we gave students. Occurrences of Slow Data Structures were far less frequent in the other assignments, likely because students were instructed to implement or return certain data structures. Instances of Slow Data Structures in the other assignments involved mostly using lists instead of sets to remove duplicates or do repeated containment checks. This may suggest that when given Slow Data Structures and design freedom, students find designing better data structures particularly challenging.

4.2.3 Suboptimal Algorithm Design. Frequencies of subcategories of this efficiency bug were mostly tied to particular assignments. For example, Misordering Subproblems was specific to SAT Solver because one of the suggested optimizations in the assignment writeup was assigning unit clauses first, so students failing to take that hint

Efficiency Bug Category	Bacon Number % Subs (/50)	Sokoban % Subs (/50)	Minesweeper % Subs (/50)	SAT Solver % Subs (/50)	Word Filter % Subs (/50)
Superfluous Computation					
Failing to Cache	12% (6)	40% (20)	84% (42)	24% (12)	20% (10)
Failing to Short Circuit	62% (31)	28% (14)	48% (24)	46% (23)	18% (9)
Executing Dead Code	14% (7)	12% (6)	54% (27)	32% (16)	48% (24)
Extra Copying	8% (4)	78% (39)	12% (6)	22% (11)	10% (5)
Failing to Hoist	2% (1)	0%	58% (29)	0%	4% (2)
Extra Loop / Recursive Overhead	14% (7)	2% (1)	10% (5)	0%	0%
Total	80% (40)	88% (44)	96% (48)	78% (39)	78% (39)
Suboptimal Data Structure Design					
Slow Data Structure	68% (34)	90% (45)	24% (12)	0%	20% (10)
Expensive Data Transformation	22% (11)	8% (4)	8% (4)	0%	2% (1)
Total	72% (36)	92% (46)	30% (15)	0%	22% (11)
Suboptimal Algorithm Design					
Solving Irrelevant Subproblems	0%	0%	62% (31)	0%	40% (20)
Misordering Subproblems	0%	0%	0%	76% (38)	0%
Repeatedly Solving Subproblems	32% (16)	10% (5)	4% (2)	0%	2% (1)
Reimplementing Builtins	0%	0%	0%	0%	20% (10)
Total	32% (16)	10% (5)	64% (32)	76% (38)	50% (25)

Table 2: Efficiency Bug Frequency in Selected Submissions, Grouped by Assignment

would have this kind of bug. Repeatedly Solving Subproblems was most frequent in Bacon Number and Sokoban due to submissions that explored unnecessary paths because they failed to track which nodes had already been visited. Interestingly, this bug was three times more frequent in Bacon Number than in the later Sokoban assignment, suggesting that students were better at avoiding this bug once they were more familiar with breadth-first search. Solving Irrelevant Subproblems mostly occurred in Minesweeper and Word Filter due to missed opportunities to constrain possible subproblems with the available data that were either not specified or overlooked in the assignment writeup. Reimplementing Builtins was mostly found in Word Filter due to submissions that sorted a list of words by their frequency with a handwritten quadratic sort instead of using Python's builtin sort method.

4.3 Threats to Validity

A potential threat to validity is selection bias, since this study looked only at submissions to a specific set of assignments in a CS2 course at a single university, with limited test cases partially due to profiling time-limitations. Some assignments included optimization hints, which may have reduced the frequency of related bugs. Some assignments advised students to avoid premature optimization and implement a brute-force solution first, so some of the submissions examined may have been intentional brute-force solutions that do not reflect student misconceptions. We excluded some submissions with runtime efficiency bugs from consideration because they experienced other failures on the grading server (e.g., running out of memory due to excessive print statements or large data structures rather than timing out).

Another threat to validity is labeling error, since identifying the cause and the severity of efficiency bugs can be challenging. There may also be labeling uncertainty because of possible overlap between categories, such as between Failing to Cache and Repeatedly Solving Subproblems. We mitigated these threats by using a performance profiler, by test-fixing a bug when in doubt, and by having multiple authors collaborate on a large fraction of the sample. However, the profiler ran with a fixed time limit and did not always measure all lines of extremely slow code. We also sometimes labeled efficiency bugs according to our course objectives of what students should be able to achieve given the provided course materials and assignment description, not by what the most optimal solution is. For example, the Minesweeper game state check can be optimized by caching the number of remaining squares, but we expected students to loop through the positions on the board and short circuit as soon as they found a non-mine square that was still covered.

5 Conclusion

Our investigation into the kinds of efficiency bugs that students make yielded a novel efficiency bug taxonomy in Table 1 as well as related bug examples. Table 2 shows that while most subcategories were present across multiple assignments, the frequency of each subcategory tended to vary by assignment. This suggests that careful assignment design may lead students to encounter specific efficiency bugs.

There are many avenues for further work that may improve course design and student learning outcomes. For example, quantifying the impact of different kinds of efficiency bugs could help determine which ones are most pedagogically relevant. Also, looking at how long various bugs take students to fix, which bugs students fail to fix the most, or comparing the efficiency bugs in slow submissions to median and fast submissions may provide insight into which efficiency bugs are most persistent or lowest impact.

References

[1] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 522–527. https://doi.org/10.1145/2676723.2677258

[2] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. "I know it when I see it" Perceptions of Code Quality: ITiCSE '17 Working Group Report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* (Bologna, Italy) *(ITiCSE-WGR '17)*. Association for Computing Machinery, New York, NY, USA, 70–85. https://doi.org/10.1145/3174781.3174785

[3] Hope Dargan, Adam Gilbert-Diamond, Adam J. Hartz, and Robert C. Miller. 2024. Efficiency Reading. https://up.csail.mit.edu/vprof/efficiency-reading.html

[4] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3160489.3160500

[5] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull.* 35, 1 (jan 2003), 153–156. https://doi.org/10.1145/792548.611956

[6] Cruz Izu and Claudio Mirolo. 2023. Exploring CS1 Student's Notions of Code Quality. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 12–18. https://doi.org/10.1145/3587102.3588808

[7] Robert Kern. 2024. line_profiler: Line-by-line profiling for Python. https://github.com/pyutils/line_profiler

[8] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 110–115. https://doi.org/10.1145/3059009.3059061

[9] Mokotsolane Ben Mase and Liezel Nel. 2022. Common Code Writing Errors Made by Novice Programmers: Implications for the Teaching of Introductory Programming. In *ICT Education*, Wai Sze Leung, Marijke Coetzee, Duncan Coulter, and Deon Cotterrell (Eds.). Springer International Publishing, Cham, 102–117. https://doi.org/10.1007/978-3-030-95003-3_7

[10] Davin McCall and Michael Kölling. 2019. A New Look at Novice Programmer Errors. *ACM Trans. Comput. Educ.* 19, 4, Article 38 (jul 2019), 30 pages. https://doi.org/10.1145/3335814

[11] Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 1–8. https://doi.org/10.1109/FIE.2014.7044420

[12] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An Empirical Study of Iterative Improvement in Programming Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 410–415. https://doi.org/10.1145/2676723.2677279

[13] Riccardo Pinciroli, Connie U. Smith, and Catia Trubiani. 2024. Modeling more software performance antipatterns in cyber-physical systems. *Software and systems modeling* 23, 4 (2024), 1003–1023. https://doi.org/10.1007/s10270-023-01137-x Place: Berlin/Heidelberg Publisher: Springer Berlin Heidelberg.

[14] K. Andrew R. Richards and Michael A. Hemphill. 2018. A Practical Guide to Collaborative Qualitative Data Analysis. *Journal of Teaching in Physical Education* 37, 2 (April 2018), 225–231. https://doi.org/10.1123/jtpe.2017-0084

[15] Connie U. Smith and Lloyd G. Williams. 2000. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance (WOSP '00)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/350391.350420

[16] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2014. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '14)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/2674683.2674702

[17] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 160–164. https://doi.org/10.1145/2999541.2999555